Contents

nDiskFS: Design and Implementation Guide

Nuka Siva Kumar
Indian Institute of Technology Kanpur

1 Introduction

nDiskFS is a Linux file system which is under development by the CDOS group in CSE@IIT Kanpur. This file system started as a course project by Mr. Pallav Agarwal (BTech-2014) in 2018. Vignesh Sunder (MTech 2017) made some major contributions to the file system during his Masters thesis. We are creating this document to provide an up-close view of the file system by explaining the detailed design and implementation internals of the file system. In this document, we explain the process of file system creation and its on-disk organization. Next, we discuss design and implementation of different VFS layer interfaces like super blocks, inodes, dentries, superblock operations, file operations and inode operations. We also describe different functionalities like caching, eviction, dirty sync operations as part of this document.

1.1 Getting started with nDiskFS source tree

Source code tree

```
nDiskFS
|------------- help
|               |------------- mkfs.c
|               |------------- nonrot.c
|               |------------- get-struct.sh
|               |------------- Makefile
|------------- src
|               |------------- addr.c
|               |------------- avl.c
|               |------------- block_cache.c
|               |------------- blockIO.c
|               |------------- defs.c
|               |------------- diff_file.c
|               |------------- dir.c
|               |------------- file.c
|               |------------- fs.c
|               |------------- fs-controller.c
|               |------------- fs-stat.c
|               |------------- glru_policy.c
|               |------------- gpattern_policy.c
|               |------------- history_of_blocks.c
|               |------------- inode.c
|               |------------- inode_iterator.c
|               |------------- lru.c
|                |------------- ndfs-kernel.patch
|                |------------- page_cache.c
|                |------------- priority.c
|                |------------- resource_ctrl.c
|                |------------- sb.c
|                |------------- include
|                |              |------------- addr.h
|                |              |------------- avl.h
|                |              |------------- block_cache.h
|                |              |------------- blockIO.h
|                |              |------------- common.h
|                |              |------------- defs.h
|                |              |------------- diff_file.h
|                |              |------------- dir.h
|                |              |------------- dm-table.h
|                |              |------------- file.h
|                |              |------------- fs-controller.h
|                |              |------------- glru_policy.h
|                |              |------------- gpattern_policy.h
|                |              |------------- history_of_blocks.h
|                |              |------------- inode.h
|                |              |------------- inode_iterator.h
|                |              |------------- lru.h
|                |              |------------- mkfs.h
|                |              |------------- page_cache.h
|                |              |------------- priority.h
|                |              |------------- resource_ctrl.h
|                |              |------------- sb.h
|------------- Makefile
|------------- tests
|               |------------- filebench
|               |------------- mmap
|               |------------- pattern_access_test
|               |------------- sqlite
|               |------------- postmark
|               |------------- start.sh
|               |------------- tests.sh
```

Above shows the nDiskFS source code tree. `src` is the source code directory. `help` directory contains `mkfs` user program and `nonrot.c` kernel module that helps the device to be non-rotational disk(mimics the SSD). `tests` contains available test suites in the nDiskFS. Makefile builds nDiskFS source code and creates `build` directory in nDiskFS.

Process of building nDiskFS

First build the nDiskFS source directory using `Makefile`, creates `ndiskfs.ko` and insert the `ndiskfs.ko` module into kernel. Setting up the disks `/dev/vdb /dev/vdc` as shown in subsection 1.2 and creates `vg0` volume group to use both HDD and SDD. In the absence of SSD, inserts the `nonrot.ko` module into kernel with disk device so that disk mimics behavior of SSD. After creating the volume group `vg0`, create file system by using `mkfs` program in the `vg0` as described in subsection 1.3. After creating the filesystem, ndiskfs can be mount to any directory and can run any benchmarks.

All this above process performs through the nDiskFS scripts `/nDiskfs/tests/test.sh`

/nDiskfs/tests/start.sh. We can add/comment/uncomment the default benchmarks running in the /nDiskfs/tests/start.sh file.

```
go to the nDiskFS source directory
 $ cd /nDiskFS

to build the nDiskFS
 ~/nDiskFS$ make

to build the nDiskFS, create the filesystem and mount it to /mnt/ndfs
 ~/nDiskFS$ ./tests/test.sh -nc -nt

along with nDiskFS creation to run default benchmarks suite
 ~/nDiskFS$ ./tests/test.sh -nc

to unmount the filesystem, clears the volume group
 ~/nDiksFS$ ./tests/test.sh -c
```

Before going to run make sure linux kernel 4.19.13 is patched with the patch file in the /nDiskFS/src/ndfs-kernel.patch. /nDiskFS/src/include/dm-table.h is the file that contains struct dm-table which should not be empty for creation of the nDiskFS mkfs. Usually, /nDiskFS/src/include/dm-table.h file filled by /nDiskFS/help/get-struct.sh script.

1.2 Creating partitions, logical volume manager(lvm)

Hard disks and SSDs store information in a persistent manner. Storage devices expose low-level APIs to store and access data at a sector/block granularity. File system software, an important subsystem of the OS, abstracts the low-level storage devices to expose a user-friendly view of the storage in the form of a file system tree. Generally, the storage devices are first partitioned into logical block devices and, on each logical partition, a file system is created (also commonly known as the process of formatting the partition). We provide a detailed description of steps involved in creating the nDiskFS file system.

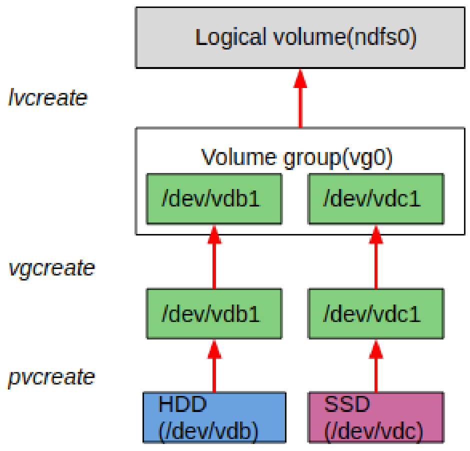


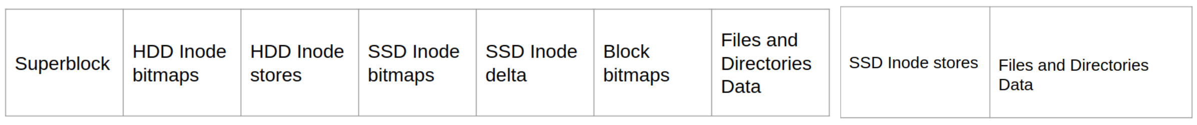
Figure 1: Partions and logical volumes

Preparing the hybrid partition: As nDiskFS is a filesystem for hybrid storage, it uses different types of disks like HDDs, SSDs etc. Logical volume manager utility (`LVM`) facilitates the filesystem to use multiple disks at the same time. `LVM` creates a block layer abstraction on top of logical partitions and exposes a single logical volume to the file system layer. The filesystem operates on the logical volume just like any physical parition. `LVM` is a powerful tool can provides a logical volume which can be—a part of a physical volume, a full physical volume itself or consting of multiple physical volumes. Figure 1 shows the creation of logical volume process in the nDiskFS setup.

nDiskFS uses the logical volume consisting of multiple logical partitions like `/dev/vdb` `/dev/vdc`. These multiple logical partitions is part of a the volume group `vg0`. This volume group `vg0` formed by the logical partitions of `/dev/vdb /dev/vdc` . First format the physical volumes with lvm usable physical volumes(`/dev/vdb1, /dev/vdc1`) by using *pvcreate* as shown in Figure 1. These lvm usable physical volumes(`/dev/vdb1, /dev/vdc1`) grouped into a volume group(`vg0`) by using *vgcreate* as shown in Figure 1. Finally this volume group(`vg0`) used as a logical volume(`ndfs0`) to the filesystem by using *lvcreate* as shown in Figure 1. `mkfs` program creates the filesystem in logical volume `ndfs0`. For the detailed setup process refer `/nDisFS/tests/start.sh` file.

1.3 Filesystem intialization

nDiskFS combines the HDD and SSD partitions into a logical volume group in such a manner that all the HDD partitions are at the beginning followed by SSD partitions. nDiskFS addresses the logical volume using block numbers (block size = 4KB) starting from zero. The starting block number of the SSD partitions is derived by querying the Linux kernel block layer using the `LVM` APIs.



(a) HDD (b) SSD

Figure 2: nDiskFS disk layout

nDiskFS uses logical volume to initialize the filesystem by calling the `mkfs` program. `mkfs` user program of the nDiskFS (`/nDiskFS/help/mkfs.c`) creates the filesystem. It initializes the filesystem layout according to the disk layout as shown in Figure 2.

Superblock: Superblock contains several information required for the working of the file system like locations (block addresses) of block bitmap, inode bitmap, HDD inodes, SSD cached inodes, and some persistent statistics regarding the file system usage history. The superblock stays in the memory as long as the file system is mounted and is synchronized whenever its contents are updated.

Superblock resides in the first block of the logical volume. The superblock is created during the file system creation (using `mkfs`) and read from the disk when the file system is mounted (using the `mount` utility). The on-disk super block is used to create an in-memory super block structure which provides an entry point to the entire filesystem. The in-memory superblock is a superset of the on-disk structure containing some additional information and VFS callback implementations for nDiskFS. We explain the in-memory superblock structure in Section 1.4.

The on-disk super block structure (defined in `/nDiskFS/src/include/mkfs.h`) is shown below. The `mkfs` program fills the on-disk superblock structure and writes it to the first block of the logical volume. All these on-disk superblock structure members are calculated while running `mkfs` program in `/nDiskFS/help/mkfs.c` This on-disk superblock structure written to the disk is shown below.

```
struct ndfs_super_block{
    uint64_t version;
    uint64_t magic;
    uint64_t block_size;
    uint64_t inodes_count;
    uint64_t inodestore_count;
    uint64_t ssd_inodes_count;
    uint64_t ssd_inodestore_count;
    uint64_t ssd_inodes_mapping_count;
    uint64_t free_map_count;
    uint64_t inode_bitmaps_count;
    uint64_t ssd_inode_bitmaps_count;
    uint64_t possible_next_free_inode;
    uint64_t possible_next_free_inode_ssd;
    uint64_t stats[num_stored_stats];
}
```

HDD inode bitmap: HDD inode bitmap contains two bits for each inode to represent the status of the corresponding inode. Table 1 shows the interpretation of the bit values

Inode status bits (binary)	Interpretation in nDiskFS
00	Inode is valid in HDD and outdated in SSD
01	Inode is valid only in HDD
10	Inode is valid in HDD and is up to date in SSD
11	Inode is invalid (i.e., free)

Table 1: Inode status bits and their meaning.

to determine the status of the inode. In nDiskFS, all used inodes are always up to date in the HDD. Some frequently used inodes are cached in SSD which becomes stale if the inode is updated (as nDiskFS avoids writing to the SSD). Inode bitmap is present in the memory for the lifetime of the file system and is synchronized whenever updated. At the time of file system creation, only the inode bitmap corresponding to the root inode is valid while bit values for all other inodes is 11.

HDD inode store: HDD inode store organizes all the inodes in the order of inode number. The inodes are directly indexed using the inode number. Inode stores are the HDD inode blocks. Root inode has written to the starting block of the HDD inode store. The total number of inodes in the file system can be calculated using the folowing expression,

$$TotalInodes = \frac{TotalBlocks}{nDiskFSInodeRatio}$$

The default value of nDiskFS_Inode_Ratio is 64 and it is calculated while running `mkfs` program in `/nDiskFS/help/mkfs.c`

SSD inode bitmap: SSD inode bitmap have single bit for each inode position in SSD inode store representing the availability of that position. SSD inode bitmap present in the memory for the lifetime of the file system and synchronized whenever updated. For every SSD inode a single bit is used to represent respective SSD inode is present or not.

SSD inode bitmap stored in HDD. SSD inodes are stored from starting block of SSD. SSD Inode in SSD, status can be derived from SSD inode bitmap in HDD.

Number of SSD inodes are calculated through

$$\#SSD\_Inodes = \frac{\#Total\_Inodes}{NdfsInodeSSDRatio}$$

The default value of NDFS_INODE_SSD_RATIO is 10.

SSD inode delta: The delta of SSD cached inodes that are modified after caching them are stored in SSD inode delta blocks. The delta also contains the inode location in SSD. The delta present in the memory for the lifetime of the file system and gets synchronized only in the end. In these blocks it stores the delta's of the SSD inode updations. Number of SSD inode delta blocks is calculated using the following expression,

$$\#InodeDeltaBlocks = \#SSDInodes * MaxDeltaSize$$

where $MaxDeltaSize$ is calculated as,

$$MaxDeltaSize = \frac{sizeof(InodeMainMembers)}{2} + 3 * 8$$

$InodeMainMembers$ is a structure named `struct ndfs_inode_main_members` defined in `/nDiskFS/src/include/mkfs.h` and it is member in `ndfs_inode_info` which is described in Section 2.

Block bitmap: The file system maintains one bit for every block to represent the status of the block (used or unused). At the time of file system creation, the `mkfs` program writes a value of one to the block bitmap corresponding to—the superblock, the first block of the inode store (root inode), the first block of the HDD inode bitmap and the first block the data blocks (root inode data).

Datablocks: Datablocks contains file data, directory data, indirect blocks and diff blocks. In the first datablock, root inode dirent (discussed in Section 4) is written at the time of nDiskFS creation.

SSD inode store: SSD inodes are stored from the starting block of SSD and starting block of the SSD known when we mount the nDiskFS. The HDD inodes that are highly read are migrated to SSD inode stores. More details explained in Section 2.

SSD data blocks: Contains file data, directory data, indirect blocks that are frequently read and not written often. These blocks are exclusive to SSD in the stable state of the file system. As mentioned earlier, the starting block of the SSD known when we mount the file system.

1.4 In memory superblock

In memory representation of the superblock is `ndfs_private` structure represents the in-memory structure of the superblock. While mounting, nDiskFS reads the on-disk super block(`ndfs_super_block`) and creates an in-memory superblock(`ndfs_private`). In-memory superblock is represented using `struct ndfs_private` defined in `/nDiskFS/src/include/commo`
This in-memory superblock structure is filled and attached to the VFS superblock. VFS superblock contains a private to pointer to `struct ndfs_private` structure and accessed using `sb->sfs_info`.

```
struct ndfs_private {
    int ndfs_layer_count;
    struct ndfs_dev_pair *ndfs_ladder;
    struct buffer_head **ndfs_free_blocks;
    struct buffer_head **ndfs_free_inodes;
    struct buffer_head **ndfs_ssd_free_inodes;
    struct buffer_head *block0;
    lock_t inodes_lock; //bitmap
    lock_t ssd_inodes_lock; //bitmap
    lock_t block_bitmap_lock;
    avl ndfs_inodes_in_cache;
    avl ndfs_inodes_in_ssd;
    uint64_t ssd_start_block;
    void *block_cache;
    void *page_caches;
    struct mutex stats_lock;
    int tmp_var;
    int umount_began;
    int *block_bitmaps_dirty_status;
}
```

`ndfs_free_blocks`: ndfs_free_blocks is array of buffer_heads and always in memory and represents block bitmaps. *block_bitmap_lock* protects these bitmaps.

`ndfs_free_inodes`: ndfs_free_inodes is array of buffer_heads and always stas in memory and represent inode bitmap. *inodes_lock* protects these bitmaps.

`ndfs_ssd_free_inodes`: ndfs_ssd_free_inodes is array of buffer_heads and always stays in memory and represent ssd inode bitmap. *ssd_inodes_lock* protects these bitmaps.

`block0`: block0 points to the buffer_head of on-disk superblock(first block of the filesystem).

`ndfs_inodes_in_cache, ndfs_inodes_in_ssd`: These are the avl trees that caches the inodes and SSD delta's. More information in the section 2

`block_cache, page_caches`: block_cache caches the inode blocks and page_caches caches the file,diretory data blocks. More information in the Section sec:caching.

`ssd_start_block`: ssd_start_block is filled when mount being called. It represents starting block of the SSD.

`stats_lock`: stats_lock locks the members of the stat array. These are statistics in the superblock.

`block_bitmaps_dirty_status`: block_bitmaps_dirty_status is array of integers represent dirty status of the block bitmaps. Every integer represent to one block of the bitmap.

1.5 Mounting the nDiskFS

As mentioned earlier, the `mkfs` program creates the on-disk superblock along with the file system layout and appropriately initializes the superblock members to points to other structures. The file system mounted only after creating the file system using `mkfs`. nDiskFS is written as a kernel module; at the time of insertion, the nDiskFS module registers the VFS callback function for the mount operation. When super user mounts the nDiskFS file system, the VFS layer invokes the nDiskFS mount implementation. `ndfs_mount` internally calls `mount_bdev` and eventually it invokes `ndfs_fill_superblock`.

The `ndfs_fill_superblock` function reads superblock, inode bitmap, SSD inode bitmap, free block bitmap into the in-memory structures and populates SSD inode mapping . SSD inode mappings are the mapping between HDD inode and updated SSD inode. SSD delta blocks in HDD containing the updations to the SSD inodes. While mounting nDiskFS populates these modifications as deltas from the previous unmounts deltas. Here delta refers to structure `ndfs_inode_delta` in `ndfs_inode_info` is defined in `/nDiskFS/src/include/inode.h`.

