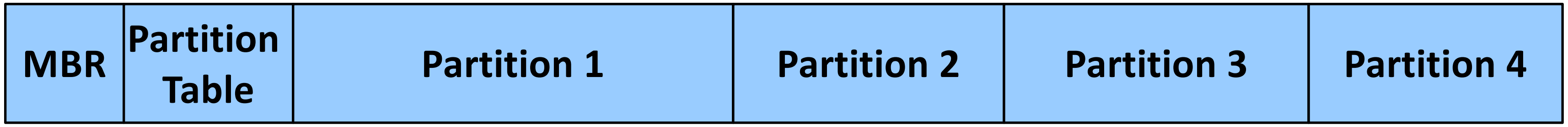
File System Design and Implementation

Disk Organization

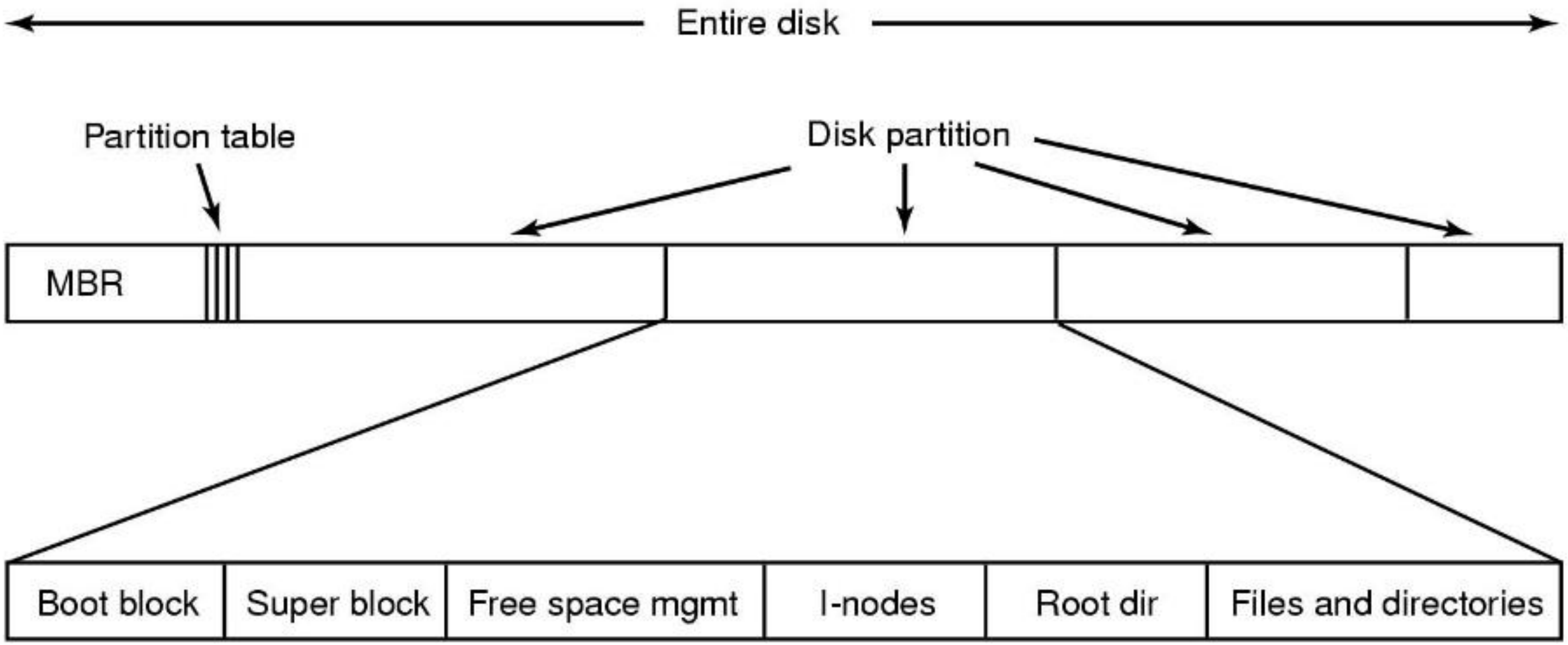
- File systems are usually created on regions of disks called Partitions.



MBR	Partition Table	Partition 1	Partition 2	Partition 3	Partition 4

- MBR: Master Boot Record, first sector that loads the boot loaders in partitions.
- Partition table: Configuration of partitions on disk. A disk can have 1 or more partitions.
- Partition table marks start and end blocks of partitions, partition type, if partition is bootable etc.
- Each partition may be formatted as a different file system.
- OS loads the partition table and interprets each partition as a different disk device.

File System Layout



- **A possible file system layout *with a Unix partition***

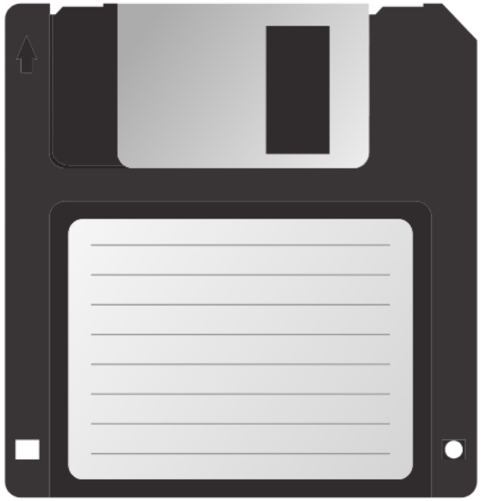
Disk Space Organization

- **Disk can be partitioned**
  - Each partition can have a different OS and/or different file system
  - One partition can be swap space for main memory
- **First block of disk has master boot record specifying primary partition**
- **Each partition has**
  - Boot block (loads OS in kernel space)
  - Superblock (contains key info about file system which is read into memory at boot time)
  - Free space management
  - List of I-nodes (or other data structure) giving info about all files
  - Directories and Files

Common File Systems

- FAT
  - by Windows and some embedded sytems
- UFS
  - Unix based: UFS, FFS, Ext2,3,4
- NTFS
  - by Windows
- HFS
  - Plus by Mac OS
- ISO9660
  - for CDROM/DVROM
- JFFS2
  - on Android
- New generation Unix: zfs, btrfs

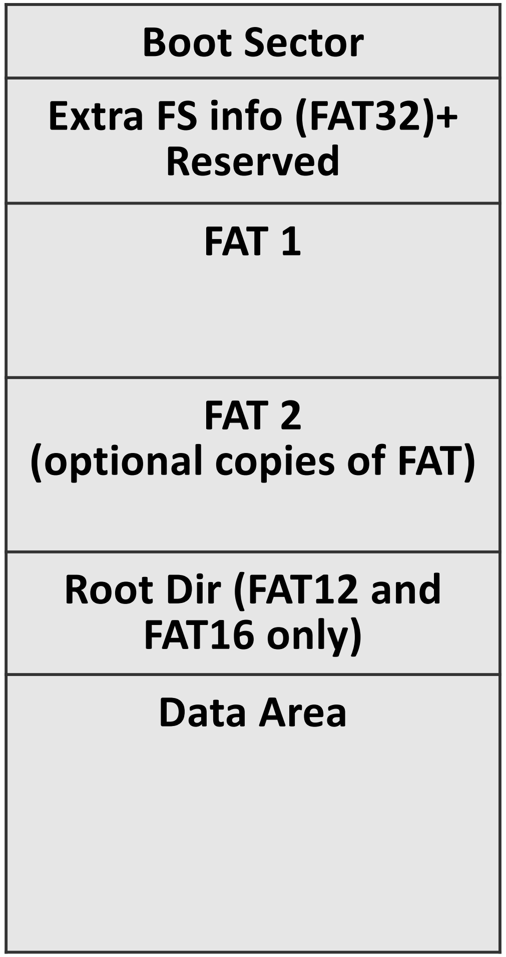
File Allocation Table (FAT)



- **Originally designed for floppy disks (360KB to 1.4MB)**
  - For MSDOS
- **Uses File Allocation Table as an array of pointers to store free block list and file linkage.**
- **FAT is a linked list implemented on an array.**
  - Index of a pointer represents the disk block number. i^{th} block information is stored on FAT_i as the next block, either as next free block or next block in the file
- **File attributes are kept in directory blocks.**
  - No inode block.
  - Identifier of a file is the number of the first block in FAT.

FAT File System Layout

- **Boot Sector:**
  - volume name,
  - #bytes/sec,
  - #sectors/cluster,
  - #clusters,
  - #FAT copies
- **FAT: array of data block Pointer**
- **Root directory block (FAT32 keeps in data, keep id in Boot)**



Boot Sector

Extra FS info (FAT32)+ Reserved

FAT 1

FAT 2 (optional copies of FAT)

Root Dir (FAT12 and FAT16 only)

Data Area

FAT: Table entries

- **FAT entries depend on FAT implementation:**
  - FAT12: 12bits (3 bytes per 2 entries),
  - FAT16: 16 bits,
  - FAT32: 28 bits per entry.
- **Initially all clusters are marked as free (0).**
- **First two entries reserved (FAT-ID, etc) .**
- **-1 (or MAX unsigned integer) used as end of list marker.**
- **Allocate a cluster: linearly search for an entry == 0.**

	FAT
0	R
1	R
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0

FAT: Finding data blocks

- **FAT keeps file blocks linkage. For example:**
  - Free = 3,4,8, 9, 14, 15,... are free clusters
  - File A = 2   2, 7,6, 13 (-1 terminates)
  - File B = 5   5, 10, 11, 12 (-1)
- **Find n’th block of a file**

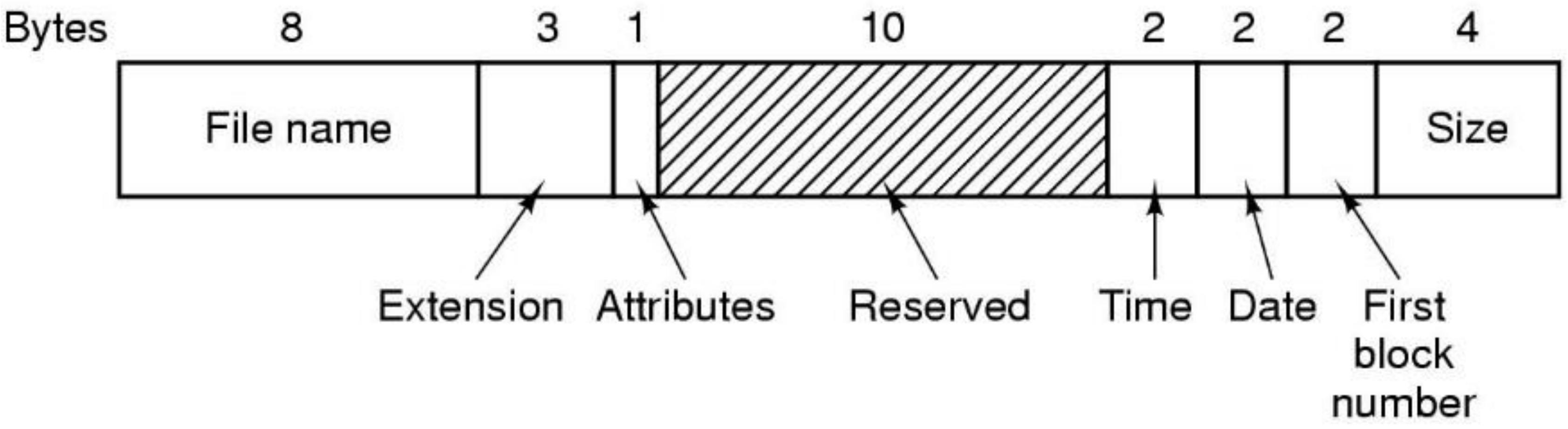
```
getblock(f,n):
     for (; n > 0 ; n--) {
          if (f == -1)
               return -1 ;   /* error */
          f = FAT[n];
     }
     return f;
```

- **Expensive when FAT is on disk!!!**
- **FAT needs to be working in main memory to be efficient.**

	FAT
0	
1	
2	7
3	0
4	0
5	10
6	13
7	6
8	0
9	0
10	11
11	12
12	-1
13	-1
14	0
15	0

FAT Directories

- 32-byte directory entries.
- Filenames are limited to 8+3 characters.. Smaller ones are left justified and padded with space.
- Attributes: read-only, archive, hidden, system
- Time of date represented with 2 bytes: correct upto +-1 second
- Date: Counts in three fields: day (5 bit), month (4 bits), year (7 bits).
  - year-since-1980 hence
    - Contains: Y2108 problem!
- File size: 2 bytes. Puts a 4GB limit.
- 10 bytes reserved for future use.



