

---

# CS304 Operating Systems

---

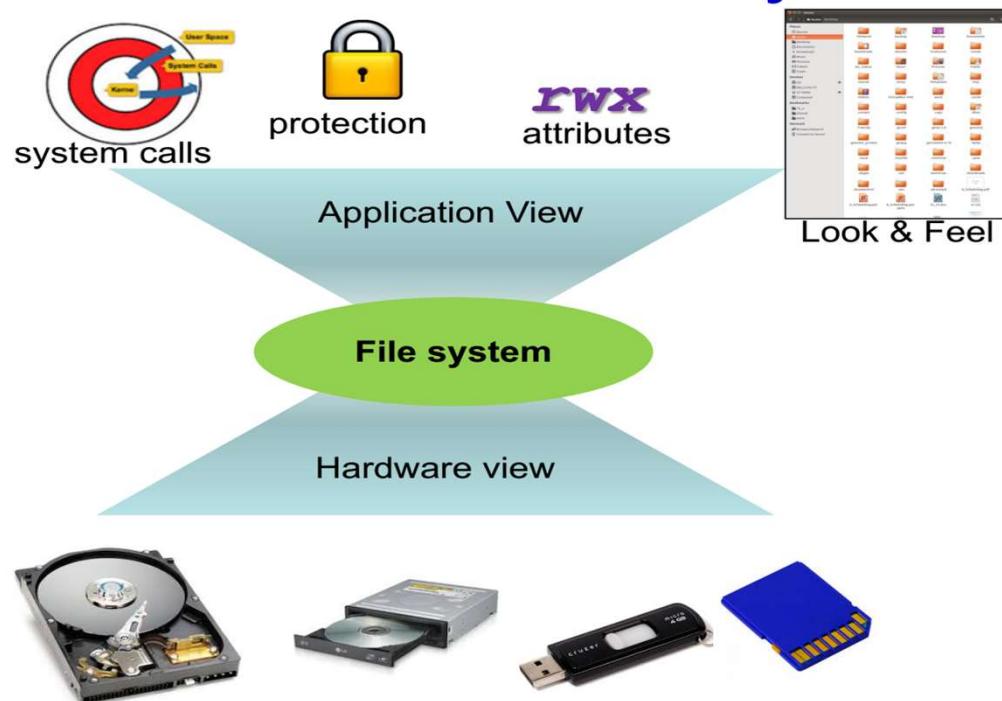
DR GAYATHRI ANANTHANARAYANAN

[gayathri@iitdh.ac.in](mailto:gayathri@iitdh.ac.in)

*Materials in these slides have been borrowed from textbooks and existing operating systems courses*

# March 1st, Files and Directories

## Two views of a file system



# Files and Directories

---

- From a **user's perspective**,
  - A byte array
  - Persistent across reboots and power failures
- From **OS perspective**,
  - Secondary (non-volatile) storage device
    - Hard disks, USB, CD, etc.
  - Map bytes as collection of **blocks** on storage device

A persistent-storage device, such as a classic harddisk drive or a more modern solid-state storage device, stores information permanently (or at least, for a long time).

How should the OS manage a persistent device?  
What are the APIs?  
What are the important aspects of the implementation?

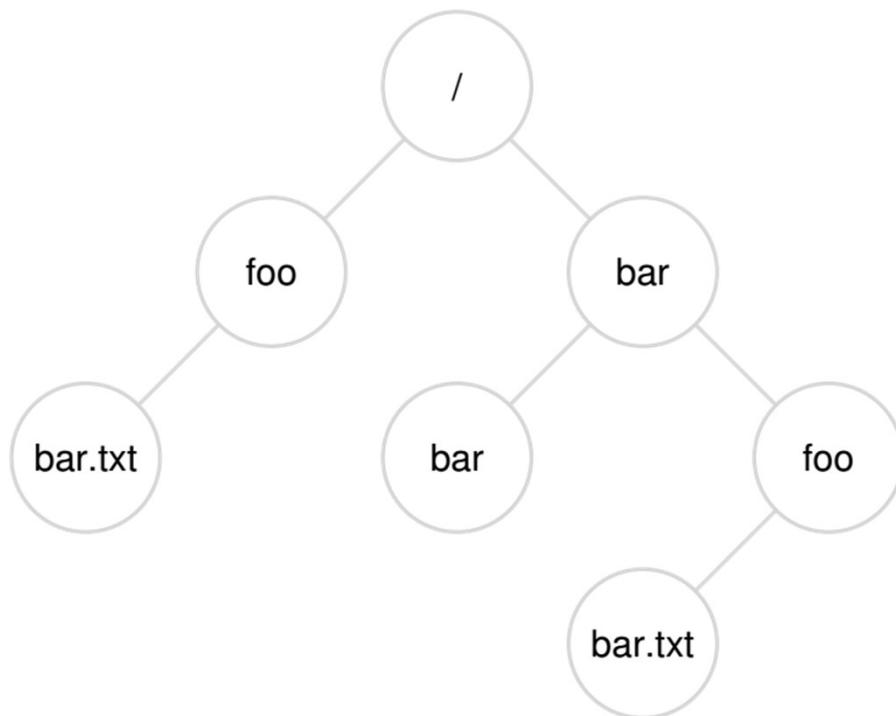
# Files and Directories

---

- File – linear array of bytes, stored persistently
  - Identified with file name (human readable) and a OS-level identifier low-level name [**“inode number”**]
  - Inode number unique within a file system
- Directory contains other subdirectories and files, along with their inode numbers
  - Stored like a file, whose contents are **filename-to-inode** mappings [eg. File – inode no. =10, name=foo, ("foo", "10")]

# Directory Tree

---



Directory hierarchy starts at a root directory (in UNIX-based systems).

root directory is simply referred to as /

uses some kind of separator to name subsequent sub-directories

# A File's Metadata

---

- **Name.** the only information kept in human readable form.
- **Identifier.** A number that uniquely identifies the file within the file system. Also called the inode number
- **Type.** File type (inode based file, pipe, etc.)
- **Location.** Pointer to location of file on device.
- **Size.**
- **Protection.** Access control information. Owner, group (r,w,x) permissions, etc. a
- **Monitoring.** Creation time, access time, etc.

```
sysad@sysad-Latitude-3490:~$ ls -il
total 1300
4614005 -rw-rw-r-- 1 sysad sysad 7617 Sep 29 12:41 AI_agri_startups
4591549 -rwxrwxr-x 1 sysad sysad 8608 Nov 27 14:29 a.out
4587533 drwxr-xr-x 3 sysad sysad 4096 Feb 22 11:21 Desktop
4587537 drwxr-xr-x 13 sysad sysad 4096 Feb 1 07:10 Documents
4587534 drwxr-xr-x 17 sysad sysad 69632 Feb 22 10:57 Downloads
4605692 -rw-rw-r-- 1 sysad sysad 94959 Oct 12 01:40 dst-tdp.pdf
4587524 -rw-r--r-- 1 sysad sysad 8980 May 7 2019 examples.desktop
4588953 -rw-rw-r-- 1 sysad sysad 25380 Nov 17 18:12 GA.tex
4603334 -rw-rw-r-- 1 sysad sysad 112231 Feb 25 21:03 index.html
4589460 -rw-rw-r-- 1 sysad sysad 849126 Nov 14 17:39 MS_and_PhD_Img_EditableFile2.odg
4587538 drwxr-xr-x 2 sysad sysad 4096 May 8 2019 Music
4588724 drwxr-xr-x 3 root root 4096 Dec 23 19:45 opt
4587539 drwxr-xr-x 3 sysad sysad 4096 Feb 22 11:10 Pictures
4587536 drwxr-xr-x 2 sysad sysad 4096 May 8 2019 Public
4588962 -rw-rw-r-- 1 sysad sysad 3715 Nov 17 18:12 research_profile.tex
4607807 -rw-rw-r-- 1 sysad sysad 317 Aug 22 2019 reviews_vlsi
4588836 -rw-rw-r-- 1 sysad sysad 66970 Oct 27 22:28 SCN.pdf
4587535 drwxr-xr-x 2 sysad sysad 4096 May 8 2019 Templates
4597814 -rw-rw-r-- 1 sysad sysad 20168 Jun 26 2020 Untitled 1.odt
4587540 drwxr-xr-x 2 sysad sysad 4096 May 8 2019 Videos
sysad@sysad-Latitude-3490:~$ █
```

- b** Block special file.
- c** Character special file.
- d** Directory.
- l** Symbolic link.
- s** Socket link.
- p** FIFO.
- Regular file.