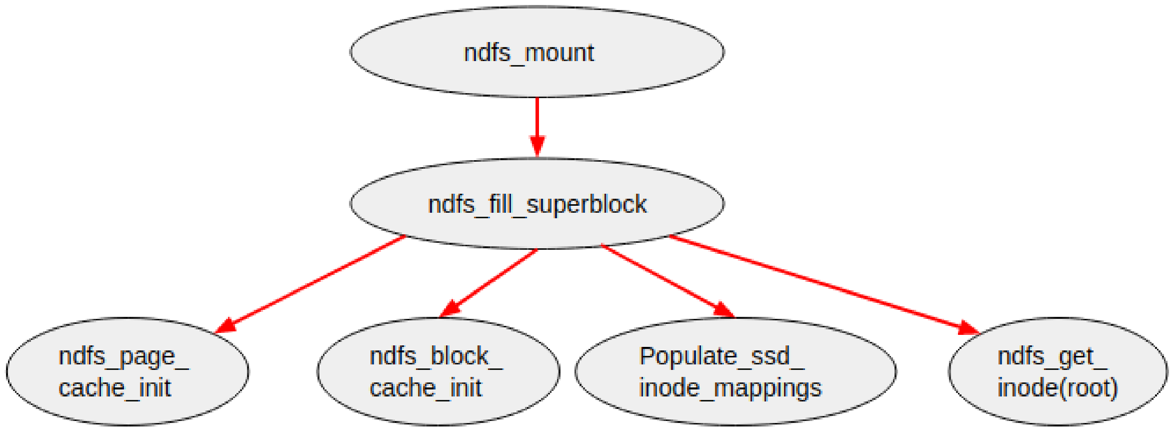


Figure 3: *ndfs_mount() call graph*

- *ndfs_fill_superblock* sets up all in-memory structures in the file system like superblock, inode bitmap, SSD inode bitmap, free block bitmap and this cache is being maintained by Linux kernel because these block pages are read through buffer_head structures.
- This initializes the *inodes_lock, ssd_inodes_lock, block_bitmap_lock* and these locks protects their respective bitmaps.
- Initializes the avl tress of *ndfs_inodes_in_cache, ndfs_inodes_in_ssd. ndfs_inodes_in_cache* caches the inodes and *ndfs_inodes_in_ssd* caches the delta versions of ssd inodes. While mounting it populate previous unmount delta's of the ssd inodes so it calls to *populate_ssd_inode_mappings()* as shown in Fig 3.
- In the mounting it initializes the page_cache, block_caches . More detailed description will be there in Section 3.
- Read root inode from the disk and initialize the directory, file operations on the root inode.
- After mounting file system is ready to use.

code locatoin `/nDiskFS/src/sb.c`

2 Superblock Operations

When mounting the file system, file system registers superblock operations. In nDiskFS during mounting, it initializes the suberblock members, pagecache, blockcache, ssd inode cache and also superblock operations. These operations are invoked from VFS layer.

`ndfs_inode_info`(`/nDiskFS/src/inclue/inode.h`) represents the inode in the file system. The structure of the inode is given below. Some `ndfs_inode_info`'s can be migrated to ssd based on their access patterns. If any updation of the ssd inode is saving to the `ndfs_inode_delta`(`/nDiskFS/src/inclue/inode.h`) structure. `ndfs_inode_delta` is the delta that contains the modifications of the ssd inode .

vfs_inode is a member in the `ndfs_inode_info`. When inode is brought to memory, **vfs_inode** intialized in the nDiskFS.

```
  struct ndfs_inode_info {
     uint64_t inode_num;
    mode_t mode;
    __le16 ni_uid;
    __le16 ni_gid;
    uint64_t file_size;
    union {
        uint32_t children_num;
        uint32_t num_links;
    };
    __le32 ni_blocks_count;
    __le32 ni_atime;
    __le32 ni_mtime;
    __le32 ni_ctime;
    int pattern;
    int migr_status;
    int num_file_reads;
    int num_file_writes;
    uint64_t data_block_nums[15];
    uint64_t diff_id;
    ndfs_inode_delta *delta;
    struct mutex ni_lock;
    struct mutex ni_delta_lock;
    struct page_cache *pc;
    int dropped;
    int ni_status;
    struct inode vfs_inode;
};

struct ndfs_inode_delta{
    __le16 delta_size;
    dirty_bits_t dirty_bits;
    uint64_t inode_num;
    uint64_t ssd_inode;
    delta_t delta_params[0];
};
```

Each inode represents a file or directory in the file system. Every inode present in disk contains necessary information related to its file or directory such as permissions, owner and group ids, file size, data blocks information, number of blocks, accessed, modified and changed timestamps and read and write counts needed for its caching decision. Apart from these, file inodes contain the number of hard links and directory inodes contain the number of children.

When the VFS requests for an inode from nDiskFS, the file system uses the block cache module(Section 3) to read the corresponding inode block from the disk. Each inode block contains multiple inodes, and the required one is copied to memory. The inode in memory has the copy of its disk counterpart along with pointers to page cache, inode delta if applicable and some flags. The inode also encapsulates the corresponding VFS inode structure, which will be primarily used by the VFS layer. The in-memory inodes are cached in a slab cache created by the file system. They are accessed from an AVL tree indexed by the inode number.

nDiskFS replicates frequently used inodes to SSD to speed up reading of inodes from disk. The decision to copy the inode is taken during the eviction based on the number of times the inode was read and written. Once copied to SSD, the corresponding bits in the inode bitmap are marked up-to-date in SSD, and an empty delta is created in memory with the SSD inode position. The next time this inode is requested, the file system will first check the dual bits, find the delta in memory, get the SSD inode position from delta and read it from SSD.

For any outdated SSD inodes, its corresponding delta also stores the modified information. The inode is divided into groups of fixed size, and whenever a field in any group gets changed, the whole group is added to or updated in the delta. When the delta size exceeds a fixed limit, the file system will consider the inode as not suitable for SSD, deletes the SSD version of the inode, its delta from memory and marks the corresponding dual bits as present only in HDD. All the updates to the inode are always synchronized with HDD version irrespective of whether it is cached in SSD or not.

2.1 alloc_inode

In nDiskFS `alloc_inode` is a superblock operation registered as `ndfs_alloc_inode` and called when the VFS layer wants to make room for inode in the file system and expects the file system to returns `vfs_inode` .

```
struct inode *ndfs_alloc_inode(struct super_block *sb)
```

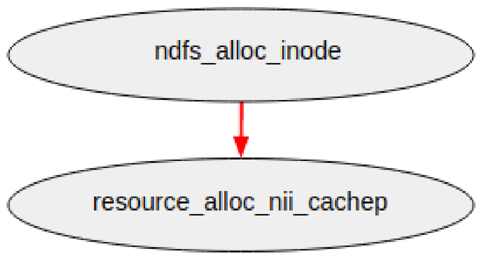


Figure 4: *ndfs_alloc_inode() call graph*

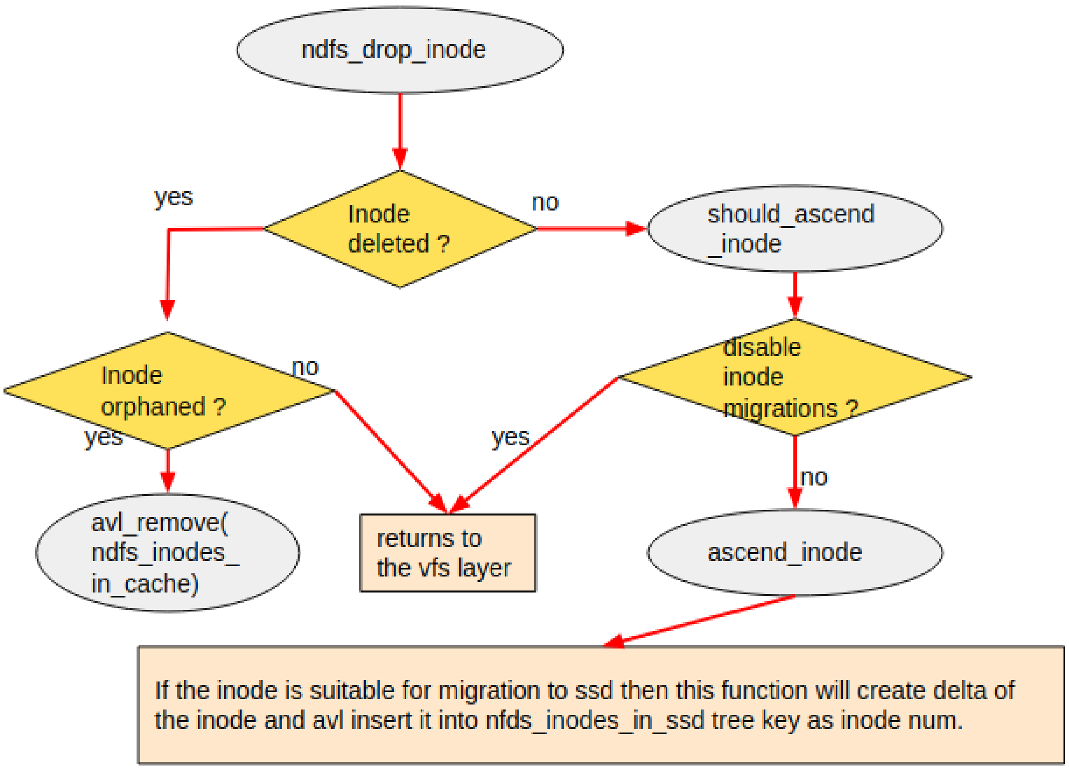
- `ndfs_alloc_inode` calls to `resource_alloc_nii_cachep` to allocate memory for in-memory inode in nDiskFS as shown in Fig 4.
- nDiskFS allocates space for `ndfs_inode_info` which representation of the in memory inode in nDiskFS. `ndfs_inode_info` internally allocates `vfs_inode` .
- `alloc_inode` allocates space, this doesn't assign inode number. Inode number will be assigned while creating a file or directory.

code locatoin `/nDiskFS/src/sb.c`

2.2 drop_inode

In nDiskFS `drop_inode` is superblock operation registered as *ndfs_drop_inode* and calls when the refcount `i_count` is refering to this inode in the VFS layer is decreases and returns 0 to the VFS layer on success. When VFS layer calls this function it expects the underlying file system to evict caches that are attached with this inode. `ndfs_drop_inode` called when `inode_i_count` becomes zero means no VFS layer file object is referencing to this inode. This function expects file system to evict the inode from `ndfs_inodes_in_cache` if it this not required anymore, otherwise it silently returns to the VFS layer. This function doesn't expect the inode's pagecache to be evict. In this function there is no compulsory eviction.

```
int ndfs_drop_inode(struct inode *inode)
```



ndfs_drop_inode
Inode deleted ?
yes
no
should_ascend _inode
Inode orphaned ?
no
yes
disable inode migrations ?
yes
no
avl_remove(ndfs_inodes_ in_cache)
returns to the vfs layer
ascend_inode
If the inode is suitable for migration to ssd then this function will create delta of the inode and avl insert it into nfds_inodes_in_ssd tree key as inode num.

Figure 5: *ndfs_drop_inode() call graph*

- `ndfs_drop_inode` first it checks `INODE_DELETED` flag is set or not. This flag is set when removing the inode(*ndfs_rm_inode*). while removing the inode it doesn't not evict from *ndfs_inodes_in_cache* because other file objects may point to this inode.
- If `INODE_DELETED` flag is sets it evicts this inode from *ndfs_inodes_in_cache* tree if `INODE_ORPHANED` was also set on that instance and decisions are shown in Fig 5.
- `INODE_ORPHANED` flag was set by the dentry operation `d_iput`. This function is called when the dentry points the inode was the last reference. when lookup `on_inode_lookup` operation of another dentry operation called on this inode this flag was unset.
- If `INODE_DELETED` unset it calls `should_ascend_inode` function to migrate this inode to the ssd.
- `should_ascend_inode` function checks the `DISABLE_INODE_MIGRATIONS` to migrate inode to the ssd if this flag unset. To migrate the inode to ssd it uses this below

equation

$$\frac{nii \rightarrow inode\_disk\_read\_count}{10} > nii \rightarrow inode\_disk\_write\_count$$

- `ascend_inode` function creates delta if delta is not created before on this inode then inserting into `ndfs_inodes_in_ssd` avl tree key as `inode_num` . This allocates a ssd inode and read the corresponding block from block cache write inode to that location.

code location `/nDiskFS/src/sb.c`

2.3 evict_inode

In nDiskFS `evict_inode` is a superblock operation registered as `ndfs_evict_inode`. This function called when inode refcount and number of links to the inode both becomes zero. VFS layer expects that this function to evicts all caches maintained by the file system and kernel both. In nDiskFS this function truncates pagecache, blockcache of the inode and returns void.

```
void ndfs_evict_inode(struct inode *inode)
```

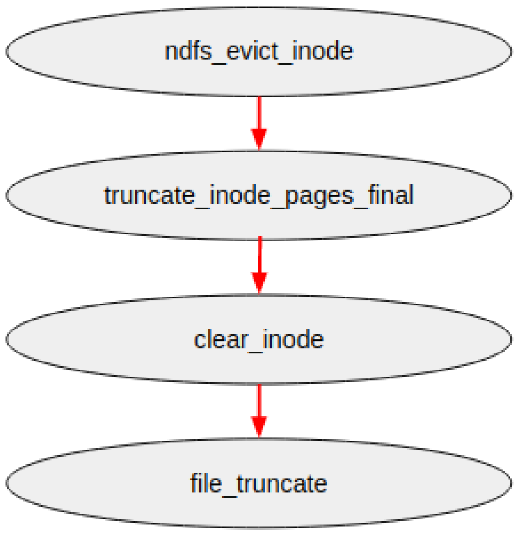


Figure 6: *ndfs_evict_inode() call graph*

- `ndfs_evict_inode` function expects the file system to evict every caching structure related to the inode and eventually it calls to `destroy_inode`.
- To evict kernel level caching structures it calls `truncate_inode_pages_final` and `clear_inode`.
- `truncate_inode_pages_final` function truncates the inode mapping which represents the pagecache maintained by the kernel.
- `clear_inode` cleans the buffer head corresponds to this inode.
- `file_truncate` called to evict the blockcache, pagecache structures maintained by the file system as shown in Fig 6. This truncates the whole file and destroys the pages used by this inode.

code location `/nDiskFS/src/sb.c`

2.4 destroy_inode

In nDiskFS `destroy_inode` is superblock operation registered as `ndfs_destroy_inode`. It is called when VFS layer wants to destroy this inode essentially memory allocated to this inode will be freed. VFS layer makes sure that before calling to the `destroy_inode` it frees every caching structure specific to this inode.

```
void ndfs_destroy_inode(struct inode *inode)
```

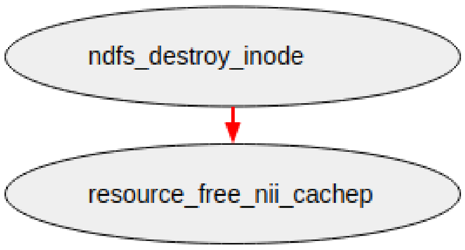


Figure 7: *ndfs_destroy_inode() call graph*

- `ndfs_destroy_inode` called to free the in memory inode representation structure (`ndfs_inode_info`) of nDiskFS. It calls to `resource_free_nii_cachep` to free up the memory as shown in Fig 7.
- After freeing up the inode structure it corrects the file system state by freeing up the inode num from inode bitmap and sync the inode bitmap, superblock.

code location `/nDiskFS/src/sb.c`

2.5 sync_fs

`sync_fs` expects the file system metadata to synchronize with on-disk file system.
code location `/nDiskFS/src/sb.c`

2.6 put_super

`put_super` superblock operation registered as `ndfs_put_super`. `put_super` is called when VFS layer wants to unmount the filesystem.

```
void ndfs_put_super(struct super_block *sb)
```

Algorithm 1: ndfs_put_super Algorithm

```
Input: sb: VFS layer superblock pointer
Result: unmounts the nDiksFS
block_cache_exit();
page_cache_exit();
avl_delete(ndfs_inodes_in_cache);
save_ssd_mappings();
avl_delete(ndfs_inodes_in_ssd) ;
sync_superblock();
sync_blockbitmaps();
free memory for in-memory block bitmap, HDD inode bitmap, SSD inode bitmap
  and superblock ;
```

code location `/nDiskFS/src/sb.c`

2.7 nr_cached_objects

In nDiskFS `nr_cached_objects` is a superblock operation registered as `ndfs_nr_cached_objects`. `nr_cached_objects` expects the file system to return number of clean pages that are ready to evict.

`long ndfs_nr_cached_objects(struct super_block *sb, struct shrink_control *sc)`

- `ndfs_nr_cached_objects` function returns

$$max(CACHE(sb) \rightarrow num\_pc\_blocks - CACHE(sb) \rightarrow num\_pc\_dirty\_blocks, 0)$$

- `num_pc_blocks` is a member of blockcache it represents number pages privately maintained by the file system and `num_pc_dirty_blocks` is also member of blockcache it represents dirty pages in the file system. More about explained in Section 3

code location `/nDiskFS/src/sb.c`

2.8 free_cached_objects

In nDiskFS `free_cached_objects` is a superblock operation registered as a `ndfs_free_cached_objects`. VFS layer expects the file system to vacate nr_pages pointed by shrinker control object.

`long ndfs_free_cached_objects(struct super_block *sb, struct shrink_control *sc)`

- shriker control object points to `nr_to_scan` represent number pages to evict. Then it invokes `evict_blocks` function to request to evict `nr_to_scan` pages. It may not evict exact number of pages.

code location `/nDiskFS/src/sb.c`

3 Caching in nDiskFS

nDiskFS is maintaining its own caching mechanism. nDiskFS is maintaining caching at 3 levels— *(i)* caching that stays always in memory *(ii)* caching the in-memory structures *(iii)* caching through pages.

Caching that stays always in memory performs through the in memory buffer heads that lives till the lifetime of the filesystem (refer Subsection 1.4). In this mechanism, nDiskFS caches the inode bitmap, SSD bitmap, superblock, block bitmap. These structures sync whenever it got dirtied except block bitmap. Block bitmap syncs at the time of unmount or explicitly tell to the nDiskFS through system calls like `sync, direct-io`.

