NILFS2

Design Document

Document Author: Vyacheslav Dubeyko (slava@dubeyko.com)

Revision History

Version	Date	Change Contents	Author
1.0	May 01, 2012	Create initial version	Vyacheslav Dubeyko
1.1	May 19, 2012	On-disk layout paragraph (draft) was elaborated	Vyacheslav Dubeyko
1.2	June 23, 2012	Raw volume dumps were excluded from tables. <TODO: Elaborate paragraphs with raw dump examples>	Vyacheslav Dubeyko
1.3	July 15, 2012	File system driver's data abstractions paragraph (draft) was elaborated	Vyacheslav Dubeyko
1.4	July 16, 2012	File system driver's subsystem APIs paragraph (draft) was elaborated	Vyacheslav Dubeyko
1.5	August 25, 2012	Persistent object allocator/deallocator subsystem (structures, API) was described	Vyacheslav Dubeyko
1.6	September 2, 2012	NILFS metadata file functionality (structures, API) was described	Vyacheslav Dubeyko
1.7	October 7, 2012	NILFS block mapping functionality (structures, API) was described	Vyacheslav Dubeyko

1. Overview

1.1. Document objectives

<TODO>

2. What is NILFS?

2.1. Brief summary

<TODO>

2.2. Goals

<TODO>

2.3. Features

<TODO>

2.4. Architecture

<TODO>

3. Core Concepts

<TODO>

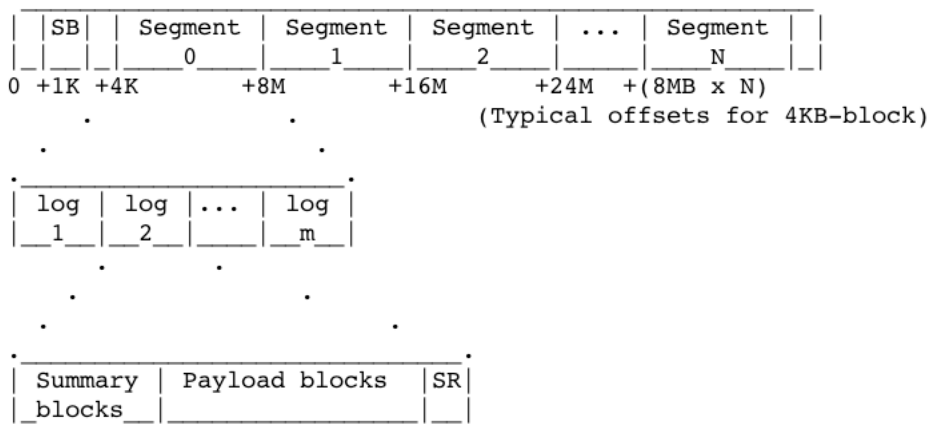
4. On-Disk Layout

4.1. Modules List

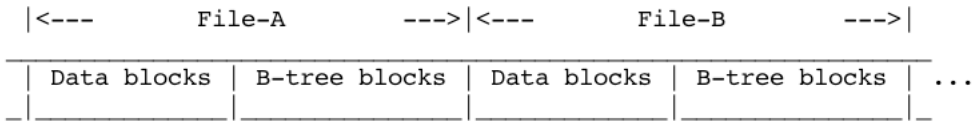
FILE	DESCRIPTION	AUTHOR
nilfs2_fs.h	NILFS2 on-disk structures and common declarations.	Koji Sato, Ryusuke Konishi
• Ryusuke Konishi <konishi.ryusuke@lab.ntt.co.jp> • Koji Sato <koji@osrg.net>		

4.2. Volume Format

A NILFS2 volume is equally divided into a number of segments except for the super block (SB) and segment #0. A segment is the container of logs. Each log is composed of summary information blocks, payload blocks, and an optional super root block (SR):

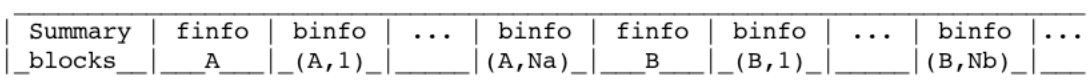


The payload blocks are organized per file, and each file consists of data blocks and B-tree node blocks:



Since only the modified blocks are written in the log, it may have files without data blocks or B-tree node blocks.

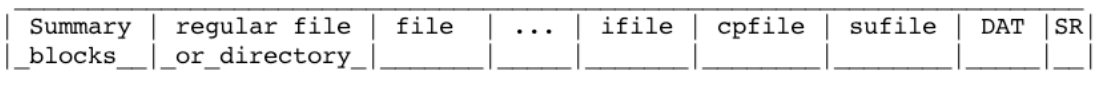
The organization of the blocks is recorded in the summary information blocks, which contains a header structure (nilfs_segment_summary), per file structures (nilfs_finfo), and per block structures (nilfs_binfo):



The logs include regular files, directory files, symbolic link files and several metadata files. The metadata files are the files used to maintain file system metadata. The current version of NILFS2 uses the following metadata files:

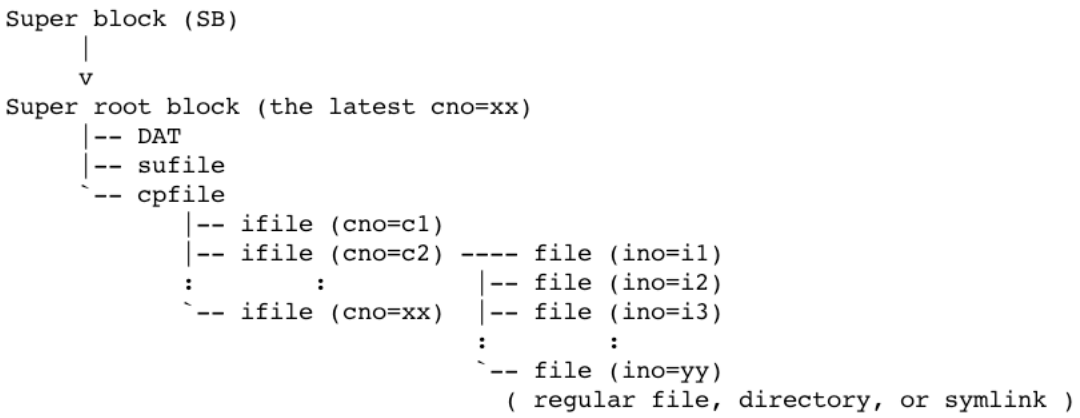
#	ITEM	DESCRIPTION
1	Inode file (ifile)	Stores on-disk inodes
2	Checkpoint file (cpfile)	Stores checkpoints
3	Segment usage file (sufile)	Stores allocation state of segments
4	Data address translation file	Maps virtual block numbers to usual (DAT) block numbers. This file serves to make on-disk blocks relocatable

The following figure shows a typical organization of the logs:

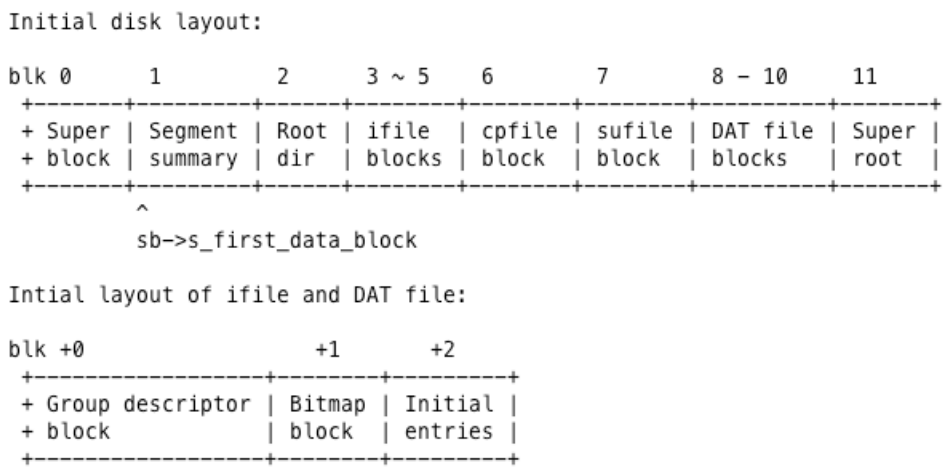


To stride over segment boundaries, this sequence of files may be split into multiple logs. The sequence of logs that should be treated as logically one log, is delimited with flags marked in the segment summary. The recovery code of nilfs2 looks this boundary information to ensure atomicity of updates.

The super root block is inserted for every checkpoint. It includes three special inodes, inode for the DAT, cpfile, and sufile. Inodes of regular files, directories, symlinks and other special files are included in the ifile. The inode of ifile itself is included in the corresponding checkpoint entry in the cpfile. Thus, the hierarchy among NILFS2 files can be depicted as follows:



A mkfs.nilfs2 utility initializes primary superblock, secondary superblock and segment #0.



4.3. Superblock

Every NILFS2 volume has two superblocks after creation. Primary superblock is located in beginning of the volume (1 KB from volume begin).

```
00000400  02 00 00 00 00 00 34 34  18 01 00 00 18 bf 65 5c  |......44......e\|
00000410  3e 41 6d 74 02 00 00 00  7f 00 00 00 00 00 00 00  |>Amt............|
00000420  00 00 00 40 00 00 00 00  01 00 00 00 00 00 00 00  |...@............|
00000430  00 08 00 00 05 00 00 00  01 00 00 00 00 00 00 00  |................|
00000440  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000450  00 f0 03 00 00 00 00 00  ac d7 a2 4f 00 00 00 00  |...........O....|
00000460  00 00 00 00 00 00 00 00  ac d7 a2 4f 00 00 00 00  |...........O....|
00000470  00 00 32 00 01 00 01 00  ac d7 a2 4f 00 00 00 00  |..2........O....|
00000480  00 4e ed 00 00 00 00 00  00 00 00 00 0b 00 00 00  |.N..............|
00000490  80 00 20 00 c0 00 10 00  3c 86 56 95 4e 2e 46 e9  |.. .....<.V.N.F.|
000004a0  a9 c0 37 10 ac e1 5f b6  74 65 73 74 31 00 00 00  |..7..._.test1...|
000004b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Secondary superblock is located in 1024 bytes from the volume's end.

```
3ffff000  02 00 00 00 00 00 34 34  18 01 00 00 18 bf 65 5c  |......44......e\|
3ffff010  3e 41 6d 74 02 00 00 00  7f 00 00 00 00 00 00 00  |>Amt............|
3ffff020  00 00 00 40 00 00 00 00  01 00 00 00 00 00 00 00  |...@............|
3ffff030  00 08 00 00 05 00 00 00  01 00 00 00 00 00 00 00  |................|
3ffff040  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
3ffff050  00 f0 03 00 00 00 00 00  ac d7 a2 4f 00 00 00 00  |...........O....|
3ffff060  00 00 00 00 00 00 00 00  ac d7 a2 4f 00 00 00 00  |...........O....|
3ffff070  00 00 32 00 01 00 01 00  ac d7 a2 4f 00 00 00 00  |..2........O....|
3ffff080  00 4e ed 00 00 00 00 00  00 00 00 00 0b 00 00 00  |.N..............|
3ffff090  80 00 20 00 c0 00 10 00  3c 86 56 95 4e 2e 46 e9  |.. .....<.V.N.F.|
3ffff0a0  a9 c0 37 10 ac e1 5f b6  74 65 73 74 31 00 00 00  |..7..._.test1...|
3ffff0b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
40000000
```

The superblock of NILFS2 filesystem is described by means of nilfs_super_block structure:

```
struct nilfs_super_block {
/*0x00*/        __le32 s_rev_level;
                __le16 s_minor_rev_level;
                __le16 s_magic;
                __le16 s_bytes;
                __le16 s_flags;
                __le32 s_crc_seed;
/*0x10*/        __le32 s_sum;
                __le32 s_log_block_size;
                __le64 s_nsegments;
/*0x20*/        __le64 s_dev_size;
                __le64 s_first_data_block;
/*0x30*/        __le32 s_blocks_per_segment;
                __le32 s_r_segments_percentage;
                __le64 s_last_cno;
/*0x40*/        __le64  s_last_pseg;
                __le64  s_last_seq;
/*0x50*/        __le64 s_free_blocks_count;
                __le64 s_ctime;
/*0x60*/        __le64 s_mtime;
                __le64 s_wtime;
/*0x70*/        __le16 s_mnt_count;
                __le16 s_max_mnt_count;
```

```
                __le16 s_state;
                __le16 s_errors;
                __le64 s_lastcheck;
/*0x80*/        __le32 s_checkinterval;
                __le32 s_creator_os;
                __le16 s_def_resuid;
                __le16 s_def_resgid;
                __le32 s_first_ino;
/*0x90*/        __le16 s_inode_size;
                __le16 s_dat_entry_size;
                __le16 s_checkpoint_size;
                __le16 s_segment_usage_size;
/*0x98*/        __u8   s_uuid[16];
/*0xA8*/        char   s_volume_name[80];
/*0xF8*/        __le32 s_c_interval;
                __le32 s_c_block_max;
/*0x100*/       __le64 s_feature_compat;
                __le64 s_feature_compat_ro;
                __le64 s_feature_incompat;
                __u32  s_reserved[186];
};
```

Meaning of the every field in the NILFS2 superblock is described in the table:

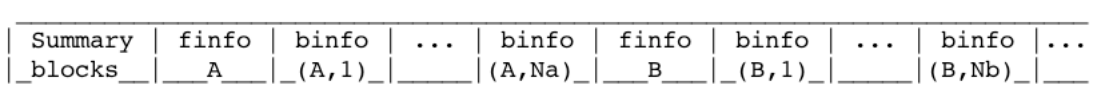
OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x4	s_rev_level	Major revision level. Currently it equals by 2.
0x0004	0x2	s_minor_rev_level	Minor revision level. Currently it equals by 0.
0x0006	0x2	s_magic	Magic signature. Should be equal by 0x3434.
0x0008	0x2	s_bytes	Bytes count of CRC calculation for this structure (s_reserved is excluded).
0x000A	0x2	s_flags	Filesystem independent flags.
0x000C	0x4	s_crc_seed	Seed value of CRC calculation.
0x0010	0x4	s_sum	Check sum of super block.
0x0014	0x4	s_log_block_size	Block size represented as follows blocksize = 1 << (s_log_block_size + 10).
0x0018	0x8	s_nsegments	Number of segments in filesystem.
0x0020	0x8	s_dev_size	Block device size in bytes.
0x0028	0x8	s_first_data_block	First segment disk block number.
0x0030	0x4	s_blocks_per_segment	Number of blocks per full segment.
0x0034	0x4	s_r_segments_percentage	Reserved segments percentage.
0x0038	0x8	s_last_cno	Last checkpoint number.
0x0040	0x8	s_last_pseg	Disk block address in which pseg (partial

			segment) was written last. Start block number of the latest segment.
0x0048	0x8	s_last_seq	Sequential number of segment that was written last.
0x0050	0x8	s_free_blocks_count	Free blocks count.
0x0058	0x8	s_ctime	Creation time (execution time of newfs).
0x0060	0x8	s_mtime	Mount time.
0x0068	0x8	s_wtime	Write time.
0x0070	0x2	s_mnt_count	Mount count.
0x0072	0x2	s_max_mnt_count	Maximal mount count.
0x0074	0x2	s_state	File system states. The flag can keep: ▪ NILFS_VALID_FS [0x0001] - Unmounted cleanly. ▪ NILFS_ERROR_FS [0x0002] - Errors detected. ▪ NILFS_RESIZE_FS [0x0004] - Resize required.
0x0076	0x2	s_errors	Behavior when detecting errors.
0x0078	0x8	s_lastcheck	Time of last check.
0x0080	0x4	s_checkinterval	Maximum time between checks. Default check interval – 180 days.
0x0084	0x4	s_creator_os	Code for OS. Code for Linux is 0. Codes from 1 to 4 are reserved to keep compatibility with ext2 creator-OS.
0x0088	0x2	s_def_resuid	Default UID for reserved blocks.
0x008A	0x2	s_def_resgid	Default GID for reserved blocks.
0x008C	0x4	s_first_ino	First user's file inode number. It equals by 11 for NILFS2.
0x0090	0x2	s_inode_size	Size of on-disk inode.
0x0092	0x2	s_dat_entry_size	DAT entry size.
0x0094	0x2	s_checkpoint_size	Size of a checkpoint.
0x0096	0x2	s_segment_usage_size	Size of a segment usage.
0x0098	0x10	s_uuid	128-bit UUID for volume.
0x00A8	0x50	s_volume_name	Volume label.
0x00F8	0x4	s_c_interval	Commit interval of segment.
0x00FC	0x4	s_c_block_max	Threshold of data amount for the segment construction (number of blocks to create segment).

0x0100	0x8	s_feature_compat	Compatible feature set.
0x0108	0x8	s_feature_compat_ro	Read-only compatible feature set.
0x0110	0x8	s_feature_incompat	Incompatible feature set.
0x0118	0x2E8	s_reserved	Padding to the end of the block.
0x0400	{END OF STRUCTURE}		

4.4. Segment Summary

The organization of the blocks is recorded in the summary information blocks, which contains a header structure (nilfs_segment_summary), per file structures (nilfs_finfo), and per block structures (nilfs_binfo):



The segment summary header is described by means of nilfs_segment_summary structure:

