Lecture 10

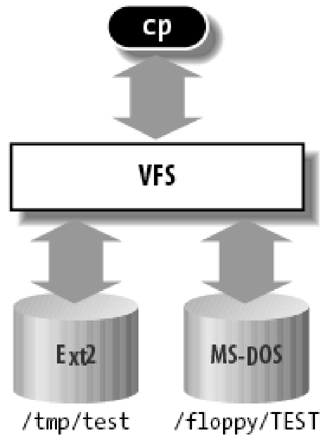
The File System

The Filesystem

- The bits stored on a storage media has to be given meaning
- Different physical media have different methods of access
- I/O drivers finally have to be invoked to perform the actual read of the bits but have to figure out which bits to read and what they mean

The Virtual Filesystem (VFS)

- VFS exposes a uniform API regardless of the underlying implementation or physical realities
- Abstracts away the complexities from the users



(a)

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

The Common File Model

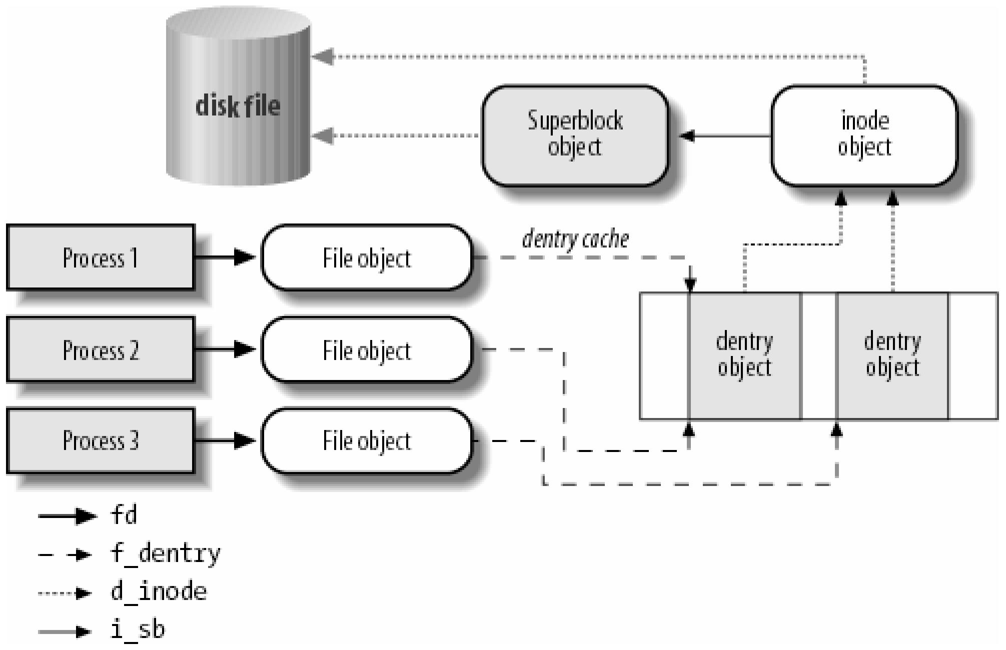
- Capable of representing all supported filesystems

- Any physical representation must be translated into the common file model

The Objects in the Common File Model

- The superblock object: stores information about a mounted filesystem
  - A filesystem is mounted if it is properly set up for use and access
- The inode object: stores the general information about a specific file
- The file object: stores information about the interaction between an opened file and a process
  - Exists only in kernel memory during the period when a process has the file opened
- The dentry object: stores information about the linking of a directory entry with the corresponding file

Relationship between the objects



Source: Daniel P. Bovet, Understanding the Linux Kernel.

The Superblock

- All superblock objects are linked in a circular doubly linked list
- Each filesystem has its own information (`s_fs_info`)
  - For disks, this may include allocation bitmaps for allocating and freeing disk blocks
- Corresponds to the filesystem control block stored on disk

The Inode

- Contains all information needed by the filesystem to handle a file
- For disk-based filesystems, it corresponds to the file control block
- Each inode is associated with an inode number that uniquely identifies the file within the filesystem
  - Accessed using “`ls -i`”
  - Maximum inode number is allocated during filesystem creation
  - Inode numbers given in sequence during file creation
  - Generally accessed using hashing

The File Object

- Describes how a process is interacting with an opened file
- Created when a file is opened
- Main information: the file pointer – the current position in the file to which the next operation will take place
- Allocated via a slab cache named **`filp`**
- Each superblock has a list of file objects

The Dentry Objects

- For each directory, there is a dentry structure
- Dentry objects also do not have a corresponding structure on disk
  - Different file systems do directory differently, hence need to “normalize”
- Dentry objects are stored in a slab allocator cache called **`dentry_cache`**

The Dentry Cache

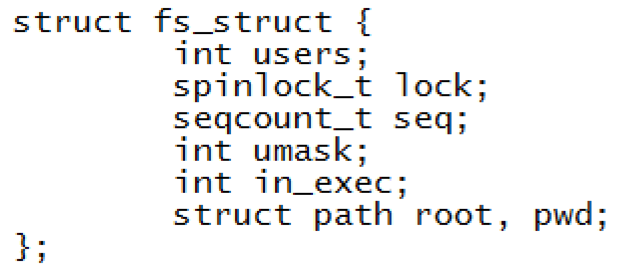
- Constructing a dentry object from disk information takes a lot of time, plus there is usually temporal locality in file usage, hence makes sense to cache

- The dentry cache consists of
  - A set of dentry objects in the in-use, unused, or negative state.
  - A hash table to derive the dentry object associated with a given filename and a given directory quickly.

- The dentry cache also controls the inode cache
  - Inodes in kernel memory that are associated with unused dentries are not discarded; they can be quickly be referenced again

Files associated with a process

- Each process descriptor has pointer to its a **`fs_struct`**
  - May be shared by several processes

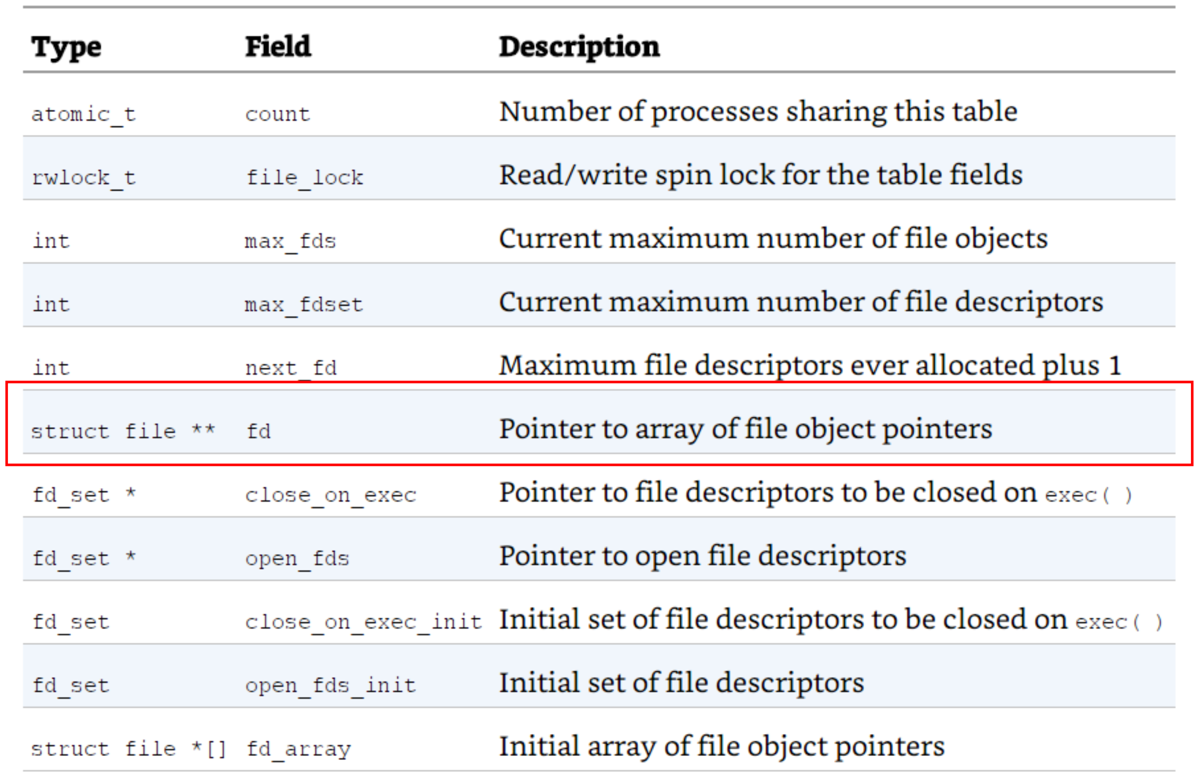


```
struct fs_struct {
        int users;
        spinlock_t lock;
        seqcount_t seq;
        int umask;
        int in_exec;
        struct path root, pwd;
};
```

- Another field, the **`files`** field of a process descriptor tells which files are opened currently by the process