In-memory inodes, in-memory delta's refereed by the SSD inodes are cached through avl trees. These are in memory structures lives in the filesystem till the inode is referred by any application level thread (refer Subsection 3.1). These are representation structures in the disks so changes will be eventually get reflected in the underlying persistent devices or explicitly tell to the nDiskFS though system calls like `sync, direct-io`

Persistent blocks are read and put in a page, serve these pages to the OS abstraction layers. This type of caching is maintained in the kernel. In the sameway, nDiskFS also maintains the it's own block caching and it serves the OS functionalities(refer Subsection 3.2).

3.1 Inode caches

Inode caches caches the in memory structures through the avl trees. These trees intialize when the mount is called. It maintains the inodes that are presently active through the application level threads. These caching structures are representation of the persistent device structures so it's syncs whenever the in memory inode, in memory delta dirtied. HDD always contains up to date inode but SSD may or may not contain up to date inode. nDiskFS always looks into SSD first while searching for the inode. If SSD doesn't contain up to date inode then it looks into delta for in `ndfs_inodes_in_ssd` tree and update the in memory inode while reading.

ndfs_inodes_in_cache

`ndfs_inodes_in_cache` is the avl tree in, in-memory superblock (`ndfs_private`). `ndfs_inodes_in_ca` tree caches the `ndfs_inode_info` members with the `inode_num` as the key .

`avl_tree(ndfs_inodes_in_cache, ndfs_inode_info , inode_num)`

ndfs_inodes_in_ssd

`ndfs_inodes_in_cache` is the avl tree in in-memory superblock (`ndfs_private`). `ndfs_inodes_in_cac` tree caches the `ndfs_inode_delta` members with the `inode_num` as the key.

`avl_tree(ndfs_inodes_in_ssd, ndfs_inode_delta , inode_num)`

3.2 Blockcache

nDiskFS maintains its own caching mechanism. In nDiskFS, blocks are cached in the filesystem itsetlf and maintains it's own mechanisms. In Block cache, nDiskFS caches the inode blocks, delta blocks. These delta blocks contains the information of the delta of the SSD inodes. These blocks read and maintain these blocks in the lru lists for the caching. Blockcache has it's own mechanisms and policy's to maintaining the pages of these blocks.
Abstract view of blockcache: Blockcache caches the blocks containing inodes and delta of the SSD inodes.

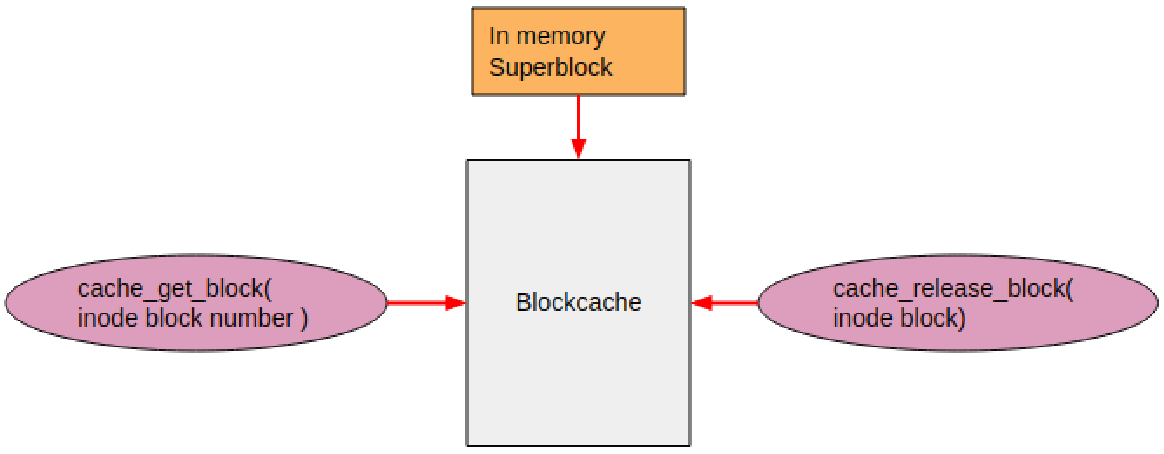

In memory Superblock
Blockcache
cache_get_block(inode block number)
cache_release_block(inode block)

Figure 8: *Blockcache abstract view*

- In a abstract view, blockcache caches the blocks of the inodes and delta blocks. These delta blocks stores information of the modifications to the SSD inodes.
- `cache_get_block( inode block number )` gives the reference to inode block from the blockcache as shown in the Figure 8.
- `cache_release_block(inode block)` releases the reference to the inode block which has taken by the `cache_get_block`.

Implementation of blockcache: While filesystem mounted, in-memory superblock creates the blockcache. Blockcache caches the inode blocks and delta blocks. Blockcache maintains it's own mechanisms and policy's of storing these blocks. These blocks read through the block layer API's to a page. These pages are maintained in a AVL tree with a key as logical block number. Whenever a block is requested, request serves with a reference to the page and release this reference after completion of the request. Blockcache manages the space with evictions, persistence the data with syncing the dirty blocks.

```
struct ndfs_block_cache {
    avl ndfs_cached_blocks;
    struct mutex cache_store_lock;
    struct list_head eviction_list[MAX_DISK_BLOCK_TYPES];
    struct list_head dirty_list[MAX_DISK_BLOCK_TYPES];

    struct mutex evict_lock;
    struct mutex dirty_lock;
    int num_referred_blocks;
    int num_released_blocks;
    int num_dirty_blocks;
    int num_pc_blocks;
    int num_pc_dirty_blocks;
    bool block_cache_stopping;
    atomic_t num_cut_dirty_blocks;
    LRU_SEQ_T last_seq_num;

    struct super_block *sb;

    unsigned num_cached_hdd_blocks;
    unsigned num_cached_ssd_blocks;

    struct task_struct *dirty_sync_thread;
    wait_queue_head_t dirty_sync_wait_queue;
    wait_queue_head_t dirty_blocks_safe_limit;
}

struct cache_block {
    struct page *page;
    lock_t local_cache_lock;
    uint64_t block_num;
    struct list_head list;
    LRU_SEQ_T seq_num;
    int ref_count;
    enum block_content_type content_type;
    int type;
    unsigned short counts[2];
    const struct cache_block_events *events;

};
```

cache_block:

- `cache_block` structure is defined in `nDiskFS/src/include/block_cache.h` file.

- `page` pointer to a page of a block number corresponding to the cache block.
- `local_cache_lock` is a mutex lock which protects the `cache_block`. Any modifications to this cache block or while adding the cache block to any lists this lock has taken.
- `block_bum` member contains the block number of this structure. This block number points to a inode or delta.
- `list` member in a `list_head`. To add cache block to the eviction list, dirty list it uses `list` member.
- `seq_num` represents at which time this cache block has been referred. It is a timestamp to this cache block.
- `ref_count` is reference counter to this cache block. It tells that how many threads are using this cache block.
- `content_type` is a flag to tell that, which type of content being stored in this cache block.
- `type` tells the type of the disk weather it is HDD(0) or SSD(1).
- `cache_block_events` registers a call back functions to this cache block. These registered functions called when creating, evicting the cache block if this functions are registered.

ndfs_cached_blocks is the AVL tree, which stores the `cache_blocks` of the nodes,delta and block number as the key to the node. AVL tree created when initializing in-memory superblock. This tree is refering to a `cache_block` node, so page may or may not be present in this cache block structure.
cache_store_lock is a mutex lock, this lock protects any operation on `ndfs_cached_blocks` AVL tree.
eviction_list[`MAX_DISK_BLOCK_TYPES`] is `list_head` structure represents a array of lists. `MAX_DISK_BLOCK_TYPES` marco represent type of the disk and these are SSD, HDD. These lists contain all the `cache_blocks` which are clean and ready to evict from the blockcache.

eviction_list[0] represents the list of `cache_blocks` in HDD. eviction_list[1] represents the list of `cache_blocks` in SSD.

If the `cache_block` has zero `ref_count` as well as clean block, then this block is in `eviction_list`. When any thread got a reference to the cache block then that cache block removed from the `eviction_list`.
evict_lock is mutex lock and it protects `eviction_list` whenever any modifications being done.
dirty_list[`MAX_DISK_BLOCK_TYPES`] is `list_head` structure represents a array of lists. These lists contain all the `cache_blocks` which are dirtied and require write to the disk.

dirty_list[0] represents the dirty list of `cache_blocks` in HDD. dirty_list[1] represents the dirty list of `cache_blocks` in SSD.

If the `cache_block` had zero `ref_count` and if the block is dirty then the block is added to respective `dirty_list`. When any thread got a reference to the cache block then that block removed from the `dirty_list`.
dirty_lock is mutex lock and it protects `dirty_list` whenever any modifications being done.
num_referred_blocks represents number of `cache_blocks` that are presently active by some thread.

num_released_blocks represents number of pages that are presently not required by any thread. These many blocks are ready to remove from block cache either by eviction or syncing.
num_dirty_blocks represents number of `cache_blocsks` that are dirtied in blockcache.
num_pc_blocks represents number of pages that are in both blockcache and pagecache.
num_pc_dirty_blocks represents the number of dirty pages that are in pagecache.
block_cache_stopping is boolean variable that represents weather the unmount is active or not.
last_seq_num represents the timestamp of the blockcache.
super_block points to VFS layer super_block pointer.
dirty_sync_thread is a kernel thread name as `ndfs_dirty_sync` . This thread in nDiskFS syncs dirty blocks in both blockcache and pagecache. `ndff_dirty_sync` thread created, when blockcache is initialized and runs `dirty_sync_thread_fn()` function. This thread waits at `dirty_sync_wait_queue`. `ndfs_dirty_sync` waits at `dirty_sync_wait_queue` queue and waits till some other process to wakeup. `dirty_sync_wait_queue` interrupts when a new dirty block is added to the filesystem. `dirty_sync_thread_fn()` syncs both pagecache pages and blockcache pages. `num_dirty_blocks` maintains dirty pages in blockcache and `num_pc_dirty_blocks` maintains dirty pages in pagecache. If dirty blocks exceeded `DIRTY_BLOCKS_LIMIT` or at regular timeouts of the thread, `ndfs_dirty_sync` syncs both blockcache , pagecache dirty pages. However, Number of blocks to sync in blockcache and pagecache decided based on over limit of dirty blocks.
dirty_blocks_safe_limit is a waiting queue and sleeps the processes in the nDiskFS untill `TOTAL_DIRTY_BLOCKS(CACHE(sb)) <= DIRTY_BLOCKS_HEAVY_LIMIT` condition mets.
Evictions in blockcache Blockcache evicts clean blocks only if it crosses `CACHED_BLOCKS_LIMIT`. When `cache_get_block` called, This function tries to add a new block to the blockcache. While adding a new block (`cache_add_cb`) if it crosses `CACHED_BLOCKS_LIMIT` limit then `cache_add_cb` evicts one block from the blockcache.

3.3 Pagecache

Pagecache is a subsystem of the nDiskFS which maintains pages in the filesystem. Genarally, caching helps in the performance of the filesystem so nDiskFS pagecache performs that. Linux kernel maintains it's own pagecache but nDiskFS doesn't let the kernel caches the filesystem pages. nDiskFS manages it's own data pages in the pagecache and has it's own mechanisms and policies.

Pagecache initialization happens while in-memory superblock is building on `mount` call. Pagecache caches the data pages and metadata pages of the files and directories. Every inode has it's own cache. These pages maintains in a AVL tree's. Usually when the data page or metadata page is requested by the VFS layer then nDiskFS first caches the page in pagecache after that page has given to the requested operation. Pagecache has data and metadata pages and serves the VFS layer requests with these pages.
Abstract view of Pagecache Genearally VFS layer operations expect the pages to be requested for their operations so these pages are serves the request of the VFS layer operations on demand.

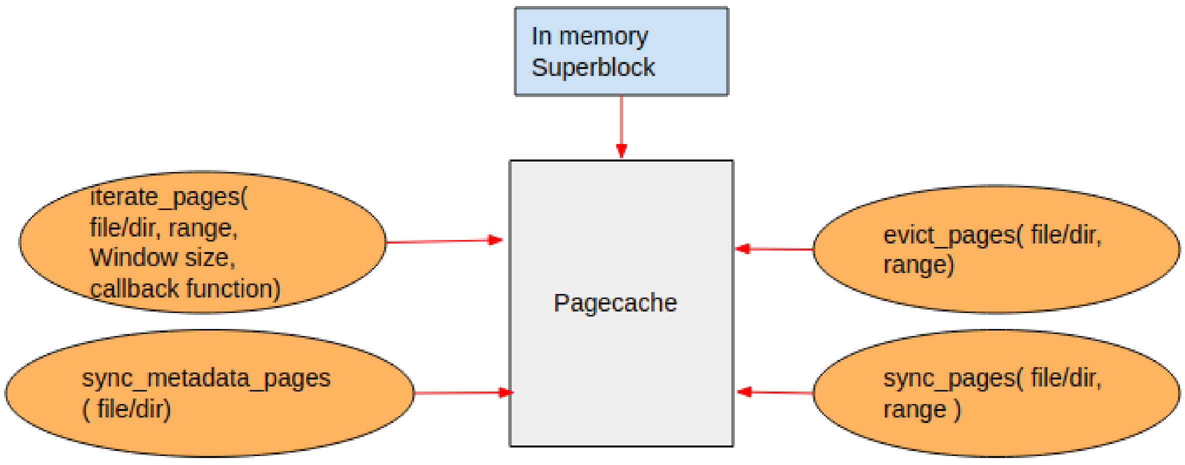

In memory Superblock
Pagecache
iterate_pages(file/dir, range, Window size, callback function)
sync_metadata_pages (file/dir)
evict_pages(file/dir, range)
sync_pages(file/dir, range)

Figure 9: *Pagecache abstract view*

- `iterate_pages` take arguments has file or directory inode , range of the data pages, window size to be iterate and callback function. This API iterate on the the pages in the range and call to callback function for every page.
- `evict_pages` take arguments has file or directory inode, range of the data pages. This API evicts these data pages in the range from the pagecache.
- `sync_pages` take arguments has file or directory inode, range of the data pages. This API syncs the dirty pages in the range from the pagecache.
- `sync_metadata_pages` take arguments has file or directory inode. This API syncs the dirty pages of the inode's metadata blocks like indirect blocks, block bitmap, inode block.

Design details of pagecache nDiskFS pagecache initialization happens when `mount` called. While initializing in-memory superblock `mount` intializes the nDiskFS pagecahe by calling `ndfs_page_cache_init`. `ndfs_page_cache_init` initializes members and structures of the pagecache.

```
struct page_cache_container {
    struct super_block *sb;
    void *system_account;

    struct list_head pcc_lru_list_head[MAX_DISK_BLOCK_TYPES];
    struct list_head pcc_dirt_list_head[MAX_DISK_BLOCK_TYPES];
    struct mutex pcc_lru_list_lock;
    struct mutex pcc_dirt_list_lock;
    struct mutex pcc_sync_lock;
    struct mutex pcc_evict_lock;

    LRU_SEQ_T max_evicted_seq[MAX_DISK_BLOCK_TYPES];

    // for file/dir migrations

    struct list_head pcc_migr_list_head;
    struct mutex pcc_migr_list_lock;
    struct task_struct *file_migrator_thread;
    wait_queue_head_t file_migrator_wait_queue;
    bool page_cache_stopping;
    struct list_head reclaim_blocks_list;   // moved_block_info
    struct mutex reclaim_blocks_list_lock;
};

struct page_cache {
    avl tree;
    avl ind_tree;
    struct mutex p_lock;
    struct debug_info *di;
    ndfs_inode_info *nii;
    void *file_account;

    /* access_pattern policy struct */
    void *access_pattern_policy;

    uint64_t inode_num;
    atomic64_t evictable_cbs[MAX_DISK_BLOCK_TYPES];

    int p_status;

    atomic_t num_iterations;
    atomic_t num_bio_syncs;
    struct list_head pc_lru_node[MAX_DISK_BLOCK_TYPES];
    struct list_head pc_dirt_node[MAX_DISK_BLOCK_TYPES];
    struct list_head pc_lru_list_head[MAX_DISK_BLOCK_TYPES];
    struct list_head pc_dirt_list_head[MAX_DISK_BLOCK_TYPES];
    struct list_head pc_migr_list_node;
    struct mutex pc_lru_list_lock[MAX_DISK_BLOCK_TYPES];
    struct mutex pc_dirt_list_lock[MAX_DISK_BLOCK_TYPES];
    struct rw_semaphore pc_migr_lock;
int pattern;

    int flags;
    unsigned long last_accessed;
};


struct cache_page {
    uint64_t page_num;  // move to cb
    uint64_t block_num;
    struct cache_page_block *cb;
    PAGE_CACHE_T pc;
    struct mutex lock;
    struct debug_info *di;
    struct cache_page_bio *cp_bio;
    struct moved_block_info *moved_info;
    // the following 2 will be used for diff
    uint8_t stored_reads;
    uint8_t stored_writes;
    int flags;
    int disk_type;
};

struct cache_page_block {
    struct page *page;
    CACHE_PAGE_T cp;

    int ref_count;
    int status;
    void *block_account;
    void *eviction_policy;

    void *evict_account;
#define STATUS_REF 1
#define STATUS_DIRTY 2
#define STATUS_DESTROY 4
    const struct pcache_block_events *events;
    void *data;

    struct list_head lru;
    struct list_head sync_list;
};
```