```
struct nilfs_segment_summary {
/*0x00*/        __le32 ss_datasum;
                __le32 ss_sumsum;
                __le32 ss_magic;
                __le16 ss_bytes;
                __le16 ss_flags;
/*0x10*/        __le64 ss_seq;
                __le64 ss_create;
/*0x20*/        __le64 ss_next;
                __le32 ss_nblocks;
                __le32 ss_nfinfo;
/*0x30*/        __le32 ss_sumbytes;
                __le32 ss_pad;
                __le64 ss_cno;
};
```

Meaning of the every field in the segment summary header is described in the table:

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x4	ss_datasum	Checksum of data.
0x0004	0x4	ss_sumsum	Checksum of segment summary.
0x0008	0x4	ss_magic	Segment summary magic number. Should be equal by 0x1eaffa11.
0x000C	0x2	ss_bytes	Size of nilfs_segment_summary structure in bytes.

0x000E	0x2	ss_flags	Segment summary flags. The flags field can keep: ▪ NILFS_SS_LOGBGN [0x0001] - begins a logical segment ▪ NILFS_SS_LOGEND [0x0002] - ends a logical segment ▪ NILFS_SS_SR [0x0004] - has super root ▪ NILFS_SS_SYNDT [0x0008] - includes data only updates ▪ NILFS_SS_GC [0x0010] - segment written for cleaner operation
0x0010	0x8	ss_seq	Segment's sequence number.
0x0018	0x8	ss_create	Creation timestamp.
0x0020	0x8	ss_next	Next segment's start block.
0x0028	0x4	ss_nblocks	Number of really used blocks in segment.
0x002C	0x4	ss_nfinfo	Number of finfo structures in segment summary block.
0x0030	0x4	ss_sumbytes	Total size of segment summary in bytes.
0x0034	0x4	ss_pad	Padding field.
0x0038	0x8	ss_cno	Checkpoint number.
0x0040	{END OF STRUCTURE}		

It follows array of file information items after segment summary header. The count of items in the array is described by ss_nfinfo field of nilfs_segment_summary structure.

The file information items is described by means of nilfs_finfo structure:

```
struct nilfs_finfo {
/*0x00*/        __le64 fi_ino;
                __le64 fi_cno;
/*0x10*/        __le32 fi_nblocks;
                __le32 fi_ndatablk;
};
```

Meaning of the every field in the file information item is described in the table:

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	fi_ino	File's inode number.
0x0008	0x8	fi_cno	Checkpoint number.
0x0010	0x4	fi_nblocks	Number of blocks (including intermediate

			blocks).
0x0014	0x4	fi_ndatablk	Number of file data blocks.
0x0018	{END OF STRUCTURE}		

It follows array of block information items after each file information item. The count of items in the array is described by fi_nblocks and fi_ndatablk fields of nilfs_finfo structure. The fi_ndatablk field describes count of data blocks then as fi_nblocks field describes the whole count of blocks including a B-Tree intermediate blocks.

The block information item is described by

```
union nilfs_binfo {
        struct nilfs_binfo_v bi_v;
        struct nilfs_binfo_dat bi_dat;
};
```

The nilfs_binfo_v structure informs about the block to which a virtual block number is assigned:

```
struct nilfs_binfo_v {
        __le64 bi_vblocknr;
        __le64 bi_blkoff;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	bi_vblocknr	Virtual block number.
0x0008	0x8	bi_blkoff	Block offset.
0x0010	{END OF STRUCTURE}		

The nilfs_binfo_dat is the structure storing summary information only applied to B-Tree intermediate blocks of the DAT meta-data file.

```
struct nilfs_binfo_dat {
        __le64 bi_blkoff;
        __u8   bi_level;
        __u8   bi_pad[7];
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	bi_blkoff	Block offset.
0x0008	0x1	bi_level	Level.
0x0009	0x7	bi_pad	Padding.
0x0010	{END OF STRUCTURE}		

For data blocks of the DAT file, a 64-bit block offset number (__le64) is allocated per block.

Let's see some case of DAT file description in segment summary:

```
00025128  03 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00025138  06 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
00025148  01 00 00 00 00 00 00 00  02 00 00 00 00 00 00 00  |................|
00025158  82 00 00 00 00 00 00 00  a2 00 00 00 00 00 00 00  |................|
00025168  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
```

It is possible to see that file with inode 3 (DAT file) contains 6 blocks: 5 data blocks and 1 intermediate block. The list of data blocks description includes 0x0, 0x1, 0x2, 0x82 and 0xa2 virtual block number. The list of intermediate blocks follows by data blocks and contains only one block with number 0x0 and level 0x1.

4.5. Root folder

Root folder is a sequence of directory entries. The directory entry can be described by nilfs_dir_entry structure:

```
struct nilfs_dir_entry {
        __le64 inode;
        __le16 rec_len;
        __u8   name_len;
        __u8   file_type;
        char   name[1];
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	inode	Inode number of the directory entry.
0x0008	0x2	rec_len	Directory entry length in bytes. It is 8 bytes aligned. Last directory entry in the sequence keeps in this field size from the entry's beginning to the block's end.
0x000A	0x1	name_len	Name length in bytes.
0x000B	0x1	file_type	NILFS directory entry types. Only the low 3 bits are used. The other bits are reserved for now. The file type can have such values: ▪ NILFS_FT_REG_FILE [0x1] – regular file ▪ NILFS_FT_DIR [0x2] – directory ▪ NILFS_FT_CHRDEV [0x3] – char device ▪ NILFS_FT_BLKDEV [0x4] – block device ▪ NILFS_FT_FIFO [0x5] – named pipe special file ▪ NILFS_FT_SOCK [0x6] – socket special file ▪ NILFS_FT_SYMLINK [0x7] –symlink
0x000C	{VARIABLE}	name	File name. It can variable size in bytes but

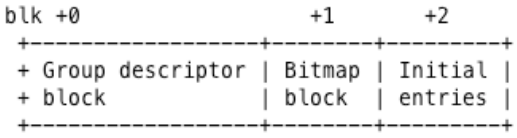
			cannot be greater than NILFS_NAME_LEN (255 bytes).
{MAX}	{END OF STRUCTURE}		

It is created ".", ".." directories and ".nilfs" file during initial nilfs2 volume creation.

```
00002000  02 00 00 00 00 00 00 00  10 00 01 02 2e 00 00 00  |................|
00002010  02 00 00 00 00 00 00 00  10 00 02 02 2e 2e 00 00  |................|
00002020  0b 00 00 00 00 00 00 00  e0 0f 06 01 2e 6e 69 6c  |.............nil|
00002030  66 73 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |fs..............|
```

4.6. Inode File

Inode file (ifile) stores on-disk inodes. The initial layout of ifile can be represented by picture:



Group descriptor block contains sequence of block group descriptors. The block group descriptor is described by nilfs_palloc_group_desc structure:

```
struct nilfs_palloc_group_desc {
        __le32 pg_nfrees;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x4	pg_nfrees	Number of free entries in block group.
0x0004	{END OF STRUCTURE}		

It follows bitmap block after group descriptor block. The bitmap block contains bitmap that defines used and free blocks in a block group.

Finally, ifile contains block that keeps initial on-disk inodes list. The on-disk inode is described by nilfs_inode structure:

```
struct nilfs_inode {
/*0x00*/        __le64 i_blocks;
                __le64 i_size;
/*0x10*/        __le64 i_ctime;
                __le64 i_mtime;
/*0x20*/        __le32 i_ctime_nsec;
                __le32 i_mtime_nsec;
                __le32 i_uid;
                __le32 i_gid;
/*0x30*/        __le16 i_mode;
                __le16 i_links_count;
```

```
                __le32 i_flags;
                __le64 i_bmap[7];
/*0x70*/        __le64 i_xattr;
                __le32 i_generation;
                __le32 i_pad;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	i_blocks	Blocks count in a file, which is described by inode.
0x0008	0x8	i_size	Size in bytes of content in a file.
0x0010	0x8	i_ctime	Creation time (seconds).
0x0018	0x8	i_mtime	Last modification time (seconds).
0x0020	0x4	i_ctime_nsec	Creation time (nano seconds).
0x0024	0x4	i_mtime_nsec	Last modification time (nano seconds).
0x0028	0x4	i_uid	User ID.
0x002C	0x4	i_gid	Group ID.
0x0030	0x2	i_mode	File mode. The following flags are defined for this field: ▪ S_IFMT [0170000] – bit mask for the file type bit fields ▪ S_IFSOCK [0140000] – socket ▪ S_IFLNK [0120000] – symbolic link ▪ S_IFREG [0100000] – regular file ▪ S_IFBLK [0060000] – block device ▪ S_IFDIR [0040000] – directory ▪ S_IFCHR [0020000] – character device ▪ S_IFIFO [0010000] – FIFO ▪ S_ISUID [0004000] – set UID bit ▪ S_ISGID [0002000] – set-group-ID bit ▪ S_ISVTX [0001000] – sticky bit ▪ S_IRWXU [00700] – mask for file owner permissions ▪ S_IRUSR [00400] – owner has read permission ▪ S_IWUSR [00200] – owner has write permission ▪ S_IXUSR [00100] – owner has execute permission ▪ S_IRWXG [00070] – mask for group permissions ▪ S_IRGRP [00040] – group has read permission ▪ S_IWGRP [00020] – group has write

			permission ▪ S_IXGRP [00010] – group has execute permission ▪ S_IRWXO [00007] – mask for permissions for others (not in group) ▪ S_IROTH [00004] – others have read permission ▪ S_IWOTH [00002] – others have write permission ▪ S_IXOTH [00001] – others have execute permission
0x0032	0x2	i_links_count	Links count.
0x0034	0x4	i_flags	Inode flags.
0x0038	0x38	i_bmap	Block mapping. This is a direct block mapping sequence or B-Tree root node.
0x0070	0x8	i_xattr	Extended attributes.
0x0078	0x4	i_generation	File generation (for NFS).
0x007C	0x4	i_pad	Padding.
0x0080	{END OF STRUCTURE}		

The mkfs.nilfs2 creates such inodes:

ID	ABBREVIATION	DESCRIPTION
1		
2	NILFS_ROOT_INO	Root dir
3	NILFS_DAT_INO	DAT file
4	NILFS_CPFILE_INO	Checkpoint file (cpfile)
5	NILFS_SUFILE_INO	Segment usage file (sufile)
6	NILFS_IFILE_INO	Inode file (ifile)
7	NILFS_ATIME_INO	Atime file (reserved)
8	NILFS_XATTR_INO	Xattribute file (reserved)
9		
10	NILFS_SKETCH_INO	Sketch file
11	NILFS_USER_INO	Fisrt user's file inode number. It is created “.nilfs” file.

Inodes of DAT file, cpfile, and sufile are written in super root block instead of ifile. Inode of ifile is written to checkpoint file. All other inodes are written in ifile.

4.7. Checkpoint file

Checkpoint file keeps a sequence of checkpoints' descriptors. It begins from header that is described by

nilfs_cpfile_header structure:

```
struct nilfs_cpfile_header {
        __le64 ch_ncheckpoints;
        __le64 ch_nsnapshots;
        struct nilfs_snapshot_list ch_snapshot_list;
};

struct nilfs_snapshot_list {
        __le64 ssl_next;
        __le64 ssl_prev;
};
```

OFFSET	SIZE		NAME	DESCRIPTION
0x0000	0x8		ch_ncheckpoints	Number of checkpoints.
0x0008	0x8		ch_nsnapshots	Number of snapshots.
0x0010	0x10	0x8	ch_snapshot_lists.sl_next	Next checkpoint number on snapshot list.
		0x8	ch_snapshot_lists.ssl_prev	Previous checkpoint number on snapshot list.
0x0020	{END OF STRUCTURE}			

Then, it is located the beginning of checkpoint descriptors' sequence on the ((sizeof(struct nilfs_cpfile_header) + sizeof(struct nilfs_checkpoint) - 1) / sizeof(struct nilfs_checkpoint)) offset in bytes from the beginning of cpfile's first block. Really, this offset equals by 0xC0 bytes. The checkpoint is described by nilfs_checkpoint structure:

```
struct nilfs_checkpoint {
/*0x00*/        __le32 cp_flags;
                __le32 cp_checkpoints_count;
                struct nilfs_snapshot_list cp_snapshot_list;
                __le64 cp_cno;
/*0x20*/        __le64 cp_create;
                __le64 cp_nblk_inc;
/*0x30*/        __le64 cp_inodes_count;
                __le64 cp_blocks_count;
/*0x40*/        struct nilfs_inode cp_ifile_inode;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x4	cp_flags	Checkpoint flags. It can have such values: ▪ NILFS_CHECKPOINT_SNAPSHOT [0x1] ▪ NILFS_CHECKPOINT_INVALID [0x2] ▪ NILFS_CHECKPOINT_SKETCH [0x4] ▪ NILFS_CHECKPOINT_MINOR [0x8]
0x0004	0x4	cp_checkpoints_count	Checkpoints count in a block.

0x0008	0x10	0x8	cp_snapshot_lists.sl_next	Next checkpoint number on snapshot list.
		0x8	cp_snapshot_lists.ssl_prev	Previous checkpoint number on snapshot list.
0x0018	0x8		cp_cno	Checkpoint number.
0x0020	0x8		cp_create	Creation timestamp.
0x0028	0x8		cp_nblk_inc	Number of blocks incremented by this checkpoint.
0x0030	0x8		cp_inodes_count	Inodes count.
0x0038	0x8		cp_blocks_count	Used blocks count.
0x0040	0x80		cp_ifile_inode	Inode of ifile.
0x00C0	{END OF STRUCTURE}			

The mkfs.nilfs2 utility initializes block of cpfile by ((blocksize / sizeof(struct nilfs_checkpoint)) – 1) checkpoint description items. The rest items are declared as NILFS_CHECKPOINT_INVALID. The every checkpoint description item has number after initialization.

4.8. Segment usage file

Segment usage file stores allocation state of segments. It contains a sequence of segment usage description items. The sufile begins from a header that is described by nilfs_sufile_header structure:

```
struct nilfs_sufile_header {
        __le64 sh_ncleansegs;
        __le64 sh_ndirtysegs;
        __le64 sh_last_alloc;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	sh_ncleansegs	Number of clean segments on the volume.
0x0008	0x8	sh_ndirtysegs	Number of dirty segments on the volume.
0x0010	0x8	sh_last_alloc	Last allocated on volume segment number.
0x0018	{END OF STRUCTURE}		

Then, it is located the beginning of segment usage descriptors' sequence on the ((sizeof(struct nilfs_sufile_header) + sizeof(struct nilfs_segment_usage) - 1) / sizeof(struct nilfs_segment_usage)) offset in bytes from the beginning of sufile's first block. Really, this offset equals by 0x20 bytes. The segment usage description item is described by nilfs_segment_usage structure:

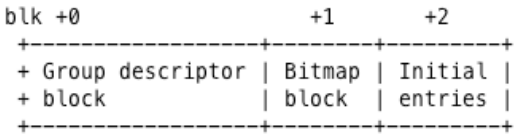
```
struct nilfs_segment_usage {
        __le64 su_lastmod;
        __le32 su_nblocks;
        __le32 su_flags;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	su_lastmod	Modification timestamp.
0x0008	0x4	su_nblocks	Number of live blocks in segment.
0x000C	0x4	su_flags	Segment usage flags. The field can have such values: ▪ NILFS_SEGMENT_USAGE_ACTIVE [0x1] ▪ NILFS_SEGMENT_USAGE_DIRTY [0x2] ▪ NILFS_SEGMENT_USAGE_ERROR [0x4]
0x0010	{END OF STRUCTURE}		

The mkfs.nilfs2 utility initializes only two segment usage description items.

4.9. Data Address Translation File

Data address translation (DAT) file maps virtual block numbers to usual block numbers. This file serves to make on-disk blocks relocatable. The initial layout of DAT file can be represented by picture:



Group descriptor block contains sequence of block group descriptors. The block group descriptor is described by nilfs_palloc_group_desc structure:

```
struct nilfs_palloc_group_desc {
        __le32 pg_nfrees;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x4	pg_nfrees	Number of free entries in block group.
0x0004	{END OF STRUCTURE}		

It follows bitmap block after group descriptor block. The bitmap block contains bitmap that defines used and free blocks in a block group.

Finally, DAT file contains block that keeps disk address translation entries. The disk address translation entry is described by nilfs_dat_entry structure:

```
struct nilfs_dat_entry {
                __le64 de_blocknr;
                __le64 de_start;
```

```
        __le64 de_end;
        __le64 de_rsv;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x8	de_blocknr	Block number.
0x0008	0x8	de_start	Start checkpoint number.
0x0010	0x8	de_end	End checkpoint number.
0x0018	0x8	de_rsv	Reserved for future use.
0x0020	{END OF STRUCTURE}		

The mkfs.nilfs2 utility initializes 6 address translation entries.

