File Systems

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

Introduction

- **Human end user's view of file sytem on a modern desktop operating system**
  - **Files, directories(folders), hierarchy – acyclic graph like structure**
  - **Windows Vs Linux logical organization: multiple partitions (C:, D:,etc.), vs single logical namespace starting at "/"**

Introduction

- Secondary and Tertiary memory
  - Hard disks, Pen drives, CD-ROMs, DVDs, Magnetic Tapes, Portable disks, etc. Used for storing files
  - Each comes with a hardware “controller” that acts as an intermediary in the hardware/software boundary
  - IDE, SATA, SCSI, SAS, etc. Protocols : Different types of cables, speeds, signaling mechanisms
  - Controllers provide a block/sector based read/write access
    - Block size is most typically 512 bytes
    - Can’t read byte no. 33 directly. Must read sector 0 (byte 0 to 511) in memory and then access the byte no 33 in memory.

Introduction

- OS and File system
  - OS bridges the gap between end user and stoage hardware controller
  - Provides data structure to map the logical view of end users onto disk storage
  - Essentially an implementation of the acyclic graph on the sequential sector-based disk storage
    - Both in memory and on-disk
  - Provides system calls (open, read, write, ..., etc.) to enable access to files, folders

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

What is a file?

- A sequence of bytes , with
  - A name
  - Permissions
  - Owner
  - Timestamps,
  - Etc.
- Types: Text files, binary files
  - Text: All bytes are human readable
  - Binary: Non-text
- Types: ODT, MP4, TXT, DOCX, etc.
  - Most typically describing the organization of the data inside the file
  - Each type serving the needs of a particular application

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 to be
  - Just a sequence of bytes
    - E.g. a text file
  - Sequence of records/structures
    - E.g. a file of student records
  - 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

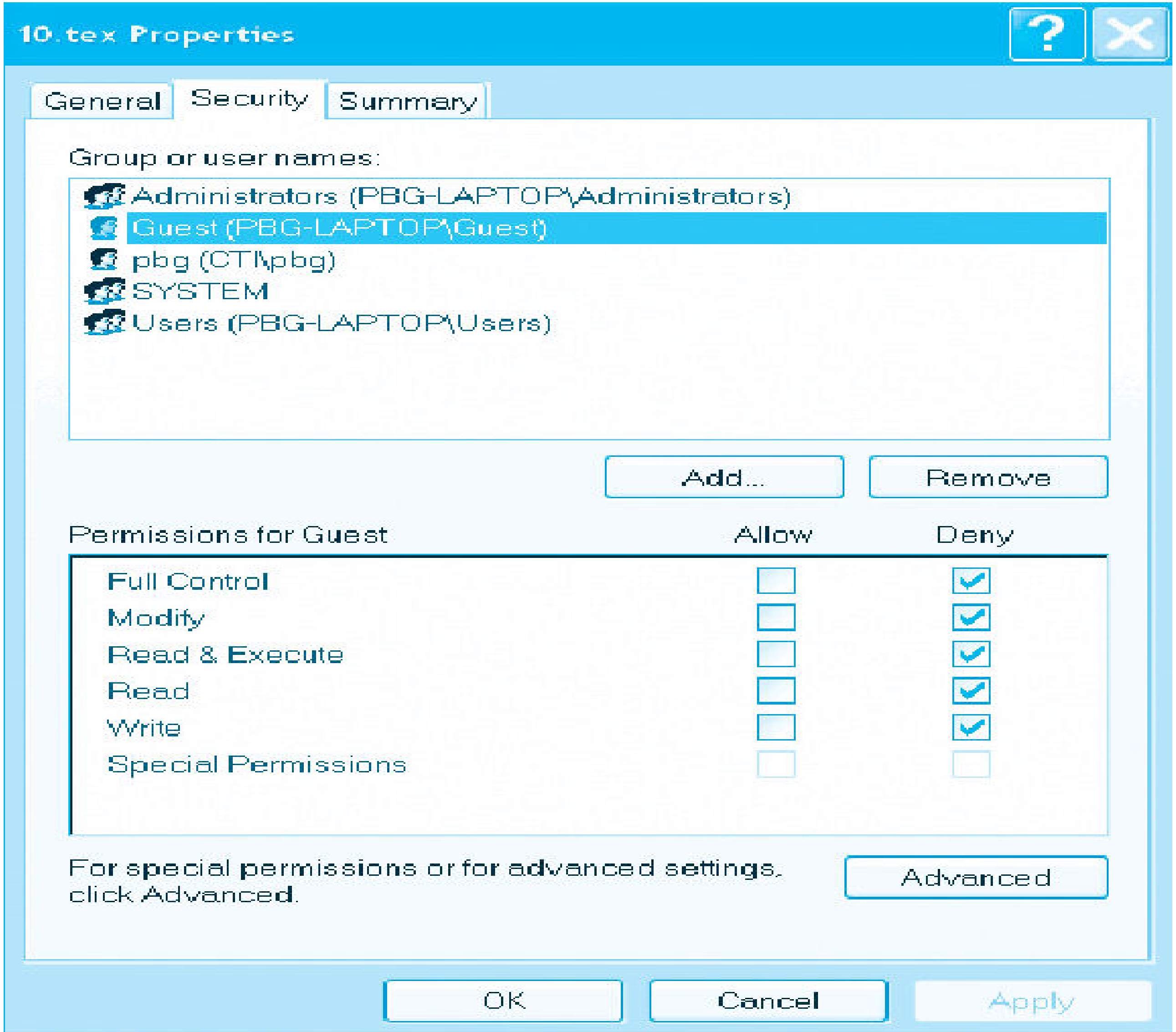
File protection attributes

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List
- Linux: See commands: "chown", "chgrp" and "chmod"

- Linux file permissions
  - For owner, group and others
  - Read, write and execute
  - Total 9 permissions

A Sample UNIX Directory Listing

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5	pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/



Windows XP Access List

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

Device Drivers

- Hardware manufacturers provide “hardware controllers” for their devices
- Hardware controllers can operate the hardware, as instructed
- Hardware controllers are instructed by writing to particular I/O ports using CPU’s machine instructions
  - This bridges the hardware-software gap
- OS programmers, typically, write one “device driver” code that interacts with one hardware controller
  - This is pure C code
  - Which calls I/O instructions for hardware controller
  - This code is independent of most of the OS code
  - Often this code is like an add-on Module , eg. Linux kernel

Disk device driver

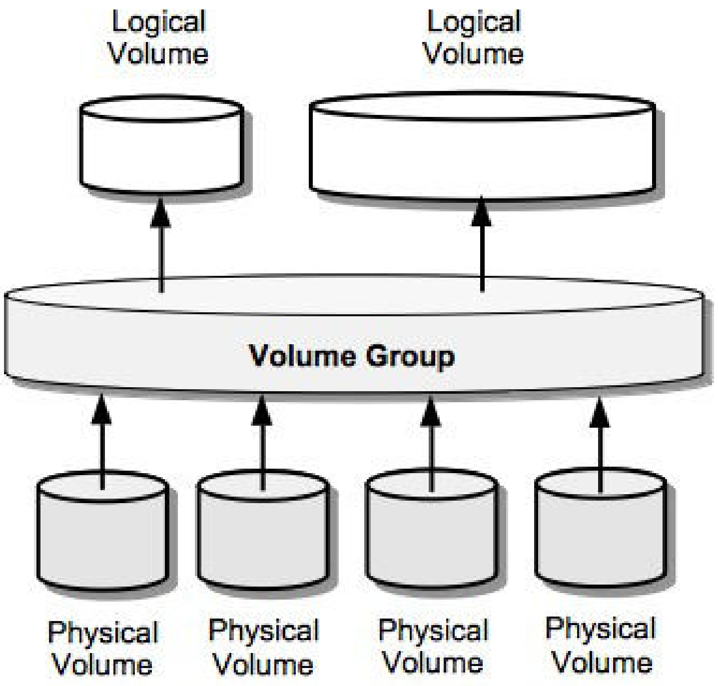
- OS views the disk as a logical sequence of blocks
  - OS’s assumed block size may be > sector size
- OS Talks to disk controller
- Helps the OS Convert it’s view of “logical block” of the disk, into physical sector numbers
- Acts as a translator between rest of OS and hardware controller
- xv6: ide.c

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

Volume Managers

- **Special type of kernel device drives, which reside on top of disk device drivers**
- **Provide a more abstract view of the underlying hardware**
- **E.g. Can combine two physical hard disks, and present them as one**
- **Allow end users to**
  - **“combine one more more physical disks to create physical volumes”**
  - **“create volume groups out of a set of physical volumes”,**
  - **“create logical volums within volume groups”**

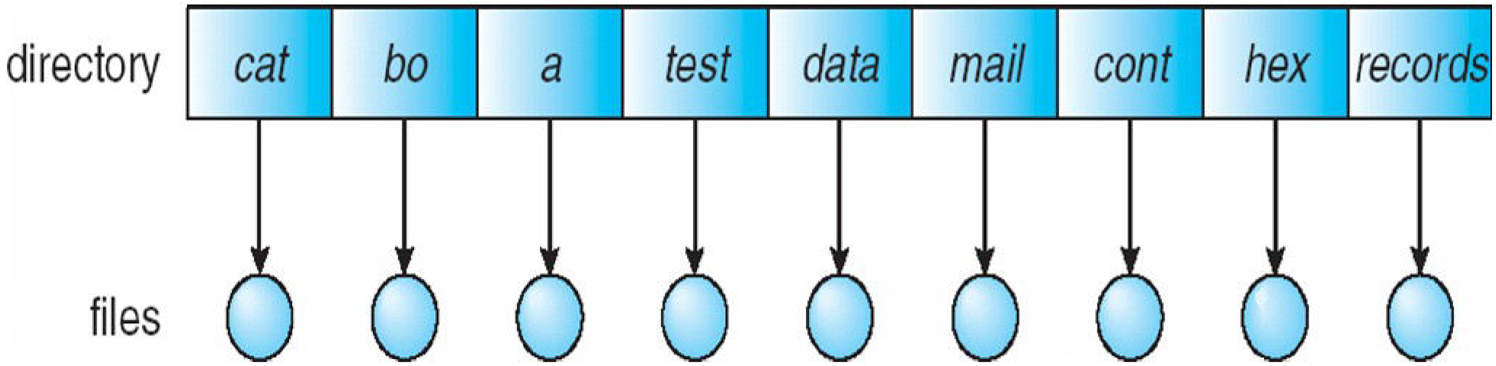


Logical Volume Management Components

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”

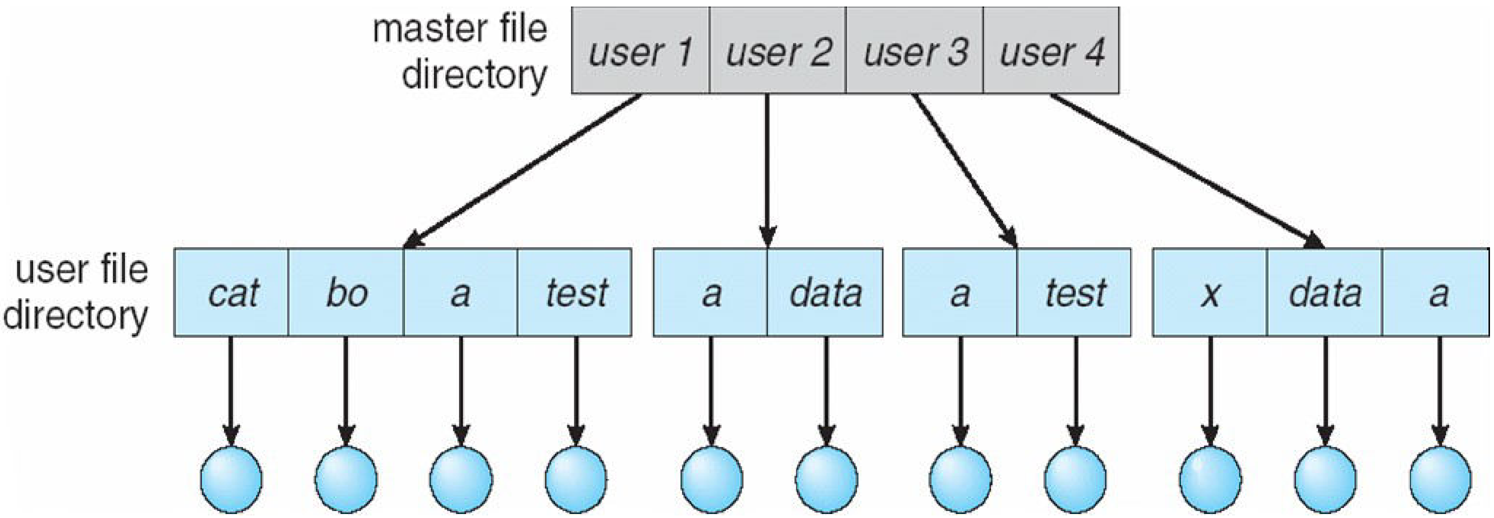
Different types of “layouts”
Single level directory