`files_struct`

- Pointed to by the `files` field

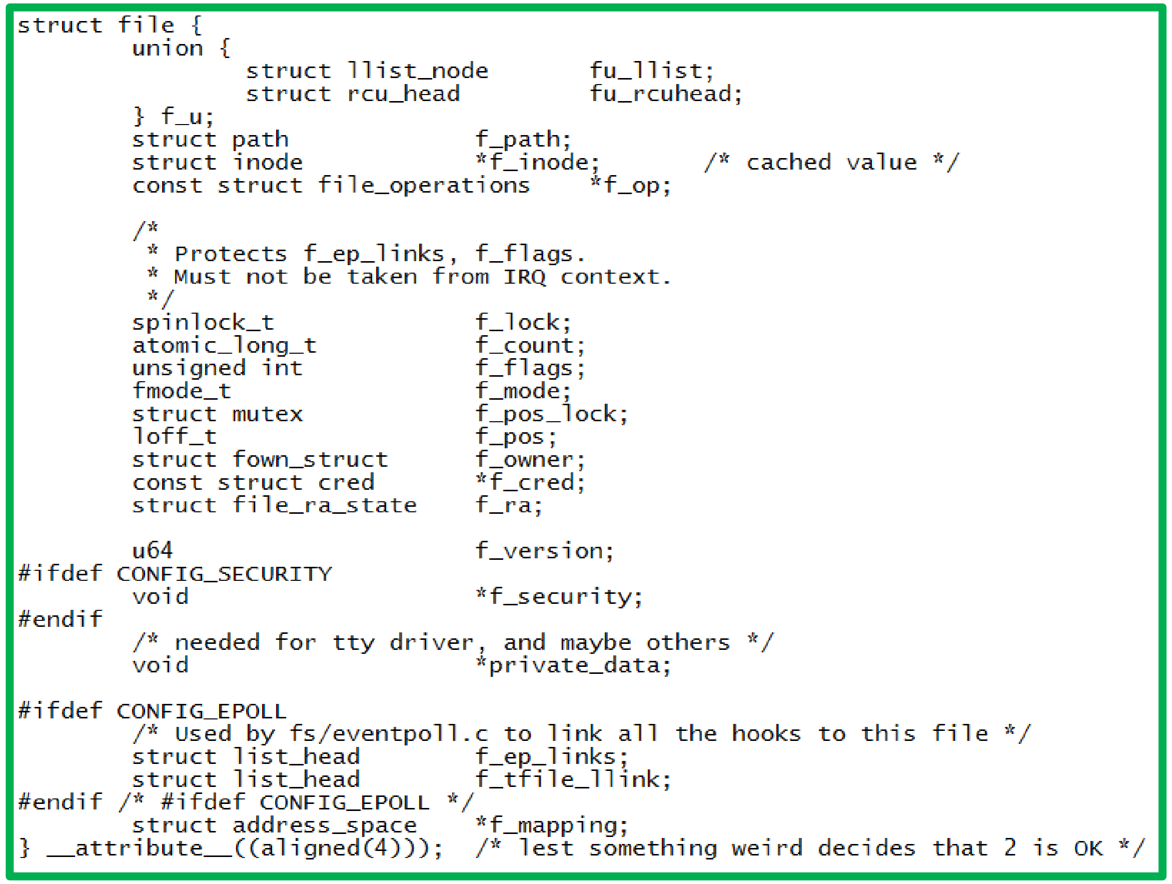


Type	Field	Description
`atomic_t`	`count`	Number of processes sharing this table
`rwlock_t`	`file_lock`	Read/write spin lock for the table fields
`int`	`max_fds`	Current maximum number of file objects
`int`	`max_fdset`	Current maximum number of file descriptors
`int`	`next_fd`	Maximum file descriptors ever allocated plus 1
`struct file **`	`fd`	Pointer to array of file object pointers
`fd_set *`	`close_on_exec`	Pointer to file descriptors to be closed on `exec( )`
`fd_set *`	`open_fds`	Pointer to open file descriptors
`fd_set`	`close_on_exec_init`	Initial set of file descriptors to be closed on `exec( )`
`fd_set`	`open_fds_init`	Initial set of file descriptors
`struct file *[]`	`fd_array`	Initial array of file object pointers