```
0000a000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000a010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000a020  02 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
0000a030  ff ff ff ff ff ff ff ff  00 00 00 00 00 00 00 00  |................|
0000a040  03 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
0000a050  ff ff ff ff ff ff ff ff  00 00 00 00 00 00 00 00  |................|
0000a060  04 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
0000a070  ff ff ff ff ff ff ff ff  00 00 00 00 00 00 00 00  |................|
0000a080  05 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
0000a090  ff ff ff ff ff ff ff ff  00 00 00 00 00 00 00 00  |................|
0000a0a0  06 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
0000a0b0  ff ff ff ff ff ff ff ff  00 00 00 00 00 00 00 00  |................|
0000a0c0  07 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
0000a0d0  ff ff ff ff ff ff ff ff  00 00 00 00 00 00 00 00  |................|
```

It describes:

- Root dir block: [0x2] [offset in bytes: 0x2000].
- Inode file blocks: [0x3 - 0x5] [offset in bytes: 0x3000 - 0x5000].
- Checkpoint file block: [0x6] [offset in bytes: 0x6000].
- Segment usage file block: [0x7] [offset in bytes: 0x7000].

The DAT file blocks are described in segment summary. It follows the sequence of 64-bit block offset numbers (__le64) after DAT's file information item.

4.10. Super Root

The super root block is inserted for every checkpoint. It includes three special inodes, inode for the DAT, cpfile, and sufile.

The super root is described by nilfs_super_root structure:

```
struct nilfs_super_root {
/*0x000*/       __le32 sr_sum;
                __le16 sr_bytes;
                __le16 sr_flags;
                __le64 sr_nongc_ctime;
/*0x010*/       struct nilfs_inode sr_dat;
```

```
/*0x090*/       struct nilfs_inode sr_cpfile;
/*0x110*/       struct nilfs_inode sr_sufile;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x4	sr_sum	Super root checksum.
0x0004	0x2	sr_bytes	Byte count of the structure.
0x0006	0x2	sr_flags	Flags (reserved).
0x0008	0x8	sr_nongc_ctime	Write time of the last segment (not for cleaner operation).
0x0010	0x80	sr_dat	DAT file inode.
0x0090	0x80	sr_cpfile	Checkpoint file inode.
0x0110	0x80	sr_sufile	Segment usage file inode.
0x0190	{END OF STRUCTURE}		

4.11. NILFS B-Tree

NILFS uses B-Trees for file blocks' placement description. It describes regular file's and DAT file's blocks mapping by means of B-Tree.

B-Tree begins from root node that placed in i_bmap field of raw inode. Every node of B-Tree begins from header that is described by means of nilfs_btree_node structure:

```
struct nilfs_btree_node {
        __u8 bn_flags;
        __u8 bn_level;
        __le16 bn_nchildren;
        __le32 bn_pad;
};
```

OFFSET	SIZE	NAME	DESCRIPTION
0x0000	0x1	bn_flags	Node flags. The field can have such values: ▪ NILFS_BTREE_NODE_ROOT [0x01]
0x0001	0x1	bn_level	Level of the node. It can't be greater than NILFS_BTREE_LEVEL_MAX [0xE].
0x0002	0x2	bn_nchildren	Number of children.
0x0004	0x4	bn_pad	Padding.
0x0008	{END OF STRUCTURE}		

The rest of node's space is divided on two equal parts: 1) first part contains sequence of dkey items

sizeof(__le64); 2) second part contains sequence of dptr items sizeof(__le64). The dkey is a virtual block number then as dptr is a real block offset.

Let's see B-Tree structure of DAT file. Root node of B-Tree is placed in inode of DAT file:

```
00033010  1b 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00033020  a1 88 a3 4f 00 00 00 00  a1 88 a3 4f 00 00 00 00  |...O.......O....|
00033030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00033040  00 80 01 00 00 00 00 00  01 02 01 00 00 00 00 00  |................|
00033050  00 00 00 00 00 00 00 00  14 00 00 00 00 00 00 00  |................|
00033060  15 00 00 00 00 00 00 00  32 00 00 00 00 00 00 00  |........2.......|
00033070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00033080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

It begins from node header. The nilfs_btree_node structure has such content:

```
01 02 01 00 00 00 00 00
```

It means that the node is root node (flag field equal by NILFS_BTREE_NODE_ROOT [0x01]). It placed on level equal by 2 and has 1 child. Moreover, it is possible to extract from the rest of node that dkey with value 0x0 correspond to dptr with value 0x32. It means that child node of the root node is located in 0x32 block of the volume.

```
00032000  00 01 05 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00032010  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
00032020  02 00 00 00 00 00 00 00  82 00 00 00 00 00 00 00  |................|
00032030  a2 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00032040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00032800  00 00 00 00 00 00 00 00  2d 00 00 00 00 00 00 00  |........-.......|
00032810  2e 00 00 00 00 00 00 00  2f 00 00 00 00 00 00 00  |......../.......|
00032820  30 00 00 00 00 00 00 00  31 00 00 00 00 00 00 00  |0.......1.......|
00032830  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

The child node begins from node header also:

```
00 01 05 00 00 00 00 00
```

This is a child node (flag field is set as zero) that is located on 1-th level and has 5 children. This node contains such sequence of [dkey : dptr] pairs:

dkey	dptr
0x0	0x2d
0x1	0x2e
0x2	0x2f
0x82	0x30
0xa2	0x31

Thereby, data blocks of DAT file are located in 0x2d, 0x2e, 0x2f, 0x30 and 0x31 blocks.

5. Internal Mechanisms

<TODO>

6. Driver Design

6.1. Modules List

MODULE	DESCRIPTION	TYPE	FILE	AUTHOR
nilfs2_fs	NILFS2 on-disk structures and common declarations	Decl	nilfs2_fs.h	Koji Sato, Ryusuke Konishi
the_nilfs	Multiple NILFS mount points supervisor	Decl	the_nilfs.h	Ryusuke Konishi
		Impl	the_nilfs.c	
nilfs	NILFS local header file	Decl	nilfs.h	Koji Sato, Ryusuke Konishi
alloc	Persistent object (dat entry/disk inode) allocator/deallocator	Decl	alloc.h	Koji Sato, Ryusuke Konishi, Amagai Yoshiji
		Impl	alloc.c	
bmap	NILFS block mapping	Decl	bmap.h	Koji Sato
		Impl	bmap.c	
btnode	NILFS B-tree node cache	Decl	btnode.h	Seiji Kihara, Ryusuke Konishi
		Impl	btnode.c	
btree	NILFS B-tree	Decl	btree.h	Koji Sato
		Impl	btree.c	
cpfile	NILFS checkpoint file	Decl	cpfile.h	Koji Sato
		Impl	cpfile.c	
dat	NILFS disk address translation functionality	Decl	dat.h	Koji Sato
		Impl	dat.c	
dir	NILFS directory entry operations functionality	Impl	dir.c	Amagai Yoshiji
direct	NILFS's direct block pointer functiuonality	Decl	direct.h	Koji Sato
		Impl	direct.c	
export	Declarations for making NILFS "exportable" filesystem	Decl	export.h	
file	NILFS regular file handling primitives	Impl	file.c	Amagai Yoshiji, Ryusuke Konishi
gcinode	Dummy inodes functionality to buffer blocks for garbage collection	Impl	gcinode.c	Seiji Kihara, Amagai Yoshiji, Ryusuke Konishi
ifile	NILFS inode file	Decl	ifile.h	Amagai Yoshiji, Ryusuke Konishi
		Impl	ifile.c	
inode	NILFS inode operations functionality	Impl	inode.c	Ryusuke Konishi
ioctl	NILFS ioctl operations functionality	Impl	ioctl.c	Koji Sato
mdt	NILFS meta data file functionality	Decl	mdt.h	Ryusuke Konishi
		Impl	mdt.c	
namei	NILFS pathname lookup operations functionality	Impl	namei.c	Amagai Yoshiji, Ryusuke Konishi
page	Buffer/Page management	Decl	page.h	Ryusuke Konishi,

	functionality specific to NILFS	Impl	page.c	Seiji Kihara
recovery	NILFS recovery logic functionality	Impl	recovery.c	Ryusuke Konishi
segbuf	NILFS segment buffer functionality	Decl	segbuf.h	Ryusuke Konishi
		Impl	segbuf.c	
segment	NILFS segment constructor functionality	Decl	segment.h	Ryusuke Konishi
		Impl	segment.c	
sufile	NILFS segment usage file functionality	Decl	sufile.h	Koji Sato
		Impl	sufile.c	Koji Sato, Ryusuke Konishi
super	NILFS module and super block management functionality	Impl	super.c	Ryusuke Konishi

- Ryusuke Konishi <konishi.ryusuke@lab.ntt.co.jp>
- Koji Sato <koji@osrg.net>
- Amagai Yoshiji <amagai@osrg.net>
- Seiji Kihara <kihara@osrg.net>

6.2. Driver Architecture

<TODO>

6.3. Data Abstractions

On-disk layout structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_inode	Structure of an inode on disk.	nilfs2_fs.h
2.	nilfs_super_root	Structure of super root block on disk.	
3.	nilfs_super_block	Structure of super block on disk.	
4.	nilfs_dir_entry	Structure of directory entry on disk.	
5.	nilfs_finfo	Structure of file information item in segment summary on disk.	
6.	nilfs_binfo_v	Structure informs about the block to which a virtual block number is assigned.	
7.	nilfs_binfo_dat	Structure storing summary information only applied to B-Tree intermediate blocks of the DAT meta-data file. For data blocks of the DAT file, a 64-bit block offset number (__le64) is allocated per block.	
8.	nilfs_binfo	Union describes structure of a block information item on disk.	
9.	nilfs_segment_summary	Structure of segment summary header on disk.	
10.	nilfs_btree_node	Structure of b-tree node header on disk.	
11.	nilfs_palloc_group_desc	Structure of block group descriptor on disk.	
12.	nilfs_dat_entry	Structure of disk address translation entry on disk.	
13.	nilfs_snapshot_list	Structure of snapshot list item descriptor on disk.	
14.	nilfs_checkpoint	Structure of checkpoint descriptor on disk.	
15.	nilfs_cpfile_header	Structure of checkpoint file header on disk.	

16.	nilfs_segment_usage	Structure of segment usage descriptor on disk.	
17.	nilfs_sufile_header	Structure of segment usage file header on disk.	

The the_nilfs module structures:

#	NAME	DESCRIPTION	FILE
1.	the_nilfs	Structure to supervise multiple nilfs mount points.	the_nilfs.h
2.	nilfs_root	The nilfs root object (mounted checkpoint).	

The alloc module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_req	Persistent allocator request and reply.	alloc.h
2.	nilfs_bh_assoc	Block offset and buffer head association.	
3.	nilfs_palloc_cache	Persistent object allocator cache.	

The bmap module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_ptr_req	Request for bmap pointer.	bmap.h
2.	nilfs_bmap_stats	The bmap statistics.	
3.	nilfs_bmap_operations	The bmap operation table.	
4.	nilfs_bmap	The bmap structure.	
5.	nilfs_bmap_store	Shadow copy of bmap state.	

The btnode module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btnode_chkey_ctxt	Change key context.	btnode.h

The btree module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_path	A path on which B-tree operations are executed.	btree.h

The cpfile module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_cpinfo	Checkpoint information.	nilfs2_fs.h
2.	nilfs_cpmode	Change checkpoint mode structure.	
3.	nilfs_cpstat	Checkpoint statistics.	
4.	nilfs_sustat	Segment usage statistics.	

The dat module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_info	On-memory private data of DAT file.	dat.c

The direct module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_node	Direct node.	direct.h

The export module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_fid	NILFS file id type.	export.h

The ifile module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_ifile_info	On-memory private data of ifile.	ifile.c

The inode module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_inode_info	The nilfs_inode_info structure describes nilfs inode data in memory representation.	nilfs.h
2.	nilfs_iget_args	Arguments used during comparison between inodes.	inode.c

The ioctl module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_argv	Argument vector.	nilfs2_fs.h
2.	nilfs_period	Period of checkpoint numbers.	
3.	nilfs_vinfo	Virtual block number information.	
4.	nilfs_vdesc	Descriptor of virtual block number.	
5.	nilfs_bdesc	Descriptor of disk block number.	

The mdt module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_shadow_map	Shadow mapping of meta data file.	mdt.h
2.	nilfs_mdt_info	On-memory private data of meta data file.	

The recovery module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_recovery_block	Work structure for recovery.	recovery.c
2.	nilfs_segment_entry	<TODO>	

The segbuf module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_segsum_info	On-memory segment summary.	segbuf.h
2.	nilfs_segment_buffer	Segment buffer.	
3.	nilfs_write_info	<TODO>	segbuf.c

The segment module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_recovery_info	Recovery information.	segment.h
2.	nilfs_cstage	Context of collection stage.	
3.	nilfs_segsum_pointer	<TODO>	
4.	nilfs_sc_info	Segment constructor information.	
5.	nilfs_sc_operations	Operations depending on the segment construction mode and file type.	segment.c
6.	nilfs_transaction_info	Context information for synchronization.	nilfs.h

The sufile module structures:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sufile_info	On-memory private data of sufile.	sufile.c
2.	nilfs_suinfo	Segment usage information.	nilfs2_fs.h

6.4. Subsystem APIs

The the_nilfs module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sb_need_update	<TODO>	the_nilfs.h
2.	nilfs_sb_will_flip	<TODO>	
3.	nilfs_set_last_segment	<TODO>	the_nilfs.c
4.	alloc_nilfs	Allocate a nilfs object.	
5.	destroy_nilfs	Destroy nilfs object.	
6.	init_nilfs	Initialize a NILFS instance. It performs common initialization per block device (e.g. reading the super block, getting disk layout information, initializing shared fields in the_nilfs).	
7.	load_nilfs	Load and recover the nilfs. It searches and load the latest super root, attaches the last segment, and does recovery if needed. The caller must call this exclusively for simultaneous mounts.	
8.	nilfs_nrsvsegs	Calculate the number of reserved segments.	
9.	nilfs_set_nsegments	<TODO>	
10.	nilfs_discard_segments	<TODO>	

11.	nilfs_count_free_blocks	<TODO>	
12.	nilfs_lookup_root	<TODO>	
13.	nilfs_find_or_create_root	<TODO>	
14.	nilfs_put_root	<TODO>	
15.	nilfs_near_disk_full	<TODO>	
16.	nilfs_fall_back_super_block	<TODO>	
17.	nilfs_swap_super_block	<TODO>	
18.	nilfs_get_root	<TODO>	
19.	nilfs_valid_fs	<TODO>	
20.	nilfs_get_segment_range	<TODO>	
21.	nilfs_get_segment_start_blocknr	<TODO>	
22.	nilfs_get_segnum_of_block	<TODO>	the_nilfs.h
23.	nilfs_terminate_segment	<TODO>	
24.	nilfs_shift_to_next_segment	<TODO>	
25.	nilfs_last_cno	<TODO>	
26.	nilfs_segment_is_active	<TODO>	

The alloc module API:

- Persistent allocator API.
- Persistent deallocator API.
- Persistent object allocator cache API.

Persistent allocator API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_init_blockgroup	Initialize private variables for allocator.	alloc.c
2.	nilfs_palloc_prepare_alloc_entry	Prepare to allocate a persistent object.	
3.	nilfs_palloc_commit_alloc_entry	Finish allocation of a persistent object.	
4.	nilfs_palloc_abort_alloc_entry	Cancel allocation of a persistent object.	

Persistent deallocator API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_prepare_free_entry	Prepare to deallocate a persistent object.	alloc.c
2.	nilfs_palloc_commit_free_entry	Finish deallocating a persistent object.	
3.	nilfs_palloc_abort_free_entry	Cancel deallocating a persistent object.	
4.	nilfs_palloc_freev	Deallocate a set of persistent objects.	

Persistent object allocator cache API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_setup_cache	Initialize meta data file's persistent object allocator cache.	alloc.c
2.	nilfs_palloc_clear_cache	Clear meta data file's persistent object allocator cache.	
3.	nilfs_palloc_destroy_cache	Destroy meta data file's persistent object allocator cache.	

The bmap module API:

- Block mapping initialization, saving and shadowing API.
- Block mapping modification API.
- Block mapping lookup API.
- Block mapping state management API.
- Internal API for bmap, direct and b-tree subsystems.

Block mapping initialization, saving and shadowing API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_init_gc	Initialize nilfs_bmap structure of GC inode.	bmap.c
2.	nilfs_bmap_read	Read a bmap from raw inode.	
3.	nilfs_bmap_write	Write back a bmap to raw inode.	
4.	nilfs_bmap_save	Save bmap in shadow map.	
5.	nilfs_bmap_restore	Restore bmap from shadow map.	

Block mapping modification API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_insert	Insert a new key-record pair into a bmap.	bmap.c
2.	nilfs_bmap_delete	Delete a key-record pair from a bmap.	
3.	nilfs_bmap_truncate	Truncate a bmap to a specified key. It removes key-record pairs whose keys are greater than or equal to key from bmap.	
4.	nilfs_bmap_clear	Free resources a bmap holds.	
5.	nilfs_bmap_assign	Assign the block number to the buffer specified by buffer head.	

Block mapping lookup API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_lookup_at_level	Find a data block or node block. It finds a record whose key matches a key in the block at level of the bmap.	bmap.c
2.	nilfs_bmap_lookup_contig	Find a contiguous region of blocks.	
3.	nilfs_bmap_last_key	Get last valid key in bmap.	
4.	nilfs_bmap_lookup_dirty_buffers	Build the list of all dirty buffers associated with bmap.	
5.	nilfs_bmap_lookup	Find a data block or node block from the level equals to one.	bmap.h

Block mapping state management API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_test_and_clear_dirty	Test and clear a bmap dirty state.	bmap.c
2.	nilfs_bmap_propagate	Propagate dirty state. It marks the buffers that	

		directly or indirectly refer to the block specified by buffer head as dirty.	
3.	nilfs_bmap_mark	Mark the block specified by a key and level as dirty.	

