File Systems

Abhijit A M
abhijit.comp@coep.ac.in

What we are going to learn

- The operating system interface (system calls, commands/utilities) for accessing files in a file-sysetm
- Design aspects of OS to implement the file system
  - On disk data structure
  - In memory kernel data structures

What is a file?

- A (dumb!) sequence of bytes (typically on a permanent storage:seconary, tertiary) , with
  - A name
  - Permissions
  - Owner
  - Timestamps,
  - Etc.
- Types: Text files, binary files (one classification)
  - Text: All bytes are human readable
  - Binary: Non-text
- Types: ODT, MP4, TXT, DOCX, etc. (another classification)
  - Most typically describing the organization of the data inside the file
  - Each type serving the needs of a particular application (not kernel)

File types and kernel

- For example, MP4 file
  - *vlc* will do a open(...) on the file, and call read(...), interpret the contents of the file as movie and show movie
  - Kernel will simply provide open(...) read(...), write(...) to access file data
  - Meaning of the file contents is known to VLC and not to kernel!

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be *interpreted (by an application)* to be
  - Just a sequence of bytes
    - E.g. a text file
  - Sequence of records/structures
    - E.g. a file of student records , by database application, etc
  - A complexly organized, collection of records and bytes
    - E.g. a “ODT” or “DOCX” file
- What’s the role of OS in above mentioned file type, and organization?
  - Mostly NO role on Unixes, Linuxes!
  - They are handled by applications !
  - Types handled by OS: normal file, directory, block device file, character device file, FIFO file (named pipe), etc.
  - Also types handled by OS: executable file, non-executable file

File attributes

- Run

  `$ ls -l`

  on Linux
  To see file listing with different attributes

- Different OSes and file-systems provide different sets of file attributes
  - Some attributes are common to most, while some are different
  - E.g. name, size, owner can be found on most systems
  - “Executable” permission may not be found on all systems

Access methods

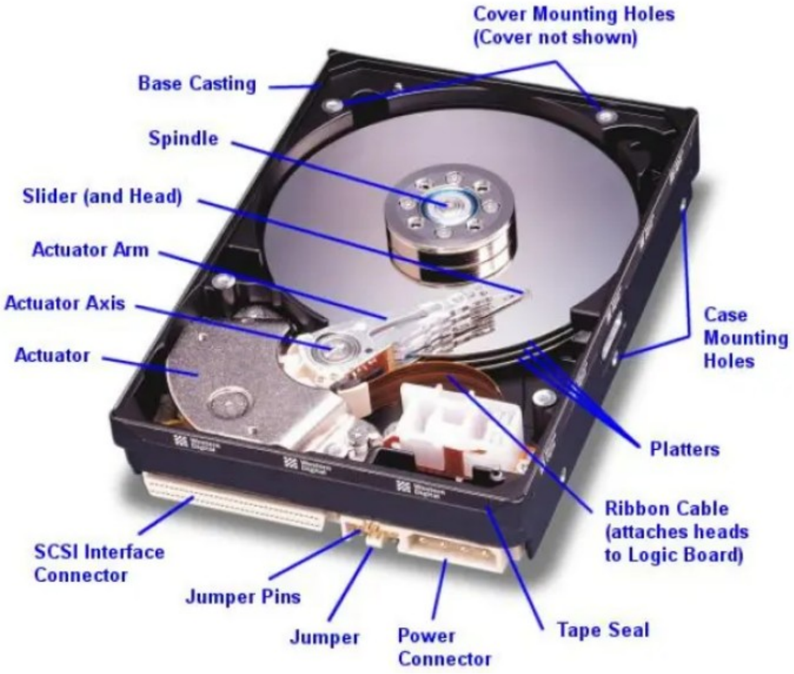
- OS system calls may provide two types of access to files
  - Sequential Access
    - read next
    - write next
    - reset
    - no read after last write (rewrite)
  - Linux provides sequential access using open(), read(), write(), ...

- Direct Access
  - read n
  - write n
  - position to n
    read next
    write next
  - rewrite n

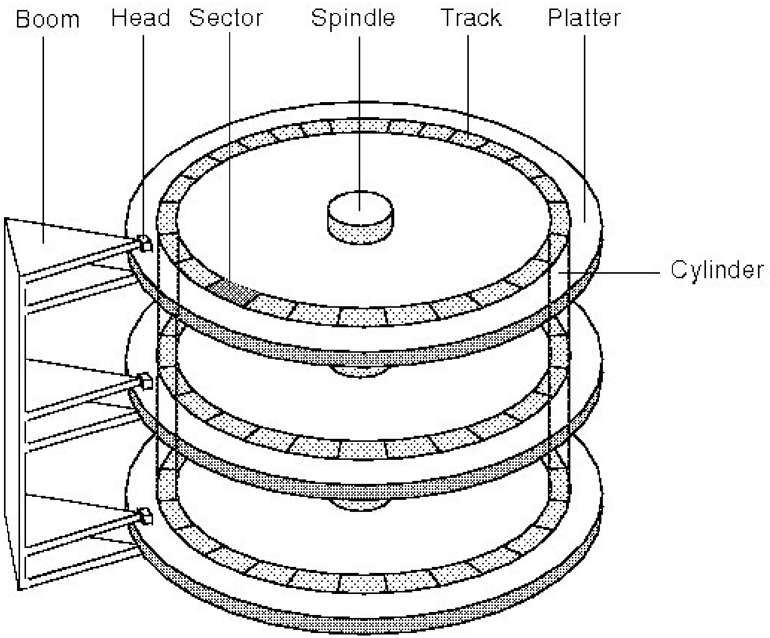
  n = relative block number

- pread(), pwrite() on Linux
  - ssize_t pread(int fd, void *buf, size_t count, off_t offset);
  - ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);

Disk

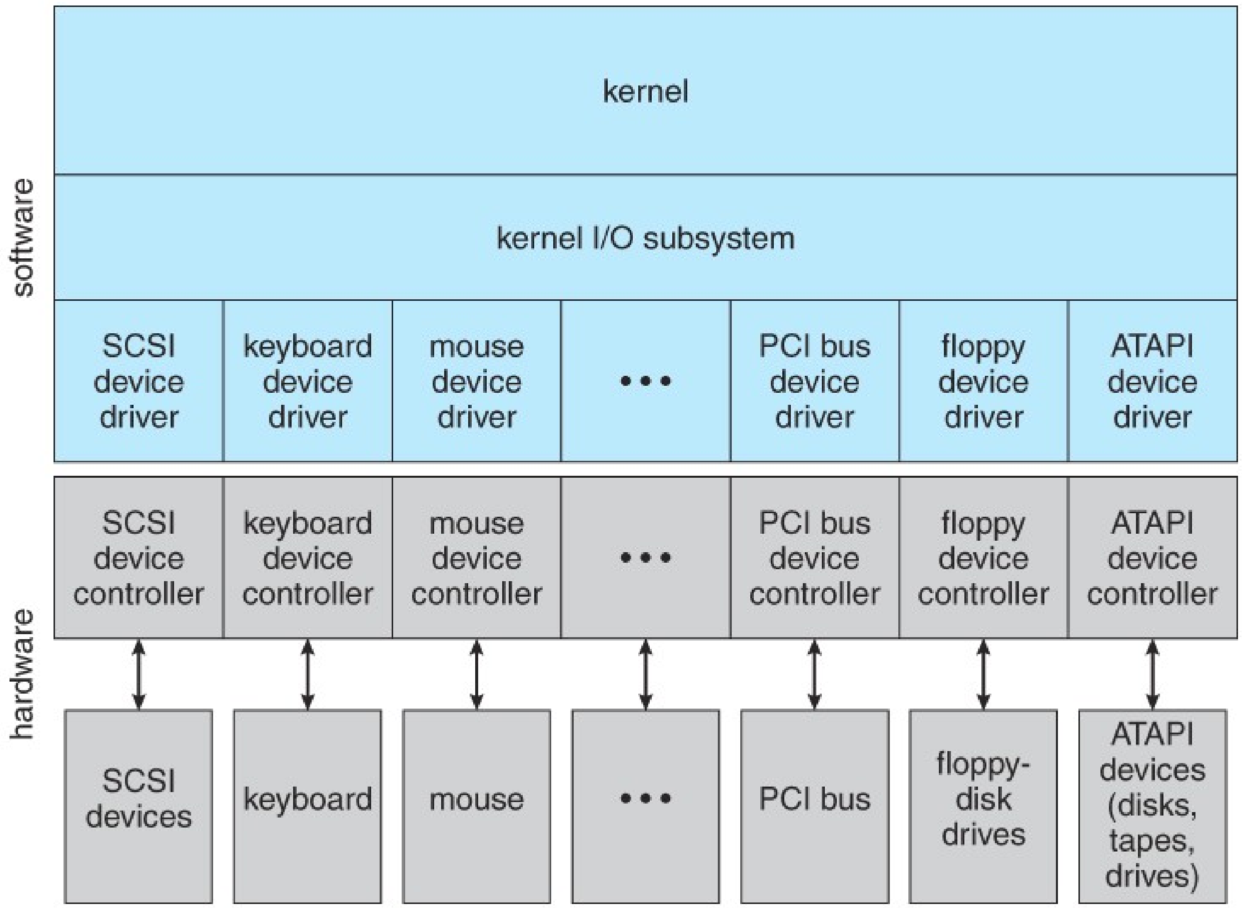


Cover Mounting Holes
(Cover not shown)
Base Casting
Spindle
Slider (and Head)
Actuator Arm
Actuator Axis
Actuator
Case
Mounting
Holes
Platters
Ribbon Cable
(attaches heads
to Logic Board)
SCSI Interface
Connector
Jumper Pins
Jumper
Power
Connector
Tape Seal

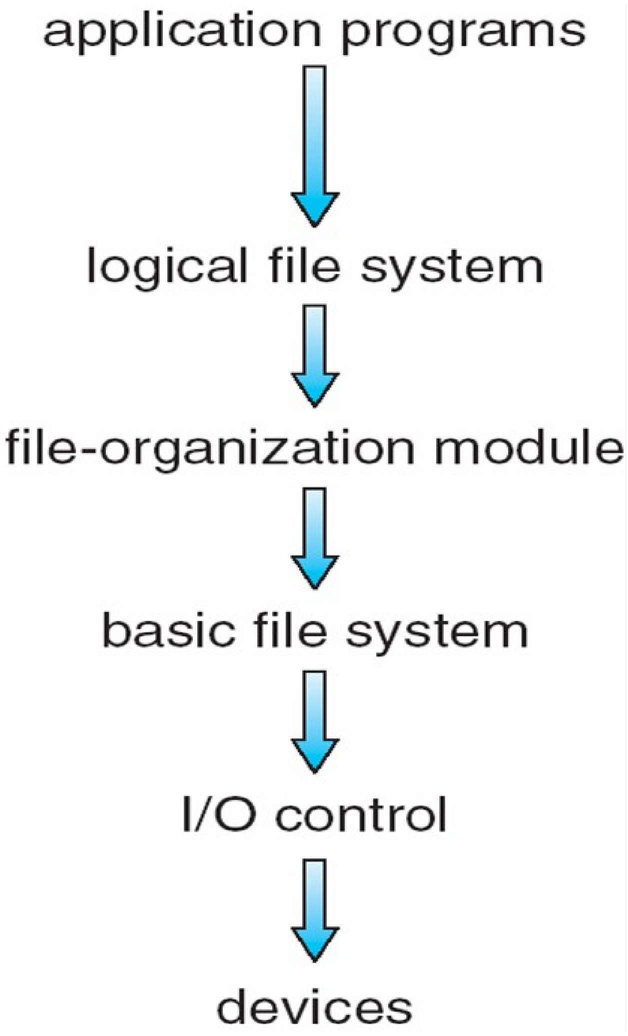


Boom
Head
Sector
Spindle
Track
Platter
Cylinder

Device Driver



File system implementation: layering



Application programs

```
int main() {
	char buf[128]; int count;
	fd = open(...):
	read(fd, buf, count);
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {
	file *fp = currproc->fdarray[fd];
	file_read(fp, ...);
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {
	offset = fp->current-offset;
	translate offset into blockno;
	basic_read(blockno, buf, count);

}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {
	os_buffer *bp;
	sectorno = calculation on blockno;
	disk_driver_read(sectono, bp );
	move-process-to-wait-queue;
	copy-data-to-user-buffer(bp, buf);
}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {
	issue instructions to disk controller
(often assembly code)
	to read sectorno  into specific
location;
}
```

XV6 does it slightly differently, but following the layering principle!

