WHAT IS Flash-Friendly File System (F2FS)?

NAND flash memory-based storage devices, such as SSD, eMMC, and SD cards, have been equipped on a variety systems ranging from mobile to server systems. Since they are known to have different characteristics from the conventional rotating disks, a file system, an upper layer to the storage device, should adapt to the changes from the sketch in the design level.

F2FS is a file system exploiting NAND flash memory-based storage devices, which is based on Log-structured File System (LFS). The design has been focused on addressing the fundamental issues in LFS, which are snowball effect of wandering tree and high cleaning overhead.

Since a NAND flash memory-based storage device shows different characteristic according to its internal geometry or flash memory management scheme, namely FTL, F2FS and its tools support various parameters not only for configuring on-disk layout, but also for selecting allocation and cleaning algorithms.

The following git tree provides the file system formatting tool (mkfs.f2fs), a consistency checking tool (fsck.f2fs), and a debugging tool (dump.f2fs).

- git://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs-tools.git

For reporting bugs and sending patches, please use the following mailing list:

- linux-f2fs-devel@lists.sourceforge.net

Background and Design issues

Log-structured File System (LFS)

"A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently. In order to maintain large free areas on disk for fast writing, we divide the log into segments and use a segment cleaner to compress the live information from heavily fragmented segments." from Rosenblum, M. and Ousterhout, J. K., 1992, "The design and implementation of a log-structured file system", ACM Trans. Computer Systems 10, 1, 26–52.

Wandering Tree Problem

In LFS, when a file data is updated and written to the end of log, its direct pointer block is updated due to the changed location. Then the indirect pointer block is also updated due to the direct pointer block update. In this manner, the upper index structures such as inode, inode map, and checkpoint block are also updated recursively. This problem is called as wandering tree problem [1], and in order to enhance the performance, it should eliminate or relax the update propagation as much as possible.

[1] Bityutskiy, A. 2005. JFFS3 design issues. http://www.linux-mtd.infradead.org/

Cleaning Overhead

Since LFS is based on out-of-place writes, it produces so many obsolete blocks scattered across the whole storage. In order to serve new empty log space, it needs to reclaim these obsolete blocks seamlessly to users. This job is called as a cleaning process.

The process consists of three operations as follows.

1. A victim segment is selected through referencing segment usage table.
2. It loads parent index structures of all the data in the victim identified by segment summary blocks.
3. It checks the cross-reference between the data and its parent index structure.
4. It moves valid data selectively.

This cleaning job may cause unexpected long delays, so the most important goal is to hide the latencies to users. And also definitely, it should reduce the amount of valid data to be moved, and move them quickly as well.

Key Features

Flash Awareness

- Enlarge the random write area for better performance, but provide the high spatial locality
- Align FS data structures to the operational units in FTL as best efforts

Wandering Tree Problem

- Use a term, “node”, that represents inodes as well as various pointer blocks
- Introduce Node Address Table (NAT) containing the locations of all the “node” blocks; this will cut off the update propagation.

Cleaning Overhead

- Support a background cleaning process
- Support greedy and cost-benefit algorithms for victim selection policies
- Support multi-head logs for static/dynamic hot and cold data separation
- Introduce adaptive logging for efficient block allocation

Mount Options

background_gc=%s	Turn on/off cleaning operations, namely garbage collection, triggered in background when I/O subsystem is idle. If background_gc=on, it will turn on the garbage collection and if background_gc=off, garbage collection will be turned off. If background_gc=sync, it will turn on synchronous garbage collection running in background. Default value for this option is on. So garbage collection is on by default.
gc_merge	When background_gc is on, this option can be enabled to let background GC thread to handle foreground GC requests, it can eliminate the sluggish issue caused by slow foreground GC operation when GC is triggered from a process with limited I/O and CPU resources.
nogc_merge	Disable GC merge feature.
disable_roll_forward	Disable the roll-forward recovery routine
norecovery	Disable the roll-forward recovery routine, mounted read- only (i.e., -o ro,disable_roll_forward)
discard/nodiscard	Enable/disable real-time discard in f2fs, if discard is enabled, f2fs will issue discard/TRIM commands when a segment is cleaned.
no_heap	Disable heap-style segment allocation which finds free segments for data from the beginning of main area, while for node from the end of main area.
nouser_xattr	Disable Extended User Attributes. Note: xattr is enabled by default if CONFIG_F2FS_FS_XATTR is selected.
noacl	Disable POSIX Access Control List. Note: acl is enabled by default if CONFIG_F2FS_FS_POSIX_ACL is selected.
active_logs=%u	Support configuring the number of active logs. In the current design, f2fs supports only 2, 4, and 6 logs. Default number is 6.
disable_ext_identify	Disable the extension list configured by mkfs, so f2fs is not aware of cold files such as media files.
inline_xattr	Enable the inline xattrs feature.
noinline_xattr	Disable the inline xattrs feature.
inline_xattr_size=%u	Support configuring inline xattr size, it depends on flexible inline xattr feature.
inline_data	Enable the inline data feature: Newly created small (<~3.4k) files can be written into inode block.
inline_dentry	Enable the inline dir feature: data in newly created directory entries can be written into inode block. The space of inode block which is used to store inline dentries is limited to ~3.4k.
noinline_dentry	Disable the inline dentry feature.
flush_merge	Merge concurrent cache_flush commands as much as possible to eliminate redundant command issues. If the underlying device handles the cache_flush command relatively slowly, recommend to enable this option.
nobarrier	This option can be used if underlying storage guarantees its cached data should be written to the novolatile area. If this option is set, no cache_flush commands are issued but f2fs still guarantees the write ordering of all the data writes.
fastboot	This option is used when a system wants to reduce mount time as much as possible, even though normal performance can be sacrificed.
extent_cache	Enable an extent cache based on rb-tree, it can cache as many as extent which map between contiguous logical address and physical address per inode, resulting in increasing the cache hit ratio. Set by default.
noextent_cache	Disable an extent cache based on rb-tree explicitly, see the above extent_cache mount option.
noinline_data	Disable the inline data feature, inline data feature is enabled by default.
data_flush	Enable data flushing before checkpoint in order to persist data of regular and symlink.
reserve_root=%d	Support configuring reserved space which is used for allocation from a privileged user with specified uid or gid, unit: 4KB, the default limit is 0.2% of user blocks.
resuid=%d	The user ID which may use the reserved blocks.
resgid=%d	The group ID which may use the reserved blocks.
fault_injection=%d	Enable fault injection in all supported types with specified injection rate.