FAT: Directory Structure

- FAT directories are stored in data blocks as an array of file entries in the form:

```
struct fat_dir_entry {
        uint8   name[8] ;    /* file name */
        uint8   ext[3] ;     /* file extension */
        uint8   attr;        /* attributes, RO, Hidden, System, Subdir */
        uint8   lcase;       /* Case for base and extension */
        uint8   ctime_cs;    /* Creation time, centiseconds (0-199) */
        uint16  ctime;       /* Creation time */
        uint16  cdate;       /* Creation date */
        uint16  adate;       /* Last access date */
        uint16  starthi;     /* High 16 bits of cluster in FAT32 */
        uint16  time, date;  /* time, date */
        uint16  start;       /* first cluster */
        uint32  size;        /* file size (in bytes) */
}
```

- First cluster of the file = starthi << 16 & start.
- For the rest, follow FAT.

FAT: Directory Structure

- **First byte of name field marks:**
  - End of directory table (0x00)
  - Deleted file, entry can be reused (0xE5)
- **Deleting a file:**
  - deallocate its FAT chain, set: name[0] = 0xE5
- **For file names longer than 8+3, VFAT adds dummy entries (1 per 13 characters) preceding original entry (LFN, Long File Name)**
- **Directory table grows like a file.**
  - If all entries are deleted in a cluster, directory table may shrink as well.
- **File lookup:**

```
fat_directory_entry D[MAXENT];
    for (all clusters “D” of directory) {
        for (i = 0; i < MAXENT; i++) {
            if D[i] is invalid continue;
            if (D[i].name[0] == 0)
                return -1 ; /* not found */
            if (strcmp(filename == D[i].name))

                return D[i];
    }   }
```

Block/Cluster Size

- **Block size affects and is affected by:**
  - Storage device native block size. (no smaller read, smaller writes require, read, update in mem, write
  - VM page size (caching)
- **Filesystems may choose a cluster of blocks as unit to support larger disks and file sizes.**
- **Large cluster size → Internal fragmentation**
- **Small cluster size bad → locality.**

FAT: Performance

- **File random access → Linear scan of FAT**
- **Changing attributes of a file → change directory entry**
- **FAT limits disk size.**
  - Solution: increase cluster size
- **Large clusters → internal fragmentation**
  Small clusters → small file system, large FAT
- **Size field in directory entry is 32 bits.**
  - Largest possible file size is 4GB.
- **FAT gets “fat” :) , even in memory, difficult to manage.**

FAT versions

- **FAT limits disk size.**
  - Solution: increase cluster size:
  - Large clusters → internal fragmentation
  - Small clusters → small file system, large FAT
- **Size field in directory entry is 32 bits. Largest possible file size is 4GB.**
- **FAT gets “fat” :) , even in memory, difficult to manage.**

	Max FAT entries	**Max FAT size in memory**	**Max volume for 512B cluster**	**Max volume for 16KB clusters**	**Max volume for 32KB clusters**
FAT12	4K	12KB	2MB	64MB	128M
FAT16	64K	128KB	64MB	2GB	4GB
FAT32	256M	1GB	512GB	16TB	32TB

More info about FAT

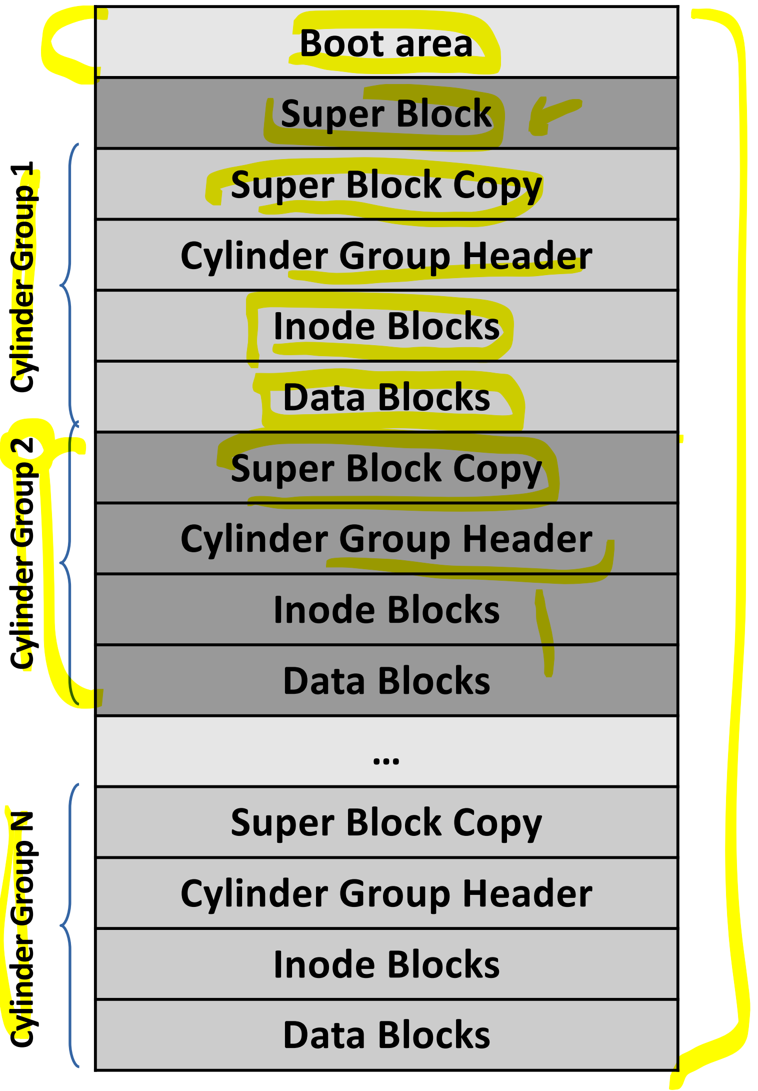
- https://en.wikipedia.org/wiki/File_Allocation_Table

- https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html

Unix File System

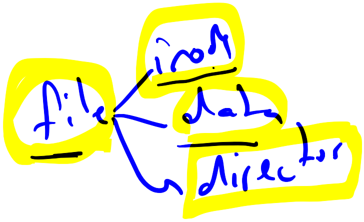
UFS: Organization

- Boot blocks: reserved for second stage boot loaders
- Super Block: FS meta data,
- Cylinder Group Header: Cylinder Group Description
- inode blocks: Blocks containing i-nodes
- Data blocks: Blocks containing file and indirect pointer data

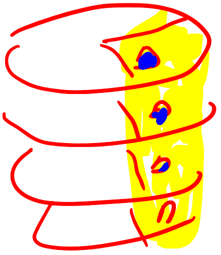


UFS: Organization

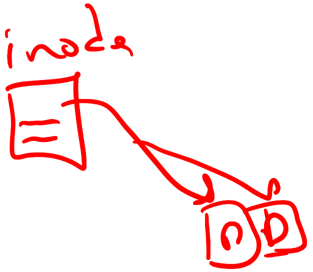
- **Super Block:**
  - Disk geometry.
  - number of CG's, inode and data blocks per CG.
  - FS mount statistics and state.
  - It is replicated in each CG for recovery (in case of a bad block)



- **Cylinder Groups (CG) :**
  - Provide spatial locality for hard disks. Minimize head moves.
  - UFS tries to allocate inodes and data, directories and inodes in the same cylinder group if possible.
  - Each CG has a bitmap for block allocation. N inodes block M data blocks: N+M/8/BLOCKSIZE blocks. → data block bitmap
  - Bitmap blocks are followed by i-node blocks and data blocks.
  - Alert! The number of inodes and data blocks are fixed at FS creation.



UFS: inode structure



- **inode:**
  - Each file has an inode block.
  - Multiple inode blocks fit in a single block (typical value: 128bytes).
  - Inode contains all file attributes and pointers to data blocks.
  - Data block pointers are kept as a tree like structure.

2^7 inode
4KB block
2^{12}

```
struct inode {
     uint16  mode;              /* file type and permissions */
     uint16  nlinks;            /* number of hard links */
     uint16 uid, gid;      /* owner and group ids */
     uint64 size;          /* total file size */
     timeval atime, mtime, ctime;  /* last access, modification, change timestamps */
     fs32 direct[UFS_NDIRECT]; /* pointers to direct data blocks  (12)  */
     fs32 indirect;                /* pointer to indirect index data block */
     fs32 double_indirect;     /* pointer to double indirect index block */
     fs32 triple_indirect;         /* pointer to triple indirect index block */
     uint32 flags;                 /* file flags, append only etc */
     uint32 blocks;                /* number of blocks used  */
     …
}
```

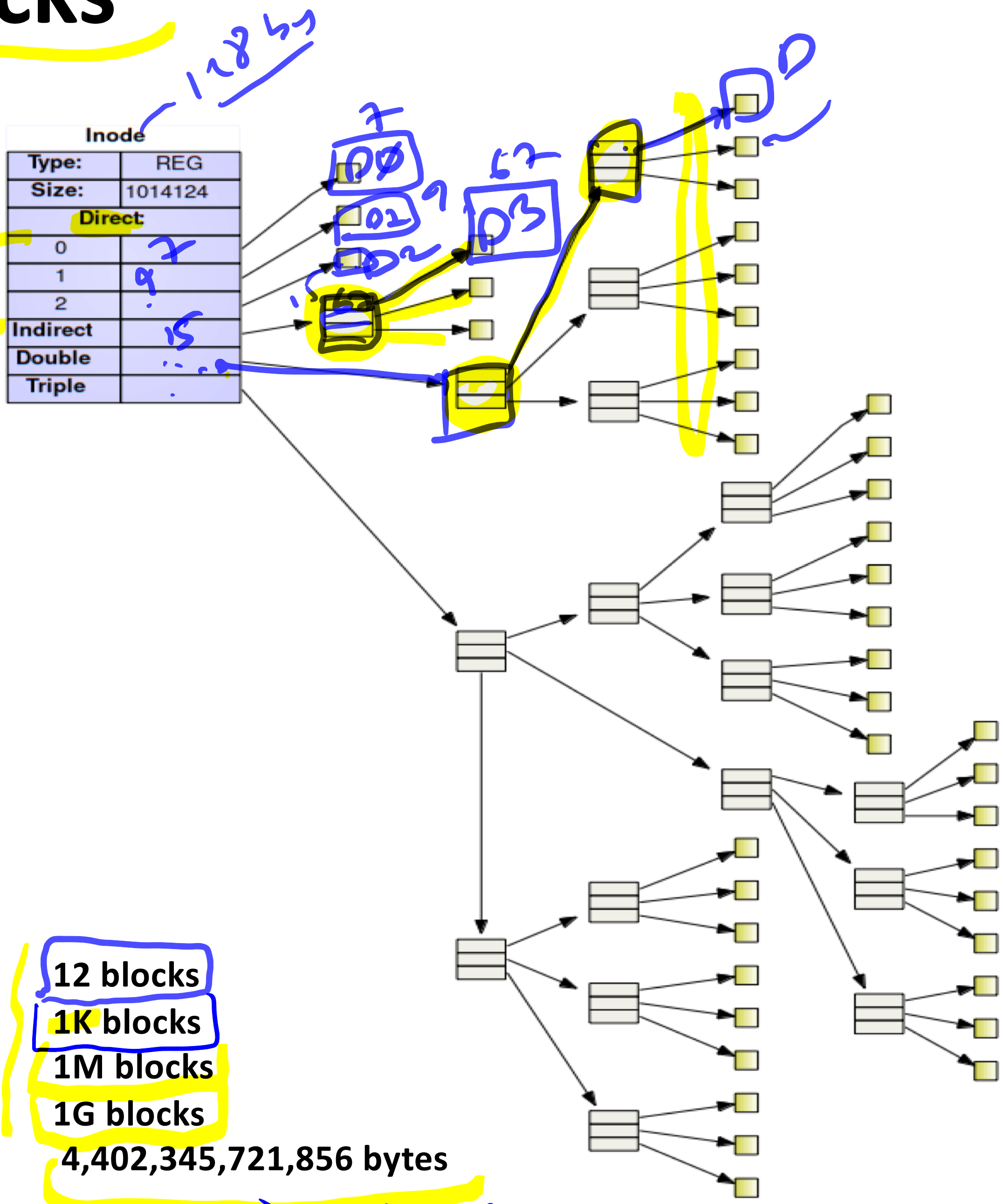
UFS: uid/gid and mode

- Uid/gid: who owns the file? User/group
- Permissions: which type of accesses granted for different groups.
  - Traditional Unix: 12 bits: `ugtrwxrwxrwx`
    - a **rwx** for owner, same group, and other processes
  - ACL and Windows: Each file has an Access Control List where each user/group can be granted or revoked access.
- Timestamps: Last access, modification and attribute change times of the file

D0, D1, D2 . D3 . . .