Naming problem

Grouping
problem

Different types of “layouts”
Two level directory



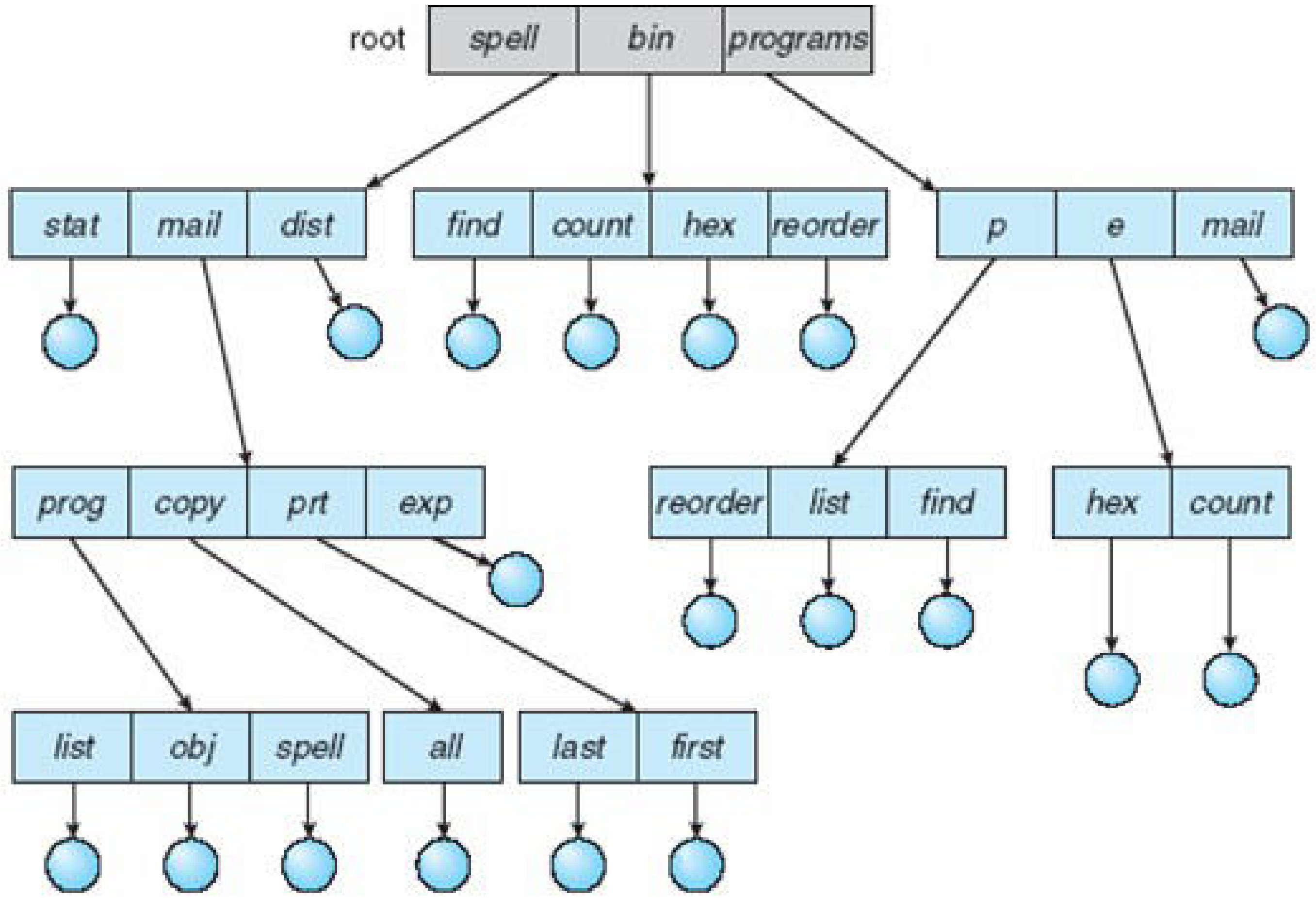
Path name

Can have the same file name for different user

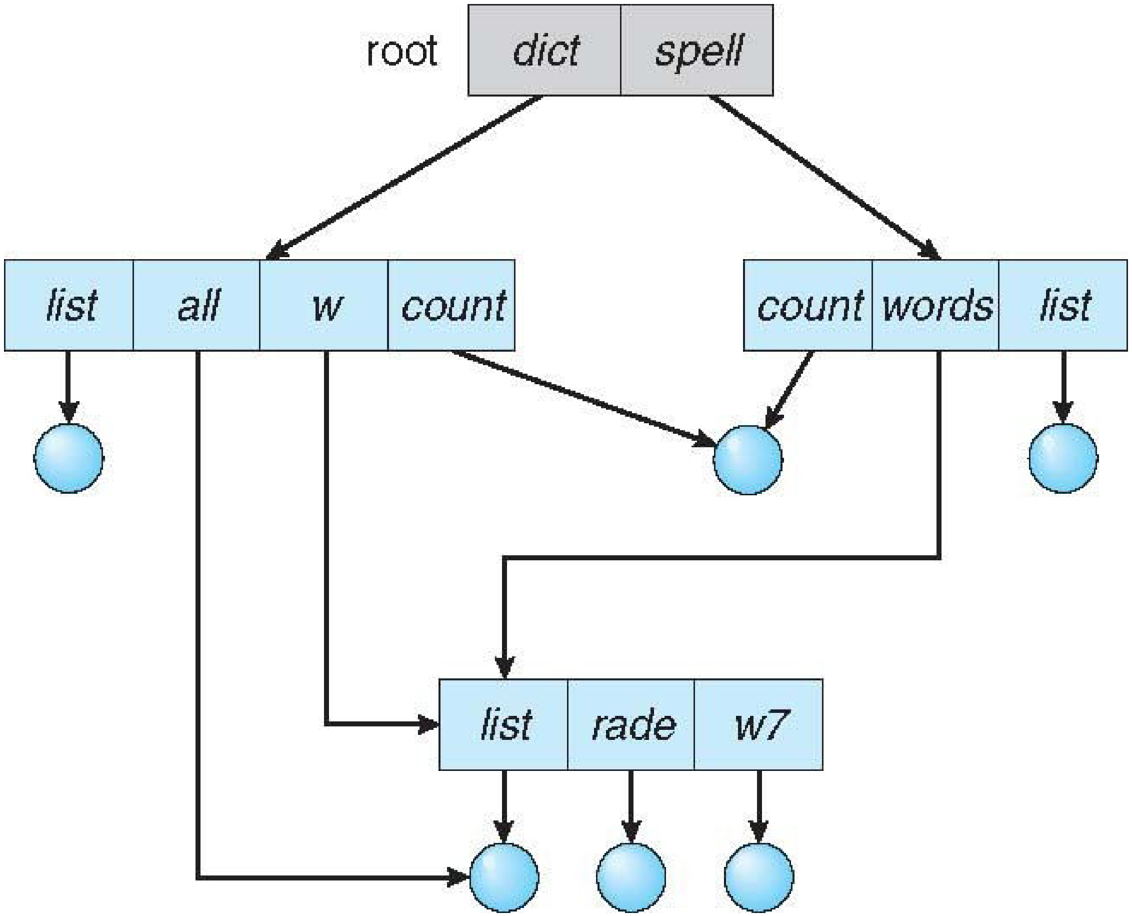
Efficient searching

No grouping capability

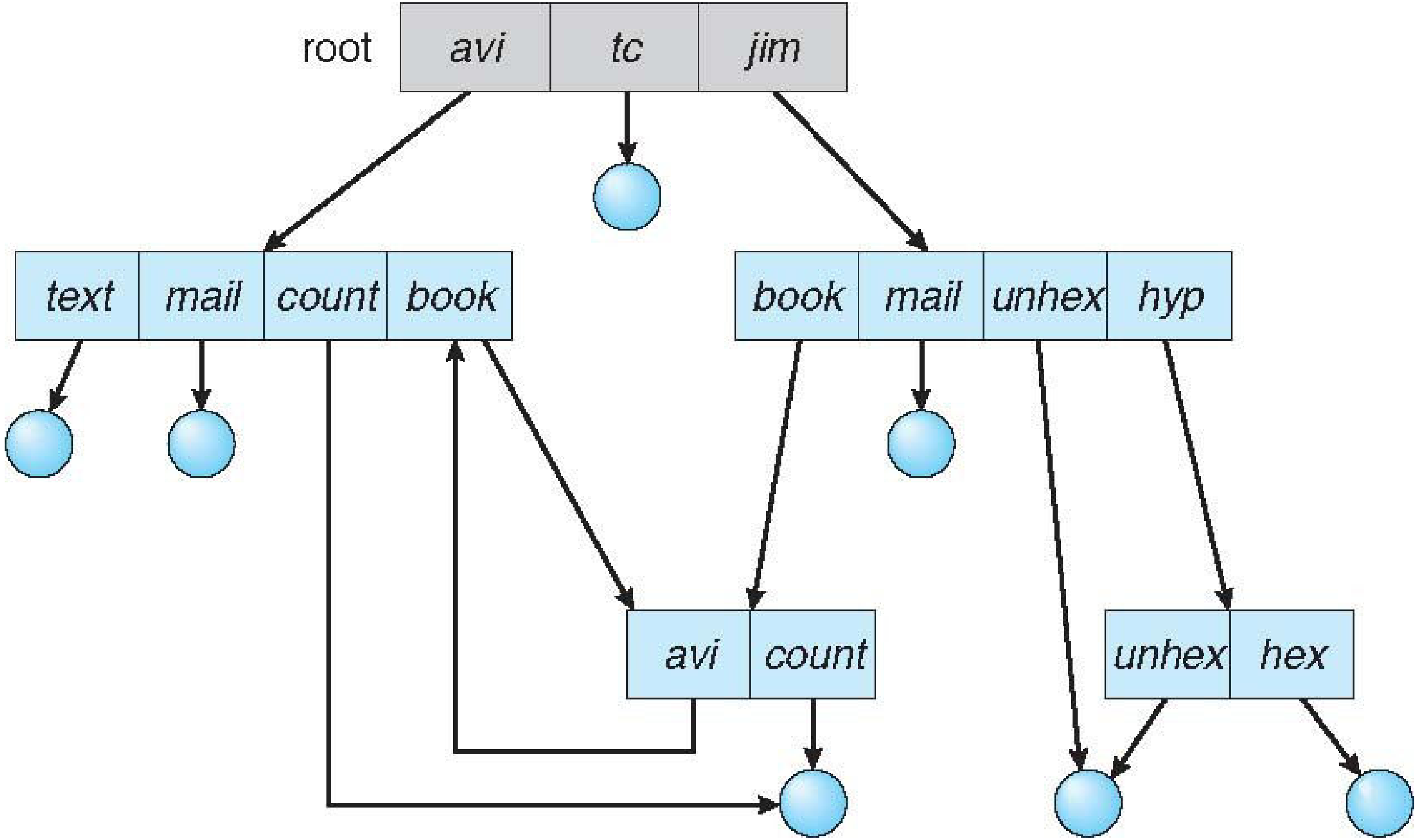
Tree Structured directories



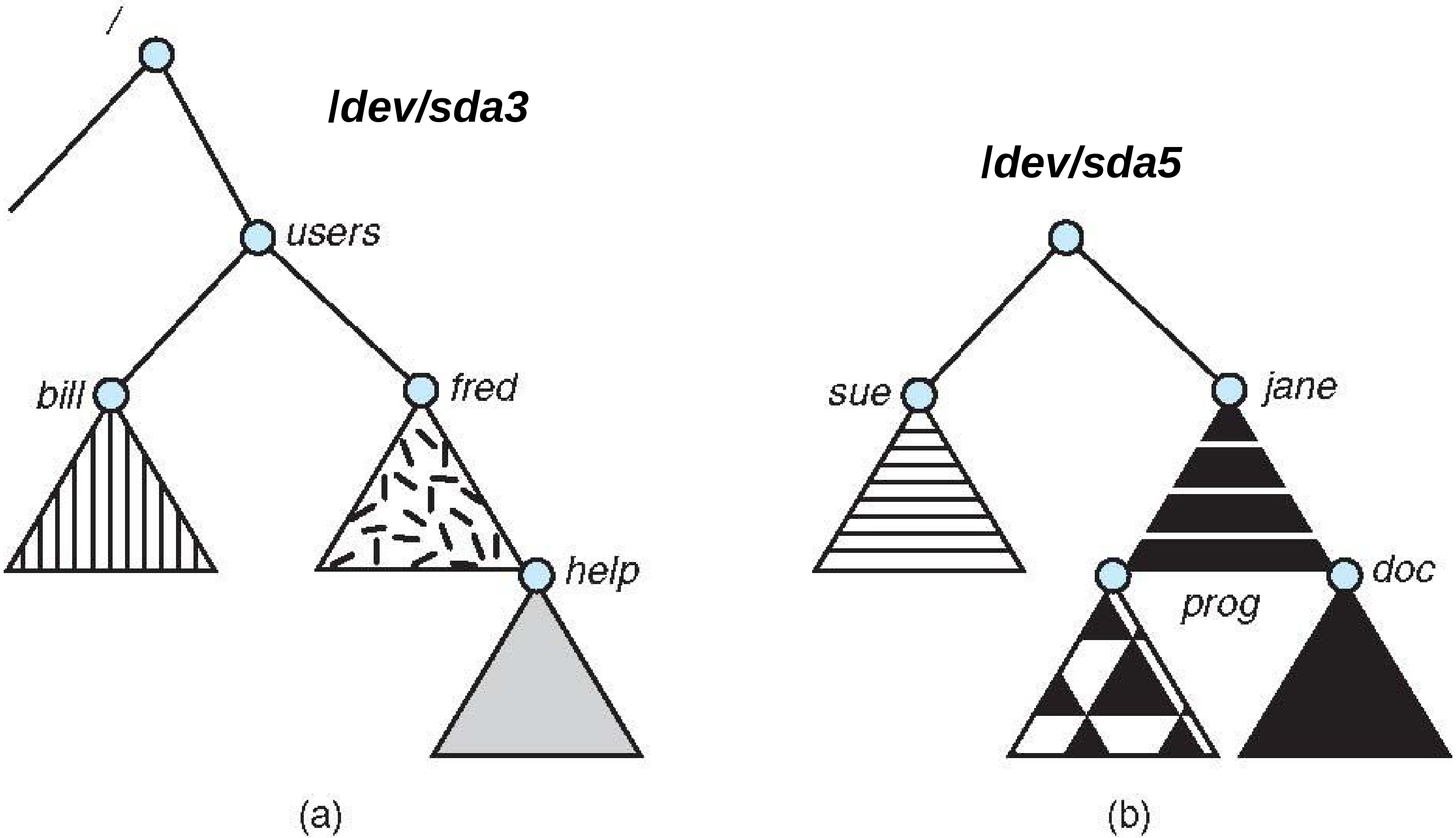
Acyclic Graph Directories



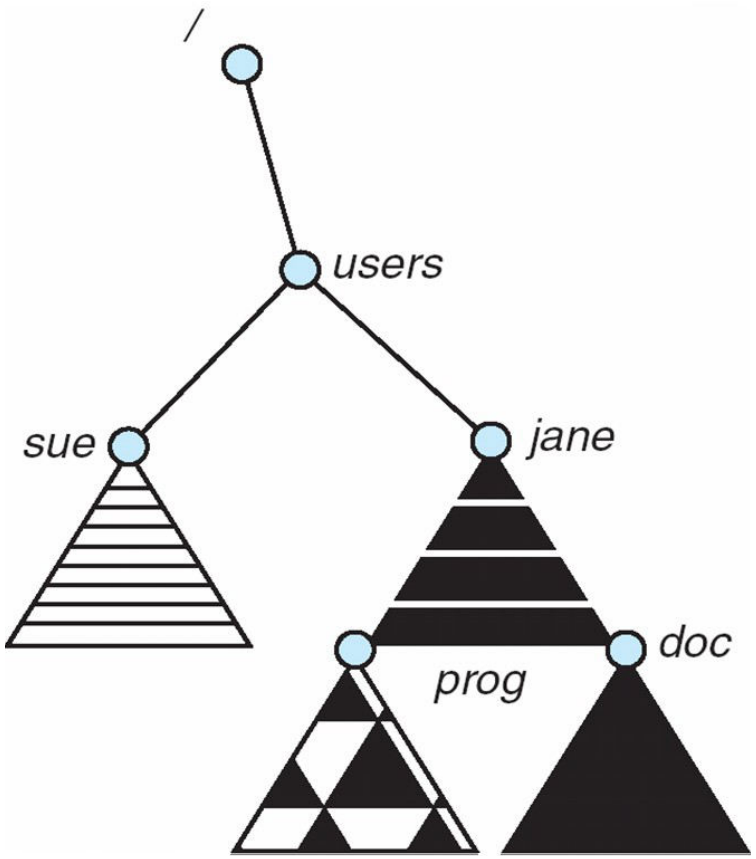
General Graph directory



Mounting of a file system: before



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

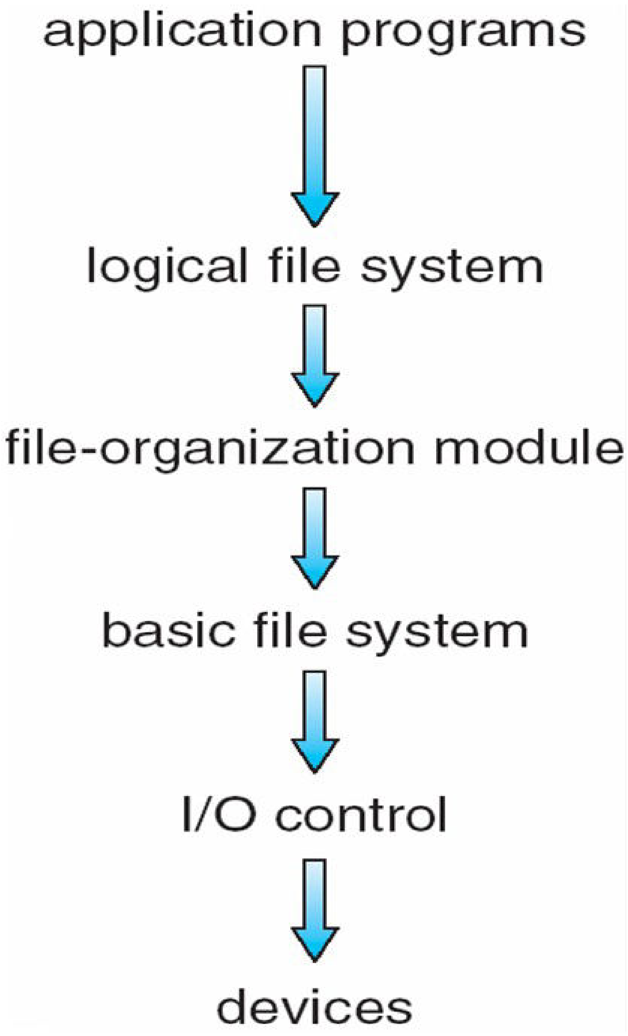
- Consistency semantics specify how multiple users are to access a shared file simultaneously
- Unix file system (UFS) implements:
  - Writes to an open file visible immediately to other users of the same open file
  - One mode of sharing file pointer to allow multiple users to read and write concurrently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

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

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

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 slightely differently, but following the layering principle!

Layering advantages

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with extended file system ext2 and ext3 leading; plus distributed file systems, etc)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE
  - The Virtual File System (to be discussed later) helps us combine multiple physically different file systems into one logical layer at the top

File system implementation: Different problems to be solved

- What to do at boot time ?
- 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
- How to mount a file system , how to unmount

File system implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures. Let's see some of the important ones.
  - Boot control block contains info needed by system to boot OS from that volume
    - Not always needed. Needed if volume contains OS, usually first block of volume
  - Volume control block (superblock, master file table) contains volume details
    - Total # of blocks, # of free blocks, block size, free block pointers or array
  - Per-file File Control Block (FCB) contains many details about the file
    - Inode(FCB) number, permissions, size, dates
  - Directory structure organizes the files
    - Names and inode numbers, master file table

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

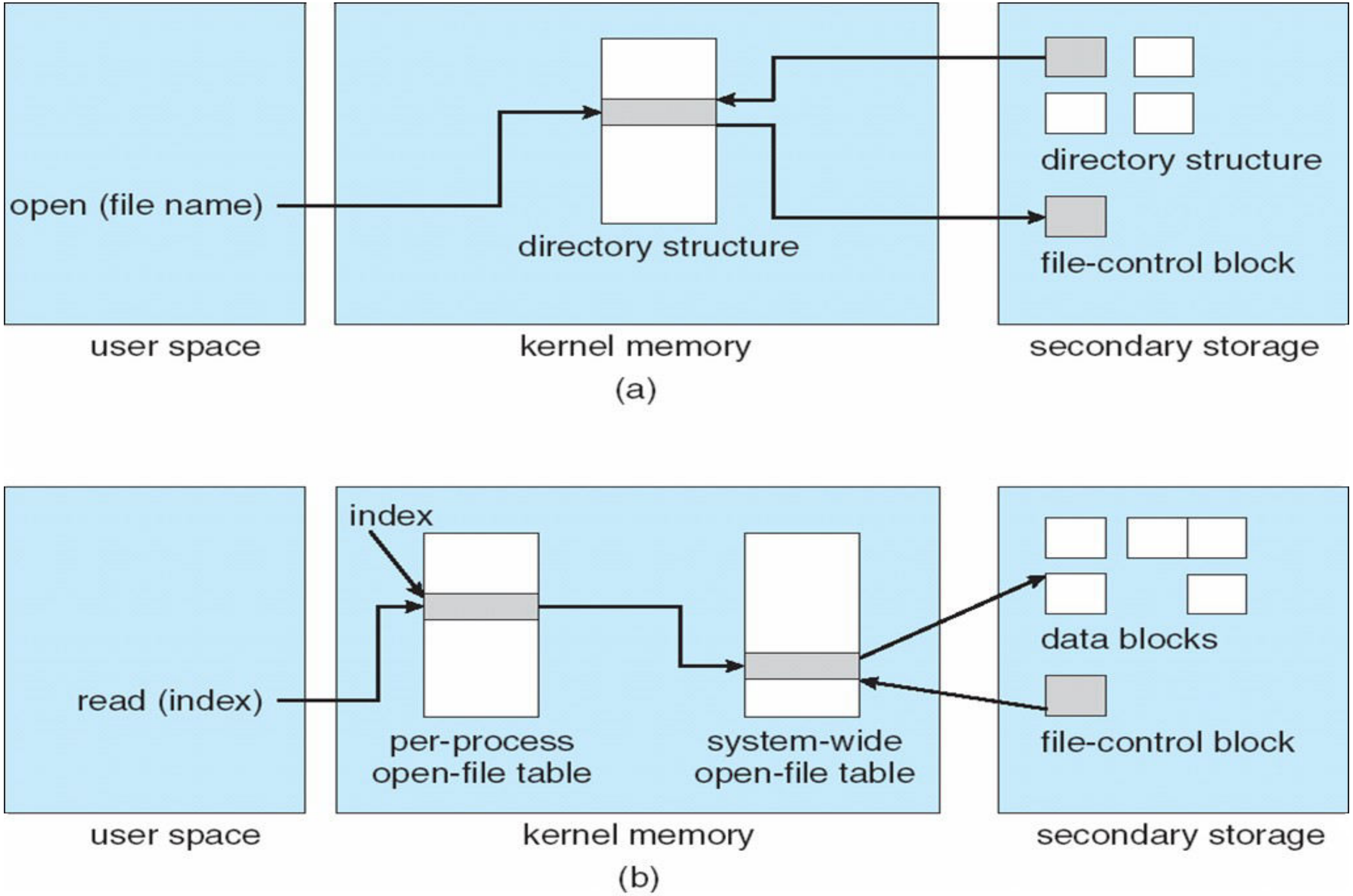
Why does it NOT contain the

Name of the file ?

In memory data structures

- Mount table
  - storing file system mounts, mount points, file system types
- See next slide for “file” realated 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

