Addition of Ext4 Extent and Ext3 HTree DIR Read-Only Support in NetBSD

Hrishikesh
<hrishi.goyal@gmail.com>

Christos Zoulas
<christos@NetBSD.org>

Abstract

This paper discusses the project ***'Implementation of Ext4 and Ext3 Read Support for NetBSD kernel'*** done as a part of Google Summer of Code 2016. The objective of this project was to add the support of Ext3 and Ext4 filesystem features viz., Ext4 Extents and Ext3 HTree DIR in read only mode by extending the code of existing Ext2fs implementation in NetBSD kernel,.

1 Introduction

The fourth extended file systems or Ext4 as it is commonly known, is the default filesystem of many Linux distributions. Ext4 is the enhancement of Ext3 and Ext2 filesystems which brought improved efficiency and more reliability. In many ways, Ext4 is a deeper improvement over Ext3, as it introduces a number of new features like Extent support, delayed allocation and many more.

Traditionally, in NetBSD, Ext2 is the base filesystem and some features from Ext3 and Ext4 are supported. However, there has been lack of support of main features of Ext3 and Ext4 filesystems in NetBSD. In machines running NetBSD, dual booted with other OS supporting Ext4 filesystem, getting access to Ext4 files through NetBSD, is a primary need. Read-only support of Ext4 filesystem in NetBSD is also important for further evolution of Ext4 into full read-write support.

Earlier Ext2 and Ext3 had the limitation on the size of the file. They used 32 bit block number to access the data blocks. So, that limited the maximum size of file to be 2^32 * block size(eg. 4kb) = 16TB. Despite this Ext4 filesystem supports very large files as it has 48 bits to address a block. The access time for such large files on following traditional indirect block pointer approach is significantly high, because accessing a data block involves lots of indirection. Ext4 Extent provides a more efficient way of indexing file data blocks which especially reduces access time of large files by allocating contiguous disk blocks for the file data.

The directory operations (create, open or delete) in Ext2fs requires a linear search of an entire directory file. This results in a quadratically increasing cost of operating on all the files of a directory, as the number of files in the directory increases. The HTree directory indexing extension added in Ext3fs, addresses this issue and reduces the respective operating cost to nlog(n).

This project added these two features in the NetBSD kernel. The project was started with the implementation of Ext4 Extent in read-only mode and then the next feature Ext3 Htree DIR read support was implemented.

2 Ext4 Extents

As mentioned earlier Extent block mapping is an efficient approach of mapping logical to physical blocks for large contiguous files. But before understanding the Extents, let's try to understand the traditional indirect block mapping approach used in Ext2/3 filesystems, for indexing the file data blocks.

2.1 Indirect Block Mapping

The information for the data blocks of a file is stored in the i_data field of the inode structure. The first 12 entries of i_data contain the block numbers of the first 12 data blocks of the file. Then it contains the block number for the Indirect blocks. That block contains the array of block numbers which point to the data. Similarly, there is double indirect block and triple indirect block (figure 1).

So if we need to get the data from a very large file, we need to go through those indirections.