UFS: inode data blocks

- For small data, direct blocks are used
- An indirect block contains an array of data block pointers
- If file is larger, double indirect block contains array of pointers to indirect blocks
- For larger files, triple indirect pointers contains pointers to double indirect pointers
- For 4K block size, 4 byte block pointers and 12 direct blocks. Maximum file size is:



Direct:	12 blocks
Indirect: 4K/4 =	1K blocks
Double: 1K*1K =	1M blocks
Triple: 1K*1K*1K =	1G blocks
Total: 4TB	4,402,345,721,856 bytes

$(12 + 1K + 1M + 1G) \times 4KB$

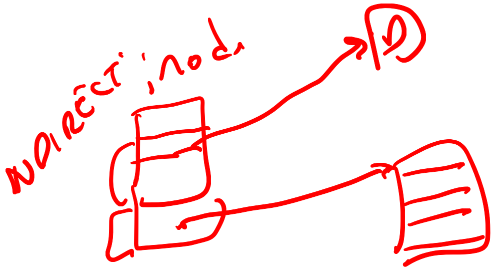
file: D(0) D[1] .. D(N)

UFS: File block mapping

```
fs32 indblk[1024];
if N < NDIRECT then
        return inode->direct[N]
N = N-NDIRECT
if N < 1K then                                   /* It is an indirect pointer */
        read inode->indirect to indblk   /* read indirect block */
        return indblk[N]
N = N-1024
if N < 1K*1K                                     /* it is a double indirect pointer */
        read inode->double_indirect to indblk  /* read D.Ind. block */
        read indblk[N/1024] to indblk     /* read indirect block */
        return indblk[N % 1024]
N = N – 1024*1024
        read inode->triple_indirect to indblk  /* read T. Ind.  block */
        read indblk[N/(1024*1024)] to indblk   /* get D.ind. block */
        read indblk[N%(1024 * 1024)/1024] to indblk /* get indirect block */
        return indblk[N%(1024 * 1024) % 1024]
```

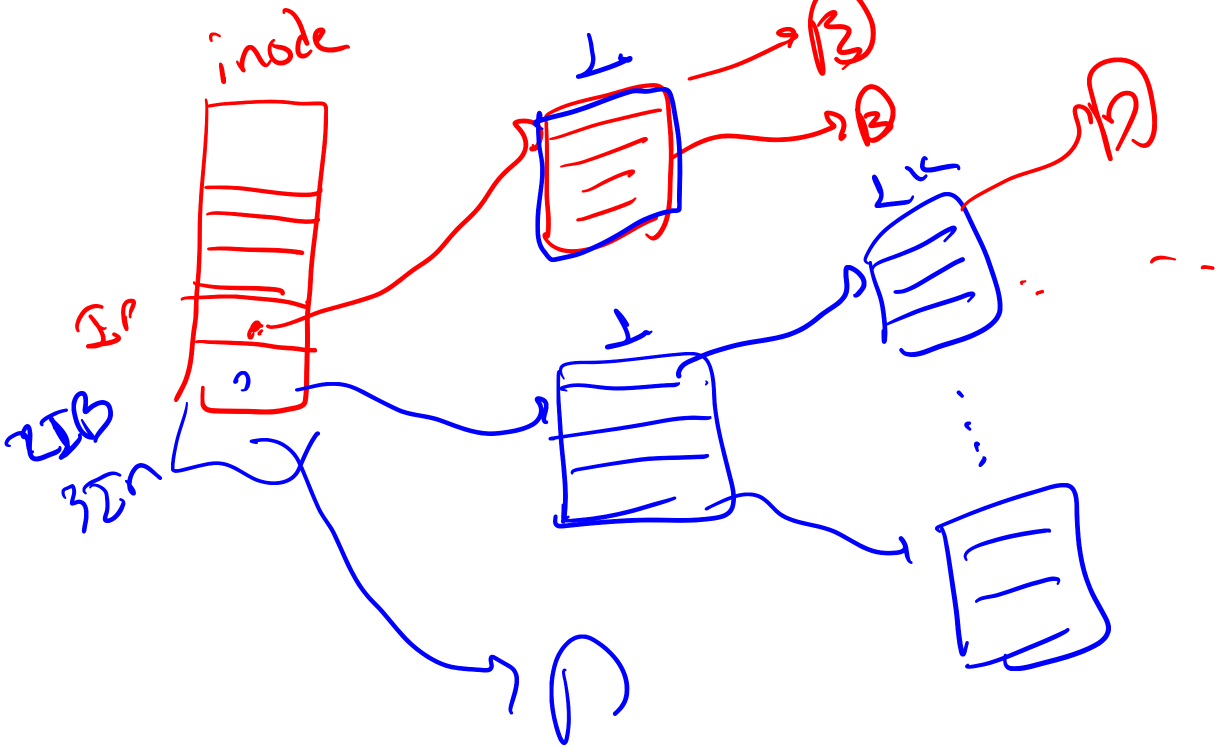
0 < N < NDIRECT

NDIRECT < N < 1K + NDIRECT

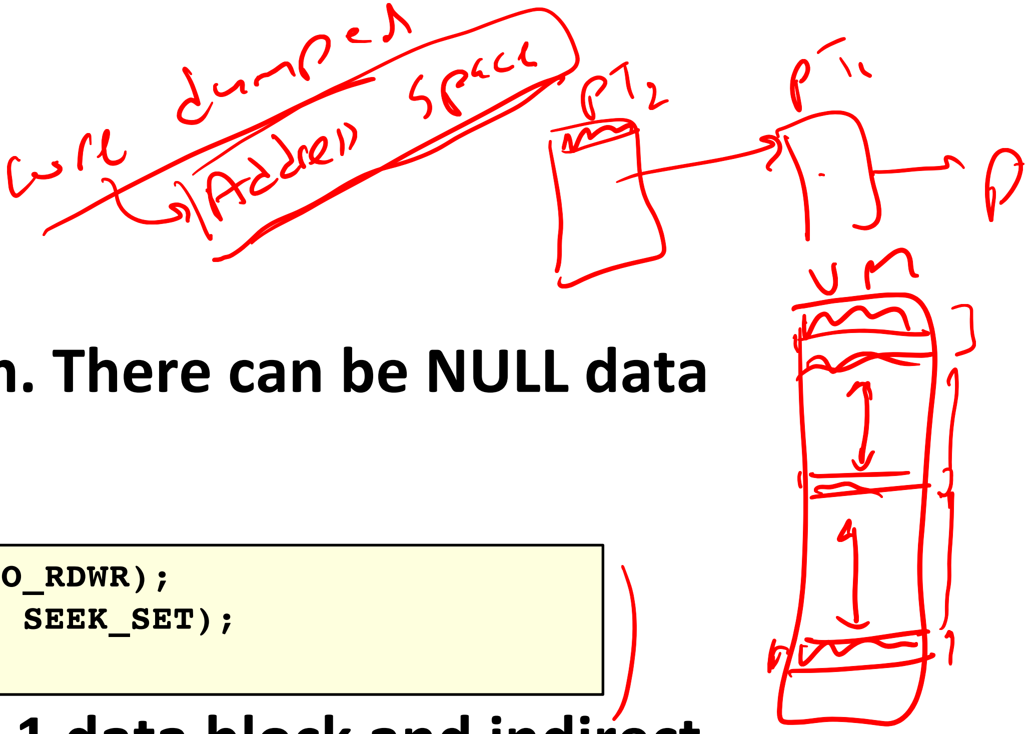


UFS

- A 0 value on block pointer indicates that block is not used.
- Indirect pointers are not allocated if they are not used.
- Largest possible file requires 1+ (1+1K)+(1+1K+1K*1K) indirect blocks besides data blocks.
- Smallest file uses no data blocks (direct pointers are NULL)



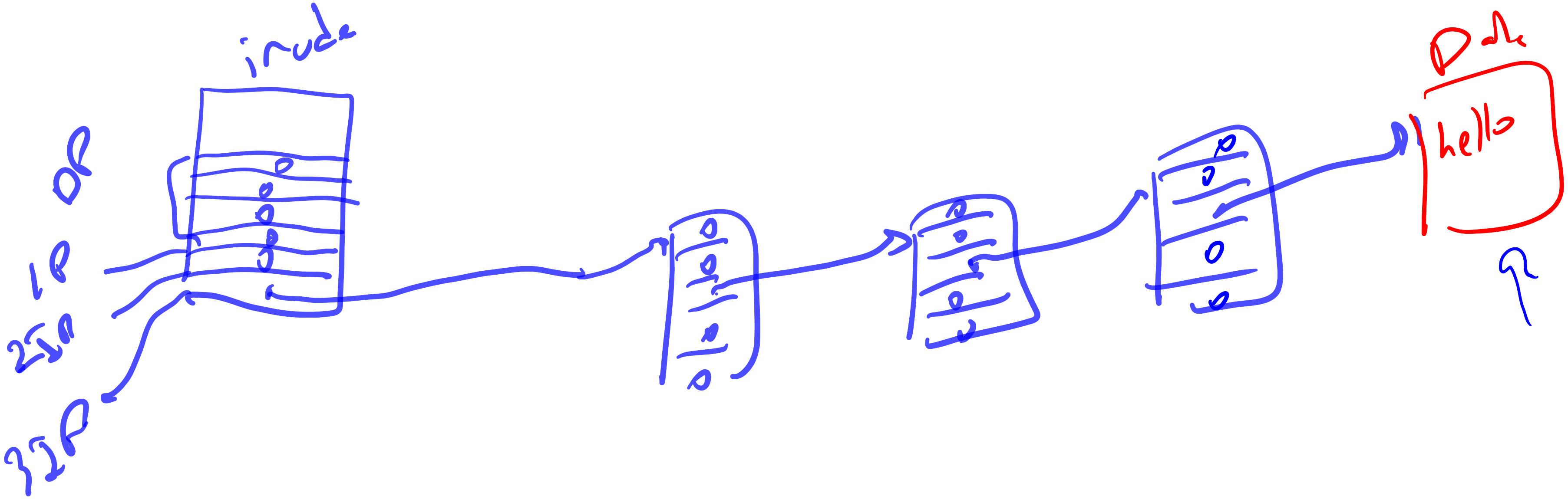
UFS: Holes



- Supports holes in filesystem. There can be NULL data blocks inside the file. Try:

```
fd = open("newfile.txt", O_RDWR);
lseek(fd, 1024*1024*1024, SEEK_SET);
write(fd, "hello\n", 6);
```

- File size is 1G however only 1 data block and indirect blocks in the path are used.