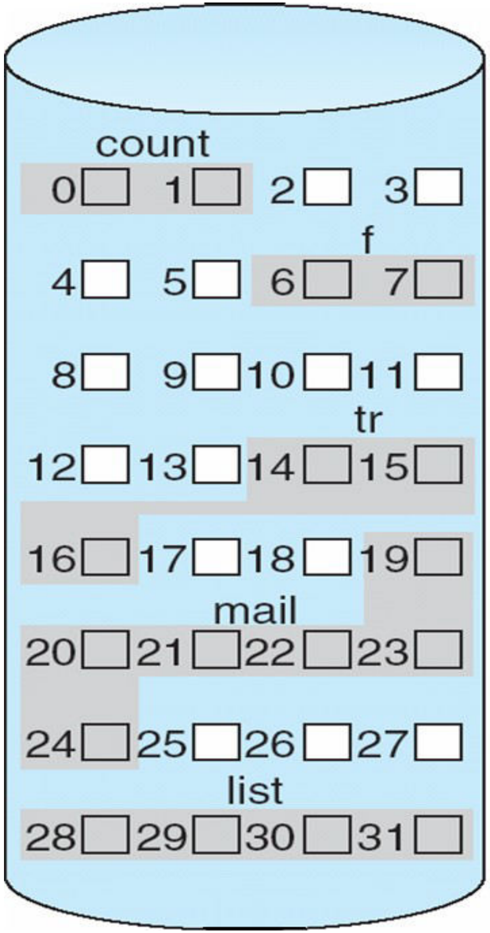
Directory Implementation

- 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



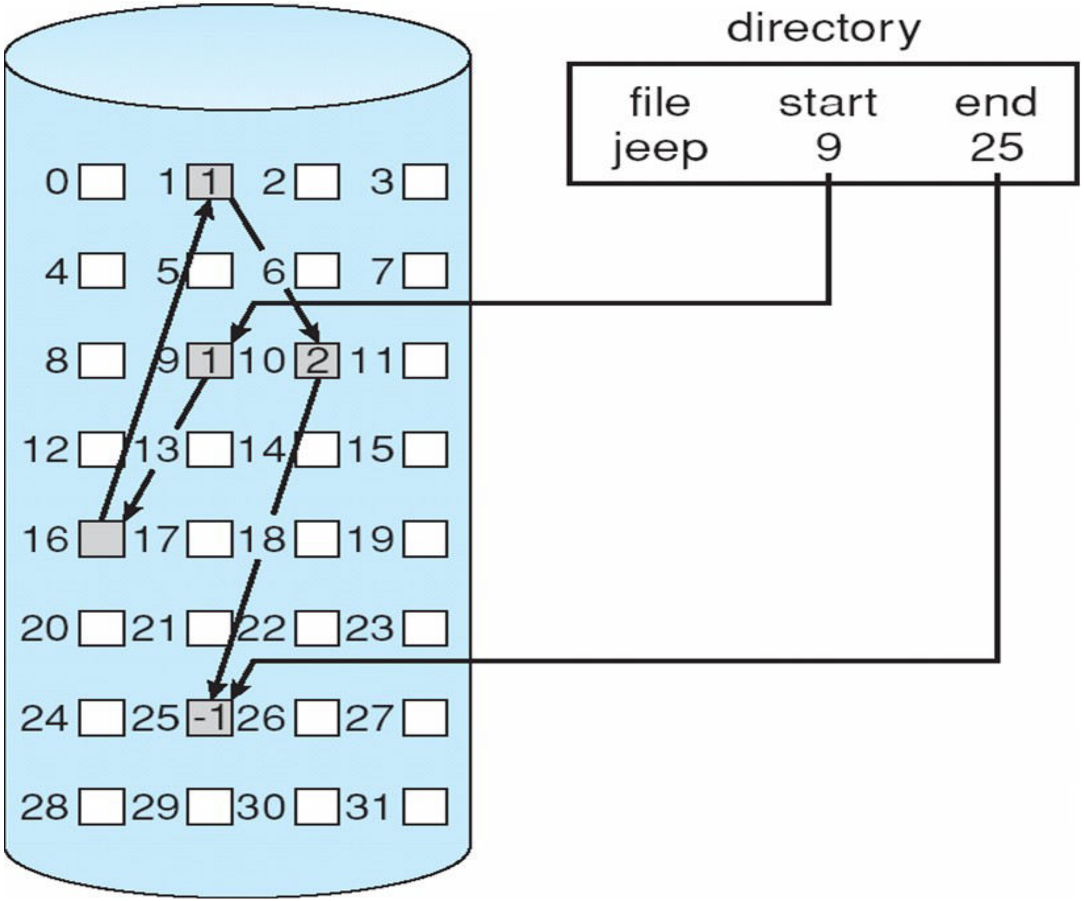
directory

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

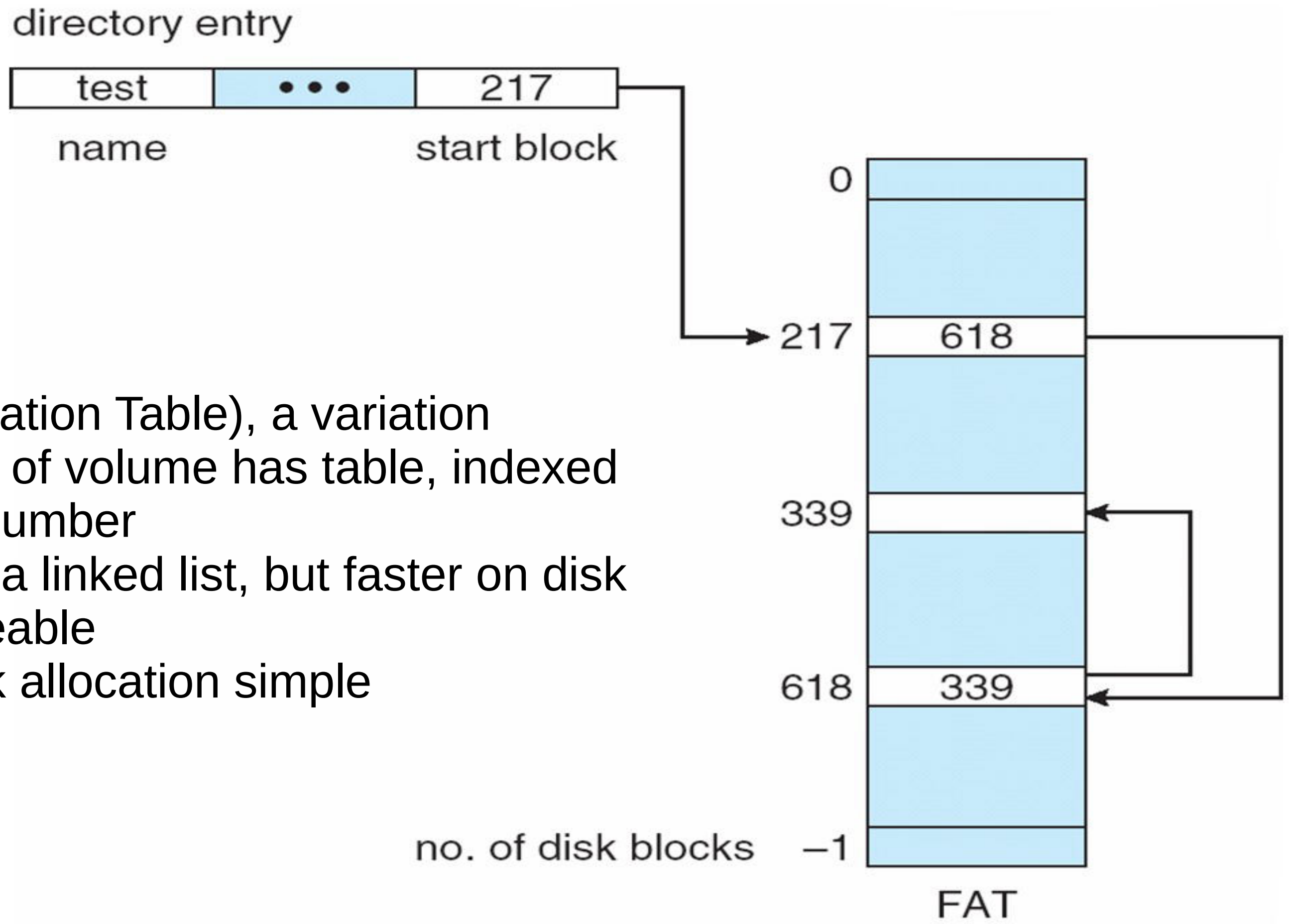


Linked allocation of blocks to a file

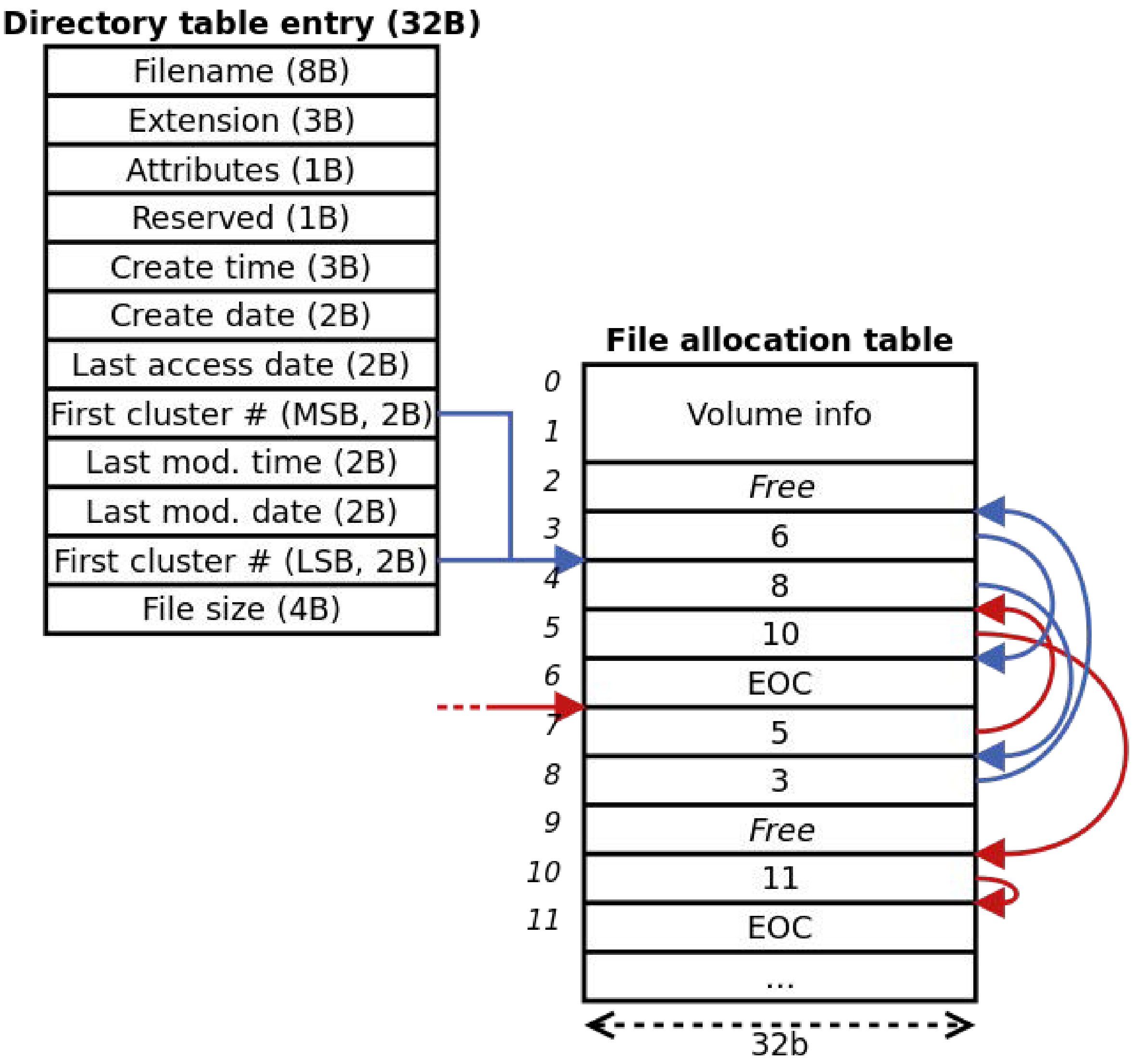
- Linked allocation
  - Each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block (i.e. data + pointer to next block)
  - No compaction, external fragmentation

- 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



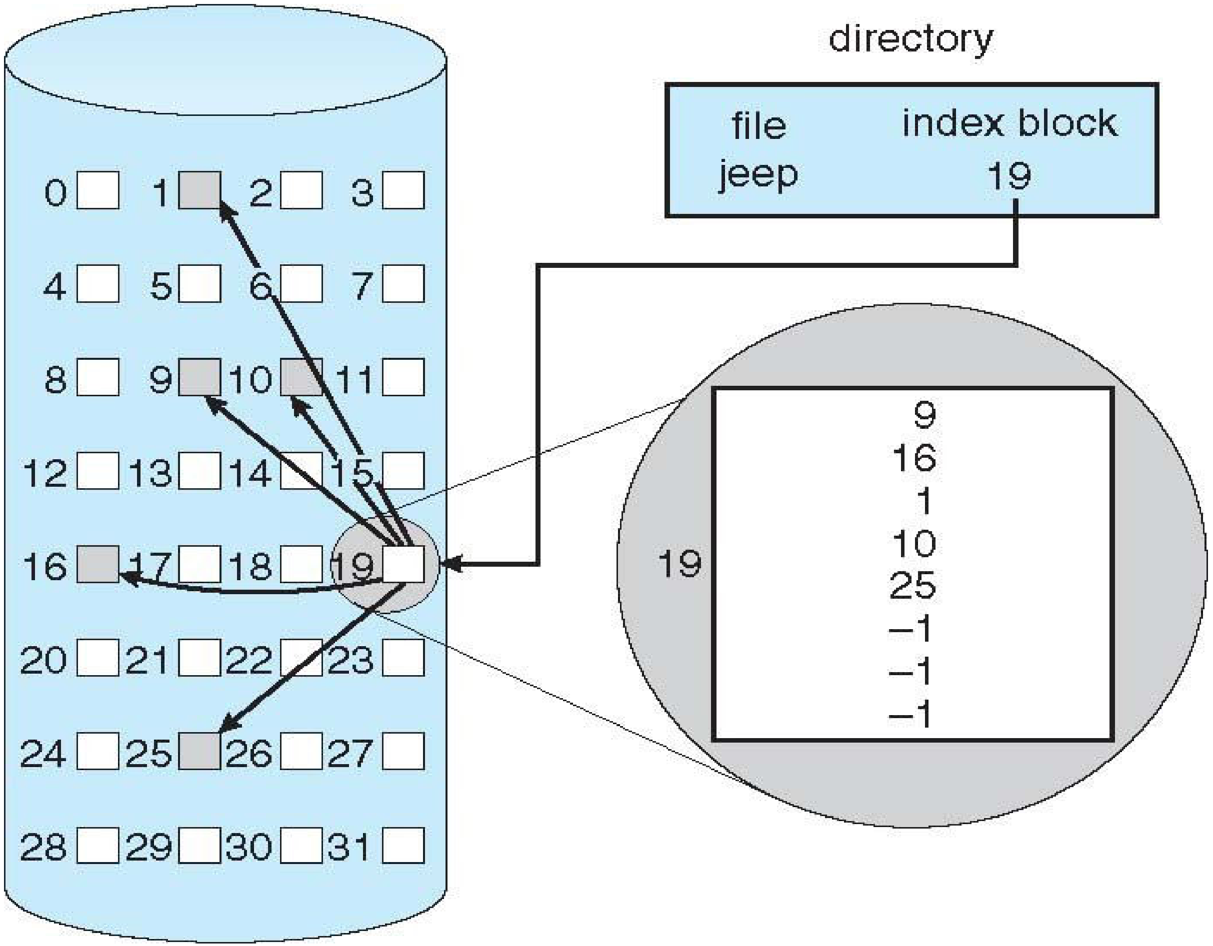
- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple



FAT: File Allocation Table

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

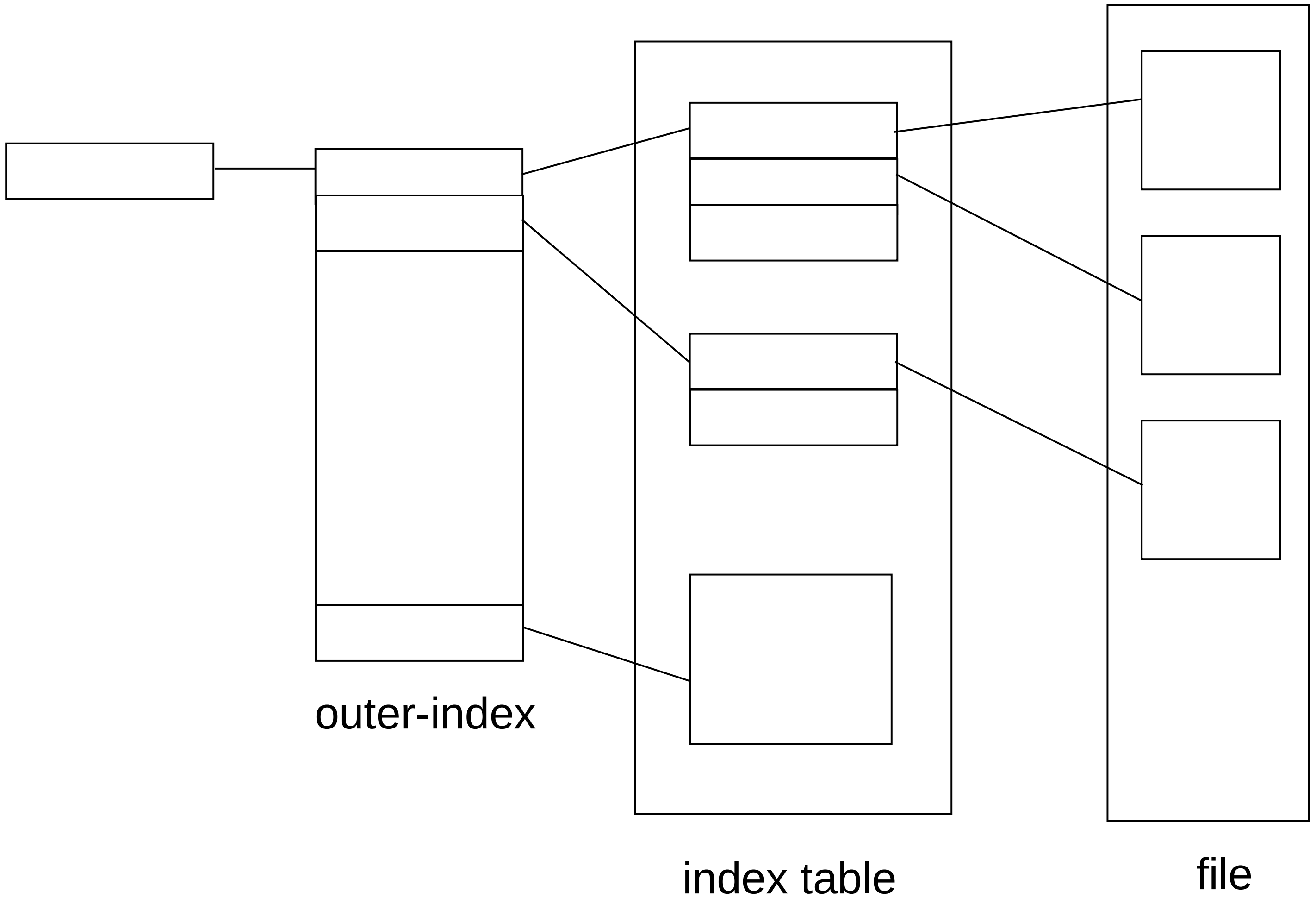
Indexed allocation



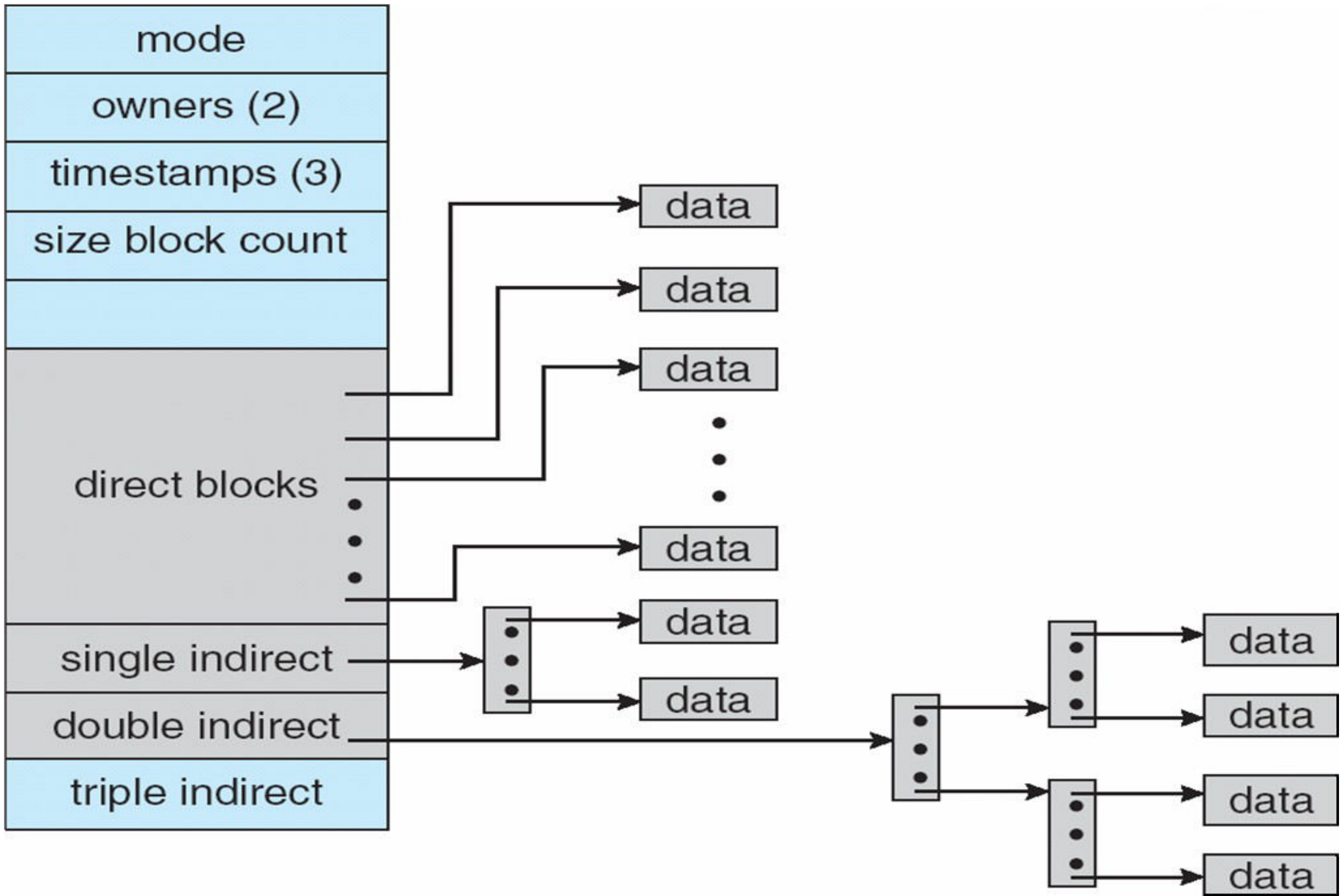
Indexed allocation

- 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



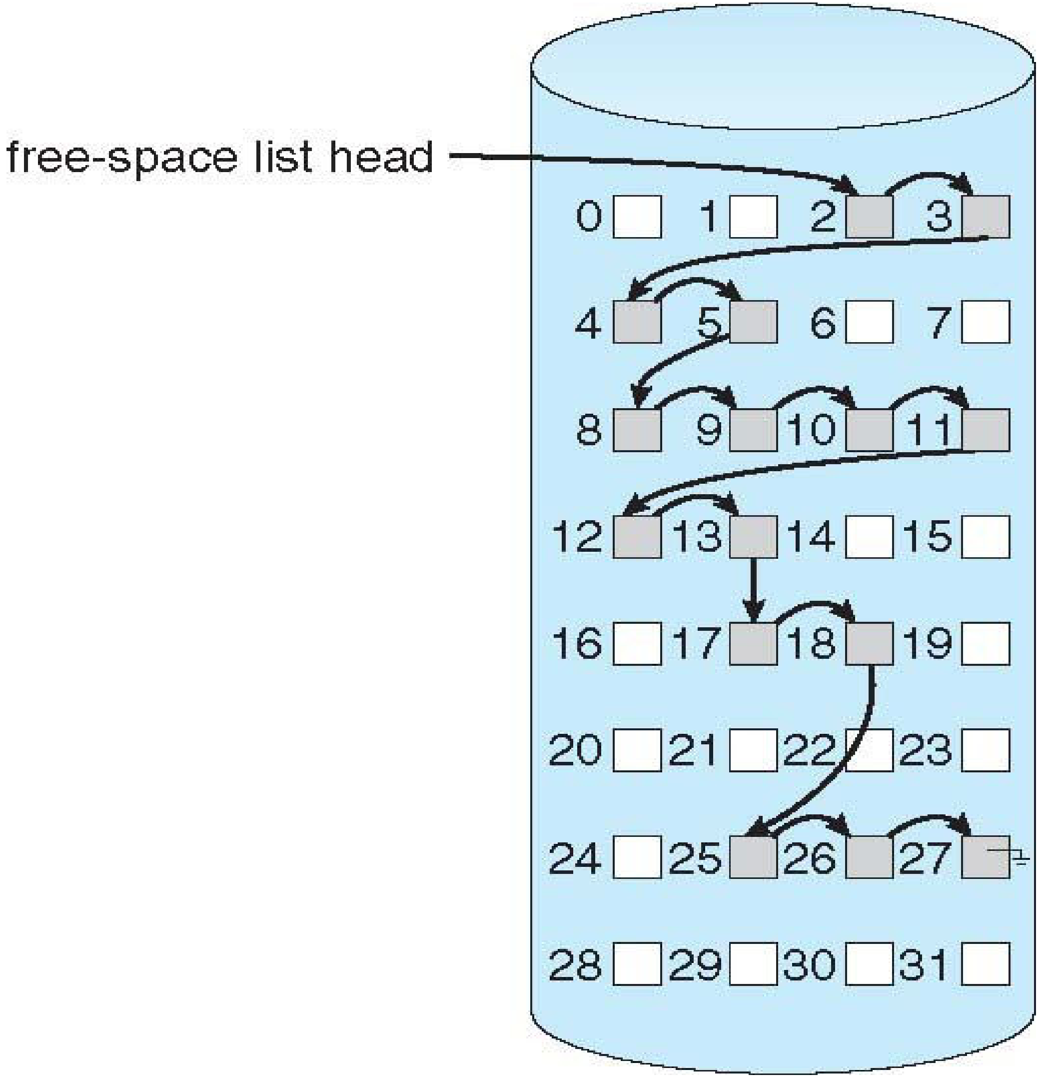
Free Space Management

- 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!)