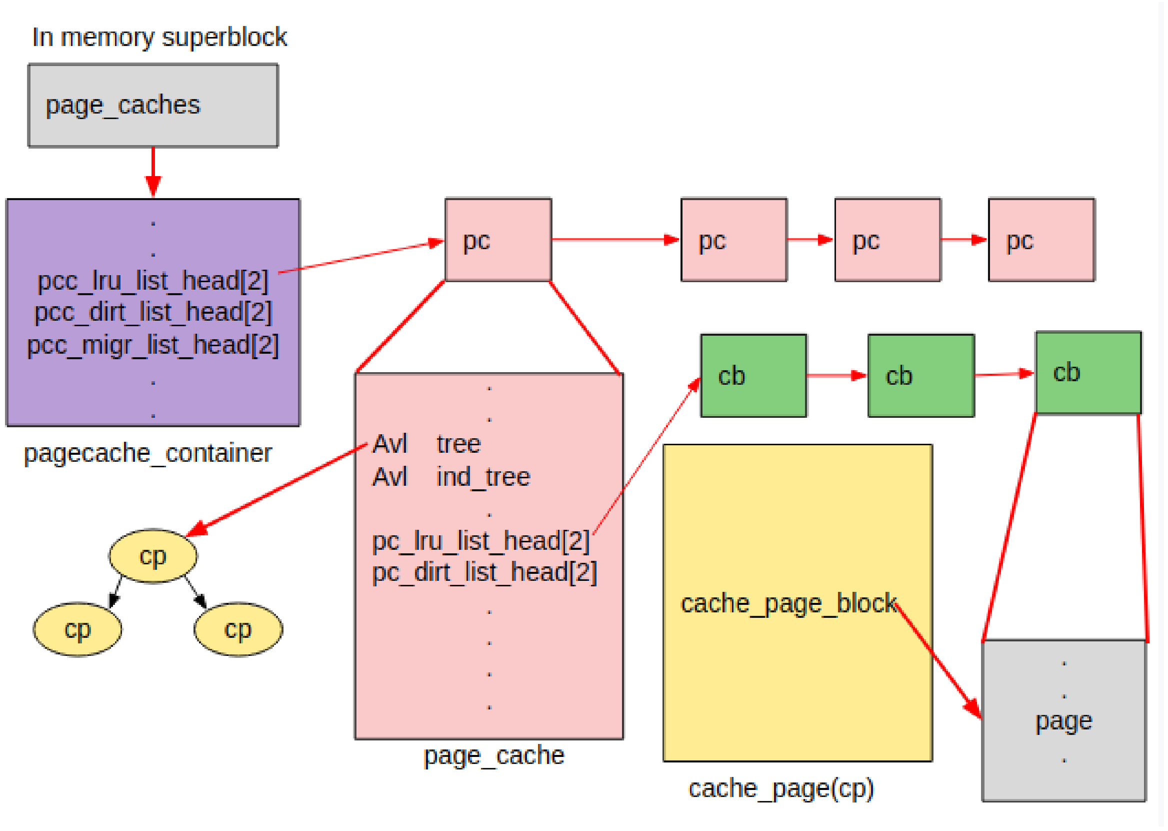


Figure 10: *Pagecache design*

3.3.1 Pagecache structures

page_cache_container

This structure is defined in `/nDiskFS/src/include/page_cache.h` file.

- page_cache_container contains all the page_caches(as shown in Figure 10).
- `sb` points to in-memory superblock.
- `system_account` points to `accounting_system_level` structure. This structure maintains in-memory system level accounting details.
- `pcc_lru_list_head[2]` is list_head structure contains all the pc's(as shown in Figure 10). If the pc is in this structure this represents pc is available to evict it's pages. `pcc_lru_list_lock` protects this list.
- `pcc_dirt_list_head[2]` is list_head structure contains all the pc's who are having dirty pages in their page_cache. While syncing dirty pages, dirty syncer thread get pages from `pcc_dirt_list_head[2]`. `pcc_dirt_list_lock` protects this list.
- `pcc_migr_list_head` is list_head structure contains all the pc's who are ready to migrate to SSD. Migrator thread get page_cache from `pcc_migr_list_head` and migrate it to SSD. `pcc_migr_list_lock` protects this list.
- `max_evicted_seq[2]` contains timestamp of recent eviction of their respective disks.
- `file_migrator_thread` is a kernel thread that migrates files from HDD to SSD. `file_migrator_wait_queue` is the waiting queue for the `file_migrator_thread`.

- `reclaim_blocks_list` list_head of the cache_pages and used in the reverse migrations from SSD to HDD. Currently filesystem disables the reverse migrations. `reclaim_blocks_list_lock` protects the `reclaim_blocks_list` list.

page_cache
This structure is defined in `/nDiskFS/src/include/page_cache.h` file.

- page_cache structure is a member in, in-memory inode . For every inode, pagecache initializes when file wants read or write to a data pages by calling `init_pc_for_inode`. page_cache structure represents like a file in filesystem.
- `avl tree` is member in page_cache. AVL tree caches the cache_page structures(as shown in Figure 10) in the AVL tree and page offset as the key to the AVL tree. This tree only caches the data pages.
- `avl ind_tree` is member in page_cache . This AVL tree caches cache_page structures of the indirect blocks and block number as the key to this AVL tree.
- `p_lock` protects page_cache from accessing multiple threads. More detailed explained in understanding the locks section.
- `debug_info` is structure to the debug `p_lock`.
- `nii` is member in page_cache points to in-memory inode `ndfs_inode_info`.
- `file_account` points to `accounting_file_level` structure. This structure accounts statistics at a file level.
- `access_pattern_policy` structure initializes the access pattern policy to this `page_cache` .
- `inode_num` represents the logical inode number of this `page_cache` inode.
- `evictable_cbs[2]` counts numbers of `cache_page_blocks` avialble to evict for this file in their respective disks.
- `p_status` represents the status of this pagecache. This pagecache may be in iteration, in migration or in idle.
- `num_iterations` counts the number of iteratiors currently performs operations on this `page_cache`(file).
- `num_bio_syncs` represent number of threads that are actively syncing dirty pages on this file.
- `pc_lru_node[2]` is a `list_head` member and this node helps to attach and de-attach to the global lru lists in the page cache container. `pc_lru_node[2]` are the members in the `pcc_lru_list_head[2] list_head`.
- `pc_dirt_node[2]` is also a `list_head` member and helps to attach and de-attach to the global dirty lists in page cache container. `pc_dirt_node[2]` are the members in the `pcc_dirt_list_head[2]`.
- `pc_migr_node` is `list_head` member to a `pcc_migr_list_head` in page cache container. If a file need to be migrated then filesystem add this page_cache to `pcc_migr_list_head` later migration thread takes `page_cache` from `pcc_migr_list_head` and migrate it to SSD.

- `pc_lru_list_head[2]` is a `list_head` member and this is head of the list to maintain cache_page_blocks as shown in Figure 10 and these are ready to evict. `pc_lru_list_locks[2]` are the locks to protect `pc_lru_list_head[2]` lists.

- `pc_dirt_list_head[2]` is a`list_head` member and this head of list to maintain `cache_page_blocks` and these blocks are need to be synced . `pc_dirt_list_locks[2]` are the locks to protect `pc_dirt_list_head[2]` lists.

- `pc_migr_lock` synchronizes the iterators and migrator threads .

- `pattern` represents the pattern of this file. pattern set by access pattern policy.

cache_page
This structure is defined in `/nDiskFS/src/include/page_cache.h` file.

- `cache_page` structure is a container to a page (as shown in Figure 10). `cache_page` initializes when a file wants to read or write to a page. `cache_page` is a node to a AVL trees.

- `pagenum` represents page offset of the file corresponding the block containing in the `cache_page`.

- `blocknum` represents logical block number of this physical block.

- `cache_page_block` is member in `cache_page` and initializes when page is assigned to this `cache_page` .

- `pc` points to the `page_cache` of this node reside in.

- `lock` protects the modifications to this `cache_page` structure.

- `debug_info` structure used to debug the `lock` member.

- `cache_page_bio` helps the nDiskFS to perform read/write to/from the disk by creating block I/O.

- `moved_info` structure uses when this node moves from SSD to HDD. Currently filesystem disables the revers migrations.

- `disk_type` tells the about disk type of this node weather this node block corresponds to SSD or HDD.

cache_page_block
This structure is defined in `/nDiskFS/src/include/page_cache.h` file.

- `cache_page_block` is a member in `cache_page` and initializes when a page is assigned to the `cache_page` structure. `cache_page_block` contains the page.

- `page` is a pointer to a page corresponding the page offset in `cache_page` structure.

- `cp` is pointer to the `cache_page`.

- `ref_count` tells that number of references to this page.

- `status` represents the status of the block. This tells the weather the block is dirtied or destroyed.

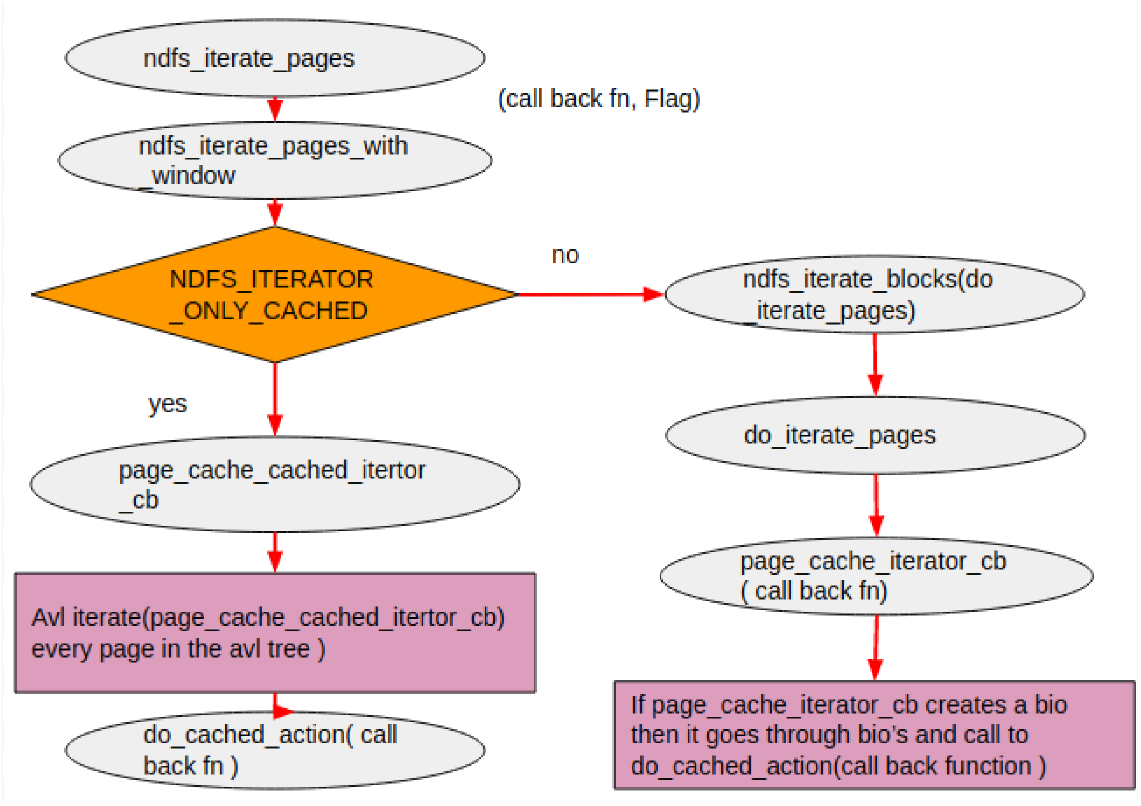
- `block_account` is structure pointer to a `accounting_block_level`. This maintains the statistics at the block level.
- `events` is a structure of event functions and these are called when a event occur on this block. Events are like`on_dirty`, `on_move`, `on_sync`, `on_evict`.
- `lru` is `list_head` member in `pc_lru_list_head[2]`, `pc_dirt_list_head[2]`. These `cache_page_blocks` are added to file level lru lists or dirty lists.
- `sync_list` is also `list_head` member and `cache_page_block` added to a other list_head member when this block is being synced. This member marks that cache block is being synced.

3.3.2 API's of pagecache

In nDiskFS these are threads that run concurrently. Processes that perform reading or writing the file data run iterator thread. File data syncher thread, File metadata syncer thread, File data evictor threads run while a explicit call come from the VFS layer like `direct-IO` or `fsync`. OS forces the filesystem by invoking this Evictor thread to evict pages from the filesystem. Dirty syncer thread is a kernel thread in the filesystem that frequently syncs the dirty pages to the disk. Migration thread also part of the nDiskFS which migrates the files to SSD.

Iterators thread

Processes perform reads/writes the data from/to the disks by using nDiskFS pagecache. Processes gets the pages from the pagecache, this allows the processes concurrently running in the filesystem.



ndfs_iterate_pages
(call back fn, Flag)
ndfs_iterate_pages_with _window
NDFS_ITERATOR _ONLY_CACHED
no
yes
ndfs_iterate_blocks(do _iterate_pages)
do_iterate_pages
page_cache_cached_itertor _cb
page_cache_iterator_cb (call back fn)
Avl iterate(page_cache_cached_itertor_cb) every page in the avl tree)
If page_cache_iterator_cb creates a bio then it goes through bio's and call to do_cached_action(call back function)
do_cached_action(call back fn)

Figure 11: *ndfs iterator call graph*

- `ndfs_iterate_pages` takes arguments as callback function, range of page offsets, inode and iterate through the pages of pagecache and calls to the call back function.

- `ndfs_iterate_pages` calls to `ndfs_iterate_pages_with_window` with a window size. Usually window size for the directory's lookup is 4 and all other iterate with maximum window size.

- `ndfs_iterate_pages_with_window` updates the pc status to iteration, increase number of iterators. This function removes the pc from the global lru lists and dirty lists , migration list because while iterating filesystem doesn't allow anyone to do eviction or migration on this pc.

- VFS layer allows multiple readers come into interator or only one writer comes to iterator.

- If `NDFS_ITERATOR_ONLY_CACHED` sets by the any operation it forces the iterator only searches the already cached pages by calling AVL iterate pages with `page_cache_cached_iterator_` function and calls to the call back function to do action.

- If `NDFS_ITERATOR_ONLY_CACHED` is not set then this function goes through every block and builds `cache_page` and read the data to a page. To iterate on blocks it calls to `ndfs_iteratate_blocks` with `do_iterate_pages` call back function.

- For every block `ndfs_iterate_block` calls to `do_iterate_pages` to build the `cache_page` and put it in AVL tree. `do_iterate_pages` calls to `page_cache_iterator_cb`.

- `page_cache_iterator_cb` looks for page in `cache_page_block`. If page is present then calls to call back function to do action.

- if `page_cache_iterator_cb` doesn't found any page then it constructs the BIO later `ndfs_iterate_pages_with_window` performs the action required.

- `ndfs_iterate_pages_with_window` goes through the BIOs and calls to the call back function to do action.

- After completion of the action, `ndfs_iterate_pages_with_window` performs the accounting and calls for access pattern analyzer and adds the pc to the appropriate global lists.

This function definition is given in `/nDiskFS/src/page_cache.c` file.

File data sync

In the proposed design, nDiskFS page-cache uses a new API i.e., `do_file_sync` method to syncs data pages of a file. This function write a range of dirty data pages of an inode to the disk. This method can be used to implement both `fsync` and direct I/O features.

Algorithm 2: File data syncing Algorithm

```
Input: filePTR: file pointer, startPage, lastPage
Result: sync range of data pages for a inode
pageCache = get_pagecache(filePTR) ;
pageCache→status = iteration ;
pageCache→itr_count ++ ;
pageOffset = startPage;
while pageOffset is in range(startPage, lastPage) do
    locks( fileDirtyLists );
    page = get_page_from_pageCache(pageOffset);
    if page is valid then
        if page is dirty then
            sync_page(page);
            remove_page( fileDirtyList, page ) ;
            if ref_count(page) = 0 then
                add_page( fileLruList, page ) ;
            end
        end
    end
    unlocks( fileDirtyLists );
    pageOffset ++;
end
pageCache→itr_count - - ;
```

`do_file_sync` pagecache API syncs the range of file pages as shown in Algorithm 2. This is an another iterator in the nDiskFS and it sets the page-cache status (maintained for eviction of inode etc.) to "on-iteration" and increments active thread count iterating on this file. This function locks the file LRU dirty lists so that no other iterator can add or remove blocks from the file LRU dirty list till the page is submitted to block layer for performing device write. Next, the function iterates over every dirty page in the range from the AVL tree of pagecache and creates block I/O requests (BIOs) and submits to the block layer to perform syncing. After syncing, the unreferenced clean blocks are added to the file LRU lists and file LRU dirty list is unlocked. Finally it decrements the active iterators count on inode; if the count becomes zero, the method sets the file status to idle.

File metadata sync

nDiskFS pagecache has `do_file_metadata_sync` API to syncs metadata pages of a file. `do_file_metadata_sync` syncs dirty pages of a inode. Usually `fsync`, direct-IO uses this API.

Algorithm 3: File metadata syncing Algorithm

```
Input: sb: superblock, inodePTR: inode pointer
Result: sync metadata pages of a inode
pageCache = get_pagecache(inodePTR) ;
pageCache→status = iteration ;
pageCache→itr_count ++ ;
sync_inode();
sync_block_bitmaps(sb);
while page from the AVL indirect tree do
    lock ( fileDirtyLists ) ;
    if page is valid then
        if page is dirty then
            sync_page(page);
            remove_page( fileDirtyList, page ) ;
            if ref_count(page) = 0 then
                add_page( fileLruList, page ) ;
            end
        end
    end
    unlock( fileDirtyLists );
end
pageCache→itr_count - - ;
```

We created a new nDiskFS page-cache API i.e., `do_file_metadata_sync` to sync file system meta-data like inodes, bitmaps etc. This method can be used to implement both `fsync` and direct I/O features.