Directory = Special file
↳ inode
↳ data

UFS: Directory Structure

- Directory is nothing but a sequence of filename+inode values. Records size is not fixed.

```
struct dirent {
    uint32      d_inode;
    uint16      d_reclen;
    uint16      d_namelen;
    char        d_name[MAXLEN+1];
};  /* record size is truncated at d_reclen bytes */
```

4				2		2		4							
144123				12		1		.	x	x	x	542332			
12		2		.	.			21401231				16		7	
D	e	s	k	t	o	p		1231441				16		5	
h	w	2	.	c				454342				20		17	
c	e	n	g	3	3	4	-	f	i	l	e	.	p	p	t
x				4341121				12		3		h	w	2	

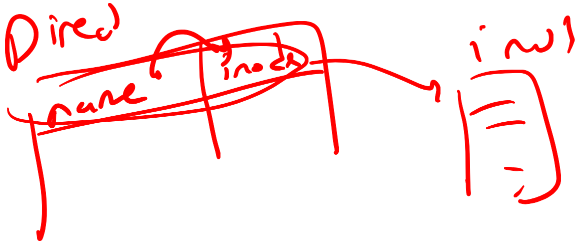
inode #

ls –I output

.	144123
..	542332
Desktop	2140123
Hw2.c	1231441
ceng334-file.pptx	454342
hw2	4341121

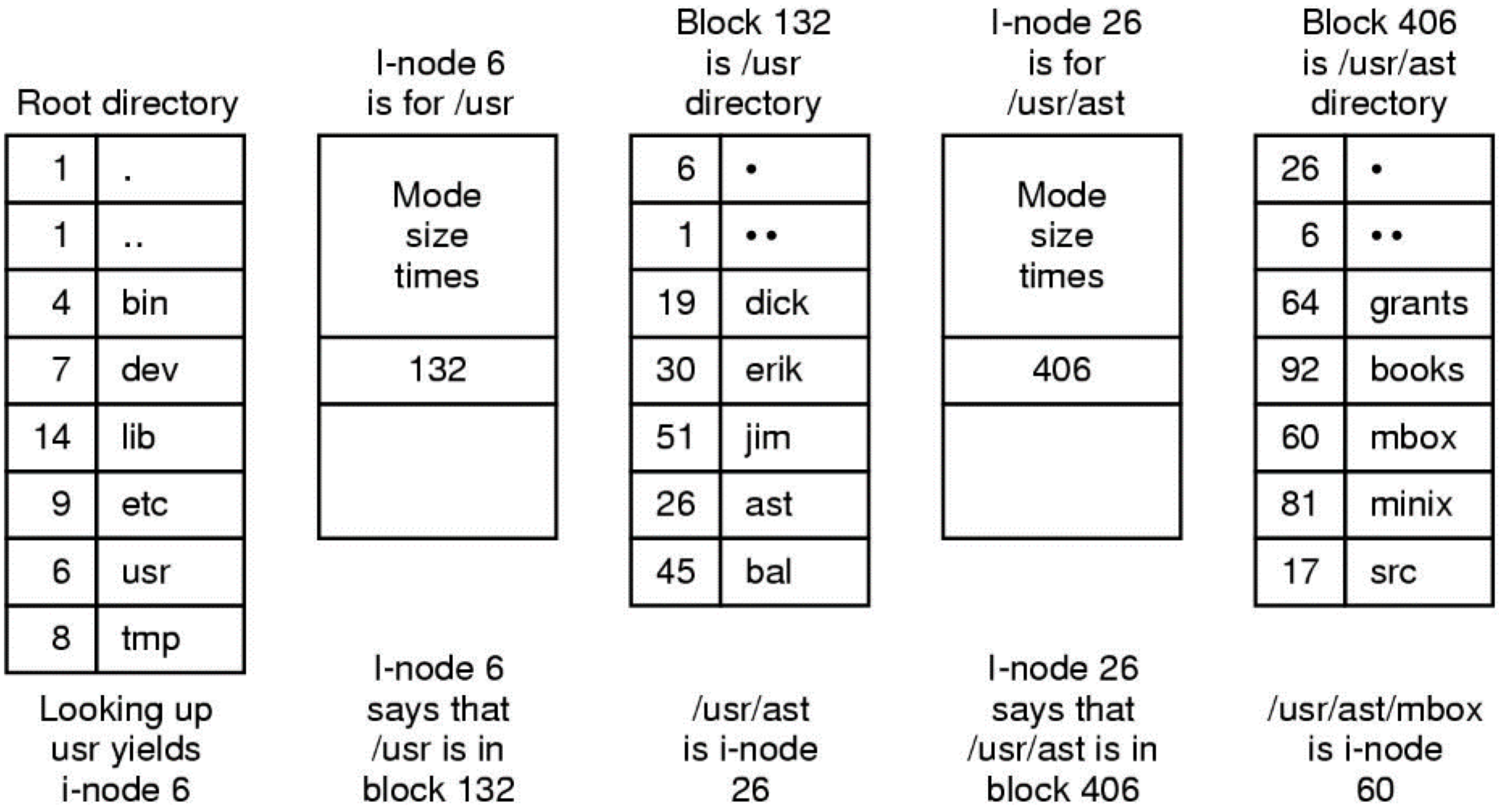
hw2.c

UFS: Lookup



- File lookup in a directory requires sequential traversal of full directory content.
- Reads/updates of a directory are mostly carried out in cache but still expensive for directories with large number of files.
- Deleting a file is similar to deleting a line from a text file.
- OS keeps a directory cache, mapping from vnode+filename to new vnode to increase the performance.

Traversing Unix Directory Structure



Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up usr yields i-node 6

I-node 6 is for /usr

Mode size times
132

I-node 6 says that /usr is in block 132

Block 132 is /usr directory

6	•
1	••
19	dick
30	erik
51	jim
26	ast
45	bal

/usr/ast is i-node 26

I-node 26 is for /usr/ast

Mode size times
406

I-node 26 says that /usr/ast is in block 406

Block 406 is /usr/ast directory

26	•
6	••
64	grants
92	books
60	mbox
81	minix
17	src

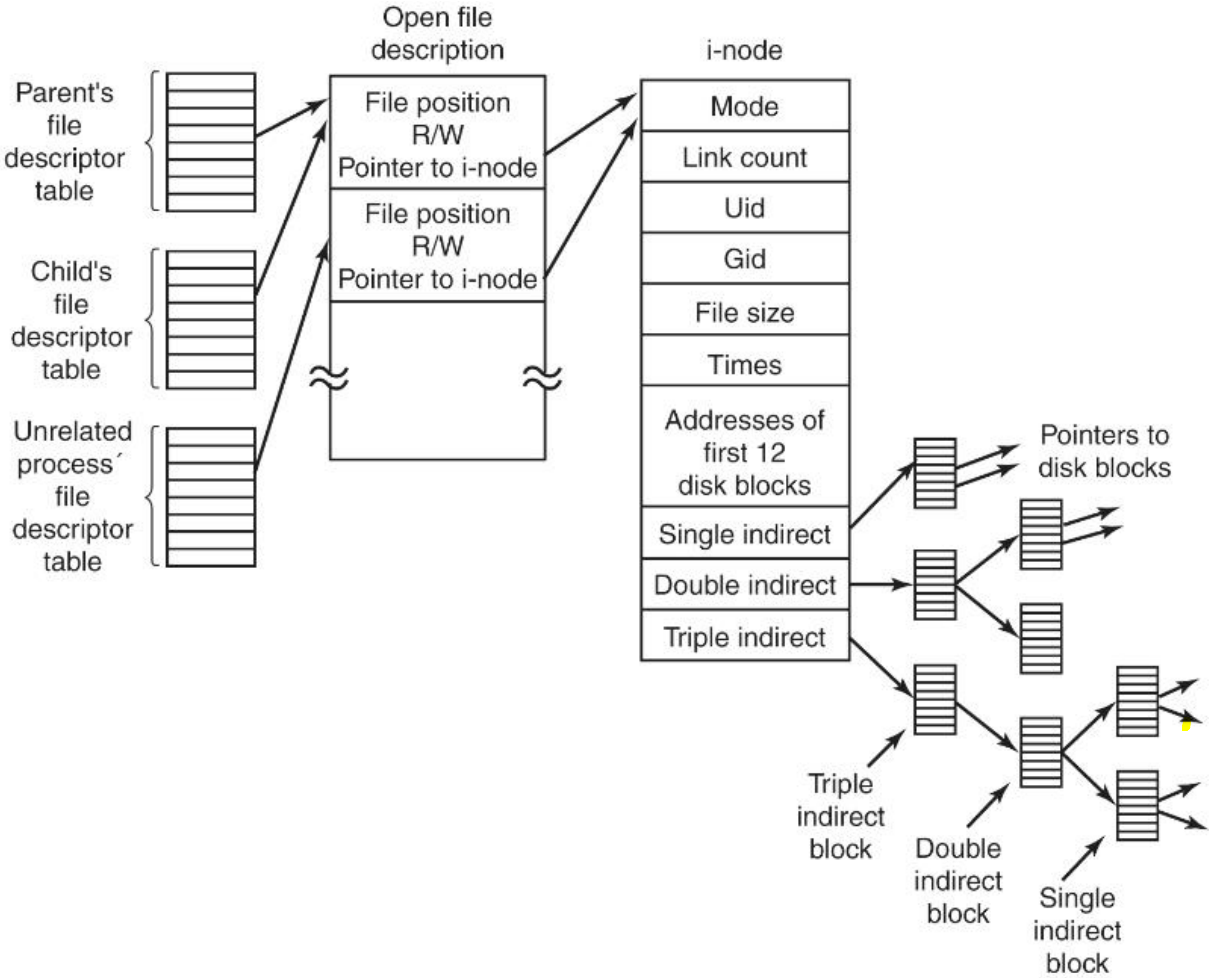
/usr/ast/mbox is i-node 60

- The steps in looking up */usr/ast/mbox*

Traversing Unix Directory Structure

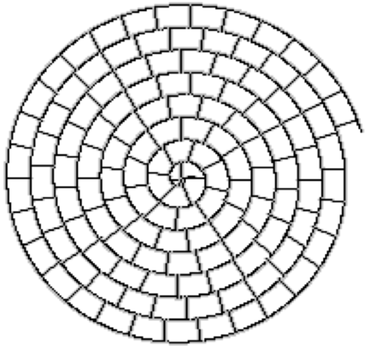
- **Suppose we want to read the file /usr/ast/mbox**
- **Location of a i-node, given i-number, can be computed**
- **i-number of the root directory known, say, 2**
- **Read in the i-node 2 from the disk into memory**
- **Find out the location of the root directory file on disk, and read the directory block in memory**
  - If directory spans multiple blocks, then read blocks until usr found
- **Find out the i-number of directory usr, which is 6**
- **Read in the i-node 6 from disk**
- **Find out the location of the directory file usr on disk, and read in the block containing this directory**
- **Find out the i-number of directory ast (26), and repeat**

The Big picture: Accessing a File in UNIX



CD-ROM and the ISO 9660 filesystem

- The CD has one single track, starts at the center of the disk and spirals out to the circumference of the disk
- This track is divided into sectors of equal size
- The base standard defines three levels of compliance
  - Level 1 limits file names to 8+3 format. Many special characters (space, hyphen, equals, and plus) are forbidden
  - Level 2 and 3 allow longer filenames (up to 31) and deeper directory structures (32 levels instead of 8)
  - Level 2 and 3 are not usable on some systems, like MS-DOS

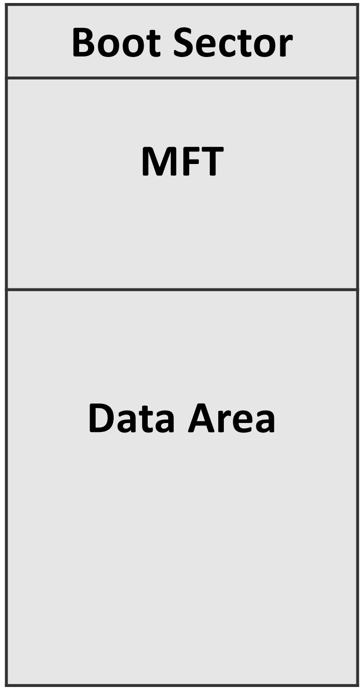


ISO 9660 Extensions

- **Rock Ridge**
  - Extensions to ISO-9660 file system
  - Favored in the Unix world
  - Lifts file name restrictions, but also allows Unix-style permissions and special files to be stored on the CD
  - Machines that don't support Rock Ridge can still read the files because it's still an ISO-9660 file system (they won't see the long forms of the names)
  - UNIX systems and the Mac support Rock Ridge
  - DOS and Windows currently don't support it
- **Joliet**
  - Favored in the MS Windows world
  - Allows Unicode characters to be used for all text fields (including file names and the volume name)
  - Disk is readable as ISO-9660, but shows the long filenames under MS Windows
- **HFS (Hierarchical File System)**
  - Used by the Macintosh in place of the ISO-9660, making the disk unusable on systems that don't support HFS

NTFS

- Introduced in Windows NT and used in Windows systems as default file system for hard disks.
- All metadata of FS is kept in Master File Table
  - A B tree on disk.
- MFT stores a record per file.
  - Each record is like an inode, contains file attributes and data block positions.
- MFT records have fixed size but can grow by extends, additional records.
- ${Mft} master file table, content of files and directories
- $(Bitmap} Allocation information of clusters
- Attributes are not fixed, new attributes can be added.
- Allocation information is stored in MFT as a bitmap by a Special filename $Bitmap.
- NTFS supports: hard links, sparse files, journal ($LogFile), encryption, compression, alternate data streams.
- Maximum partition size 256TB, maximum file size 16TB.