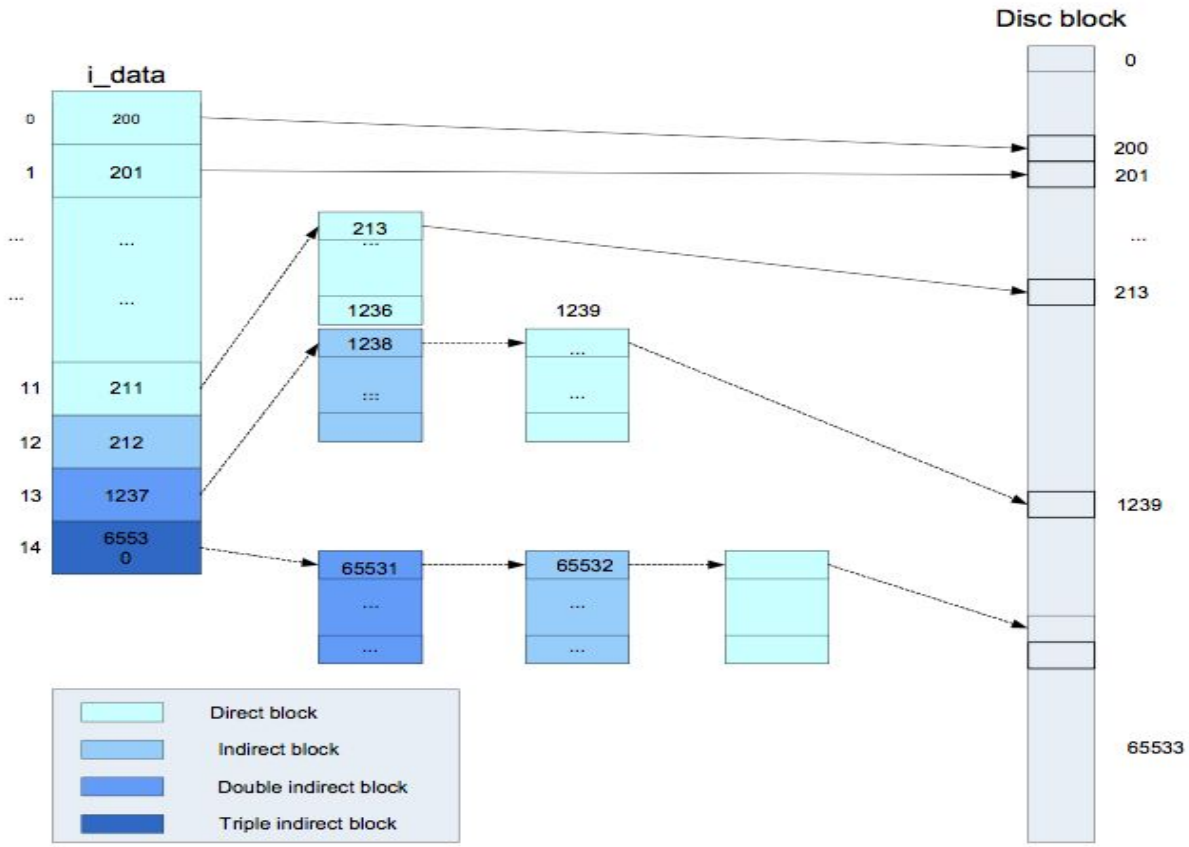


Figure 1 Indirect block mapping

2.2 Extent

In extent based block mapping, the i_data of inode contains Extent structures [4]. There is a extent header, Extent and Extent index.

The Extent is implemented as a B+ Tree, with a disk block allocated for each node in the tree except for the root node. All nodes contain a header and either extent structures or extent index structures(Table 1). Specifically the leaf nodes have the Extent structure and others nodes have Extent Index Structure. The header contains the number of valid entries in the node, the capacity of entries the node can store, the depth of the tree, and a magic number. The magic number can be used to differentiate between different versions of extents, as new enhancements are made to the feature, such as increasing to 64-bit block numbers.

The Extent struct is a single descriptor which represents a range of contiguous physical blocks (Figure 2). The physical block field in an extents structure takes 48 bits. A single extent can represent 215 contiguous blocks, or 128 MB, with 4 KB block size. The MSB of the extent length is used to flag uninitialized extents, used for the preallocation feature. The extent index structures are used for non leaf nodes. It contains the block number of the block where next level of nodes are stored. The logic can be described by the following figure 2. Four extents can be stored in the ext4 inode structure directly. This is generally sufficient to represent small or contiguous files. For very large, highly fragmented, or sparse files, new blocks with extent index structures are used. In this case a constant depth extent tree is used to store the extents map of a file.

ext4_extent_idx	
ei_blk	/* indexes logical blocks */
ei_leaf_lo	/* points to physical block of the * next level */
ei_leaf_hi	/* high 16 bits of physical block */
ei_unused	

Table 1 (a) Extent Index Structure

ext4_extent_header	
eh_magic	/* magic number: 0xf30a */
eh_ecount	/* number of valid entries */
eh_max	/* capacity of store in entries */
eh_depth	/* the depth of extent tree */
eh_gen	/* generation of extent tree */

Table 1(b) Extent Header Structure

ext4_extent	
e_blk	/*first logical block */
e_len	/* number of blocks */
e_start_hi	/* high 16 bits of physical block*/
e_start_lo	/* low 32 bits of physical block */

Table 1 (c) Extent tree extent structures

Figure 2 shows the layout of the extents tree. The root of this tree is stored in the ext4 inode structure and extents are stored in the leaf nodes of the tree.

2.3 Extent Tree Traversals

In the read operation for the requested data of a file having Extent tree as an index for file data blocks, getting the map of logical block number to physical block number is a crucial step. To map file's logical block number to corresponding disk block number, one needs to travel Extent tree beginning from the root stored in the i_data field of the inode. The header field 'eh_ecount' in each node gives the number of children of current node as each extent index contains a pointer to a child node. The nodes in next level of the Extent tree may further be non leaf nodes. All the nodes in the deepest (leaf) level of the tree contain extent structs which represent a chunk of contiguous file data blocks. All the indices in a node are sorted according to 'ei_blk' value in the index, hence binary search is used in block search algorithms [11].