`do_file_metadata_sync` is also another iterator which iterates on the file page-cache. Firstly this function sets the pagecache status to "on-iteration" and increments the active iterators count on the file's inode. It syncs the inode, block bitmaps to the block device as shown in Algorithm 3. This function locks the file LRU dirty lists so that no other iterator can add or remove blocks from the file LRU dirty lists, till the page is submitted to block layer to sync. The method collects dirty indirect blocks from the AVL tree, creates BIO requests and submits the BIOs to the block layer to write the indirect blocks to the disk. After syncing, the function adds the unreferenced clean blocks to file LRU lists and finally it unlocks the file LRU dirty lists. Before returning to the upper layer, the function decrements the active iterators count on inode; if the count becomes zero, it sets the file status to idle.

File data eviction nDiskFS page-cache implementation is extended with `do_evict_from_page_cache` API to evict range of clean pages of a file's inode. Usually direct I/O operations invoke this API.

Algorithm 4: File data eviction Algorithm

```
Input: filePTR: file pointer, startPage, lastPage
Result: evict range of data pages from a file
pageCache = get_pagecache(filePTR) ;
pageCache→status = iteration ;
pageCache→itr_count ++ ;
pageOffset = startPage;
while pageOffset is in range(startPage, lastPage) do
    lock( fileLruLists );
    page = get_page_from_pagecache(pageOffset);
    if page is valid then
        if page is !dirty then
            remove_page( fileLruList, page ) ;
            evict_page(page);
        end
    end
    unlocks( fileLruLists ) ;
    pageOffset ++ ;
end
pageCache→itr_count - - ;
```

`do_evict_from_page_cache` pagecache API evicts the range of file pages as shown in Algorithm 4. `do_evict_from_page_cache` function is an iterator which sets the file pagecache status to "on iteration" and increments the active iterators count of the the file inode. The function first acquires a lock on the file LRU lists to prevent other iterator from adding or removing blocks from the file LRU lists while eviction is active. This function gets the range of the pages from the AVL tree and evicts those pages and frees up the internal structures. Finally it unlocks the file LRU lists and decrements the active iterators count on the file inode. nDiskFS pagecache has `do_evict_from_page_cache` API to evict range of pages of a file. `do_evict_from_page_cache` evicts clean pages of a inode. Usually direct-IO uses this API.

3.3.3 Syncing dirty pages

When a large number of dirty blocks are accumulated over time, the file system writes some of them to their corresponding disks. nDiskFS maintains two weighted dirty lists(`pcc_dirty_list_head[2`
one for HDD blocks and the other for SSD blocks and each dirty list contains a list of files (`pc_dirt_node`) in the increasing order of their last written time. A file(pc) can be present in the both the lists if it contains dirty blocks from both the disks. For each file (pc) in the dirty lists, there is a list(`pc_dirt_list_head`) of dirty blocks maintained in the increasing order of their write time. The file system also performs period synchronization of dirty blocks. When the total number of dirty blocks crosses a threshold, or after every timeout, the first file from each of the list is chosen if present and they are evaluated on the basis of their recency and the sync factor of disk type to get the most preferred file to sync. The weights are given such that the blocks from HDD are preferably written to its disk than the blocks of HDD with same recency.

`dirty_sync_thread` is a kernel thread name as "ndfs_dirty_sync". This thread in nDiskFS syncs dirty blocks in both blockcache and pagecache. `ndff_dirty_sync` thread created when blockcache is intialized and runs `dirty_sync_thread_fn()`.

dirty_sync_thread_fn() syncs both pagecache pages and blockcache pages. `num_dirty_blocks` maintains dirty pages in blockcache and `num_pc_dirty_blocks` maintains dirty pages in pagecache. If dirty blocks exceeded `DIRTY_BLOCKS_LIMIT` or at regular timeouts of the thread, `ndfs_dirty_sync` syncs both blockcache , pagecache dirty pages . How many blocks to sync in blockcache and pagecache decided based on over limit of dirty blocks.

`ndfs_dirty_sync` thread calls to `sync_blocks(num pages)` to sync dirty pages in the pagecache. After syncing `sync_blocks dirty_sync_thread_fn` updates number of dirty blocks in pagecache.

3.3.4 Evictions in pagecache

The file system may need to evict some of the cached blocks due to memory pressure in the nDiskFS. It maintains two weighted evict lists(`pcc_lru_list_head[2]`), one for HDD blocks and the other for SSD blocks and each evict list contains a list of files in the increasing order of their last access time. A file can be present in the both the lists if it contains blocks from both the disks. If a file is being iterated or migrated, it won't be present in either of the lists. For each file(`pc_lru_node`) in the evict lists, there is a list of clean blocks(`pc_lru_list_head[2]`) maintained in the increasing order of their last access time. During memory pressure, the first file from each of the list is chosen if present and they are evaluated on the basis of their recency and the eviction factor of disk type to get the most preferred file to evict. The weights are given such that the blocks from SSD are preferably evicted than the blocks of HDD with same recency.

nDiskFS `free_cached_objectes` superblock operation comes with a argument of shrinker control object. shriker control object points to `nr_to_scan` represent number pages to evict. Then it invokes `evict_blocks` function to request to evict `nr_to_scan` pages.

3.3.5 Access pattern Analyzer

Every file and directory will be analyzed its pattern of access after any read or write operation on it. The pattern analysis is done as pre-cursor to data migrations. The analysis will be done only on the condition that the file doesn't have a confirmed pattern yet, is kept idle for a fixed time after its last operation and its inode is not evicted or deleted. The analysis will try to detect abnormal reads or writes on part of the file or if the file is being uniformly read. An access pattern for the file will be confirmed only if the same pattern repeats for three times(current policy). Currently, the supported patterns are evenly read, unevenly read and written and remaining are considered to be unknown patterns.

To enable the functionality of pattern analysis, during every iteration of the file its total block reads, toal block writes, maximum reads and writes for any block are maintained. After every iteration of the file, `do_policy_file_accounting` being and if the file is not accessed until then, its access pattern is detected. This analysis will repeated until an access pattern for the same is confirmed.

`do_policy_file_accounting` calls it's interface function `pattern_ops->file_accessed(pc, flags)`. `pattern_ops` are the interface operations for the underlying access pattern policy and these are registered for every file when `page_cache` has initialized for that file.

4 Inode Operations

When creating the directory object these directory operations are intialized. These directory operations change the inode state and these invoked from the VFS layer to create inode object , to remove inode object, to rename the inode object, to get or set the inode attributes, to unlink the file object.

In this section directory operation means directory inode operation and dirent means `ndfs_dirent` .

Directory's inode data blocks represents data of the that inode. In directory's inode data blocks contains all the children inodes of that directory. Directory's inode data blocks are consists of dirents. Each dirent represents a new inode in the this Directory. Dirent structure is given below.

```
typedef struct ndfs_dirent {
    uint64_t inode_num;
    uint32_t lookup_count_in_hdd;
    uint16_t flags; // unused for now
    uint16_t filename_len;
    char filename[0]; // should start at 8-byte offset
} ndfs_dirent;

This structure is defined in /nDiskFS/src/include/mkfs.h.
```

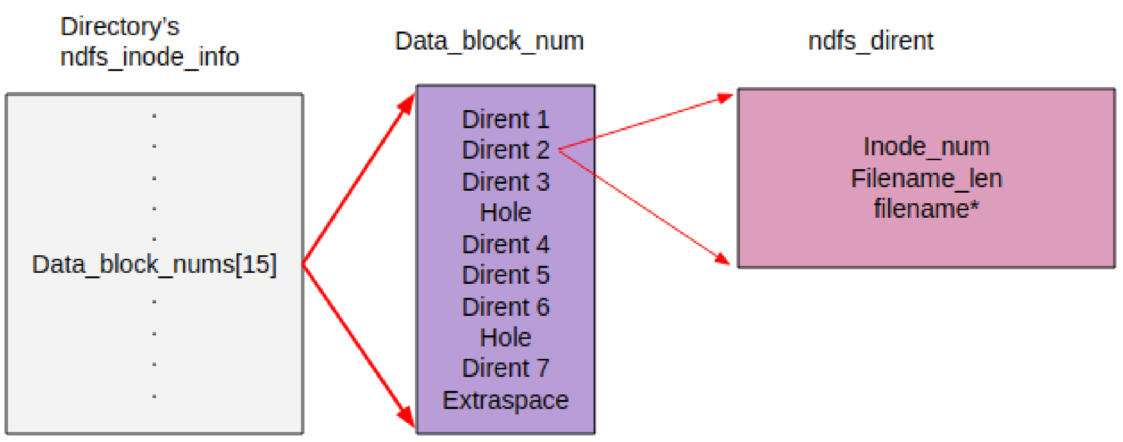


Figure 12: *ndfs_dirent structure*

These directory's indoe data contains variable length dirents. If the dirent is invalid then it's inode number is zero. After deletion of the dirents in these data blocks holes will be formed. While allocating new dirent first it looks into these holes to fit the new dirent.

struct dentry

struct dentry is a VFS layer dentry object that represent's a dirent in the inode data. This dentry points to VFS inode after successfully completion of the `lookup` operation on the parent inode.

4.1 lookup

In nDiskFS `lookup` is a directory operation registered as a `ndfs_lookup` and called to lookup in a directory. Usually, this `looup` operation called before calling other directory operations so that it calls other directory operations with `struct direct` object.

`struct dentry *ndfs_lookup(struct inode *parent, struct dentry *child, unsigned int flags);`

```
  struct ndfs_lookup_state {
     struct super_block *sb;
     struct inode *parent;
```

```
        struct dentry *child;
        uint64_t rem_size;
        uint64_t inode_num_found;
        uint32_t children_num;
};
This structure is defined in /nDiskFS/src/dir.c
```

`ndfs_lookup_state` is a structure to a call back function from `ndf_lookup` to iterate on every page to lookup the child dentry in the parent inode data.

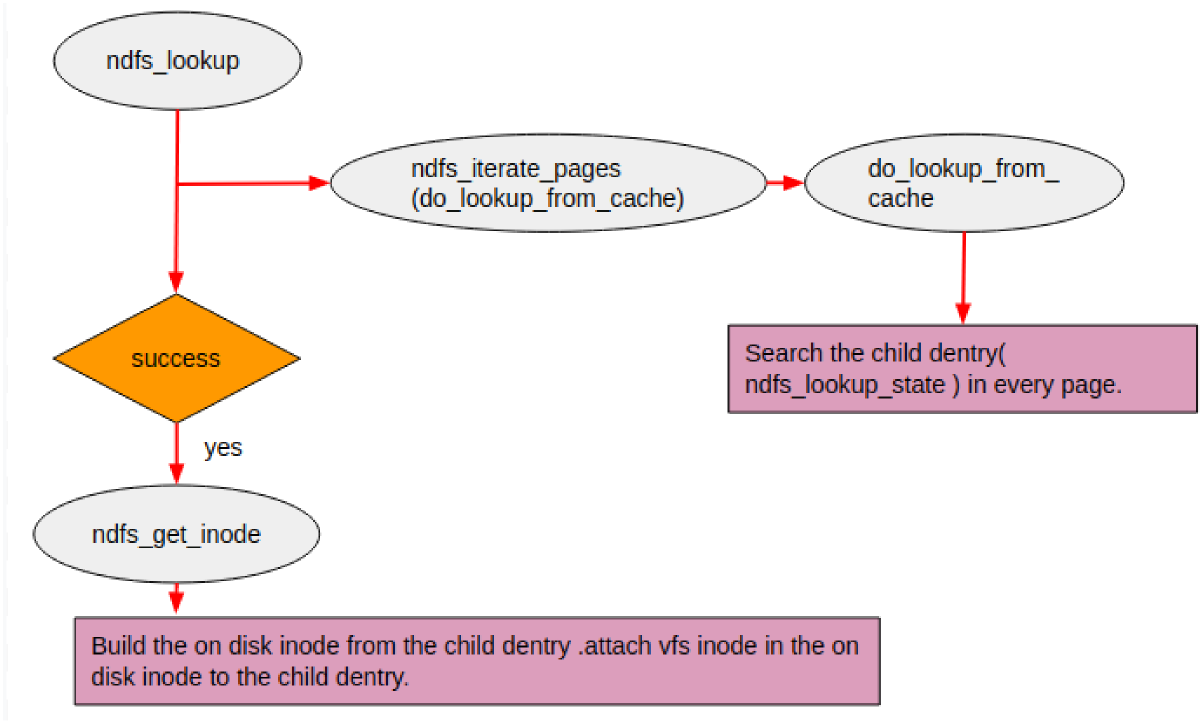

ndfs_lookup
ndfs_iterate_pages (do_lookup_from_cache)
do_lookup_from_ cache
success
yes
ndfs_get_inode
Search the child dentry(ndfs_lookup_state) in every page.
Build the on disk inode from the child dentry .attach vfs inode in the on disk inode to the child dentry.

Figure 13: *ndfs_lookup() call graph*

Algorithm 5: ndfs_lookup algorithm

```
Input: parent: parentInode, child: childInode
Result: if the dentry found it links the inode to that dentry
initialization;
intializing ndfs_lookup_state;
getPageCache() ;
if pageCache_Exists() then
    while pageCache → Page == endPage do
        do_lookup_from_cache() ;
        direct = firstDirent();
        while dirent != endDirectOfpage do
            if dirent→ hole fit ndfs_lookup_state dentry then
                markDentryLocation() ;
            end
            if match( currentDirent, ndfs_lookup_state dentry ) then
                assign ndfs_lookup_state inode_num = currentDirentInode_num;
                return success;
            end
            getNextDirent() ;
        end
        getNextPage(pageCache) ;
    end
else
    while every block in the directory data do
        buildPage( block ) ;
        do_lookup_from_cache();
        getNextBlock();
    end
end
if lookup success then
    build in-memory Inode from blockCache;
    attach vfs_inode(reside in on disk inode)to the child dentry;
else
    return failure;
end
```

- `ndfs_lookup` called to lookup child dentry in the parent inode and usally happens before every directory operation.
- `ndfs_lookup` initializes the `ndfs_lookup_state` call back structure.
- First `ndfs_lookup` searches the child dentry in the pagecache of the parent inode. To look in the parent inode this function calls to `ndfs_iterate_pages` function with the call back function `do_lookup_from_cache`, with the call back structure `ndfs_lookup_state` and with the flag `NDFS_ITERATOR_ONLY_CACHED`.
- For every page in the pagecache, `do_lookup_from_cache` invokes and get the dirents from the page to search.
- `do_lookup_from_cache` searches the child dentry in the dirents of the page if it found dirent inode number is invalid (zero) and child dentry can fit into this dirent hole

then child dentry (`child→d_fsdata`) remembers this page offset because later it may create this dirent in this inode.

- While searching dirents if it founds the child dentry then `ndfs_lookup_state` assigns dentry inode number to the state `inode_num_found` as shown in Algorithm 5.
- If child dentry doesn't found in the pagecache pages then `ndfs_lookup` calls to `ndfs_iterate_pages_with_window` with the `WINDOW_SIZE`.
- `ndfs_iterate_pages_with_window` (detailed description in Sectoin **??**) internally go through all the blocks in the inode and build the pages, put it in pagecache and call back to the `do_lookup_from_cache` function for every page.
- After completion of the iterating through the pages `ndfs_lookup` function checks the `inode_num_found` of `ndfs_lookup_state`. If it founds the lookup it creates `ndfs_inode_info` on disk object and read the inode from block cache by calling `ndfs_get_inode`.
- `ndfs_inode_info` internally allocates memory for `vfs_inode` and fills the `vfs_inode` structure and attach this inode to child dentry so that next time any directory operation `vfs_inode` is already found in the child dentry.

code locatoin `/nDiskFS/src/dir.c`

4.2 create

In nDiskFS `create` is a directory operation registered as a `ndfs_create`. `create` called from VFS layer to create file in the directory. `ndfs_create` allocates a new inode and create a dirent in the parent directory if the file is not exists in the parent directory. `ndfs_create` internally calls `ndfs_create_obj` with the mode.

```
int ndfs_create_obj(struct inode *dir, struct dentry *dentry, umode_t mode);
```

```
  struct ndfs_add_dentry_state {
    struct super_block *sb;
    ndfs_inode_info *nii;
    unsigned block_offset;
    const char *filename;
    int filename_len;
    int hinted_add;
};
This structure define in /nDiskFS/src/dir.c
```

Algorithm 6: ndfs_create_obj algorithm

```
Input: dir: parentInode, dentry: childDentry
Result: create inode in the parent directory
initialization;
allocate a newInode() ;
VFSinode = new_inode();
if obj is Directory then
    register inode operations as ndfs_dir_ops;
    register fops as ndfs_dir_file_ops;
else
    register inode operations as ndfs_reg_ops;
    register fops as ndfs_reg_file_ops;
end
initialize members in the inode;
get the inode number = ndfs_put_inode();
if can allocate new inode number then
    add dentry in the parent inode by calling ndfs_add_in_parent();
    attach vfsinode to the dentry by calling d_add();
    set_dentry_ops();
    avl_insert(ndfs_inodes_in_cache,, inode number);
else
    ndfs_destroy_inode();
end
```

- `ndfs_create` calls to `ndfs_create_obj` to create a file or directory in a inode.
- `ndfs_create_obj` calls to `new_inode` to create a vfs inode. `new_inode` is kernel function, it internally calls to `alloc_inode` superblock operation to create in-memory ndiskfs inode. ndiskfs inode structure contains `struct vfs_inode` structure.
- After allocating memory to the inode it initializes the inode members.
- During initialization if inode to create is a file then it registers `ndfs_reg_ops`, `ndfs_dir_file_ops` if the inode to create is directory then it registers `ndfs_dir_ops`, `ndfs_dir_file_ops`.
- `ndfs_create_obj` calls to `ndfs_put_inode` to get the inode number as shown in Algorithm 6.
- If `ndfs_put_inode` fails to return valid inode number then `ndfs_put_inode` destorys the inode by calling `ndfs_destroy_inode`.