Boot Sector

MFT

Data Area

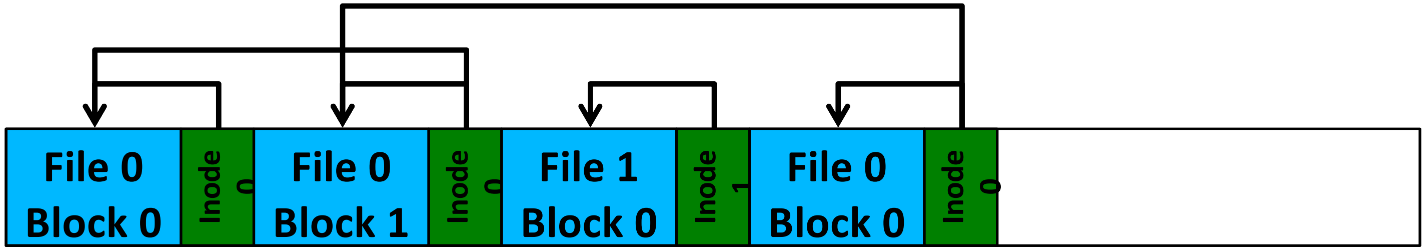
Log Structured File System (LSFS)

- **Hard disk:**
  - read/write direct access.
- **DVD's:**
  - read direct, write sequential,
  - requires full erase in order to write again
- **SSD/Flash disks:**
  - read direct access,
  - write direct for empty blocks,
  - not allowed for non-empty blocks,
  - erase work in large segments.
  - Segments have a limited lifetime for erase cycles.

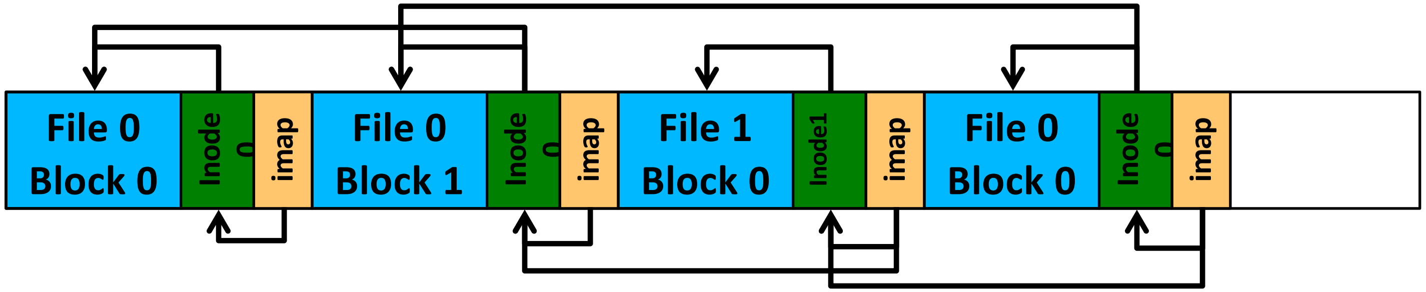
Log Structured File System

- Traditional filesystems: Meta-data is updated frequently.
- Log structured file systems do not overwrite blocks but keep a log of new versions of blocks.
- Each new write is appended at the end of the log.
- Latest written version of a block is the valid data

Log Structured File System



- **When new data block written, inode also changes.**
  - inode is written too.
- **Data block is overwritten.**
  - Writes a new version. Old block is not valid.
- **New inode version written.**
  - Old is invalid. Keep up-to-date versions of inode. In memory?
- **Keep an inode map, inode num. to block mapping. Write that too.**



Log Structured File System

- **Further enhanced by segment summaries to find inode maps correctly.**
- **On mount:**
  - Traverse all segment summaries for complete segments
  - Traverse all incomplete segments (only a few)
- **What happens when disk is full?**
  - Need to erase blocks: Completely obsoleted segments
- **Garbage collector passes over oldest segments, copies valid blocks to new areas and free whole segments.**
- **Some embedded systems and mobile systems working on flash uses LSFS based file systems**
  - like JFFS2, F2FS, YAFFS2 on Android.

Network and Distributed File Systems

- Transparent access to files in one or more remote hosts.
- Not a secondary storage data structure problem anymore. Network protocols are involved.
- Server utilities works on remote hosts to serve FS requests. Client is the OS and utilities making files available.
- Scenarios get complicated since network, server, and client may fail.
- Commonly used ones: NFS, SMBFS, Lustre, Google File System, AndrewFS.

Network File System (NFS)

- **Popular in simple Unix/Linux configuration works in many OS.**
- **NFS works stateless.**
  - Server does not keep states of clients.
  - Each client requests contains offset and size of the block to read/write.
  - Open looks up file path on remote server and gets a handle.
  - All following operations use same handle.
- **If handle known in advance, I/O can be done without opening the file.**
- **Stateless operation does not need crash recovery when one of the peers reboot.**
- **Client OS, file system dependent layer plays the client role and converts I/O requests into network requests.**

Multi-Filesystem Support

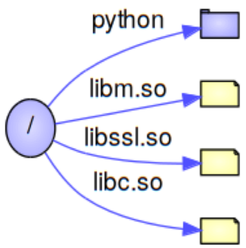
- Most OS's support multiple filesystems.
- Linux supports:
  - adfs affs afs autofs4 befs bfs btrfs ceph cifs coda configfs ecryptfs efivarfs efs exofs ext4 f2fs fat freevxfs fscache fuse gfs2 hfs isofs jbd2 jffs2 jfs minix ncpfs nfs nilfs2 nls ntfs ocfs2 omfs overlayfs pstore qnx6 reiserfs romfs squashfs sysv ubifs udf ufs xfs
  - Some designed for special storage requirements,
  - some are vendor specific, some exists for historical reasons.
- Systems may require multiple filesystems/partitions active in the same system.

Mounting a Filesystem

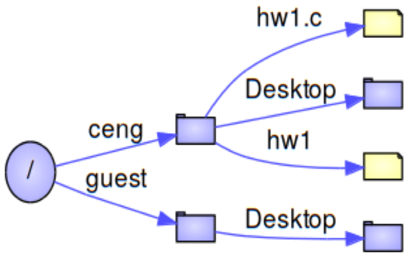
- Files may be organized into distinct filesystems.
  - E.g. system files vs. user files.
  - Files on a USB flash vs files on network.
- Purpose: logical isolation and providing filesystem grow (adding storage etc.)
- Unix/Linux support a `mount` operation for this isolation.
- Windows uses volume label, C:, D:, E:, ….
  - Newly attached filesystem gets a new label.

Mount

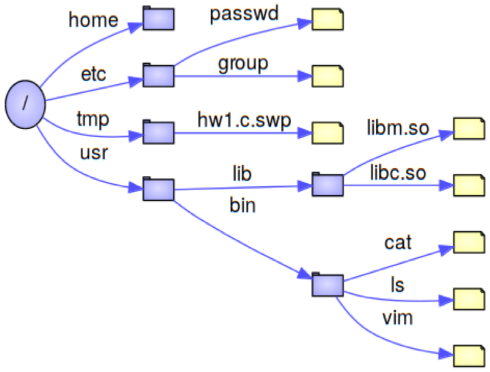
- Making another filesystem available under a directory in current file hierarchy.
- Existing content of directory is hidden and all accesses to that directory follow new filesystem transparently.



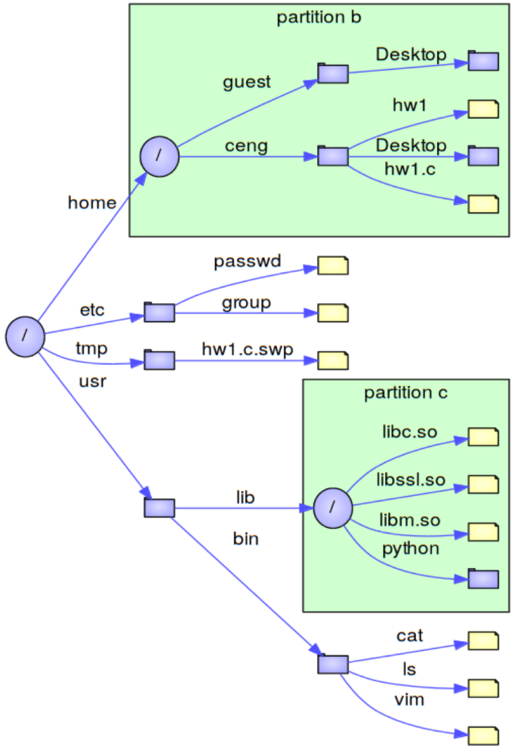
partition c



partition b



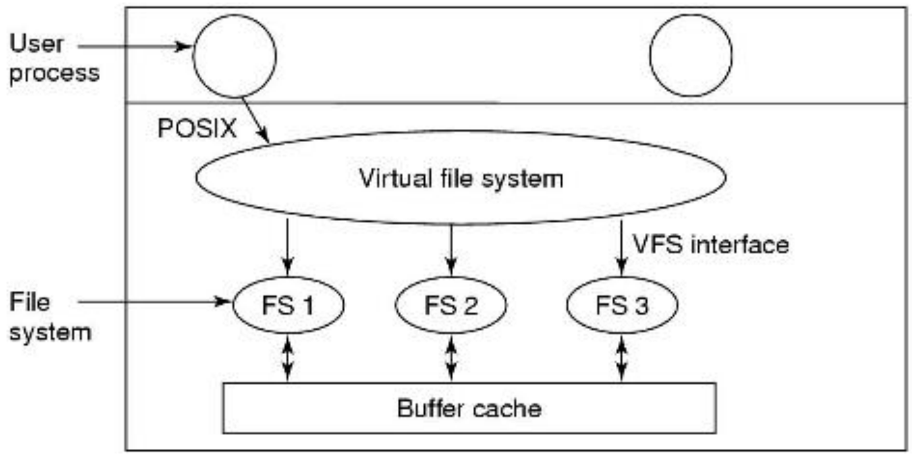
partition a



partition a

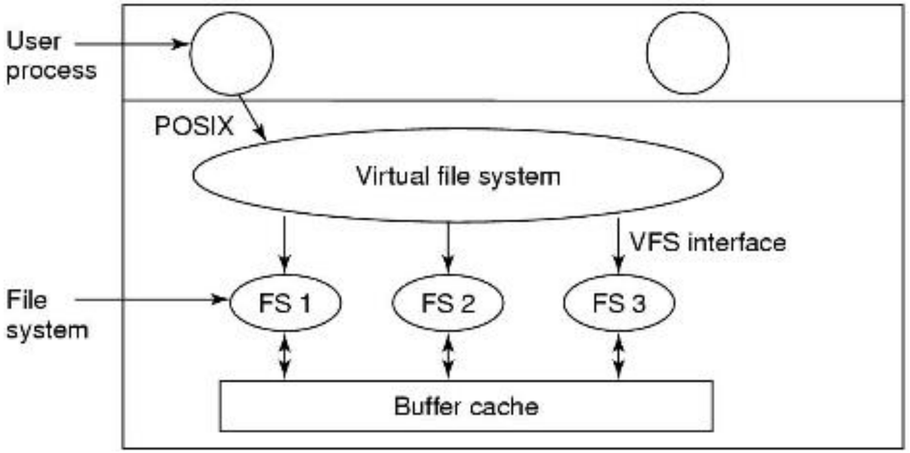
Virtual File System (VFS)

- Multiple supported filesystems bring a challenge to OS. Each file can be handled differently based on its filesystem type.
- Solution is an abstraction layer. Kernel calls filesystem independent parts, filesystems implement a uniform interface and file operations are mapped in this interface based on file system.