# Files vs Memory

---

- Every memory location has an address that can be directly accessed
- In files, everything is **relative**
  - A location of a file depends on the directory it is stored in
  - A pointer must be used to store the current read or write position within the file
  - Eg. To read a byte in a specific file.
    - First search for the file in the directory path and resolve the identifier  
**expensive for each access !!!**
    - Use the read pointer to seek the byte position
  - **Solution :** Use open system call to open the file before any access  
(and close system call to close the file after all accesses are complete)

# Basics of creating, accessing, and deleting files

---

**Create a file** - accomplished with the open system call; by calling open() and passing it the O\_CREAT flag, a program can create a new file.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,  
              S_IRUSR|S_IWUSR);
```

Important aspect of open() is what it returns: **a file descriptor**.

file descriptor - an integer, private per process, and is used in UNIX systems to access files;

once a file is opened, you use the file descriptor to read or write the file; [pointer to an object of type file]

# Opening a File

---

- Steps involved
  - **Resolve Name** : search directories for file names and check permissions
  - Read file metadata into **open file table**
  - Return index in the open file table (this is the familiar **file descriptor**)

# Opening a File

---

- Two **open file tables** used
  - **system wide table**
    - Contains information about inode, size, access dates, permission, location, etc.
    - Reference count (tracks number of processes that have opened the file)
  - **per process table**
    - Part of PCBs proc structure
    - Pointer to entry in the system wide table

```
struct {  
    struct spinlock lock;  
    struct file file[NFILE];  
} ftable;
```

# How fd is managed by OS?

---

```
struct proc {  
    ...  
    struct file *ofile[NOFILE]; // Open files  
    ...  
};
```

```
struct file {  
    int ref;  
    char readable;  
    char writable;  
    struct inode *ip;  
    uint off;  
};
```

Simple array (with a maximum of NOFILE open files) tracks which files are opened on a per-process basis.

Each entry of the array - a pointer to a struct file

Used to track information about the file being read or written;

# Operations on Files

---

- Reading/writing files: `read()`/`write()` system calls
  - Arguments: file descriptor, buffer with data, size
- Reading and writing happens sequentially by default
  - Successive read/write calls fetch from current offset

---

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)              = 6
write(1, "hello\n", 6)                = 6
hello
read(3, "", 4096)                   = 0
close(3)                            = 0
```

Why does the first call to open() return 3, not 0 or perhaps 1 ?

cat uses the read() system call to repeatedly read some bytes from a file.

First argument to read() - fd;

Second argument - points to a buffer where the result of the read() will be placed;

Third argument - size of the buffer, which in this case is 4 KB.

call to read() returns successfully - returning the number of bytes it read [Why 6?]

# Random read

---

Should be able to read or write to a specific offset within a file;

Eg. if you build an index over a text document, and use it to look up a specific word – need to read from some random offsets within the document. [**Iseek() system call**]

```
off_t lseek(int fildes, off_t offset, int whence);
```

Offset- which positions the file offset to a particular location within the file.  
whence - determines exactly how the seek is performed.

If whence is SEEK\_SET, the offset is set to offset bytes.

If whence is SEEK\_CUR, the offset is set to its current location plus offset bytes.

If whence is SEEK\_END, the offset is set to the size of the file plus offset bytes.

OS tracks a “current” offset, which determines where the next read or write will begin reading from or writing to within the file.

---

let's track a process that opens a file (of size 300 bytes)

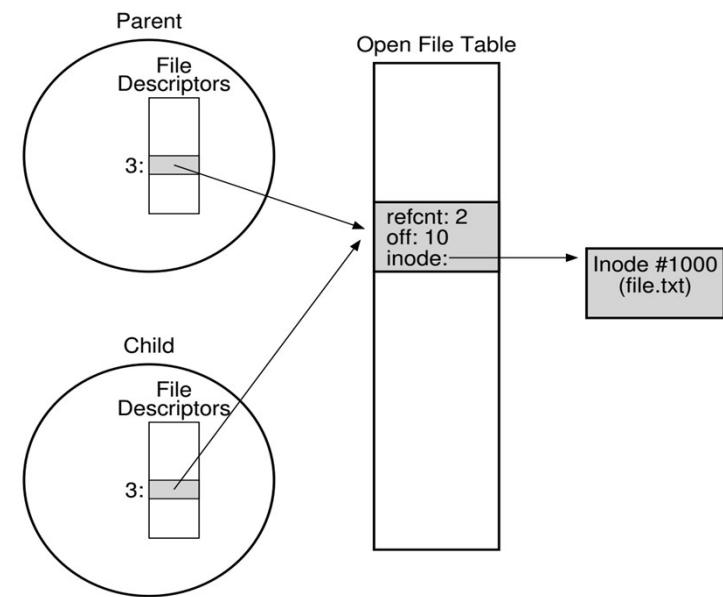
reads it by calling the read() system call repeatedly,  
each time reading 100 bytes.

System Calls	Return Code	Current Offset	System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
fd = open("file", O_RDONLY);	3	0	fd1 = open("file", O_RDONLY);	3	0	-
read(fd, buffer, 100);	100	100	fd2 = open("file", O_RDONLY);	4	0	0
read(fd, buffer, 100);	100	200	read(fd1, buffer1, 100);	100	100	0
read(fd, buffer, 100);	100	300	read(fd2, buffer2, 100);	100	100	100
read(fd, buffer, 100);	0	300	close(fd1);	0	-	100
close(fd);	0	-	close(fd2);	0	-	-
System Calls	Return Code	Current Offset	System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
fd = open("file", O_RDONLY);	3	0	fd1 = open("file", O_RDONLY);	3	0	-
lseek(fd, 200, SEEK_SET);	200	200	fd2 = open("file", O_RDONLY);	4	0	0
read(fd, buffer, 50);	50	250	read(fd1, buffer1, 100);	100	100	0
close(fd);	0	-	read(fd2, buffer2, 100);	100	100	100

# Shared File Table entries

```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
               (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
```



When a file table entry is shared, its reference count is incremented; only when both processes close the file (or exit) will the entry be removed.

---

Writes are buffered in memory temporarily, so `fsync()` system call flushes all writes to disk.

When a process calls `fsync()` for a particular file descriptor, the file system responds by forcing all dirty (i.e., not yet written) data to disk, for the file referred to by the specified file descriptor.