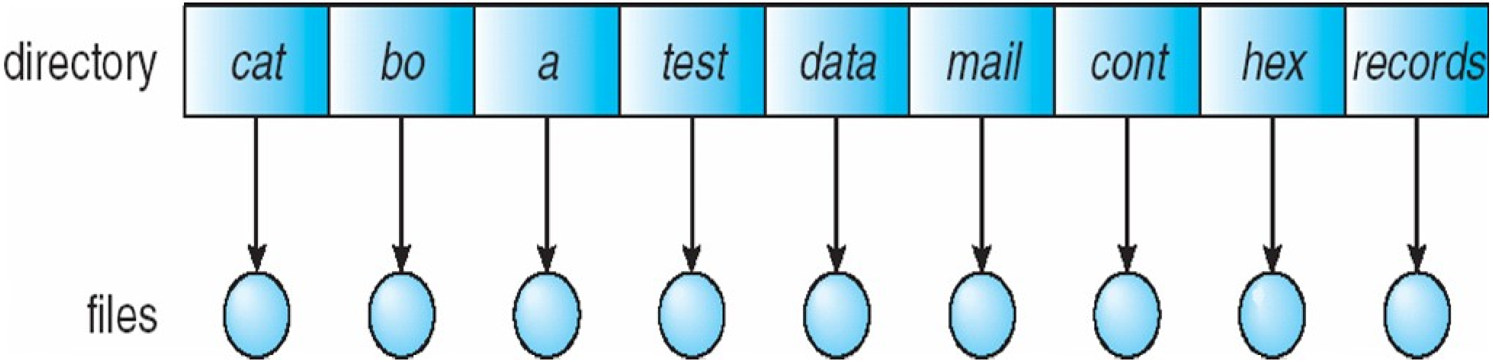
OS’s job now

- To implement the logical view of file system as seen by end user
- Using the logical block-based view offerred by the device driver

Formatting

- Physical hard disk divided into partitions
  - Partitions also known as minidisks, slices
- A raw disk partition is accessible using device driver – but no block contains any data !
  - Like an un-initialized array, or sectors/blocks
- Formatting
  - Creating an initialized data structure on the partition, so that it can start storing the acyclic graph tree structure on it
  - Different formats depending on different implementations of the directory tree structure: ext4, NTFS, vfat, VxFS, ReiserFS, WafleFS, etc.
- Formatting happens on “a physical partition” or “a logical volume made available by volume manager”

