Filesystems

Introduction

- Literature refers to three different storage levels
  - 1st level: CPU registers cache, main memory, …- volatile & very fast
  - 2nd level: hard disk drives (HDD), solid state disks (SSD), optical discs … – persistent & fast
  - 3rd level: tape drives, … - persistent & slow

- Challenges to work with 2nd level storage? Variety of
  - Devices (classes, manufacturers, models)
  - Applications (commodity, performance, reliability)

- Low-level to high-level mapping
  - Devices read/write data blocks
  - Application developers usually deal with files & directories

Introduction (2)

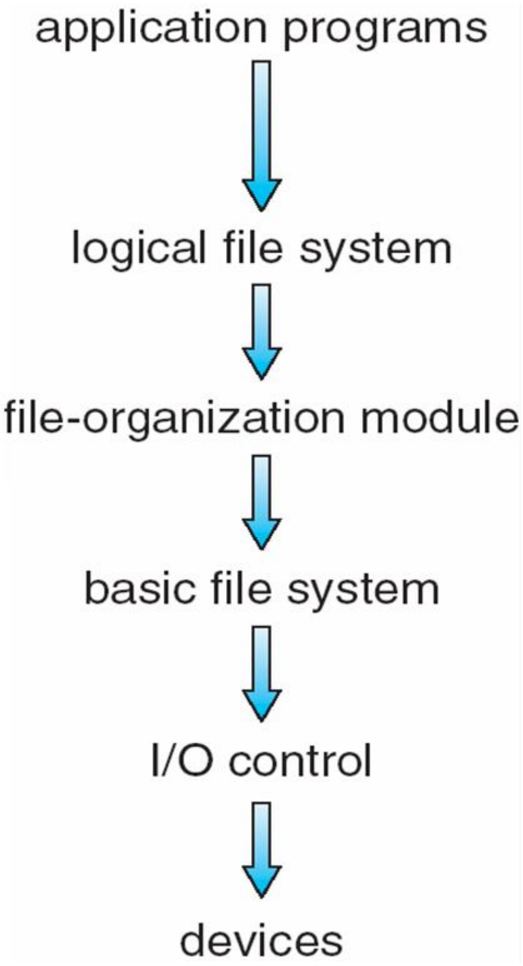
- OS has to abstract from physical properties and provide tunability in regard of applications → file system (FS) addresses that

- FS provides
  - Disk organization (map bytes to blocks, manage free space, etc.)
  - Naming (use file names instead of block numbers)
  - Robustness (prevent data loss after crashes)
  - Security (define & prohibit illegitimate accesses)

- Plenty FS exist, OS should provide generic interface
  - Concurrent support for many FSs by the OS
  - Applications developers should not deal with specifics of individual FSs

Outline

- Today's lecture:
  - Basic FS concepts
  - Virtual File System
  - Implementations: ext3 & ext4
  - I/O control