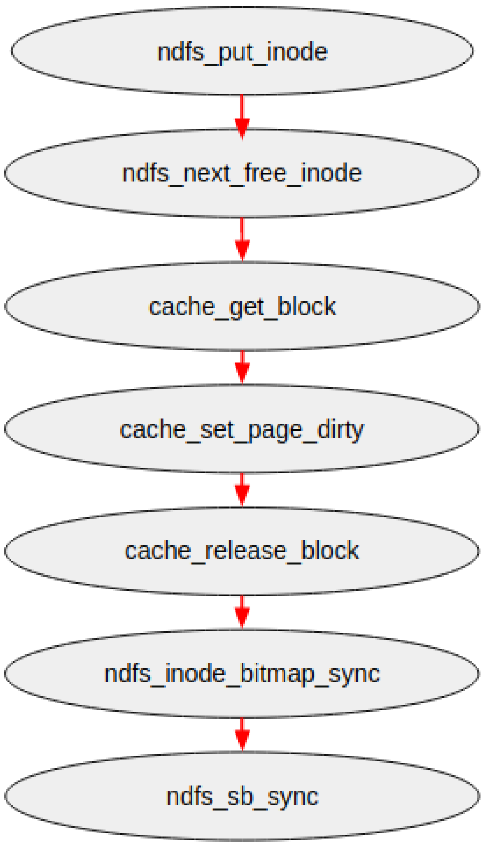


Figure 14: *ndfs_put_inode() call graph*

- `ndfs_put_inode` gets the unique inode number and read the corresponding block from blockcache and put it in page. Write this inode to this page and mark as dirty. sync inode bitmap, and superblock. This call graph has shown in Figure 14.
- `ndfs_next_free_inode` returns the free inode number based on the flag sent to this function. If —tt GET_FREE_INODE_FLAG_FOR_SSD flag sent to this function then `ndfs_next_free_inode` returns SSD free inode number.
- After getting valid inode number `cache_get_block` gets the page from blockcache corresponding to the inode number and write new inode into that page.
- `cache_set_page_dirty` makes page to be dirty because new ndiskfs inode has to be written.
- `cache_release_block` releases to the page referred by this inode number.
- After releasing the page it syncs the inode bitmap and super block by calling `ndfs_inode_bitmap_sync`, `ndfs_sb_sync`

• If `ndfs_put_inode` returns valid inode num then this function call to `ndfs_add_in_parent` to add child dentry to the parent inode and register dentry operations by calling `set_dentry_ops` and attach the VFS inode to the dentry by calling `d_add`.

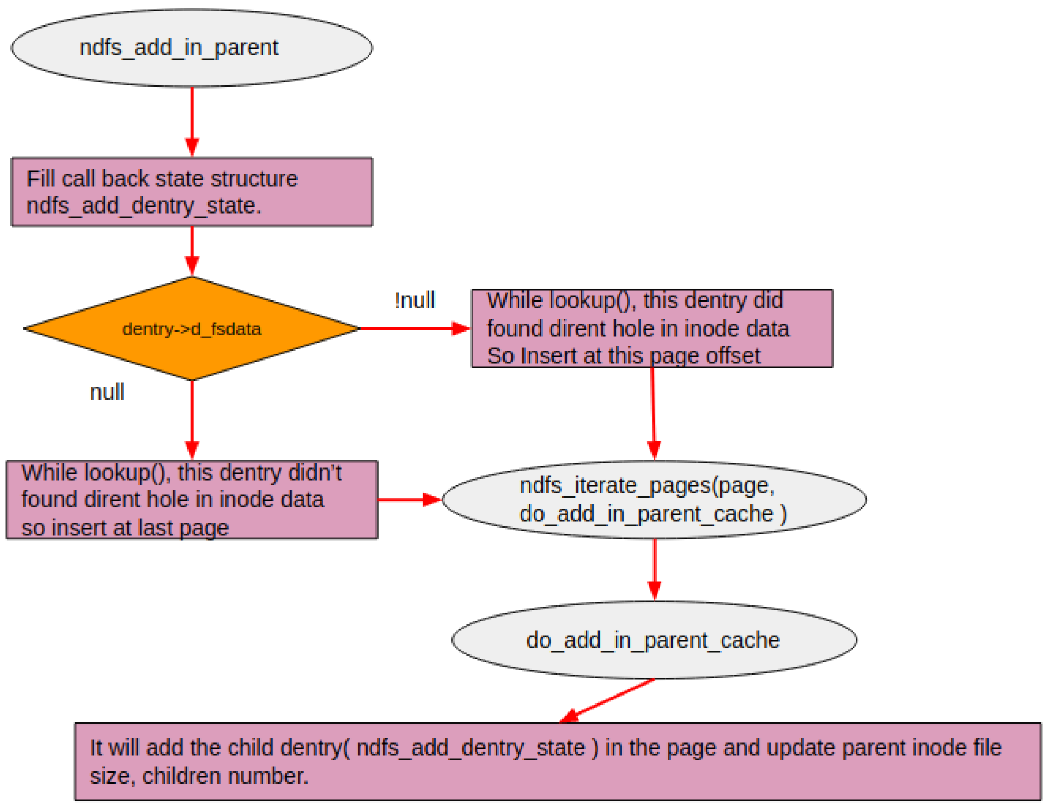

ndfs_add_in_parent
Fill call back state structure ndfs_add_dentry_state.
dentry->d_fsdata
!null
null
While lookup(), this dentry did found dirent hole in inode data So Insert at this page offset
While lookup(), this dentry didn't found dirent hole in inode data so insert at last page
ndfs_iterate_pages(page, do_add_in_parent_cache)
do_add_in_parent_cache
It will add the child dentry(ndfs_add_dentry_state) in the page and update parent inode file size, children number.

Figure 15: *ndfs_add_in_parent() call graph*

- `ndfs_add_in_parent` function adds the child dentry to the parent inode.
- `ndfs_add_in_parent` initializes the `ndfs_add_dentry_state` with dentry from the VFS layer.
- `dentry→fs_data` represents the page offset pointed to the inode data. while `lookup` call if hole is found in the pages it marks that page offset to this pointer so that next time while creating or deleting the dirent from the inode, process makes easier.
- If `dentry→fs_data` null `ndfs_add_in_parent` adds to the lastpage.
- Based on the `dentry→fs_data` pointer `ndfs_add_in_parent` adds to either a lastpage or page pointed by the `dentry→fs_data`.
- To iterate on that page this function calls to `ndfs_iterate_pages` calls with the page and the call back function `do_add_in_parent_cache`.
- `do_add_in_parent_cache` adds dirent to the page with child dentry pointed by call back structure `ndfs_add_dentry_state`.
- After successfully added the dirent in the parent inode, parent inode updates it's children number and file size if required.

• Finally add this inode to the inode cache by calling `avl_insert(ndfs_inodes_in_cache, inode number)`.

code locatoin `/nDiskFS/src/dir.c`

4.3 link

In nDiskFS `link` VFS layer call registered as `ndfs_link`. `ndfs_link` function creates hardlink to the inode pointed by the old_ dentry but it creates a new dirent in parent's inode of the new dentry.

`int ndfs_link(struct dentry * old_dentry, struct inode *dir, struct dentry *new_dentry)`

code locatoin `/nDiskFS/src/dir.c`

4.4 mkdir

In nDiskFS `mkdir` is a directory operation registered as `ndfs_mkdir`. `mkdir` called from VFS layer to create directory in the parent's directory. `ndfs_mkdir` allocates a new inode and create a dirent in the parent directory if the directory is not exists in the parent directory. `ndfs_mkdir` internally calls `ndfs_create_obj` with the mode contains that it is the directory to be created.

`int ndfs_create_obj(struct inode *dir, struct dentry *dentry, umode_t mode);`

This `ndfs_create_obj` function is using to create inode's object. So this function detailed description provided at `ndfs_create` directory operation.
code locatoin `/nDiskFS/src/dir.c`

4.5 rmdir

In nDiskFS `rmdir` is directory operation registered as `ndfs_rmdir`. `rmdir` called from VFS layer to remove directory from the parent directory. `ndfs_rmdir` removes the dirent from the parent inode data if the directory is empty. `ndfs_rmdir` internally calls `ndfs_remove` with the flag as it is directory to be removed.

`int ndfs_remove(struct inode *parent, struct dentry *child, int is_dir)`

```
  struct ndfs_update_dentry_state {
    struct dentry *child;
    uint64_t rem_size;
    uint64_t new_inode_num;
    uint32_t children_num;
};

This structure defined in /nDiskFS/src/dir.c
```

`ndfs_update_dentry_state` is a callback structure to `do_update_entry_in_cache` funtion.This structure is represention to a new updated dentry parameters.

Algorithm 7: ndfs_remove algorithm

```
Input: dir: parentInode, dentry: childDentry, is_dir: Directory flag
Result: To remove inode from the parent inode
if isDirectory() then
    if num_childeren then
        can't remove inode;
        return ERROR;
    else
        update dirent in the parent inode;
        ndfs_update_dentry();
    end
else
    update dirent in the parent inode;
    ndfs_update_dentry();
    if vfsinode→i_nline ¿ 1 then
        return;
    end
end
remove inode from parent ;
ndfs_rm_inode();
deattach vfsinode from the child dentry by d_drop(child);
```

- `ndfs_remove` function removes file and try to remove directory from the parent inode if the directory is empty as shown in the Algorithm 7.
- To remove the inode from the parent inode it deletes the dirent in the parent inode's data blocks. so update the dirent structure in the parent's inode data it calls to `ndfs_update_dentry`. More detailed description of this function is given below.

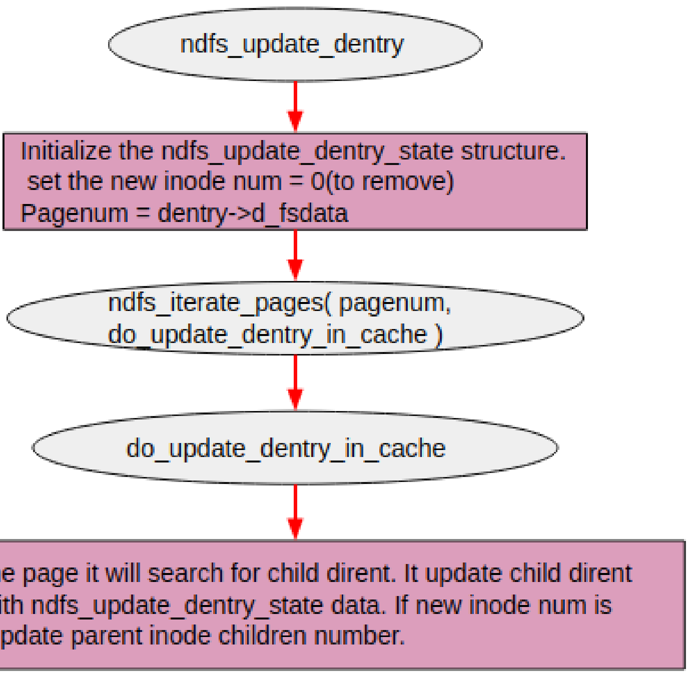

ndfs_update_dentry
Initialize the ndfs_update_dentry_state structure.
set the new inode num = 0(to remove)
Pagenum = dentry->d_fsdata
ndfs_iterate_pages(pagenum, do_update_dentry_in_cache)
do_update_dentry_in_cache
from the page it will search for child dirent. It update child dirent data with ndfs_update_dentry_state data. If new inode num is zero, update parent inode children number.

Figure 16: *ndfs_update_dentry() call graph*

  – `dentry`→`fs_data` is member in the dentry, it points dirent location in the inode page. Before calling to the `remove` VFS layer call it calls to `lookup`. While `lookup` it marks the dirent location in `fs_data` pointer.

- After compeletion of the dirent updation as shown in Figure 16, `ndfs_remove` tries to remove the inode. If inode to be remove is a file, then it looks for vfsinode i_nlink to zero because no other file object is referencing to this vfsinode.

- To remove inode(remove only from SSD) `ndfs_remove` calls to `ndfs_rm_inode`. More detailed description is given below.

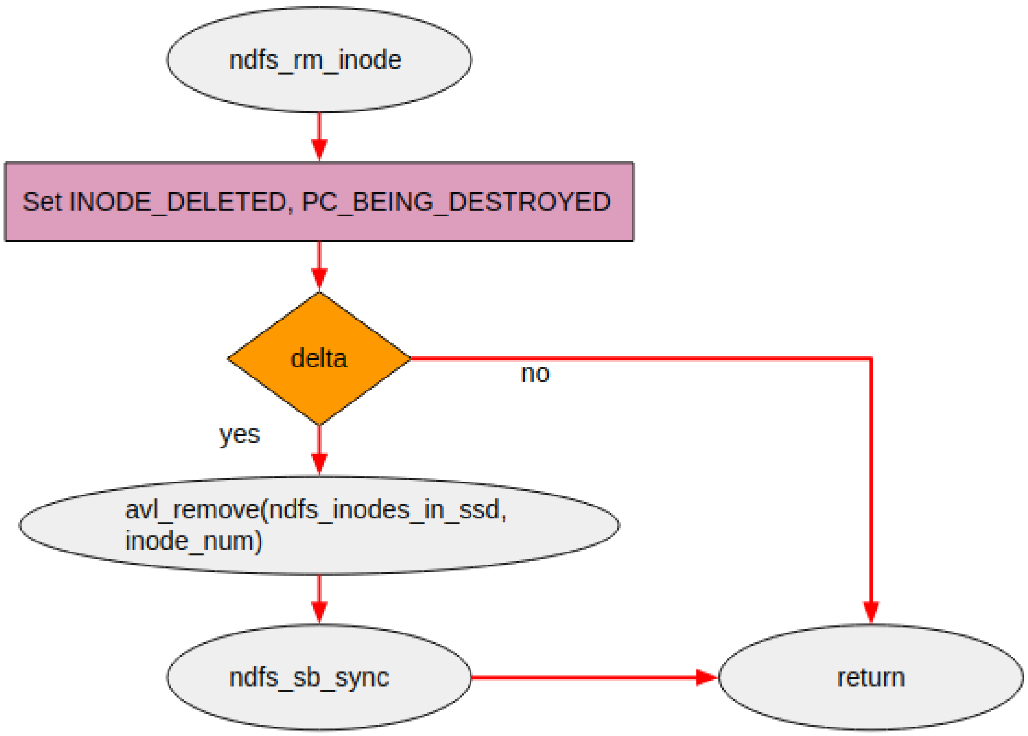

ndfs_rm_inode
Set INODE_DELETED, PC_BEING_DESTROYED
delta
no
yes
avl_remove(ndfs_inodes_in_ssd, inode_num)
ndfs_sb_sync
return

Figure 17: *ndfs_rm_inode() call graph*

- `ndfs_rm_inode` removes inode from the SSD if the inode had delta to it But it doesn't remove the inode from the HDD. HDD inode removed later by the VFS layter while calling superblock operation `destroy_inode`.
- While removing the inode from the SSD, it sets the `INODE_DELETED, PC_BEING_DESTROYED` flags.
- `INODE_DELETED` be usefull not to ascend the inode to the SSD while *ndfs_drop_inode* called.
- `PC_BEING_DESTROYED` sets to destroy pagecache of this inode.
- If this inode had delta then it deletes the delta from the `ndfs_inodes_in_ssd` tree by calling `avl_remove`. After removal of the inode delta it frees from SSD inode bitmap, decrease the SSD inodes count in the superblock as shown in Figure 17.
- Superblock has been updated so it syncs the superblock by calling `ndfs_sb_sync`.

- After removing the inode, `ndfs_remove` deattaches the vfsinode pointed by the dentry by calling `d_drop`.

code location `/nDiskFS/src/dir.c`

4.6 unlink

In nDiskFS `unlink` is directory operation registered as `_unlink`. `unlink` called from VFS layer to remove file from the parent directory. `ndfs_unlink` removes the dirent from the parent inode data. `ndfs_unlink` internally calls `ndfs_remove` with the flag as it is file to be removed.

```
int ndfs_remove(struct inode *parent, struct dentry *child, int is_dir)
```

This `ndfs_remove` function is using to remove inode's object. So this function detailed description provided at `ndfs_rmdir` directory operation.
code location `/nDiskFS/src/dir.c`

4.7 rename

In nDiskFS `rename` is directory operation registered as `ndfs_rename`. `rename` is operated between two directories within the filesystem not across the filesystems. This VFS layer function called when a inode moves from it's parent directory or inode wants to rename its name.

`int ndfs_rename(struct inode *old_dir, struct dentry *old_dentry, struct inode *new_dir, struct dentry *new_dentry, unsigned int flags)`

- `old_dentry is dentry object of the inode to be moved from its parent. It contains old inode number.`
- `old_dir is parent of the old_dentry.`
- `new_dir is the new parent of the old_dentry.`
- `new_dentry is the new dentry object to be added to the new_dir. It contains inode number if the file already exist in the new directory.`

When moving a inode from one directory to another directory with same name, then old_dentry, new_dentry contains same name but inode number is different because old_dentry has valid inode number, if new_dentry has valid inode number then the file is already exist in the new directory. If new_dentry has invalid inode number then no file exist with the same name in the new directory.