<table>
<tr><td>fault_type=%d</td><td>Support configuring fault injection type, should be enabled with fault_injection option, fault type value is shown below, it supports single or combined type.<table><tr><th>Type_Name</th><th>Type_Value</th></tr><tr><td>FAULT_KMALLOC</td><td>0x000000001</td></tr><tr><td>FAULT_KVMALLOC</td><td>0x000000002</td></tr><tr><td>FAULT_PAGE_ALLOC</td><td>0x000000004</td></tr><tr><td>FAULT_PAGE_GET</td><td>0x000000008</td></tr><tr><td>FAULT_ALLOC_BIO</td><td>0x000000010 (obsolete)</td></tr><tr><td>FAULT_ALLOC_NID</td><td>0x000000020</td></tr><tr><td>FAULT_ORPHAN</td><td>0x000000040</td></tr><tr><td>FAULT_BLOCK</td><td>0x000000080</td></tr><tr><td>FAULT_DIR_DEPTH</td><td>0x000000100</td></tr><tr><td>FAULT_EVICT_INODE</td><td>0x000000200</td></tr><tr><td>FAULT_TRUNCATE</td><td>0x000000400</td></tr><tr><td>FAULT_READ_IO</td><td>0x000000800</td></tr><tr><td>FAULT_CHECKPOINT</td><td>0x000001000</td></tr><tr><td>FAULT_DISCARD</td><td>0x000002000</td></tr><tr><td>FAULT_WRITE_IO</td><td>0x000004000</td></tr><tr><td>FAULT_SLAB_ALLOC</td><td>0x000008000</td></tr><tr><td>FAULT_DQUOT_INIT</td><td>0x000010000</td></tr><tr><td>FAULT_LOCK_OP</td><td>0x000020000</td></tr></table></td></tr>
<tr><td>mode=%s</td><td>Control block allocation mode which supports "adaptive" and "lfs". In "lfs" mode, there should be no random writes towards main area. "fragment:segment" and "fragment:block" are newly added here. These are developer options for experiments to simulate filesystem fragmentation/after-GC situation itself. The developers use these modes to understand filesystem fragmentation/after-GC condition well, and eventually get some insights to handle them better. In "fragment:segment", f2fs allocates a new segment in ramdom position. With this, we can simulate the after-GC condition. In "fragment:block", we can scatter block allocation with "max_fragment_chunk" and "max_fragment_hole" sysfs nodes. We added some randomness to both chunk and hole size to make it close to realistic IO pattern. So, in this mode, f2fs will allocate 1..<max_fragment_chunk> blocks in a chunk and make a hole in the length of 1..<max_fragment_hole> by turns. With this, the newly allocated blocks will be scattered throughout the whole partition. Note that "fragment:block" implicitly enables "fragment:segment" option for more randomness. Please, use these options for your experiments and we strongly recommend to re-format the filesystem after using these options.</td></tr>
<tr><td>io_bits=%u</td><td>Set the bit size of write IO requests. It should be set with "mode=lfs".</td></tr>
<tr><td>usrquota</td><td>Enable plain user disk quota accounting.</td></tr>
<tr><td>grpquota</td><td>Enable plain group disk quota accounting.</td></tr>
<tr><td>prjquota</td><td>Enable plain project quota accounting.</td></tr>
<tr><td>usrjquota=<file></td><td>Appoint specified file and type during mount, so that quota</td></tr>
<tr><td>grpjquota=<file></td><td>information can be properly updated during recovery flow,</td></tr>
<tr><td>prjjquota=<file></td><td><quota file>: must be in root directory;</td></tr>
<tr><td>jqfmt=<quota type></td><td><quota type>: [vfsold,vfsv0,vfsv1].</td></tr>
<tr><td>offusrjquota</td><td>Turn off user journalled quota.</td></tr>
<tr><td>offgrpjquota</td><td>Turn off group journalled quota.</td></tr>
<tr><td>offprjjquota</td><td>Turn off project journalled quota.</td></tr>
<tr><td>quota</td><td>Enable plain user disk quota accounting.</td></tr>
<tr><td>noquota</td><td>Disable all plain disk quota option.</td></tr>
<tr><td>whint_mode=%s</td><td>Control which write hints are passed down to block layer. This supports "off", "user-based", and "fs-based". In "off" mode (default), f2fs does not pass down hints. In "user-based" mode, f2fs tries to pass down hints given by users. And in "fs-based" mode, f2fs passes down hints with its policy.</td></tr>
<tr><td>alloc_mode=%s</td><td>Adjust block allocation policy, which supports "reuse" and "default".</td></tr>
<tr><td>fsync_mode=%s</td><td>Control the policy of fsync. Currently supports "posix", "strict", and "nobarrier". In "posix" mode, which is default, fsync will follow POSIX semantics and does a light operation to improve the filesystem performance. In "strict" mode, fsync will be heavy and behaves in line with xfs, ext4 and btrfs, where xfstest generic/342 will pass, but the performance will regress. "nobarrier" is based on "posix", but doesn't issue flush command for non-atomic files likewise "nobarrier" mount option.</td></tr>
<tr><td>test_dummy_encryption</td><td></td></tr>
<tr><td>test_dummy_encryption=%s</td><td>Enable dummy encryption, which provides a fake fscrypt context. The fake fscrypt context is used by xfstests. The argument may be either "v1" or "v2", in order to select the corresponding fscrypt policy version.</td></tr>
</table>