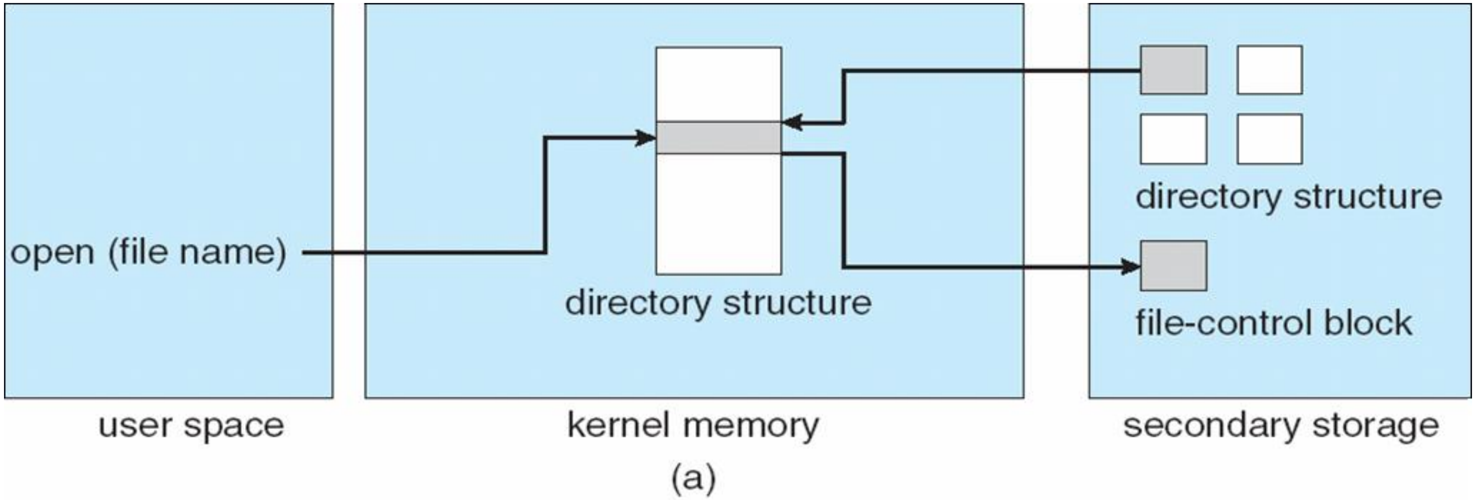
Layered File system

Basic FS concepts

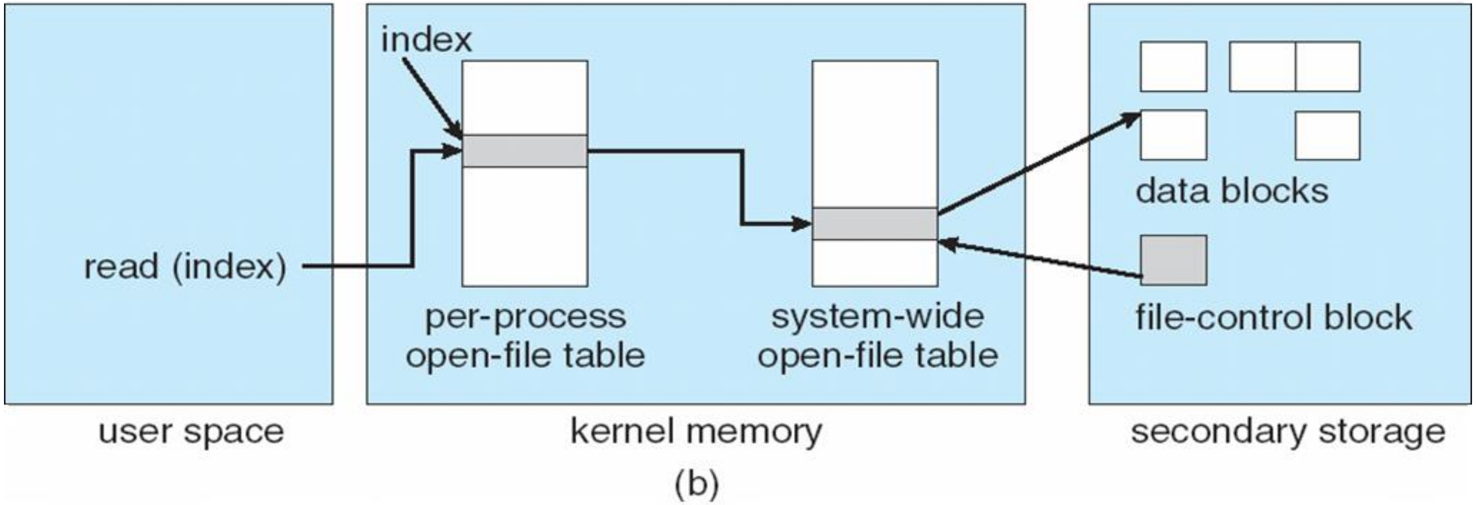
- File – a logical storage unit, collection of related info.
- mapped by the OS to physical devices, has
  - Attributes: name, identifier, type, location, time stamps (creation, last modification, last use), access rights, ownership
  - Operations: create, write, read, reposition, deletion
  - Access method: sequential, direct
- Managed by the OS with a file control block (FCB)

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

Basic FS concepts (2)