Different types of "layouts" Single level directory

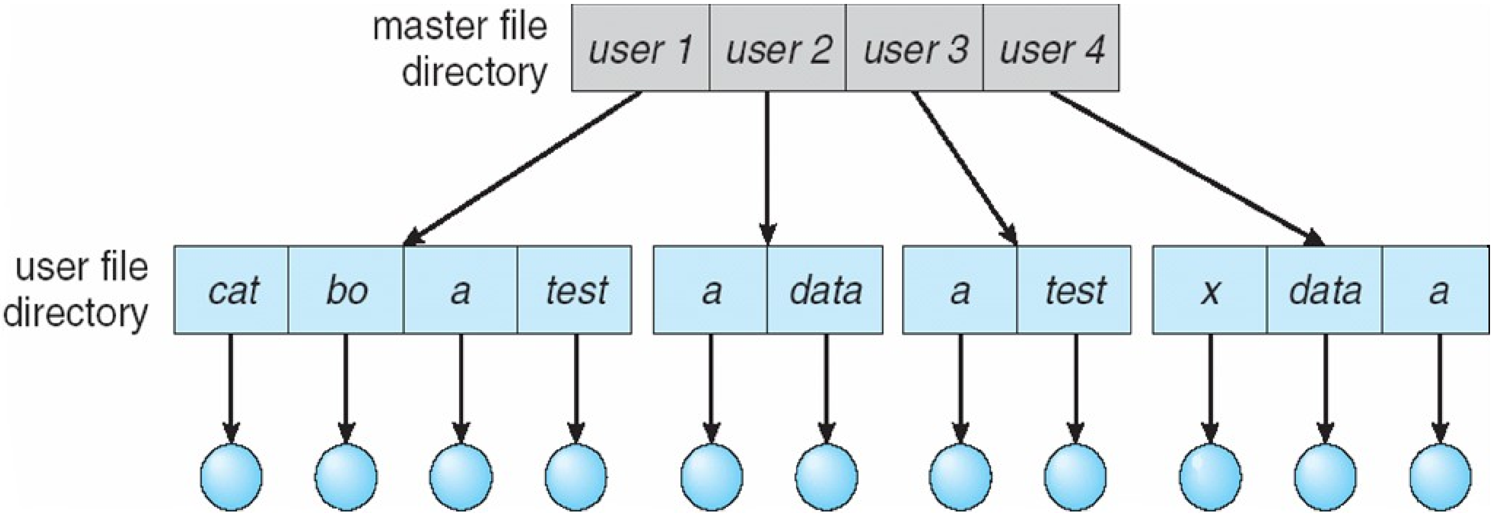


Naming problem

Grouping problem

Example: RT-11, from 1970s https://en.wikipedia.org/wiki/RT-11

Different types of “layouts”
Two level directory



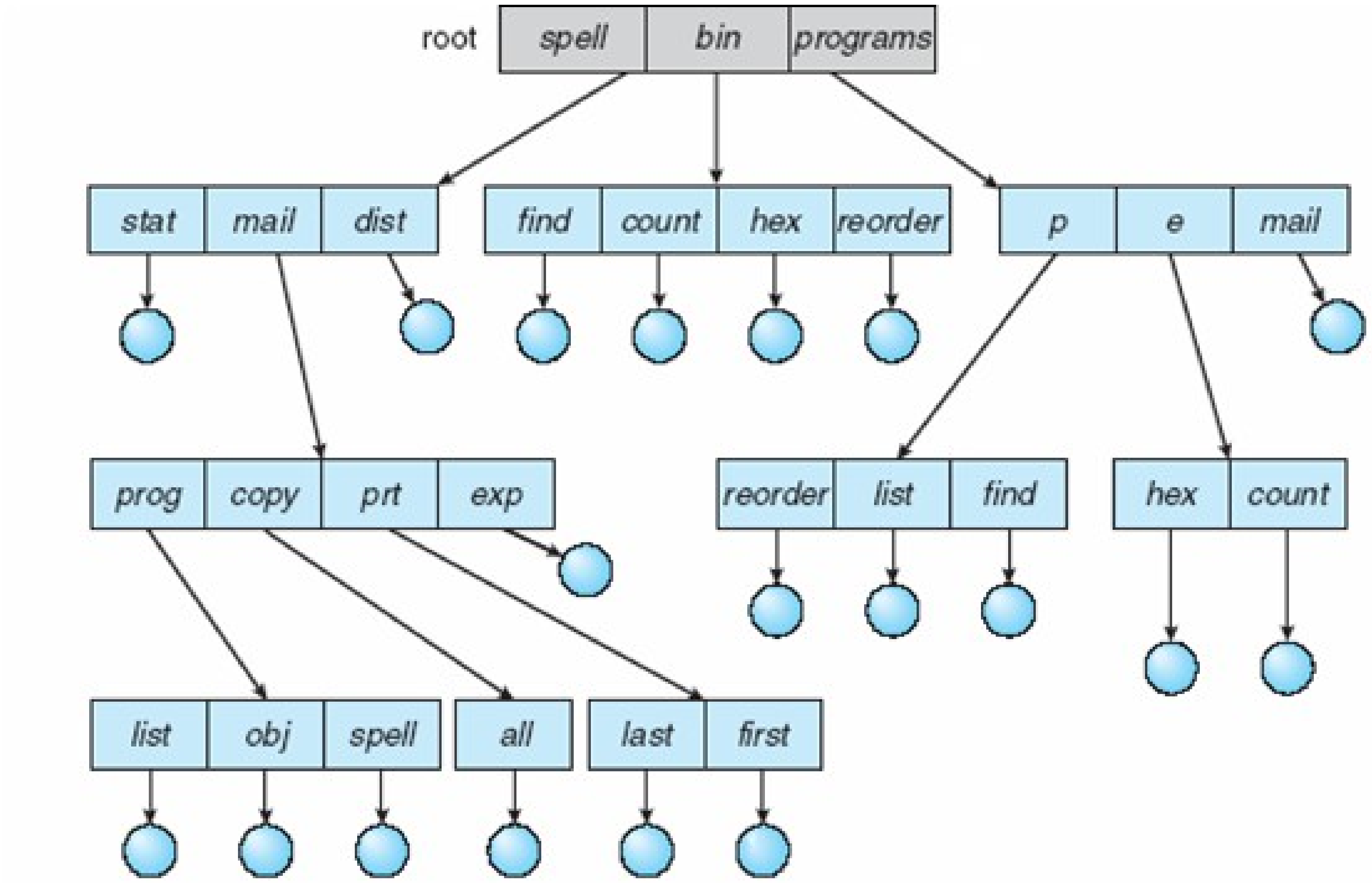
Path name

Can have the same file name for different user

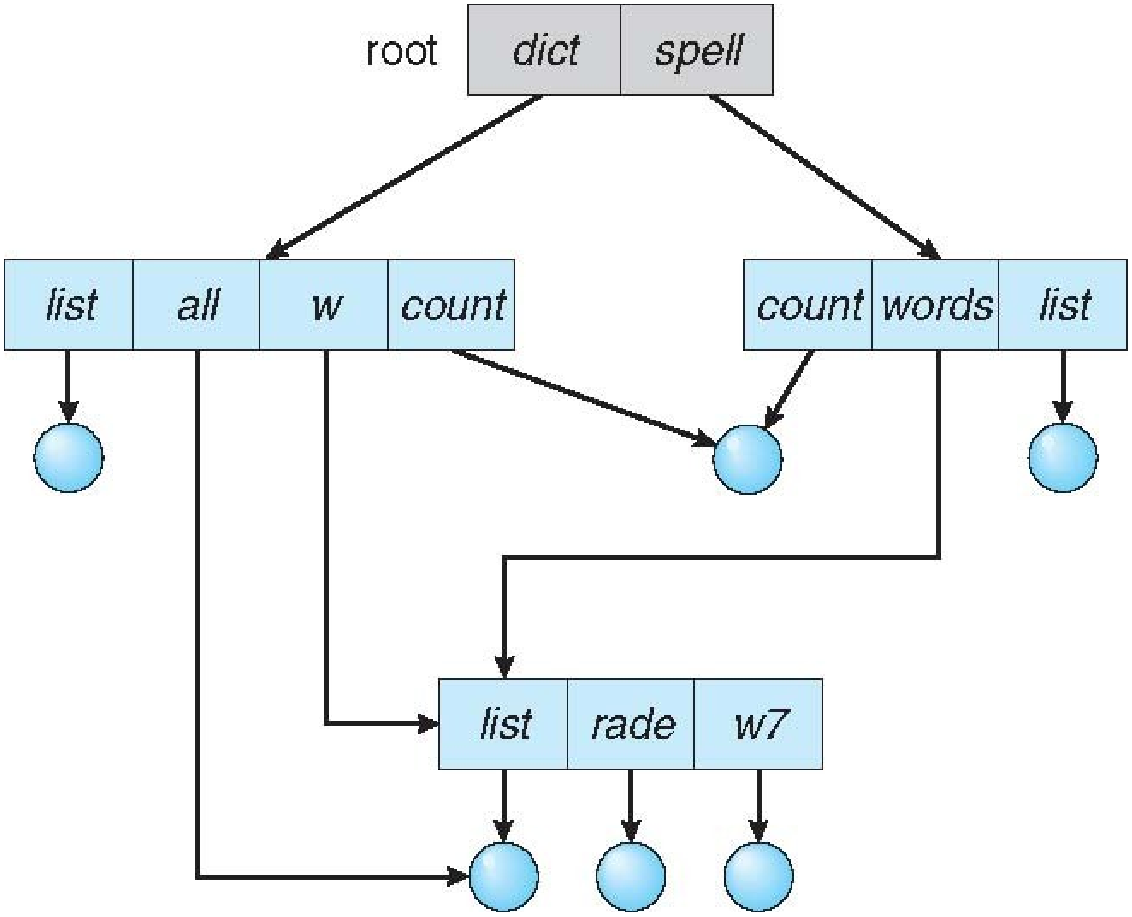
Efficient searching

No grouping capability

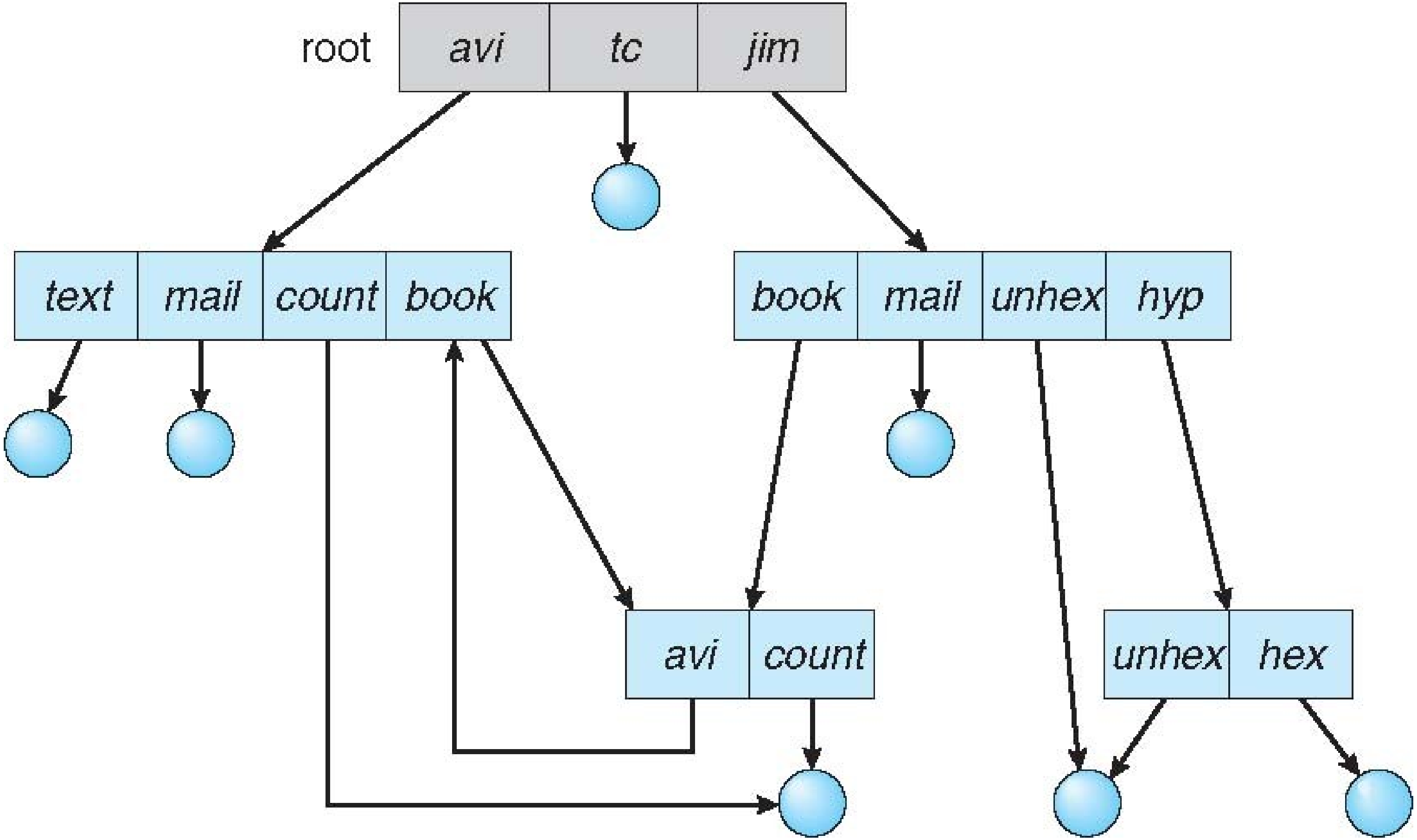
Tree Structured directories



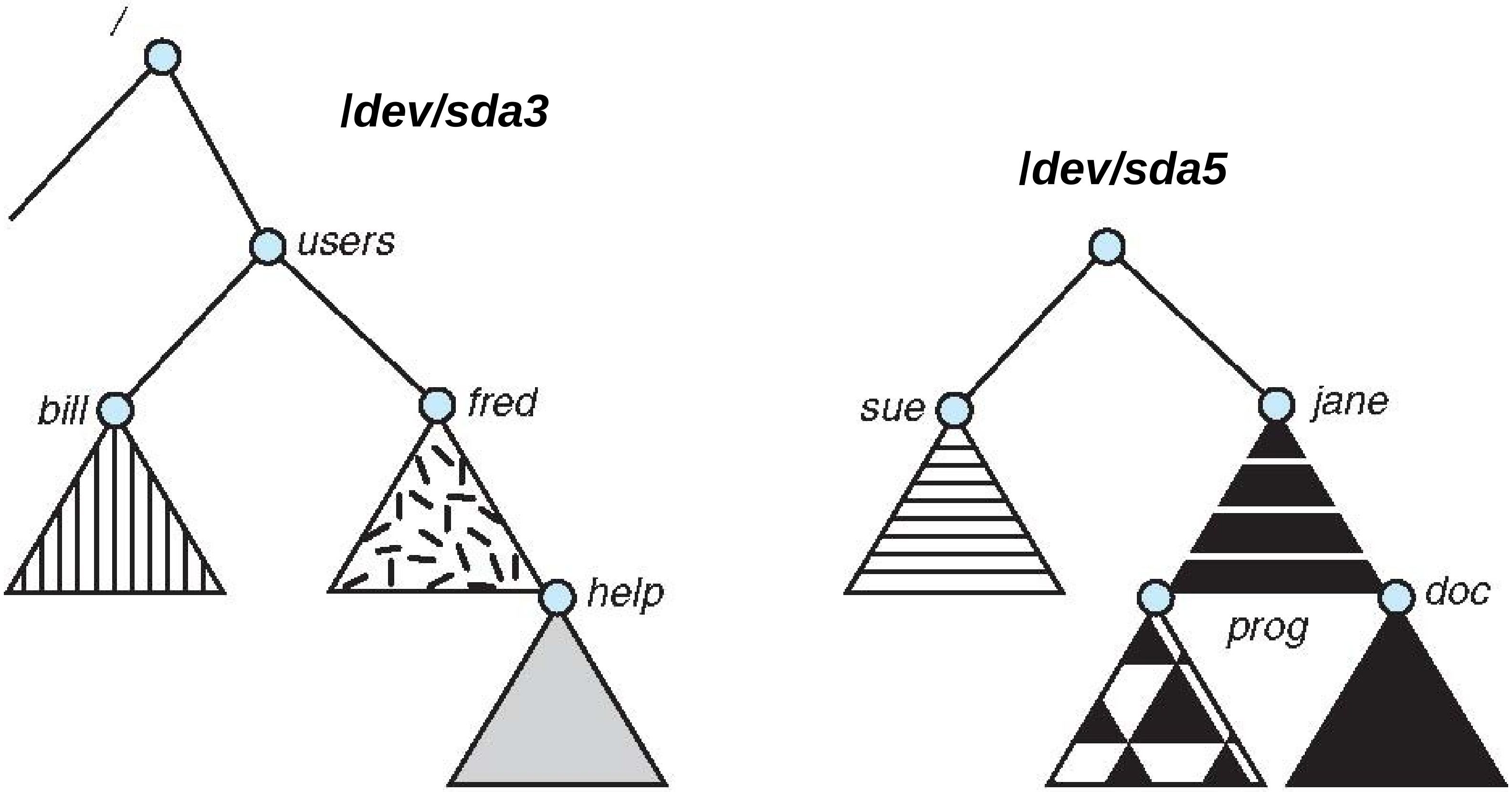
Acyclic Graph Directories



General Graph directory

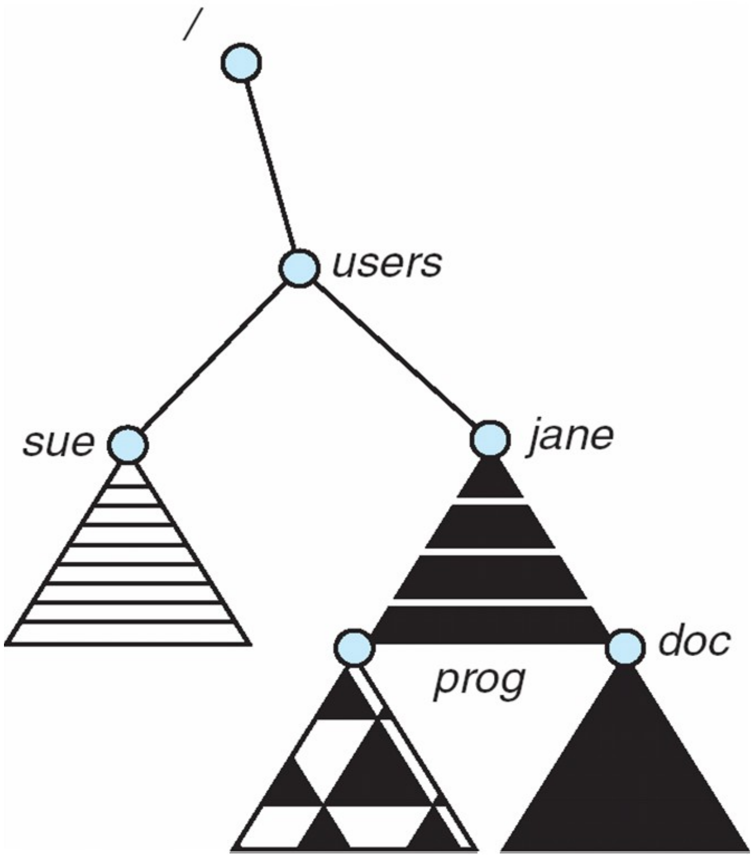


Mounting of a file system: before



(a) (b)