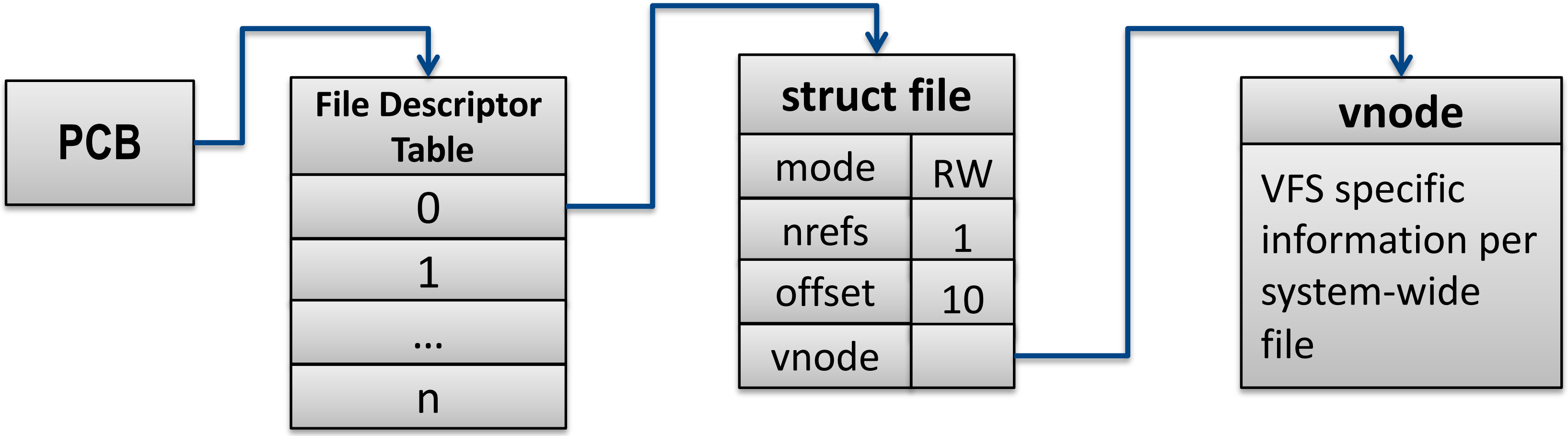
Virtual File System

- VFS translates the POSIX calls to the calls of the filesystems under it.
  - Filesystem independent part : for processes implementing POSIX interface
  - Filesystem dependent part : for concrete file systems



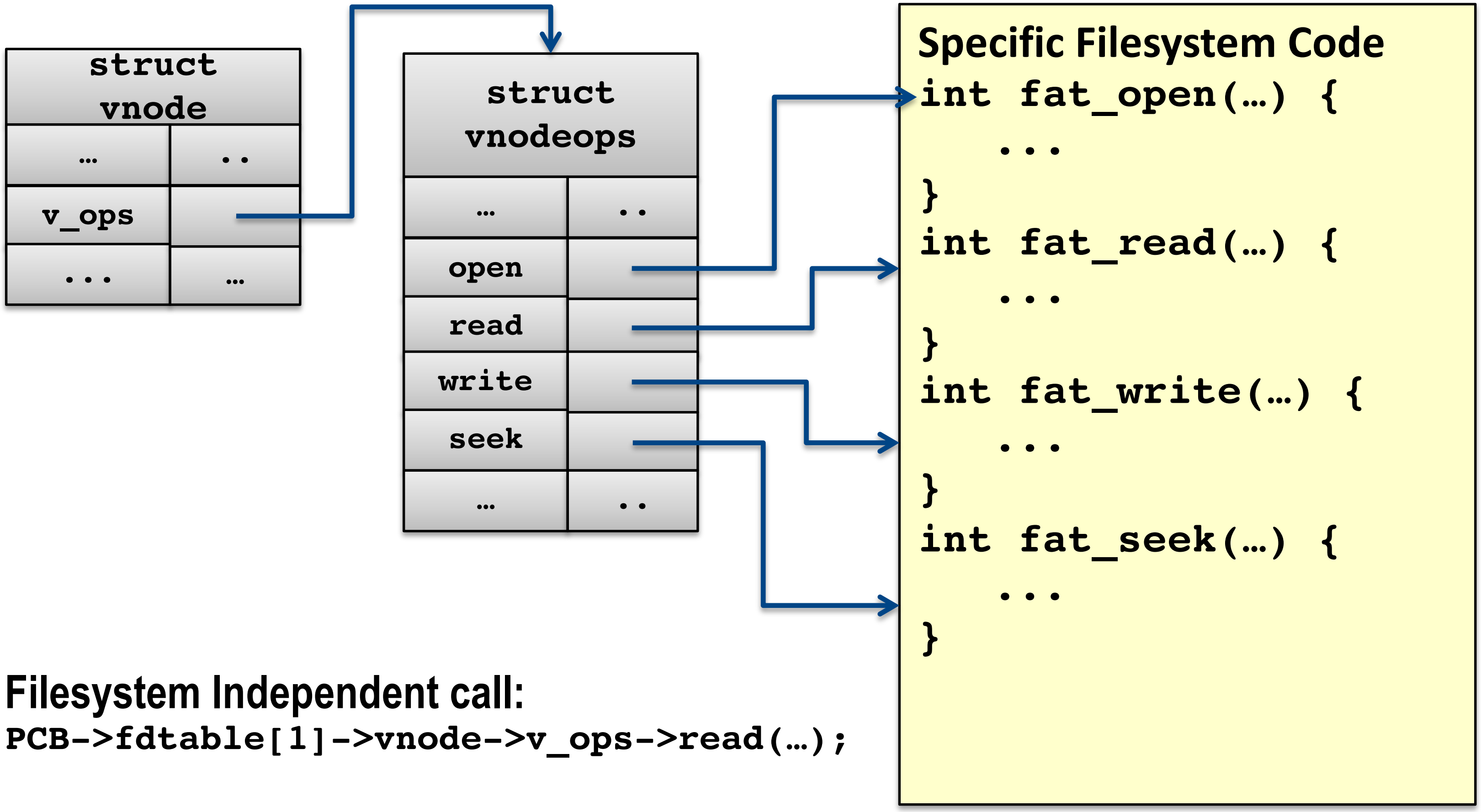
VNode

- In core structure for a file. System wide structure per file kept in kernel memory.
- Created when a file is opened by any process for the first time. All following open() operations by any processes use same vnode.
- Typically a pointer indirection in vnode is used to access filesystem dependent part.



VFS: Specific FS calls

- vnode points to a struct of function pointers keeping entry points of each actual FS operation



struct vnode	
…	..
v_ops	
...	…

struct vnodeops	
…	..
open	
read	
write	
seek	
…	..

Specific Filesystem Code

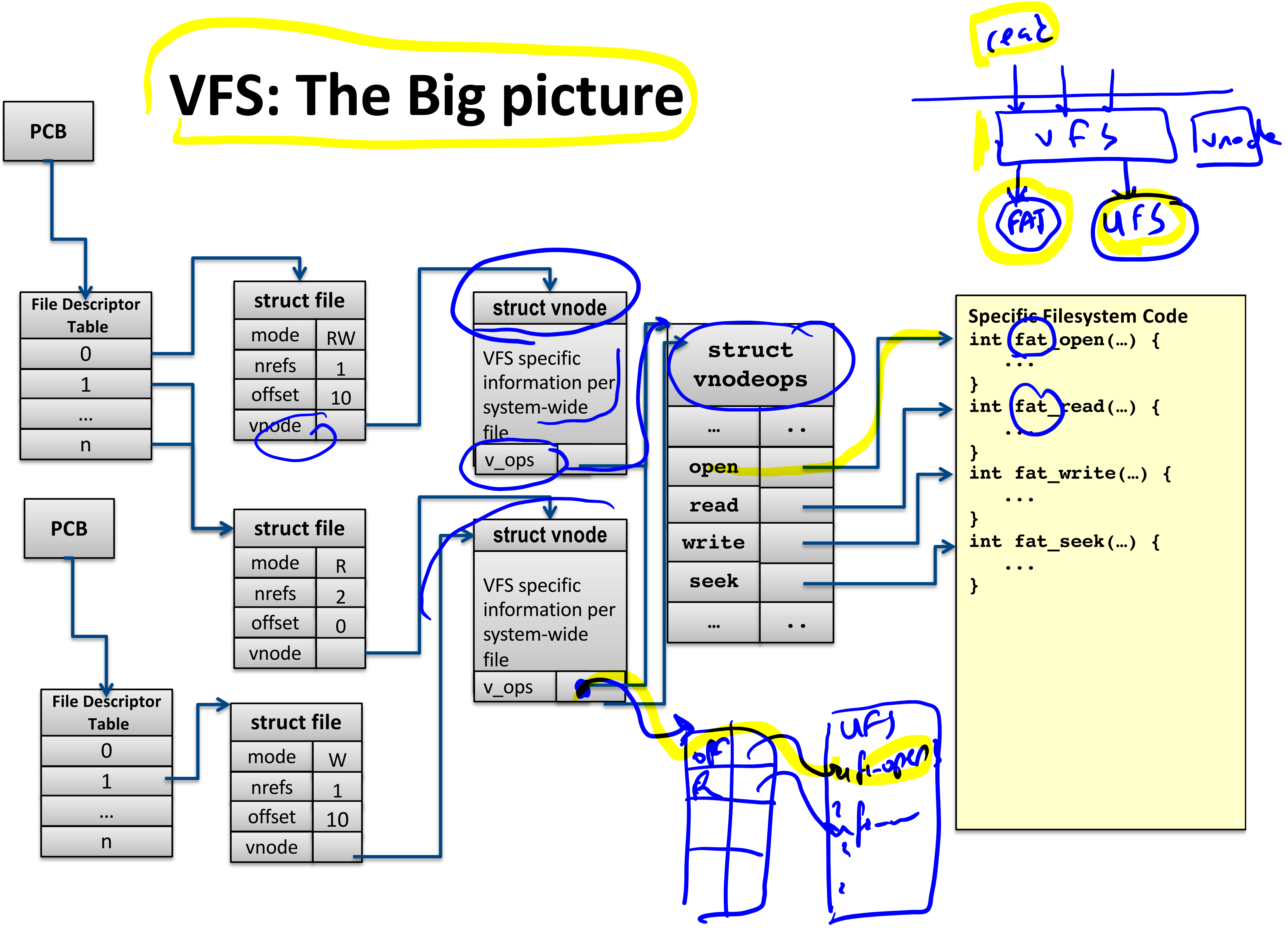
```
int fat_open(…) {
    ...
}
int fat_read(…) {
    ...
}
int fat_write(…) {
    ...
}
int fat_seek(…) {
    ...
}
```

Filesystem Independent call:

```
PCB->fdtable[1]->vnode->v_ops->read(…);
```

- **Each filesystem supported introduces its own set of implementations for file operations and operations structure.**
- **When a file of this filesystem type is accessed file operations structure is set to this structure.**

VFS: The Big picture



PCB

File Descriptor Table

File Descriptor Table
0
1
…
n

struct file	
mode	RW
nrefs	1
offset	10
vnode	

struct vnode

VFS specific information per system-wide file

v_ops

struct vnodeops	
…	. .
open	
read	
write	
seek	
…	. .

Specific Filesystem Code

```
int fat_open(…) {
    ...
}
int fat_read(…) {
    ...
}
int fat_write(…) {
    ...
}
int fat_seek(…) {
    ...
}
```

struct file	
mode	R
nrefs	2
offset	0
vnode	

struct vnode

VFS specific information per system-wide file

v_ops

PCB

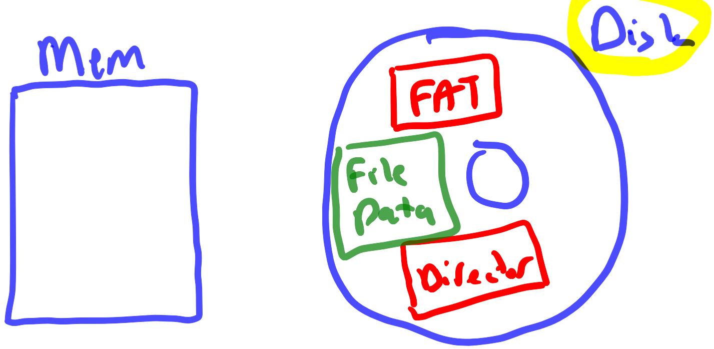
File Descriptor Table
0
1
…
n

struct file	
mode	W
nrefs	1
offset	10
vnode	

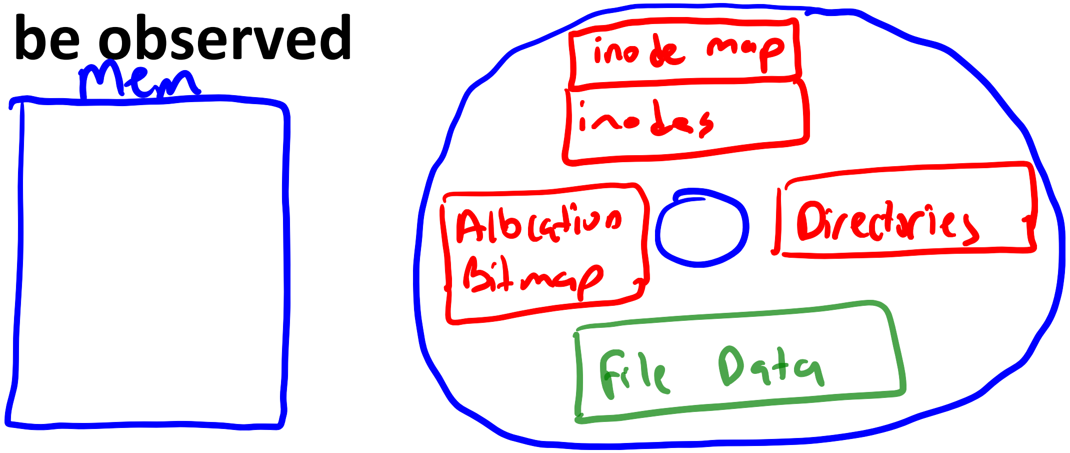
VFS- 2

- At boot time, the root filesystem is registered with VFS.
- When other filesystems are mounted, they must also register with VFS.
- When a filesystem registers, it provides the list of addresses of the functions that the VFS demands, such as reading a block.
- After registration, when one opens a file:
- open("/usr/include/unistd.h", O_RDONLY)
- VFS creates a v-node and makes a call to the actual filesystem implementation to return all the information needed.
- The created v-node also contains pointers to the table of functions for the concrete filesystem that the file resides.