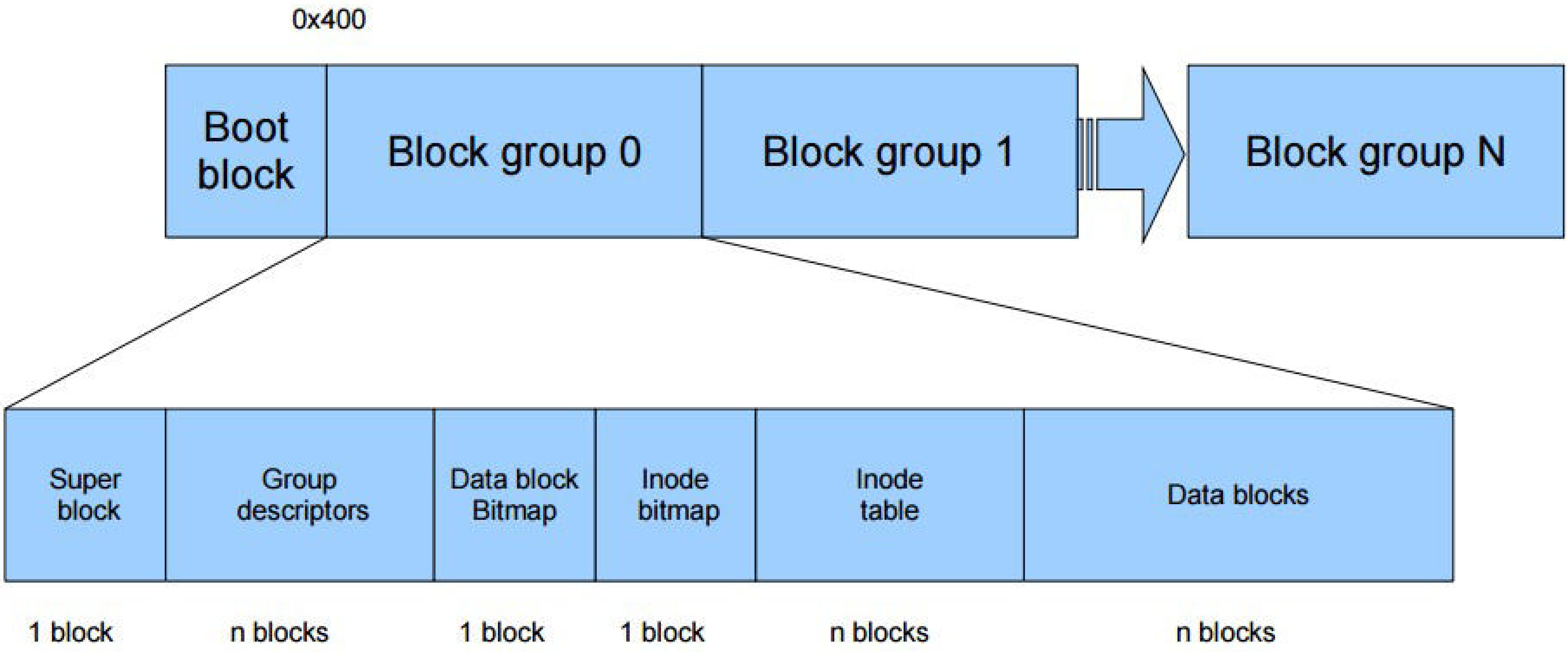


Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Ext2 FS layout

Ext2 FS Layout



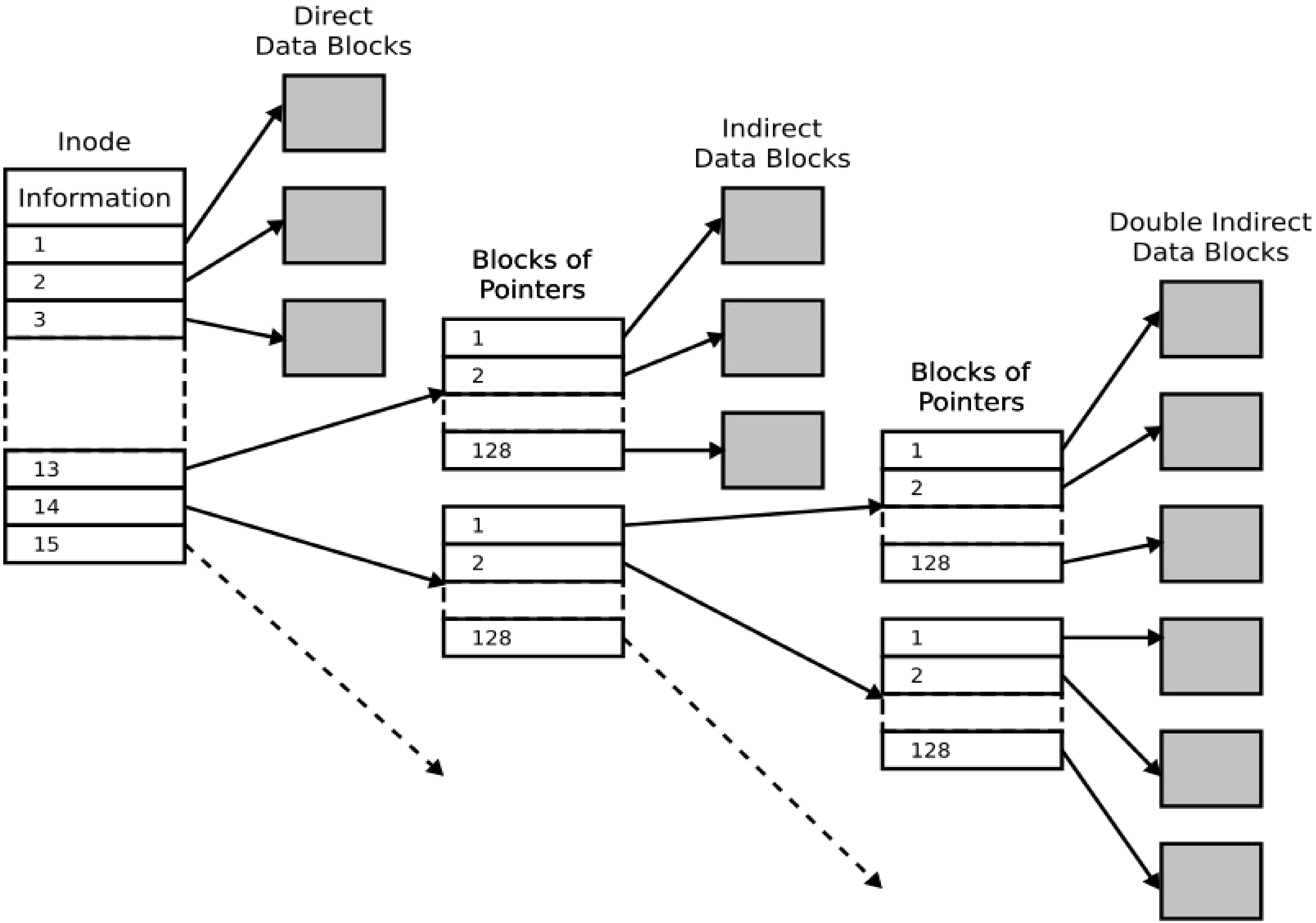
```
struct ext2_super_block {
    __le32  s_inodes_count;     /* Inodes count */
    __le32  s_blocks_count;     /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes 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];
};
```

```
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: Directory entry

	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

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

Efficiency and Performance
(and the risks created
while trying to achieve it!)

Efficiency

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file’s directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
  -

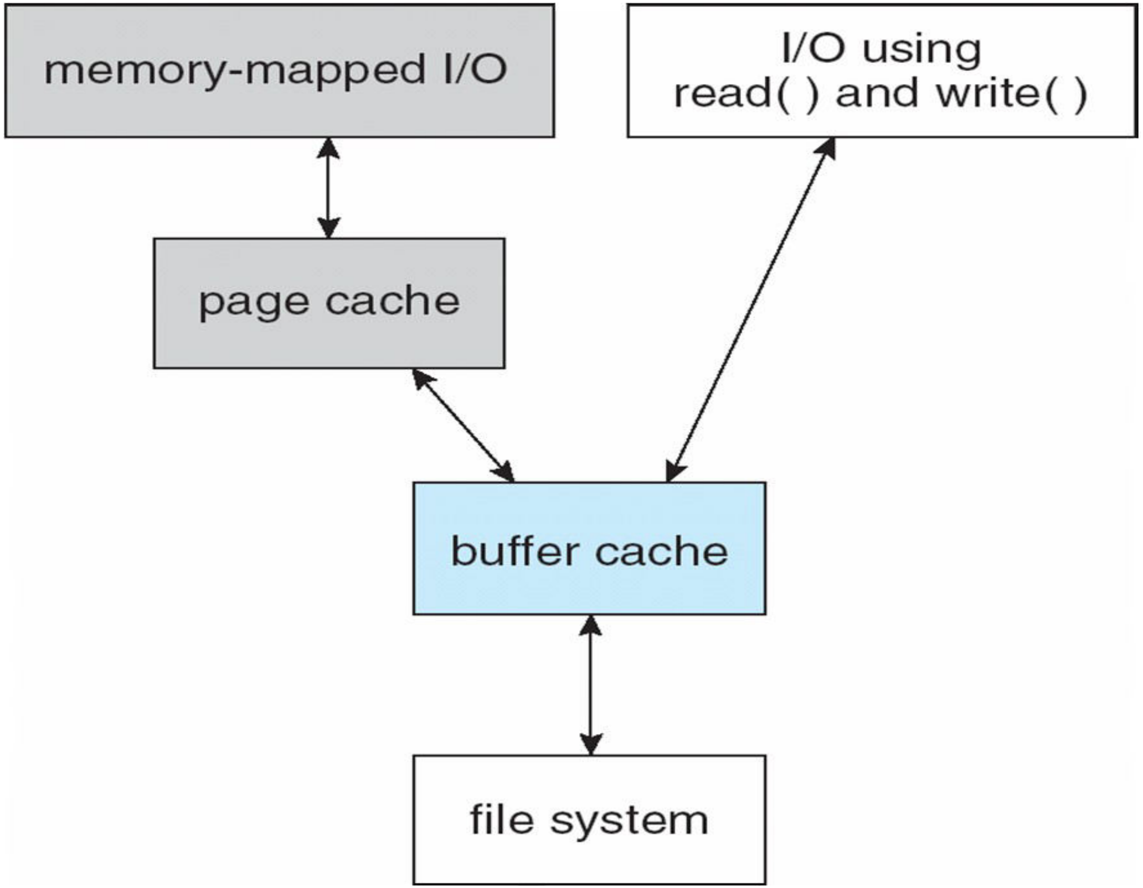
Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

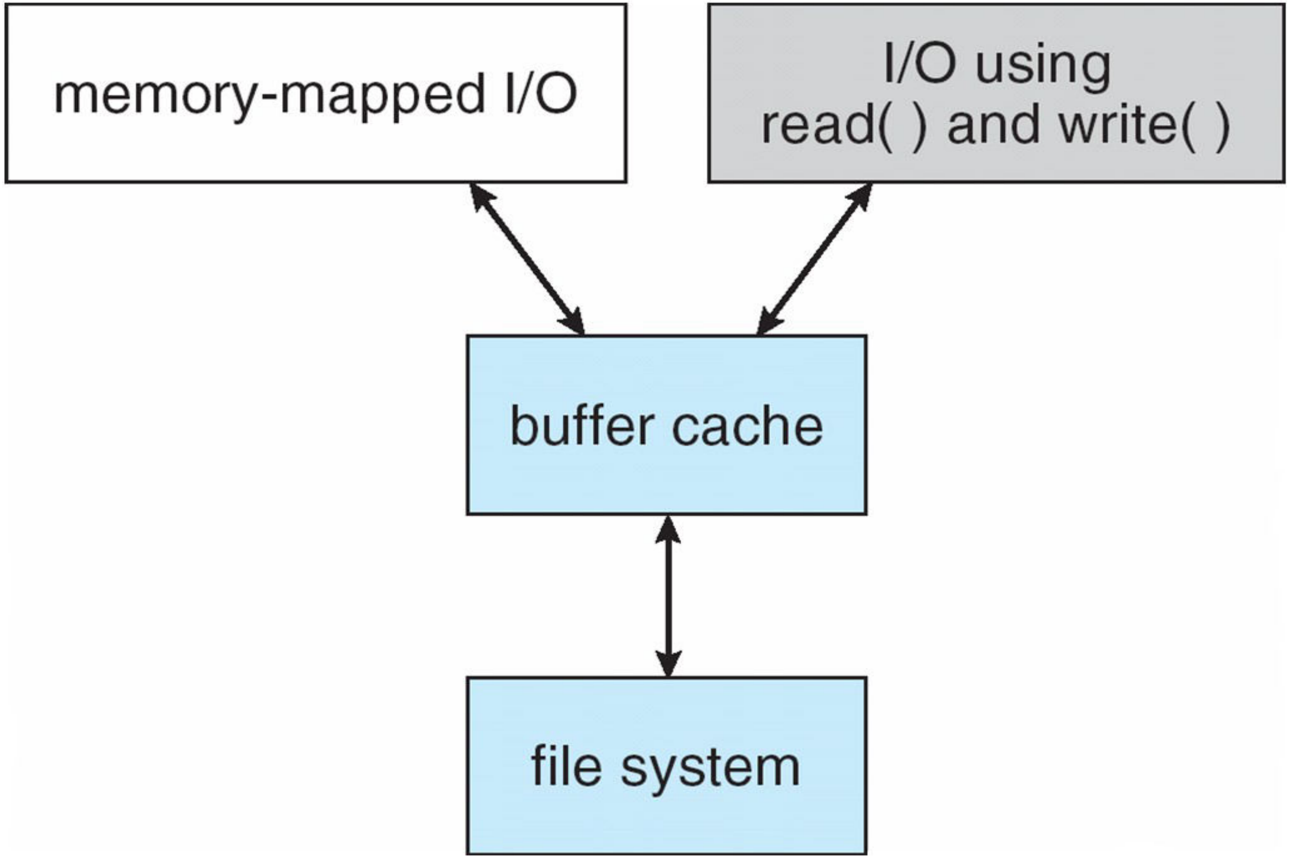
I/O Without a Unified Buffer Cache



Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- Problem. Consider creating a file on ext2 file system.
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
- Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

File Systems

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

System calls related to files/file-system

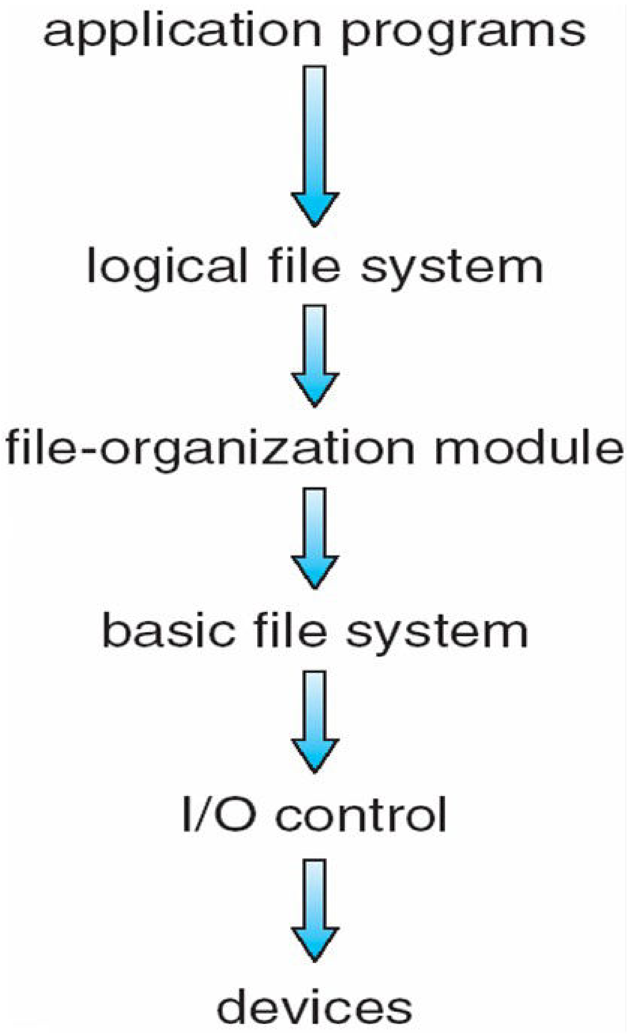
- Open(2), chmod(2), chown(2), close(2), dup(2), fcntl(2), link(2), lseek(2), mknod(2), mmap(2), mount(2), read(2), stat(2), umask(2), unlink(2), write(2), fstat(2), access(2), readlink(2), ...

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

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 slightlety differently, but following the layering principle!

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