Internal API for bmap, direct and b-tree subsystems:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_get_dat	Return DAT file inode.	bmap.h
2.	nilfs_bmap_prepare_alloc_ptr	Prepare to allocate a persistent object.	
3.	nilfs_bmap_commit_alloc_ptr	Finish allocation of a persistent object.	
4.	nilfs_bmap_abort_alloc_ptr	Cancel allocation of a persistent object.	
5.	nilfs_bmap_prepare_end_ptr	<TODO>	
6.	nilfs_bmap_commit_end_ptr	<TODO>	
7.	nilfs_bmap_abort_end_ptr	<TODO>	
8.	nilfs_bmap_set_target_v	<TODO>	
9.	nilfs_bmap_data_get_key	<TODO>	
10.	nilfs_bmap_find_target_seq	<TODO>	
11.	nilfs_bmap_find_target_in_group	<TODO>	
12.	nilfs_bmap_dirty	Check that bmap is in dirty state.	
13.	nilfs_bmap_set_dirty	Set bmap as dirty.	
14.	nilfs_bmap_clear_dirty	Unset bmap as dirty.	

The btnode module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btnode_create_block	Initialize buffer for requested block number.	btnode.c
2.	nilfs_btnode_submit_block	Add node buffer and submit read request.	
3.	nilfs_btnode_delete	Delete B-tree node buffer.	
4.	nilfs_btnode_cache_clear	Invalidate and truncate all pages in page cache of b-tree node cache.	
5.	nilfs_btnode_prepare_change_key	Prepare to move contents of the block for old key to one of new key.	
6.	nilfs_btnode_commit_change_key	Commit the change_key operation.	
7.	nilfs_btnode_abort_change_key	Abort the change_key operation.	

The btree module API:

- B-tree API.
- B-tree operation table (struct nilfs_bmap_operations).
- GC inode's bmap operation table (struct nilfs_bmap_operations).

B-tree API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_init	Make special initializations of bmap for the case of inode with b-tree.	btree.c
2.	nilfs_btree_init_gc	Make special initializations of bmap for the	

		case of gcinode.	
3.	nilfs_btree_convert_and_insert	Insert key-record pair in b-tree.	
4.	nilfs_btree_broken_node_block	Verify consistency of btree node that is kept in a buffer.	

B-tree operation table (struct nilfs_bmap_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_lookup	Find a data block or node block.	btree.c
2.	nilfs_btree_lookup_contig	Find a contiguous area of data blocks or node blocks.	
3.	nilfs_btree_insert	Insert a new key-record pair into a b-tree.	
4.	nilfs_btree_delete	Delete a key-record pair from a b-tree.	
5.	nilfs_btree_propagate	Propagate dirty state. Mark the buffers that directly or indirectly refer to the block as dirty.	
6.	nilfs_btree_lookup_dirty_buffers	Find all dirty buffers.	
7.	nilfs_btree_assign	Assign a new block number to a block.	
8.	nilfs_btree_mark	Mark block as dirty.	
9.	nilfs_btree_last_key	<TODO>	
10.	nilfs_btree_check_delete	Check correctness and possibility to execute deletion of a key-record pair from a b-tree.	
11.	nilfs_btree_gather_data	Collect translation items for some count of keys.	

GC inode's bmap operation table (struct nilfs_bmap_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_propagate_gc	Propagate dirty state. Mark the buffers that directly or indirectly refer to the block as dirty.	btree.c
2.	nilfs_btree_lookup_dirty_buffers	Find all dirty buffers.	
3.	nilfs_btree_assign_gc	Assign a new block number to a block.	

The cpfile module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_cpfile_get_checkpoint	Get a checkpoint. It acquires the checkpoint specified by checkpoint number. A new checkpoint will be created if requested checkpoint number equals by the current checkpoint number and create flag is nonzero.	cpfile.c
2.	nilfs_cpfile_put_checkpoint	Put a checkpoint. It releases the checkpoint specified by checkpoint number.	
3.	nilfs_cpfile_delete_checkpoints	Delete checkpoints. It deletes the checkpoints in the period from start checkpoint number to end checkpoint number, excluding end checkpoint number itself. The checkpoints which have been already deleted are ignored.	
4.	nilfs_cpfile_delete_checkpoint	Delete checkpoint.	

5.	nilfs_cpfile_change_cpmode	Change checkpoint mode. It changes the mode of the checkpoint (checkpoint or snapshot mode).	
6.	nilfs_cpfile_is_snapshot	Check that checkpoint specified by checkpoint number is a snapshot, or not.	
7.	nilfs_cpfile_get_stat	Get checkpoint statistics. It returns information about checkpoints.	
8.	nilfs_cpfile_get_cpinfo	<TODO>	
9.	nilfs_cpfile_read	Read or get cpfile inode.	

The dat module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_read	Read or get dat inode.	dat.c
2.	nilfs_dat_translate	Translate a virtual block number to a block number.	
3.	nilfs_dat_prepare_alloc	<TODO>	
4.	nilfs_dat_commit_alloc	<TODO>	
5.	nilfs_dat_abort_alloc	<TODO>	
6.	nilfs_dat_prepare_start	<TODO>	
7.	nilfs_dat_commit_start	<TODO>	
8.	nilfs_dat_prepare_end	<TODO>	
9.	nilfs_dat_commit_end	<TODO>	
10.	nilfs_dat_abort_end	<TODO>	
11.	nilfs_dat_prepare_update	<TODO>	
12.	nilfs_dat_commit_update	<TODO>	
13.	nilfs_dat_abort_update	<TODO>	
14.	nilfs_dat_mark_dirty	<TODO>	
15.	nilfs_dat_freev	Free virtual block numbers.	
16.	nilfs_dat_move	Change a block number.	
17.	nilfs_dat_get_vinfo	<TODO>	

The dir module API:

- Directory inode operation table (struct inode_operations).
- Directory operation table (struct file_operations).
- Directory entry operations.

Directory inode operation table (struct inode_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_create	Called by the open(2) and creat(2) system calls. Only required if you want to support regular files.	namei.c
2.	nilfs_lookup	Called when the VFS needs to look up an inode in a parent directory.	
3.	nilfs_link	Called by the link(2) system call. Only required if you want to support hard links.	

4.	nilfs_unlink	Called by the unlink(2) system call. Only required if you want to support deleting inodes.	
5.	nilfs_symlink	Called by the symlink(2) system call. Only required if you want to support symlinks.	
6.	nilfs_mkdir	Called by the mkdir(2) system call. Only required if you want to support creating subdirectories.	
7.	nilfs_rmdir	Called by the rmdir(2) system call. Only required if you want to support deleting subdirectories.	
8.	nilfs_mknod	Called by the mknod(2) system call to create a device (char, block) inode or a named pipe (FIFO) or socket. Only required if you want to support creating these types of inodes.	
9.	nilfs_rename	Called by the rename(2) system call to rename the object to have the parent and name given by the second inode and dentry.	
10.	nilfs_setattr	Called by the VFS to set attributes for a file. This method is called by chmod(2) and related system calls.	inode.c
11.	nilfs_permission	Called by the VFS to check for access rights on a POSIX-like filesystem.	
12.	nilfs_fiemap	The fiemap call is responsible for defining its set of supported fiemap flags, and calling a helper function on each discovered extent.	

Directory operation table (struct file_operations):

#	NAME	DESCRIPTION	FILE
1.	generic_file_llseek	Called when the VFS needs to move the file position index.	fs/read_write.c
2.	generic_read_dir	Called by read(2) and related system calls.	fs/libfs.c
3.	nilfs_readdir	Called when the VFS needs to read the directory contents.	dir.c
4.	nilfs_ioctl	<TODO>	ioctl.c
5.	nilfs_compat_ioctl	<TODO>	
6.	nilfs_sync_file	<TODO>	file.c

Directory entry operations:

#	NAME	DESCRIPTION	FILE
1.	nilfs_add_link	<TODO>	dir.c
2.	nilfs_inode_by_name	Finds an entry in the specified directory with the wanted name.	
3.	nilfs_make_empty	Set the first fragment of directory.	
4.	nilfs_find_entry	Finds an entry in the specified directory with the wanted name.	
5.	nilfs_delete_entry	Delete a directory entry by merging it with	

		the previous entry.	
6.	nilfs_empty_dir	Check that the specified directory is empty (for rmdir).	
7.	nilfs_dotdot	Get ".." directory entry	
8.	nilfs_set_link	<TODO>	

The direct module API:

- Direct block pointer API.
- Direct block pointer operation table (struct nilfs_bmap_operations).

Direct block pointer API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_init	Make special initializations of bmap for the case of inode with direct block pointers.	direct.c
2.	nilfs_direct_delete_and_convert	<TODO>	

Direct block pointer operation table (struct nilfs_bmap_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_lookup	Get direct pointer on block number.	direct.c
2.	nilfs_direct_lookup_contig	Get contiguous area of direct pointers on block numbers.	
3.	nilfs_direct_insert	<TODO>	
4.	nilfs_direct_delete	<TODO>	
5.	nilfs_direct_propagate	Propagate dirty state. Mark the buffers that directly or indirectly refer to the block as dirty.	
6.	nilfs_direct_assign	Assign a new block number to a block.	
7.	nilfs_direct_last_key	<TODO>	
8.	nilfs_direct_check_insert	Check possibility to do an insert operation in bmap.	
9.	nilfs_direct_gather_data	Collect translation items for some count of keys.	

The file module API:

- File inode operation table (struct node_operations).
- File object operation table (struct file_operations).
- Virtual memory management functions (struct vm_operations_struct).

File inode operation table (struct node_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_truncate	Called by the VFS to change the size of a file. This method is called by the truncate(2) system call and related functionality.	inode.c
2.	nilfs_setattr	Called by the VFS to set attributes for a file. This method is called by chmod(2) and related	

		system calls.	
3.	nilfs_permission	Called by the VFS to check for access rights on a POSIX-like filesystem.	
4.	nilfs_fiemap	The fiemap call is responsible for defining its set of supported fiemap flags, and calling a helper function on each discovered extent.	

File object operation table (struct file_operations):

#	NAME	DESCRIPTION	FILE
1.	generic_file_llseek	Generic llseek implementation for regular files.	fs/read_write.c
2.	do_sync_read	<TODO>	
3.	do_sync_write	<TODO>	
4.	generic_file_aio_read	Generic filesystem read routine. This is the "read()" routine for all filesystems that can use the page cache directly.	mm/filemap.c
5.	generic_file_aio_write	Write data to a file. This function does all the work needed for actually writing data to a file.	
6.	nilfs_ioctl	<TODO>	ioctl.c
7.	nilfs_compat_ioctl	<TODO>	
8.	nilfs_file_mmap	Map file into memory.	file.c
9.	generic_file_open	Called when an inode is about to be open.	fs/open.c
10.	nilfs_sync_file	Write back data in range [start..end] and metadata for file to disk.	file.c
11.	generic_file_splice_read	Splice data from file to a pipe. It reads pages from given file and fill them into a pipe.	fs/splice.c

Virtual memory management functions (struct vm_operations_struct):

#	NAME	DESCRIPTION	FILE
1.	filemap_fault	Read in file data for page fault handling. filemap_fault() is invoked via the vma operations vector for a mapped memory region to read in file data during a page fault.	mm/filemap.c
2.	nilfs_page_mkwrite	Notification that a previously read-only page is about to become writable.	file.c

The gcinode module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_init_gcinode	Initialize GC inode.	gcinode.c
2.	nilfs_gccache_submit_read_data	Add data buffer and submit read request.	
3.	nilfs_gccache_submit_read_node	Add node buffer and submit read request.	
4.	nilfs_gccache_wait_and_mark_dirty	<TODO>	

5.	nilfs_remove_all_gcinodes	Remove all unprocessed gc inodes.	

The ifile module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_ifile_map_inode	Get kernel address of inode item in ifile.	ifile.h
2.	nilfs_ifile_unmap_inode	Unmap memory page that contains inode item.	
3.	nilfs_ifile_create_inode	Create a new disk inode.	ifile.c
4.	nilfs_ifile_delete_inode	Delete a disk inode.	
5.	nilfs_ifile_get_inode_block	Get buffer head on block contains on-disk inode with requested number.	
6.	nilfs_ifile_read	Read or get ifile inode.	

The inode module API:

- Special file inode operation table (struct inode_operations).
- Symlink inode operation table (struct inode_operations).
- Exportable to NFS operation table (struct export_operations).
- Page cache operation table (struct address_space_operations).
- Inode subsystem API.

Special file inode operation table (struct inode_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_setattr	Called by the VFS to set attributes for a file. This method is called by chmod(2) and related system calls.	inode.c
2.	nilfs_permission	Called by the VFS to check for access rights on a POSIX-like filesystem.	

Symlink inode operation table (struct inode_operations):

#	NAME	DESCRIPTION	FILE
1.	generic_readlink	Reading symbolic links.	fs/namei.c
2.	page_follow_link_light	Follow a symbolic link to the inode it points to.	
3.	page_put_link	Release resources allocated by follow_link().	
4.	nilfs_permission	Called by the VFS to check for access rights on a POSIX-like filesystem.	inode.c

Exportable to NFS operation table (struct export_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_encode_fh	Takes a dentry and creates a filehandle fragment which can later be used to find or create a dentry for the same object.	namei.c
2.	nilfs_fh_to_dentry	Given a filehandle fragment, this should find	

		the implied object and create a dentry for it.	
3.	nilfs_fh_to_parent	Given a filehandle fragment, this should find the parent of the implied object and create a dentry for it.	
4.	nilfs_get_parent	Given a filehandle fragment, this should find the parent of the implied object and create a dentry for it.	

Page cache operation table (struct address_space_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_writepage	Called by the VM to write a dirty page to backing store. This may happen for data integrity reasons (i.e. 'sync'), or to free up memory (flush).	inode.c
2.	nilfs_readpage	Called by the VM to read a page from backing store.	
3.	nilfs_writepages	Called by the VM to write out pages associated with the address_space object.	
4.	nilfs_set_page_dirty	Called by the VM to set a page dirty.	
5.	nilfs_readpages	Called by the VM to read pages associated with the address_space object. This is essentially just a vector version of readpage. Instead of just one page, several pages are requested.	
6.	nilfs_write_begin	Called by the generic buffered write code to ask the filesystem to prepare to write len bytes at the given offset in the file.	
7.	nilfs_write_end	After a successful write_begin, and data copy, write_end must be called. len is the original len passed to write_begin, and copied is the amount that was able to be copied.	
8.	block_invalidatepage	Invalidate part or all of a buffer-backed page.	fs/buffer.c
9.	nilfs_direct_IO	Called by the generic read/write routines to perform direct_IO - that is IO requests which bypass the page cache and transfer data directly between the storage and the application's address space.	inode.c
10.	block_is_partially_uptodate	Check whether buffers within a page are uptodate or not.	fs/buffer.c

Inode subsystem API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_inode_add_blocks	<TODO>	inode.c
2.	nilfs_inode_sub_blocks	<TODO>	
3.	nilfs_new_inode	<TODO>	
4.	nilfs_get_block	Get a file block on the filesystem (callback	

		function). This function does not issue actual read request of the specified data block. It is done by VFS.	
5.	nilfs_set_inode_flags	<TODO>	
6.	nilfs_read_inode_common	<TODO>	
7.	nilfs_write_inode_common	<TODO>	
8.	nilfs_ilookup	<TODO>	
9.	nilfs_iget_locked	<TODO>	
10.	nilfs_iget	<TODO>	
11.	nilfs_iget_for_gc	<TODO>	
12.	nilfs_update_inode	<TODO>	
13.	nilfs_evict_inode	<TODO>	
14.	nilfs_load_inode_block	<TODO>	
15.	nilfs_inode_dirty	<TODO>	
16.	nilfs_set_file_dirty	<TODO>	
17.	nilfs_mark_inode_dirty	<TODO>	
18.	nilfs_dirty_inode	<TODO>	

The ioctl module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_ioctl	<TODO>	ioctl.c
2.	nilfs_compat_ioctl	<TODO>	
3.	nilfs_ioctl_prepare_clean_segments	<TODO>	

The mdt module API:

- Metadata file initialization API.
- Metadata file access API.
- Shadow map API.
- Metadata file state management API.

Metadata file initialization API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_init	Initialize metadata file inode object.	mdt.c
2.	nilfs_mdt_set_entry_size	Define metadata file's entry size.	
3.	nilfs_mdt_setup_shadow_map	Setup shadow map and bind it to metadata file.	

Metadata file access API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_get_block	Read or create a buffer on meta data file. It looks up the specified buffer and tries to create a new buffer if create is not zero. On success, the returned buffer is assured to be either existing or formatted using a buffer lock on success.	mdt.c

2.	nilfs_mdt_delete_block	Make a hole on the meta data file.	
3.	nilfs_mdt_forget_block	Discard dirty state and try to remove the page. It clears a dirty flag of the specified buffer, and tries to release the page including the buffer from a page cache.	
4.	nilfs_mdt_mark_block_dirty	Mark a block on the meta data file dirty.	
5.	nilfs_mdt_fetch_dirty	Test that metadata file is dirty.	