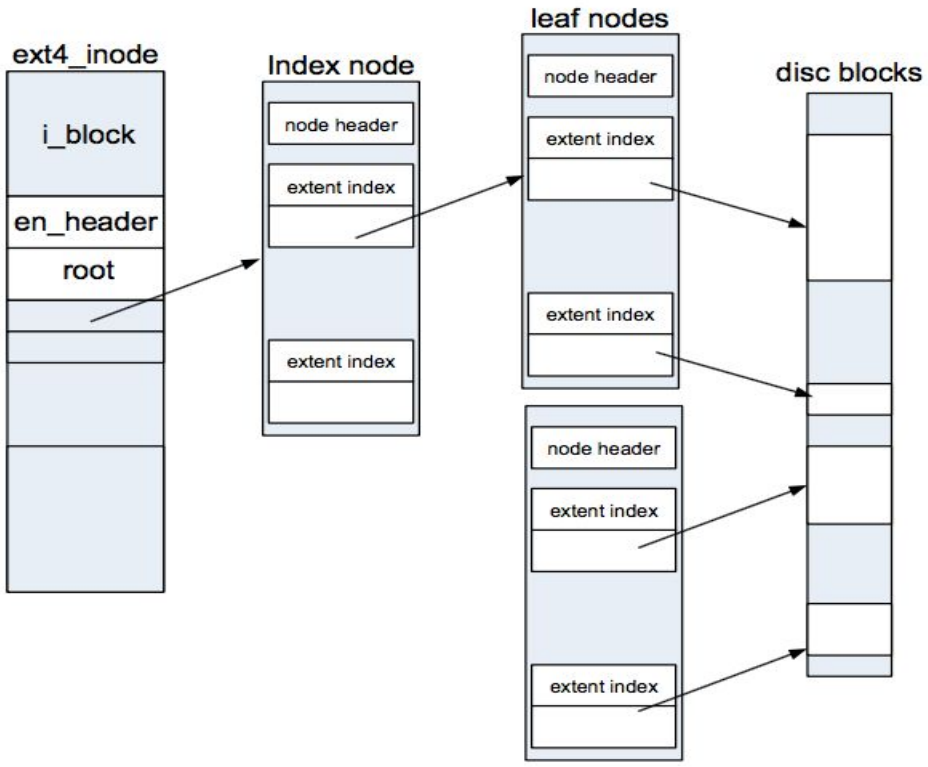


Figure 2 A typical extent tree

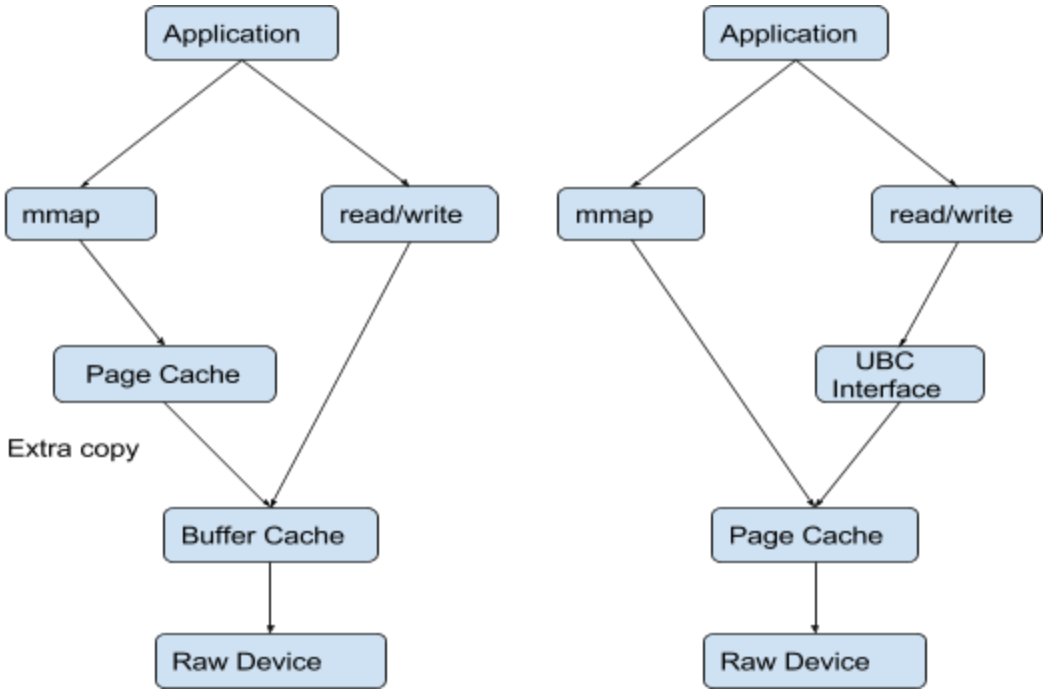
To find a physical block number corresponding to a file logical block, Extent tree is traversed from top to bottom in the direction of the extent index with 'ei_blk' value just less than or equal to the target hash value at each level. At the leaf level, nearest extent is found in the same manner which contains the pointer to the contiguous physical blocks. During tree traversal, path is stored into an array which represents the exact path of a disk block corresponding to file logical block.