The File Object - **`struct file`**

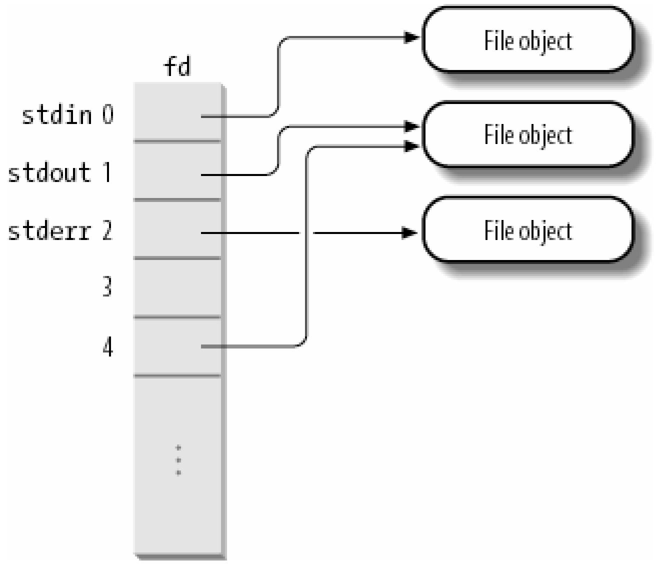
- Entry in **`fd`** array

- **`fd`** is the integer used in **`open()`**, **`read()`**, **`write()`**, **`close()`** (user level) calls



```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
        struct inode            *f_inode;       /* cached value */
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hooks to this file */
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

The `fd` array

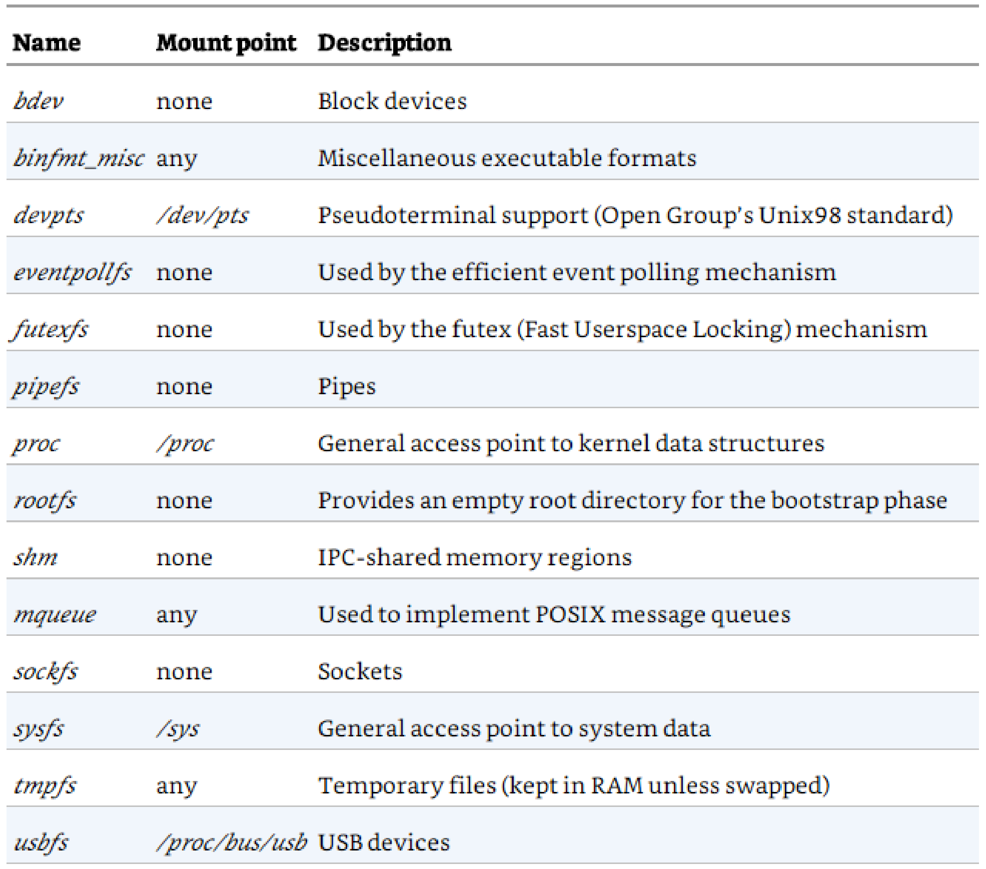

fd
stdin 0
stdout 1
stderr 2
3
4
File object
File object
File object

Filesystems

- In a traditional Unix system, there is only one tree of mounted filesystems: starting from the system's root filesystem, each process can potentially access every file in a mounted filesystem by specifying the proper pathname.

- In Linux 2.6 every process might have its own tree of mounted filesystems — the namespace of the process.
  - Though most of the time, all processes share the same mounted filesystem

Common types of filesystems



Name	Mount point	Description
bdev	none	Block devices
binfmt_misc	any	Miscellaneous executable formats
devpts	*/dev/pts*	Pseudoterminal support (Open Group's Unix98 standard)
eventpollfs	none	Used by the efficient event polling mechanism
futexfs	none	Used by the futex (Fast Userspace Locking) mechanism
pipefs	none	Pipes
proc	*/proc*	General access point to kernel data structures
rootfs	none	Provides an empty root directory for the bootstrap phase
shm	none	IPC-shared memory regions
mqueue	any	Used to implement POSIX message queues
sockfs	none	Sockets
sysfs	*/sys*	General access point to system data
tmpfs	any	Temporary files (kept in RAM unless swapped)
usbfs	*/proc/bus/usb*	USB devices

Mounting a filesystem

- Before a filesystem can be used, it must be mounted
  - Mounted gives it a pathname
- In Linux, you can mount the same filesystem multiple times
  - But only one superblock object for these multiple mount points
- Mounting forms a hierarchy
  - You can mount another filesystem under the directory of one

The File System Formats

(for disks)

Purpose

- Controls how data is stored, organized, managed and retrieved on a storage media
- Provides a logical namespace
- Provides an abstract user interface
- Provides security

Journaling File Systems

- Keeps track of changes not yet committed to the file system's main part
  - Certain file operations requires multiple writes to different parts of a disk
  - Example: to delete a file, one has to (1) remove directory entry, (2) release the inode, and (3) move blocks into the free pool
    - What if system crash in between?

- Records the intentions of such changes in a journal

- If system crashes, will be able to can recover to a consistent state

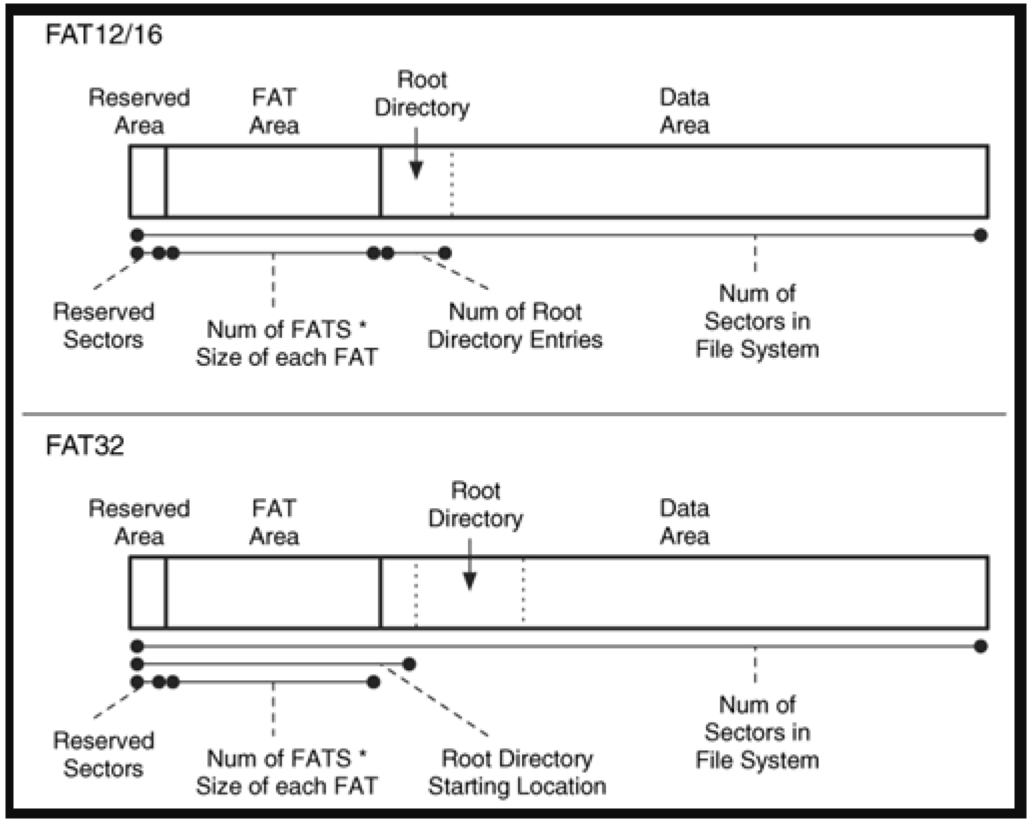
File Allocation Table

(FAT)

History

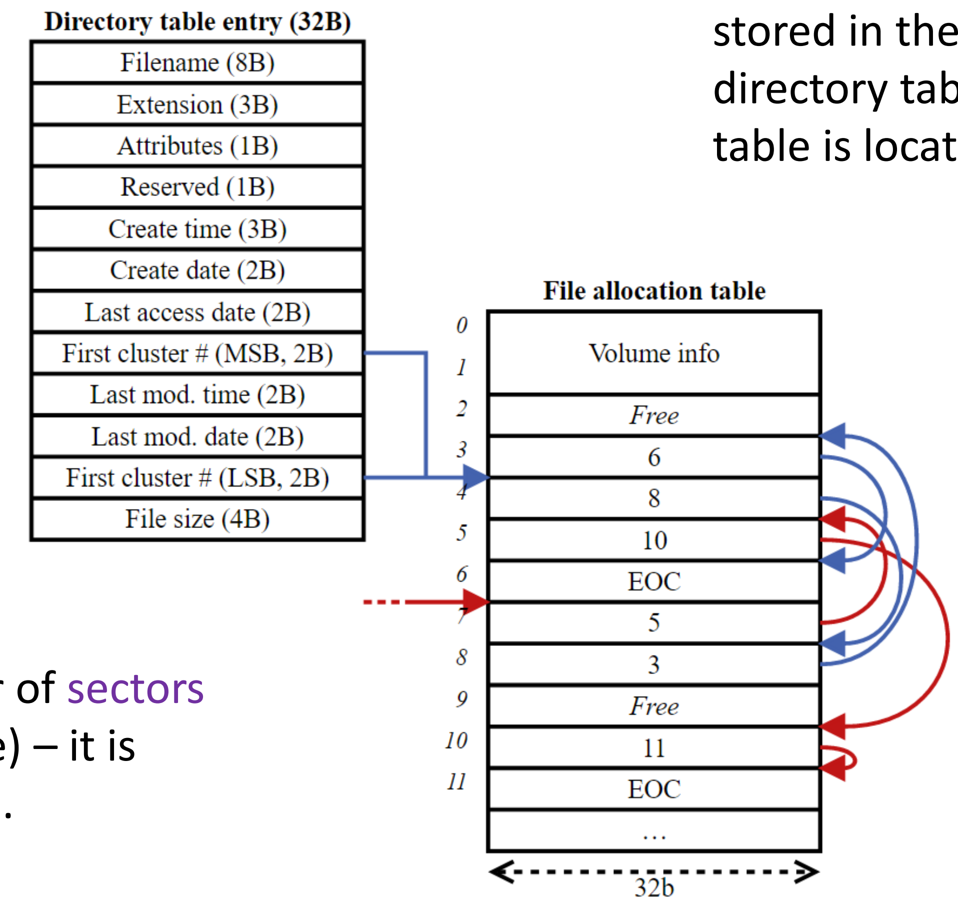
- Originally developed by Microsoft, NCR, SCP, IBM, Compaq, Digital Research, Novell, and Caldera
- Used in MS-DOS, still used in many embedded media
- FAT (8 bit) → FAT12 → FAT16 → FAT32 → exFAT (64 bits)
- A linked list format

FAT layout


FAT12/16
Reserved Area
FAT Area
Root Directory
Data Area
Reserved Sectors
Num of FATS * Size of each FAT
Num of Root Directory Entries
Num of Sectors in File System
FAT32
Reserved Area
FAT Area
Root Directory
Data Area
Reserved Sectors
Num of FATS * Size of each FAT
Root Directory Starting Location
Num of Sectors in File System