checkpoint=%s[:%u[%]]	Set to "disable" to turn off checkpointing. Set to "enable" to reenable checkpointing. Is enabled by default. While disabled, any unmounting or unexpected shutdowns will cause the filesystem contents to appear as they did when the filesystem was mounted with that option. While mounting with checkpoint=disabled, the filesystem must run garbage collection to ensure that all available space can be used. If this takes too much time, the mount may return EAGAIN. You may optionally add a value to indicate how much of the disk you would be willing to temporarily give up to avoid additional garbage collection. This can be given as a number of blocks, or as a percent. For instance, mounting with checkpoint=disable:100% would always succeed, but it may hide up to all remaining free space. The actual space that would be unusable can be viewed at /sys/fs/f2fs/<disk>/unusable This space is reclaimed once checkpoint=enable.
checkpoint_merge	When checkpoint is enabled, this can be used to create a kernel daemon and make it to merge concurrent checkpoint requests as much as possible to eliminate redundant checkpoint issues. Plus, we can eliminate the sluggish issue caused by slow checkpoint operation when the checkpoint is done in a process context in a cgroup having low i/o budget and cpu shares. To make this do better, we set the default i/o priority of the kernel daemon to "3", to give one higher priority than other kernel threads. This is the same way to give a I/O priority to the jbd2 journaling thread of ext4 filesystem.
nocheckpoint_merge	Disable checkpoint merge feature.
compress_algorithm=%s	Control compress algorithm, currently f2fs supports "lzo", "lz4", "zstd" and "lzo-rle" algorithm.
compress_algorithm=%s:%d	Control compress algorithm and its compress level, now, only "lz4" and "zstd" support compress level config. algorithm level range lz4 3 - 16 zstd 1 - 22
compress_log_size=%u	Support configuring compress cluster size, the size will be 4KB * (1 << %u), 16KB is minimum size, also it's default size.
compress_extension=%s	Support adding specified extension, so that f2fs can enable compression on those corresponding files, e.g. if all files with '.ext' has high compression rate, we can set the '.ext' on compression extension list and enable compression on these file by default rather than to enable it via ioctl. For other files, we can still enable compression via ioctl. Note that, there is one reserved special extension '*', it can be set to enable compression for all files.
nocompress_extension=%s	Support adding specified extension, so that f2fs can disable compression on those corresponding files, just contrary to compression extension. If you know exactly which files cannot be compressed, you can use this. The same extension name can't appear in both compress and nocompress extension at the same time. If the compress extension specifies all files, the types specified by the nocompress extension will be treated as special cases and will not be compressed. Don't allow use '*' to specifie all file in nocompress extension. After add nocompress_extension, the priority should be: dir_flag < comp_extention,nocompress_extension < comp_file_flag,no_comp_file_flag. See more in compression sections.
compress_chksum	Support verifying chksum of raw data in compressed cluster.
compress_mode=%s	Control file compression mode. This supports "fs" and "user" modes. In "fs" mode (default), f2fs does automatic compression on the compression enabled files. In "user" mode, f2fs disables the automaic compression and gives the user discretion of choosing the target file and the timing. The user can do manual compression/decompression on the compression enabled files using ioctls.
compress_cache	Support to use address space of a filesystem managed inode to cache compressed block, in order to improve cache hit ratio of random read.
inlinecrypt	When possible, encrypt/decrypt the contents of encrypted files using the blk-crypto framework rather than filesystem-layer encryption. This allows the use of inline encryption hardware. The on-disk format is unaffected. For more details, see Documentation/block/inline-encryption.rst.
atgc	Enable age-threshold garbage collection, it provides high effectiveness and efficiency on background GC.
discard_unit=%s	Control discard unit, the argument can be "block", "segment" and "section", issued discard command's offset/size will be aligned to the unit, by default, "discard_unit=block" is set, so that small discard functionality is enabled. For blkzoned device, "discard_unit=section" will be set by default, it is helpful for large sized SMR or ZNS devices to reduce memory cost by getting rid of fs metadata supports small discard.