Mounting of a file system: after



$sudo mount */dev/sda5 /users*

Remote mounting: NFS

- **Network file system**
- **$ sudo mount 10.2.1.2:*/x/y /a/*b**
  - **The */x/*y partition on 10.2.1.2 will be made available under the folde */a/*b on this computer**
-

File sharing semantics

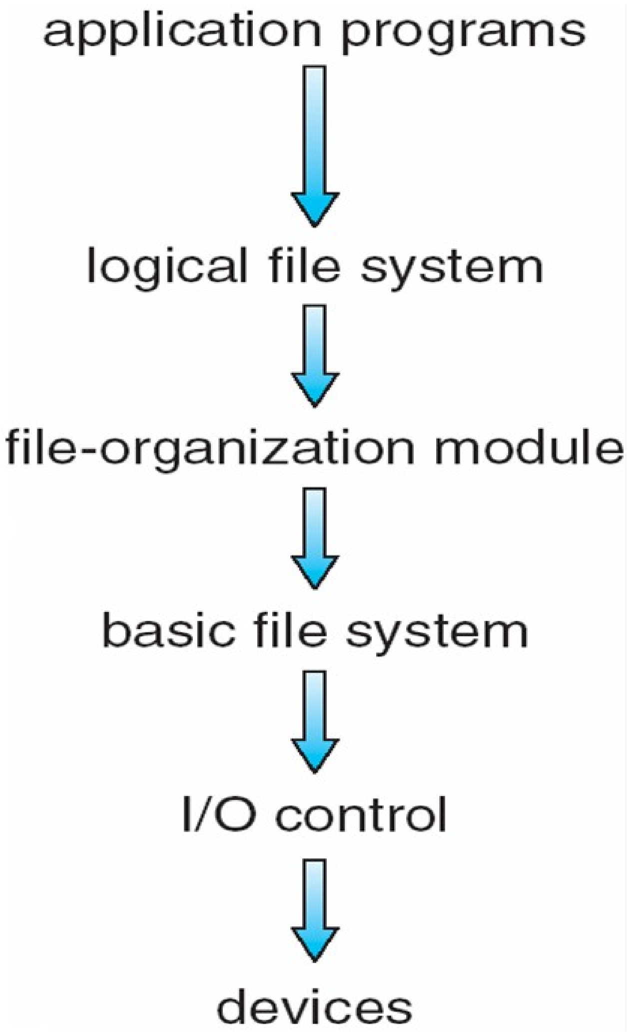
- Consistency semantics specify how multiple users are to access a shared file simultaneously
- Unix file system (UFS) implements:
  - Writes to an open file visible immediately to other users of the same open file
  - One mode of sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics
  - Writes only visible to sessions starting after the file is closed

Implementing file systems

File system on disk

- **What we know**
  - **Disk I/O in terms of sectors (512 bytes)**
  - **File system: implementation of acyclic graph using the linear sequence of sectors**
    - **Store a acyclic graph into array of “blocks”/“sectors”**
  - **Device driver available: gives sector/block wise access to the disk**

File system implementation: layering



File system: Layering

- Device drivers manage I/O devices at the I/O control layer
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- Basic file system given command like “retrieve block 123” translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- File organization module understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation
- Logical file system manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (inodes in Unix)
  - Directory management
  - Protection

File system implementation: Different problems to be solved

- What to do at boot time, how to locate kernel ?
- How to store directories and files on the partition ?
  - Complex problem. Hiearchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)
- How to manage list of free sectors/blocks?
- How to store the summary information about the complete file system : #files, #free-blocks, ...
- How to mount a file system , how to unmount?

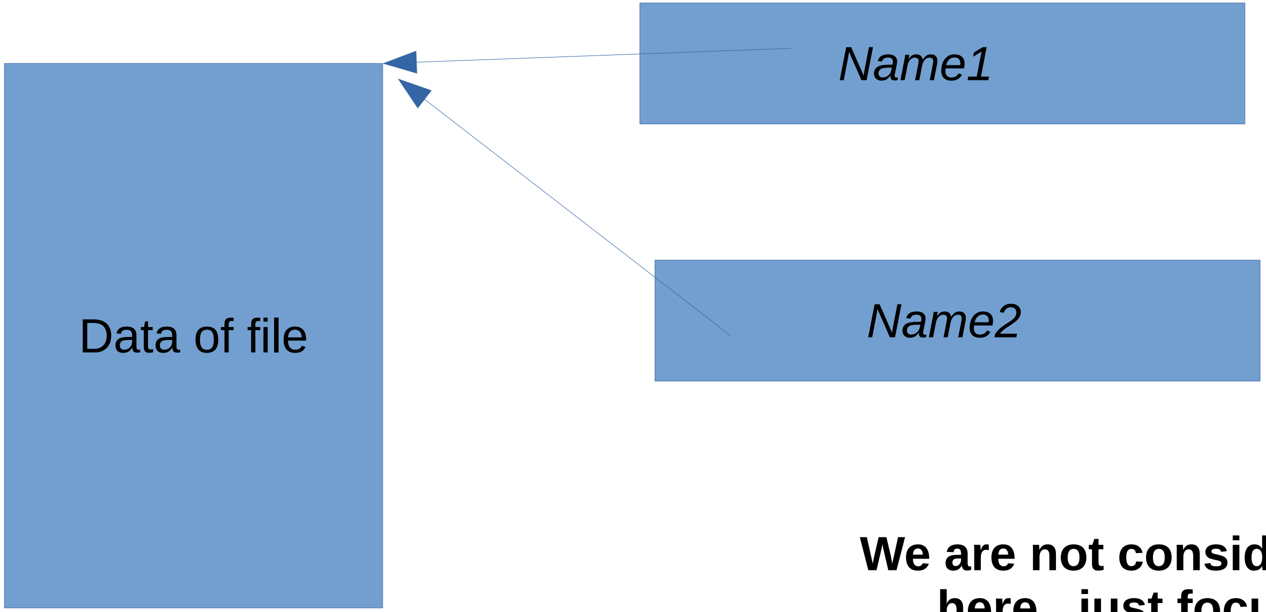
File system implementation: Different problems to be solved

- **About storing a file, how to store**
  - **Data**
  - **Attributes**
  - **Name**
  - **Link count**

The hard link problem

- Need to separate name from data !
  - */x/y   and /a/b* should be same file. How?
  - Both names should refer to same data !
  - Data is separated separately from name, and the name gives a “reference” to data
- What about attributes ?
  - They go with data! (not with name!)
- So solution was: indirection !

The hard link problem



We are not considering other problems here , just focussing on hard link problem

A typical file control block (inode)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Name is stored separately

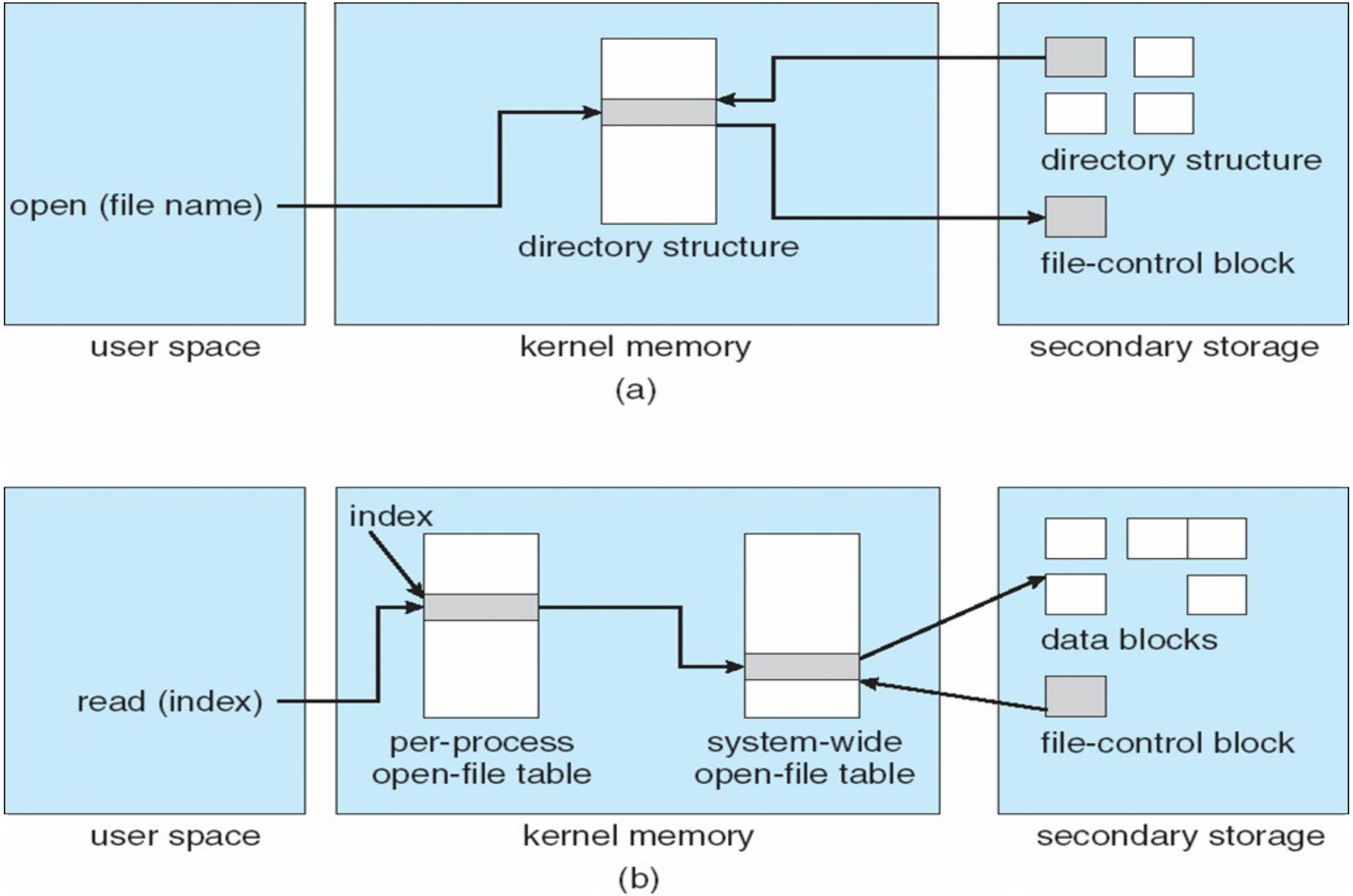
Where?