Crash Recovery



- Performance of most File System designs rely on caching of metadata.
  - Such as FAT/allocation bitmaps, inodes.
- Caching helps a lot in speed but improper shutdown causes data kept in memory not written on disk.
- In addition to data, integrity of file system can be lost.
- Cached I/O of operations may end up in missing any subset of the operations above on disk in time of a crash.
- Problems as getting a garbage on disk to a dangling directory entry will be observed

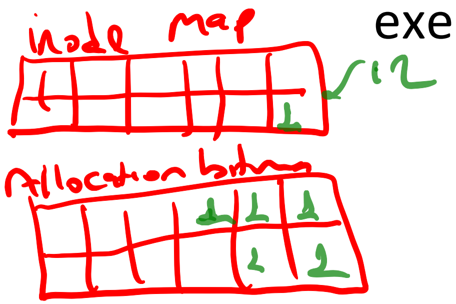


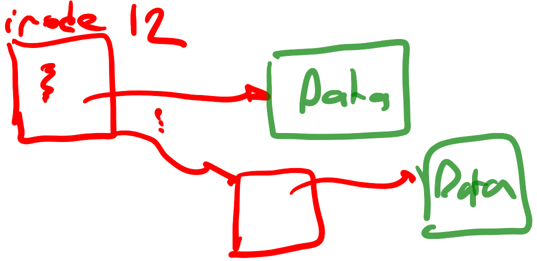
UFS: Filesystem operations a closer look

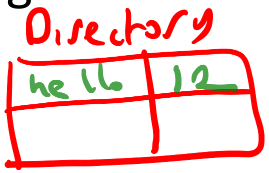
- **Deletion of a file:**
  - Clear data blocks to 0 (for security)
  - Mark data blocks as free on bitmap
  - Mark inode as free on bitmap
  - Delete directory entry referring the inode

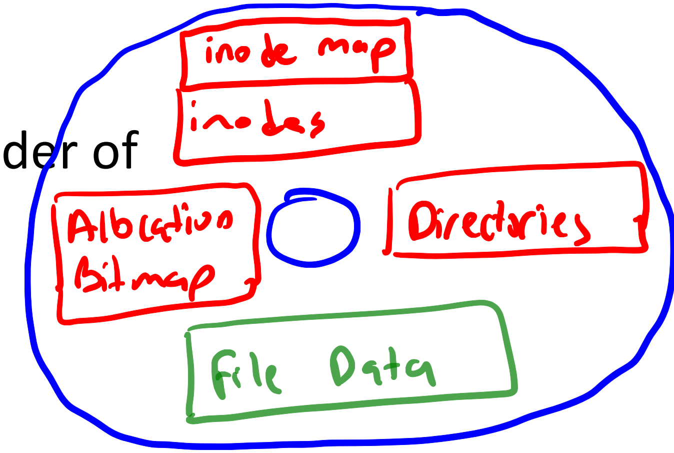
- **Creation of a file of 1 block size:**
  - Mark a data block allocated in allocation table
  - Write its content
  - Allocate an inode block in allocation table
  - Write inode content
  - Update directory block to get a reference to new inode

- **In the absence of crashes the order these steps taken do not matter.**
- **In the presence of crashes, however, it does!**
  - Note that due to I/O scheduling on the disk, the order of execution varies.







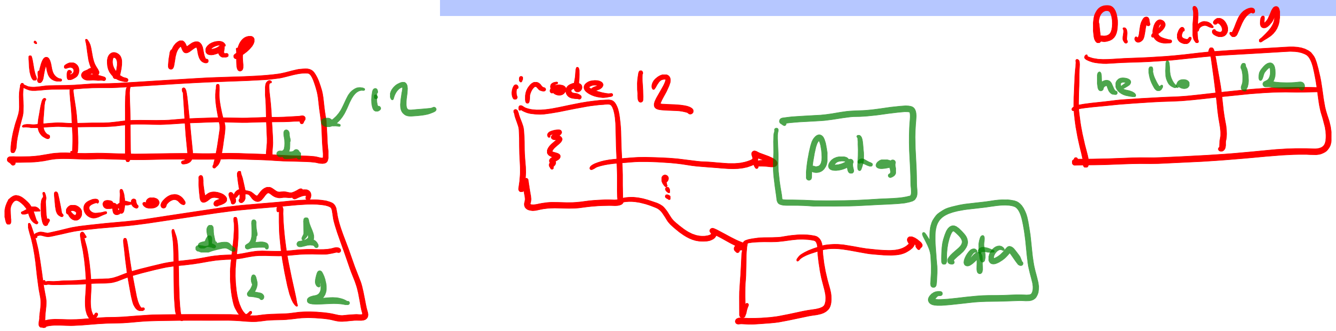


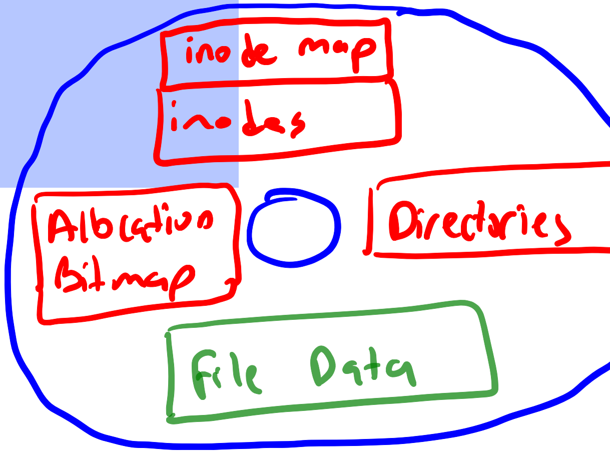
UFS - crash case 1

- **Deletion of a file:**
  - Clear data blocks to 0 (for security)
  - Mark data blocks as free on bitmap
  - Mark inode as free on bitmap
  - Delete directory entry referring the inode

not completed

completed

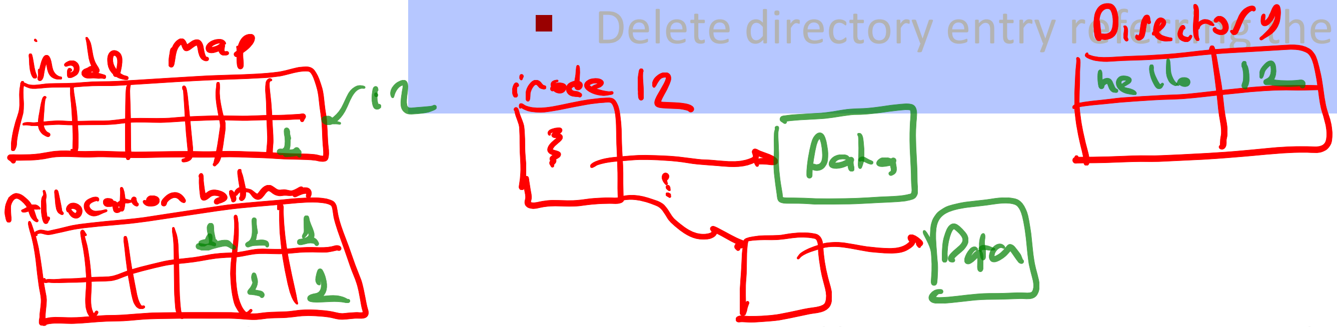


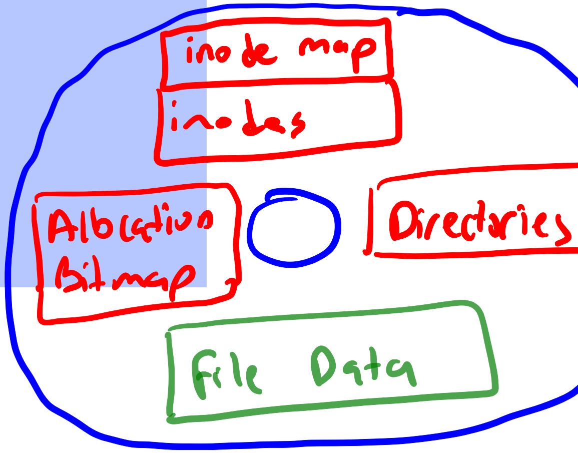


- **The inodes and file blocks will not be accessible from any file yet they will not be available for reassignment.**

UFS - crash case 2

- **Deletion of a file:**
  - Clear data blocks to 0 (for security)
  - Mark data blocks as free on bitmap
  - Mark inode as free on bitmap
  - Delete directory entry referring the inode

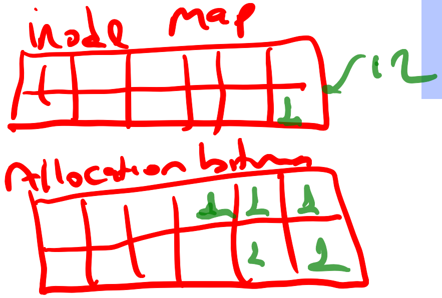


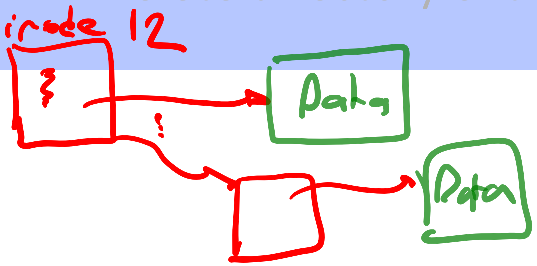


- **The directory node will point to an invalid inode or (if the inode is reassigned) point to a different file.**
- **The blocks of the file will not be available for reassignment.**

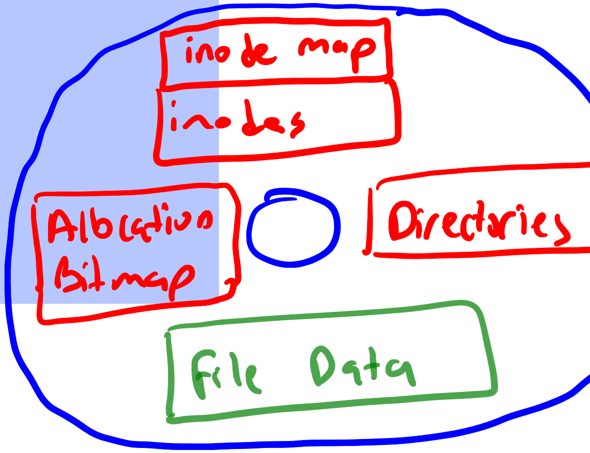
UFS - crash case 3

- **Deletion of a file:**
  - Clear data blocks to 0 (for security)
  - Mark data blocks as free on bitmap
  - Mark inode as free on bitmap
  - Delete directory entry referring the inode