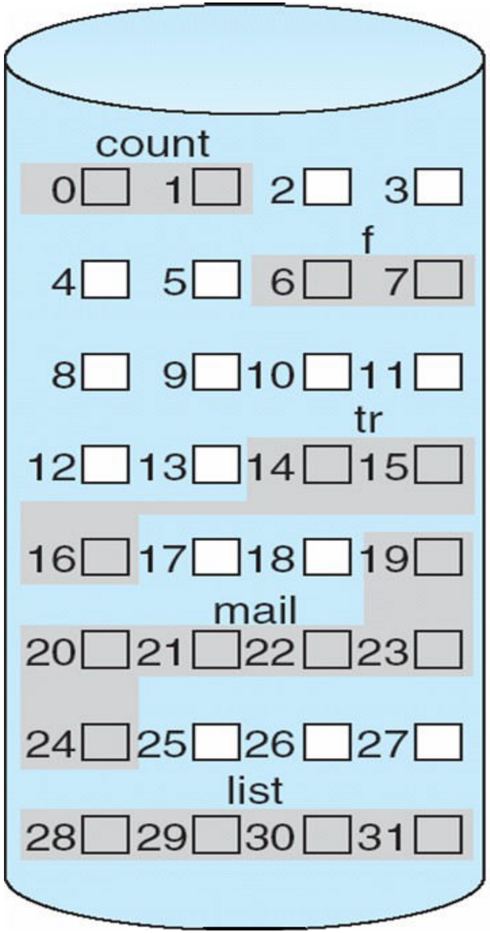
Why does it NOT contain the

Name of the file ?

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



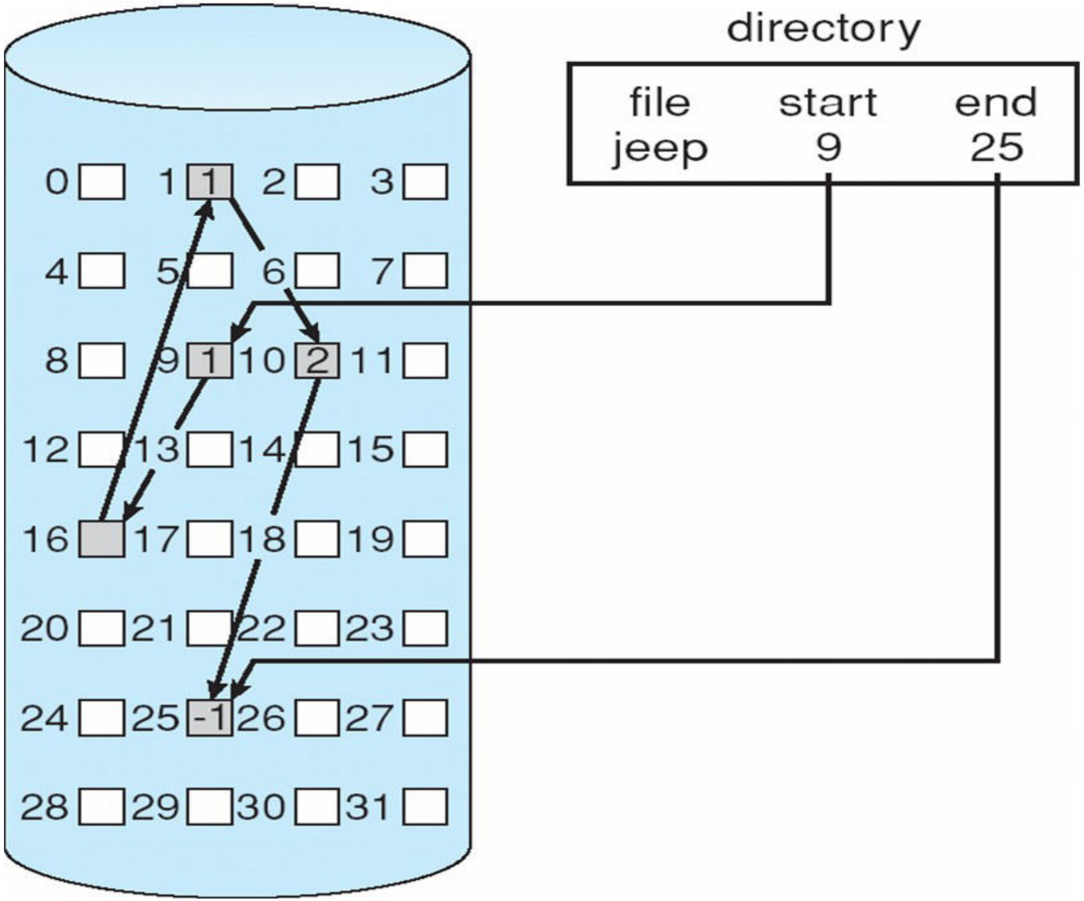
directory

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

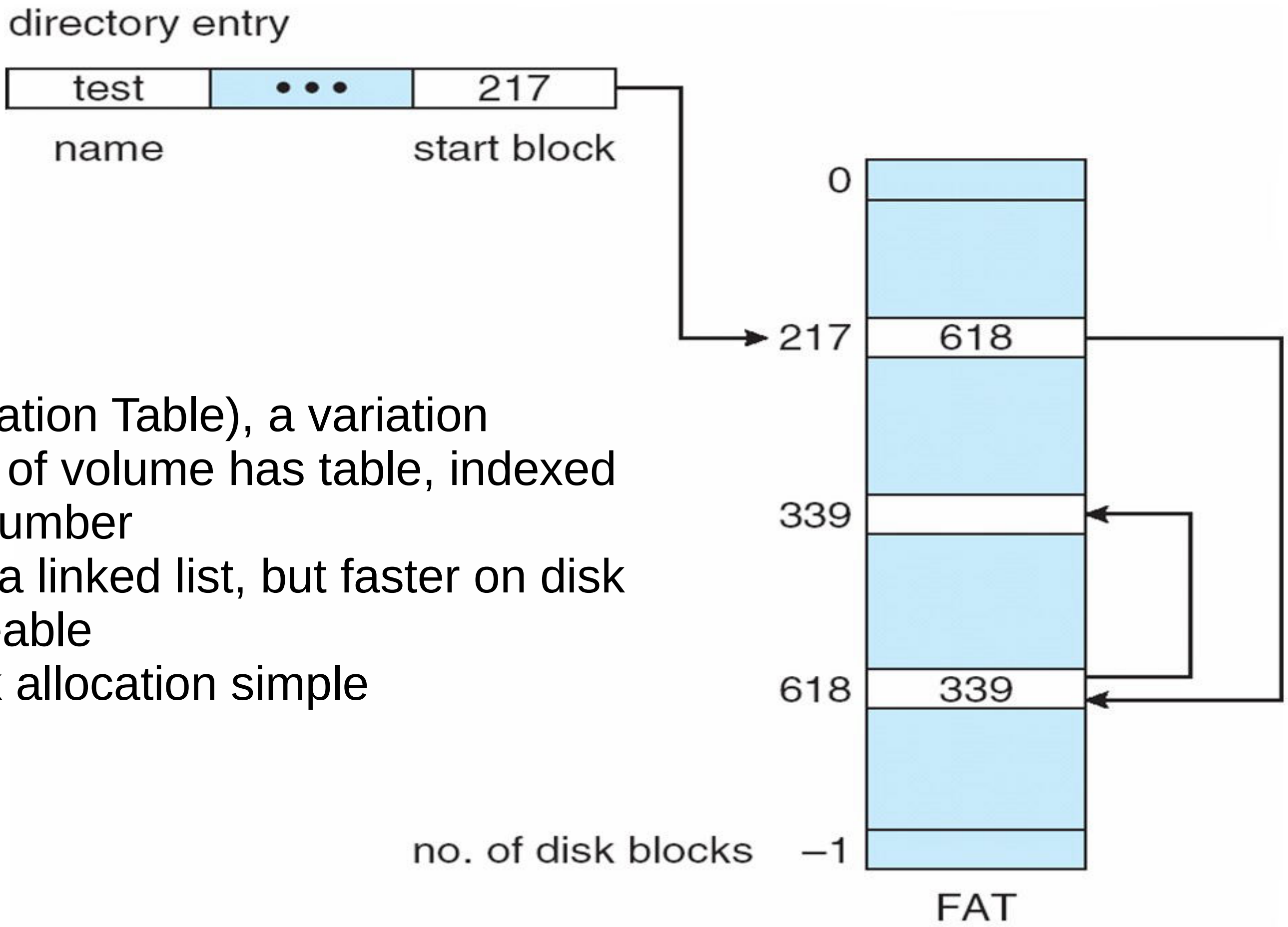
Linked allocation of blocks to a file



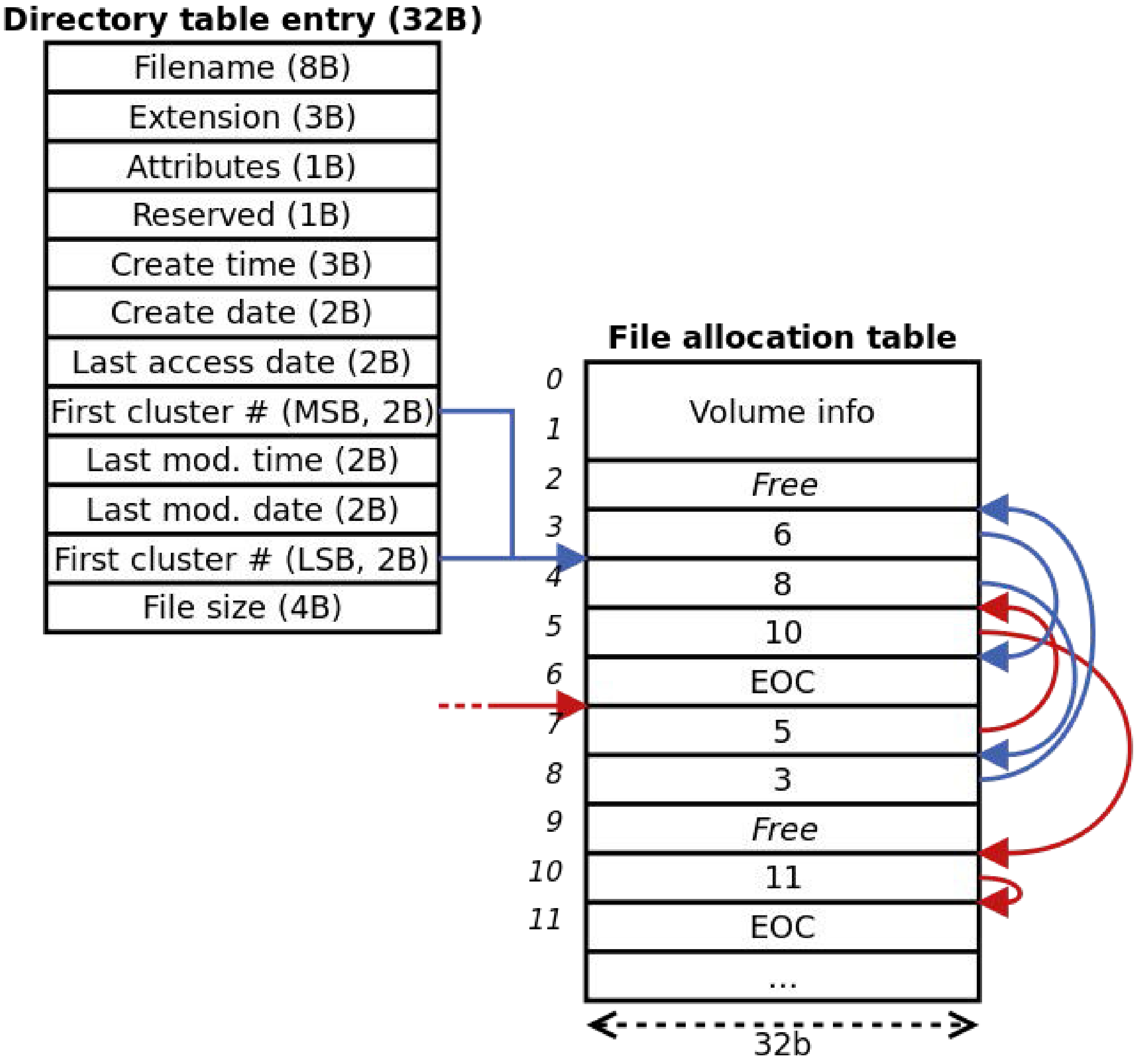
Linked allocation of blocks to a file

- 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



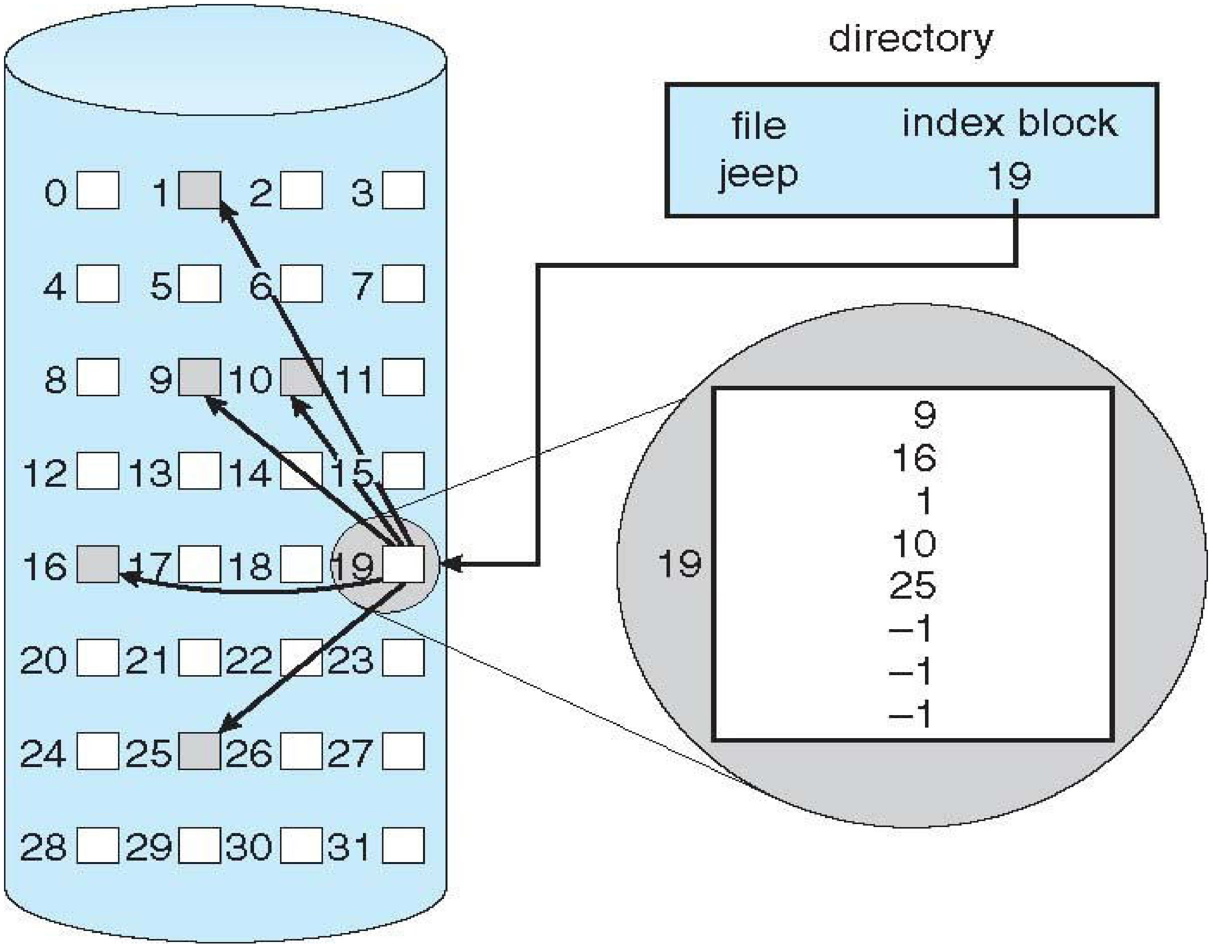
- FAT (File Allocation Table), a variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple



FAT: File Allocation Table

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

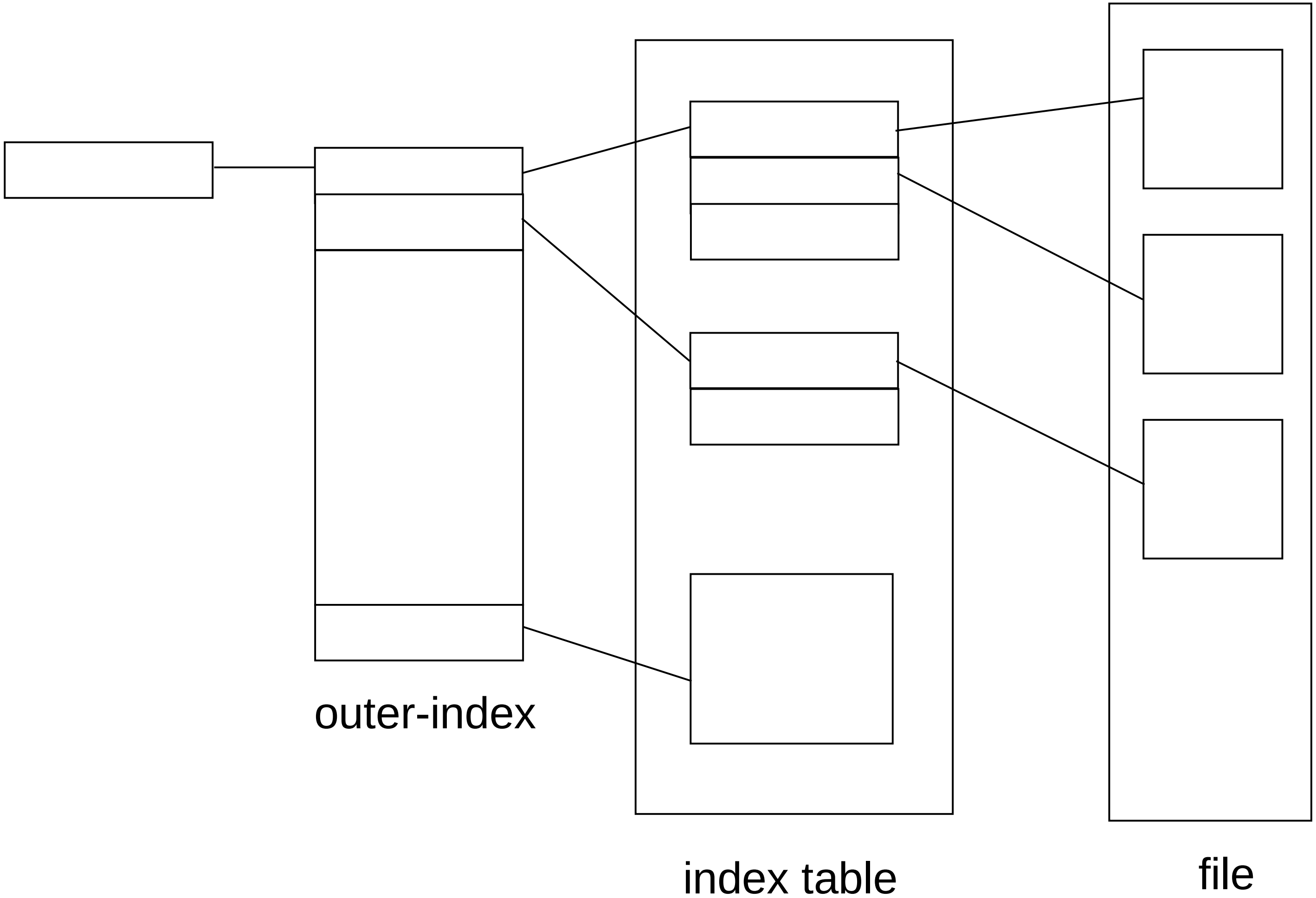
Indexed allocation



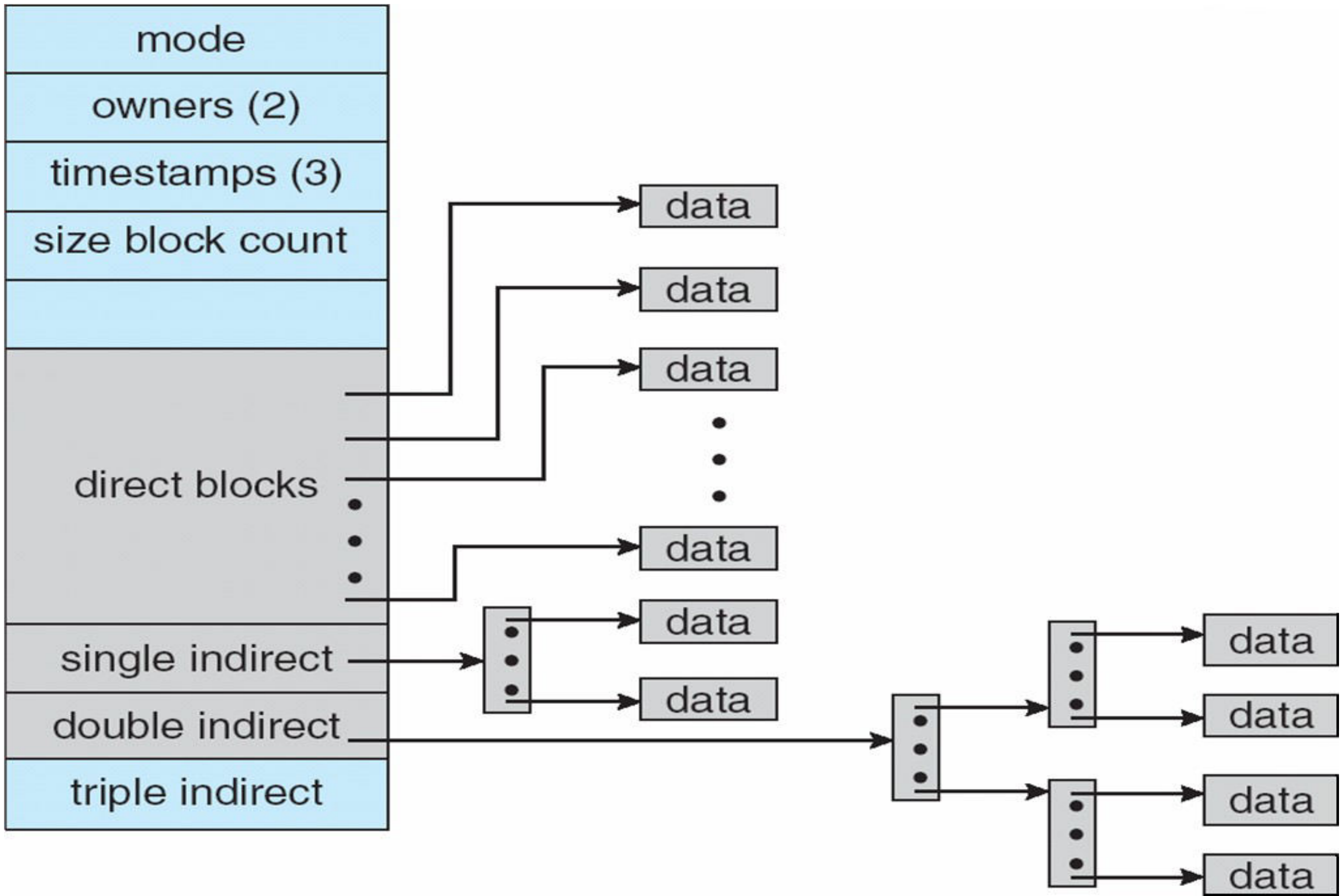
Indexed allocation

- 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



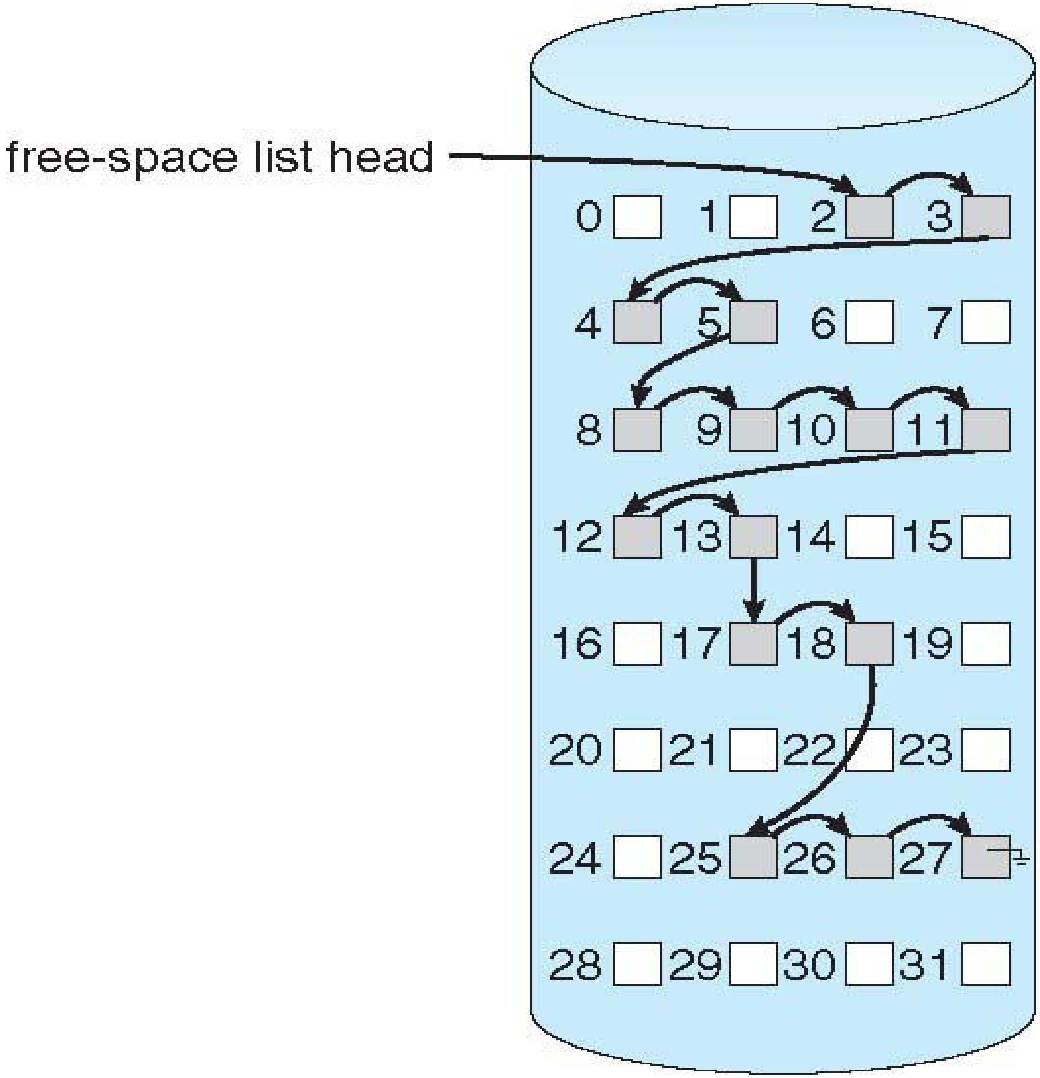
Free Space Management

- **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!)



Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Directory Implementation

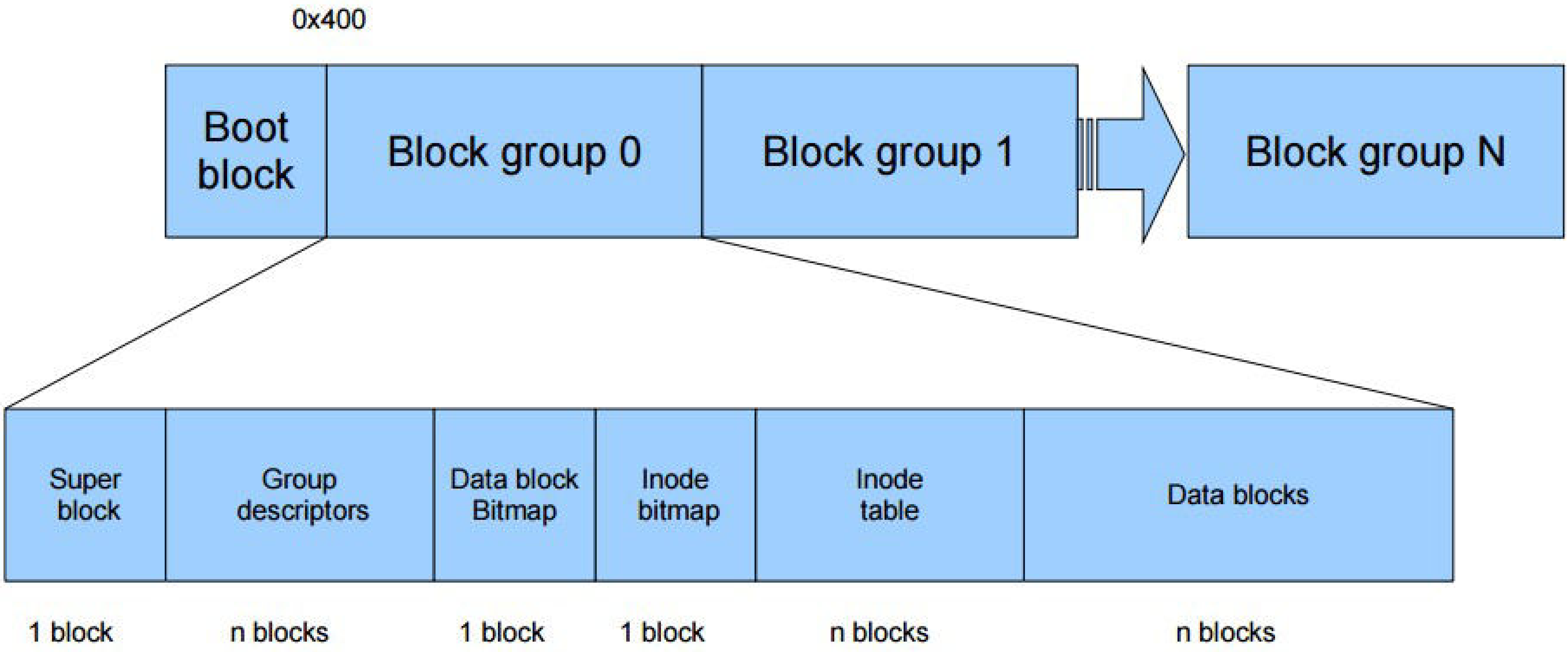
- **Problem**
  - **Directory contains files and/or subdirectories**
  - **Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.**
  - **Directory needs to give location of each file on disk**

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

Ext2 FS layout

Ext2 FS Layout



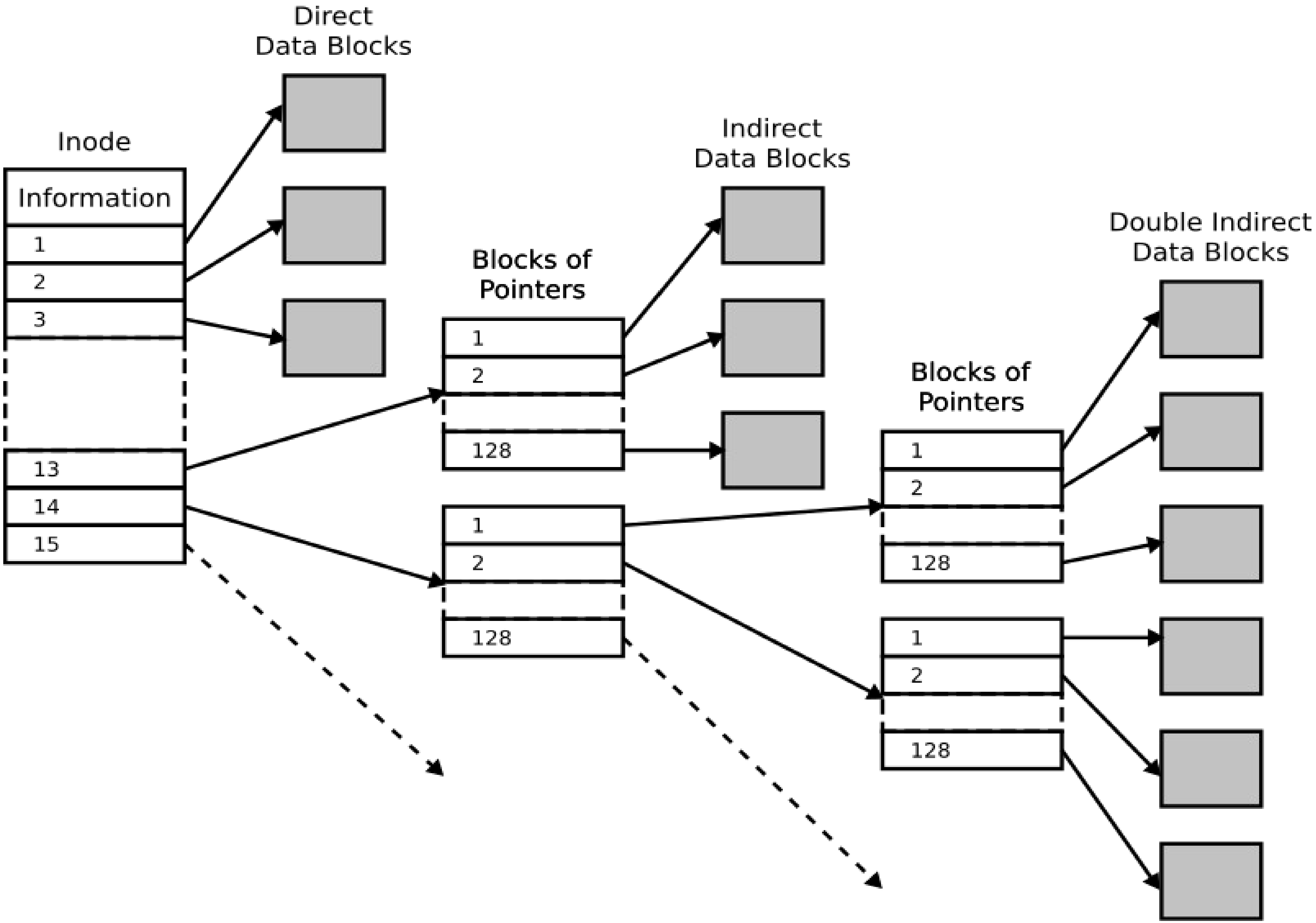
```
struct ext2_super_block {
    __le32  s_inodes_count;     /* Inodes count */
    __le32  s_blocks_count;     /* Blocks count */
    __le32  s_r_blocks_count;   /* Reserved blocks count */
    __le32  s_free_blocks_count;    /* Free blocks count */
    __le32  s_free_inodes_count;    /* Free inodes 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];
};
```