IN data block of directory

In memory data structures

- **Mount table**
  - storing file system mounts, mount points, file system types
- **See next slide for “file” related data structures**
- **Buffers**
  - hold data blocks from secondary storage

In memory data structures: for open,read,write, ...



Open returns a file handle for subsequent use

Data from read eventually copied to specified user process memory address

At boot time

- Root partition
  - Contains the file system hosting OS
  - “mounted” at boot time – contains “/”
    - Normally can’t be unmounted!
- Check all other partitions
  - Specified in /*etc*/fstab on Linux
  - Check if the data structure on them is consistent
    - Consistent != perfect/accurate/complete

Directory Implementation

- Problem
  - Directory contains files and/or subdirectories
  - Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
  - Directory needs to give location of each file on disk

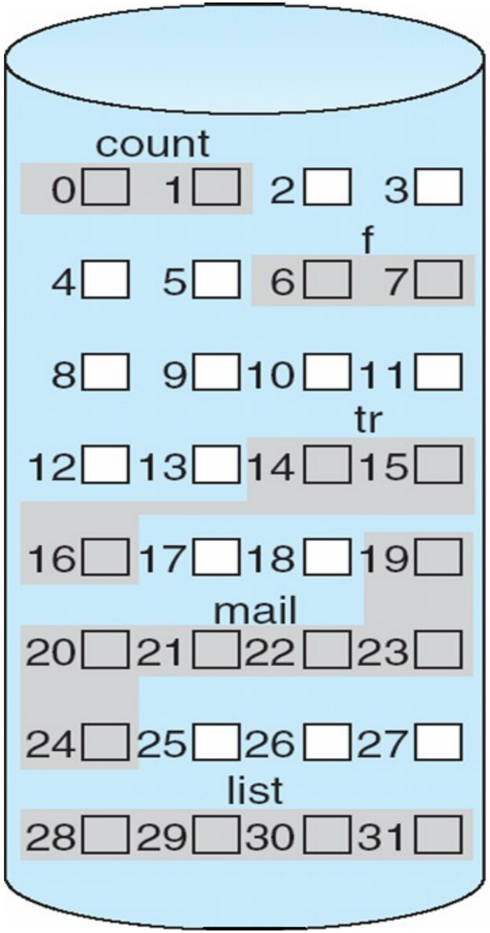
Directory Implementation

- Linear list of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
  - Ext2 improves upon this approach.
- Hash Table – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
  - Be created, expanded, deleted, shrunk, etc.
  - How to accommodate these requirements?

Contiguous Allocation of Disk Space



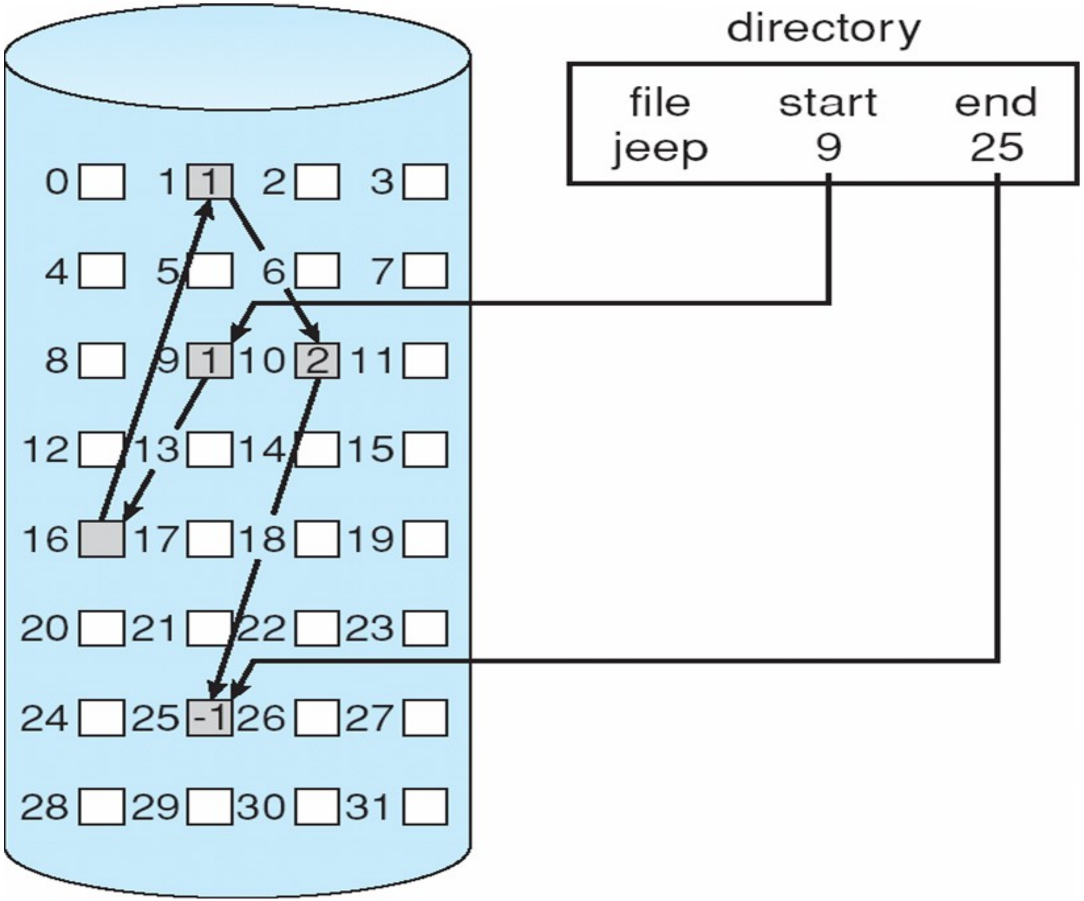
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