3 UBC Interface in NetBSD

Previous section gives an overview of extent structs and process of getting physical block corresponding to target file block. The core logic of the extent remains same irrespective of the underlying filesystem and OS. There already exists a number of extent map implementations for the file data block indexing. For example, in FreeBSD, Extent read support had already been implemented. Needless to say, the reimplementation of Extent core-logic was not a challenge here, the challenge was to deal with different standards of both kernel and to figure out how FreeBSD's extent implementation fit the NetBSD kernel without the need of much modifications. So let's understand the difference in VFS interfaces in the two kernels, FreeBSD and NetBSD. The major differences in the VFS layer of both the kernels are due to having different strategies to approach the *Double Cache Problem* [12]. This section, describes the *Double Cache Problem* and UBC interface which addresses this problem in NetBSD. We will also look at FreeBSD's strategy to address this problem.

3.1 Double Cache Problem

There are two mechanisms to access filesystem data. One is memory mapping and the other is I/O system calls such as read() and write(). These two mechanisms are handled by two different underlying subsystems which are virtual memory subsystem and I/O subsystem respectively. These two subsystems have their own (different) caching mechanisms. Virtual memory subsystem uses "page cache", and I/O subsystem uses "buffer cache"



(a) *(b)*

Figure 3 Double Cache problem and UBC Interface

to store their data (see figure 3). These two caching techniques eventually result into dual caching of data. This double-caching of data results in many problems viz *Memory Inefficiency, Bad Performance, Data Inconsistency.* These problems are collectively known as *Double Cache Problem* (BCP).

3.2 Addressing DCP: In NetBSD and FreeBSD

FreeBSD handles this problem by allocating same physical memory for both buffer cache and page cache. Whereas NetBSD provides Unified Buffer Cache (UBC) interface to read file data directly into the page cache without going through the buffer cache. Table 2 below summarizes the different mechanisms to access file data in NetBSD and FreeBSD and their subsystems which handle this. In FreeBSD, extents read code was implemented above buffer cache memory interface and in NetBSD, it is implemented above UBC interface. So before understanding how FreeBSD's extent code can be imported to NetBSD over UBC interface we must understand the UBC interface.

3.3 UBC interface and VFS in NetBSD

UBC is a subsystem in NetBSD, which solves the problems with the two-cache model. UBC model allows to store file data in the page cache for both read()/write() and mmap() accesses. File data is read directly into the page cache without going through the buffer cache by creating two new VOPs which return page cache pages with the desired data, calling into the device driver to read the data from disk if necessary. Since page cache pages aren't always mapped, it has a new mechanism for providing temporary mappings of page cache pages, which is used by read() and write() while copying the file data to the application's address space. Figure 5 shows the complete flow of control of mmap() and read/write system calls in NetBSD. The mmap() system call is handled by UVM which eventually calls VOP_GETPAGES() and VOP_PUTPAGES(). The only thing remained here, to be implemented by a filesystem are these functions. Fortunately, the kernel already has the genfs_getpages()/ genfs_putpages() routines which can be mapped directly to VOP_GETPAGES/ VOP_PUTPAGES(). genfs_getpages() / genfs_putpages are generic functions which call VOP_BMAP() for getting file logical block to physical block mapping. Hence the filesystem specific code resides in only VOP_BMAP().

UBC introduces these new functions:

- **VOP_GETPAGES(), VOP_PUTPAGES()**

These new VOPs are provided by the filesystems to allow the VM system to request ranges of pages to be read into the memory from the disk or written from memory back to the disk. VOP_GETPAGES() must allocate pages from the VM system for data which is not already cached and then initiate device I/O operations to read all the disk blocks which contain the data for those pages.

Mechanisms→	IO system calls such as read(), write()	Memory mapping such as mmap()
FreeBSD	Buffer cache	Page cache
NetBSD	Page cache using UBC	Page cache