File Allocation Table

A directory table (a.k.a. a folder) is a special file stored in the data area that contains a number of directory table entries. A special root directory table is located in the start of the data area.



A cluster consists of a number of sectors (the storage unit of the device) – it is predefined at formatting time.

New Technology File System

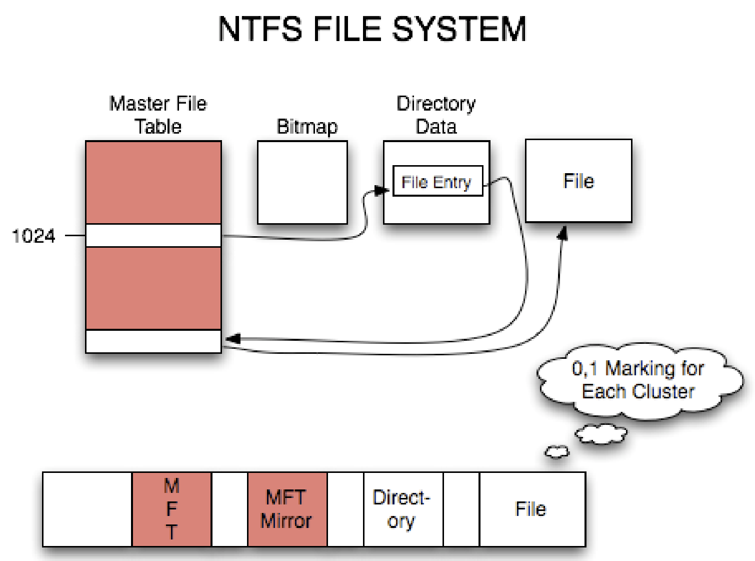
(NTFS)

Source: David Riley, UW @ La Crosse

Basic Implementation

- Fundamental Data Structure of NTFS is the Metafile.
- The Metafiles:

$MFT	Master File Table — ***THE*** Metafile
$MFTMIRR	Copy of the beginning records of the MFT
$LOGFILE	Transactional logging file
$VOLUME	Contains volume/partition information
$ATTRDEF	Attribute definitions
.	Root directory of the disk (C:\)
$BITMAP	Contains drive's cluster map (used vs. free)
$BOOT	Boot record
$BADCLUS	Lists bad clusters on the drive
$QUOTA	User quota information (NTFS 5.0)
$UPCASE	Maps lowercase characters to their uppercase version



NTFS Security Features

- Availability and Integrity
  - $LOGFILE
  - $MFTMIRR
  - $BADCLUS
- Confidentiality
  - $MFT
  - Encrypting File System (added in NTFS 5.0)
- Authenticity
  - $MFT CREATOR_OWNER

Availability and Integrity

- $LOGFILE
  - At the same time files are modified, certain information about the changes are written in two different record types to $LOGFILE.
    - Redo records are written with information about the modification that must be redone if a modify or delete process is interrupted.
    - An undo record is written in order to facilitate the rollback of an append if the process fails between the time the file is extended and the data is actually written in the new free space created.
  - Example: CHKDSK uses the information in $LOGFILE to ensure data integrity and availability if a system is powered down without flushing the disk buffers.

Availability and Integrity

- $MTFMIRR
  - Metafile that is stored “in the middle” of the disk as a backup-copy of $MFT.
  - Used in case MFT is corrupted.
  - $BOOT (which can be stored at either the first or last sector of the disk) holds pointers to both the $MFT and $MTFMIRR

Availability and Integrity

- $BADCLUS
  - If an error occurs while reading data off of an NTFS formatted partition NT will assume that the cluster is a "bad cluster."
  - The error recovery process will then enter the cluster into the $BADCLUS metafile
  - It will then recover what it can of the data and place it in another location.
  - This feature is enhanced greatly with fault-tolerant file system drivers.

Confidentiality and Security

- $MTF
  - Each file and folder has it's own record in the $MFT.
  - Within that record is a pointer to an attribute record $SECURITY_DESCRIPTOR
  - The Security Descriptor holds information that allows NT to map permissions (Allow or Deny of: Read, Write, Execute, Modify, Full, etc) to Users (represented by their internal SID)
    - S-1-5-21-XXXXXXXXX-XXXXXXXXX-XXXXXXXXX-500 represents the local machine's "Administrator" account.
  - If no SID exists in $SECURITY_DESCRIPTOR the permission is assumed to be an outright "deny" of all rights.
  - As demonstrated earlier, $SECURITY_DESCRIPTOR is read by the NT operating system, not by the hardware. Therefore, it can be bypassed and the data accessed directly on disk.

Confidentiality and Security

- Encrypting File System (EFS)
  - New in NTFS 5.0 (Windows 2000)
  - Uses Windows 2000 Cryptography Services.