Debugfs Entries

/sys/kernel/debug/f2fs/ contains information about all the partitions mounted as f2fs. Each file shows the whole f2fs information.

/sys/kernel/debug/f2fs/status includes:

- major file system information managed by f2fs currently
- average SIT information about whole segments
- current memory footprint consumed by f2fs.

Sysfs Entries

Information about mounted f2fs file systems can be found in /sys/fs/f2fs. Each mounted filesystem will have a directory in /sys/fs/f2fs based on its device name (i.e., /sys/fs/f2fs/sda). The files in each per-device directory are shown in table below.

Files in /sys/fs/f2fs/<devname> (see also Documentation/ABI/testing/sysfs-fs-f2fs)

Usage

1. Download userland tools and compile them.
2. Skip, if f2fs was compiled statically inside kernel. Otherwise, insert the f2fs.ko module:

```
# insmod f2fs.ko
```

3. Create a directory to use when mounting:

```
# mkdir /mnt/f2fs
```

4. Format the block device, and then mount as f2fs:

```
# mkfs.f2fs -l label /dev/block_device
# mount -t f2fs /dev/block_device /mnt/f2fs
```

mkfs.f2fs

The mkfs.f2fs is for the use of formatting a partition as the f2fs filesystem, which builds a basic on-disk layout.

The quick options consist of:

`-l [label]`	Give a volume label, up to 512 unicode name.
`-a [0 or 1]`	Split start location of each area for heap-based allocation. 1 is set by default, which performs this.
`-o [int]`	Set overprovision ratio in percent over volume size. 5 is set by default.
`-s [int]`	Set the number of segments per section. 1 is set by default.
`-z [int]`	Set the number of sections per zone. 1 is set by default.
`-e [str]`	Set basic extension list. e.g. "mp3,gif,mov"
`-t [0 or 1]`	Disable discard command or not. 1 is set by default, which conducts discard.

Note: please refer to the manpage of mkfs.f2fs(8) to get full option list.

fsck.f2fs

The fsck.f2fs is a tool to check the consistency of an f2fs-formatted partition, which examines whether the filesystem metadata and user-made data are cross-referenced correctly or not. Note that, initial version of the tool does not fix any inconsistency.

The quick options consist of:

```
-d debug level [default:0]
```

Note: please refer to the manpage of fsck.f2fs(8) to get full option list.

dump.f2fs

The dump.f2fs shows the information of specific inode and dumps SSA and SIT to file. Each file is dump_ssa and dump_sit.

The dump.f2fs is used to debug on-disk data structures of the f2fs filesystem. It shows on-disk inode information recognized by a given inode number, and is able to dump all the SSA and SIT entries into predefined files, ./dump_ssa and ./dump_sit respectively.

The options consist of:

```
-d debug level [default:0]
-i inode no (hex)
-s [SIT dump segno from #1~#2 (decimal), for all 0~-1]
-a [SSA dump segno from #1~#2 (decimal), for all 0~-1]
```

Examples:

```
# dump.f2fs -i [ino] /dev/sdx
# dump.f2fs -s 0~-1 /dev/sdx (SIT dump)
# dump.f2fs -a 0~-1 /dev/sdx (SSA dump)
```

Note: please refer to the manpage of dump.f2fs(8) to get full option list.

sload.f2fs

The sload.f2fs gives a way to insert files and directories in the exisiting disk image. This tool is useful when building f2fs images given compiled files.

Note: please refer to the manpage of sload.f2fs(8) to get full option list.

resize.f2fs

The resize.f2fs lets a user resize the f2fs-formatted disk image, while preserving all the files and directories stored in the image.

Note: please refer to the manpage of resize.f2fs(8) to get full option list.

defrag.f2fs

The defrag.f2fs can be used to defragment scattered written data as well as filesystem metadata across the disk. This can improve the write speed by giving more free consecutive space.