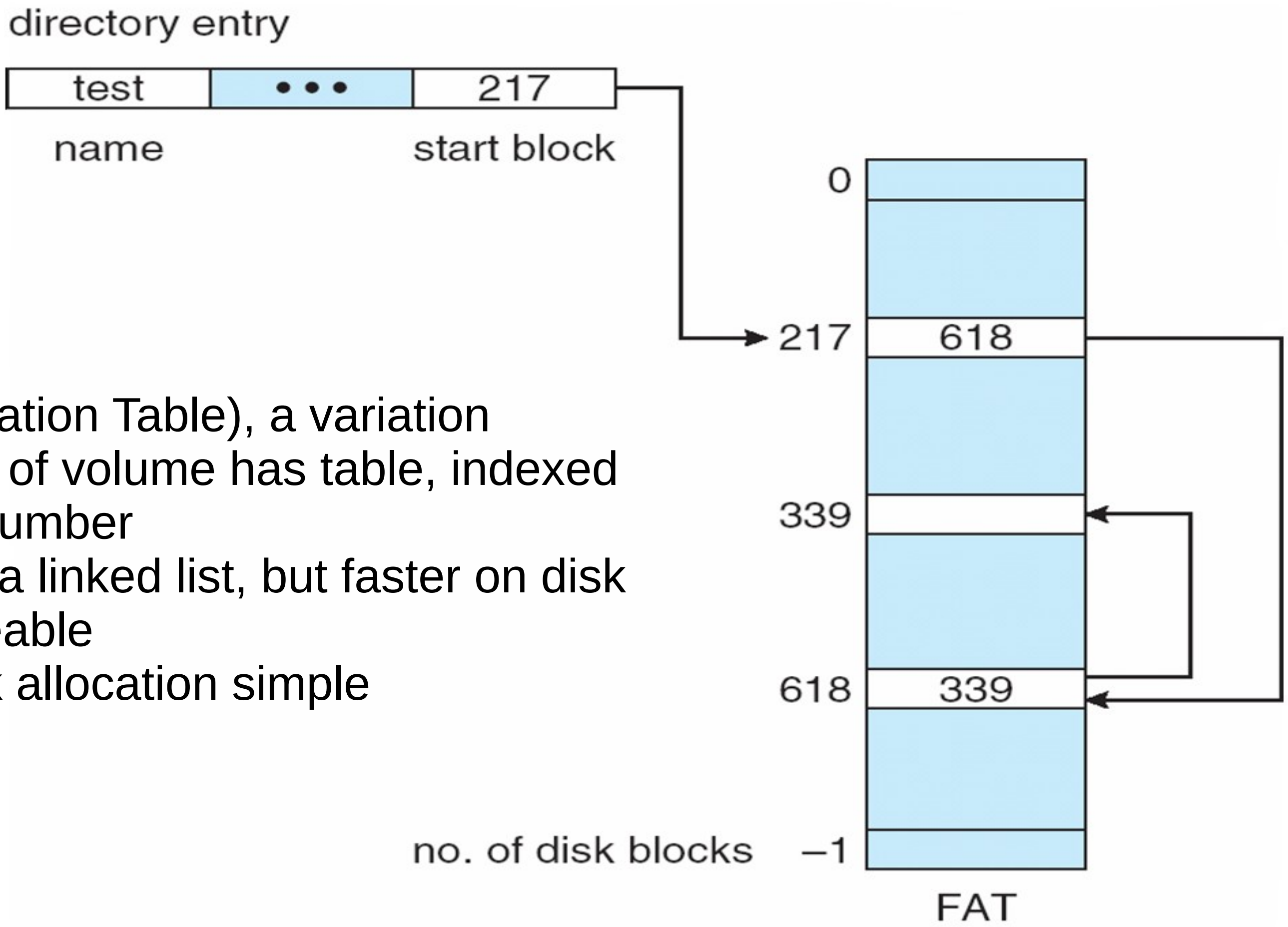
Linked allocation of blocks to a file



Linked allocation of blocks to a file

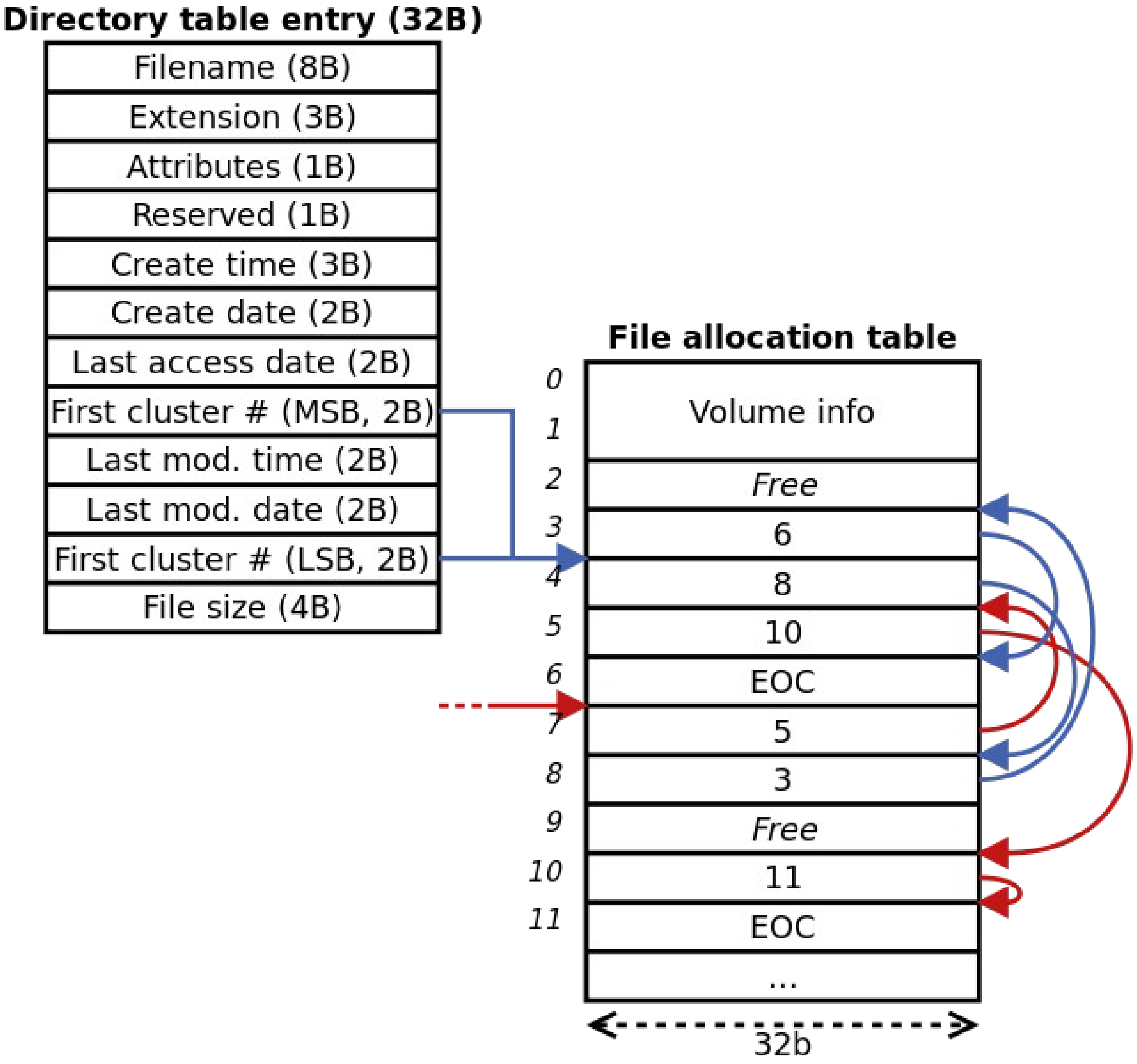
- Linked allocation
  - Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block (i.e. data + pointer to next block)
  - No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

FAT: File Allocation Table



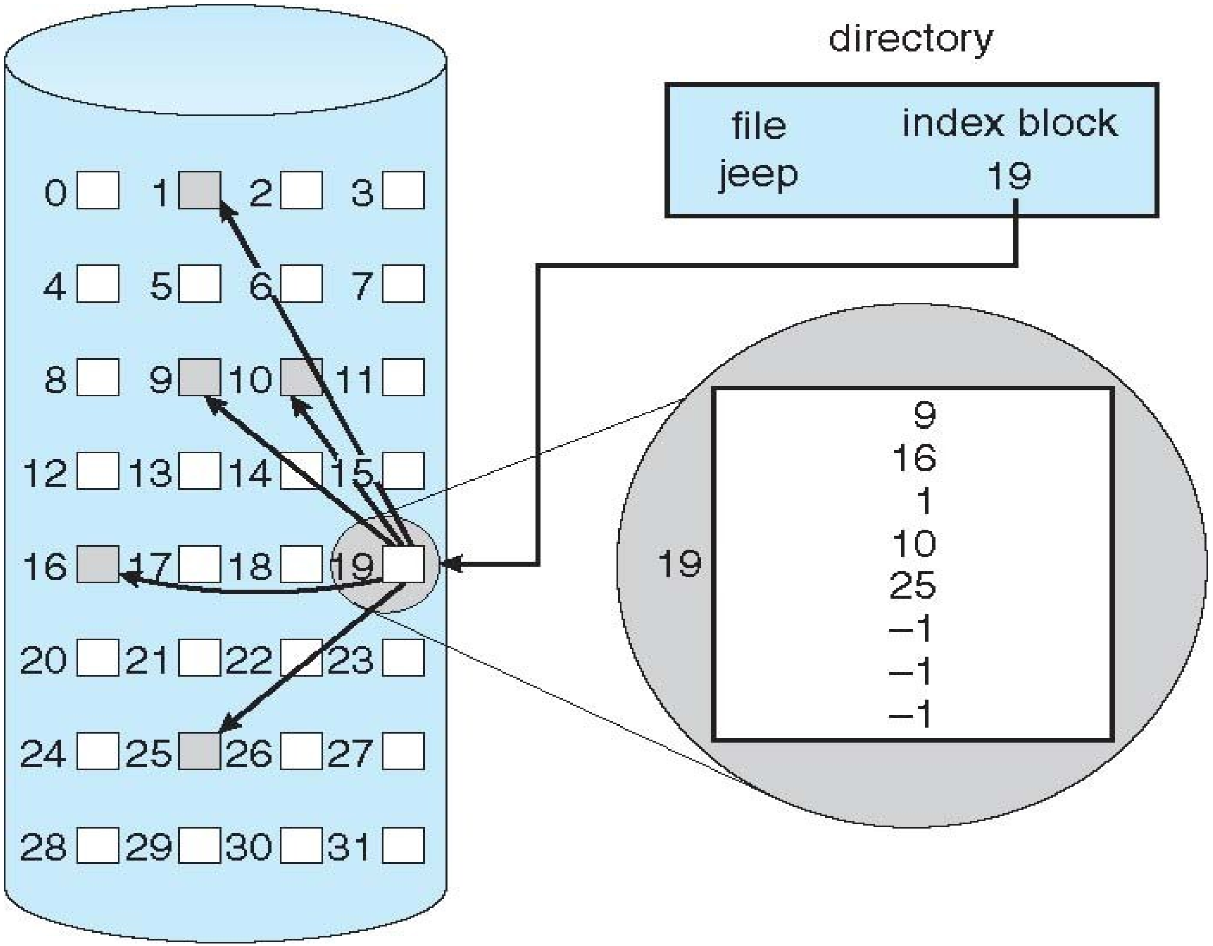
- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