  - Users can explicitly specify to encrypt a file, or Windows 2000 will automatically encrypt files that are within a folder that has been specified as encrypted.
  - Uses a stronger variant of DES called DESX. This process is symmetric and quicker than asymmetric technologies which is ideal when encrypting what can be huge (up to 2TB) files.

Confidentiality and Security

- The EFS Process
  - The first time a user encrypts a file, Windows 2000 Cryptography services creates for the user a unique private and public key for use in File System Encryption.
  - When a file is encrypted, EFS generates a random number, the File Encryption Key (FEK), and uses that number as the key in the DESX encryption process.
  - For each user that is given permission to decrypt the file, EFS encrypts the FEK with that user’s public key and stores that encrypted FEK in a special location inside the encrypted file.
  - Therefore only the certain user(s) can decrypt the FEK with their private key and then decrypt the file.

Extended Filesystem

(EXT)

EXT2

- Introduced in 1993
- No journaling
- Maximum individual file size can be from 16 GB to 2 TB
- Overall file system size can be from 2 TB to 32 TB

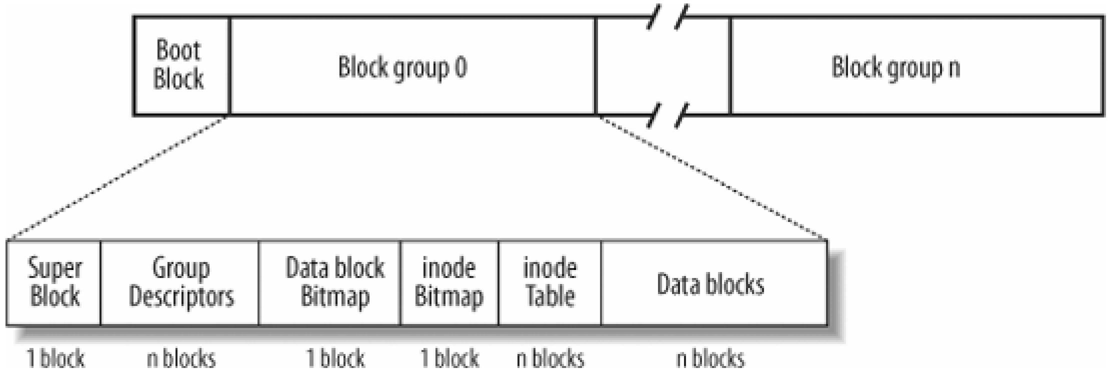
EXT3

- Introduced in 2001
- Starting from Linux Kernel 2.4.15 ext3 was available.
- The main benefit of ext3 is that it allows journaling.
- Journaling has a dedicated area in the file system, where all the changes are tracked.
- Maximum individual file size can be from 16 GB to 2 TB
- Overall ext3 file system size can be from 2 TB to 32 TB
- There are three types of journaling available in ext3 file system
  - Journal – Metadata and content are saved in the journal.
  - Ordered – Only metadata is saved in the journal. Metadata are journaled only after writing the content to disk. This is the default.
  - Writeback – Only metadata is saved in the journal. Metadata might be journaled either before or after the content is written to the disk.
- You can convert a ext2 file system to ext3 file system directly (without backup/restore)

EXT4

- Introduced in 2008.
- Starting from Linux Kernel 2.6.19 ext4 was available.
- Maximum individual file size can be from 16 GB to 16 TB
- Overall maximum ext4 file system size is 1 EB (exabyte). 1 EB = 1024 PB (petabyte). 1 PB = 1024 TB (terabyte).
- Directory can contain a maximum of 64,000 subdirectories (as opposed to 32,000 in ext3)
- You can also mount an existing ext3 fs as ext4 fs (without upgrade).
- Several other new features are introduced in ext4 to improved the performance and reliability of the filesystem when compared to ext3.
- In ext4, you also have the option of turning the journaling feature “off”.

Layout of ext2 Filesystem



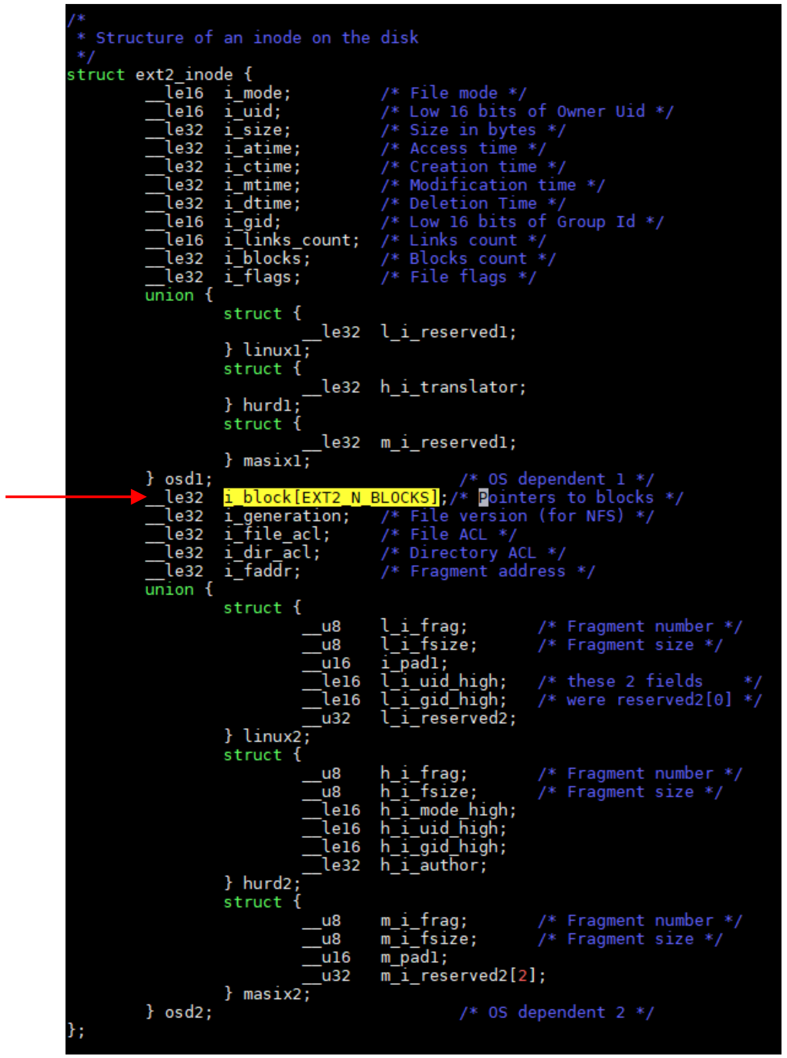
Block size is selectable at creation. From 1024 to 4096 bytes.

Ext2

- First block reserved for boot sector
  - A block, just like a cluster in FAT, consists of several contiguous sectors.
- The rest partitioned into block groups – all of identical size
  - Up to 8$\times$ block-size blocks
- Each block group contains one of the following:
  - A copy of the filesystem's superblock
  - A copy of the group of block group descriptors
  - A data block bitmap
  - An inode bitmap
  - A table of inodes
  - A chunk of data that belongs to a file; i.e., data blocks
  - Unused, i.e., free

The ext2 inode structure

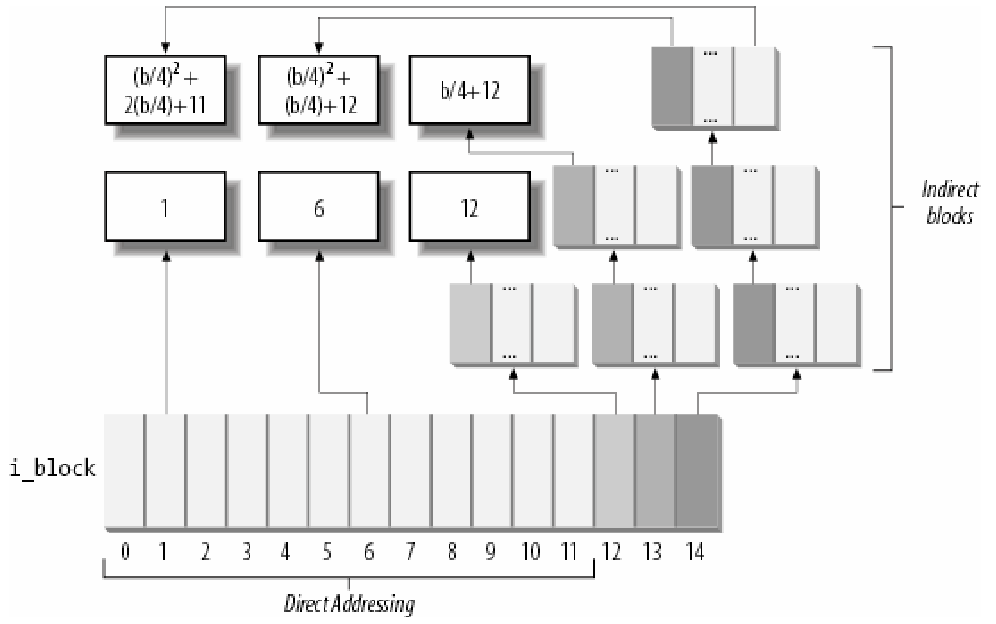
- fs/ext2/ext2.h