Opening a file

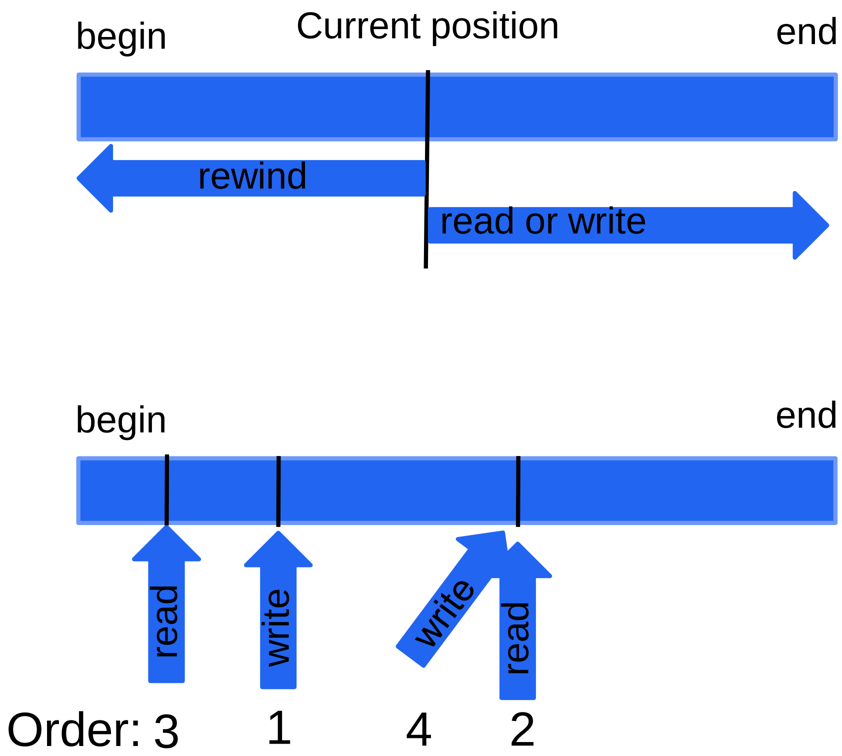


Reading a file

Basic FS concepts (3)

Access methods

- Sequential access
  - Most common
  - E.g., editors or compilers use seq. access
- Direct access
  - File considered as a numbered sequence of blocks
  - Read/write operations in no particular order
  - E.g., databases use direct access



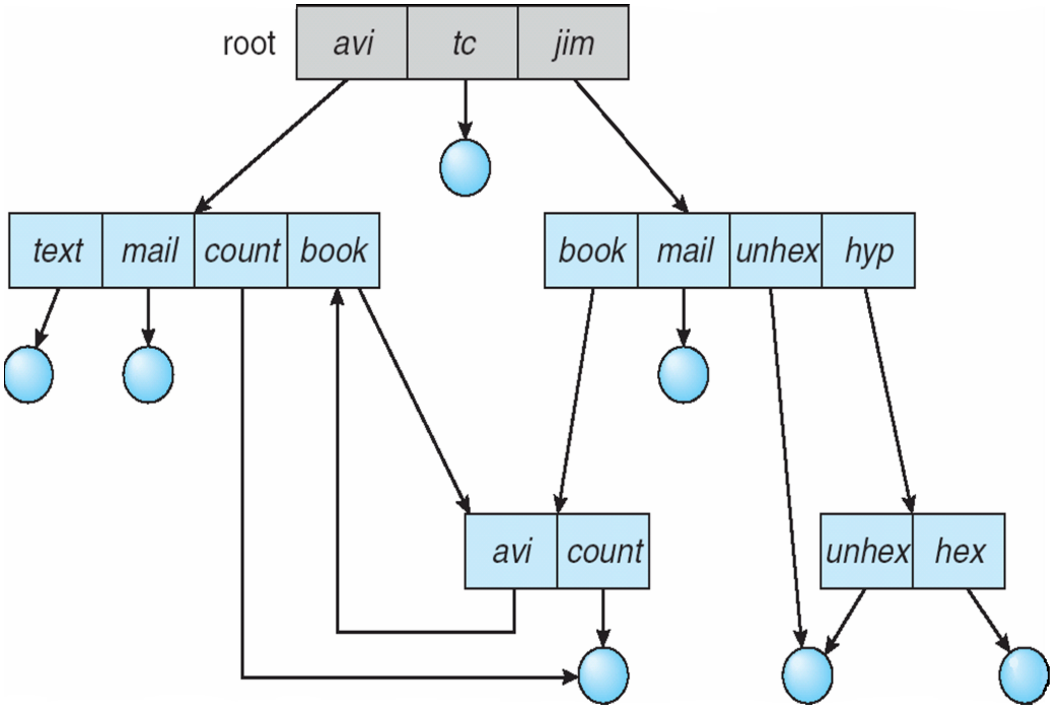
Logical disk structure

- Terminology ambigious, let's use:
- Partitions (per disk)
- Volumes (for FS, may span along various partitions of indep. disks)