Note: please refer to the manpage of defrag.f2fs(8) to get full option list.

f2fs_io

The f2fs_io is a simple tool to issue various filesystem APIs as well as f2fs-specific ones, which is very useful for QA tests.

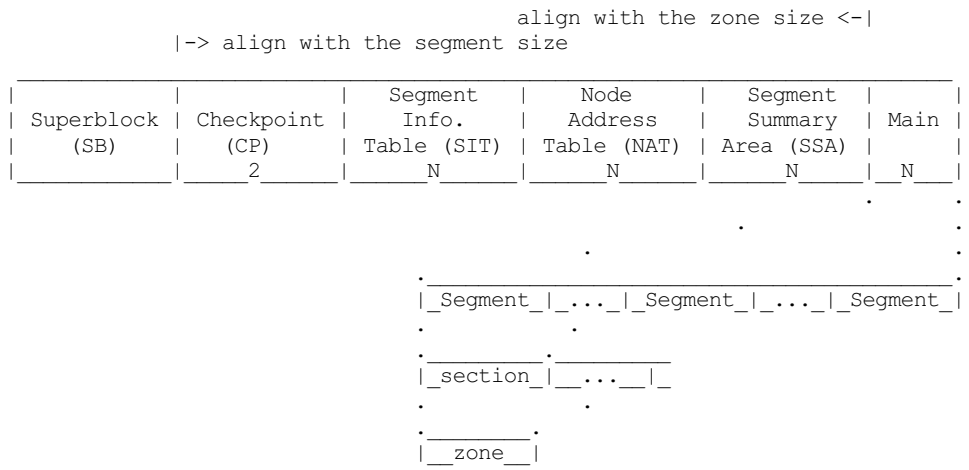
Note: please refer to the manpage of f2fs_io(8) to get full option list.

Design

On-disk Layout

F2FS divides the whole volume into a number of segments, each of which is fixed to 2MB in size. A section is composed of consecutive segments, and a zone consists of a set of sections. By default, section and zone sizes are set to one segment size identically, but users can easily modify the sizes by mkfs.

F2FS splits the entire volume into six areas, and all the areas except superblock consist of multiple segments as described below:



- Superblock (SB)
  It is located at the beginning of the partition, and there exist two copies to avoid file system crash. It contains basic partition information and some default parameters of f2fs.
- Checkpoint (CP)
  It contains file system information, bitmaps for valid NAT/SIT sets, orphan inode lists, and summary entries of current active segments.
- Segment Information Table (SIT)
  It contains segment information such as valid block count and bitmap for the validity of all the blocks.
- Node Address Table (NAT)
  It is composed of a block address table for all the node blocks stored in Main area.
- Segment Summary Area (SSA)
  It contains summary entries which contains the owner information of all the data and node blocks stored in Main area.
- Main Area
  It contains file and directory data including their indices.

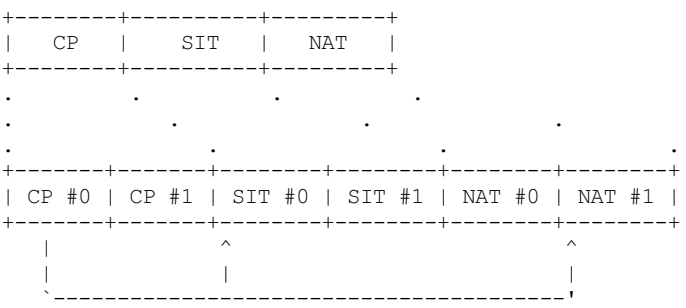
In order to avoid misalignment between file system and flash-based storage, F2FS aligns the start block address of CP with the segment size. Also, it aligns the start block address of Main area with the zone size by reserving some segments in SSA area.

Reference the following survey for additional technical details.
https://wiki.linaro.org/WorkingGroups/Kernel/Projects/FlashCardSurvey

File System Metadata Structure

F2FS adopts the checkpointing scheme to maintain file system consistency. At mount time, F2FS first tries to find the last valid checkpoint data by scanning CP area. In order to reduce the scanning time, F2FS uses only two copies of CP. One of them always indicates the last valid data, which is called as shadow copy mechanism. In addition to CP, NAT and SIT also adopt the shadow copy mechanism.

For file system consistency, each CP points to which NAT and SIT copies are valid, as shown as below:



Index Structure

The key data structure to manage the data locations is a "node". Similar to traditional file structures, F2FS has three types of node: inode, direct node, indirect node. F2FS assigns 4KB to an inode block which contains 923 data block indices, two direct node pointers, two indirect node pointers, and one double indirect node pointer as described below. One direct node block contains 1018 data blocks, and one indirect node block contains also 1018 node blocks. Thus, one inode block (i.e., a file) covers:

```
4KB * (923 + 2 * 1018 + 2 * 1018 * 1018 + 1018 * 1018 * 1018) := 3.94TB.

 Inode block (4KB)
   |- data (923)
   |- direct node (2)
   |          `- data (1018)
   |- indirect node (2)
   |            `- direct node (1018)
   |                        `- data (1018)
   `- double indirect node (1)
                       `- indirect node (1018)
                                    `- direct node (1018)
                                               `- data (1018)
```

Note that all the node blocks are mapped by NAT which means the location of each node is translated by the NAT table. In the consideration of the wandering tree problem, F2FS is able to cut off the propagation of node updates caused by leaf data writes.