```
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        union {
                struct {
                        __le32  l_i_reserved1;
                } linux1;
                struct {
                        __le32  h_i_translator;
                } hurd1;
                struct {
                        __le32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
        union {
                struct {
                        __u8    l_i_frag;       /* Fragment number */
                        __u8    l_i_fsize;      /* Fragment size */
                        __u16   i_pad1;
                        __le16  l_i_uid_high;   /* these 2 fields    */
                        __le16  l_i_gid_high;   /* were reserved2[0] */
                        __u32   l_i_reserved2;
                } linux2;
                struct {
                        __u8    h_i_frag;       /* Fragment number */
                        __u8    h_i_fsize;      /* Fragment size */
                        __le16  h_i_mode_high;
                        __le16  h_i_uid_high;
                        __le16  h_i_gid_high;
                        __le32  h_i_author;
                } hurd2;
                struct {
                        __u8    m_i_frag;       /* Fragment number */
                        __u8    m_i_fsize;      /* Fragment size */
                        __u16   m_pad1;
                        __u32   m_i_reserved2[2];
                } masix2;
        } osd2;                         /* OS dependent 2 */
};
```

The ext2 file

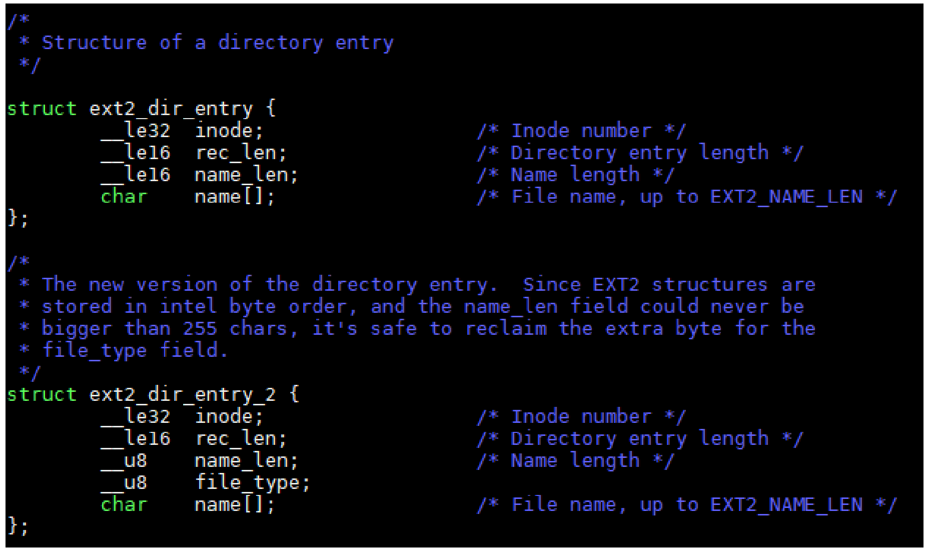
- There are pointers to the first 12 blocks which contain the file's data in the inode.
- There is a pointer to an indirect block (which contains pointers to the next set of blocks), a pointer to a doubly indirect block and a pointer to a trebly indirect block.



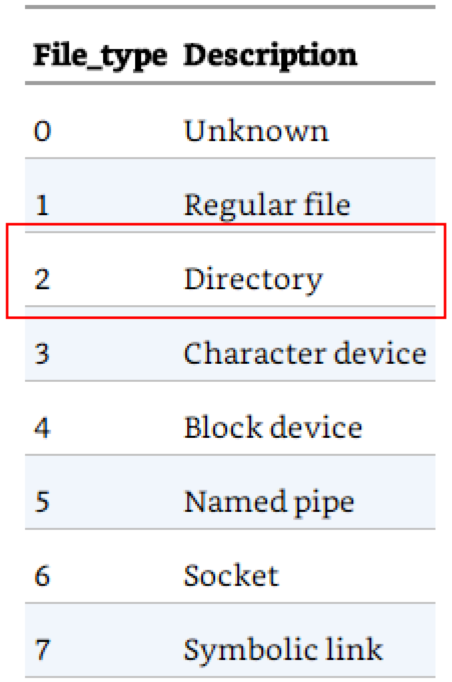
Directory file

- Directories are maintained using special file
  - A file of a special type '2'

- Data is a simple array of these:

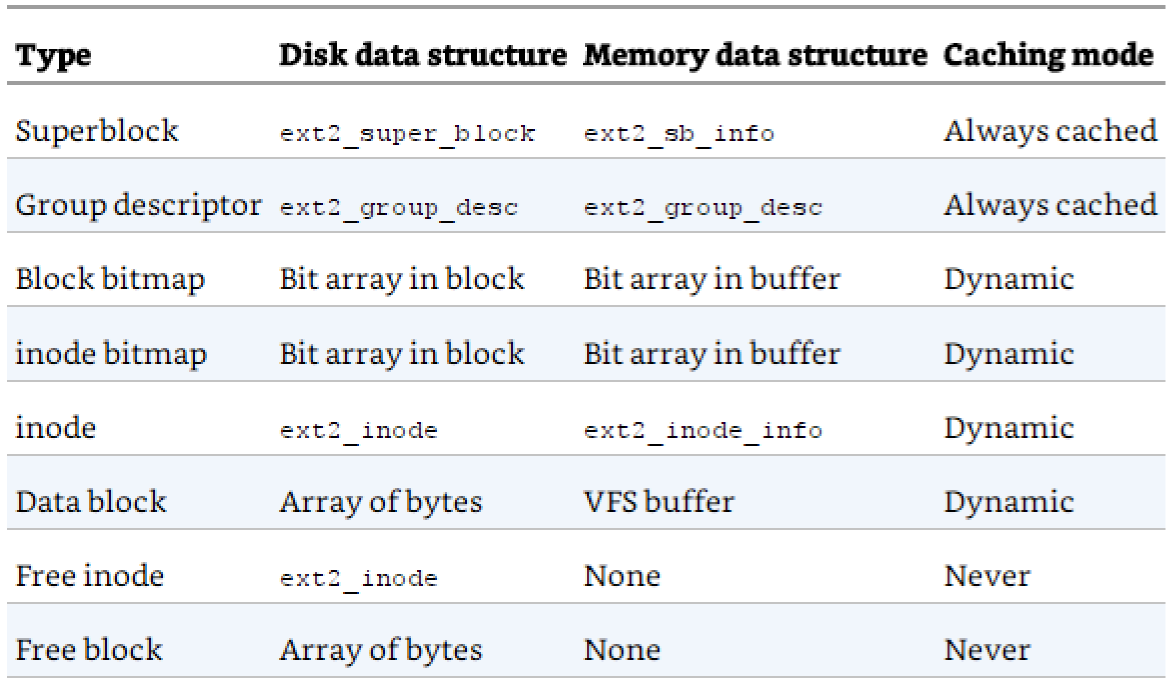