Shadow map API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_save_to_shadow_map	Copy bmap and dirty pages to shadow map.	mdt.c
2.	nilfs_mdt_restore_from_shadow_map	Restore dirty pages and bmap state.	
3.	nilfs_mdt_clear_shadow_map	Truncate pages in shadow map caches.	
4.	nilfs_mdt_freeze_buffer	Add buffer to the list of frozen buffers in shadow map.	
5.	nilfs_mdt_get_frozen_buffer	Find frozen copy of the buffer in the list of frozen buffers in shadow map.	

Metadata file state management API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_mark_dirty	Mark metadata file's inode as dirty.	mdt.h
2.	nilfs_mdt_clear_dirty	Unset dirty bit of metadata file's inode state.	
3.	nilfs_mdt_cno	Get actual checkpoint number for metadata file.	
4.	nilfs_mdt_bgl_lock	Get spinlock for blockgroup.	

The page module API:

#	NAME	DESCRIPTION	FILE
1.	set_buffer_nilfs_node	<TODO>	page.h
2.	buffer_nilfs_node	<TODO>	
3.	clear_buffer_nilfs_node	<TODO>	
4.	set_buffer_nilfs_volatile	<TODO>	
5.	buffer_nilfs_volatile	<TODO>	
6.	clear_buffer_nilfs_volatile	<TODO>	
7.	set_buffer_nilfs_checked	<TODO>	
8.	buffer_nilfs_checked	<TODO>	
9.	clear_buffer_nilfs_checked	<TODO>	
10.	set_buffer_nilfs_redirected	<TODO>	
11.	buffer_nilfs_redirected	<TODO>	
12.	clear_buffer_nilfs_redirected	<TODO>	
13.	__nilfs_clear_page_dirty	NILFS2 needs clear_page_dirty() in the following two cases: 1) For B-tree node pages and data pages of the dat/gcdat, NILFS2 clears page dirty flags when it copies back pages from the	page.c

		shadow cache (gcdat->{i_mapping,i_btnode_cache}) to its original cache (dat->{i_mapping,i_btnode_cache}). 2) Some B-tree operations like insertion or deletion may dispose buffers in dirty state, and this needs to cancel the dirty state of their pages.	
14.	nilfs_grab_buffer	<TODO>	
15.	nilfs_forget_buffer	Discard dirty state.	
16.	nilfs_copy_buffer	Copy buffer data and flags.	
17.	nilfs_page_buffers_clean	Check if a page has dirty buffers or not.	
18.	nilfs_page_bug	<TODO>	
19.	nilfs_copy_dirty_pages	<TODO>	
20.	nilfs_copy_back_pages	Copy back pages to original cache from shadow cache.	
21.	nilfs_clear_dirty_pages	<TODO>	
22.	nilfs_mapping_init	<TODO>	
23.	nilfs_page_count_clean_buffers	<TODO>	
24.	nilfs_find_uncommitted_extent	Find extent of uncommitted data.	
25.	nilfs_page_get_nth_block	<TODO>	page.h

The recovery module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_read_super_root_block	Read super root block.	recovery.c
2.	nilfs_search_super_root	Search the latest valid super root. It looks for the latest super-root from a partial segment pointed by the superblock. It sets up struct the_nilfs through this search. It fills nilfs_recovery_info (ri) required for recovery.	
3.	nilfs_salvage_orphan_logs	Salvage logs written after the latest checkpoint.	
4.	nilfs_dispose_segment_list	<TODO>	

The segbuf module API:

#	NAME	DESCRIPTION	FILE
1.	NILFS_LIST_SEGBUF	<TODO>	segbuf.h
2.	NILFS_NEXT_SEGBUF	<TODO>	
3.	NILFS_PREV_SEGBUF	<TODO>	
4.	NILFS_LAST_SEGBUF	<TODO>	
5.	NILFS_FIRST_SEGBUF	<TODO>	
6.	NILFS_SEGBUF_IS_LAST	<TODO>	
7.	NILFS_SEGBUF_FIRST_BH	<TODO>	
8.	NILFS_SEGBUF_NEXT_BH	<TODO>	

9.	NILFS_SEGBUF_BH_IS_LAST	<TODO>	
10.	nilfs_segbuf_new	<TODO>	segbuf.c
11.	nilfs_segbuf_free	<TODO>	
12.	nilfs_segbuf_map	<TODO>	
13.	nilfs_segbuf_map_cont	Map a new log behind a given log.	
14.	nilfs_segbuf_set_next_segnum	<TODO>	
15.	nilfs_segbuf_reset	<TODO>	
16.	nilfs_segbuf_extend_segsum	<TODO>	
17.	nilfs_segbuf_extend_payload	<TODO>	
18.	nilfs_segbuf_fill_in_segsum	Setup segment summary.	
19.	nilfs_segbuf_simplex	<TODO>	segbuf.h
20.	nilfs_segbuf_empty	<TODO>	
21.	nilfs_segbuf_add_segsum_buffer	<TODO>	
22.	nilfs_segbuf_add_payload_buffer	<TODO>	
23.	nilfs_segbuf_add_file_buffer	<TODO>	
24.	nilfs_clear_logs	<TODO>	segbuf.c
25.	nilfs_truncate_logs	<TODO>	
26.	nilfs_write_logs	<TODO>	
27.	nilfs_wait_on_logs	<TODO>	
28.	nilfs_add_checksums_on_logs	Add checksums on the logs.	
29.	nilfs_destroy_logs	<TODO>	segbuf.h

The segment module API:

- Segment constructor logic API.
- Transaction logic API.
- nilfs_sc_file_ops <TODO> (struct nilfs_sc_operations).
- nilfs_sc_dat_ops <TODO> (struct nilfs_sc_operations).
- Data-only logical segment operation table (struct nilfs_sc_operations).

Segment constructor logic API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_relax_pressure_in_lock	<TODO>	segment.c
2.	nilfs_construct_segment	Construct a logical segment.	
3.	nilfs_construct_dsync_segment	Construct a data-only logical segment.	
4.	nilfs_flush_segment	Trigger a segment construction for resource control.	
5.	nilfs_clean_segments	<TODO>	
6.	nilfs_attach_log_writer	Attach log writer. It allocates a log writer object, initializes it, and starts the log writer.	
7.	nilfs_detach_log_writer	Destroy log writer. It kills log writer daemon, frees the log writer object, and destroys list of dirty files.	

Transaction logic API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_transaction_begin	Start indivisible file operations.	segment.c
2.	nilfs_transaction_commit	Commit indivisible file operations.	
3.	nilfs_transaction_abort	<TODO>	
4.	nilfs_set_transaction_flag	<TODO>	nilfs.h
5.	nilfs_test_transaction_flag	<TODO>	
6.	nilfs_doing_gc	<TODO>	
7.	nilfs_doing_construction	<TODO>	

nilfs_sc_file_ops <TODO> (struct nilfs_sc_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_collect_file_data	<TODO>	segment.c
2.	nilfs_collect_file_node	<TODO>	
3.	nilfs_collect_file_bmap	<TODO>	
4.	nilfs_write_file_data_binfo	<TODO>	
5.	nilfs_write_file_node_binfo	<TODO>	

nilfs_sc_dat_ops <TODO> (struct nilfs_sc_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_collect_dat_data	<TODO>	segment.c
2.	nilfs_collect_file_node	<TODO>	
3.	nilfs_collect_dat_bmap	<TODO>	
4.	nilfs_write_dat_data_binfo	<TODO>	
5.	nilfs_write_dat_node_binfo	<TODO>	

Data-only logical segment operation table (struct nilfs_sc_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_collect_file_data	<TODO>	segment.c
2.	nilfs_write_file_data_binfo	<TODO>	

The sufile module API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sufile_get_nsegments	<TODO>	sufile.h
2.	nilfs_sufile_get_ncleansegs	<TODO>	sufile.c
3.	nilfs_sufile_set_alloc_range	<TODO>	
4.	nilfs_sufile_alloc	Allocate a segment.	
5.	nilfs_sufile_mark_dirty	Mark the buffer having a segment usage dirty.	
6.	nilfs_sufile_set_segment_usage	Set usage of a segment.	
7.	nilfs_sufile_get_stat	Get segment usage statistics.	
8.	nilfs_sufile_get_suinfo	<TODO>	
9.	nilfs_sufile_updatev	<TODO>	
10.	nilfs_sufile_update	<TODO>	

11.	nilfs_sufile_do_scrap	<TODO>	
12.	nilfs_sufile_do_free	<TODO>	
13.	nilfs_sufile_do_cancel_free	<TODO>	
14.	nilfs_sufile_do_set_error	<TODO>	
15.	nilfs_sufile_resize	Resize segment array.	
16.	nilfs_sufile_read	Read or get sufile inode.	
17.	nilfs_sufile_scrap	<TODO>	sufile.h
18.	nilfs_sufile_free	<TODO>	
19.	nilfs_sufile_freev	<TODO>	
20.	nilfs_sufile_cancel_freev	<TODO>	
21.	nilfs_sufile_set_error	<TODO>	

The super module API:

- NILFS module and super block management API.
- Superblock operation table (struct super_operations).
- File system description (struct file_system_type).

NILFS module and super block management API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_error	<TODO>	super.c
2.	nilfs_warning	<TODO>	
3.	nilfs_read_super_block	<TODO>	
4.	nilfs_store_magic_and_option	<TODO>	
5.	nilfs_check_feature_compatibility	<TODO>	
6.	nilfs_set_log_cursor	<TODO>	
7.	nilfs_prepare_super	<TODO>	
8.	nilfs_commit_super	<TODO>	
9.	nilfs_cleanup_super	<TODO>	
10.	nilfs_resize_fs	<TODO>	
11.	nilfs_attach_checkpoint	<TODO>	
12.	nilfs_checkpoint_is_mounted	<TODO>	

Superblock operation table (struct super_operations):

#	NAME	DESCRIPTION	FILE
1.	nilfs_alloc_inode	This method is called by inode_alloc() to allocate memory for struct inode and initialize it. If this function is not defined, a simple 'struct inode' is allocated. Normally alloc_inode will be used to allocate a larger structure which contains a 'struct inode' embedded within it.	super.c
2.	nilfs_destroy_inode	This method is called by destroy_inode() to release resources allocated for struct inode.	
3.	nilfs_dirty_inode	This method is called by the VFS to mark	inode.c

		an inode dirty.	
4.	nilfs_evict_inode	<TODO>	
5.	nilfs_put_super	Called when the VFS wishes to free the superblock (i.e. unmount).	super.c
6.	nilfs_sync_fs	Called when VFS is writing out all dirty data associated with a superblock. The second parameter indicates whether the method should wait until the write out has been completed.	
7.	nilfs_freeze	Called when VFS is locking a filesystem and forcing it into a consistent state.	
8.	nilfs_unfreeze	Called when VFS is unlocking a filesystem and making it writable again.	
9.	nilfs_statfs	Called when the VFS needs to get filesystem statistics.	
10.	nilfs_remount	Called when the filesystem is remounted.	
11.	nilfs_show_options	Called by the VFS to show mount options for /proc/<pid>/mounts.	

File system description (struct file_system_type):

#	NAME	DESCRIPTION	FILE
1.	nilfs_mount	The method to call by kernel when a new instance of this file system should be mounted.	super.c

6.5. alloc

Persistent object (dat entry/disk inode) allocator/deallocator.

6.5.1. Summary

<TODO>

6.5.2. Architecture

<TODO>

6.5.3. Structures

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_req	Persistent allocator request and reply.	alloc.h
2.	nilfs_palloc_cache	Persistent object allocator cache.	

6.5.3.1. nilfs_palloc_req

Persistent allocator request and reply:

```
struct nilfs_palloc_req {
        __u64                   pr_entry_nr;
        struct buffer_head      *pr_desc_bh;
        struct buffer_head      *pr_bitmap_bh;
        struct buffer_head      *pr_entry_bh;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
__u64	pr_entry_nr	Entry number (vblocknr or inode number).
struct buffer_head *	pr_desc_bh	Buffer head of the buffer containing block group descriptors.
struct buffer_head *	pr_bitmap_bh	Buffer head of the buffer containing a block group bitmap.
struct buffer_head *	pr_entry_bh	Buffer head of the buffer containing translation entries.
{END OF STRUCTURE}		

6.5.3.2. nilfs_palloc_cache

Persistent object allocator cache:

```
struct nilfs_palloc_cache {
        spinlock_t              lock;
        struct nilfs_bh_assoc   prev_desc;
        struct nilfs_bh_assoc   prev_bitmap;
        struct nilfs_bh_assoc   prev_entry;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
spinlock_t	lock	Lock protecting persistent object allocator cache.
struct nilfs_bh_assoc	prev_desc	Cached pair of block offset and buffer head of the buffer containing block group descriptors.
struct nilfs_bh_assoc	prev_bitmap	Cached pair of block offset and buffer head of the buffer containing a block group bitmap.
struct nilfs_bh_assoc	prev_entry	Cached pair of block offset and buffer head of the buffer containing translation entries.
{END OF STRUCTURE}		

6.5.4. APIs

Persistent allocator API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_init_blockgroup	Initialize private variables for allocator.	alloc.c
2.	nilfs_palloc_prepare_alloc_entry	Prepare to allocate a persistent object.	
3.	nilfs_palloc_commit_alloc_entry	Finish allocation of a persistent object.	
4.	nilfs_palloc_abort_alloc_entry	Cancel allocation of a persistent object.	

Persistent deallocator API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_prepare_free_entry	Prepare to deallocate a persistent object.	alloc.c
2.	nilfs_palloc_commit_free_entry	Finish deallocating a persistent object.	
3.	nilfs_palloc_abort_free_entry	Cancel deallocating a persistent object.	
4.	nilfs_palloc_freev	Deallocate a set of persistent objects.	

Persistent object allocator cache API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_setup_cache	Initialize meta data file's persistent object allocator cache.	alloc.c
2.	nilfs_palloc_clear_cache	Clear meta data file's persistent object allocator cache.	
3.	nilfs_palloc_destroy_cache	Destroy meta data file's persistent object allocator cache.	

6.5.4.1. Persistent Allocator API

6.5.4.1.1. nilfs_palloc_init_blockgroup

Summary: Initialize private variables for allocator.

Declaration:
int nilfs_palloc_init_blockgroup(struct inode *inode,
unsigned entry_size);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, it returns 0. On error, the following negative error code is returned: • [-ENOMEM] - Insufficient memory available.
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in]	unsigned	entry_size	Size of the persistent object.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_set_entry_size	Define metadata file's entry size.	mdt.c
2.	nilfs_palloc_entries_per_group	Get the number of entries per group. The number of entries per group is defined by the number of bits that a bitmap block can maintain.	alloc.h
3.	nilfs_palloc_groups_per_desc_block	Get the number of groups that a descriptor block can maintain.	alloc.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_ifile_read	Read or get ifile inode.	ifile.c
2.	nilfs_dat_read	Read or get dat inode.	dat.c

6.5.4.1.2. nilfs_palloc_prepare_alloc_entry

Summary: Prepare to allocate a persistent object.

Declaration:
int nilfs_palloc_prepare_alloc_entry(struct inode *inode,
struct nilfs_palloc_req *req);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in|out]	struct nilfs_palloc_req *	req	The nilfs_palloc_req structure exchanged for the allocation.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_groups_count	Get maximum number of groups.	alloc.c
2.	nilfs_palloc_group	Get group number and offset from an entry number.	
3.	nilfs_palloc_entries_per_group	Get the number of entries per group. The number of entries per group is defined by the number of bits that a bitmap block can maintain.	alloc.h
4.	nilfs_palloc_groups_per_desc_block	Get the number of groups that a descriptor block can maintain.	alloc.c
5.	nilfs_palloc_get_desc_block	Get buffer head of a group descriptor	

		block.	
6.	nilfs_palloc_block_get_group_desc	Get kernel address of a group descriptor.	
7.	nilfs_palloc_rest_groups_in_desc_block	Get the remaining number of groups in a group descriptor block.	
8.	nilfs_palloc_group_desc_nfrees	Get the number of free entries in a group.	
9.	nilfs_palloc_get_bitmap_block	Get buffer head of a bitmap block.	
10.	nilfs_palloc_find_available_slot	Find available slot in a group.	
11.	nilfs_palloc_group_desc_add_entries	Adjust count of free entries.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_prepare_alloc	<TODO>	dat.c
2.	nilfs_ifile_create_inode	Create a new disk inode.	ifile.c

6.5.4.1.3. nilfs_palloc_commit_alloc_entry

Summary: Finish allocation of a persistent object.

Declaration:
void nilfs_palloc_commit_alloc_entry(struct inode *inode,
struct nilfs_palloc_req *req);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in|out]	struct nilfs_palloc_req *	req	The nilfs_palloc_req structure exchanged for the allocation.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_mark_dirty	Mark meta data file inode as dirty.	mdt.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_commit_alloc	<TODO>	dat.c
2.	nilfs_ifile_create_inode	Create a new disk inode.	ifile.c

6.5.4.1.4. nilfs_palloc_abort_alloc_entry

Summary: Cancel allocation of a persistent object.