FAT: File Allocation Table



Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

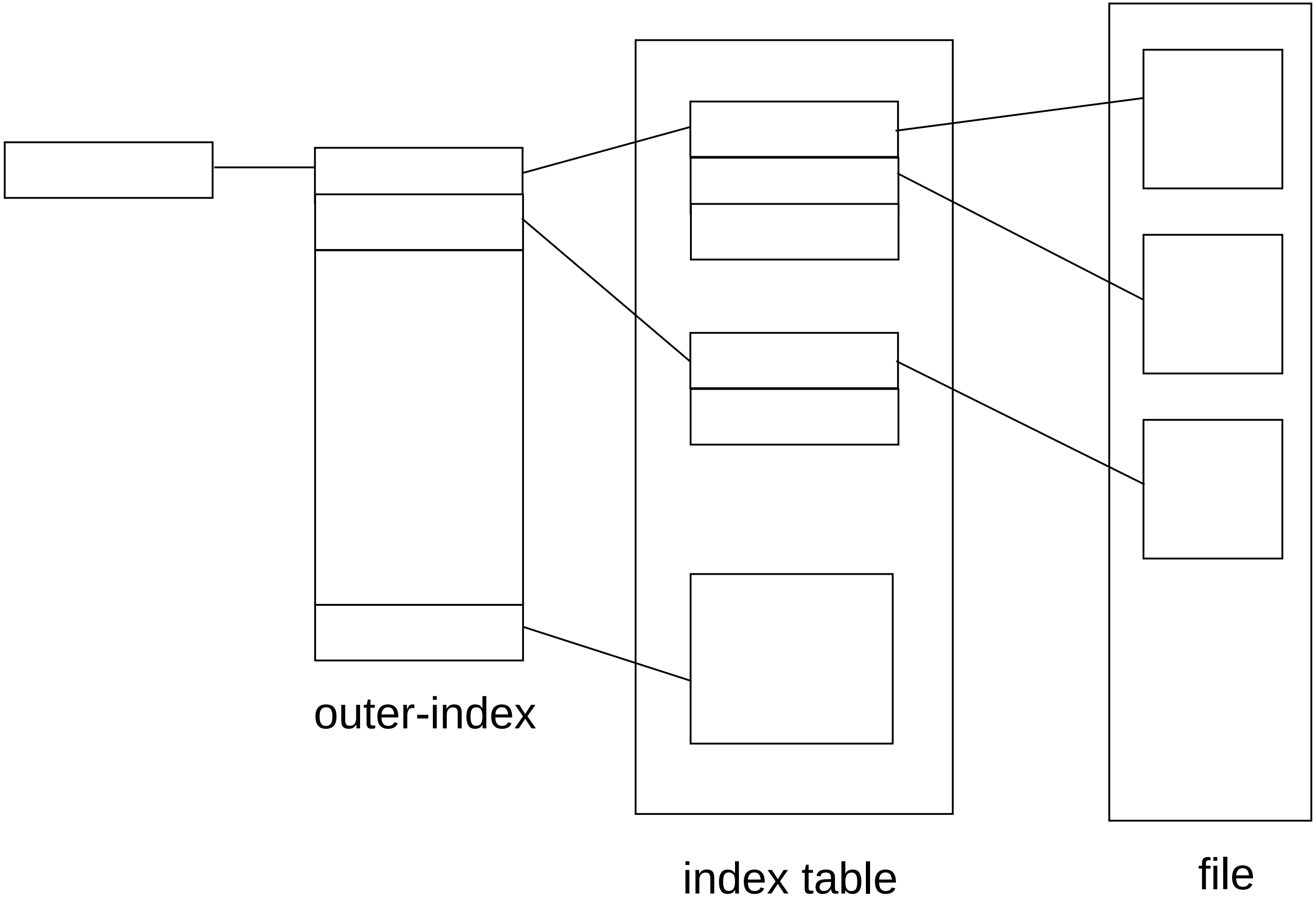
Indexed allocation



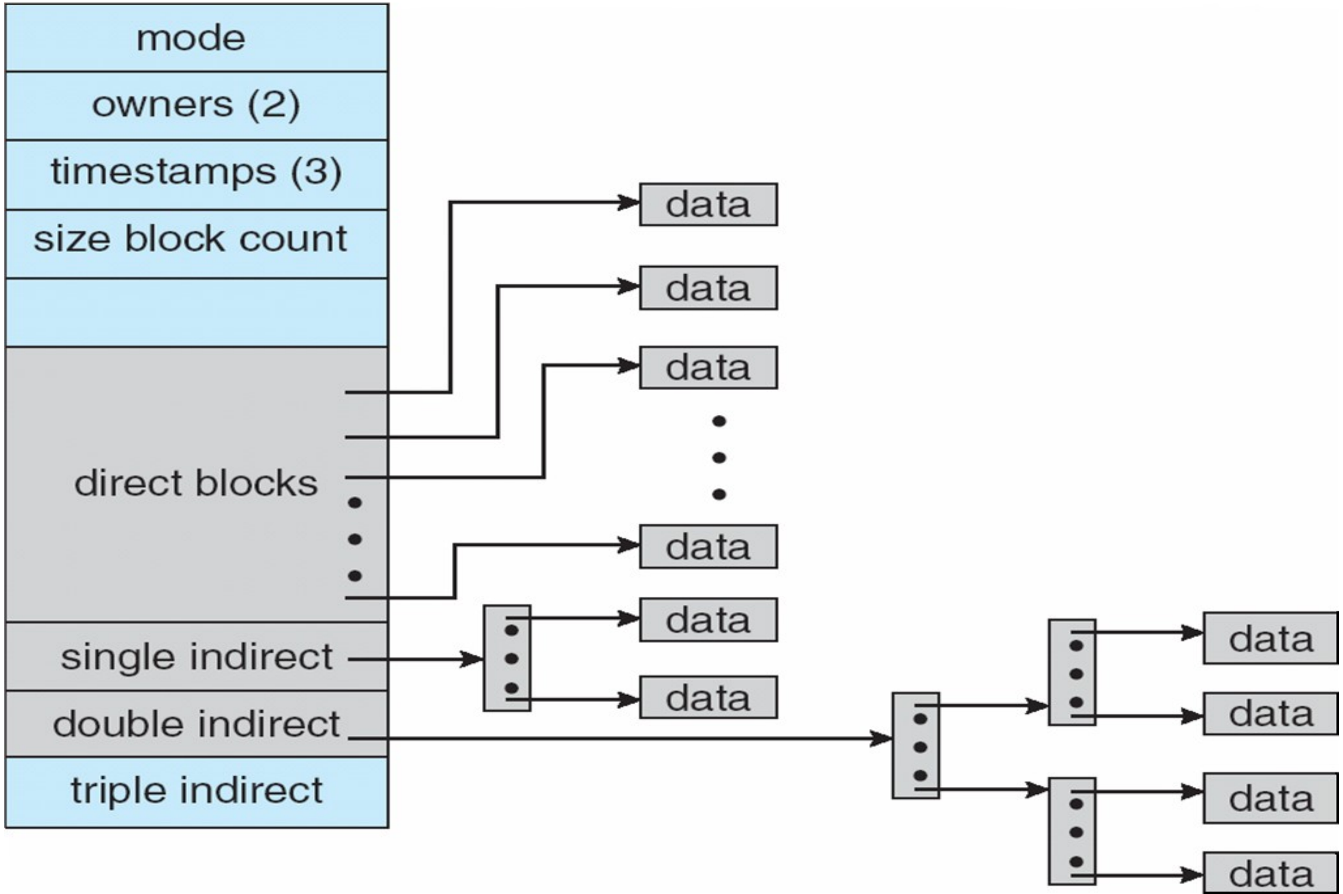
Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table

Multi level indexing



Unix UFS: combined scheme for block allocation



Free Space Management

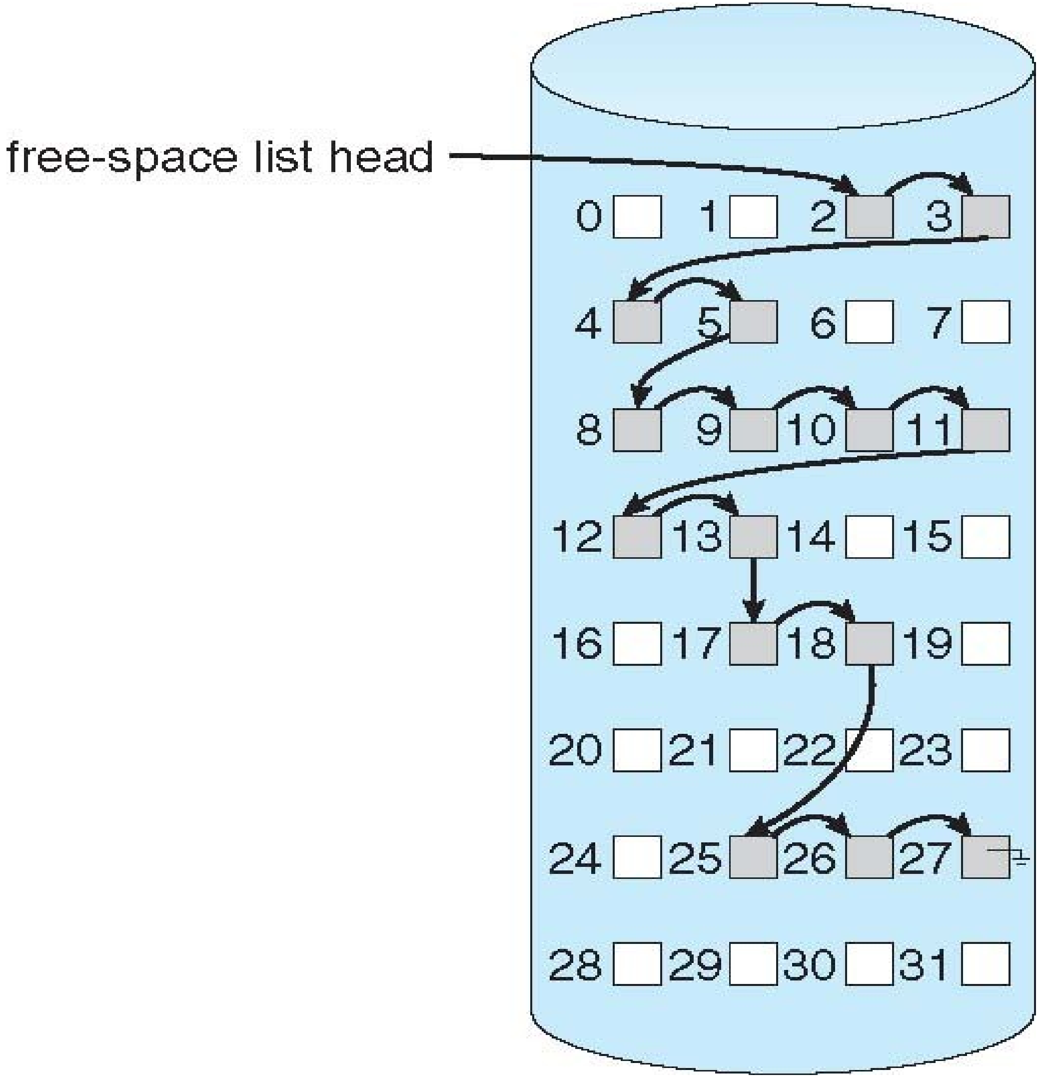
- File system maintains free-space list to track available blocks/clusters
  - Bit vector or bit map (n blocks)
  - Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
  - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

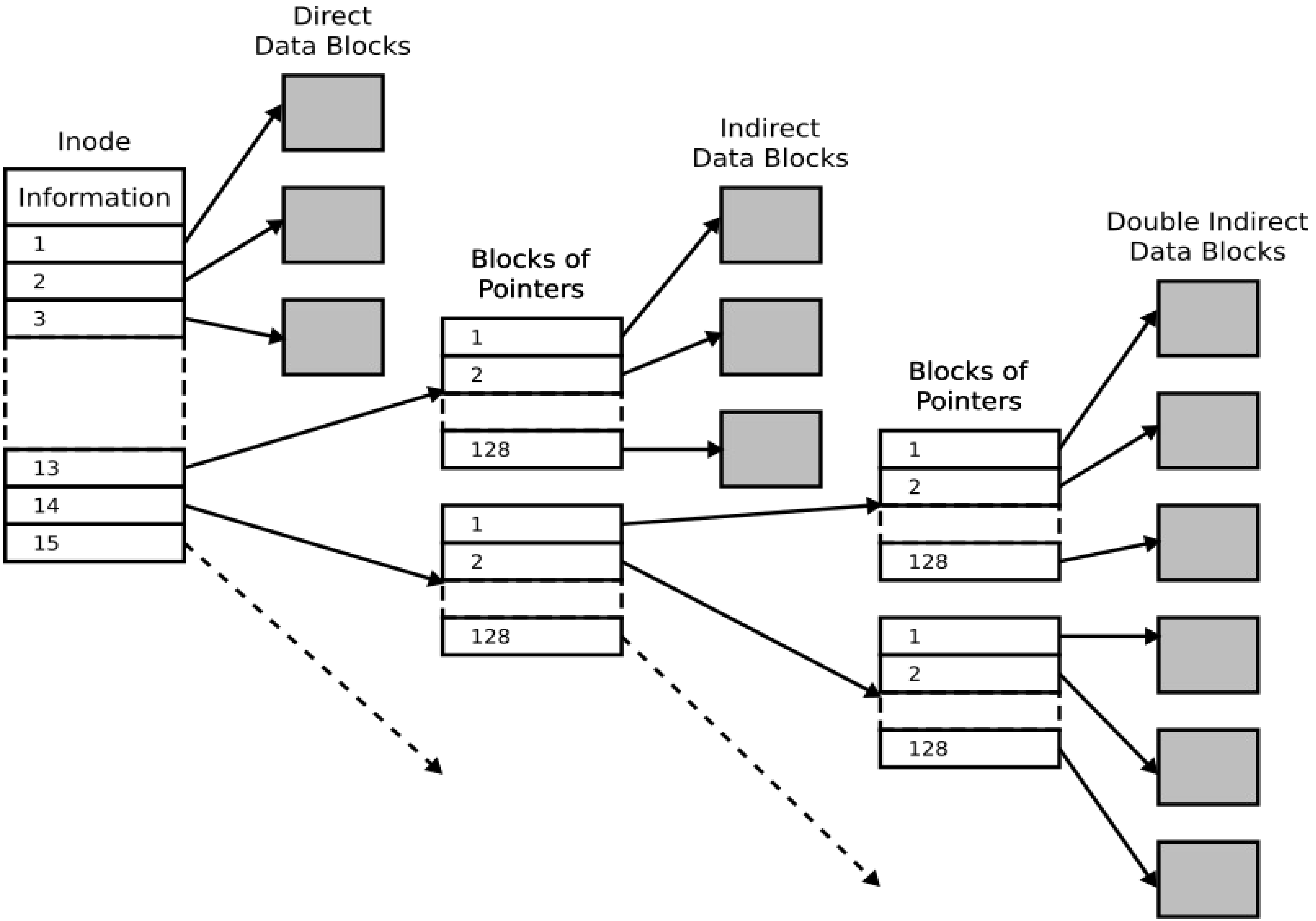
    001111001111110001100000011100000 ...
- A 1- TB disk with 4- KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^5 MB) to store its bitmap

Free Space Management: Linked list (not in memory, on disk!)