The `fsync()` routine returns once all of these writes are complete.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,  
              S_IRUSR|S_IWUSR);  
assert(fd > -1);  
int rc = write(fd, buffer, size);  
assert(rc == size);  
rc = fsync(fd);  
assert(rc == 0);
```

```
struct stat {  
    dev_t      st_dev;        // ID of device containing file  
    ino_t      st_ino;        // inode number  
    mode_t     st_mode;       // protection  
    nlink_t    st_nlink;      // number of hard links  
    uid_t      st_uid;        // user ID of owner  
    gid_t      st_gid;        // group ID of owner  
    dev_t      st_rdev;       // device ID (if special file)  
    off_t      st_size;       // total size, in bytes  
    blksize_t  st_blksize;    // blocksize for filesystem I/O  
    blkcnt_t   st_blocks;     // number of blocks allocated  
    time_t     st_atime;      // time of last access  
    time_t     st_mtime;      // time of last modification  
    time_t     st_ctime;      // time of last status change  
};  
-----
```

```
prompt> echo hello > file  
prompt> stat file  
  File: 'file'  
  Size: 6  Blocks: 8  IO Block: 4096  regular file  
Device: 811h/2065d Inode: 67158084  Links: 1  
Access: (0640/-rw-r-----)  Uid: (30686/remzi)  
          Gid: (30686/remzi)  
Access: 2011-05-03 15:50:20.157594748 -0500  
Modify: 2011-05-03 15:50:20.157594748 -0500  
Change: 2011-05-03 15:50:20.157594748 -0500
```

# Directories

---

- Maps file names to location on disk
- Directories also stored on disk
- Structure
  - Single-level directory
    - One directory for all files -- simple
    - Issues when multiple users are present
    - All files should have unique names
  - Two-level directory
    - One directory for each user
    - Solves the name collision between users
    - Still not flexible enough (difficult to share files between users)

# Operations on Directories

---

- Directories can also be accessed like files—Operations like create, open, read, close
- For example, the “ls” program opens and reads all directory entries –Directory entry contains file name, inode number, type of file (file/directory) etc.

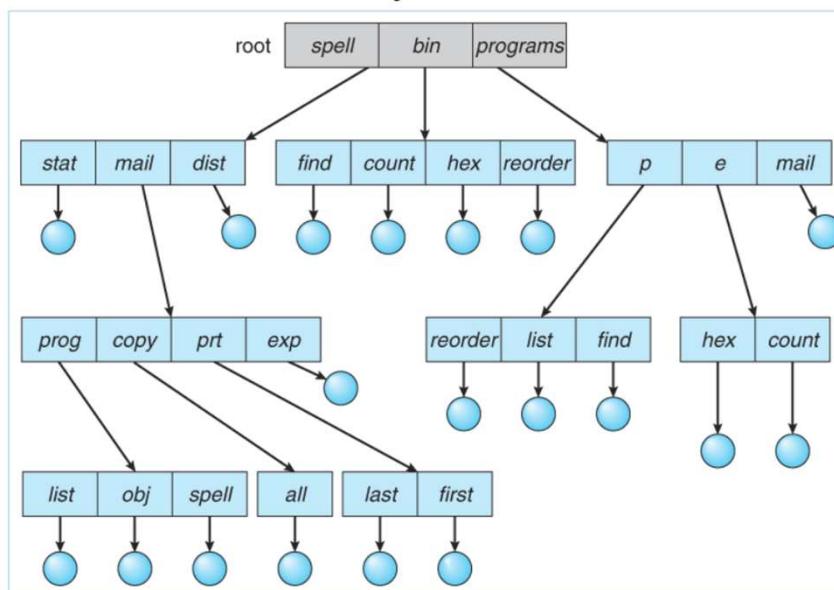
```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino,
               d->d_name);
    }
    closedir(dp);
    return 0;
}

struct dirent {
    char          d_name[256]; // filename
    ino_t         d_ino;       // inode number
    off_t         d_off;      // offset to the next dirent
    unsigned short d_reclen; // length of this record
    unsigned char  d_type;    // type of file
};
```

# Tree structured directories

- Directory stored as files on disk
  - Bit in file system used to identify directory
  - Special system calls to read/write/create directories
  - Referenced by slashes between directory names

Special directories  
/ → root  
. → current directory  
.. → parent directory



# Acyclic Graph Directories

---

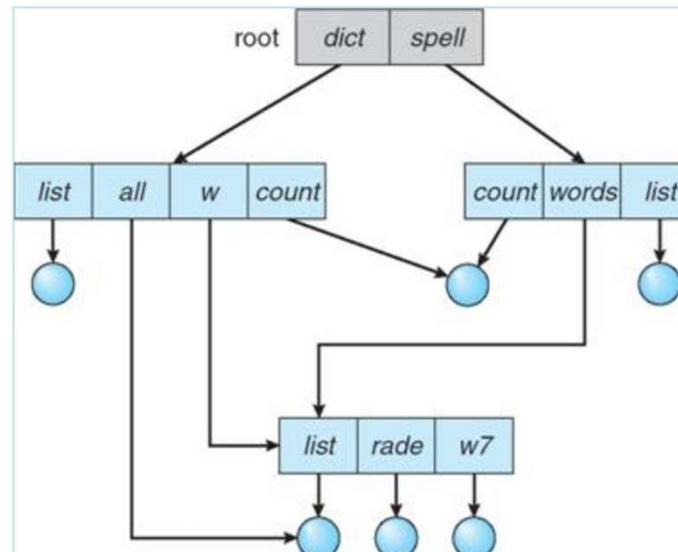
- Directories can share files
- Create links between files
  - Hard links  
it's a link to the actual file on disk  
(Multiple directory entries point to the same file)

`$ln a.txt ahard.txt`

- Soft links

it's a symbolic link to the path where the other file is stored

`$ln -s a.txt asoft.txt`



# Hard links

---

- Hard linking creates another file that points to the same inode number (and hence, same underlying data)
  - simply creates another name in the directory you are creating the link to, and refers it to the same inode number of the original file.
- If one file deleted, file data can be accessed through the otherlinks
- Inode maintains a link count, file data deleted only when no further links to it
- You can only unlink, OS decides when to delete

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

```
prompt> ls -i file file2
67158084 file
67158084 file2
```

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

# Hard vs Soft links

---

- Hard links cannot link directories. Cannot cross file system boundaries. Soft links can do both these
- Hard links always refer to the source, even if moved or removed. Soft links are destroyed if the source is moved or removed.
- Implementation difference...hard links store reference count in file metadata.

# Soft Links

---

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

- Soft link is a file that simply stores a pointer to another filename
- Formed by holding the pathname of the linked-to file as the data of the link file.
- If the main file is deleted, then the link points to an invalid entry: dangling reference

# Protection

---

- Types of access
  - Read, write, execute, ...
- Access Control
  - Which user can use which file!
- Classification of users
  - User, group, others

# Mounting a File system

---

How to assemble a full directory tree from many underlying file systems ?

-- accomplished via first making file systems, and then mounting them to make their contents accessible.

**mkfs** -- input, a device (such as a disk partition, e.g., /dev/sda1) and a file system type (e.g., ext3), and it simply writes an empty file system, starting with a root directory, onto that disk partition.

**mount ()** -take an existing directory as a target mount point and essentially paste a new file system onto the directory tree at that point. Mounting a filesystem connects the files to a specific point in the directory tree

# Mounting a File system

---

```
prompt> mount -t ext3 /dev/sda1 /home/users  
prompt> ls /home/users/  
a b
```

Several devices and file systems are mounted on a typical machine, accessed with `mount` command