Declaration:

void nilfs_palloc_abort_alloc_entry(struct inode *inode,
struct nilfs_palloc_req *req);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in|out]	struct nilfs_palloc_req *	req	The nilfs_palloc_req structure exchanged for the allocation.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_group	Get group number and offset from an entry number.	alloc.c
2.	nilfs_palloc_block_get_group_desc	Get kernel address of a group descriptor.	
3.	nilfs_clear_bit_atomic	ext2_clear_bit_atomic	alloc.h
4.	nilfs_palloc_group_desc_add_entries	Adjust count of free entries.	alloc.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_prepare_alloc	<TODO>	dat.c
2.	nilfs_dat_abort_alloc	<TODO>	dat.c
3.	nilfs_ifile_create_inode	Create a new disk inode.	ifile.c

6.5.4.2. Persistent Deallocator API

6.5.4.2.1. nilfs_palloc_prepare_free_entry

Summary: Prepare to deallocate a persistent object.

Declaration:
int nilfs_palloc_prepare_free_entry(struct inode *inode,
struct nilfs_palloc_req *req);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in|out]	struct nilfs_palloc_req *	req	The nilfs_palloc_req structure exchanged for the allocation.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_group	Get group number and offset from an entry number.	alloc.c
2.	nilfs_palloc_get_desc_block	Get buffer head of a group descriptor block.	
3.	nilfs_palloc_get_bitmap_block	Get buffer head of a bitmap block.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_prepare_end	<TODO>	dat.c
2.	nilfs_ifile_delete_inode	Delete a disk inode.	ifile.c

6.5.4.2.2. nilfs_palloc_commit_free_entry

Summary: Finish deallocating a persistent object.

Declaration:
void nilfs_palloc_commit_free_entry(struct inode *inode,
struct nilfs_palloc_req *req);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in|out]	struct nilfs_palloc_req *	req	The nilfs_palloc_req structure exchanged for the allocation.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_group	Get group number and offset from an entry number.	alloc.c
2.	nilfs_palloc_block_get_group_desc	Get kernel address of a group descriptor.	
3.	nilfs_clear_bit_atomic	ext2_clear_bit_atomic	alloc.h
4.	nilfs_palloc_group_desc_add_entries	Adjust count of free entries.	alloc.c
5.	nilfs_mdt_mark_dirty	Mark meta data file inode as dirty.	mdt.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_commit_free	<TODO>	dat.c
2.	nilfs_ifile_delete_inode	Delete a disk inode.	ifile.c

6.5.4.2.3. nilfs_palloc_abort_free_entry

Summary: Cancel deallocating a persistent object.

Declaration:
void nilfs_palloc_abort_free_entry(struct inode *inode,
struct nilfs_palloc_req *req)

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in|out]	struct nilfs_palloc_req *	req	The nilfs_palloc_req structure exchanged for the allocation.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_abort_end	<TODO>	dat.c
2.	nilfs_ifile_delete_inode	Delete a disk inode.	ifile.c

6.5.4.2.4. nilfs_palloc_freev

Summary: Deallocate a set of persistent objects.

Declaration:
int nilfs_palloc_freev(struct inode *inode,
__u64 *entry_nrs,
size_t nitems);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[in]	struct inode *	inode	Inode of metadata file using this allocator.
[in]	__u64 *	entry_nrs	Array of entry numbers to be deallocated.
[in]	size_t	items	Number of entries stored in array.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_group	Get group number and offset from an entry number.	alloc.c
2.	nilfs_palloc_get_desc_block	Get buffer head of a group descriptor block.	
3.	nilfs_palloc_get_bitmap_block	Get buffer head of a bitmap block.	

4.	nilfs_palloc_block_get_group_desc	Get kernel address of a group descriptor.	
5.	nilfs_palloc_group_is_in	Judge if an entry is in a group.	
6.	nilfs_clear_bit_atomic	ext2_clear_bit_atomic	alloc.h
7.	nilfs_mdt_bgl_lock	Get blockgroup lock.	mdt.h
8.	nilfs_palloc_group_desc_add_entries	Adjust count of free entries.	alloc.c
9.	nilfs_mdt_mark_dirty	Mark meta data file inode as dirty.	mdt.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_freev	Free virtual blocks number.	dat.c

6.5.4.3. Persistent Object Allocator Cache API

6.5.4.3.1. nilfs_palloc_setup_cache

Summary: Initialize metadata file's persistent object allocator cache.

Declaration:
void nilfs_palloc_setup_cache(struct inode *inode,
struct nilfs_palloc_cache *cache);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct inode *	inode	Inode of metadata file.
[in]	struct nilfs_palloc_cache *	cache	Pointer on persistent object allocator cache of concrete metadata file.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_read	Read or get dat inode.	dat.c
2.	nilfs_ifile_read	Read or get ifile inode.	ifile.c

6.5.4.3.2. nilfs_palloc_clear_cache

Summary: Clear metadata file's persistent object allocator cache.

Declaration:
void nilfs_palloc_clear_cache(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct inode *	inode	Inode of metadata file.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_destroy_cache	Destroy meta data file's persistent object allocator cache.	alloc.c
2.	nilfs_mdt_restore_from_shadow_map	Restore dirty pages and bmap state.	mdt.c

6.5.4.3.3. nilfs_palloc_destroy_cache

Summary: Destroy metadata file's persistent object allocator cache.

Declaration:
void nilfs_palloc_destroy_cache(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct inode *	inode	Inode of metadata file.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_clear_inode	<TODO>	inode.c

6.6. bmap

NILFS block mapping.

6.6.1. Summary

<TODO>

6.6.2. Architecture

<TODO>

6.6.3. Structures

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_ptr_req	Request for bmap pointer.	bmap.h
2.	nilfs_bmap_stats	bmap statistics.	
3.	nilfs_bmap_operations	bmap operation table.	
4.	nilfs_bmap	bmap structure.	
5.	nilfs_bmap_store	Shadow copy of bmap state.	

6.6.3.1. nilfs_bmap_ptr_req

Request for bmap pointer:

```
union nilfs_bmap_ptr_req {
        __u64                   bpr_ptr;
        struct nilfs_palloc_req bpr_req;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
__u64	bpr_ptr	bmap pointer.
struct nilfs_palloc_req	bpr_req	Request for persistent allocator.
{END OF STRUCTURE}		

6.6.3.2. nilfs_bmap_stats

bmap statistics:

```
struct nilfs_bmap_stats {
        unsigned int        bs_nblocks;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
unsigned int	bs_nblocks	Number of blocks created or deleted.
{END OF STRUCTURE}		

6.6.3.3. nilfs_bmap_operations

bmap operation table:

```
struct nilfs_bmap_operations {
        int (*bop_lookup)(const struct nilfs_bmap *, __u64, int, __u64 *);
        int (*bop_lookup_contig)(const struct nilfs_bmap *, __u64, __u64 *, unsigned);
        int (*bop_insert)(struct nilfs_bmap *, __u64, __u64);
        int (*bop_delete)(struct nilfs_bmap *, __u64);
        void (*bop_clear)(struct nilfs_bmap *);
        int (*bop_propagate)(struct nilfs_bmap *, struct buffer_head *);
        void (*bop_lookup_dirty_buffers)(struct nilfs_bmap *, struct list_head *);
        int (*bop_assign)(struct nilfs_bmap *, struct buffer_head **, sector_t, union nilfs_binfo *);
        int (*bop_mark)(struct nilfs_bmap *, __u64, int);
        int (*bop_last_key)(const struct nilfs_bmap *, __u64 *);
        int (*bop_check_insert)(const struct nilfs_bmap *, __u64);
        int (*bop_check_delete)(struct nilfs_bmap *, __u64);
        int (*bop_gather_data)(struct nilfs_bmap *, __u64 *, __u64 *, int);
};
```

Meaning of the fields:

TYPE	DESCRIPTION
bop_lookup	Find a data block or node block.
bop_lookup_contig	Find a contiguous area of data blocks or node blocks.
bop_insert	Insert a new key-record pair into a bmap.
bop_delete	Delete a key-record pair from a bmap.
bop_clear	Free resources a bmap holds.
bop_propagate	Propagate dirty state. Mark the buffers that directly or indirectly refer to the block as dirty.
bop_lookup_dirty_buffers	Find all dirty buffers.
bop_assign	Assign a new block number to a block.
bop_mark	Mark block as dirty.
bop_last_key	
bop_check_insert	Check possibility to insert a new key-record pair into a bmap.
bop_check_delete	Check correctness and possibility to execute deletion of a key-record pair from a bmap.
bop_gather_data	Collect translation items for some count of keys.
{END OF STRUCTURE}	

6.6.3.4. nilfs_bmap

bmap structure:

```
struct nilfs_bmap {
        union {
                __u8                            u_flags;
                __le64                          u_data[7];
        } b_u;
        struct rw_semaphore                     b_sem;
        struct inode                            *b_inode;
        const struct nilfs_bmap_operations      *b_ops;
        __u64                                   b_last_allocated_key;
        __u64                                   b_last_allocated_ptr;
        int                                     b_ptr_type;
        int                                     b_state;
        __u16                                   b_nchildren_per_block;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
__le64	b_u.u_data[7]	Raw data from i_bmap field of on-disk inode.
struct rw_semaphore	b_sem	Semaphore protecting bmap during

		operations.
struct inode *	b_inode	Owner of bmap.
const struct nilfs_bmap_operations *	b_ops	bmap operation table.
__u64	b_last_allocated_key	Last allocated key for data block.
__u64	b_last_allocated_ptr	Last allocated ptr for data block.
int	b_ptr_type	Pointer type. The field can have such values: • NILFS_BMAP_PTR_P [0] - physical block number (i.e. LBN) • NILFS_BMAP_PTR_VS [1] - virtual block number (single version) • NILFS_BMAP_PTR_VM [2] - virtual block number (has multiple versions) • NILFS_BMAP_PTR_U [-1] - never perform pointer operations
int	b_state	State. The field can have such values: • 0 • NILFS_BMAP_DIRTY [0x00000001]
__u16	b_nchildren_per_block	Maximum number of child nodes for non-root nodes.
{END OF STRUCTURE}		

6.6.3.5. nilfs_bmap_store

Shadow copy of bmap state:

```
struct nilfs_bmap_store {
        __le64          data[7];
        __u64           last_allocated_key;
        __u64           last_allocated_ptr;
        int             state;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
__le64	data[7]	Raw data from i_bmap field of on-disk inode.
__u64	last_allocated_key	Last allocated key for data block.
__u64	last_allocated_ptr	Last allocated ptr for data block.

int	state	State. The field can have such values: • 0 • NILFS_BMAP_DIRTY [0x00000001]
{END OF STRUCTURE}		

6.6.4. APIs

Block mapping initialization, saving and shadowing API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_init_gc	Initialize nilfs_bmap structure of GC inode.	bmap.c
2.	nilfs_bmap_read	Read a bmap from raw inode.	
3.	nilfs_bmap_write	Write back a bmap to raw inode.	
4.	nilfs_bmap_save	Save bmap in shadow map.	
5.	nilfs_bmap_restore	Restore bmap from shadow map.	

Block mapping modification API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_insert	Insert a new key-record pair into a bmap.	bmap.c
2.	nilfs_bmap_delete	Delete a key-record pair from a bmap.	
3.	nilfs_bmap_truncate	Truncate a bmap to a specified key. It removes key-record pairs whose keys are greater than or equal to key from bmap.	
4.	nilfs_bmap_clear	Free resources a bmap holds.	
5.	nilfs_bmap_assign	Assign the block number to the buffer specified by buffer head.	

Block mapping lookup API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_lookup_at_level	Find a data block or node block. It finds a record whose key matches a key in the block at level of the bmap.	bmap.c
2.	nilfs_bmap_lookup_contig	Find a contiguous region of blocks.	
3.	nilfs_bmap_last_key	Get last valid key in bmap.	
4.	nilfs_bmap_lookup_dirty_buffers	Build the list of all dirty buffers associated with bmap.	
5.	nilfs_bmap_lookup	Find a data block or node block from the level equals to one.	bmap.h

Block mapping state management API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_test_and_clear_dirty	Test and clear a bmap dirty state.	bmap.c
2.	nilfs_bmap_propagate	Propagate dirty state. It marks the buffers that directly or indirectly refer to the block	

		specified by buffer head as dirty.	
3.	nilfs_bmap_mark	Mark the block specified by a key and level as dirty.	

Internal API for bmap, direct and b-tree subsystems:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_get_dat	Return DAT file inode.	bmap.h
2.	nilfs_bmap_prepare_alloc_ptr	Prepare to allocate a persistent object.	
3.	nilfs_bmap_commit_alloc_ptr	Finish allocation of a persistent object.	
4.	nilfs_bmap_abort_alloc_ptr	Cancel allocation of a persistent object.	
5.	nilfs_bmap_prepare_end_ptr	<TODO>	
6.	nilfs_bmap_commit_end_ptr	<TODO>	
7.	nilfs_bmap_abort_end_ptr	<TODO>	
8.	nilfs_bmap_set_target_v	<TODO>	
9.	nilfs_bmap_data_get_key	<TODO>	
10.	nilfs_bmap_find_target_seq	<TODO>	
11.	nilfs_bmap_find_target_in_group	<TODO>	
12.	nilfs_bmap_dirty	Check that bmap is in dirty state.	
13.	nilfs_bmap_set_dirty	Set bmap as dirty.	
14.	nilfs_bmap_clear_dirty	Unset bmap as dirty.	

6.6.4.1. Block Mapping Initialization, Saving and Shadowing API

6.6.4.1.1. nilfs_bmap_init_gc

Summary: Initialize nilfs_bmap structure of GC inode.

Declaration:
void nilfs_bmap_init_gc(struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[in|out]	struct nilfs_bmap *	bmap	Pointer on nilfs_bmap structure that to be initialized.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_init_gc	Initialize nilfs_bmap structure of GC inode.	btree.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_init_gcinode	Initialize GC inode.	gcinode.c

6.6.4.1.2. nilfs_bmap_read

Summary: Read a bmap from raw inode. It initializes the bmap.

Declaration:
int nilfs_bmap_read(struct nilfs_bmap *bmap,
struct nilfs_inode *raw_inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned. On error, the following negative error code is returned: • [-ENOMEM] - Insufficient amount of memory available.
[out]	struct nilfs_bmap *	bmap	Pointer on nilfs_bmap structure that to be initialized.
[in]	struct nilfs_inode *	raw_inode	On-disk inode.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_init	Initialize nilfs_bmap structure for the b-tree case.	btree.c
2.	nilfs_direct_init	Initialize nilfs_bmap structure for the direct block pointer case.	direct.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_new_inode	<TODO>	inode.c
2.	nilfs_read_inode_common	<TODO>	

6.6.4.1.3. nilfs_bmap_write

Summary: Write back a bmap to raw inode. It stores bmap in raw inode.

Declaration:
void nilfs_bmap_write(struct nilfs_bmap *bmap,
struct nilfs_inode *raw_inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[in|out]	struct nilfs_bmap *	bmap	Pointer on nilfs_bmap structure that to be written.
[out]	struct nilfs_inode *	raw_inode	On-disk inode.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_fill_in_file_bmap	<TODO>	segment.c
2.	nilfs_write_inode_common	<TODO>	inode.c

6.6.4.1.4. nilfs_bmap_save

Summary: Save bmap in shadow map.

Declaration:
```
void nilfs_bmap_save(const struct nilfs_bmap *bmap,
                     struct nilfs_bmap_store *store);
```

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	const struct nilfs_bmap *	bmap	Pointer on nilfs_bmap structure that to be saved in shadow map.
[out]	struct nilfs_bmap_store *	store	Pointer on shadow copy of bmap state.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_save_to_shadow_map	Copy bmap and dirty pages to shadow map.	mdt.c

6.6.4.1.5. nilfs_bmap_restore

Summary: Restore bmap from shadow map.

Declaration:
```
void nilfs_bmap_restore(struct nilfs_bmap *bmap,
                        const struct nilfs_bmap_store *store);
```

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	struct nilfs_bmap *	bmap	Pointer on nilfs_bmap structure that to be restored from shadow map.
[in]	const struct nilfs_bmap_store *	store	Pointer on shadow copy of bmap state.

Called in:

#	NAME	DESCRIPTION	FILE

1.	nilfs_mdt_restore_from_shadow_map	Restore dirty pages and bmap state.	mdt.c

6.6.4.2. Block Mapping Modification API

6.6.4.2.1. nilfs_bmap_insert

Summary: Insert a new key-record pair into a bmap. It inserts the new key-record pair specified by key and record into bmap.

Declaration:
int nilfs_bmap_insert(struct nilfs_bmap *bmap,
unsigned long key,
unsigned long rec);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned. On error, one of the following negative error codes is returned: • [-EIO] - I/O error. • [-ENOMEM] - Insufficient amount of memory available. • [-EEXIST] - A record associated with key already exists.
[in|out]	struct nilfs_bmap *	bmap	Bmap that to be kept a new key-record pair.
[in]	unsigned long	key	Inserted key.
[in]	unsigned long	rec	Inserted record.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_do_insert	Auxiliary function that do real insert.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_insert_new_block		mdt.c
2.	nilfs_get_block	Get a file block on the filesystem (callback function). This function does not issue actual read request of the specified data block. It is done by VFS.	inode.c

6.6.4.2.2. nilfs_bmap_delete

Summary: Delete a key-record pair from a bmap.