Ext2 FS layout

```
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 */
```

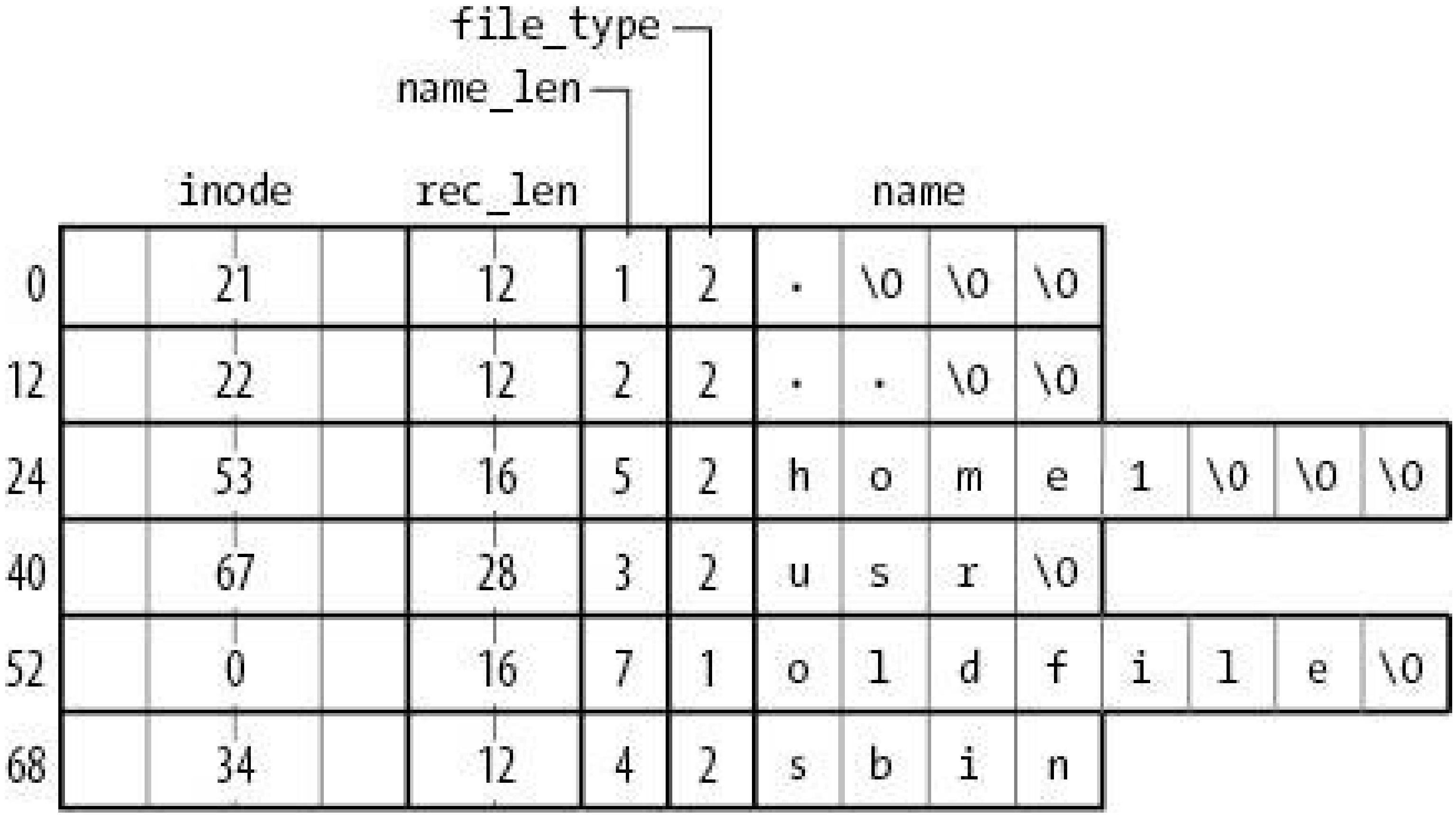


Inode in ext2

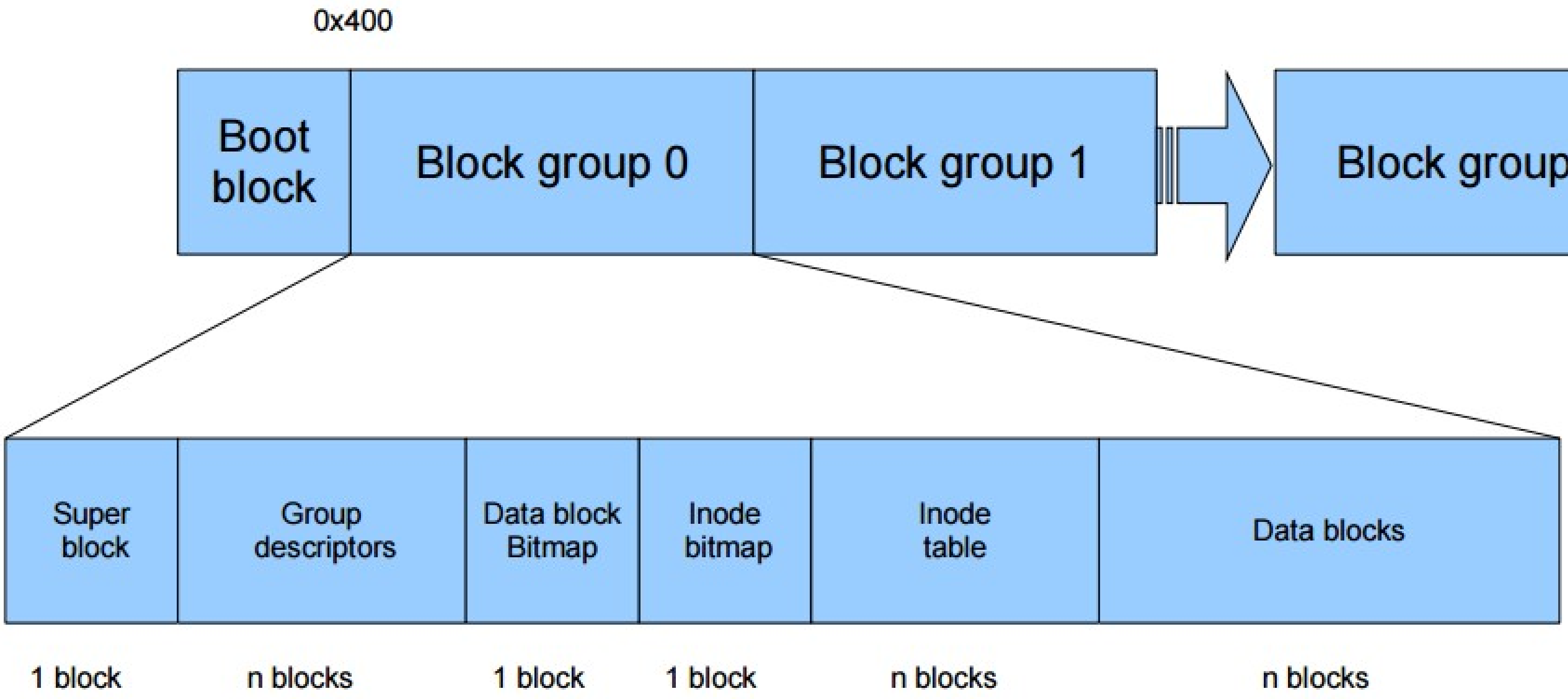
```
struct ext2_inode {
   ...
    union {
        struct {
            __le32  l_i_reserved1;
        } linux1;
        struct {
            __le32  h_i_translator;
        } hurd1;
        struct {
            __le32  m_i_reserved1;
        } masix1;
    } osd1;             /* OS dependent 1 */
    __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 */
```

```
struct ext2_inode {
  ...
   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 */
```

Ext2 FS Layout: Entries in directory's data blocks



Ext2 FS Layout



Size of block 4k,1k, etc

Determined at time of mkfs (formatting)

Calculations done by “mkfs” like this

- Block size = 4KB (specified to mkfs)
- Number of total blocks = size of partition / 4KB
  - How to get size of partition ?
- 4KB = 4 * 1024 * 8 = 32768 bits
- Data Block Bitmap, Inode Bitmap are always one block
- So
  - size of a group is 32,768 Blocks
  - #groups = #blocks-in-partition / 32,768

```
struct ext2_super_block {
    __le32  s_inodes_count;      /* Inodes count */
    __le32  s_blocks_count;      /* Blocks count */
    __le32  s_r_blocks_count;    /* Reserved blocks count */
    __le32  s_free_blocks_count;     /* Free blocks count */
    __le32  s_free_inodes_count;     /* Free inodes count */
    __le32  s_first_data_block; /* First Data Block */
    __le32  s_log_block_size;    /* Block size */
    __le32  s_log_frag_size;     /* Fragment size */
    __le32  s_blocks_per_group; /* # Blocks per group */
    __le32  s_frags_per_group;  /* # Fragments per group */
    __le32  s_inodes_per_group; /* # Inodes per group */
    __le32  s_mtime;         /* Mount time */
    __le32  s_wtime;         /* Write time */
   __le16  s_mnt_count;         /* Mount count */
    __le16  s_max_mnt_count;     /* Maximal mount count */
    __le16  s_magic;         /* Magic signature */
    __le16  s_state;         /* File system state */
    __le16  s_errors;        /* Behaviour when detecting errors */
```

```
struct ext2_super_block {
...
    __le16  s_minor_rev_level;  /* minor revision level */
    __le32  s_lastcheck;        /* time of last check */
    __le32  s_checkinterval;    /* max. time between checks */
    __le32  s_creator_os;       /* OS */
    __le32  s_rev_level;        /* Revision level */
    __le16  s_def_resuid;       /* Default uid for reserved blocks */
    __le16  s_def_resgid;       /* Default gid for reserved blocks */
    __le32  s_first_ino;        /* First non-reserved inode */
    __le16  s_inode_size;       /* size of inode structure */
    __le16  s_block_group_nr;   /* block group # of this superblock */
    __le32  s_feature_compat;   /* compatible feature set */
    __le32  s_feature_incompat;     /* incompatible feature set */
    __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
    __u8    s_uuid[16];     /* 128-bit uuid for volume */
    char    s_volume_name[16];  /* volume name */
    char    s_last_mounted[64];     /* directory where last mounted */
    __le32  s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {
...
    __u8    s_prealloc_blocks;  /* Nr of blocks to try to preallocate*/
    __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
    __u16   s_padding1;
    /*
     * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
     */
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */
    __u32   s_journal_inum;     /* inode number of journal file */
    __u32   s_journal_dev;      /* device number of journal file */
    __u32   s_last_orphan;      /* start of list of inodes to delete */
    __u32   s_hash_seed[4];     /* HTREE hash seed */
    __u8    s_def_hash_version; /* Default hash version to use */
    __u8    s_reserved_char_pad;
    __u16   s_reserved_word_pad;
    __le32  s_default_mount_opts;
    __le32  s_first_meta_bg;    /* First metablock block group */
    __u32   s_reserved[190];    /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;        /* Blocks bitmap block */
    __le32  bg_inode_bitmap;        /* Inodes bitmap block */
    __le32  bg_inode_table;     /* Inodes table block */
    __le16  bg_free_blocks_count;   /* Free blocks count */
    __le16  bg_free_inodes_count;   /* Free inodes count */
    __le16  bg_used_dirs_count; /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

Traversal / path-name resolution

```
//resolving /a/b

s = read_superblock(); // struct
g = read_bg_descriptors(); // array
inode getinode(int n) {
 calculate the block number for n'th inode
 (using info from superblock, bg descriptors, block-size etc)
 read that block
 extract inode from block
 return inode
```

Let’s see a program to read superblock of an ext2 file system.