```
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 */
```

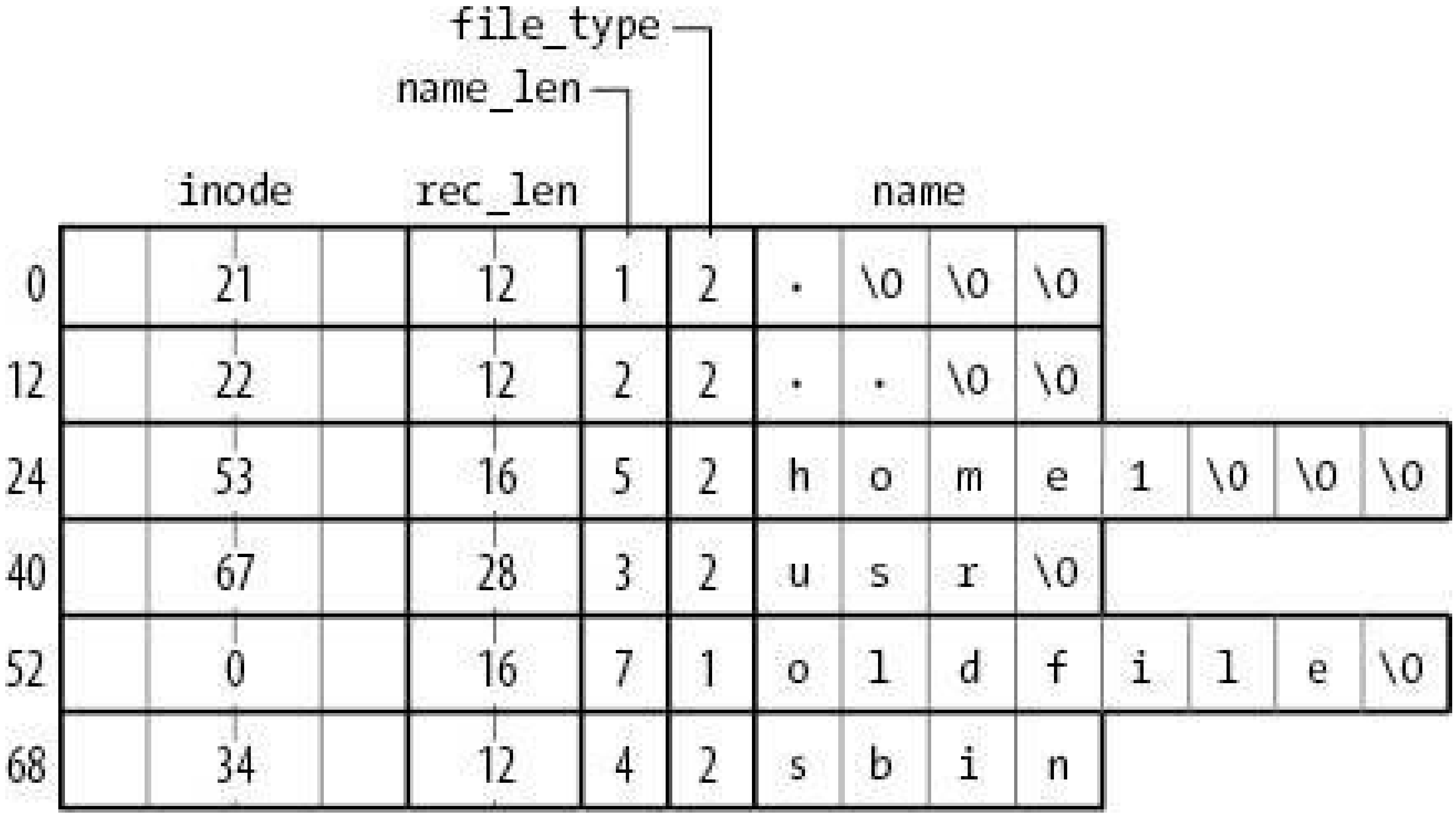


Inode in ext2

```
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: Directory entry



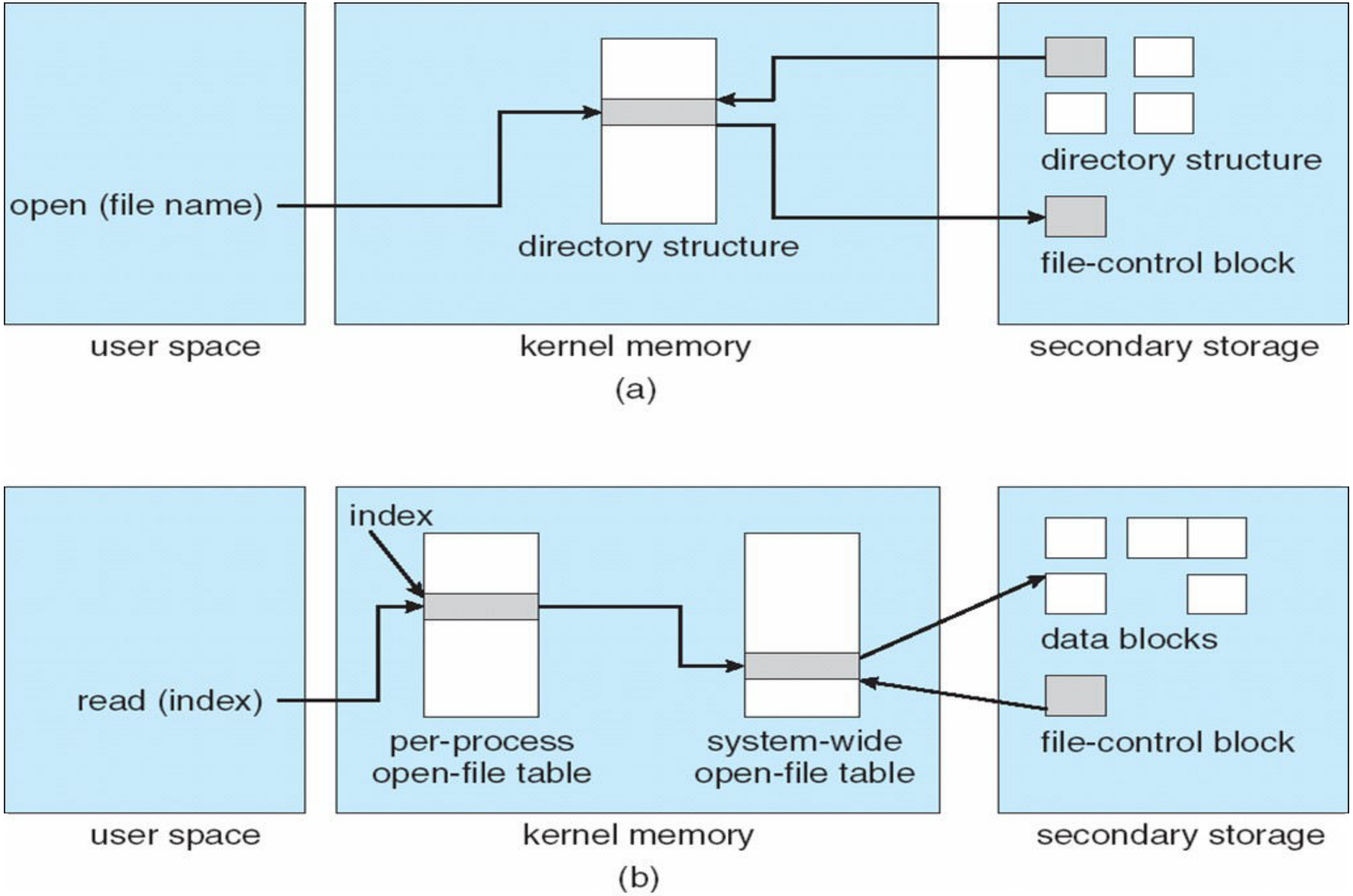
Let’s see a program to read superblock of an ext2 file system.

Efficiency and Performance (and the risks created while trying to achieve it!)

In memory data structures

- Mount table
  - storing file system mounts, mount points, file system types
- See next slide for “file” realated 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

Efficiency

- **Efficiency dependent on:**
  - **Disk allocation and directory algorithms**
  - **Types of data kept in file’s directory entry**
  - **Pre-allocation or as-needed allocation of metadata structures**
  - **Fixed-size or varying-size data structures**
  -

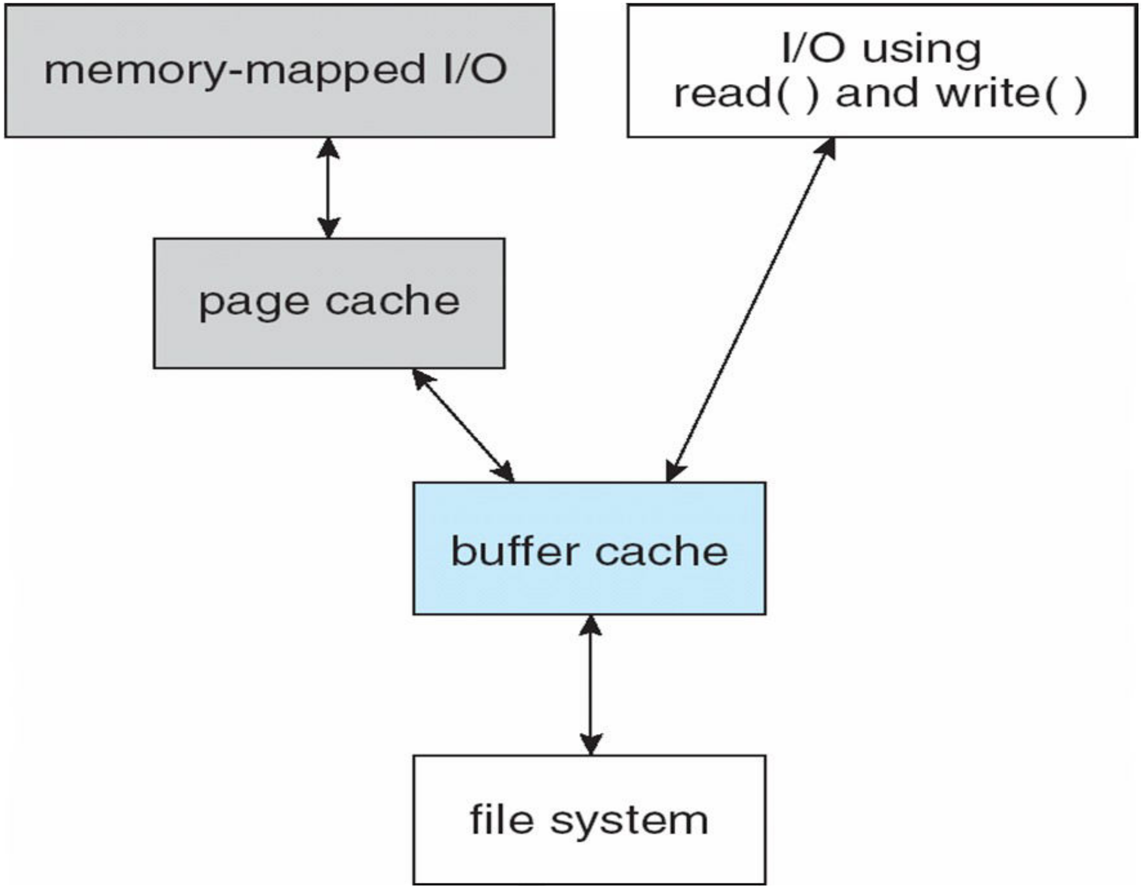
Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

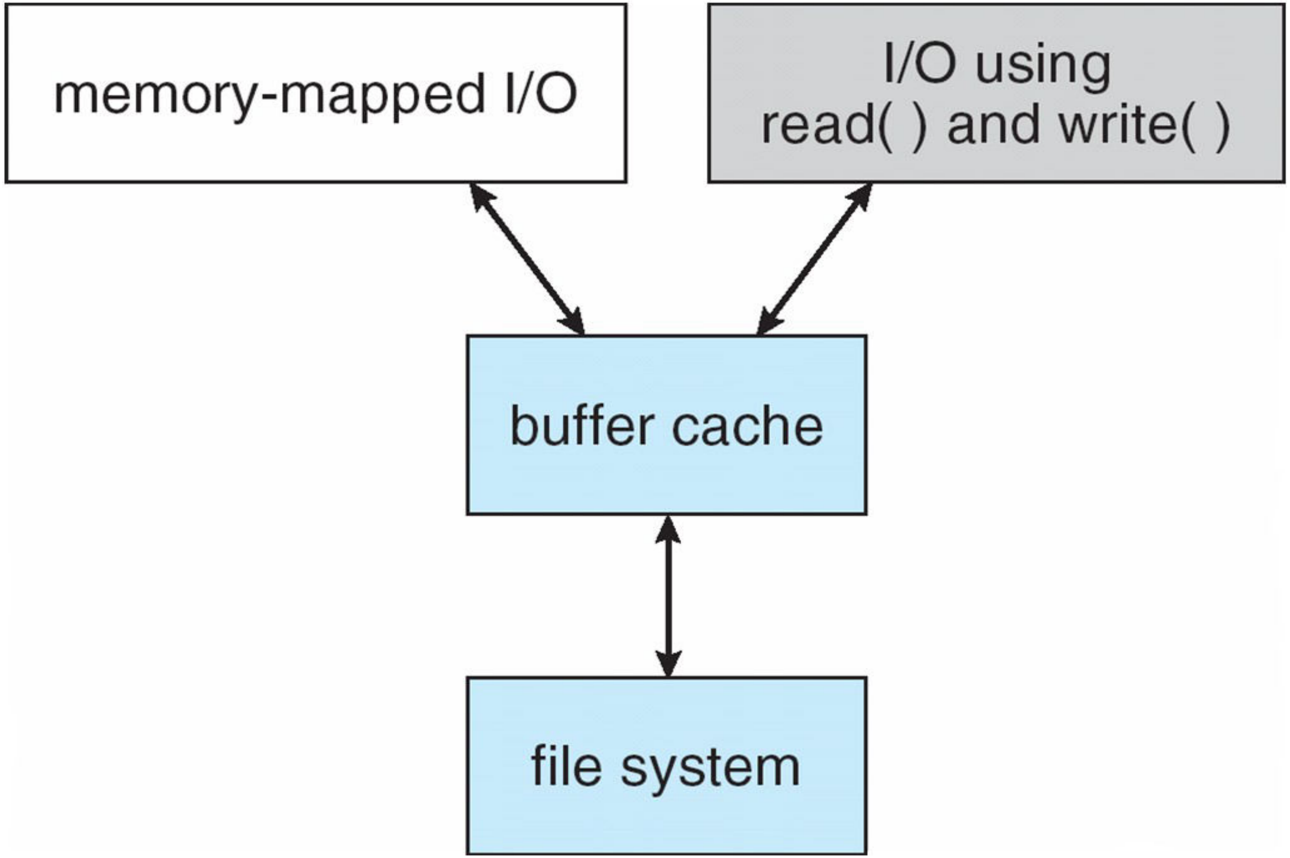
I/O Without a Unified Buffer Cache



Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- Problem. Consider creating a file on ext2 file system.
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
- Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

File System Code

open,read, write, close, pipe, fstat, chdir, dup, mknod, link, unlink, mkdir,

Files, Inodes, Buffers

What we already know

- **File system related system calls**
  - **deal with ‘fd’ arrays (ofile in xv6). open() returns first empty index. open should ideallly locate the inode on disk and initialize some data structures**
  - **maintain ‘offsets’ within a ‘file’ to support sequential read/write**
  - **dup() like system calls duplicate pointers in fd-array**
  - **read/write like system calls, going through ‘ofile’ array, should locate data of file on disk**
  - **We need functions to read/write from disk – that is disk driver**
  - **cache data of files in OS data structures for performance : buffering**
  - **Need to handle on disk data structures as well**
- **Faster recovery (like journaling in ext3) is desired**

xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartamentalized

Layers of xv6 file system code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,

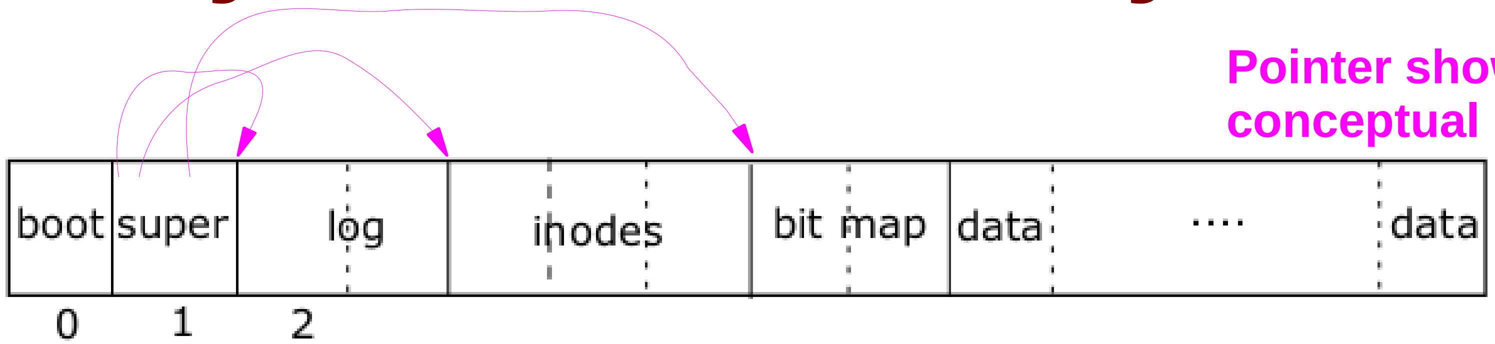
Layer	Code
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