Table 2 The approaches of addressing DCP in FreeBSD and NetBSD

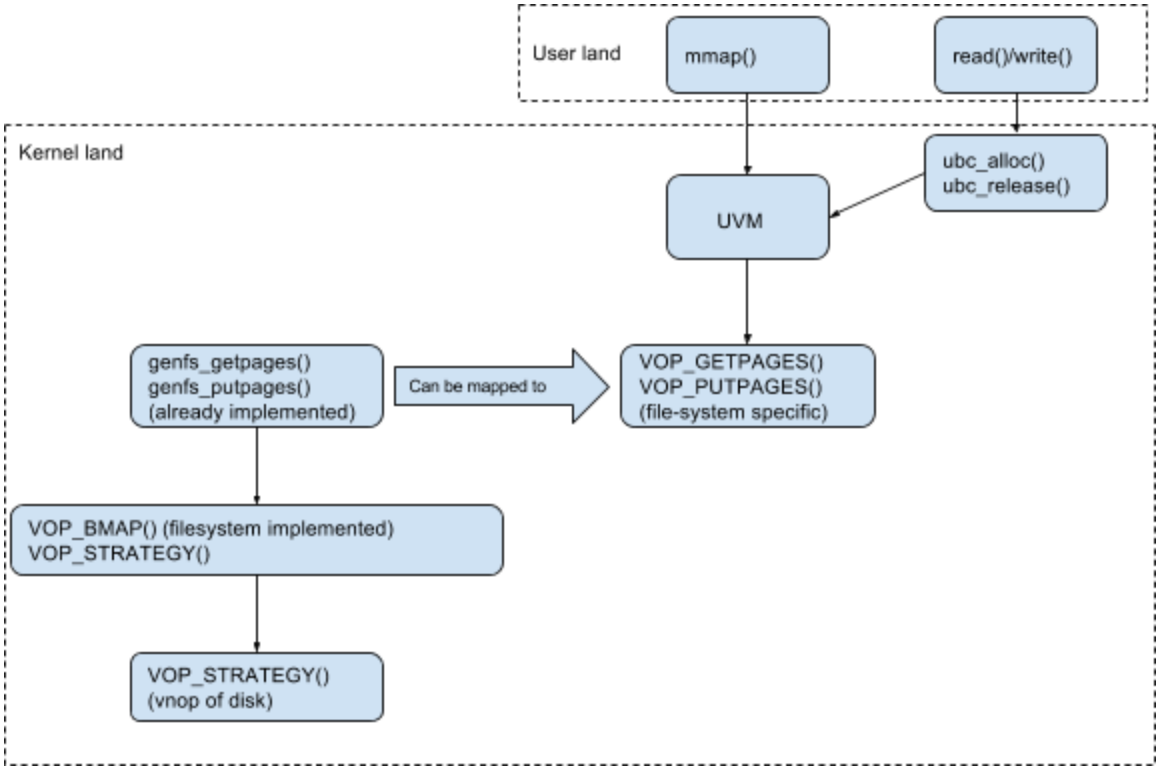


Figure 4 Trace of read() and mmap() in NetBSD

VOP_PUTPAGES() must initiate device I/Os to write dirty pages back to the disk.

- **ubc_alloc(), ubc_release(), ubc_pager**

These functions allocate and free temporary mappings of page cache file data. These are the page cache equivalents of the buffer cache functions getblk() and brelse(). ubc_pager is a UVM pager which handles page faults on the mappings created by ubc_alloc(). A UVM pager is an abstraction which embodies knowledge of page-fault resolution and other VM data management. The only action performed by ubc_pager is to call the new VOP_GETPAGES() operation to get pages as needed to resolve the faults.

- **VOP_BMAP**

This function is used by genfs_getpages() / genfs_putpages to get the file logical data block to physical data block mapping. Since logical block number to physical block number mapping purely depends on the filesystem itself, this function was left to be written by the filesystem developer.

3.4 Coding work

Previous sections describes, how file data blocks are read into VM pages and how UBC interfaces avail them to the read/write system calls. After having this description, it is clear where the extent mapping code needs to go in the filesystem. The implementation was done as described below :

1. Ext4 compatible feature flags was first updated to include extent support as a new feature. While mounting the filesystem, the kernel checks the feature flags that a filesystem-image (on disk) needs, and allows filesystem to mount the filesystem-image only if all the essential features have been implemented in the filesystem. The updation in the feature flags was to turn the kernel support on for Ext4 extents in read-only mount mode.
2. Writing Ext4 Extent data structures in a separate file viz “ext2fs_extents.h”, and the implementation of extents functions in a

new file ext2fs_extents.c was the next step. In ext2fs_extents.c, the main functions were ext2fs_ext_find_extent() - function to add caching, and few other utility functions. These structures and functions were imported from FreeBSD [3] as these required the OS independent extent logic to be transformed into code.

3. Next step was to implement file logical block no. to physical block no. map to access the files supporting extents. There was a function '***ext2fs_bmap'*** in ext2fs_bmap.c which used to invoke ext2fs_bmaparray() to get the mapping for the file having indirect pointers for file data index. There, ext2fs_bmapext() was added to get the mapping in case file has extent based index for file data blocks. Also a sanity check was added before invoking these functions based on the value of Ext4_EXTENTS bit of ***'di_flag'*** field in the file inode.