```
/dev/sda1 on / type ext3 (rw)  
proc on /proc type proc (rw)  
sysfs on /sys type sysfs (rw)  
/dev/sda5 on /tmp type ext3 (rw)  
/dev/sda7 on /var/vice/cache type ext3 (rw)  
tmpfs on /dev/shm type tmpfs (rw)  
AFS on /afs type afs (rw)
```

# Mounting a File system

---

- Just like a file needs to be opened, a file system needs to be mounted
- OS needs to know
  - The location of the device (partition) to be mounted
  - The location in the current file structure where the file system is to be attached
- \$ mount /dev/sda3 /media/xyz -t ext3
- OS does,
  - Verify that the device has a valid file system
  - Add new file system to the mount point (/media/xyz)

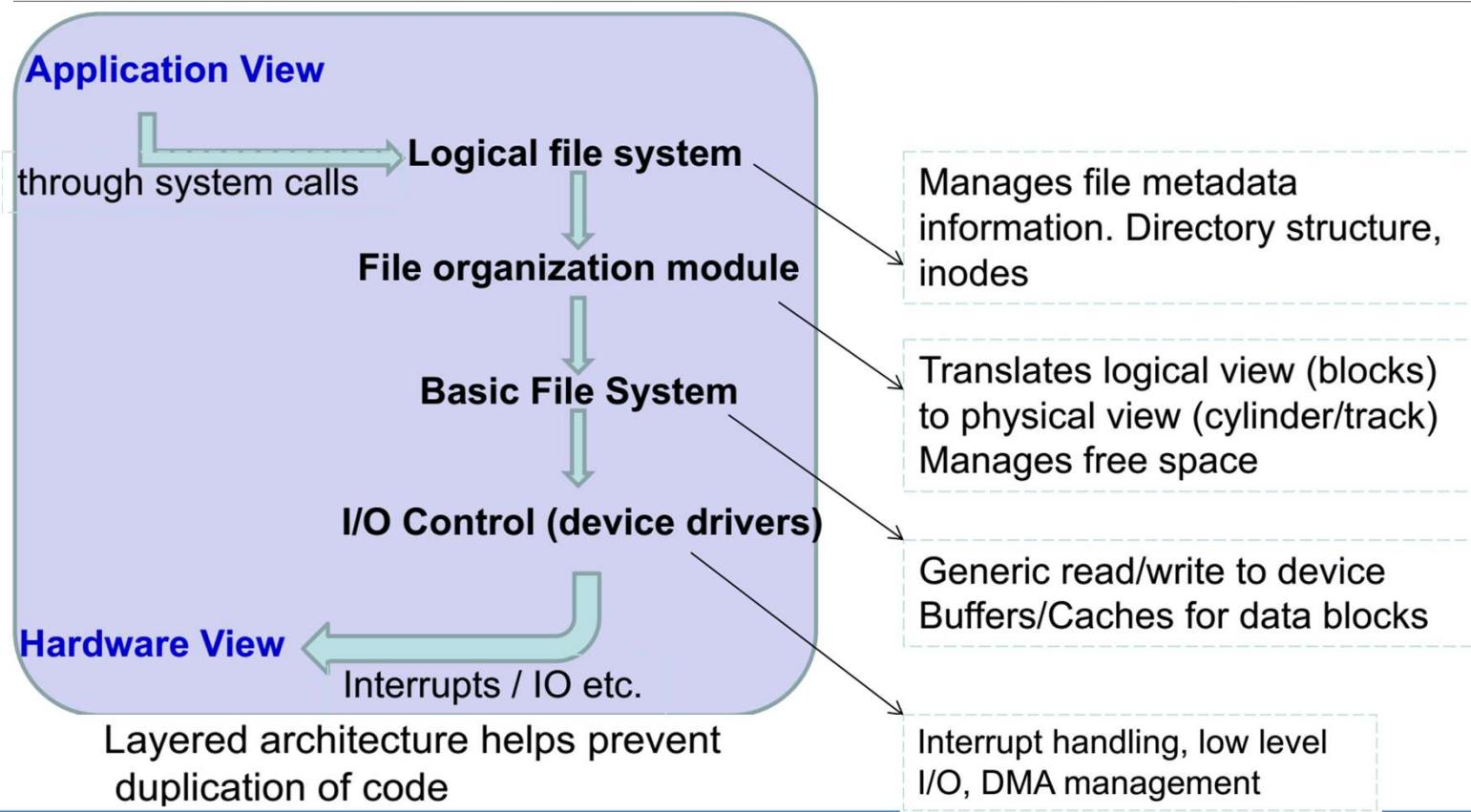
# Memory Mapping a file

---

Alternate way of accessing a file, instead of using file descriptors and read/writesyscalls

- `mmap()` allocates a page in the virtual address space of a process
  - “Anonymous” page: used to store program data
  - “File-backed page”: contains data of file (filename provided as arg to `mmap`)
- When file is mmaped, file data copied into one or more pages in memory, can be accessed like any other memory location in program

# Implementing a File system



# File System

---

- An organization of files and directories on disk and OS has one or more file systems
  - Two main aspects of file systems
    - Data structures to organize data and metadata on disk
    - Implementation of system calls like open, read, write using the data structures [Access Methods]
  - Disks expose a set of blocks (usually 512 bytes)
  - File system organizes files onto blocks
    - System calls translated into reads and writes on blocks

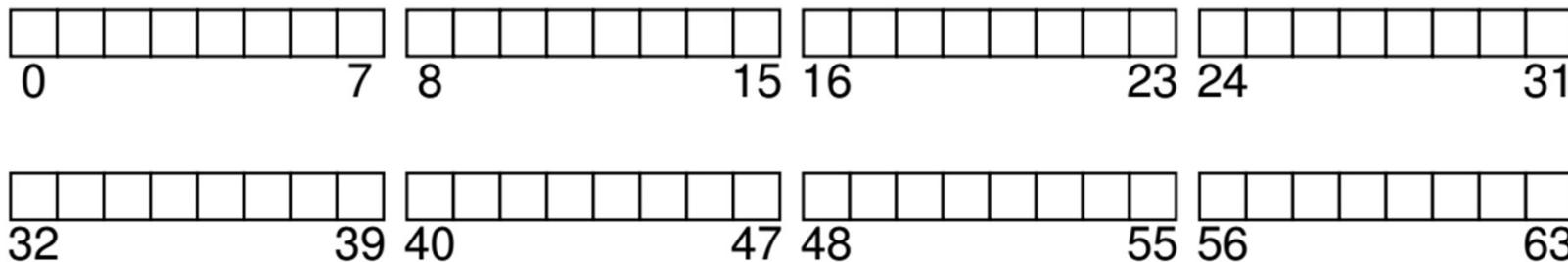
# Simple file system

---

Overall on-disk organization of the data structures

Divide disks into blocks and use a single block size [eg . 4KB]

A series of blocks, each of size 4 KB. The blocks are addressed from 0 to N – 1, in a partition of size N 4-KB blocks. Assume we have a really small disk, with just 64 blocks:

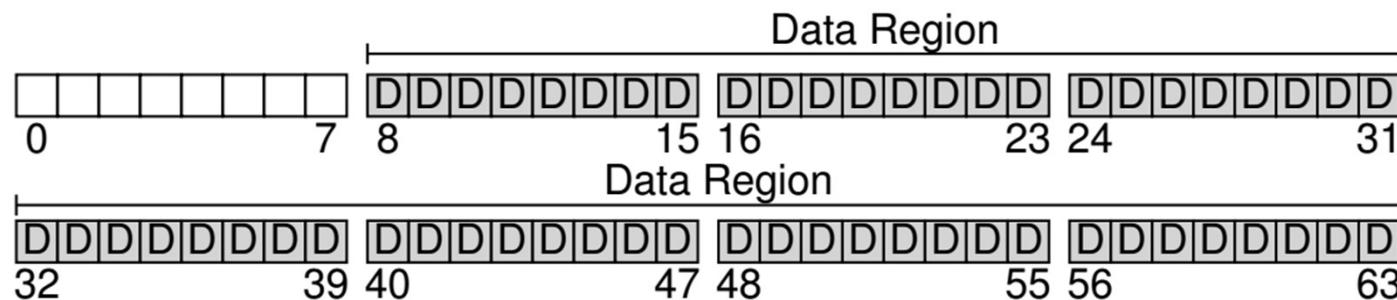


Think about what we need to store in these blocks to build a file system.

# Data blocks

---

The first thing that comes to mind is user data -- region of the disk we use for user data the **data region or data blocks**

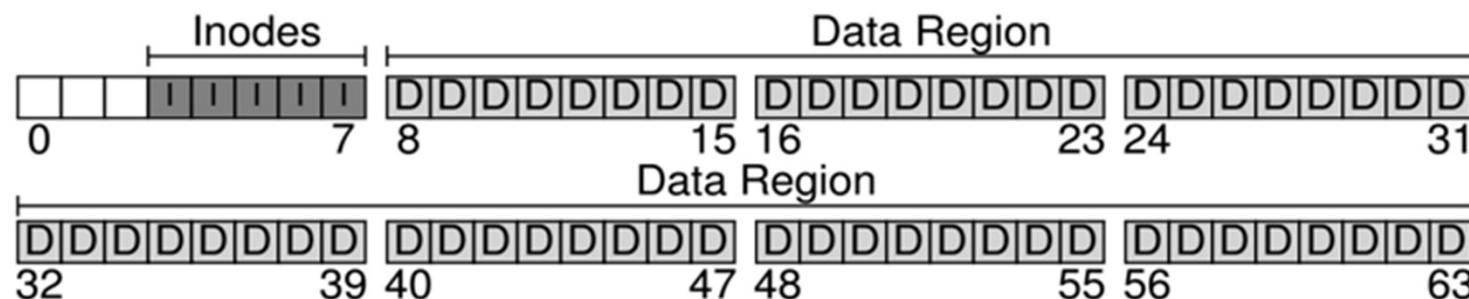


# Inode [index node]

---

The file system has to track information about each file

- key piece of metadata, and tracks things like which data blocks (in the data region) comprise a file, the size of the file, its owner and access rights, access and modify times
- To store this information, file systems usually have a structure called an **inode**



# Inode

---