Algorithm 8: ndfs_rename algorithm

```
Input: old_dir: oldInode, old_dentry: oldDentry, new_dir: newInode, new_dentry:
       newDentry
Result: move inode from one directory to another directory
initialization;
kernel lookup on the both directories.
delete the oldDirent in the oldDirectory inode;
ndfs_update_dentry(oldDirectory, oldDentry, 0) ;
if if newDentry already exists in the newDirectory then
    update the newDirent in the newDirecoty with old inode number;
    ndfs_update_dentry(newDirectory, newDentry, old inode number ) ;
    delete the new dentry's inode;
    ndfs_rm_inode(newDentry inode number);
else
    add the newDentry's dirent with old inode number in the new directory's
     inode data ;
    ndfs_add_in_parent(sb, newDirectory, newDirent, old inode number );
end
```

- `ndfs_rename` operation steps are explained in the ndfs_rename Algorithm 8.
- `ndfs_update_dentry`, `ndfs_rm_inode` functions detailed descrition is given in the `rmdir` directory operation.
- `ndfs_add_in_parent` function description is given in the `create` directory operation.

code location `/nDiskFS/src/dir.c`

4.8 getattr

In nDiskFS `getattr` is a directory operation as well as file inode operation registered as `ndfs_getattr`. This function called from the VFS layer while `stat` system call calls. This function fills the `kstat` kernel object from the inode meta data. `ndfs_getattr` internally calls `ndfs_getattr_common` with different attributes depends upon the linux kernel version

`int ndfs_getattr_common(struct inode *ino, struct kstat *stat)`

`ndfs_getattr_common` fills the `kstat` object with mode, permissions, file size, block size, inode number, number of blocks in the inode.
code location `/nDiskFS/src/dir.c`

4.9 setattr

In nDiskFS `setattr` is a directory operation and also file inode operation registered as `ndfs_setattr`. This function called from the VFS layer while changing inode's mode, permissions.

`int ndfs_setattr(struct dentry *dentry, struct iattr *attr)`

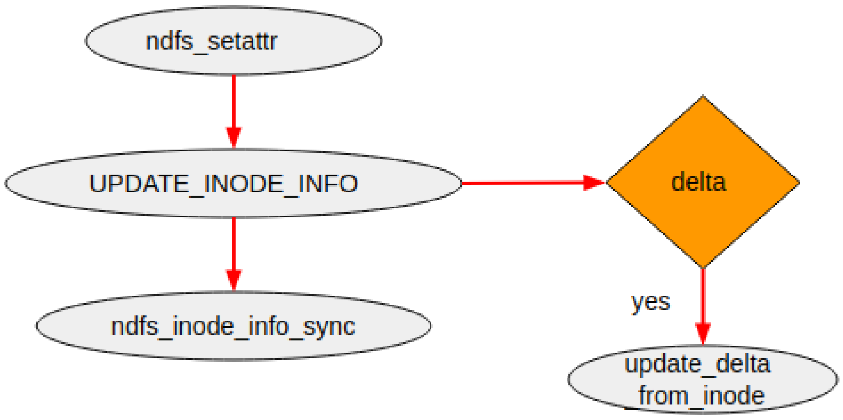


Figure 18: *ndfs_setattr() call graph*

- `ndfs_setattr` sets the mode, group permissions, user permissions based on the flags sent by the vfslayer `ATTR_MODE, ATTR_UID, ATTR_GID, ATTR_SIZE`.
- This function called while `chmod` system call calls.
- To update the inode in the SSD it called to `UPDATE_INODE_INFO` macro. This function checks weather the inode is present in SSD or not. If inode is in SSD it updates delta structure represented by this inode.
- After updation of the delta, it sets the SSD inode as outdated.
- `ndfs_setattr` syncs the inode by calling `ndfs_inode_info_sync`

code locatoin `/nDiskFS/src/dir.c`

5 File Operations

When VFS inode intailized, nDiskFS file operations are registered to the VFS inode. VFS inode is a member of in-memeory `ndfs_inode_info` and initialized when `looup`, `create` inode operations called on the parent inode.

Inode has 15 block pointers in the inode out of these 12 direct block pointer, 1 single indirect block pointer, 1 double indirect block pointer , 1 triple indirect block pointer.

5.1 read

In nDiskFS `read` regular file operation and registered as `ndfs_read`. This function take arguments of file pointer from VFS layer, user buffer, number of bytes to be copies to user buffer and offset of the file pointer. This function reads the data from the file, copies number of bytes to user buffer from the offset of the file pointer.

`ssize_t ndfs_read(struct file *filp, char __user *ubuf, size_t count, loff_t *offp)`

Algorithm 9: ndfs_read Algorithm

```
Input: filp: filePointer, ubuf: userBuffer, count: numberOfBytes, offp: fileOffset
Result: Copies file data to user buffer
ndfs_inode_info = file_inode(filp);
fileOffset = offp;
startPage = offset / NDFS_BLOCK_SIZE;
sizeToCopy = min ( count, filesize - offset );
lastPage = (startPage + fileOffset)/ NDFS_BLOCK_SIZE ;
intialize ndfs_rw_state ;
if is !DIRECT_IO then
    while page in the range(startPage, lastPage) do
        do_read_from_cache();
        copy_to_user();
    end
else
    do_direct_io(filp, user buffer, count, is_write);
    while Every block in the range(startPage, lastPage) do
        do block read to the userBuffer directly;
    end
end
fileOffset = fileOffset + count ;
return count;
```

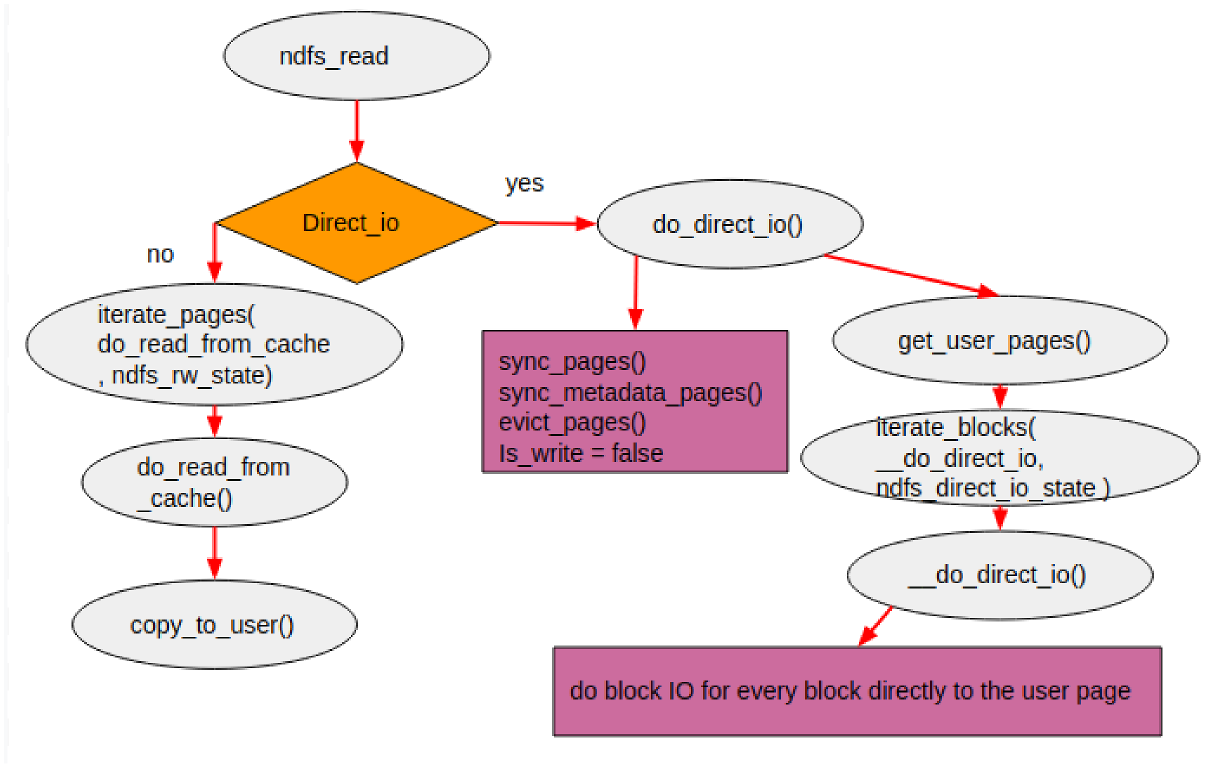

ndfs_read
Direct_io
yes
no
do_direct_io()
iterate_pages(do_read_from_cache , ndfs_rw_state)
sync_pages() sync_metadata_pages() evict_pages() Is_write = false
get_user_pages()
do_read_from _cache()
iterate_blocks(__do_direct_io, ndfs_direct_io_state)
__do_direct_io()
copy_to_user()
do block IO for every block directly to the user page

Figure 19: *ndfs_read() call graph*

```
    struct ndfs_rw_state {
    ndfs_inode_info *nii;
    size_t size;
    loff_t offset;
    uint64_t copied;
    char __user *buffer;
    uint64_t first_page_num;
    uint64_t last_page_num;
    unsigned int is_write;
    int last_block_overwrite;
    uint64_t last_block;
    struct bio *bio;
    struct list_head bios;
};

struct ndfs_direct_io_state {
    char __user *buffer;
    size_t count;
    int new_blocks_count;
    int num_newly_added_blocks;
    int num_new_dirtied_blocks;
    struct page **pages;
    uint64_t start_page;
    struct super_block *sb;
    uint64_t offset;
    uint64_t prev_block_num;
    uint64_t last_page;
    size_t rw_flag;
    struct bio *bio;
    struct list_head bios;
};

 These structures are defined in /nDiskFS/src/file.c
```

ndfs_rw_state is callback structure from ndfs_read to the function do_read_from_cache and ndfs_direct_io_state is callback structure from do_direct_io to __do_direct_io as shown in Figure 19.

If the file pointer sets the O_DIRECT or O_FSDIRECT flag then nDiskFS reads the data directly from the diskblocks to the user pages. If the DIRECT_IO flag is not set then filesystem reads the data from pagecache and copied it to the user buffer.

Regular ndfs_read

- ndfs_read reads the data from the pagecache pages and copies it to the user pages.
- ndfs_read calculates range of the pagecache pages to be copied by using offset and file size and initializes the ndfs_rw_state structure as shown in Algorithm 9.
- ndfs_read calls the iterage_pages with arguments initialized before. iterate_pages iterates through the pagecache pages and calls the call back function with call back structure.
- do_read_from_cache gets the page from the pagecache and calculate buffer offsets, page offsets. Buffer offset tells the starting position of the user buffer to be copied and page offset tells the offset of the page to be copied from.
- do_read_from_cache calls to copy_to_user function to do copy to the user buffer. copy_to_user is a kernel API to copy data from the pages to the user pages.

Direct-IO ndfs_read

- ndfs_read checks the file flags and if O_FSDIRECT or O_DIRECT is being set then this function calls to the do_direct_io with the flag is_write as false.
- nDiskFS performs the direct-io into the smallest units of block sizes. nDiskFS will fall back to regular read if the file offset is not aligned with blocks.
- While doing direct-io, filesystem directly read the data from the blocks to the user pages. Before doing direct-io, inode metadata, data to be sync with the ondisk data so do_direct_io calls the pagecache API's to sync with the on disk data and evict all the data from the pagecache.
- do_direct_io calls pagecache APIs like sync_pages, sync_metadata_pages to sync the data and evict_pages to evict the pages from the pagecache. These functions are shown in Algorithm 9
- After sync the pagecache state with on disk data and evict the pagecache pages from the pagecache, it initializes the ndfs_direct_io_state call back structure.
- To directly copy to user pages, kernel doesn't let to do that because kernel doesn't have the mappings to the user pages. To map the user pages to the kernel do_direct_io calls the get_user_pages.

- `do_direct_io` calls to `iterate_blocks` with the `ndfs_direct_io_state` call back structre and `__do_direct_io` call back function.
- `__do_direct_io` gets the blocks and do blockIO to the the user pages.
- After copied it to the user pages, `do_direct_io` unmaps the kernel mapping of the user pages and update the pagecache statistics.

Finally `ndfs_read` updates the file offset and return the count to the VFS layer.
code location `/nDiskFS/src/file.c`

5.2 write

In nDiskFS `write` regular file operation and registered as `ndfs_write`. This function take arguments of file pointer from VFS layer, user buffer, number of bytes to be copied from user buffer to the page and offset of the file pointer. This function reads the data from the user buffer, copies number of bytes to page from the offset of the file pointer.

`ssize_t ndfs_write(struct file *filp, const char __user *ubuf, size_t count, loff_t *offp)`

Algorithm 10: ndfs_write Algorithm

```
Input: filp: filePointer, ubuf: userBuffer, count: numberOfBytes, offp: fileOffset
Result: Copies data from user buffer to file pages
ndfs_inode_info = file_inode(filp);
if O_APPEND then
    fileOffset = filesize ;
end
startPage = fileOffset / NDFS_BLOCK_SIZE ;
sizeToCopy = count lastPage = (startPage + fileOffset)/ NDFS_BLOCK_SIZE ;
initialize ndfs_rw_state ;
if is !DIRECT_IO then
    while page in the range(startPage, lastPage) do
        do_write_in_cache();
        copy_from_user();
    end
else
    do_direct_io(filePointer, userBuffer, count, isWrite);
    while Every block in the range(startPage, lastPage) do
        do block write to the page from the userPage directly;
    end
end
offp = offp + count ;
return count;
```

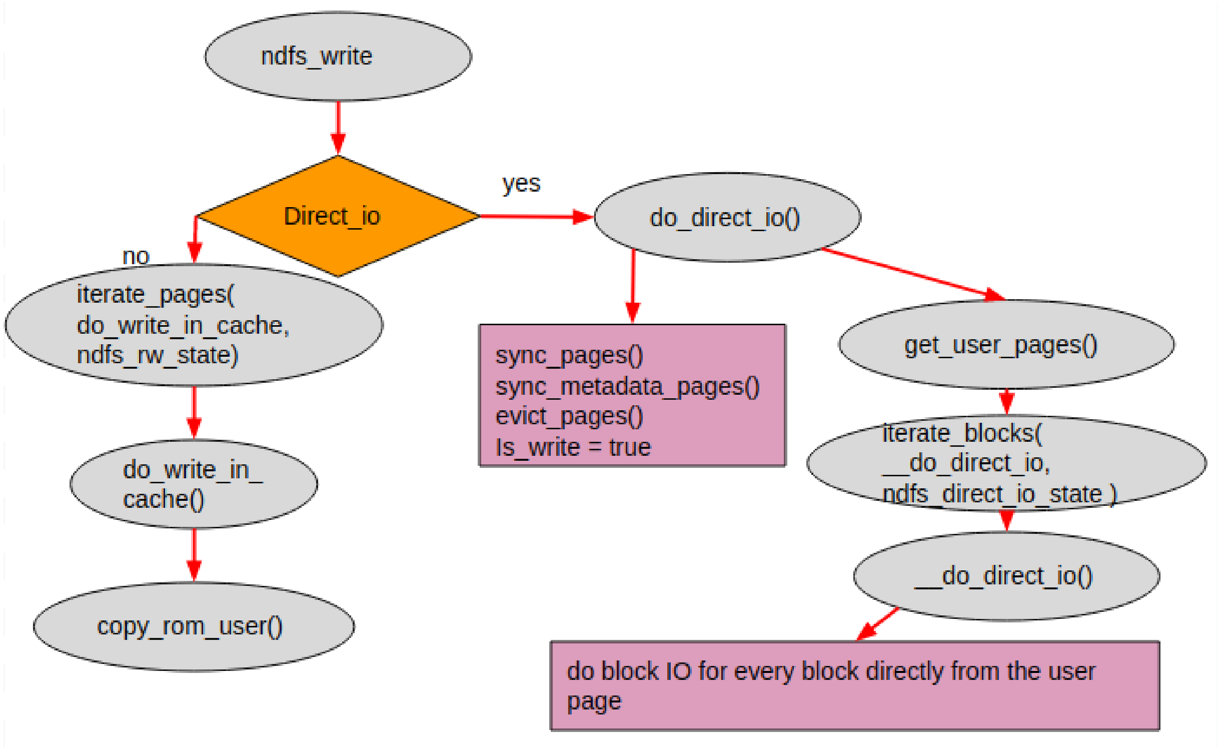

ndfs_write
Direct_io
yes
no
do_direct_io()
iterate_pages(do_write_in_cache, ndfs_rw_state)
sync_pages() sync_metadata_pages() evict_pages() Is_write = true
get_user_pages()
iterate_blocks(__do_direct_io, ndfs_direct_io_state)
do_write_in_ cache()
__do_direct_io()
copy_rom_user()
do block IO for every block directly from the user page

Figure 20: *ndfs_write() call graph*

`ndfs_rw_state` is callback structure from `ndfs_write` to the function `do_write_from_cache`.

If the file pointer sets the O_DIRECT or O_FSDIRECT flag then nDiskFS writes the data from the uses buffer to the disk blocks otherwise filesystem write the data from the user buffer to the page.

Regular write

- `ndfs_write` writes to the pagecache pages of file from the user buffer and call graph is shown in Figure 20.
- `ndfs_write` calculates the range of the pages to be written to the pagecache . If `O_APPEND` flag is set then nDiskFS writes to the end of the file and it initializes the `ndfs_rw_state` call back structure as shown in Algorithm 10.
- `ndfs_write` writes the pagecache pages of this file by calling to the `iterate_pages` with `do_write_in_cache` call back function and `ndfs_rw_state` call back structure.
- `do_write_in_cache` calculates the buffer offset and page offsets of the user buffer and page because buffer and page may not be a block aligned. After calculates the the buffer offset, page offset it calls to `copy_from_user` kernel API to write the data from the user buffer to the page.

Direct-IO write

`ndfs_read` checks the file flags and if O_FSDIRECT or O_DIRECT is being set then this function calls to the `do_direct_io` with the flag `is_write` as true. nDiskFS performs the direct-io into the smallest units of block sizes. nDiskFS will fall back to regular write if the file offset is not aligned with blocks. `do_direct_io` function calling description has given in `ndfs_read` file operation.

Finally `ndfs_write` updates the file offset and return the count to the VFS layer. code location `/nDiskFS/src/file.c`

5.3 check_flags

In nDiskFS `check_flags` regular file operation and registered as `ndfs_check_flags`. When the user space applications set any flags, VFS layer calls to this function. Filesystem can validate the flags if require file system can invalidate the flags also.