4 Ext3 HTree DIR Index

In Ext2fs, directory entries follow linked-list data structure in which each directory operation (create, open or delete) requires a linear search of an entire directory file. This results in a quadratically increasing cost of operating on all the files of a directory, as the number of files in the directory increases. HTree directory indexing extension is designed to address this issue. In addition, the HTree indexing method is backward and forward-compatible with existing Ext2 volumes. Let's begin with understanding the linked-list directory structures and then jump to Ext3 HTree DIR.

4.1 Linked List Directory

In this directory structure, a directory file is a linked list of directory entry structures. Each structure contains the name of the entry, the inode associated with the data of this entry, and the distance within the directory file to the next entry. The rec_len field in the directory entry stores the 16 bits unsigned displacement to the next directory from the start of the current directory entry.

Offset(bytes)	Size(bytes)	Fields
0	4	inode
4	2	rec_len
6	1	name_len
7	1	file_type
8	0-255	name

Table 3 Linked Directory Entry Structure

This field must have a value at least equal to the length of the current record. Since rec_len value cannot be negative, when a file is removed, the previous record within the block has to be modified to point to the next valid record within the block or to the end of the block when no other directory entry is present. If the first entry within the block is removed, a blank record will be created and pointed to the next directory entry or to the end of the block. The directory file is traversed linearly by getting the offset of the next entry which is calculated by rec_len + offset of the current entry [6].

4.2 HTree DIR Index

HTree (hashed-BTree) is the data structure that is used by Ext3/4 as directory layout. It uses hashes of the file names as keys of the HTree. There are basically two types of blocks in an HTree indexed directory:

1. **Directory Index Block** (DX-block): Stores hash-value and block-ID pairs:

   Hash-value: hash value of the entry name.

   Block-ID: File logical block number of leaf block, or the next level indices block.

2. **Directory Entries Block** (DE-block): stores directory entries (filenames).

The root of an HTree index is the first block of a directory file. The leaves of an HTree are normal Ext2 directory blocks, referenced by the root or indirectly through intermediate HTree index blocks.

References within the directory file are by means of logical block offsets within the file. An HTree uses hashes of names as keys, rather than the names themselves. Each hash key references not an individual directory entry, but a range of entries that are stored within a single leaf block. An HTree first level index block contains an array of index entries, each consisting of a hash keys and a logical pointer to the indexed block. Each hash key is the lower bound of all the hash values in a leaf block and the entries are arranged in ascending order of hash key. Both hash keys and logical pointers are 32-bit quantities. The lowest bit of a hash key is used to flag the possibility of a hash collision, leaving 31 bits available for the hash itself. The HTree root index block contains an array of index entries in the same format as a first level index block. The pointers refer to index blocks rather than leaf blocks and the hash keys give the lower bounds for the hash keys of the referenced index block. The HTree root index also contains a short header, providing information such as the depth of the HTree and information oriented towards checking the HTree index integrity. Figure 5 below illustrates an indexed directory stored as an htree [10].

ext2fs_fake_direct
e2d_ino /* inode number of entry */ e2d_reclen /* length of this record */ e2d_namlen /* length of string in d_name */ e2d_type; /* file type */

Table 4(a) Fake directory structure

ext2fs_htree_entry
h_hash h_blk

Table 4(b) Index node entry structure

ext2fs_htree_node
ext2fs_fake_direct h_fake_dirent ext2fs_htree_entry h_entries[0]

Table 4(c) Index node structure

ext2fs_htree_root
ext2fs_fake_direct h_dot h_dot_name[4] ext2fs_fake_direct h_dotdot h_dotdot_name[4] ext2fs_htree_root_info h_info ext2fs_htree_entry h_entries[0]

Table 4(d) htree root structure

Lookup() in HTree

lookup for a name-entry in the HTree is begun by reading the root, the first block of the directory file. Then a number of tests are performed against the header of the index root block in an attempt to eliminate any possibility of a corrupted index causing a program fault in the operating system kernel. Next the hash value of the target name is computed, and from that a decision is made for which leaf block to search. The desired leaf block is the one whose hash range includes the hash of the target entry name. Since index entries are of fixed size and maintained in sorted order, a binary search is used here. The format of an index entry is the same, whether it references a leaf block or an interior index node, so this step is simply repeated if the index tree has more than one level. As the hash probe descends through index entries, an access chain is created, until the target leaf block is read [7].

Once a target leaf block has been obtained, lookup proceeds exactly as without an index, i.e., by linearly searching through the entries in the block. If the target name is not found then there is still a possibility that the target could be found in the following leaf block due to a hash collision. In this case, the parent index is examined to see if the hash value of the successor leaf block hash has its low bit set, indicating that a hash collision does exist. If set, then the hash value of the target string is compared for equality to the successor block's hash value (minus the collision bit). If it is the same then the successor leaf block is read, the access chain updated, and the search is repeated.