Layout of xv6 file system

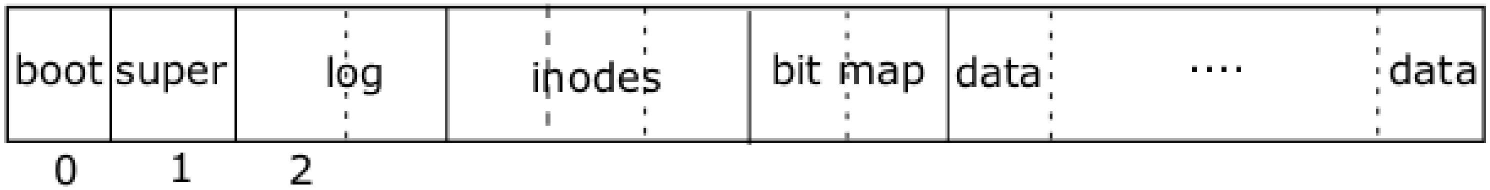
Pointer shown are conceptual



May see the code of mkfs.c to get insight into the layout

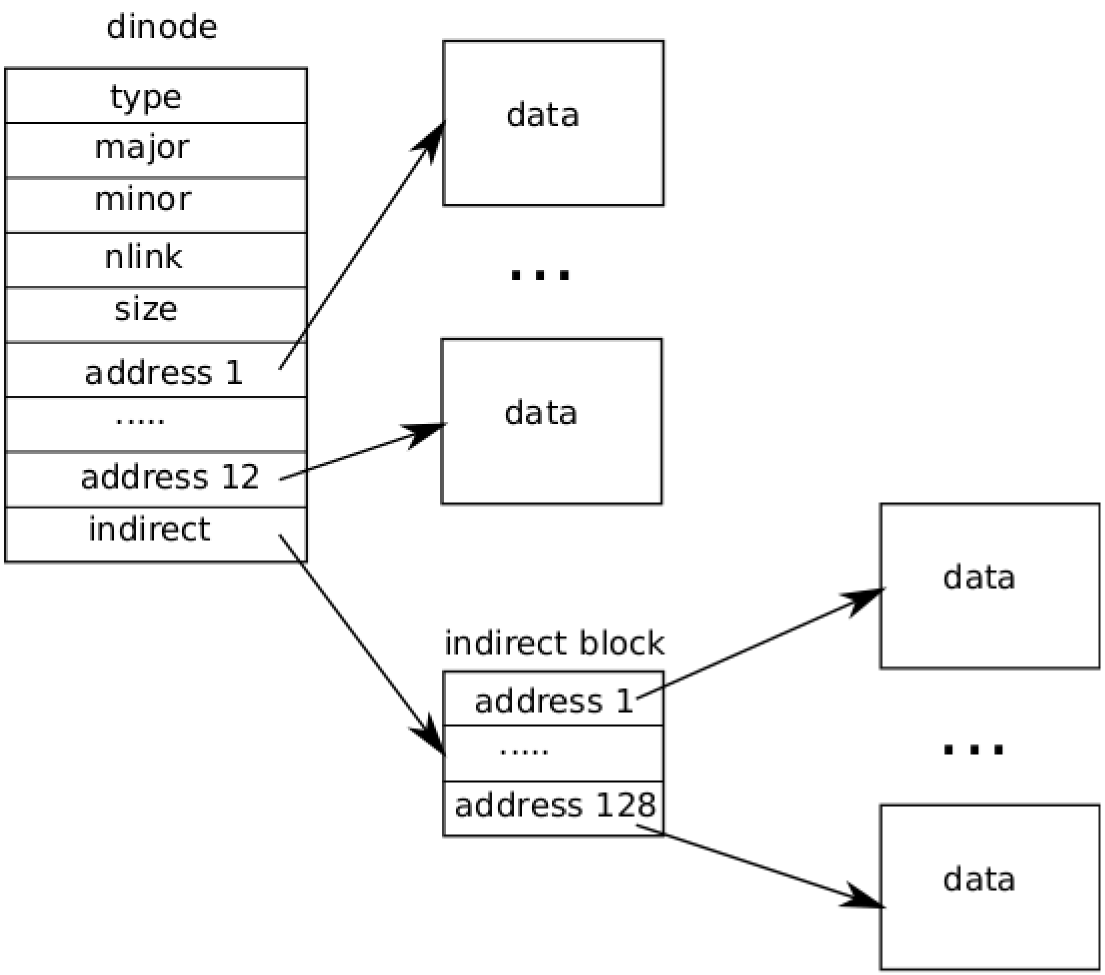
```
struct superblock {
  uint size;        // Size of file system image (blocks)
  uint nblocks;     // Number of data blocks
  uint ninodes;     // Number of inodes.
  uint nlog;        // Number of log blocks
  uint logstart;    // Block number of first log block
  uint inodestart;  // Block number of first inode block
  uint bmapstart;   // Block number of first free map block
};
#define ROOTINO 1  // root i-number
#define BSIZE 512  // block size
```

Layout of xv6 file system



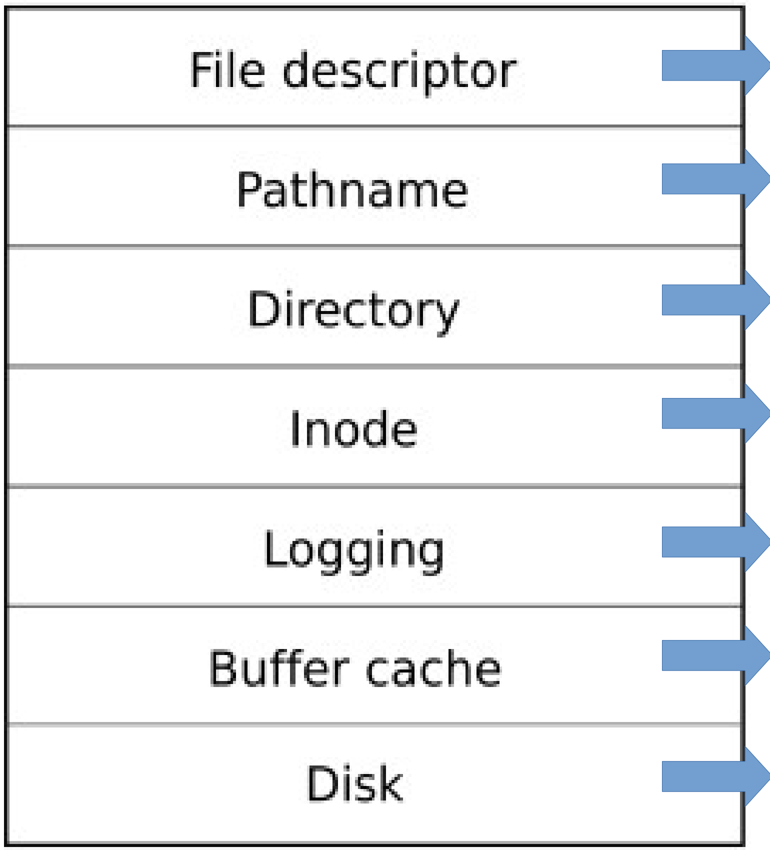

```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};

#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

File on disk


dinode
type
major
minor
nlink
size
address 1
.....
address 12
indirect
data
...
data
indirect block
address 1
.....
address 128
data
...
data

Let’s discuss lowest layer first

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

Block allocation on disk: balloc, bfree

log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

ide.c: idewait, ideinit, idestart, ideintr, iderw

static struct spinlock idelock;

static struct buf *idequeue;

static int havedisk1;

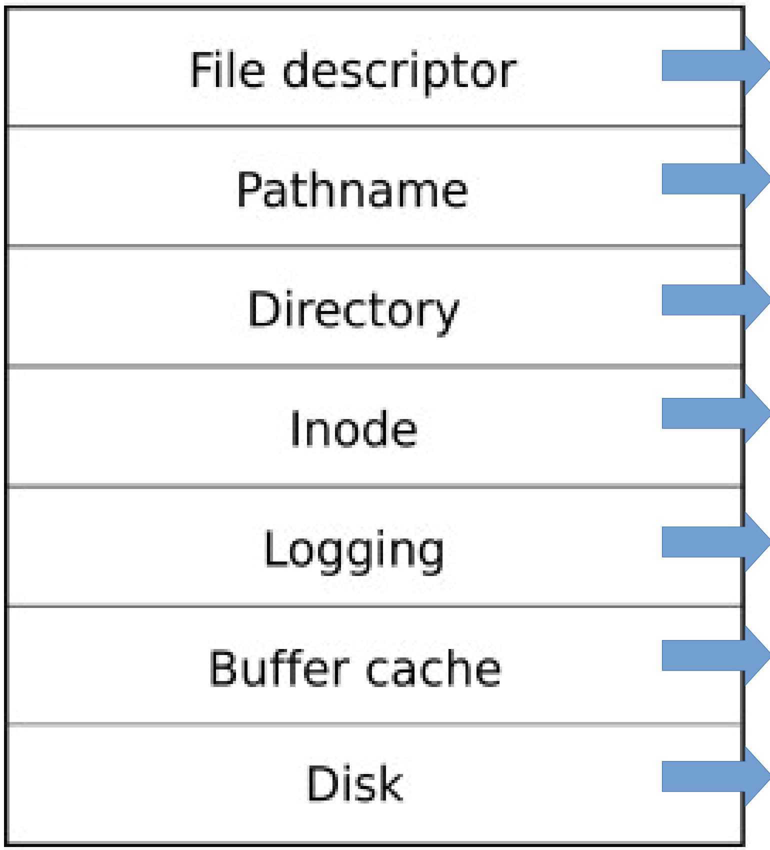
- ideinit
  - was called from main.c: main()
  - Initialized IDE controller by writing to certain ports
  - havedisk=1 setup
  - Initialize idelock
- idewait
  - BUSY loop waiting for IDE to be ready

ide.c: idewait, ideinit, idestart, ideintr, iderw

- void idestart(buf *b)
  - static void idestart(struct buf *b)
  - Calculate sector number on disk using b->blockno
  - Issue a read/write command to IDE controller.
  - (This is the first buf on idequeue)
- ideintr
  - Take idelock. Called on IDE interrupt (through alltraps()->trap())
  - Wakeup the process waiting on first buffer in buffer *idequeue;
  - call idestart(). Release idelock.
- iderw(buf *b)
  - Move buf b to end of idequeue
  - Call idestart() if not running, sleep on idelock

Let’s see buffer cache layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

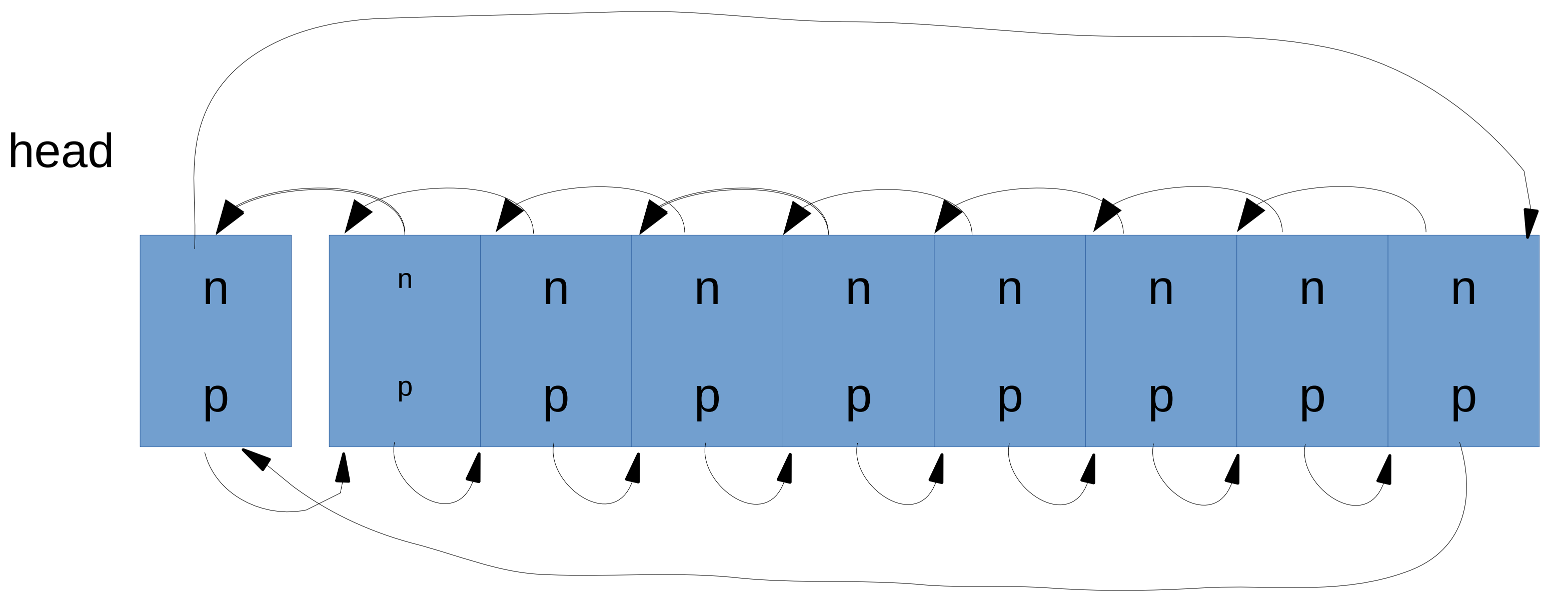
Block allocation on disk: balloc, bfree
log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

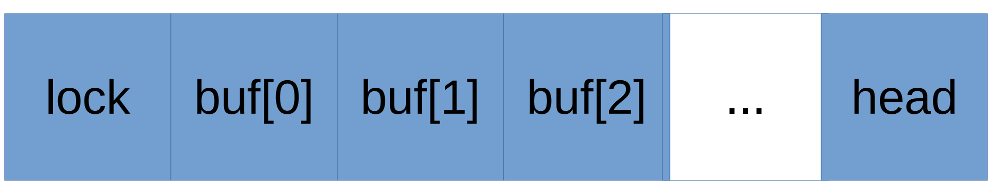
Normally, any upper layer can call any lower layer below

Reminder: After main()->binit()



Conceptually Linked liks this

Buffers keep moving on list, as LRU



struct bcache

struct buf

```
struct buf {
  int flags;  // 0 or B_VALID or B_DIRTY
  uint dev;   // device number
  uint blockno; // seq block number on device
  struct sleeplock lock; // Lock to be held by process using it
  uint refcnt; // Number of live accesses to the buf
  struct buf *prev; //  cache list
  struct buf *next; // cache list
  struct buf *qnext; // disk queue
  uchar data[BSIZE];  // data 512 bytes
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

buffer cache:
static struct buf* bget(uint dev, uint blockno)

- The bcache.head list is maintained on Most Recently Used (MRU) basis
  - head.next is the Most Recently Used (MRU) buffer
  - hence head.prev is the Least Recently Used (LRU)
- Look for a buffer with b->blockno = blockno and b->dev = dev
  - Search the head.next list for existing buffer (MRU order)
  - Else search the head.prev list for empty buffer
  - panic() if found in-use or empty buffer
- Increment b->refcnt ; Returns buffer locked
- Does not change the list structure, just returns a buf in use

buffer cache:
struct buf* bread(uint dev, uint blockno)

```
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;
  b = bget(dev, blockno);
  if((b->flags & B_VALID) == 0) {
    iderw(b);
  }
  return b; // locked buffer
}
```

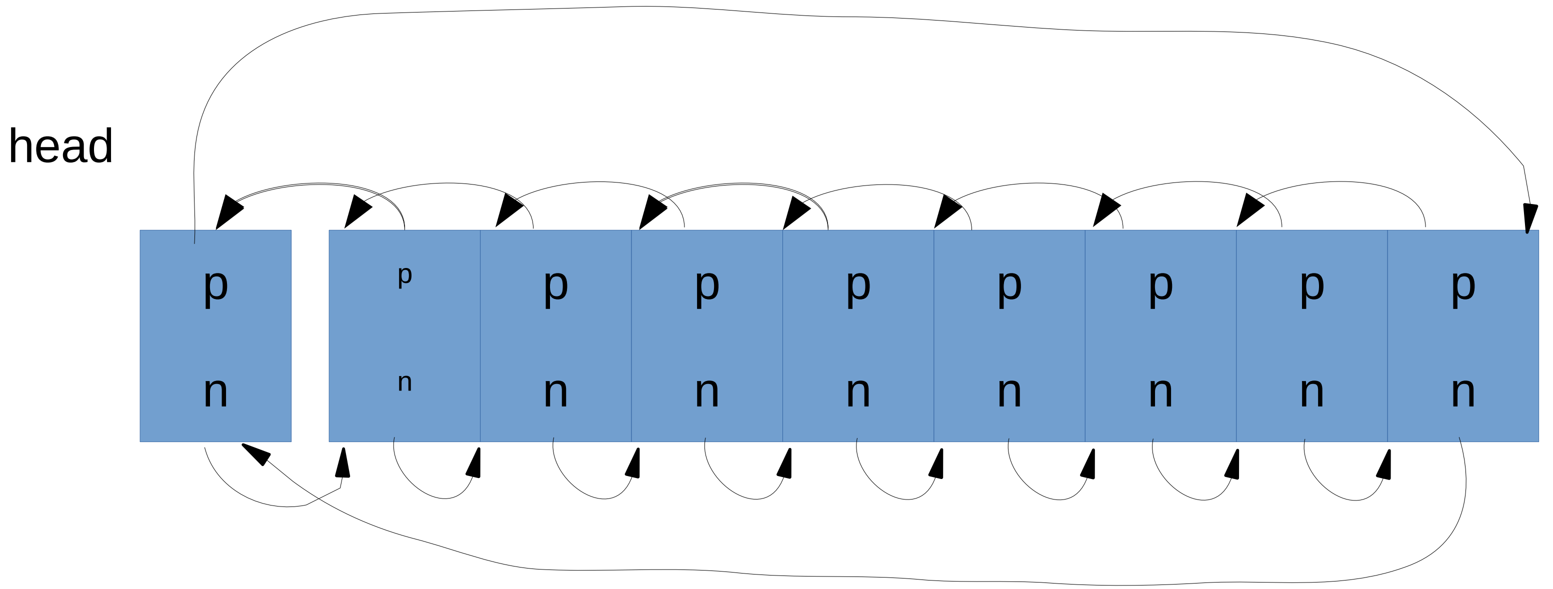
```
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  b->flags |= B_DIRTY;
  iderw(b);
}
```

Recollect: iderw moves buf to tail of idequeue, calls idestart() and sleep()

buffer cache:
void brelse(struct buf *b)

- release lock on buffer
- b->refcnt = 0
- If b->refcnt = 0
  - Means buffer will no longer be used
  - Move it to front of the front of bcache.head

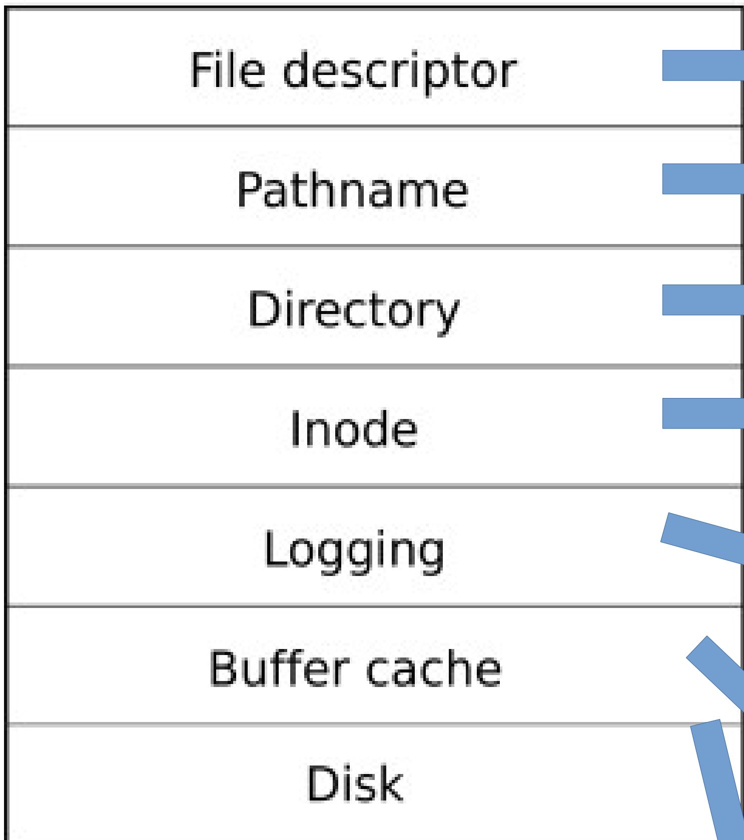
Overall in this diagram



Buffers keep moving to the front of the list and around
The list always contains NBUF=30 buffers
head.next is always the MRU and head.prev is always LRU buffer

Let’s see logging layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

Block allocation on disk: balloc, bfree

log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
  - some writes succed and some don’t
  - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be “rerun” and incomplete operations neglected