Declaration:
int nilfs_bmap_delete(struct nilfs_bmap *bmap,
unsigned long key);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned. On error, one of the following negative error codes is returned: • [-EIO] - I/O error. • [-ENOMEM] - Insufficient amount of memory available. • [-ENOENT] - A record associated with the key does not exist.
[in|out]	struct nilfs_bmap *	bmap	Bmap from which the key-record pair to be deleted.
[in]	unsigned long	key	Deleted key.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_do_delete	Auxiliary function that do real deletion.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_delete_block	Make a hole on the meta data file.	mdt.c

6.6.4.2.3. nilfs_bmap_truncate

Summary: Truncate a bmap to a specified key. It removes key-record pairs whose keys are greater than or equal to key from bmap.

Declaration:
int nilfs_bmap_truncate(struct nilfs_bmap *bmap,
unsigned long key);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned. On

			error, one of the following negative error codes is returned: • [-EIO] - I/O error. • [-ENOMEM] - Insufficient amount of memory available.
[in|out]	struct nilfs_bmap *	bmap	Bmap from which key-record pairs to be deleted.
[in]	unsigned long	key	A key from which key-record pairs to be deleted.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_do_truncate	Auxiliary function that do real truncation.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_truncate_bmap	<TODO>	inode.c

6.6.4.2.4. nilfs_bmap_clear

Summary: Free resources a bmap holds.

Declaration:
void nilfs_bmap_clear(struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[in|out]	struct nilfs_bmap *	bmap	Bmap which resources to be freed.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_clear	Callback function.	bmap.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_clear_inode	<TODO>	inode.c

6.6.4.2.5. nilfs_bmap_assign

Summary: Assign the block number to the buffer specified by buffer head.

Declaration:
int nilfs_bmap_assign(struct nilfs_bmap *bmap,
struct buffer_head **bh,
unsigned long blocknr,
union nilfs_binfo *binfo);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned and the buffer head of a newly create buffer and the block information associated with the buffer are stored in the place pointed by bh and binfo, respectively. On error, one of the following negative error codes is returned: • [-EIO] - I/O error. • [-ENOMEM] - Insufficient amount of memory available.
[in]	struct nilfs_bmap *	bmap	Pointer on bmap.
[out]	struct buffer_head **	bh	Pointer to buffer head.
[in]	unsigned long	blocknr	Block number.
[out]	union nilfs_binfo *	binfo	Block information.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_assign	Callback function.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_segctor_update_payload_blocknr	<TODO>	segment.c

6.6.4.3. Block Mapping Lookup API

6.6.4.3.1. nilfs_bmap_lookup_at_level

Summary: Find a data block or node block. It finds a record whose key matches a key in the block at level of the bmap.

Declaration:
int nilfs_bmap_lookup_at_level(struct nilfs_bmap *bmap,
__u64 key,

int level,
__u64 *ptrp);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned and the record associated with key is stored in the place pointed by ptrp. On error, one of the following negative error codes is returned: • [-EIO] - I/O error. • [-ENOMEM] - Insufficient amount of memory available. • [-ENOENT] - A record associated with a key does not exist.
[in]	struct nilfs_bmap *	bmap	Pointer on bmap.
[in]	__u64	key	Key value.
[in]	int	level	Level in b-tree.
[out]	__u64 *	ptrp	Place to store the value associated to key.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_lookup	Callback function.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	
3.	nilfs_dat_translate	Translate a virtual block number to a block number.	dat.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_lookup	Find a data block or node block from the level equals to one.	bmap.h
2.	nilfs_ioctl_do_get_bdescs	<TODO>	ioctl.c
3.	nilfs_ioctl_mark_blocks_dirty	<TODO>	

6.6.4.3.2. nilfs_bmap_lookup_contig

Summary: Find a contiguous region of blocks.

Declaration:
int nilfs_bmap_lookup_contig(struct nilfs_bmap *bmap,
__u64 key,
__u64 *ptrp,

unsigned maxblocks);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[in]	struct nilfs_bmap *	bmap	Pointer on bmap.
[in]	__u64	key	Key value.
[out]	__u64 *	ptrp	Place to store the value associated to key.
[in]	unsigned	maxblocks	Maximum number of blocks in contiguous region.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_lookup_contig	Callback function.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_get_block	Get a file block on the filesystem (callback function). This function does not issue actual read request of the specified data block. It is done by VFS.	inode.c
2.	nilfs_fiemap	<TODO>	

6.6.4.3.3. nilfs_bmap_last_key

Summary: Get last valid key in bmap.

Declaration:
int nilfs_bmap_last_key(struct nilfs_bmap *bmap,
unsigned long *key);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[in]	struct nilfs_bmap *	bmap	Pointer on bmap.
[out]	unsigned long *	key	Pointer on found last key.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_last_key	Callback function.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in	

		special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_truncate_bmap	<TODO>	inode.c

6.6.4.3.4. nilfs_bmap_lookup_dirty_buffers

Summary: Build the list of all dirty buffers associated with bmap.

Declaration:
void nilfs_bmap_lookup_dirty_buffers(struct nilfs_bmap *bmap,
struct list_head *listp);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct nilfs_bmap *	bmap	Pointer on bmap.
[out]	struct list_head *	listp	Pointer to buffer head list.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_lookup_dirty_buffers	Callback function.	bmap.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_segctor_scan_file	<TODO>	segment.c

6.6.4.3.5. nilfs_bmap_lookup

Summary: Find a data block or node block from the level equals to one.

Declaration:
int nilfs_bmap_lookup(struct nilfs_bmap *bmap,
__u64 key,
__u64 *ptr);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned and the record associated with key is stored in the place pointed by ptrp. On error, one of the following negative error codes is returned: • [-EIO] - I/O error.

			• [-ENOMEM] - Insufficient amount of memory available. • [-ENOENT] - A record associated with a key does not exist.
[in]	struct nilfs_bmap *	bmap	Pointer on bmap.
[in]	__u64	key	Key value.
[out]	__u64 *	ptrp	Place to store the value associated to key.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_lookup_at_level	Find a data block or node block. It finds a record whose key matches a key in the block at level of the bmap.	bmap.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_submit_block	<TODO>	mdt.c

6.6.4.4. Block Mapping State Management API

6.6.4.4.1. nilfs_bmap_test_and_clear_dirty

Summary: Test and clear a bmap dirty state.

Declaration:
int nilfs_bmap_test_and_clear_dirty(struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	1 is returned if bmap is dirty, or 0 if clear.
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_dirty	Check that bmap is marked as dirty.	bmap.h
2.	nilfs_bmap_clear_dirty	Mark bmap as dirty.	bmap.h

Called in:

#	NAME	DESCRIPTION	FILE

1.	nilfs_mdt_fetch_dirty	Check that metadata file inode is marked as dirty and set it as dirty otherwise.	mdt.c

6.6.4.4.2. nilfs_bmap_propagate

Summary: Propagate dirty state. It marks the buffers that directly or indirectly refer to the block specified by buffer head as dirty.

Declaration:
int nilfs_bmap_propagate(struct nilfs_bmap *bmap,
struct buffer_head *bh);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned. On error, one of the following negative error codes is returned: • [-EIO] - I/O error. • [-ENOMEM] - Insufficient amount of memory available.
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	struct buffer_head *	bh	Buffer head.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_propagate	Callback function.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_collect_file_data	<TODO>	segment.c
2.	nilfs_collect_file_node	<TODO>	
3.	nilfs_collect_dat_data	<TODO>	

6.6.4.4.3. nilfs_bmap_mark

Summary: Mark the block specified by a key and level as dirty.

Declaration:
int nilfs_bmap_mark(struct nilfs_bmap *bmap,
__u64 key,
int level);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned. On error, one of the following negative error codes is returned. • [-EIO] - I/O error. • [-ENOMEM] - Insufficient amount of memory available.
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	__u64	key	Key value.
[in]	int	level	Level in b-tree.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	bop_mark	Callback function.	bmap.c
2.	nilfs_bmap_convert_error	Auxiliary function for conversion errors in special bmap errors.	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_ioctl_mark_blocks_dirty	<TODO>	ioctl.c

6.6.4.5. Internal API for bmap, Direct and B-tree Subsystems

6.6.4.5.1. nilfs_bmap_get_dat

Summary: Return DAT file inode.

Declaration:
struct inode *nilfs_bmap_get_dat(const struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	struct inode *	[return value]	DAT file inode.
[in]	const struct nilfs_bmap *	bmap	Pointer on bmap structure.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_lookup_at_level	Find a data block or node block. It finds a record whose key matches a key in the block at level of the bmap.	bmap.c
2.	nilfs_bmap_find_target_in_group	<TODO>	
3.	nilfs_direct_lookup_contig	<TODO>	direct.c
4.	nilfs_direct_insert	<TODO>	

5.	nilfs_direct_delete	<TODO>	
6.	nilfs_direct_propagate	<TODO>	
7.	nilfs_direct_assign_v	<TODO>	
8.	nilfs_btree_lookup_contig	<TODO>	btree.c
9.	nilfs_btree_prepare_insert	<TODO>	
10.	nilfs_btree_commit_insert	<TODO>	
11.	nilfs_btree_delete	<TODO>	
12.	nilfs_btree_prepare_convert_and_insert	<TODO>	
13.	nilfs_btree_commit_convert_and_insert	<TODO>	
14.	nilfs_btree_propagate_v	<TODO>	
15.	nilfs_btree_propagate_gc	<TODO>	
16.	nilfs_btree_assign_v	<TODO>	
17.	nilfs_btree_assign_gc	<TODO>	

6.6.4.5.2. nilfs_bmap_prepare_alloc_ptr

Summary: <TODO>.

Declaration:
int nilfs_bmap_prepare_alloc_ptr(struct nilfs_bmap *bmap,
union nilfs_bmap_ptr_req *req,
struct inode *dat);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[out]	union nilfs_bmap_ptr_req *	req	Request for bmap ptr.
[in]	struct inode *	dat	DAT file inode.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_prepare_alloc	<TODO>	dat.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_insert	<TODO>	direct.c
2.	nilfs_btree_prepare_insert	<TODO>	btree.c
3.	nilfs_btree_prepare_convert_and_insert	<TODO>	

6.6.4.5.3. nilfs_bmap_commit_alloc_ptr

Summary: <TODO>.

Declaration:
void nilfs_bmap_commit_alloc_ptr(struct nilfs_bmap *bmap,
union nilfs_bmap_ptr_req *req,
struct inode *dat);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	union nilfs_bmap_ptr_req *	req	Request for bmap ptr.
[in]	struct inode *	dat	DAT file inode.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_commit_alloc	<TODO>	dat.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_insert	<TODO>	direct.c
2.	nilfs_btree_commit_insert	<TODO>	btree.c
3.	nilfs_btree_commit_convert_and_insert	<TODO>	

6.6.4.5.4. nilfs_bmap_abort_alloc_ptr

Summary: <TODO>.

Declaration:
void nilfs_bmap_abort_alloc_ptr(struct nilfs_bmap *bmap,
union nilfs_bmap_ptr_req *req,
struct inode *dat);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	union nilfs_bmap_ptr_req *	req	Request for bmap ptr.
[in]	struct inode *	dat	DAT file inode.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_abort_alloc	<TODO>	dat.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_prepare_insert	<TODO>	btree.c

2.	nilfs_btree_prepare_convert_and_insert	<TODO>	

6.6.4.5.5. nilfs_bmap_prepare_end_ptr

Summary: <TODO>.

Declaration:
int nilfs_bmap_prepare_end_ptr(struct nilfs_bmap *bmap,
union nilfs_bmap_ptr_req *req,
struct inode *dat);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	union nilfs_bmap_ptr_req *	req	Request for bmap ptr.
[in]	struct inode *	dat	DAT file inode.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_prepare_end	<TODO>	dat.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_delete	<TODO>	direct.c
2.	nilfs_btree_prepare_delete	<TODO>	btree.c

6.6.4.5.6. nilfs_bmap_commit_end_ptr

Summary: <TODO>.

Declaration:
void nilfs_bmap_commit_end_ptr(struct nilfs_bmap *bmap,
union nilfs_bmap_ptr_req *req,
struct inode *dat);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	union nilfs_bmap_ptr_req *	req	Request for bmap ptr.
[in]	struct inode *	dat	DAT file inode.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_commit_end	<TODO>	dat.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_delete	<TODO>	direct.c
2.	nilfs_btree_commit_delete	<TODO>	btree.c

6.6.4.5.7. nilfs_bmap_abort_end_ptr

Summary: <TODO>.

Declaration:
void nilfs_bmap_abort_end_ptr(struct nilfs_bmap *bmap,
union nilfs_bmap_ptr_req *req,
struct inode *dat);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	union nilfs_bmap_ptr_req *	req	Request for bmap ptr.
[in]	struct inode *	dat	DAT file inode.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_abort_end	<TODO>	dat.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_btree_prepare_delete	<TODO>	btree.c

6.6.4.5.8. nilfs_bmap_set_target_v

Summary: <TODO>.

Declaration:
void nilfs_bmap_set_target_v(struct nilfs_bmap *bmap,
__u64 key,
__u64 ptr);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	struct nilfs_bmap *	bmap	Pointer on bmap structure.

[in]	__u64	key	Key for data block.
[in]	__u64	ptr	Ptr for data block.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_insert	<TODO>	direct.c
2.	nilfs_btree_commit_insert	<TODO>	btree.c
3.	nilfs_btree_commit_convert_and_insert	<TODO>	

6.6.4.5.9. nilfs_bmap_data_get_key

Summary: <TODO>.

Declaration:
```
__u64 nilfs_bmap_data_get_key(const struct nilfs_bmap *bmap,
                              const struct buffer_head *bh);
```

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	__u64	[return value]	
[in]	const struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	const struct buffer_head *	bh	Buffer head.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_propagate	<TODO>	direct.c
2.	nilfs_direct_assign	<TODO>	
3.	nilfs_btree_propagate	<TODO>	btree.c
4.	nilfs_btree_assign	<TODO>	
5.	nilfs_btree_assign_gc	<TODO>	

6.6.4.5.10. nilfs_bmap_find_target_seq

Summary: <TODO>.

Declaration:
```
__u64 nilfs_bmap_find_target_seq(const struct nilfs_bmap *bmap,
                                 __u64 key);
```

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	__u64	[return value]	Ptr for data block.
[in]	const struct nilfs_bmap *	bmap	Pointer on bmap structure.
[in]	__u64	key	Key for data block.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_keydiff_abs	<TODO>	bmap.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_find_target_v	<TODO>	direct.c
2.	nilfs_btree_find_target_v	<TODO>	btree.c

6.6.4.5.11. nilfs_bmap_find_target_in_group

Summary: <TODO>.

Declaration:
__u64 nilfs_bmap_find_target_in_group(const struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	__u64	[return value]	
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_get_dat	Return DAT file inode.	bmap.c
2.	nilfs_palloc_entries_per_group	Get the number of entries per group. The number of entries per group is defined by the number of bits that a bitmap block can maintain.	alloc.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_find_target_v	<TODO>	direct.c
2.	nilfs_btree_find_target_v	<TODO>	btree.c

6.6.4.5.12. nilfs_bmap_dirty

Summary: Check that bmap is in dirty state.

Declaration:
int nilfs_bmap_dirty(const struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	TRUE [1] – bmap is dirty; FALSE [0] – bmap is untouched.
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_test_and_clear_dirty	Test and clear a bmap dirty state.	bmap.c
2.	nilfs_direct_insert	<TODO>	direct.c
3.	nilfs_btree_commit_insert	<TODO>	btree.c
4.	nilfs_btree_commit_delete	<TODO>	
5.	nilfs_btree_commit_convert_and_insert	<TODO>	
6.	nilfs_btree_mark	<TODO>	

6.6.4.5.13. nilfs_bmap_set_dirty

Summary: Set bmap as dirty.

Declaration:
void nilfs_bmap_set_dirty(struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_direct_insert	<TODO>	direct.c
2.	nilfs_btree_commit_insert	<TODO>	btree.c
3.	nilfs_btree_commit_delete	<TODO>	
4.	nilfs_btree_commit_convert_and_insert	<TODO>	
5.	nilfs_btree_mark	<TODO>	

6.6.4.5.14. nilfs_bmap_clear_dirty

Summary: Unset bmap as dirty.

Declaration:
void nilfs_bmap_clear_dirty(struct nilfs_bmap *bmap);

Parameters:

	TYPE	NAME	DESCRIPTION
[in]	struct nilfs_bmap *	bmap	Pointer on bmap structure.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_test_and_clear_dirty	Test and clear a bmap dirty state.	bmap.c

6.7. mdt

NILFS metadata file functionality.

6.7.1. Summary

<TODO>

6.7.2. Architecture

<TODO>

6.7.3. Structures

#	NAME	DESCRIPTION	FILE
1.	nilfs_shadow_map	Shadow mapping of metadata file.	mdt.h
2.	nilfs_mdt_info	On-memory private data of meta data file.	