Assume that we use 5 of our 64 blocks for inodes and 256 bytes per inode, a 4-KB block can hold 16 inodes

Total file system contains how many inodes - ??

This number represents the maximum number of files we can have in our file

Note -- the same file system, built on a larger disk, could simply allocate a larger inode table and thus accommodate more files.

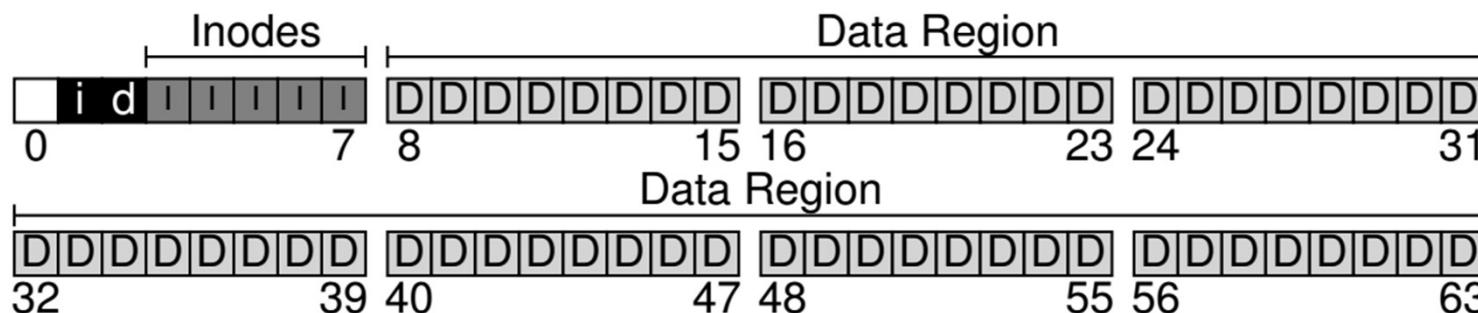
File system thus far has data blocks (D), and inodes (I), but a few things are still missing.

# Allocation structures

One primary component that is still needed -- some way to track whether inodes or data blocks are free or allocated.

Many allocation-tracking methods are possible

- **free list** -points to the first free block, which then points to the next free block, and so forth
- **Bitmap**-each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1). one for the data region (the data bitmap), and one for the inode table (the inode bitmap)

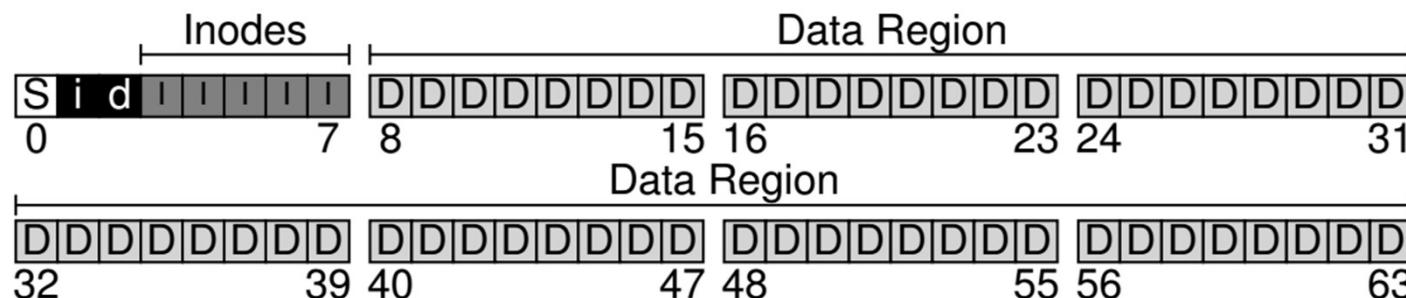


# Superblock

---

Superblock -- contains information about this particular file system

For eg., how many inodes and data blocks are in the file system (80 and 56, respectively in this instance); where the inode table begins (block 3), and so forth. It will likely also include some kind of a number to identify the file system type

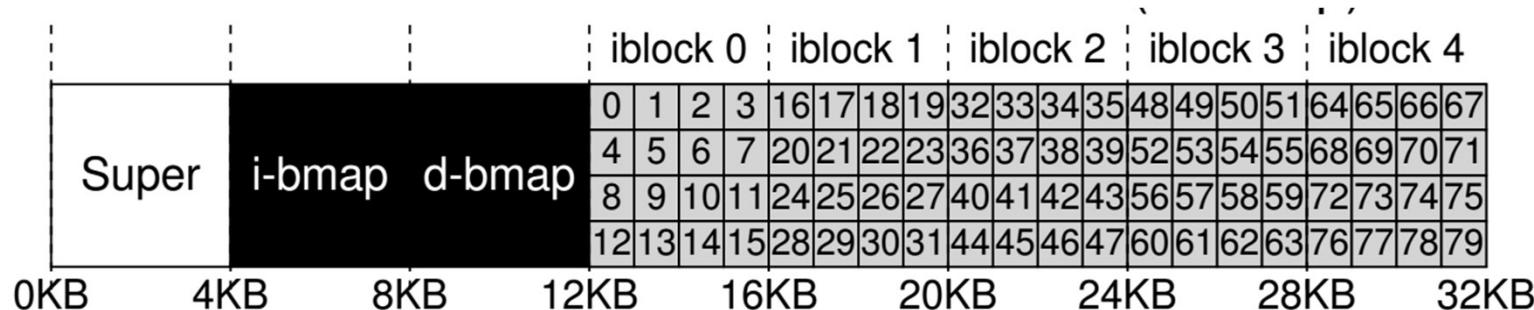


OS will read the superblock first, to initialize various parameters, and then attach the volume to the file-system tree. When files within the volume are accessed, the system will thus know exactly where to look for the needed on-disk structures.

# Inode Table

---

Given an inode-number, you should directly be able to calculate where on the disk the corresponding inode is located.



20KB in size (five 4KB blocks) and thus consisting of 80 inodes (assuming each inode is 256 bytes); assume that the inode region starts at 12KB (i.e., the superblock starts at 0KB, the inode bitmap is at address 4KB, the data bitmap at 8KB)

Thus the inode table comes right after

---

To read inode number 32, the file system would

- first calculate the offset into the inode region ( $32 \cdot \text{sizeof(inode)}$  or 8192)
- add it to the start address of the inode table on disk (inodeStartAddr=12KB)
- arrive upon the correct byte address of the desired block of inodes:20KB

Disks are not byte addressable –consist of a large number of addressable sectors, usually 512 bytes.

To fetch the block of inodes that contains inode 32, the file system would issue a read to sector  $20 \times 1024 / 512$ , or 40, to fetch the desired inode block.

---

Generally the sector address of the inode block can be calculated as follows:

```
blk      = (inumber * sizeof(inode_t)) / blockSize;  
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

<b>Size</b>	<b>Name</b>	<b>What is this inode field for?</b>
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

# Inode structure

---

- File data not stored contiguously on disk, need to track multiple block numbers of a file
- How does inode track disk block numbers?
  - Direct pointers: - each pointer refers to one disk block that belongs to the file - numbers of first few blocks are stored in inode itself (suffices for small files)
  - Indirect block: for larger files, inode stores number of indirect block, which has block numbers of file data
  - Similarly, double and triple indirect blocks (multi-level index)

# Allocation Methods

---

- How does the OS allocate blocks in the disk?

- Contiguous allocation
- Linked allocation
- Indexed allocation

Key Metrics :

Fragmentation (internal & external)? ; Grow file over time after initial creation?

Fast to find data for sequential and random access?; Easy to implement?

Storage overhead?

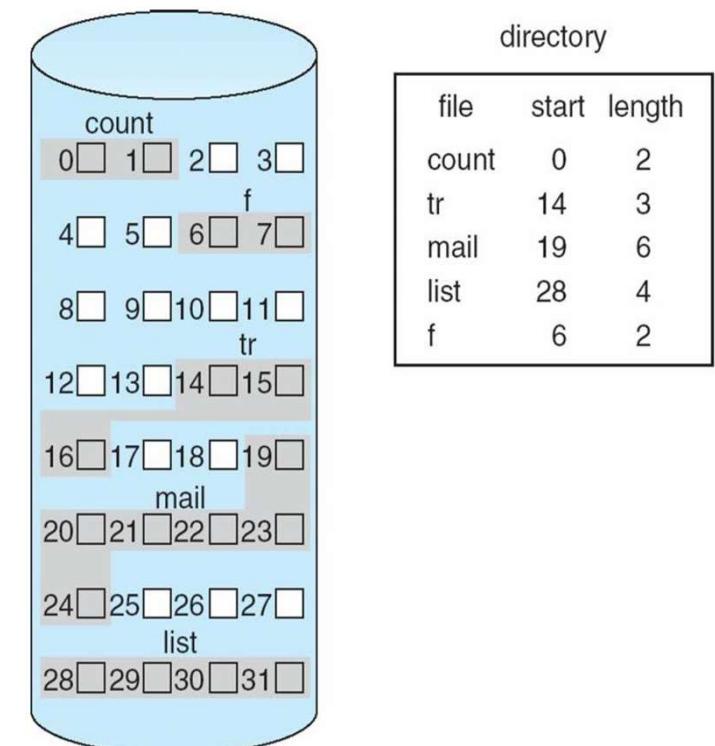
# Contiguous Allocation

Allocate files like continuous memory allocation(base & limit)

User specifies length, file system allocates space all at once

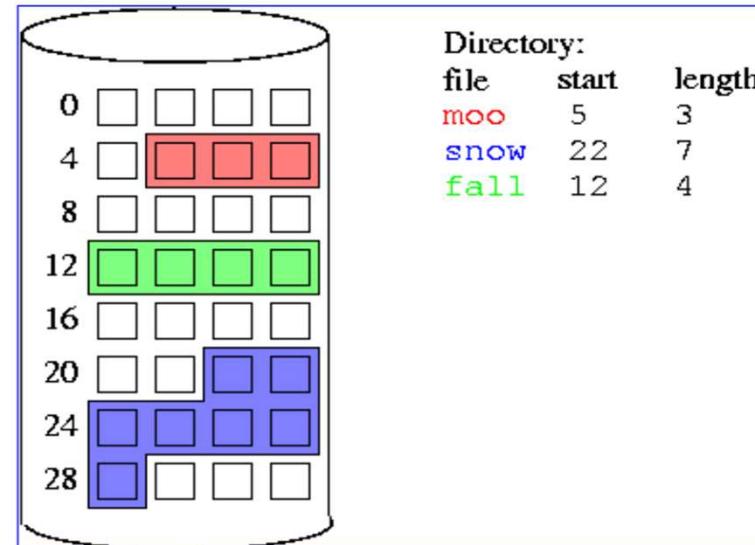
Can find disk space by examining bitmap

Metadata: contains starting location and size of file



# Contiguous Allocation

- Each file is allocated contiguous blocks on the disk
- Directory entry keeps the start and length
- **Allocation**
  - First fit / best fit ?
- **Advantages**
  - Easy / simple
- **Disadvantages**
  - External fragmentation  
(may need regular defragmentation)
  - Users need to specify the maximum file size at creation  
(may lead to internal fragmentation – a file may request a much large space and not use it)