Directory Structure

A directory entry occupies 11 bytes, which consists of the following attributes.

- hash hash value of the file name
- ino inode number
- len the length of file name
- type file type such as directory, symlink, etc

A dentry block consists of 214 dentry slots and file names. Therein a bitmap is used to represent whether each dentry is valid or not. A dentry block occupies 4KB with the following composition.

```
Dentry Block(4 K) = bitmap (27 bytes) + reserved (3 bytes) +
                    dentries(11 * 214 bytes) + file name (8 * 214 bytes)

                        [Bucket]
             +--------------------------------+
             |dentry block 1 | dentry block 2 |
             +--------------------------------+
             .               .
         .                               .
.       [Dentry Block Structure: 4KB]       .
+--------+----------+----------+------------+
| bitmap | reserved | dentries | file names |
+--------+----------+----------+------------+
[Dentry Block: 4KB] .   .
                .               .
            .                       .
            +------+------+-----+------+
```

```
            | hash | ino  | len | type |
            +------+------+-----+------+
            [Dentry Structure: 11 bytes]
```

F2FS implements multi-level hash tables for directory structure. Each level has a hash table with dedicated number of hash buckets as shown below. Note that "A(2B)" means a bucket includes 2 data blocks.

```
    ----------------------
    A : bucket
    B : block
    N : MAX_DIR_HASH_DEPTH
    ----------------------

    level #0   | A(2B)
               |
    level #1   | A(2B) - A(2B)
               |
    level #2   | A(2B) - A(2B) - A(2B) - A(2B)
        .      |   .       .       .       .
    level #N/2 | A(2B) - A(2B) - A(2B) - A(2B) - A(2B) - ... - A(2B)
        .      |   .       .       .       .
    level #N   | A(4B) - A(4B) - A(4B) - A(4B) - A(4B) - ... - A(4B)
```

The number of blocks and buckets are determined by:

```
                              ,- 2, if n < MAX_DIR_HASH_DEPTH / 2,
    # of blocks in level #n = |
                              `- 4, Otherwise

                               ,- 2^(n + dir_level),
                               |        if n + dir_level < MAX_DIR_HASH_DEPTH / 2,
    # of buckets in level #n = |
                               `- 2^((MAX_DIR_HASH_DEPTH / 2) - 1),
                                        Otherwise
```

When F2FS finds a file name in a directory, at first a hash value of the file name is calculated. Then, F2FS scans the hash table in level #0 to find the dentry consisting of the file name and its inode number. If not found, F2FS scans the next hash table in level #1. In this way, F2FS scans hash tables in each levels incrementally from 1 to N. In each level F2FS needs to scan only one bucket determined by the following equation, which shows O(log(# of files)) complexity:

```
    bucket number to scan in level #n = (hash value) % (# of buckets in level #n)
```

In the case of file creation, F2FS finds empty consecutive slots that cover the file name. F2FS searches the empty slots in the hash tables of whole levels from 1 to N in the same way as the lookup operation.

The following figure shows an example of two cases holding children:

```
       --------------> Dir <--------------
       |                                  |
     child                              child

     child - child                      [hole] - child

     child - child - child              [hole] - [hole] - child

    Case 1:                            Case 2:
    Number of children = 6,            Number of children = 3,
    File size = 7                      File size = 7
```

Default Block Allocation

At runtime, F2FS manages six active logs inside "Main" area: Hot/Warm/Cold node and Hot/Warm/Cold data.

- Hot node contains direct node blocks of directories.
- Warm node contains direct node blocks except hot node blocks.
- Cold node contains indirect node blocks
- Hot data contains dentry blocks
- Warm data contains data blocks except hot and cold data blocks
- Cold data contains multimedia data or migrated data blocks

LFS has two schemes for free space management: threaded log and copy-and-compac- tion. The copy-and-compaction scheme which is known as cleaning, is well-suited for devices showing very good sequential write performance, since free segments are served all the time for writing new data. However, it suffers from cleaning overhead under high utilization. Contrarily, the threaded log scheme suffers from random writes, but no cleaning process is needed. F2FS adopts a hybrid scheme where the copy-and-compaction scheme is adopted by default, but the policy is dynamically changed to the threaded log scheme according to the file system status.

In order to align F2FS with underlying flash-based storage, F2FS allocates a segment in a unit of section. F2FS expects that the

section size would be the same as the unit size of garbage collection in FTL. Furthermore, with respect to the mapping granularity in FTL, F2FS allocates each section of the active logs from different zones as much as possible, since FTL can write the data in the active logs into one allocation unit according to its mapping granularity.