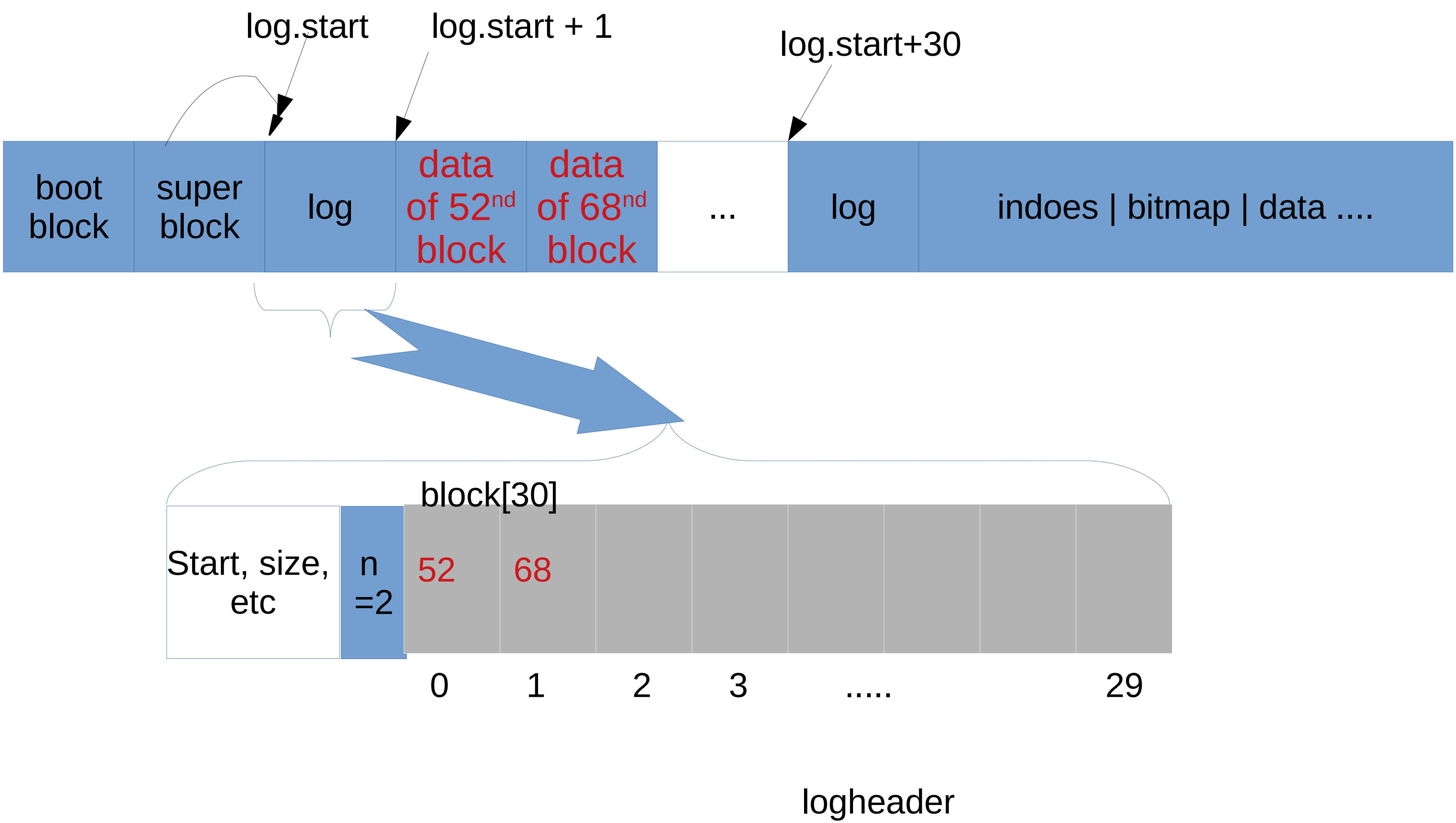
log in xv6

- xv6 system call does not directly write the on-disk file system data structures.
- A system call calls begin_op() at begining and end_op() at end
  - begin_op() increments log.outstanding
  - end_op() decrements log.outstanding, and if it’s 0, then calls commit()
- During the code of system call, whenever a buffer is modified, (and done with)
  - log_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in it’s actual place in FS as of now
- when finally commit() is called, all modified blocks are copied to disk in the file system

log

```
struct logheader { // ON DISK
  int n; // number of entries in use in block[] below
  int block[LOGSIZE]; // List of block numbers stored
};
struct log { // only in memory
  struct spinlock lock;
  int start;  // first log block on disk (starts with logheader)
  int size; // total number of log blocks (in use out of 30)
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev; // FS device
  struct logheader lh;  // copy of the on disk logheader
};
struct log log;
```

log on disk



log.start
log.start + 1
log.start+30
boot block
super block
log
data of 52nd block
data of 68nd block
...
log
indoes | bitmap | data ....
block[30]
Start, size, etc
n =2
52
68
0
1
2
3
.....
29
logheader

Typical use case of logging

```
/* In a system call code * /
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

prepare for logging. Wait if logging system is not ready or ‘committing’. ++outstanding

read and get access to a data block – as a buffer

modify buffer

note down this buffer for writing, in log. proxy for bwrite(). Mark B_DIRTY. Absorb multiple writes into one.

Syscall done. write log and all blocks. --outstanding.

If outstanding = 0, commit().

match colors in code and comments on right-side

Example of calls to logging

//file_write() code

begin_op();

ilock(f->ip);

/*loop */ r = writei(f->ip, ...);

iunlock(f->ip);

end_op();

- each writei() in turn calls bread(), log_write() and brelse()
  - also calles iupdate(ip) which also calls bread, log_write and brelse
- Multiple writes are combined between begin_op() and end_op()

Logging functions

- Initlog()
  - Set fields in global log.xyz variables, using FS superblock
  - Recovery if needed
  - Called from first forkret()
- Following three called by FS code
- begin_op(void)
  - Increment log.outstanding
- end_op(void)
  - Decrement log.oustanding and call commit() if it's zero
- log_write(buf *)
  - Remember the specified block number in log.lh.block[] array
  - Set the block to be dirty

- write_log(void)
  - Called only from commit()
  - Use block numbers specified in log.lh.block and copy those blocks from memory to log-blocks
- commit(void)
  - Called only from end_op()
  - write_log()
  - Write header to disk log-header
  - Copy from log blocks to actual FS blocks
  - Reset and write log header again

Let’s see block allocation layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	**file.c** fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	**fs.c** namex, namei, nameiparent, skipelem
Directory	**fs.c** dirlookup, dirlink
Inode	**fs.c** iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree **log.c :** begin_op, end_op, initlog, commit,
Buffer cache	**bio.c** binit, bget, bread, bwrite, brelse
Disk	**ide.c:** idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

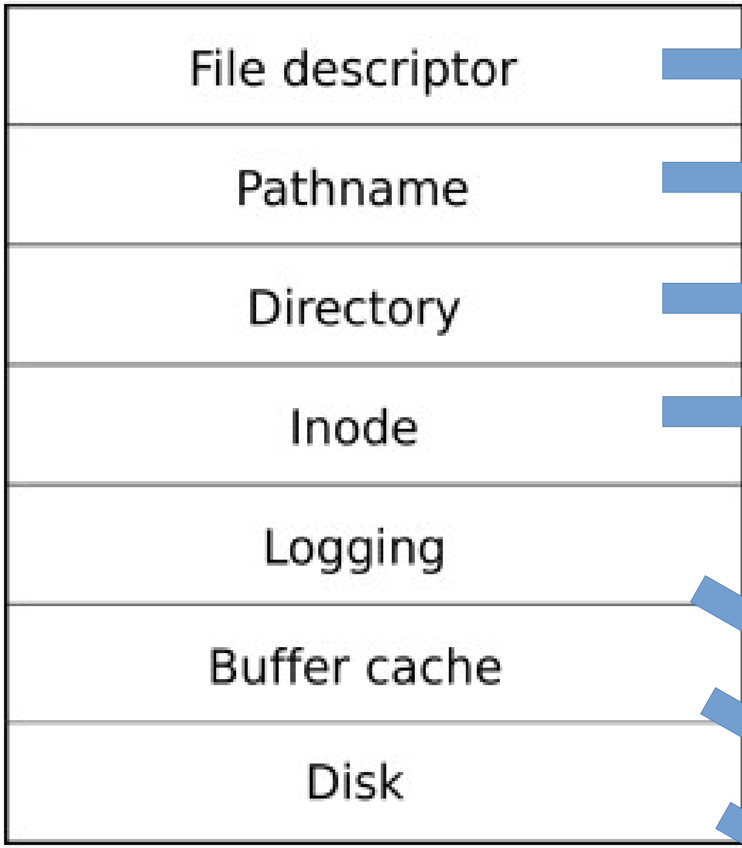
Abhijit: Block allocator should be considered as another Layer!

allocating & deallocating blocks on DISK

- **balloc(devno)**
  - **looks for a block whose bitmap bit is zero, indicating that it is free.**
  - **On finding updates the bitmap and returns the block.**
  - **balloc() calls bread()->bget to get a block from disk in a buffer.**
    - **Race prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time.**
  - **Calls log_write(bp);**
    - **Thus writes to bitmap blocks are also logged**
- **bfree(devno, blockno)**
  - **finds the right bitmap block and clears the right bit.**
  - **Also calls log_write()**

Let’s see Inode Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

Block allocation on disk: balloc, bfree

log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

On disk & in memory inodes

```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} icache;
```

```
// On-disk inode structure
struct dinode {
  short type;     // File type
  short major;   // T_DEV Major device
number
  short minor;   // Minor device number
  short nlink;    // Number of links
  uint size;       // Size of file (bytes)
  uint addrs[NDIRECT+1];   /
};
```

```
// in-memory copy of an inode
struct inode {
  uint dev;          // Device number
  uint inum;          // Inode number
  int ref;          // Reference count
  struct sleeplock lock; // protects
everything below here
  int valid;         //  been read from disk?

  short type;        // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
  - as long as ‘ref’ is >0
- The iget and iput functions acquire and release pointers to an inode, modifying the ref count.
- See the caller graph of iget()
  - all those who call iget()
- Sleep lock in ‘inode’ protects
  - fields in inode
  - data blocks of inode

iget and iupdate

- iget
  - searches for an existing/free inode in icache and returns pointer to one
  - if found, increments ref and returns pointer to inode
  - else gets empty inode , initializes, ref=1 and return
  - No lock held after iget()
  - Code must call ilock() after iget() to get lock
  - During lookup (later), many processes can iget() an inode, but only one holds the lock
- iupdate(inode *ip)
  - read on disk block of inode
  - get on disk inode
  - modify it as specified in ‘ip’
  - modify disk block of inode
  - log_write(disk block of inode)

itrunc , iput

- iput(ip)
  - if ref is 1
    - itrunc(ip)
    - type = 0
    - iupdate(ip)
    - i->valid = 0 // free in memory
  - else
    - ref--
- itrunc(ip)
  - write all data blocks of inode to disk
    - using bfree()
  - ip->size = 0
    - Inode is freed from use
  - iupdate(ip)
  - called from iput() only when ‘ref’ becomes zero

race in iput ?

- A concurrent thread might be waiting in ilock to use this inode
  - and won't be prepared to find the in ode is not longer allocated
- This is not possible. Why?
  - no way for a syscall to get a ref to a inode with ip->ref = 1

```
void
iput(struct inode *ip)
{
  acquiresleep(&ip->lock);
  if(ip->valid && ip->nlink == 0){
    acquire(&icache.lock);
    int r = ip->ref;
    release(&icache.lock);
    if(r == 1){
      // inode has no links and no other references: truncate and free.
      itrunc(ip);
```

buffer and inode cache

- to read an inode, it's block must be read in a buffer
- So the buffer always contains a copy of the on-disk dinode
  - duplicate copy in in-memory inode
- The inode cache is write-through,
  - code that modifies a cached inode must immediately write it to disk with iupdate
- Inode may still exist in the buffer cache

allocating inode

- ialloc(dev, type)
  - Loop over all disk inodes
  - read inode (from it's block)
  - if it's free (note inum)
  - zero on disk inode
  - write on disk inode (as zeroes)
  - return iget(dev, inum)
- panic if no free inodes

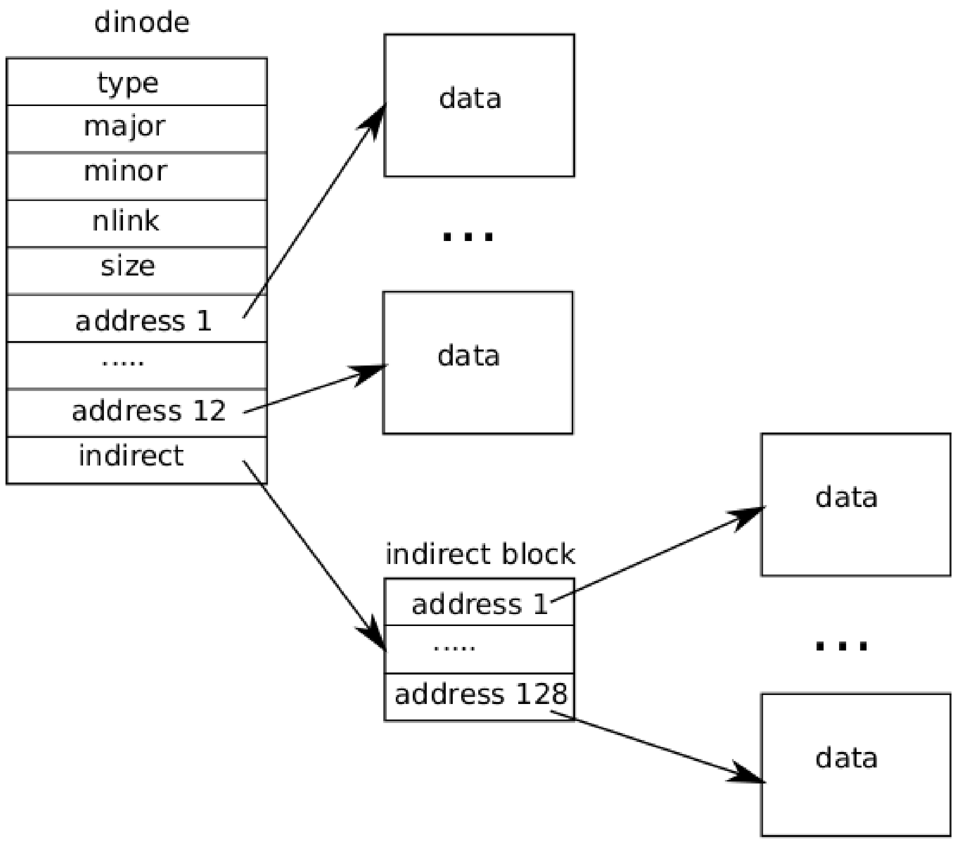
- ilock
  - code must acquire ilock before using inode's data/fields
  - Ilock reads inode if it's already not in memory

Trouble with iput() and crashes

- iput() doesn't truncate a file immediately when the link count for the file drops to zero, because
  - some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it.
- if a crash happens before the last process closes the file descriptor for the file,
  - then the file will be marked allocated on disk but no directory entry points to it
- Unsolved problem.
- How to solve it?

Get Inode data: bmap(ip, bn)

- **Allocate 'bn'th block for the file given by inode 'ip'**
- **Allocate block on disk and store it in either direct entries or block of indirect entries**
  - **allocate block of indirect entries if needed using balloc()**



writing/reading data at a given offset in file

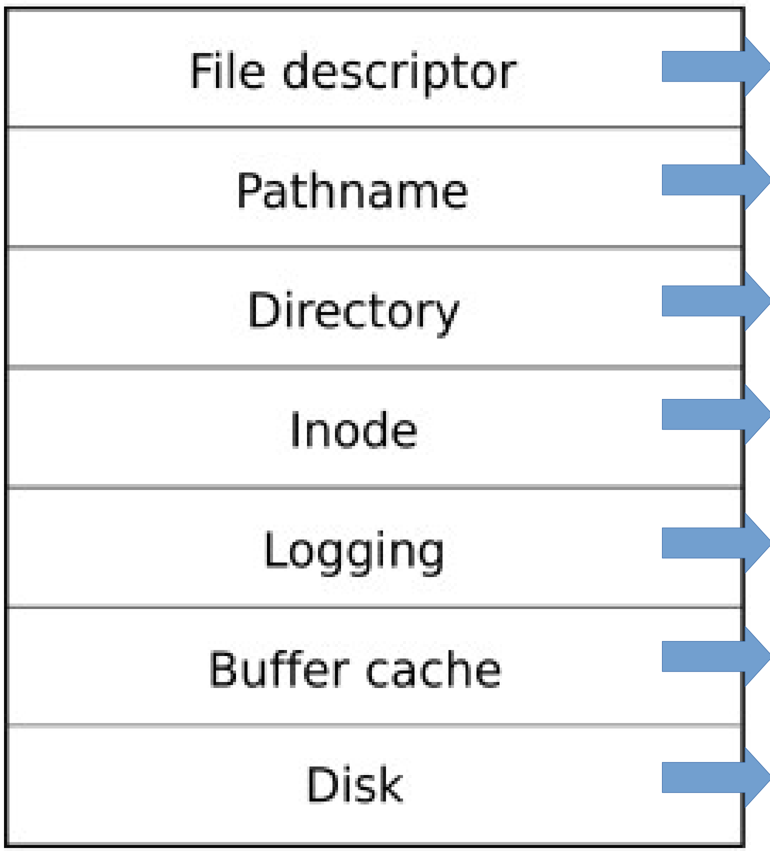
readi(struct inode *ip, char *dst, uint off, uint n)

writei(struct inode *ip, char *src, uint off, uint n)

- Calculate the block number in file where ‘off’ belongs
- Read sufficient blocks to read ‘n’ bytes
- using bread(), brelse()
- Call devsw.read if inode is a device Inode.
- Writei() also updates size if required

Reading Directory Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

Block allocation on disk: balloc, bfree

log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

directory entry

```
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

Data of a directory file is a sequence of such entries. To find a name, just get all the data blocks and search the name

How to get the data for a directory? We already know the ans!

struct inode* dirlookup(struct inode *dp, char *name, uint *poff)

- Given a pointer to directory inode (dp), name of file to be searched
  - return the pointer to inode of that file (NULL if not found)
  - set the ‘offset’ of the entry found, inside directories data blocks, in poff
- How was ‘dp’ obtained? Who should be calling dirlookup? Why is poff returned?
  - During resolution of pathnames?
- Code: call readi() to get data of dp, search name in it, name comes with inode-num, iget() that inode-num

int
dirlink(struct inode *dp, char *name, uint inum)

- Create a new entry for ‘name’_’inum’ in directory given by ‘dp’
  - inode number must have been obtained before calling this. How to do that?
- Use dirlookup() to verify entry does not exist!
- Get empty slot in directory’s data block
- Make directory entry
- Update directory inode! writei()

namex

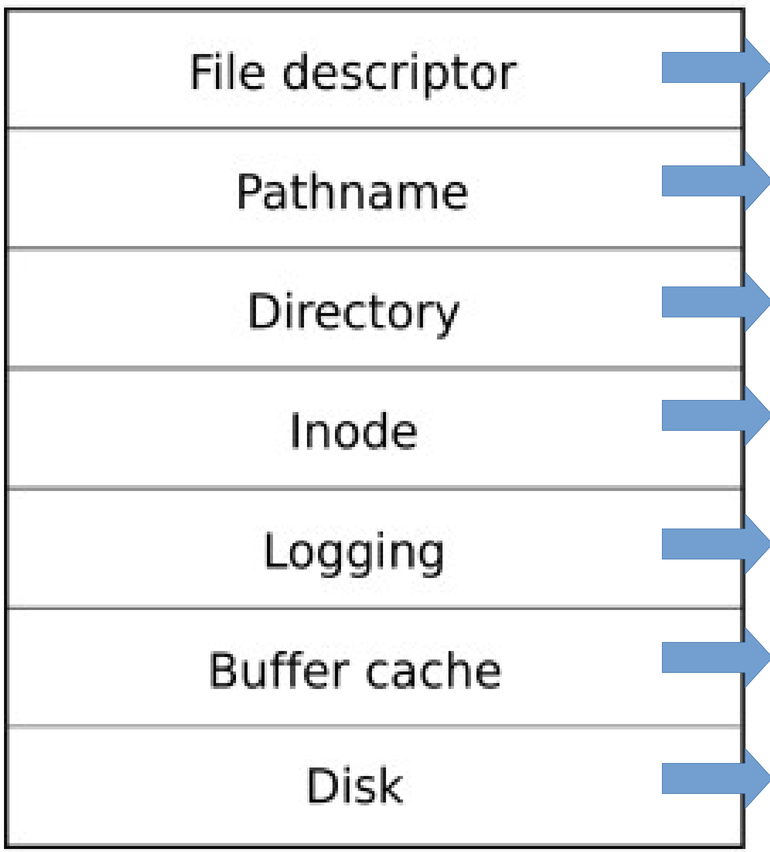
- Called by namei(), or nameiparent()
- Just iteratively split a path using “/” separator and get inode for last component
- iget() root inode, then
- Repeatedly calls
  - split on “/”, dirlookup() for next component
-

races in namex()

- Crucial. Called so many times!
- one kernel thread is looking up a pathname another kernel thread may be changing the directory by calling unlink
  - when executing dirlookup in namex, the lookup thread holds the lock on the directory and dirlookup() returns an inode that was obtained using iget.
- Deadlock? next points to the same inode as ip when looking up ".". Locking next before releasing the lock on ip would result in a deadlock.
  - namex unlocks the directory before obtaining a lock on next.

File descriptor layer code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,



file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,

fs.c namex, namei, nameiparent, skipelem

fs.c dirlookup, dirlink

fs.c iiinit, ialloc, iupdate,iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,

Block allocation on disk: balloc, bfree

log.c : begin_op, end_op, initlog, commit,

bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

data structures related to “file” layer

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // used only if it works as a pipe
  struct inode *ip;
  uint off;
};
// interesting no lock in struct file !
```

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

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable; //global table from which ‘file’ is allocated to every process
```

Lock is used to protect updates to every entry in the array

Multiple processes accessing same file.

- Each will get a different ‘struct file’
  - but share the inode !
  - different offset in struct file, for each process
  - Also true, if same process opens file many times
- File can be a PIPE (more later)
  - what about STDIN, STDOUT, STDERR files ?
  - Figure out!
- ref
  - used if the file was ‘duped’ or process forked . in that case the ‘struct file’ is shared

file layer functions

- filealloc
  - find an empty struct file in ‘ftable’ and return it
  - set ref = 1
- filedup(file *)
  - simply ref++
- fileclose
  - --ref
  - if ref = 0
    - free struct file
    - iput() / pipeclose()
    - note – transaction if iput() called
- filestat
  - simply return fields from inode, after holding lock. on inodes for files only.

file layer functions

- fileread
  - call readi() or piperead()
  - readi() later calls device-read or inode read (using bread())
- filewrite
  - call pipewrite() or writei()
  - writei() is called in a loop, within a transaction

- Why does readi() call read on the device , why not fileread() itself call device read ?

pipes

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;
 // number of bytes read
  uint nwrite;
 // number of bytes written
  int readopen;
  // read fd is still open
  int writeopen;
 // write fd is still open
};
```

- functions
  - pipealloc
  - pipeclose
  - piperead
  - pipewrite
-

pipes

- pipealloc
  - allocate two struct file
  - allocate pipe itself using kalloc (it’s a big structure with array)
  - init lock
  - initialize both struct file as 2 ends (r/w)
- pipewrite
  - wait if pipe full
  - write to pipe
  - wakeup processes waiting to read
- piperead
  - wait if no data
  - read from pipe
  - wakeup processes waiting to write
- Good producer consumer code !

Further to reading system call code now

- Now we are ready to read the code of system calls on file system
  - sys_open, sys_write, sys_read , etc.
- Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.
- Also think of locks that need to be held.