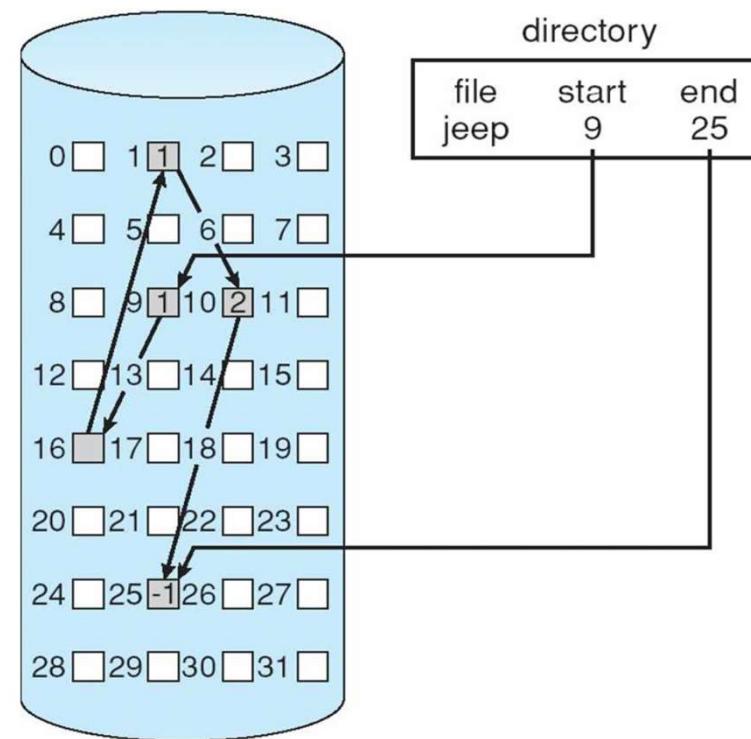


# Linked Allocation

All blocks (fixed-size) of a file on linked list

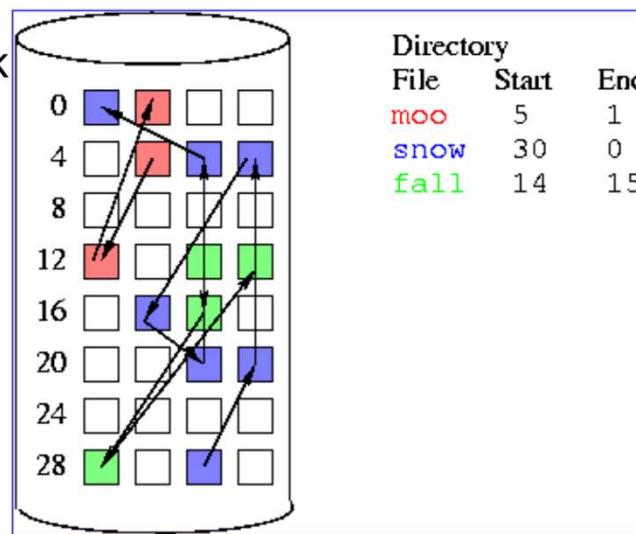
Each block has a pointer to next

Metadata: pointer to the first block



# Linked Allocation

- Directory stores link of start and end block (optionally)
- Pointer in block store link to next block
- Advantages
  - Solves external fragmentation problems
- Disadvantages
  - Not suited for direct access of files (all pointers need to be accessed)
  - Pointer needs to be stored .. **overheads!!**
    - Overheads reduced by using clusters (ie. cluster of sequential blocks associated with one pointer)
  - Reliability.
    - If a pointer is damaged (or lost), rest of file is lost.
    - A bug in the OS may result in a wrong pointer being picked up.



# File Allocation Table

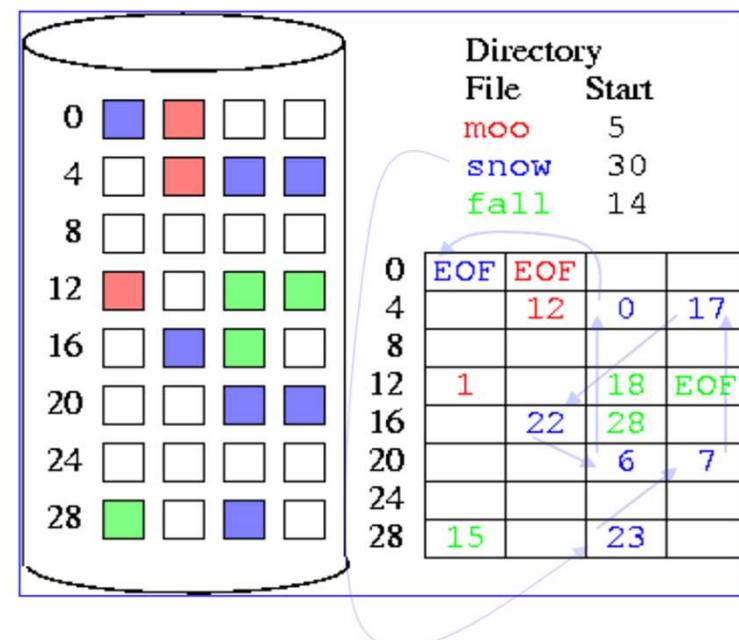
---

- Alternate way to track file blocks
- keep link structure in memory
  - in fixed-size "FAT" (file allocation table)
  - pointer chasing now happens in RAM
- FAT stores next block pointer for each block
  - FAT has one entry per disk block
  - Entry has number of next file block, or null (if last block)
  - Pointer to first block stored in inode

# FAT file

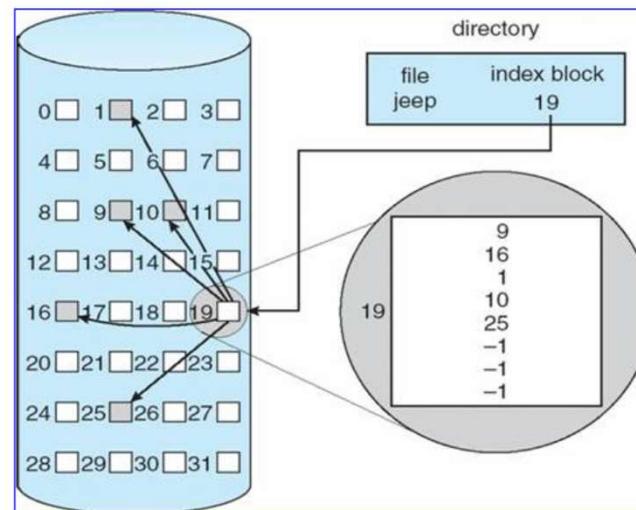
## (a variation linked allocation scheme)

- Invented by Marc McDonald and Bill Gates
- FAT is a table that
  - contains one entry for each block
  - and is indexed by block number.
- Files represented by linking pointers in the FAT
- FAT table generally cached
- **Advantages,**
  - Solves direct access problems of linked allocation
  - Easy to grow files
  - Greater reliability
    - A bad block implies only one block is corrupted
- **Disadvantages,**
  - Volume size determined by FAT size



# Indexed Allocation

- Advantages,
  - Supports direct access
  - No external fragmentation
  - Easy to grow files
- How large should the index block be?
  - Files typically, one or two blocks long
    - The index block will therefore have only one or two entries
    - A large index block → huge wastage
  - A small index block will limit the size of file
    - Need an additional mechanism to deal with large files
- Disadvantage,
  - Sequential access may be slow
    - May use clusters



Use disk blocks as index blocks that don't hold file data, but hold pointers to the disk blocks that hold file data.

# Multi level indexing

---

Indirect pointers - Instead of pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data.

Double indirect pointer - refers to a block that contains pointers to indirect blocks, each of which contain pointers to data blocks.

Eg. with 12 direct pointers, as well as both a single and a double indirect block.

Assuming a block size of 4 KB, and 4-byte pointers, this structure can accommodate a file of just over 4 GB in size [ ]

$$(12 + 1024 + 1024^2) \times 4 KB$$

- Used in UNIX (numbers below are for traditional UNIX, BSD UNIX 4.1)
- Each inode (file descriptor) contains 13 *block pointers*
  - First 10 pointers point to data blocks (each 512 bytes long) of a file
  - If the file is bigger than 10 blocks (5,120 bytes), the 11th pointer points to a *single indirect block*, which contains 128 pointers to 128 more data blocks (can support files up to 70,656 bytes)
    - If the file is bigger than that, the 12th pointer points to a *double indirect block*, which contains 128 pointers to 128 more single indirect blocks (can support files up to 8,459,264 bytes)
      - » If the file is bigger than that, the 13th pointer points to a *triple indirect block*, which contains 128 pointers to 128 more double indirect blocks
  - Max file size is 1,082,201,087 bytes