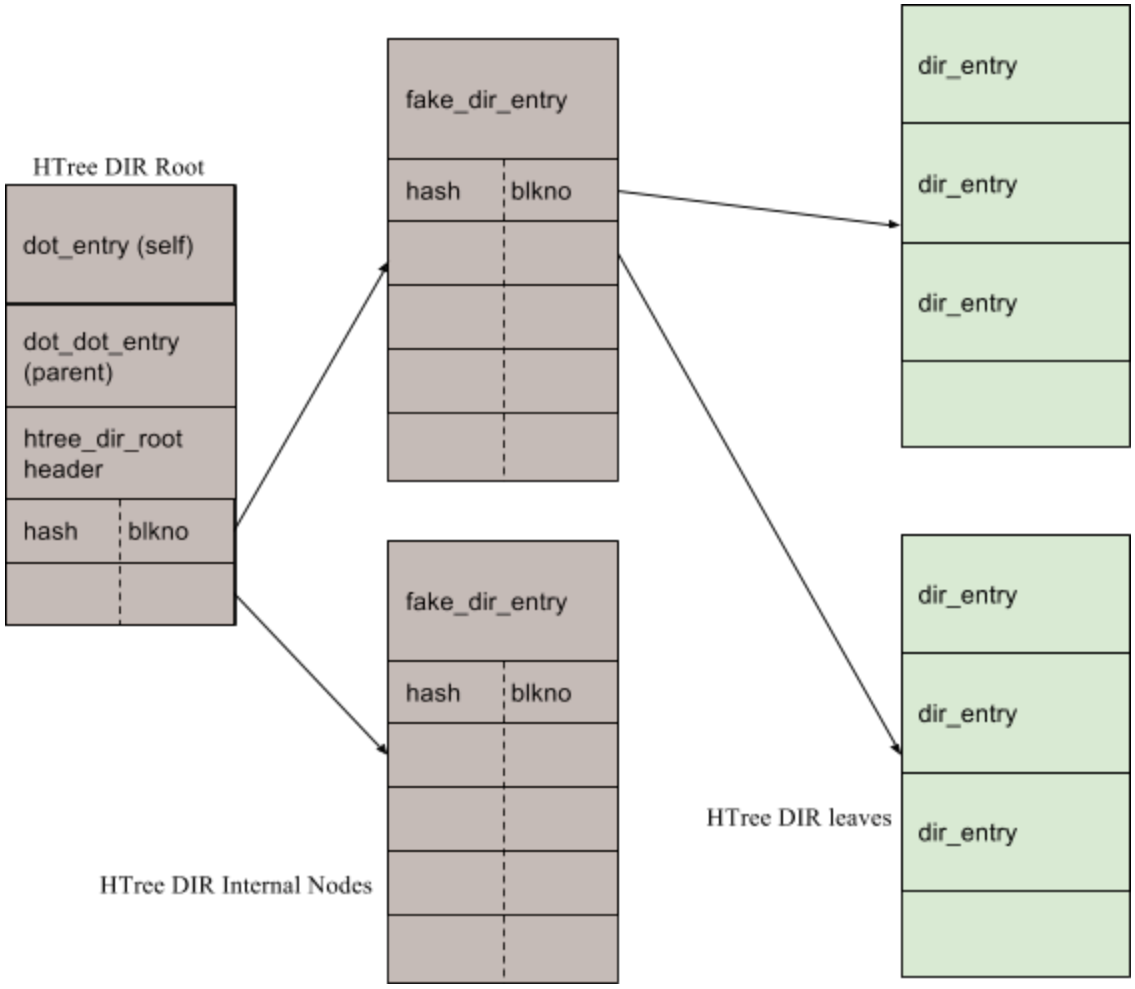


Figure 5 A typical HTree DIR Index

Implementation in NetBSD

The core logic of the HTree DIR index had already been implemented in other kernels like FreeBSD and Linux. Moreover the logic of the HTree traversals remains same irrespective of the underlying kernel except for the few differences in the code. FreeBSD and NetBSD had quite similarity in the directory lookup code, hence HTree DIR core implementations were directly imported from FreeBSD. To dig a bit into the HTree index directory code, let's split the code into following segments.

1. Add kernel support for HTree DIR

Ext4 compatible feature flags were first updated to include HTree DIR support as a new feature. While mounting the filesystem, the kernel checks the feature flags that a filesystem-image(on disk) needs, and allows filesystem to mount the filesystem-image only if all the essential features are implemented in the filesystem. The updation in the feature flags was to turn the kernel support on for Ext3 HTree DIR in read-only mount mode.

HTree index support at directory level is checked by reading ***Ext2_INDEX*** bit of ***'e2di_flags'*** field of the directory inode. If this bit is set then the first directory block is interpreted as the root of an HTree index.