6.7.3.1. nilfs_shadow_map

Shadow mapping of metadata file:

```
struct nilfs_shadow_map {
        struct nilfs_bmap_store     bmap_store;
        struct address_space        frozen_data;
        struct address_space        frozen_btnodes;
        struct list_head            frozen_buffers;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
struct nilfs_bmap_store	bmap_store	Shadow copy of bmap state.
struct address_space	frozen_data	Shadowed dirty data pages.
struct address_space	frozen_btnodes	Shadowed dirty B-Tree nodes' pages.
struct list_head	frozen_buffers	Frozen buffers’ list of shadow map.
{END OF STRUCTURE}		

6.7.3.2. nilfs_mdt_info

On-memory private data of metadata file:

```
struct nilfs_mdt_info {
```

```
        struct rw_semaphore         mi_sem;
        struct blockgroup_lock      *mi_bgl;
        unsigned                    mi_entry_size;
        unsigned                    mi_first_entry_offset;
        unsigned long               mi_entries_per_block;
        struct nilfs_palloc_cache   *mi_palloc_cache;
        struct nilfs_shadow_map     *mi_shadow;
        unsigned long               mi_blocks_per_group;
        unsigned long               mi_blocks_per_desc_block;
};
```

Meaning of the fields:

TYPE	NAME	DESCRIPTION
struct rw_semaphore	mi_sem	Reader/writer semaphore for meta data operations.
struct blockgroup_lock *	mi_bgl	Per-blockgroup locking.
unsigned	mi_entry_size	Size of an entry.
unsigned	mi_first_entry_offset	Offset to the first entry.
unsigned long	mi_entries_per_block	Number of entries in a block.
struct nilfs_palloc_cache *	mi_palloc_cache	Persistent object allocator cache.
struct nilfs_shadow_map *	mi_shadow	Shadow of bmap and page caches.
unsigned long	mi_blocks_per_group	Number of blocks in a group.
unsigned long	mi_blocks_per_desc_block	Number of blocks per descriptor block.
{END OF STRUCTURE}		

6.7.4. APIs

Metadata file initialization API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_init	Initialize metadata file inode object.	mdt.c
2.	nilfs_mdt_set_entry_size	Define metadata file's entry size.	
3.	nilfs_mdt_setup_shadow_map	Setup shadow map and bind it to metadata file.	

Metadata file access API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_get_block	Read or create a buffer on meta data file. It looks up the specified buffer and tries to create a new buffer if create is not zero. On success, the returned buffer is assured to be either existing or formatted using a buffer lock on success.	mdt.c
2.	nilfs_mdt_delete_block	Make a hole on the meta data file.	
3.	nilfs_mdt_forget_block	Discard dirty state and try to remove the page. It clears a dirty flag of the specified buffer, and tries to release the page including the buffer from a page cache.	
4.	nilfs_mdt_mark_block_dirty	Mark a block on the meta data file dirty.	

5.	nilfs_mdt_fetch_dirty	Test that metadata file is dirty.	

Shadow map API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_save_to_shadow_map	Copy bmap and dirty pages to shadow map.	mdt.c
2.	nilfs_mdt_restore_from_shadow_map	Restore dirty pages and bmap state.	
3.	nilfs_mdt_clear_shadow_map	Truncate pages in shadow map caches.	
4.	nilfs_mdt_freeze_buffer	Add buffer to the list of frozen buffers in shadow map.	
5.	nilfs_mdt_get_frozen_buffer	Find frozen copy of the buffer in the list of frozen buffers in shadow map.	

Metadata file state management API:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_mark_dirty	Mark metadata file's inode as dirty.	mdt.h
2.	nilfs_mdt_clear_dirty	Unset dirty bit of metadata file's inode state.	
3.	nilfs_mdt_cno	Get actual checkpoint number for metadata file.	
4.	nilfs_mdt_bgl_lock	Get spinlock for blockgroup.	

6.7.4.1. Metadata File Initialization API

6.7.4.1.1. nilfs_mdt_init

Summary: Initialize metadata file inode object.

Declaration:

```
int nilfs_mdt_init(struct inode *inode,
                   gfp_t gfp_mask,
                   size_t objsz);
```

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, it returns 0. On error, the following negativeerror code is returned: • [-ENOMEM] - Insufficient memory available.
[out]	struct inode *	inode	Metadata file inode to be initialized.
[in]	gfp_t	gfp_mask	Define GFP flags for metadata inode's pages in page cache.
[in]	size_t	objsz	Object size in bytes. The object is a specimen of concrete metadata file's info structure.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sufile_read	Read or get sufile inode.	sufile.c
2.	nilfs_ifile_read	Read or get ifile inode.	ifile.c
3.	nilfs_cpfile_read	Read or get cpfile inode.	cpfile.c
4.	nilfs_dat_read	Read or get dat file inode.	dat.c

6.7.4.1.2. nilfs_mdt_set_entry_size

Summary: Define metadata file's entry size.

Declaration:
void nilfs_mdt_set_entry_size(struct inode *inode,
unsigned entry_size,
unsigned header_size);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	struct inode *	inode	Metadata file inode to be initialized.
[in]	unsigned	entry_size	Metadata file's entry size in bytes.
[in]	unsigned	header_size	Metadata file's header size in bytes.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sufile_read	Read or get sufile inode.	sufile.c
2.	nilfs_cpfile_read	Read or get cpfile inode.	cpfile.c
3.	nilfs_palloc_init_blockgroup	Initialize private variables for allocator.	alloc.c

6.7.4.1.3. nilfs_mdt_setup_shadow_map

Summary: Setup shadow map and bind it to metadata file.

Declaration:
int nilfs_mdt_setup_shadow_map(struct inode *inode,
struct nilfs_shadow_map *shadow);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Error code (0 - sucess).
[out]	struct inode *	inode	Metadata file inode to be initialized.

[in|out]	struct nilfs_shadow_map *	shadow	Shadow mapping.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mapping_init	<TODO>	page.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_read	Read or get dat file inode.	dat.c

6.7.4.2. Metadata File Access API

6.7.4.2.1. nilfs_mdt_get_block

Summary: Read or create a buffer on meta data file. It looks up the specified buffer and tries to create a new buffer if create is not zero. On success, the returned buffer is assured to be either existing or formatted using a buffer lock on success.

Declaration:

```
int nilfs_mdt_get_block(struct inode *inode,
                        unsigned long blkoff,
                        int create,
                        void (*init_block)(struct inode *, struct buffer_head *, void *),
                        struct buffer_head **out_bh);
```

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, it returns 0. On error, the following negative error code is returned: • [-ENOMEM] - Insufficient memory available. • [-EIO] - I/O error. • [-ENOENT] - The specified block does not exist (hole block). • [-EROFS] - Read only filesystem (for create mode).
[in]	struct inode *	inode	Inode of the meta data file.
[in]	unsigned long	blkoff	Block offset.
[in]	int	create	Create flag.
[in]	void (*)(struct inode *, struct buffer_head *, void *)	init_block	Callback initializer used for newly allocated block.
[out]	struct buffer_head **	out_bh	Output of a pointer to the

			buffer_head.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_read_block	<TODO>	mdt.c
2.	nilfs_mdt_create_block	<TODO>	

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sufile_get_header_block	<TODO>	sufile.c
2.	nilfs_sufile_get_segment_usage_block	<TODO>	
3.	nilfs_sufile_updatev	Modify multiple segment usages at a time.	
4.	nilfs_palloc_get_block	Get buffer head of metadata file's block.	alloc.c
5.	nilfs_cpfile_get_header_block	<TODO>	cpfile.c
6.	nilfs_cpfile_get_checkpoint_block	<TODO>	

6.7.4.2.2. nilfs_mdt_delete_block

Summary: Make a hole on the meta data file.

Declaration:
int nilfs_mdt_delete_block(struct inode *inode,
unsigned long block);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, zero is returned. On error, one of the following negative error code is returned: • [-ENOMEM] - Insufficient memory available. • [-EIO] - I/O error.
[in|out]	struct inode *	inode	Inode of the meta data file.
[in]	unsigned long	block	Block offset.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_delete	Delete a key-record pair from a bmap.	bmap.c
2.	nilfs_mdt_mark_dirty	Mark metadata file's inode as dirty.	mdt.h
3.	nilfs_mdt_forget_block	Discard dirty state and try to remove the page.	mdt.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sufile_delete_segment_usage_block	<TODO>	sufile.c
2.	nilfs_cpfile_delete_checkpoint_block	<TODO>	cpfile.c

6.7.4.2.3. nilfs_mdt_forget_block

Summary: Discard dirty state and try to remove the page.

Declaration:
int nilfs_mdt_forget_block(struct inode *inode,
unsigned long block);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, 0 is returned. On error, one of the following negative error code is returned: • [-EBUSY] – A page has an active buffer. • [-ENOENT] - Page cache has no page addressed by the offset.
[in]	struct inode *	inode	Inode of the meta data file.
[in]	unsigned long	block	Block offset.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_page_get_nth_block	<TODO>	page.h
2.	nilfs_forget_buffer	Discard dirty state.	page.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_delete_block	Make a hole on the meta data file.	mdt.c

6.7.4.2.4. nilfs_mdt_mark_block_dirty

Summary: Mark a block on the meta data file dirty.

Declaration:
int nilfs_mdt_mark_block_dirty(struct inode *inode,
unsigned long block);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, it returns 0. On error, the following negative error code is returned: • [-ENOMEM] - Insufficient memory available. • [-EIO] - I/O error. • [-ENOENT] - The specified block does not exist (hole block).
[in|out]	struct inode *	inode	Inode of the meta data file.
[in]	unsigned long	block	Block offset.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_mdt_read_block	<TODO>	mdt.c
2.	nilfs_mdt_mark_dirty	Mark metadata file's inode as dirty.	mdt.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_ioctl_mark_block_dirty	<TODO>	ioctl.c

6.7.4.2.5. nilfs_mdt_fetch_dirty

Summary: Test that metadata file is dirty.

Declaration:
int nilfs_mdt_fetch_dirty(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	Return 1 if metadata file is dirty, otherwise return 0.
[in|out]	struct inode *	inode	Inode of the meta data file.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_bmap_test_and_clear_dirty	Test and clear a bmap dirty state.	bmap.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_test_metadata_dirty	Test that ifile, cpfile, sufile and DAT file are dirty.	segment.c

6.7.4.3. Shadow Map API

6.7.4.3.1. nilfs_mdt_save_to_shadow_map

Summary: Copy bmap and dirty pages to shadow map.

Declaration:
int nilfs_mdt_save_to_shadow_map(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, it returns 0. On error, the following negative error code is returned: • [-ENOMEM] - Insufficient memory available.
[in|out]	struct inode *	inode	Inode of the meta data file.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_copy_dirty_pages	<TODO>	page.c
2.	nilfs_bmap_save	Save bmap in shadow map.	bmap.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_clean_segments	<TODO>	segment.c

6.7.4.3.2. nilfs_mdt_restore_from_shadow_map

Summary: Restore dirty pages and bmap state.

Declaration:
void nilfs_mdt_restore_from_shadow_map(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[in|out]	struct inode *	inode	Inode of the meta data file.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_clear_cache	Clear meta data file's persistent object allocator cache.	alloc.c

2.	nilfs_clear_dirty_pages	<TODO>	page.c
3.	nilfs_copy_back_pages	Copy back pages to original cache from shadow cache.	
4.	nilfs_bmap_restore	Restore bmap from shadow map.	bmap.c

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_clean_segments	<TODO>	segment.c

6.7.4.3.3. nilfs_mdt_clear_shadow_map

Summary: Truncate pages in shadow map caches.

Declaration:
void nilfs_mdt_clear_shadow_map(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[in|out]	struct inode *	inode	Inode of the meta data file.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_release_frozen_buffers	Delete items from frozen buffers' list of shadow map and drop ref-count of buffers.	mdt.c

6.7.4.3.4. nilfs_mdt_freeze_buffer

Summary: Add buffer to the list of frozen buffers in shadow map.

Declaration:
int nilfs_mdt_freeze_buffer(struct inode *inode,
struct buffer_head *bh);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	int	[return value]	On success, it returns 0. On error, the following negative error code is returned: • [-ENOMEM] - Insufficient memory available.
[in|out]	struct inode *	inode	Inode of the meta data file.
[in|out]	struct buffer_head *	bh	Buffer head on buffer to be freeze.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_page_get_nth_block	<TODO>	page.h
2.	nilfs_copy_buffer	Copy buffer data and flags.	page.c
3.	set_buffer_nilfs_redirected	Set buffer head in the state of redirection to the copy.	page.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_move	Change a block number.	dat.c

6.7.4.3.5. nilfs_mdt_get_frozen_buffer

Summary: Find frozen copy of the buffer in the list of frozen buffers in shadow map.

Declaration:
struct buffer_head * nilfs_mdt_get_frozen_buffer(struct inode *inode,
struct buffer_head *bh);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	struct buffer_head *	[return value]	Found frozen copy of the buffer.
[in]	struct inode *	inode	Inode of the meta data file.
[in]	struct buffer_head *	bh	Buffer head of a searching buffer.

Caller for:

#	NAME	DESCRIPTION	FILE
1.	nilfs_page_get_nth_block	<TODO>	page.h

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_dat_translate	Translate a virtual block number to a block number.	dat.c

6.7.4.4. Metadata File State Management API

6.7.4.4.1. nilfs_mdt_mark_dirty

Summary: Mark metadata file's inode as dirty.

Declaration:
void nilfs_mdt_mark_dirty(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[in|out]	struct inode *	inode	Inode of the meta data file.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_sufile_alloc	Allocate a segment.	sufile.c
2.	nilfs_sufile_do_cancel_free	<TODO>	
3.	nilfs_sufile_do_scrap	<TODO>	
4.	nilfs_sufile_do_free	<TODO>	
5.	nilfs_sufile_mark_dirty	<TODO>	
6.	nilfs_sufile_set_segment_usage	Set usage of a segment.	
7.	nilfs_sufile_do_set_error	<TODO>	
8.	nilfs_sufile_truncate_range	Truncate range of segment array.	
9.	nilfs_sufile_resize	Resize segment array.	
10.	nilfs_ifile_create_inode	Create a new disk inode.	ifile.c
11.	nilfs_palloc_commit_alloc_entry	Finish allocation of a persistent object.	alloc.c
12.	nilfs_palloc_commit_free_entry	Finish deallocating a persistent object.	
13.	nilfs_palloc_freev	Deallocate a set of persistent objects.	
14.	nilfs_cpfile_get_checkpoint	Get a checkpoint.	cpfile.c
15.	nilfs_cpfile_delete_checkpoints	Delete checkpoints.	
16.	nilfs_cpfile_set_snapshot	<TODO>	
17.	nilfs_cpfile_clear_snapshot	<TODO>	
18.	nilfs_mdt_insert_new_block	<TODO>	mdt.c
19.	nilfs_mdt_delete_block	Make a hole on the meta data file.	
20.	nilfs_mdt_mark_block_dirty	Mark a block on the meta data file dirty.	
21.	nilfs_segctor_create_checkpoint	<TODO>	segment.c
22.	nilfs_segctor_collect_dirty_files	<TODO>	
23.	nilfs_dat_commit_entry	<TODO>	dat.c
24.	nilfs_dat_move	Change a block number.	
25.	nilfs_mark_inode_dirty	<TODO>	inode.c
26.	nilfs_dirty_inode	Reflect changes on given inode to an inode block.	

6.7.4.4.2. nilfs_mdt_clear_dirty

Summary: Unset dirty bit of metadata file's inode state.

Declaration:
void nilfs_mdt_clear_dirty(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[in|out]	struct inode *	inode	Inode of the meta data file.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_segctor_clear_metadata_dirty	Unset dirty bit of ifile, cpfile, sufile and dat file.	segment.c

6.7.4.4.3. nilfs_mdt_cno

Summary: Get actual checkpoint number for metadata file.

Declaration:
__u64 nilfs_mdt_cno(struct inode *inode);

Parameters:

	TYPE	NAME	DESCRIPTION
[out]	__u64	[return value]	Checkpoint number.
[in]	struct inode *	inode	Inode of the meta data file.

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_cpfile_get_checkpoint	Get a checkpoint.	cpfile.c
2.	nilfs_cpfile_do_get_cpinfo	<TODO>	
3.	nilfs_cpfile_is_snapshot	<TODO>	
4.	nilfs_cpfile_get_stat	Get checkpoint statistics.	
5.	nilfs_dat_commit_start	<TODO>	dat.c
6.	nilfs_dat_commit_end	<TODO>	
7.	nilfs_dat_abort_end	<TODO>	

6.7.4.4.4. nilfs_mdt_bgl_lock

Summary: Get spinlock for blockgroup.

Declaration:
```
#define nilfs_mdt_bgl_lock(inode, bg) \
        (&NILFS_MDT(inode)->mi_bgl->locks[(bg) & (NR_BG_LOCKS-1)].lock)
```

Called in:

#	NAME	DESCRIPTION	FILE
1.	nilfs_palloc_group_desc_nfrees	Get the number of free entries in a group.	alloc.c
2.	nilfs_palloc_group_desc_add_entries	Adjust count of free entries.	
3.	nilfs_palloc_find_available_slot	Find available slot in a group.	
4.	nilfs_palloc_commit_free_entry	Finish deallocating a persistent object.	
5.	nilfs_palloc_abort_alloc_entry	Cancel allocation of a persistent object.	
6.	nilfs_palloc_freev	Deallocate a set of persistent objects.	

7. NILFS Utilities

<TODO>

8. Terminology and Abbreviations

Terminology	Description

9. References

1. Documentation/filesystems/nilfs2.txt