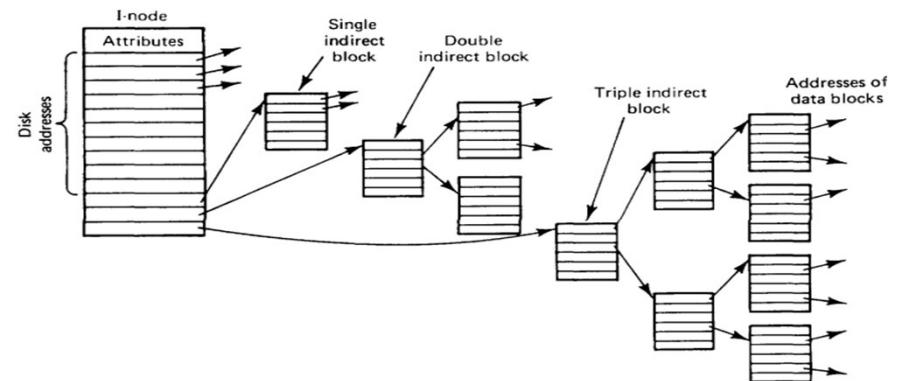


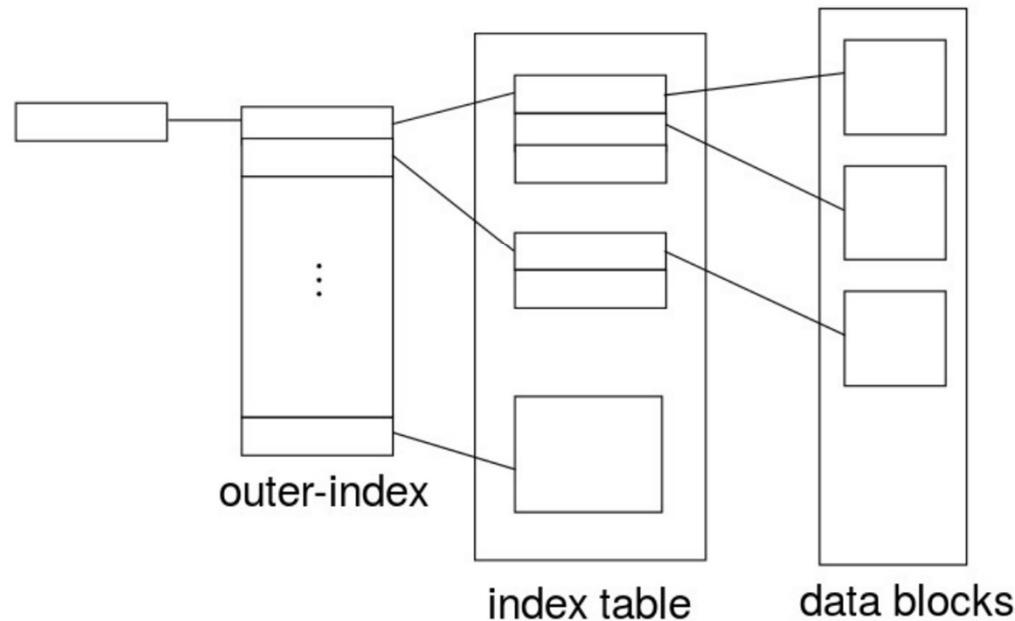
Diagram from *Modern Operating Systems*, Andrew Tanenbaum, Prentice Hall, 1992.

- BSD UNIX 4.2, 4.3:
  - Maximum block size is 4096 bytes
  - Inode contains 14 block pointers
    - 12 to data
    - 13 to single indirect block containing 1024 pointers, 14 to double indirect block...
  - Max file size is  $2^{32}$  bytes

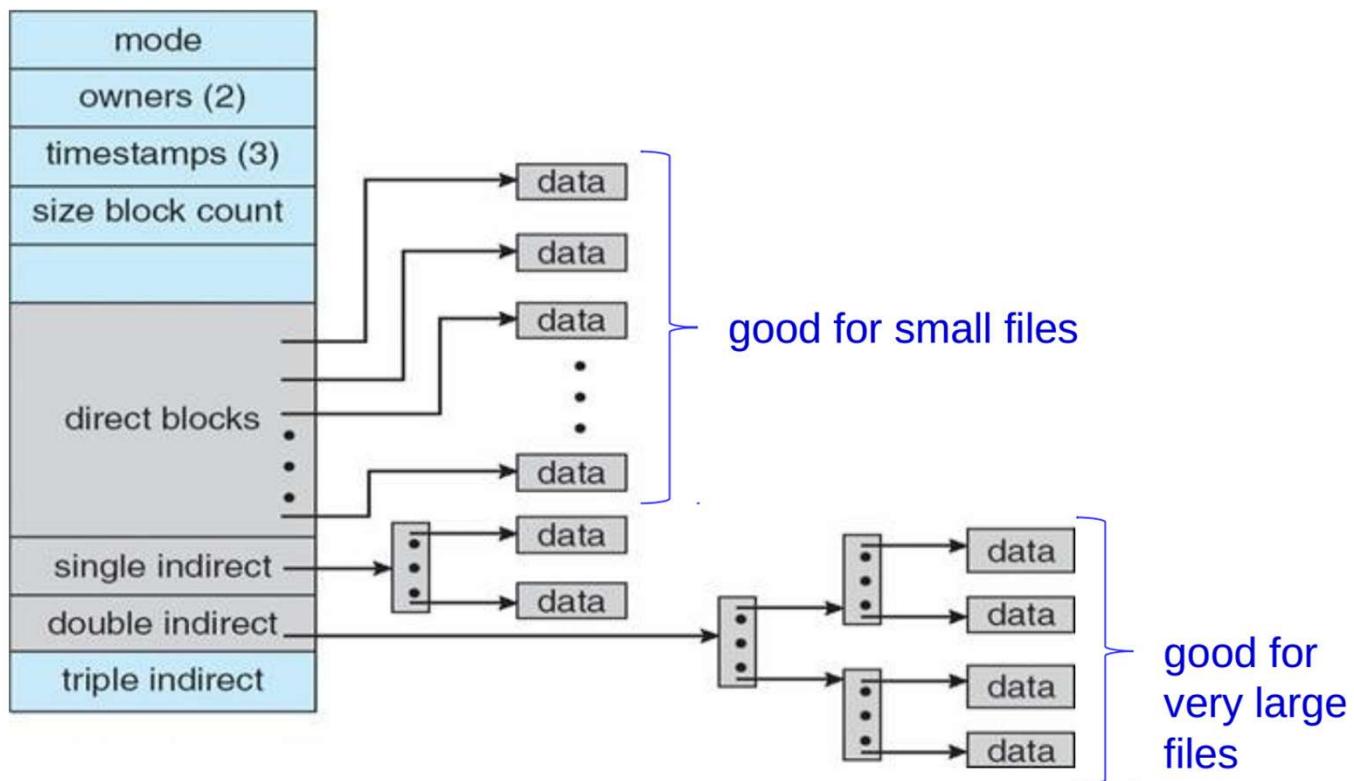
# Multilevel Indexing

---

- Block index has multiple levels



# Multi Level Indexing Linux (ext2/ext3)



## Trivia

---

Consider a disk of size **2 TB**, with **512-byte** sized blocks. Assume that the filesystem uses a multilevel inode data structure to track data blocks of a file. The inode has **64 bytes** of space available to store pointers to data blocks, including a single indirect block, a double indirect block, and several direct blocks. What is the maximum file size that can be stored in such a file system?

---

Number of data blocks =  $2^{41}/2^9=2^{32}$ ,

so 32 bits or 4 bytes are required to store the number of a data block.

Number of data block pointers in the inode =  $64/4 = 16$ , of which 14 are direct blocks.

The single indirect block stores pointers to  $512/4 = 128$  data blocks.

The double indirect blockpoints to 128 single indirect blocks, which in turn point to 128 data blocks each.

So, the total number of data blocks in a file can be  $14 + 128 + 128*128 = 16526$ , and the maximum file size is  $16526*512$  bytes

Max file size = no\_addressable\_blocks \* block size

number\_of\_addressable\_blocks = direct\_blocks + blocks\_addressable\_by\_single\_indirect + blocks\_addressable\_by\_double\_indirect + blocks\_addressable\_by\_triple\_indirect

# Directory Structure

---

Directory basically just contains a list of (entry name, inode number) pairs.

For each file or directory in a given directory, there is a string and a number in the data block(s) of the directory.

For each string, there may also be a length (assuming variable-sized names)

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a.pretty_longname

Each entry has an inode number, record length (the total bytes for the name plus any left over space), string length (the actual length of the name), and finally the name of the entry.

# Directory Structure

---

- Directory is a special type of file and has inode and data blocks (which store the file records)
  - In the inode table, an inode entry with the type field of the inode marked as “directory” instead of “regular file”)
  - The directory has data blocks pointed to by the inode (and perhaps, indirect blocks);
  - These data blocks live in the data block region of the file system. Thus, on-disk structure remains unchanged.