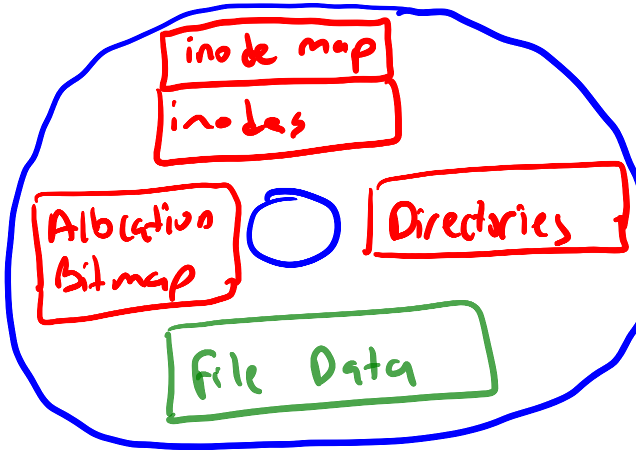
Directory
hello
12



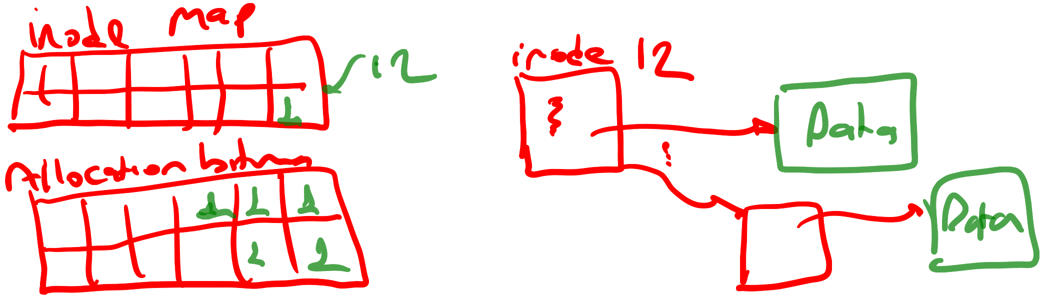
- **The blocks of the file will be available for reassignment.**
- **The content of the file may get overwritten since its blocks may be used as data blocks for other files.**

UFS - crash case 4

- **Deletion of a file:**
  - Clear data blocks to 0 (for security)
  - Mark data blocks as free on bitmap
  - Mark inode as free on bitmap
  - Delete directory entry referring the inode



- **File content is lost. All zeros.**



Directory
hello
12

Crash Recovery

- **Data loss is usually inevitable. It is a crash after all.**
- **File system integrity is a more serious problem.**
- **OS should reboot properly.**
- **Accessing a file should not end up another crash.**
- **3 solutions:**
  - Consistency check in reboot and try to recover (fsck, repair)
  - Soft updates (order disk operations such that consistency check is minimal and done at background)
  - Journalling (File system operations are written on a synchronous journal, recovery is done by replaying it)

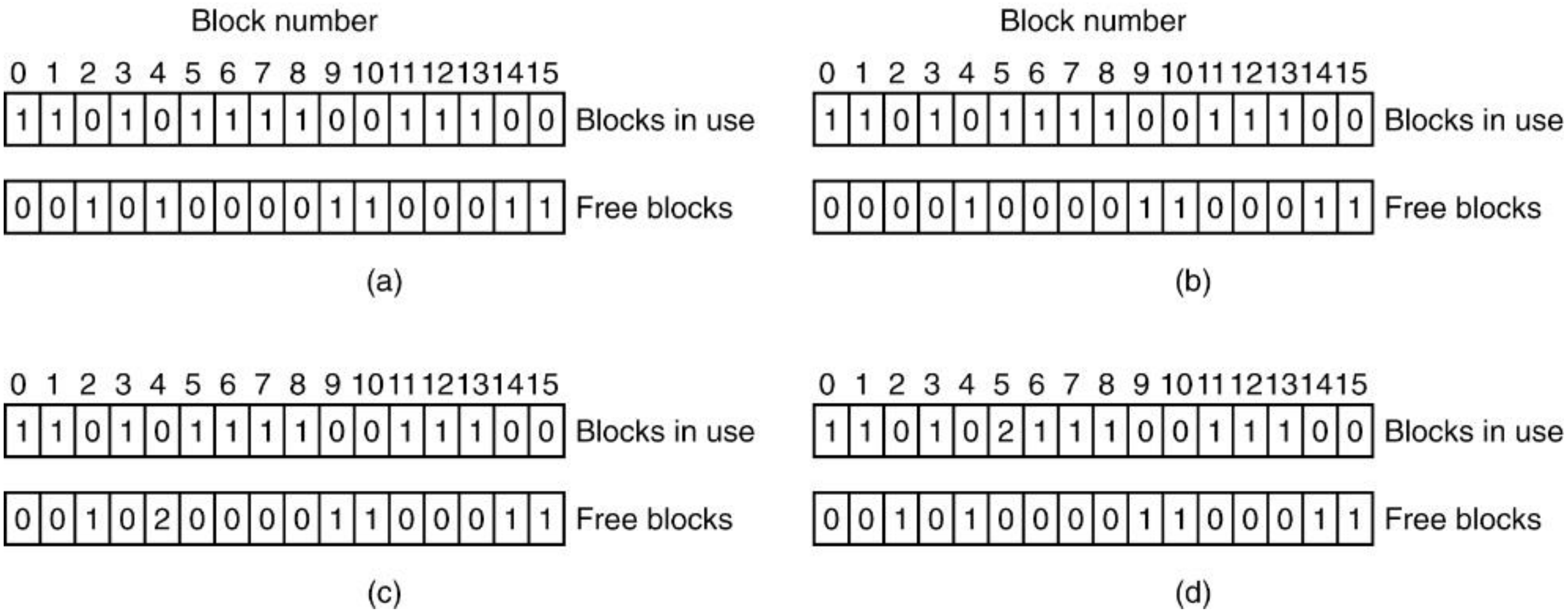
Consistency Check

- **A consistency check utility like fsck is executed on boot to check if file system is unmounted properly.**
- **Otherwise it makes a consistency check:**
  - All inodes and indirect blocks refer allocated data blocks
  - All directory entries refer allocated inodes.
  - Hard link count of inodes match total number of directory entries to that inode.
- **fsck repairs filesystem by:**
  - Mark data blocks allocated/truncated files (former may cause garbage data)
  - If inode contains valid data mark it as allocated, otherwise delete file.
  - Update inode hard link count. If no directory refers an allocated inode create a dummy file in /lost+found
- **fsck is too slow. Significantly slows down reboot time.**

Filesystem repair

- **Two tables, each containing a counter initialized to 0**
  - Blocks in use: How many times a block is present in a file
    - Read all the inodes using a raw device (not through the filesystem calls)
    - For each block that is referenced in the inode structure, increment the corresponding block use counter by one.
  - Free blocks:
    - Examine the free block list of free block bitmap structure
    - Each appearance of a free block increments the counter by one

Filesystem repair



Block number

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(a)

Block number

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(c)

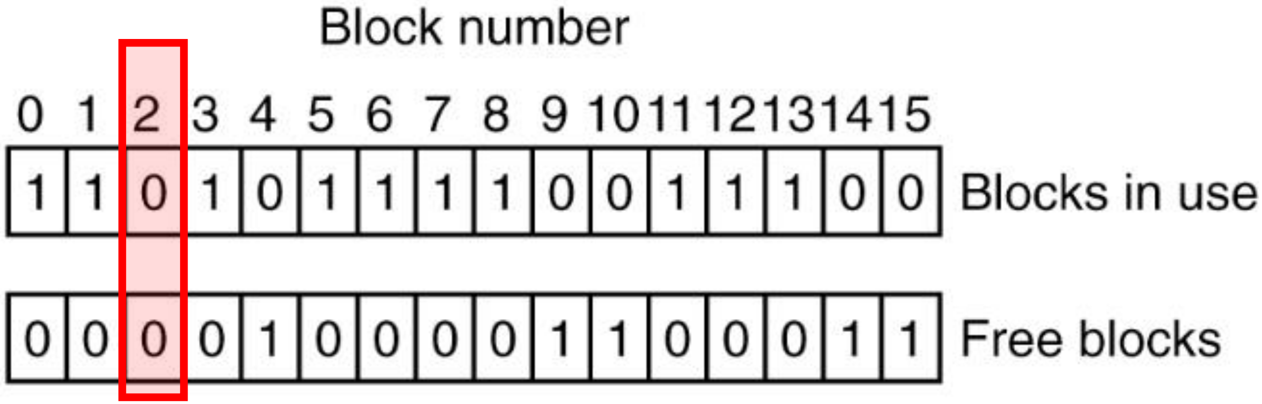
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(d)

- **File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.**

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Filesystem repair : Missing block



Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

(b)

- **Harmless but wastes space**
- **Action: Add the missing block to the free list.**

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

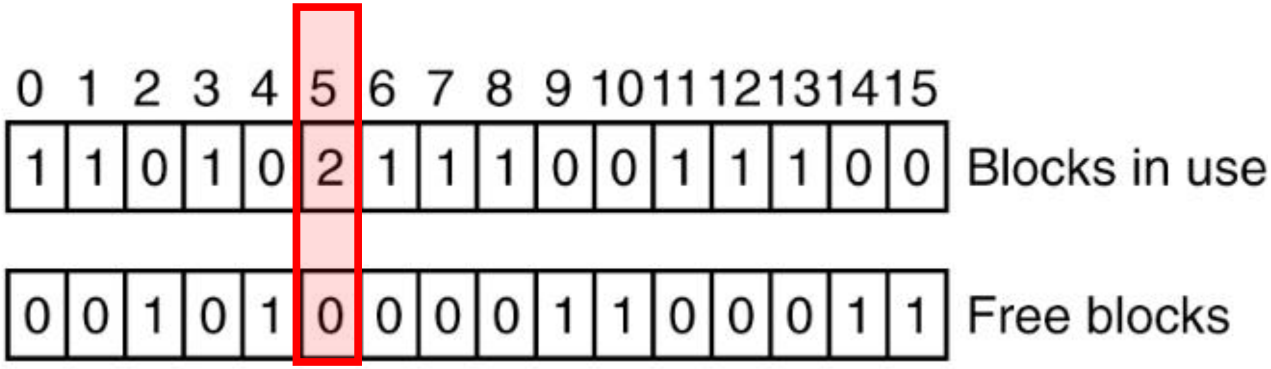
Filesystem repair : Duplicate block in free list

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(c)

- **Can only occur in linked list representation. Bitmap representation does not have this problem.**
- **Action: Rebuild the free list.**

Filesystem repair : Duplicate data block



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(d)

- **The worst thing that can happen! A data block appears in two different files..**
- **Action:**
  - Allocate a free block
  - Copy the contents into the new block.
  - Change the links such that each copy appears once in each file.
  - For sure, the contents of one of the files is garbled.
  - The filesystem is made to be consistent.
  - The user is informed.

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-**6006639**

Filesystem repair: Directory fix

- **Uses a table of counters per file (rather than per block)**
- **Starts from the root and traverses the tree**
  - For each inode, it increments the corresponding counter for that file
    - Remember due to hard links, a file can appear more than once
  - It then checks the link counts stored in the inodes to these values.
  - If the link count > counter
    - Even if the file is deleted by from all the directory entries, it will continue to exist.
    - Solution: correct the link count
  - If the counter > link count
    - Although the file is linked from, say, two directories, removal from one would cause the inode deleted leaving the other one invalid.
    - Solution: correct the link count
  - Special case: link count is 0, no directory refers it. Create a new file in a directory (lost+found)

Soft Updates

- **Also called ordered writes**
- **Device driver is forced to follow a specific write order for file system data:**
  - Never point to a structure before it has been initialized
    - e.g., an inode must be initialized before a directory entry references it.
  - Never re-use a resource before nullifying all previous pointers to it
    - e.g., an inode’s pointer to a data block must be nullified before that disk block may be re-allocated for a new inode.
  - Never reset the old pointer to a live resource before the new pointer has been set
    - e.g., when renaming a file, do not remove the old directory entry for an inode until after the new directory entry has been written.
- **Consistency problems reduce mostly to garbage. System may boot safely when fsck works in background to collect the garbage.**

Journalling

- Implemented by most of the contemporary file systems.
- Metadata (allocation bitmaps, inodes and directories) are important for integrity, so their operations are written on a special area called journal synchronously.
- Journal is forced in disk with some periods and cleaned up.
- On crash boot, journal is replayed to provide integrity.
- Journaling data is expensive, most file system implementations provide only metadata journal by default (recently written data blocks may be lost)

Other Features of File Systems

- **Compression:**
  - File system stores data on disk in compressed form
- **Encryption:**
  - Data blocks are encrypted for privacy of user
- **User and Group Quotas:**
  - File system tracks per user and per group usage of disk and reject exceeding the limit.
- **Logical volume management:**
  - File system can grow or shrink by adding new disk devices or removing them
- **Snapshots:**
  - Administrators can get snapshot of the file system.
  - Later that snapshot can be rolled back or mounted.
- **Copy on Write:**
  - File systems can use a base file system and only keep modified files/directories on device transparently.
- **Multiple streams and/or attributes:**
  - File systems can keep multiple streams of a file (i.e. revisions) and/or dynamic set of attribute-value pairs per file to keep extra information.