```
/*
 * Structure of a directory entry
 */

struct ext2_dir_entry {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __le16  name_len;               /* Name length */
        char    name[];                 /* File name, up to EXT2_NAME_LEN */
};

/*
 * The new version of the directory entry.  Since EXT2 structures are
 * stored in intel byte order, and the name_len field could never be
 * bigger than 255 chars, it's safe to reclaim the extra byte for the
 * file_type field.
 */
struct ext2_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[];                 /* File name, up to EXT2_NAME_LEN */
};
```



File_type	Description
0	Unknown
1	Regular file
2	Directory
3	Character device
4	Block device
5	Named pipe
6	Socket
7	Symbolic link

VFS and ext2



Type	Disk data structure	Memory data structure	Caching mode
Superblock	`ext2_super_block`	`ext2_sb_info`	Always cached
Group descriptor	`ext2_group_desc`	`ext2_group_desc`	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	`ext2_inode`	`ext2_inode_info`	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	`ext2_inode`	None	Never
Free block	Array of bytes	None	Never

Data block allocation

- Attempts to allocate each new directory in the group containing its parent directory
  - Accesses to parent and children directories are likely to be closely related.
- Also attempts to place files in the same group as their directory entries
  - Directory accesses often lead to file accesses
- If the group is full, then the new file or new directory is placed in some other non-full group.
- The data blocks needed to store directories and files can be found by looking in the data allocation bitmap.
- Any needed space in the inode table can be found by looking in the inode allocation bitmap.

Journaling and ext3

- As disks and memory became bigger, more and more things buffered; also multiple operations can be split over time

- Journaling in ext3 in 2 phases:
    1. First, a copy of the blocks to be written is stored in the journal; then, when the I/ O data transfer to the journal is completed (in short, data is committed to the journal), the blocks are written in the filesystem.

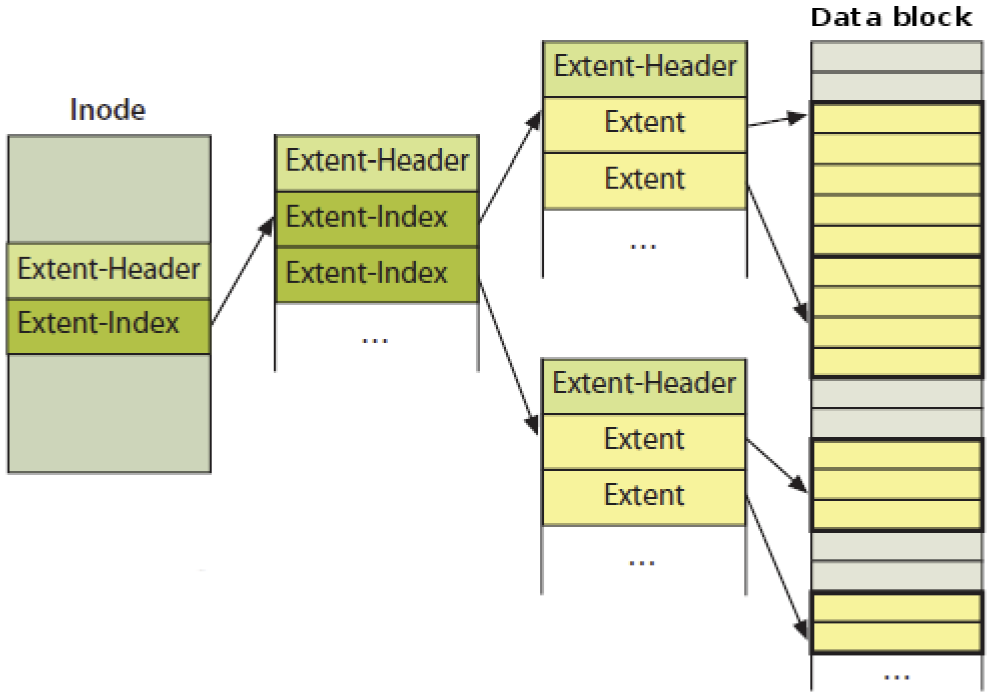
    2. When the I/ O data transfer to the filesystem terminates (data is committed to the filesystem), the copies of the blocks in the journal are discarded.

Journaling Block Device Layer

- A hidden file `.journal` located at the root of the filesystem is used
- Journaling is handled by the journaling block device (JDB) layer
  - Code in `fs/jbd2` of kernel source

Further improvement: ext4

- Introduces the extent
  - Range of contiguous physical blocks
  - Up to 128MB per extent for 4KB block size
  - There can be four extents stored in the inode.
  - When there are more than four extents to a file, the rest of the extents are indexed in a tree.



Other improvements of ext4

- Persistent pre-allocation
  - ext4 can pre-allocate on-disk space for a file.
- Delayed allocation
  - Delays block allocation until data is flushed to disk
- Unlimited number of subdirectories
- Directories can be kept in a B-tree-like format (as an option).
- Journal checksumming

Other improvements of ext4

- Metadata checksumming
- Faster file system checking
- By skipping unallocated block groups.
- Multiblock allocator
    - ext3 appends to a file, it calls the block allocator, once for each block.
    - ext4 uses delayed allocation which allows it to buffer data and allocate groups of blocks.
- Nanosecond timestamps
- Transparent encryption

The next big one: btrfs

B-tree Filesystem

btrfs vs zfs

- Storage device technology changing
  - Huge capacity
  - Solid state devices with different storage mechanism
- Sun Solaris pioneered ZFS (Zettabyte Filesystem)
  - Copyright reasons – cannot be distributed officially as part of Linux
  - Linux support by userspace driver or kernel patch
- Btrfs (B-tree Filesystem) – pronounced “better” or “butter” FS
  - stable version since 2014 by Oracle
  - GPL
  - Native Linux kernel support

Main features of btrfs

- B-tree based
  - Uses a b-tree derivative optimized for copy-on-write and concurrency created by IBM researcher Ohad Rodeh for every layer of the file system

- Copy-on-write
  - During a write operation, the old part of data is copied and not discarded

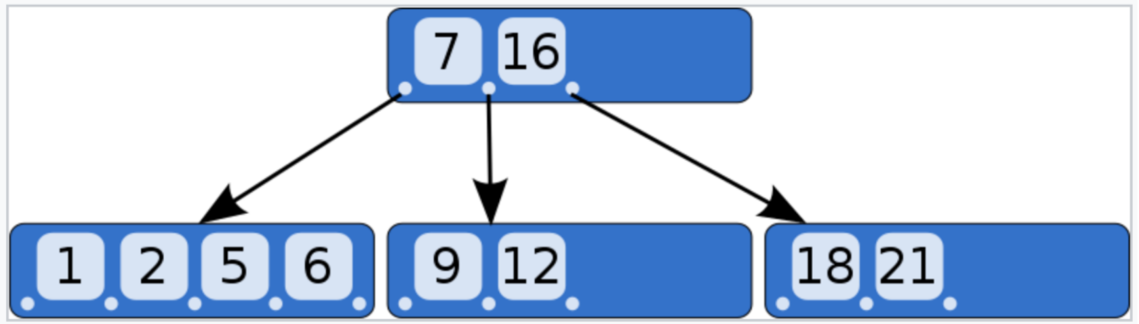
- Ref: https://www.researchgate.net/publication/262177144_BTRFS_The_linux_B-tree_filesystem

Definitions

- Page, block: a 4KB contiguous region on disk and in memory.
- Extent: A contiguous on-disk area. It is page aligned, and its length is a multiple of pages.
- Copy-on-write (COW): creating a new version of an extent or a page at a different location. Also called shadowing.
- Checksum: a hash value used to check the integrity of stored data
- Snapshot: a copy of the file system taken at a certain point in time; also called a clone

B-trees

- B-tree is a type of binary search tree
  - self-balancing
  - sorted data
  - searches, sequential access, insertions, and deletions in logarithmic time.



A B-tree (Bayer & McCreight 1972) of order 5 (Knuth 1998).

Source: Wikipedia

The core idea: COW-friendly B-tree

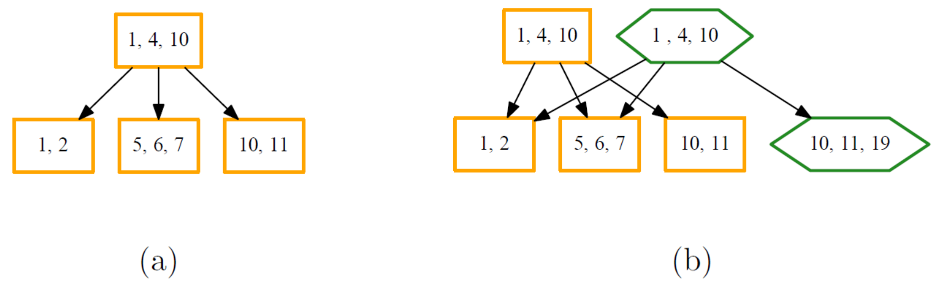


Figure 4: (a) A basic b-tree (b) Inserting key 19, and creating a path of modified pages.

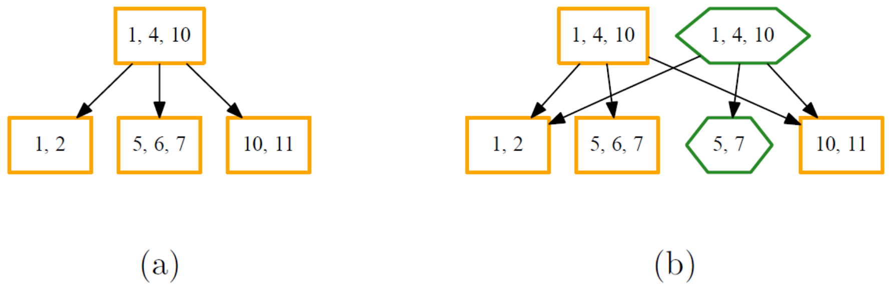


Figure 5: (a) A basic tree (b) Deleting key 6.

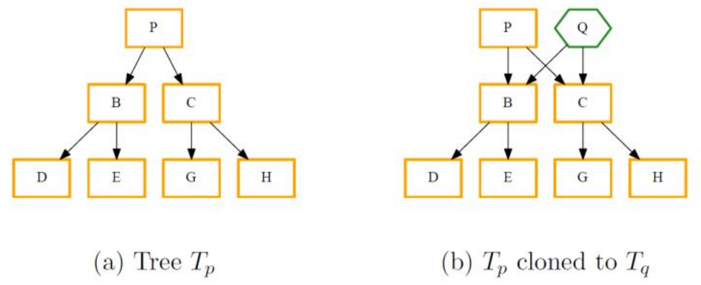


Figure 6: Cloning tree T_p. A new root Q is created, initially pointing to the same nodes as the original root P. As modifications will be applied, the trees will diverge.

Source: O. Rodeh, J. Bacik, and C. Mason, BTRFS: The linux B-tree filesystem

COW-friendly tree: inserting a node

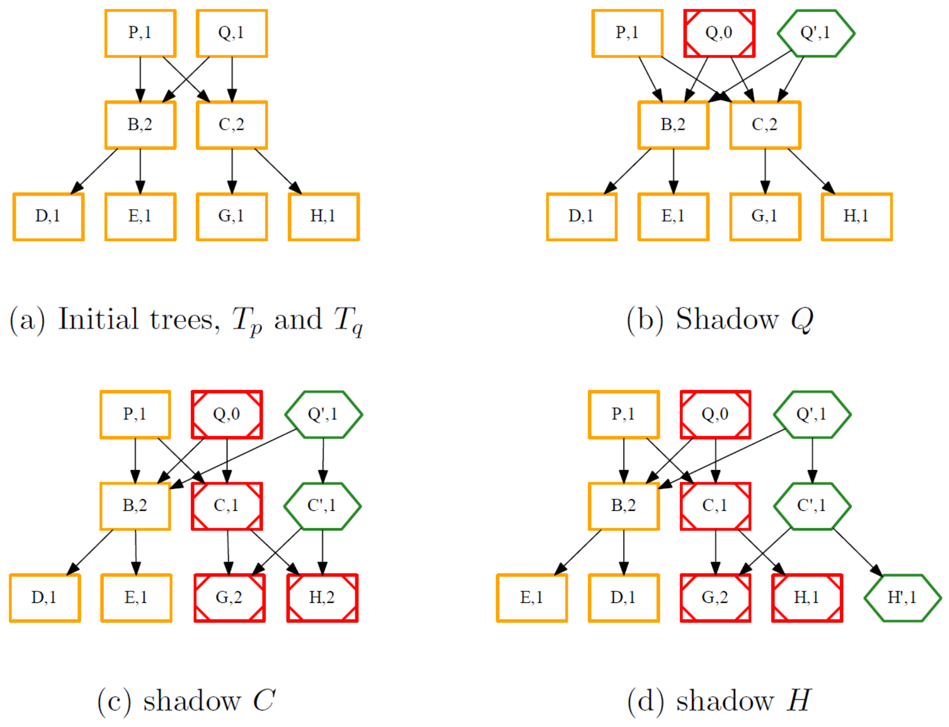


Figure 8: Inserting a key into node H of tree T_q. The path from Q to H includes nodes $\{Q, C, H\}$, these are all COWed. Sharing is broken for nodes C and H; the ref-count for C is decremented.

A btrfs B-tree

- A btrfs B-tree only has three types of data structures: keys, items, and block headers.
- The block header is fixed size and holds fields like checksums, flags, filesystem ids, generation number, etc.
- A key describes an object address:

```
struct btrfs_key {
    u64: objectid;  u8: type;  u64 offset;
}
```

Items

- An item is a btrfs key with additional offset and size fields

```
struct btrfs_item {
    struct btrfs_key key; u32 offset; u32 size;
}
```

In a btrfs B-tree

- Internal tree nodes hold only `[key, block-pointer]` pairs.
- Leaf nodes hold arrays of `[item, data]` pairs.
- Item data is variable sized.
- A leaf stores an array of items in the beginning, and a reverse sorted data array at the end. These arrays grow towards each other.

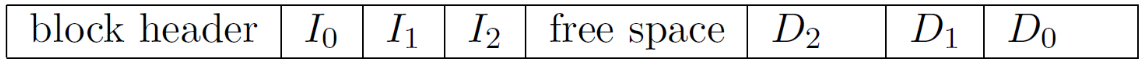


Figure 10: A leaf node with three items. The items are fixed size, but the data elements are variable sized.

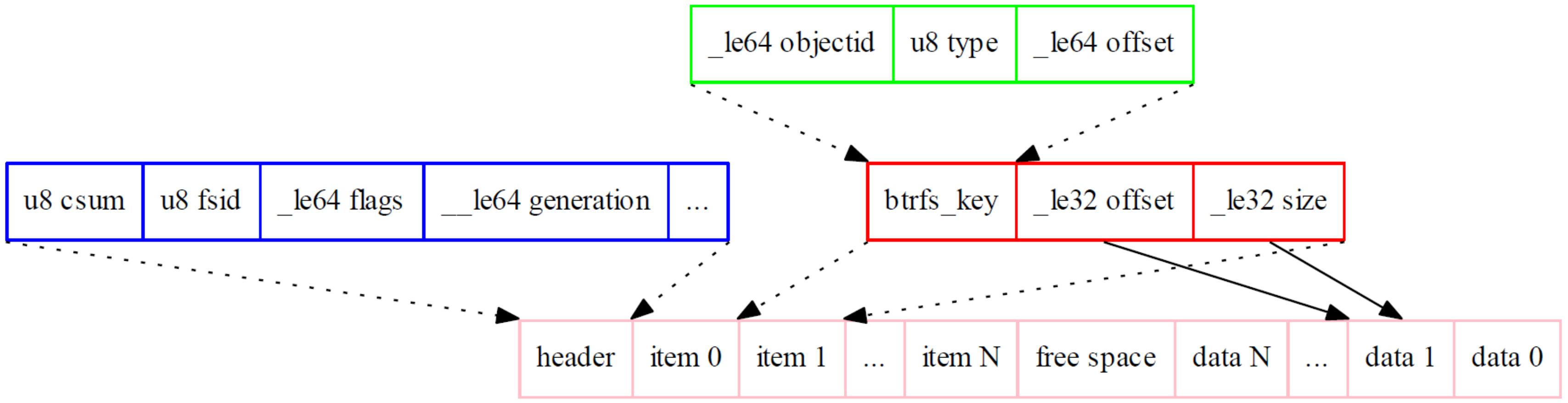


Figure 11: A detailed look at a generic leaf node holding keys and items.

Inodes on btrfs

- Inodes are stored in an inode item at offset zero in the key, and have a type value of one.
- Inode items are always the lowest valued key for a given object, and they store the traditional stat data for files and directories. This includes properties such as size, permissions, flags, and a count of the number of links to the object.
- The inode structure is relatively small, and will not contain embedded file data, extended attribute data, or extent mappings.
- These things are stored in other item types.

Small files in btrfs

- Small files that occupy less than one leaf block may be packed into the b-tree inside the extent item.
- The key offset is the byte offset of the data in the file
- The size field of the item indicates how much data is stored.
- There may be more than one of these per file.

Large files on btrfs

- Larger files are stored in extents. These are contiguous on-disk areas
- that hold user-data without additional headers or formatting.
- An extent item has
  - a generation number recording when the extent was created,
  - a `[disk block, length]` pair to record the area on disk, and
  - a `[logical file offset, length]` pair to record the file area.
- An extent is a mapping from a logical area in a file, to a physical area on disk
- If a file is stored in a small number of large extents, then a common operation such as a full file read will be efficiently mapped to a few disk operations.

Directories in btrfs

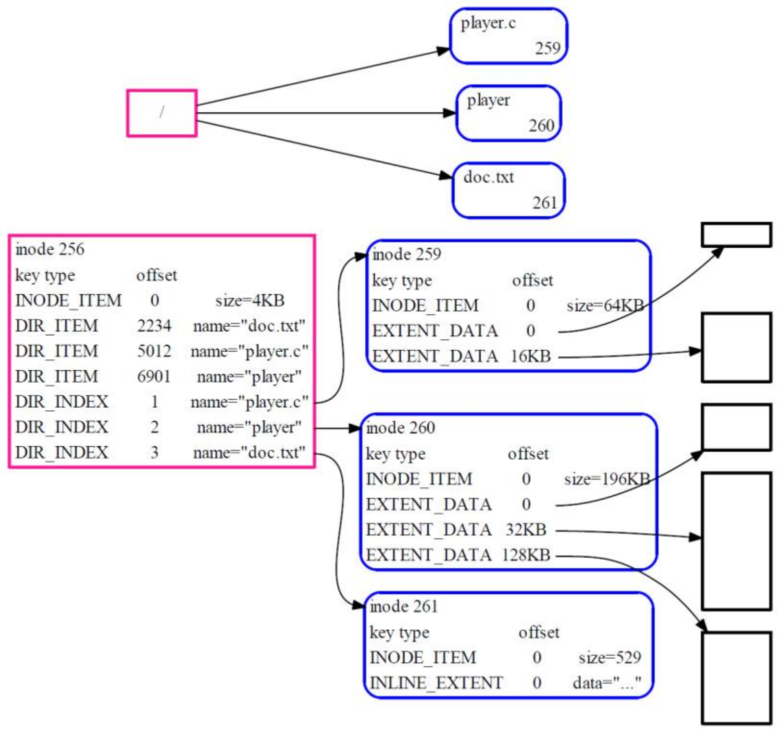
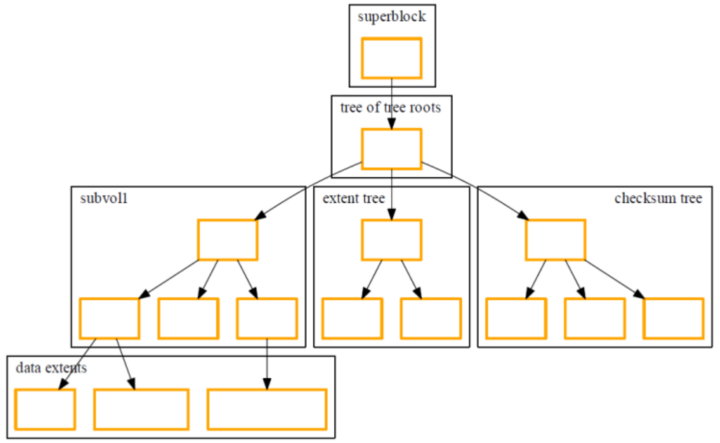


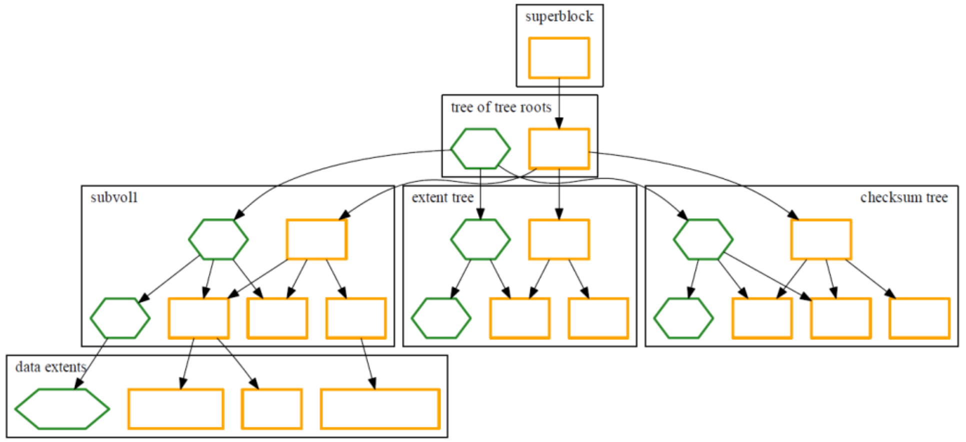
Figure 12: A directory structure with three files: `player.c`, `player`, and `doc.txt`. The files have corresponding inode numbers: 259, 260, and 261. The filesystem tree holds a directory with a double mapping, and three file elements. The `doc.txt` file is small enough to be inlined, while the other files have external extents.

btrfs Filesystem

- A filesystem is constructed from a forest of B-trees.
- A superblock located at a fixed disk location is the anchor. It points to a tree of tree roots, which indexes the b-trees making up the filesystem. The trees are:
  - Sub-volumes: store user visible files and directories.
  - Extent allocation tree: tracks allocated extents in extent items, and serves as an on-disk free-space map.
  - Checksum tree: holds a checksum item per allocated extent.
  - Chunk and device trees: indirection layer for handling physical devices. Allows mirroring/striping and RAID.
  - Reloc tree: for special operations involving moving extents.



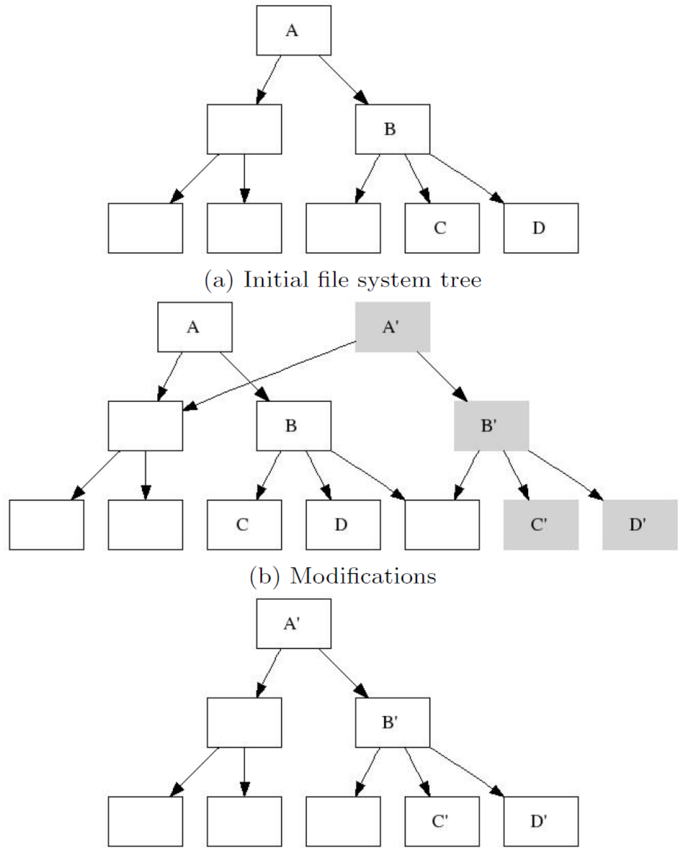
(a)



(b)

Figure 13: (a) A filesystem forest. (b) The changes that occur after modification; modified pages are colored green.

Checkpointing



(a) Initial file system tree

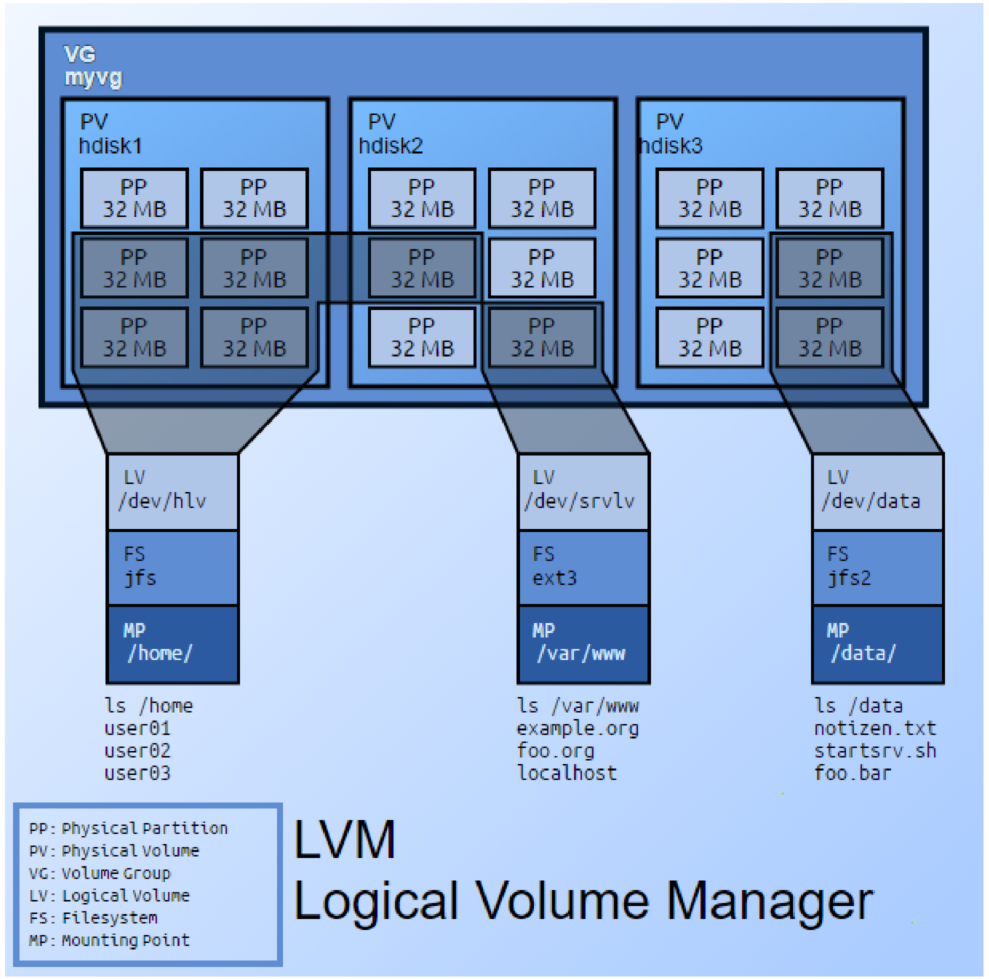
(b) Modifications

(c) New checkpoint

Fig. 4. Checkpoints.

Final digression: Logical Volumes

- A thin software layer on top of the hard disks and partitions, which creates an abstraction of continuity and ease-of-use for managing hard drive replacement, repartitioning and backup.



End