Cleaning process

F2FS does cleaning both on demand and in the background. On-demand cleaning is triggered when there are not enough free segments to serve VFS calls. Background cleaner is operated by a kernel thread, and triggers the cleaning job when the system is idle.

F2FS supports two victim selection policies: greedy and cost-benefit algorithms. In the greedy algorithm, F2FS selects a victim segment having the smallest number of valid blocks. In the cost-benefit algorithm, F2FS selects a victim segment according to the segment age and the number of valid blocks in order to address log block thrashing problem in the greedy algorithm. F2FS adopts the greedy algorithm for on-demand cleaner, while background cleaner adopts cost-benefit algorithm.

In order to identify whether the data in the victim segment are valid or not, F2FS manages a bitmap. Each bit represents the validity of a block, and the bitmap is composed of a bit stream covering whole blocks in main area.

Write-hint Policy

1. whint_mode=off. F2FS only passes down WRITE_LIFE_NOT_SET.

2) whint_mode=user-based. F2FS tries to pass down hints given by users.

User	F2FS	Block
N/A	META	WRITE_LIFE_NOT_SET
N/A	HOT_NODE	"
N/A	WARM_NODE	"
N/A	COLD_NODE	"
ioctl(COLD)	COLD_DATA	WRITE_LIFE_EXTREME
extension list	"	"
-- buffered io		
WRITE_LIFE_EXTREME	COLD_DATA	WRITE_LIFE_EXTREME
WRITE_LIFE_SHORT	HOT_DATA	WRITE_LIFE_SHORT
WRITE_LIFE_NOT_SET	WARM_DATA	WRITE_LIFE_NOT_SET
WRITE_LIFE_NONE	"	"
WRITE_LIFE_MEDIUM	"	"
WRITE_LIFE_LONG	"	"
-- direct io		
WRITE_LIFE_EXTREME	COLD_DATA	WRITE_LIFE_EXTREME
WRITE_LIFE_SHORT	HOT_DATA	WRITE_LIFE_SHORT
WRITE_LIFE_NOT_SET	WARM_DATA	WRITE_LIFE_NOT_SET
WRITE_LIFE_NONE	"	WRITE_LIFE_NONE
WRITE_LIFE_MEDIUM	"	WRITE_LIFE_MEDIUM
WRITE_LIFE_LONG	"	WRITE_LIFE_LONG

3. whint_mode=fs-based. F2FS passes down hints with its policy.

User	F2FS	Block
N/A	META	WRITE_LIFE_MEDIUM;
N/A	HOT_NODE	WRITE_LIFE_NOT_SET
N/A	WARM_NODE	"
N/A	COLD_NODE	WRITE_LIFE_NONE
ioctl(COLD)	COLD_DATA	WRITE_LIFE_EXTREME
extension list	"	"
-- buffered io		
WRITE_LIFE_EXTREME	COLD_DATA	WRITE_LIFE_EXTREME
WRITE_LIFE_SHORT	HOT_DATA	WRITE_LIFE_SHORT
WRITE_LIFE_NOT_SET	WARM_DATA	WRITE_LIFE_LONG
WRITE_LIFE_NONE	"	"
WRITE_LIFE_MEDIUM	"	"
WRITE_LIFE_LONG	"	"
-- direct io		
WRITE_LIFE_EXTREME	COLD_DATA	WRITE_LIFE_EXTREME
WRITE_LIFE_SHORT	HOT_DATA	WRITE_LIFE_SHORT
WRITE_LIFE_NOT_SET	WARM_DATA	WRITE_LIFE_NOT_SET
WRITE_LIFE_NONE	"	WRITE_LIFE_NONE

User	F2FS	Block
WRITE_LIFE_MEDIUM	"	WRITE_LIFE_MEDIUM
WRITE_LIFE_LONG	"	WRITE_LIFE_LONG

Fallocate(2) Policy

The default policy follows the below POSIX rule.

Allocating disk space

> The default operation (i.e., mode is zero) of fallocate() allocates the disk space within the range specified by offset and len. The file size (as reported by stat(2)) will be changed if offset+len is greater than the file size. Any subregion within the range specified by offset and len that did not contain data before the call will be initialized to zero. This default behavior closely resembles the behavior of the posix_fallocate(3) library function, and is intended as a method of optimally implementing that function.

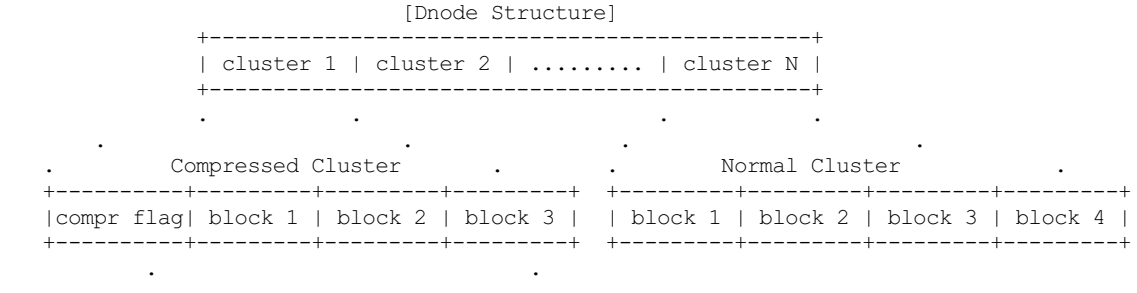
However, once F2FS receives ioctl(fd, F2FS_IOC_SET_PIN_FILE) in prior to fallocate(fd, DEFAULT_MODE), it allocates on-disk block addressess having zero or random data, which is useful to the below scenario where:

1. create(fd)
2. ioctl(fd, F2FS_IOC_SET_PIN_FILE)
3. fallocate(fd, 0, 0, size)
4. address = fibmap(fd, offset)
5. open(blkdev)
6. write(blkdev, address)

Compression implementation

- New term named cluster is defined as basic unit of compression, file can be divided into multiple clusters logically. One cluster includes 4 << n (n >= 0) logical pages, compression size is also cluster size, each of cluster can be compressed or not.
- In cluster metadata layout, one special block address is used to indicate a cluster is a compressed one or normal one; for compressed cluster, following metadata maps cluster to [1, 4 << n - 1] physical blocks, in where f2fs stores data including compress header and compressed data.
- In order to eliminate write amplification during overwrite, F2FS only support compression on write-once file, data can be compressed only when all logical blocks in cluster contain valid data and compress ratio of cluster data is lower than specified threshold.
- To enable compression on regular inode, there are four ways:
  - chattr +c file
  - chattr +c dir; touch dir/file
  - mount w/ -o compress_extension=ext; touch file.ext
  - mount w/ -o compress_extension=*; touch any_file
- To disable compression on regular inode, there are two ways:
  - chattr -c file
  - mount w/ -o nocompress_extension=ext; touch file.ext
- Priority in between FS_COMPR_FL, FS_NOCOMP_FS, extensions:
  - compress_extension=so; nocompress_extension=zip; chattr +c dir; touch dir/foo.so; touch dir/bar.zip; touch dir/baz.txt; then foo.so and baz.txt should be compresse, bar.zip should be non-compressed. chattr +c dir/bar.zip can enable compress on bar.zip.
  - compress_extension=so; nocompress_extension=zip; chattr -c dir; touch dir/foo.so; touch dir/bar.zip; touch dir/baz.txt; then foo.so should be compresse, bar.zip and baz.txt should be non-compressed. chattr+c dir/bar.zip; chattr+c dir/baz.txt; can enable compress on bar.zip and baz.txt.
- At this point, compression feature doesn't expose compressed space to user directly in order to guarantee potential data updates later to the space. Instead, the main goal is to reduce data writes to flash disk as much as possible, resulting in extending disk life time as well as relaxing IO congestion. Alternatively, we've added ioctl(F2FS_IOC_RELEASE_COMPRESS_BLOCKS) interface to reclaim compressed space and show it to user after putting the immutable bit. Immutable bit, after release, it doesn't allow writing/mmaping on the file, until reserving compressed space via ioctl(F2FS_IOC_RESERVE_COMPRESS_BLOCKS) or truncating filesize to zero.

Compress metadata layout:



```
    .                                       .
.                                                       .
+-------------+-------------+----------+----------------------------+
| data length | data chksum | reserved |      compressed data       |
+-------------+-------------+----------+----------------------------+
```

Compression mode

f2fs supports "fs" and "user" compression modes with "compression_mode" mount option. With this option, f2fs provides a choice to select the way how to compress the compression enabled files (refer to "Compression implementation" section for how to enable compression on a regular inode).

1) compress_mode=fs This is the default option. f2fs does automatic compression in the writeback of the compression enabled files.

2) compress_mode=user This disables the automatic compression and gives the user discretion of choosing the target file and the timing. The user can do manual compression/decompression on the compression enabled files using F2FS_IOC_DECOMPRESS_FILE and F2FS_IOC_COMPRESS_FILE ioctls like the below.

To decompress a file,

fd = open(filename, O_WRONLY, 0); ret = ioctl(fd, F2FS_IOC_DECOMPRESS_FILE);

To compress a file,

fd = open(filename, O_WRONLY, 0); ret = ioctl(fd, F2FS_IOC_COMPRESS_FILE);

NVMe Zoned Namespace devices

- ZNS defines a per-zone capacity which can be equal or less than the zone-size. Zone-capacity is the number of usable blocks in the zone. F2FS checks if zone-capacity is less than zone-size, if it is, then any segment which starts after the zone-capacity is marked as not-free in the free segment bitmap at initial mount time. These segments are marked as permanently used so they are not allocated for writes and consequently are not needed to be garbage collected. In case the zone-capacity is not aligned to default segment size(2MB), then a segment can start before the zone-capacity and span across zone-capacity boundary. Such spanning segments are also considered as usable segments. All blocks past the zone-capacity are considered unusable in these segments.