2. HTree data structures and hash algorithms

All the HTree specific data-structures as summarized in table 4 were added in ext2fs_htree.h, and the hash specific structures were added in ext2fs_hash.h, whereas the hash algorithms were

implemented in ext2fs_hash.c. FreeBSD implementations of hash algorithms were directly imported into NetBSD.

3. HTree Directory lookup operations

The code for traversing HTree DIR during lookup operation when flag EXT3_INDEX in directory inode is set, was written in a new file ext2fs_htree.c. Most of the code was placed in the function *ext2fs_htree_lookup()*. The algorithms of *ext2fs_htree_lookup()* was straightforward but since there are different approaches of addressing directory lookup results in NetBSD and FreeBSD, it required few modifications to fit FreeBSD's code into NetBSD kernel. The algorithm steps of lookup operation is shown below:

1. Compute a hash of the name.
2. Read the index root.
3. Use binary search to find the first index or leaf block that could contain the target hash (in tree order).
4. Repeat the above until the lowest tree level is reached
5. Read the leaf directory entry block and do a normal Ext2 directory block search in it.
6. If the name is found, return its directory entry and buffer
7. Otherwise, if the collision bit of the next directory entry is set, continue searching in the successor block.

5 Tool Used

- **VND (VNode Disk Driver)**

  The **vnd** [1] driver in NetBSD, provides a disk-like interface to a file. I used this tool to give a disk-appearance to the file containing filesystem image, so that it can readily be mounted in NetBSD and can be double checked with other OS.

- **ATF**

  ATF (an automated testing framework [1]) provides the necessary means to easily create test suites composed of multiple test programs, which in turn are a collection of test cases. It also attempts to simplify the debugging of problems when these test cases detect an error by providing as much information as possible about the failure. I used ATF to test the new features implemented in this project.

- Apart from these I used, *Linux* OS as the host machine, *VirtualBox* to run NetBSD as the guest machine and *Vim-editor* an IDE for coding purposes. I used Git for version controlling and github for my code repository. I also used Opengrok tool[16] for keyword search in NetBSD source code.

6 Testing

NetBSD has a well organised Automated Testing Framework (ATF) using rump which separates the test cases from test programs that collects and exposes the group of tests cases. NetBSD kernel also has a great variety of test cases written for users to test their system's working. I used the same test framework to test the Ext2fs module using the existing test programs and tests cases. All existing tests-cases were passed but test cases specific to features implemented as a part of this project are still TODO for now.

7 Future work

Unit tests for testing the Ext4 specific functionality of Ext2fs module are remaining to be written. Secondly, there are few feature remaining to be implemented to achieve full Ext4 read support. Those are mainly FLEX_BG, DIR_NLINK, GDT_CKSUM. But these need significant time investment, probably as a separate GSoC project.

Coming to write support, which is even more challenging, needs additional extents write support and HTree directory index write support to be implemented.

8 Acknowledgement

This project has been developed as part of Google Summer of Code 2016, so thanks to Google for sponsoring it. Special thanks to my mentor Christos Zoulas for his valuable remarks. I would like to thank all the NetBSD community members who answered my queries on IRC and provided me useful suggestions.

9 Availability

The code for this project was hosted on Github and it is under active development. It can be obtained from https://github.com/hrishikeshgoyal/ext2fs. The code has also been included into NetBSD source and resides here http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/ufs/ext2fs/?only_with_tag=MAIN

References

[1] NetBSD manual page of vnd
http://netbsd.gw.com/cgi-bin/man-cgi?vnd+4+NetBSD-current

[2] NetBSD Guide especially for cross-compiling NetBSD with build.sh
https://www.netbsd.org/docs/guide/en/index.html

[3] FreeBSD codebase
https://github.com/freebsd/freebsd/tree/master/sys/fs/ext2fs

[4] Ext4 wikis
https://kernelnewbies.org/Ext4
https://ext4.wiki.kernel.org/index.php/Main_Page

[5] Google Summer of Code home page
https://summerofcode.withgoogle.com/archive/

[6] sourceforge for Directory indexing
http://ext2.sourceforge.net/2005-ols/paper-html/node3.html

[7] Daniel Phillips for Directory Index for Ext2
http://linuxshowcase.org/2001/full_papers/phillips/phillips_html/index.html

[8] Richard Henwood for HTree index and HTree path
https://wiki.hpdd.intel.com/

[9] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya
IBM Linux Technology Center

[10] Borislav Djordjevic, Valentina Timcenko Ext4 file system in Linux Environment: Features and Performance Analysis

[11] Avantika Mathur, Mingming Cao, Suparna Bhattacharya
IBM Linux Technology Center

[12] Freenix 2000 USENIX Annual Technical Conference Paper
https://www.usenix.org/legacy/event/usenix2000/freenix/full_papers/