`int ndfs_check_flags(int flags)`

nDiskFS modifies the linux kernel and add the O_FSDIRECT new flag. This flags helps the file system to implement direct-IO. Linux kernel doesn't allow set the O_DIRECT flag if kenel doesn't register address mapping operations so nDiskFS set the O_FSDIRECT when O_DIRECT flag has set.
code location `/nDiskFS/src/file.c`

5.4 read_iter

In nDiskFS, `read_iter` file operation is implemented using the `ndfs_file_read_iter` callback and its prototype is,

`ssize_t ndfs_file_read_iter(struct kiocb *iocb, struct iov_iter *to)`

Algorithm 11: ndfs_file_read_iter Algorithm

```
Input: iocb: I/O control block, iovIter: I/O Vector
Result: Transfer data to user space buffer from the file pages
FilePTR = iocb→ki_filp;;
i = 0 ;
while i < iovIter→segments do
    ndfs_read(FilePTR, iovIter[i].base, iovIter[i].count, FilePTR.pos );
    i++;
end
```

VFS layer invokes this operation with `iov_iter` as an argument which contains segments of user buffers and lengths along with the `kiocb` object which contains the file pointer. The implementation of `ndfs_file_read_iter` is shown in Algorithm 11 where the handler iterates on the `iov_iter` vector and invokes the regular `read` implementation of nDiskFS.
code locatoin `/nDiskFS/src/file.c`

5.5 write_iter

In nDiskFS `write_iter` regular file operation registered as `ndfs_file_write_iter`. callback and its prototype is,

`ssize_t ndfs_file_write_iter(struct kiocb *iocb, struct iov_iter *to)`

Algorithm 12: ndfs_file_write_iter Algorithm

```
Input: iocb: I/O control block, iovIter: I/O Vector
Result: transfer data from file pages to user buffer in iov_iter
filePTR = iocb→ki_filp;;
i = 0 ;
while i < iovIter→segments do
    ndfs_write(FilePTR, iovIter[i].base, iovIter[i].count, FilePTR.pos );
    i++;
end
```

VFS layer invokes this operation with `iov_iter` as an argument which contains segments of user buffers and lengths along with the `kiocb` object which contains the file pointer. The implementation of `ndfs_file_write_iter` is shown in Algorithm 12 where

the handler iterates on the `iov_iter` vector and invokes the regular `write` implementation of nDiskFS.
code location `/nDiskFS/src/file.c`

5.6 open

In nDiskFS `open` regular file operation registered as `ndfs_open`. `open` called when userspace applications opens file or directory. `open` call comes with flags like `O_TRUNCATE, O_DIRECT, O_APPEND`. Before calling to `open` VFS layer first calls to `lookup` on the directory and builds the VFS inode.

`int ndfs_open(struct inode *inode, struct file *file)`

Algorithm 13: ndfs_open Algorithm

```
Input: inode: FileVFSInode, file: filePTR
Result: Opens the file in nDiskFS and set appropriate file object parameters
updateNumOpenFiles() ;
ndfs_inode_info = INODE(fileVFSInode);
if flags | O_TRUNCATE then
    updateInodeMembers() ;
    FileSize = 0 ;
    Pattern = unknown ;
    stop migrations = 0 ;
    NumFileReads = 0 ;
    file_trucate() ;
    fileOffset = 0 ;
end
if flags | O_APPEND then
    fileOffset = file_size ;
end
if flags | _DIRECT then
    flags | = O_FSDIRECT ;
end
update file mapping address space operations to be NULL;
```

- `ndfs_open` sets the the in-memory inode parameters based on the flags sent by VFS layer as shown in Algorithm 13.
- `ndfs_inode_info` gets from the VFS inode. Before open a file or directory VFS layer first performs `lookup` in the parent directory. While `lookup` called nDiskFS builds the in-memory inode.
- If `O_TRUNCATE` flag set then nDiskFS updates inode members like filesize, number of file reads. It sets the pattern to unknown so that this inode data will not be migrated to SSD. It stops the migrations by setting to zero. Finally `ndfs_open` truncates whole file.
- If `O_APPEND` flag sets then nDiskFS updates the file position to current file size.
- If `O_DIRECT` flag sets then nDiskFS update file flag to `O_FSDIRECT`.

code location `/nDiskFS/src/file.c`

5.7 release

In nDiskFS `release` file operation registered as `ndfs_release`. When file being closed then VFS layer calls to `release` function.

`int ndfs_release(struct inode *inode, struct file *filp)`

nDiskFS just returns success when `ndfs_release` called.

code location `/nDiskFS/src/file.c`

5.8 llseek

In nDiskFS `llseek` file operation registered as `ndfs_llseek`. This function called when userspace applications wants to set the file position.

`loff_t ndfs_llseek(struct file *filp, loff_t off, int whence)`

Algorithm 14: ndfs_llseek Algorithm

```
Input: file: filePTR, off: fileOffset, whence: Flag
Result: sets file position
ndfs_inode_info = INODE_INFO(VFSInode);

if whence == SEEK_SET then
    filePosition = fileOffset
end
if whence == SEEK_CUR then
    filePosition += fileOffset
end
if whence == SEEK_END then
    filePosition = fileSize + fileOoffset
end
```

- `ndfs_llseek` sets the file_position based the flag sent by the VFS layer as shown in Algorithm 14.
- If the flag is `SEEK_SET` then it sets file_position to offset sent by the VFS layer.
- If the flag is `SEEK_CUR` then it adds the offset to the current file_position
- If the flag is `SEEK_END` then it adds the offset to the current file_size.

code location `/nDiskFS/src/file.c`

5.9 mmap

nDiskFS has it's own caching mechanism and policies so `mmap` called to filesystem. In nDiskFS `mmap` file operation registered as `ndfs_mmap`. When `mmap` called, it allocates a VMA(virtual memory area) to this mmap and expects the filesystem to be register VM area operations. nDIskFS registers VM area operations and these operations being invoked from VFS layer whenever if a page faults in the VMA or if any page dirtied it tells the filesystem through the VM area registered operations.

`int ndfs_mmap(struct file *file, struct vm_area_struct *vma)`

```
static struct vm_operations_struct ndfs_file_vm_ops = {
  .fault = ndfs_filemap_fault,
  .map_pages = ndfs_filemap_map_pages,
    .page_mkwrite = ndfs_filemap_page_mkwrite,
};
This structure is defined in /nDiskFS/src/file.c
```

In nDiskFS `ndfs_mmap` registers the `vm_ops` to `ndfs_file_vm_ops`.

5.9.1 ndfs_filemap_fault

In nDiskFS `fault` VM opeartion registered as `ndfs_filemap_fault`. This function called when a page fault happens to the VM area of the file.

`int ndfs_filemap_fault( struct vm_fault *vmf )`

Algorithm 15: ndfs_filemap_fault Algorithm

```
Input: vmf: VmFaultStructure
Result: Fill the page in VmFaultStructure struct
filePtr = vmf→vma→vm_file;
pageOffset = vmf→pgoff ;
page = get_page_from_the_pagecache(fileInode, pageOffset);
vm_fault-> page = page ;
get_page(page) ;
```

`vm_fault` structure contains file pointer and page pointer to be filled and page offset. nDiskFS searches the page in the pagecache and fills the page pointer to the pagecache page and get the reference to the page as shown in Algorithm 15. VFS layer expects the filesystem to lock the page before returning to the VFS layer.
code location `/nDiskFS/src/addr.c`

5.9.2 ndfs_filemap_page_mkwrite

In nDiskFS `page_mkwrite` VM opeartion registered as `ndfs_filemap_page_mkwrite`. This function called when a page is dirtied in the VM area then tells the filesystem through `page_mkwrite` opearation.

`int ndfs_filemap_page_mkwrite( struct vm_fault *vmf )`

Algorithm 16: ndfs_filemap_page_mkwrite Algorithm

```
Input: vmf: VmFaultStructure
Result: mark the vm_fault→page as dirty
filePtr = vmf→vma→vm_file;
pageOffset = vmf→pgoff ;
page = get_page_from_the_pagecache(fileInode, pageOffset);
mark the page as dirty ;
mark_page_to_be_dirty(page) ;
```

`vm_fault` structure contains dirty page offset. nDiskFS searches in the pagecache pages and mark the page as dirty as shown in Algorithm 16 one and later this page syncs to the disk.
code location `/nDiskFS/src/addr.c`

5.9.3 ndfs_filemap_map_pages

In nDiskFS `map_pages` VM opeartion registered as `ndfs_filemap_map_pages`. This function called to map the pages in the VM area. If the filesystem performs that no more faults to this VM area.

`void ndfs_filemap_map_pages(struct vm_fault *vmf, pgoff_t start_pgoff, pgoff_t end_pgoff)`
code location `/nDiskFS/src/addr.c`

5.10 fsync

In nDiskFS `fsync` file operatoin registered as `ndfs_fsync`. This function called from VFS layer when the userspace applications wants the data to be consistent in the disk. After completion of this function the data would be written to the disk.

`int ndfs_fsync(struct file *filp, loff_t offl, loff_t offr, int datasync)`

Algorithm 17: ndfs_fsync Algorithm

```
Input: filp: filePtr, offl: startOffset, offr: lastOffset, datasync: datasyncFlag
Result: writes the file data to the disk
ndfs_inode_info(filePtr);
startPage = startOffset / NDFS_BLOCK_SIZE;
lastPage = lastOffset / NDFS_BLOCK_SIZE;
sync_pages(Inode, startPage, lastPage );
if datasync then
    sync_metadata_pages(Inode, startPage, lastPage);
end
blkdev_issue_flush(block device);
```

- When userspace applications wants the data to be persistent in the disks then applications calls `fsync`.
- `ndfs_fsync` caluculates the starting page number , last page number from the agruguments it gets.
- `ndfs_fsync` calls to `sync_pages(Inode, startPage, lastPage)` to sync the file data to the disk as shown in Algorithm 17.
- If datasync flag is not set then this function syncs the metadata pages by calling `sync_metadata_pages(Inode, startPage, lastPage)`
- Finally nDiskFS flushes the disk caches of the underlying disk. It makes sure that data is persistent in the disk.

code location `/nDiskFS/src/file.c`

5.11 unlocked_ioctl

In nDiskFS `unlocked_ioctl` file operation is registered as `ndfs_ioctl`. These functions called with a command. Currently nDiskFS supports `NDFS_IOCTL_ASCEND, NDFS_IOCTL_DESCEND` commands. `NDFS_IOCTL_ASCEND` command forces the inode to be migrated to SSD. `NDFS_IOCTL_DESCEND` command forces the inode to be removed from the SSD.

`long ndfs_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)`
code location `/nDiskFS/src/file.c`

5.12 fallocate

In nDiskFS `fallocate` file operation is registered as `ndfs_fallocate`. This function called when a userspace applications wants to allocate more space for file without doing any read or write operation.

`long ndfs_fallocate(struct file *file, int mode, loff_t offset, loff_t len)`

`ndfs_fallocate` increase the file size without doing any read or write. Later while writing to a hole(block number is zero) it allocates blocks on demand.
code location `/nDiskFS/src/file.c`

5.13 readdir

In nDiskFS `readdir` directory file operation registered as `ndfs_readdir`. This function reads the directory content and pass it to the userspace . `int ndfs_readdir(struct file *filp, struct dir_context *ctx)`

Algorithm 18: ndfs_readdir Algorithm

```
Input: filp: filePtr, ctx: directoryContext
Result: Reading directory content
startPage = file→pos / NDFS_BLOCK_SIZE ;
lastPage = file_size / NDFS_BLOCK_SIZE ;
while For every page in the pagecache range(startPage, lastPage ) do
    get_page_from_pagecache(pageOffset);
    while For every dirent in the page do
        get_dirent_from_page(page) ;
        if direntisValid() then
            ret = dir_emit(fileName, fileLen, inode_number);
            if !ret then
                return ;
            end
        end
        dirent = nextDirent ;
    end
end
```

This function emits the dirents in the directory until userspace buffer is full or directory data finishes as shown in the Algorithm 18.
code location `/nDiskFS/src/dir.c`

6 Understanding Locks in pagecache

```
Global Locks(Locks that are in PCC( struct page_cache_container )):

1. Pcc_lru_list_lock

This lock is protecting 2 global lists those are
pcc_lru_list_head[HDD(0)]
pcc_lru_list_head[SSD(1)]

When and where this is used:

delete_page_cache()
```

```
- When delete page cache from nii(ndfs_inode_info) We delele above lists from the
global lists. While deleting we protect global lists.

ndfs_iterate_pages_with_window()
-- When Every iteration starts on a PC first It removes the PC( page_cache) from the
 global lists ( mentioned above).
-- After all iterators on that PC are completed then the one who is going last will
again add this pc to global lists(( mentioned above).

migrator_thread_fn()
-- When the migrator thread is before starting to migrate that file(PC) to SSD. This
 function will make sure that no eviction is going on that PC.  So It will take out
This PC from global lists.
-- After the migrator thread completed migration this thread again add the PC to the
Global lists

Select_pc_for_eviction()
-- When eviction thread try to select PC from the global lists . evictor thread
first take lock  then select the PC.
-- When the evictor thread finds out that while doing eviction some iterator or
migrator guy comes then he removes this PC from global lists. So he took the lock.

sync_blocks()
-- when syncer thread after syncing some blocks of the PC those will be in clean
state so syncer thread add this PC to global lists.

2. Pcc_dirt_list_lock
This lock is protecting 2 global lists those are
pcc_dirt_list_head[HDD(0)]
pcc_dirt_list_head[SSD(1)]

When and where this is used:

add_cp_to_pc_dirt_list()
-- When a block is dirtied in that PC. if the above lists are not empty then this
Function adds this PC to global lists.

delete_page_cache()
- When delete page cache from nii(ndfs_inode_info) We delele above lists from the
global lists. While deleting we protect global lists.

sync_blocks()
--When dirty syncer thread try to select PC from the global dirty lists . syncer
 thread first take lock the then select the PC.

3. pcc_migr_list_lock
This lock is protecting global list pcc_migr_list_head.

When and where this is used:
```

ndfs_iterate_pages_with_window()
-- When every iterator comes he should make sure that while doing iteration no
migration should happen . So first it removes this PC from global migration list.
-- After all iterators on that PC are completed then the one who is going last checks
That If the access pattern is detected then he adds this pc to global migration list.

Migration_thread_fn()
-- If access pattern recognised then it must in the global migration list. So
migration thread lock on the global migration list and select one PC.

4. pcc_sync_lock
No one is using

5. pcc_evict_lock
No one is using

File Level Locks

1. Pc -> p_lock
This lock is protecting the page_cache structure while multiple readers,
writers are working on the same PC.

When and where this is used:
file_truncate(), ndfs_truncate()
-- which internally calls get_diff_block() to lock.

get_diff_block()
-- While multiple iterators come and trying to work on same diff structure are trying
To create diff structure. This make sure stop doing this.

cache_page_destroy()
-- Took lock while removing cp(struct cache_page) structure of diff block from avl
ind tree.

delete_page_cache()
-- when deleting pc structure and kfree all the structures in avl we took it.
But its a sin we took lock and destroy the structure.

do_iterate_pages()
-- when adding a cp structure to a avl tree. Make sure one of the iterator is adding

ndfs_iterate_pages_with_window(),do_file_sync(),
do_evict_from_page_cache(),do_file_metadata_sync()
-- Making sure that the iterator gets valid pc. Valid means in between can anyone
(evictor guy)Destory that pc.

sync_blocks()
-- If file trucate has been called it will not destroy this pc . becuase syncing
is going on, later syncer do that destruction on that pc.

```
cache_get_page()
-- when adding a indirect cp structure to a avl->ind_tree. Make sure that one of
iterator is adding

get_indirect_cache_page()
-- This function is not been used.

2. Pc ->  pc_lru_list_lock[HDD]
   Pc ->  pc_lru_list_lock[SSD]
This Lock protects when we are modifying the pc lru lists of
Pc -> pc_lru_list_head[HDD]
Pc -> pc_lru_list_head[SSD]
And to protect some variables like ( pc->num_cb( file_dec_num_cb()) )

When and where this is used:

cache_page_ref():
--when it wants the get refcount on the page. if that page is in  pc lru lists (
mentioned above). It removes the those lists.

__cache_page_rel_final()
-- when the cb ( page is in struct cache_page_block) refcount becomes 0. It
adds it to the pc lru lists.

cache_page_destroy()
-- when a cb ( struct cache_page_block) wants to delete ( while delete diff block)
It will then and there itselt removes from the pc lru lists and has been evicted.

delete_page_cache()
-- here not necessary i think

select_pc_for_eviction()
-- while doing eviction it locks on the pc lru lists and it evicts pages from
 the PC.

do_evict_from_page_cache()
-- when we are evicting a cb from the PC we hold this lock.

3. Pc ->  pc_dirt_list_lock[HDD]
   Pc ->  pc_dirt_list_lock[SSD]
This Lock protects when we are modifying the pc dirt lists of
Pc -> pc_dirt_list_head[HDD]
Pc -> pc_dirt_list_head[SSD]

When and where this is used:

cache_page_destroy()
-- when a cb ( struct cache_page_block) wants to delete ( while delete diff block)
It will then and there itselt removes from the pc dirt lists and has been evicted.
```

add_cp_to_pc_dirt_list()
-- when a cb is dirtied(page is dirited inside this) it adds that cb to the pc dirt
Lists

cache_page_moved()
-- when a cache_page_block(cb) is moved from one disk to another disk we should
Mark it has dirty. And add it to the dirty list

delete_page_cache()
-- Not necessary i think

sync_blocks(), do_file_sync(), do_file_metadata_sync()
-- while doing syncing it locks on the above lists and it sync pages from the
 PC.

4. Pc -> pc_migr_lock [semaphore]
This controls number of iterators, migrators synchronization

```
Iterators :   ndfs_iterate_pages_with_window(),
  do_file_sync(),
  do_file_metadata_sync()
  do_evict_from_page_cache()
 Migrator:   migrator_thread_fn()
```

Every iterator will do down_read(), up_read()
Migrator will do up_write() up_read()