Directories

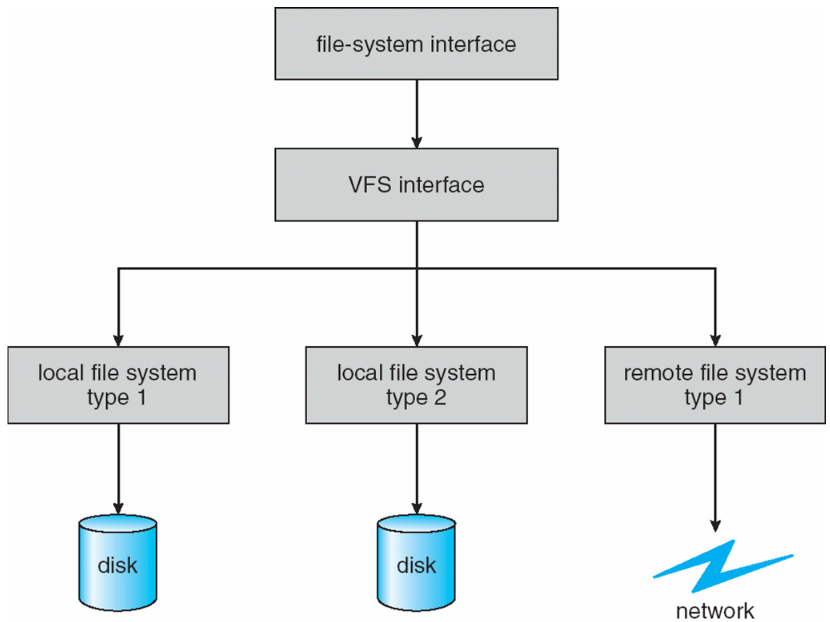
- Directory – logical structure for a FS instance, organizational unit for files, has
  - Attributes like a file
  - Operations: search/create/delete/rename files, list/rename/delete directory

•General graph directory structure
•Connected graph
•Most FS disallow user defined cycles



Virtual File System (VFS)

- Linux implements VFS
  - Abstraction for specific FS implementations
  - Concurrently support many FS, local and networked
  - Applications use various FS via fixed set of UNIX system calls



Application (user space)	VFS (kernel space)	FS (e.g. ext4) (kernel space)	Harddisk drive
write() →	vfs_write() →	FS write method →	

VFS (2)

- VFS is object oriented
- VFS has four important objects
  - inode – "index node" contains file information
  - File – an open file (currently associated to a process)
  - Dentry – "directory entry" contains single components of a path
  - Super block – detailed information about a specific FS instance
- Directories are handled like files
  - Dentry objects could also be a file, data structure only in memory, not on disk
  - Directories are special files, stored as inodes
- Each of the objects provides a set of operations

VFS – inode object

- ❑ UNIX: inodes can be read directly from disk
- ❑ Are created by the FS
- ❑ In UNIX "everything is a file" → special inodes exist:
  - i_pipe - pipes
  - i_bdev – block devices
  - i_cdev – character devices
- ❑ Direct and indirect data addressing (more about that later, originates from UFS)

```c
struct inode {
	/* RCU path lookup touches following: */
	umode_t			i_mode;
	uid_t			i_uid;
	gid_t			i_gid;
	const struct inode_operations	*i_op;
	struct super_block	*i_sb;

	spinlock_t		i_lock;	/* i_blocks, i_bytes, maybe i_size */
	unsigned int		i_flags;
	unsigned long		i_state;
#ifdef CONFIG_SECURITY
	void			*i_security;
#endif
	struct mutex		i_mutex;


	unsigned long		dirtied_when;	/* jiffies of first dirtying */

	struct hlist_node	i_hash;
	struct list_head	i_wb_list;	/* backing dev IO list */
	struct list_head	i_lru;		/* inode LRU list */
	struct list_head	i_sb_list;
	union {
		struct list_head	i_dentry;
		struct rcu_head		i_rcu;
	};
	unsigned long		i_ino;
	atomic_t		i_count;
	unsigned int		i_nlink;
	dev_t			i_rdev;
	unsigned int		i_blkbits;
	u64			i_version;
	loff_t			i_size;
#ifdef __NEED_I_SIZE_ORDERED
	seqcount_t		i_size_seqcount;
#endif
	struct timespec		i_atime;
	struct timespec		i_mtime;
	struct timespec		i_ctime;
	blkcnt_t		i_blocks;
	unsigned short		i_bytes;
	struct rw_semaphore	i_alloc_sem;
	const struct file_operations	*i_fop;	/* former ->i_op->default_file_ops */
	struct file_lock	*i_flock;
	struct address_space	*i_mapping;
	struct address_space	i_data;
#ifdef CONFIG_QUOTA
	struct dquot		*i_dquot[MAXQUOTAS];
#endif
	struct list_head	i_devices;
	union {
		struct pipe_inode_info	*i_pipe;
		struct block_device	*i_bdev;
		struct cdev		*i_cdev;
	};

	__u32			i_generation;

#ifdef CONFIG_FSNOTIFY
	__u32			i_fsnotify_mask; /* all events this inode cares about */
	struct hlist_head	i_fsnotify_marks;
#endif

#ifdef CONFIG_IMA
	atomic_t		i_readcount; /* struct files open RO */
#endif
	atomic_t		i_writecount;
#ifdef CONFIG_FS_POSIX_ACL
	struct posix_acl	*i_acl;
	struct posix_acl	*i_default_acl;
#endif
	void			*i_private; /* fs or device private pointer */
};
```

VFS – Inode object operations

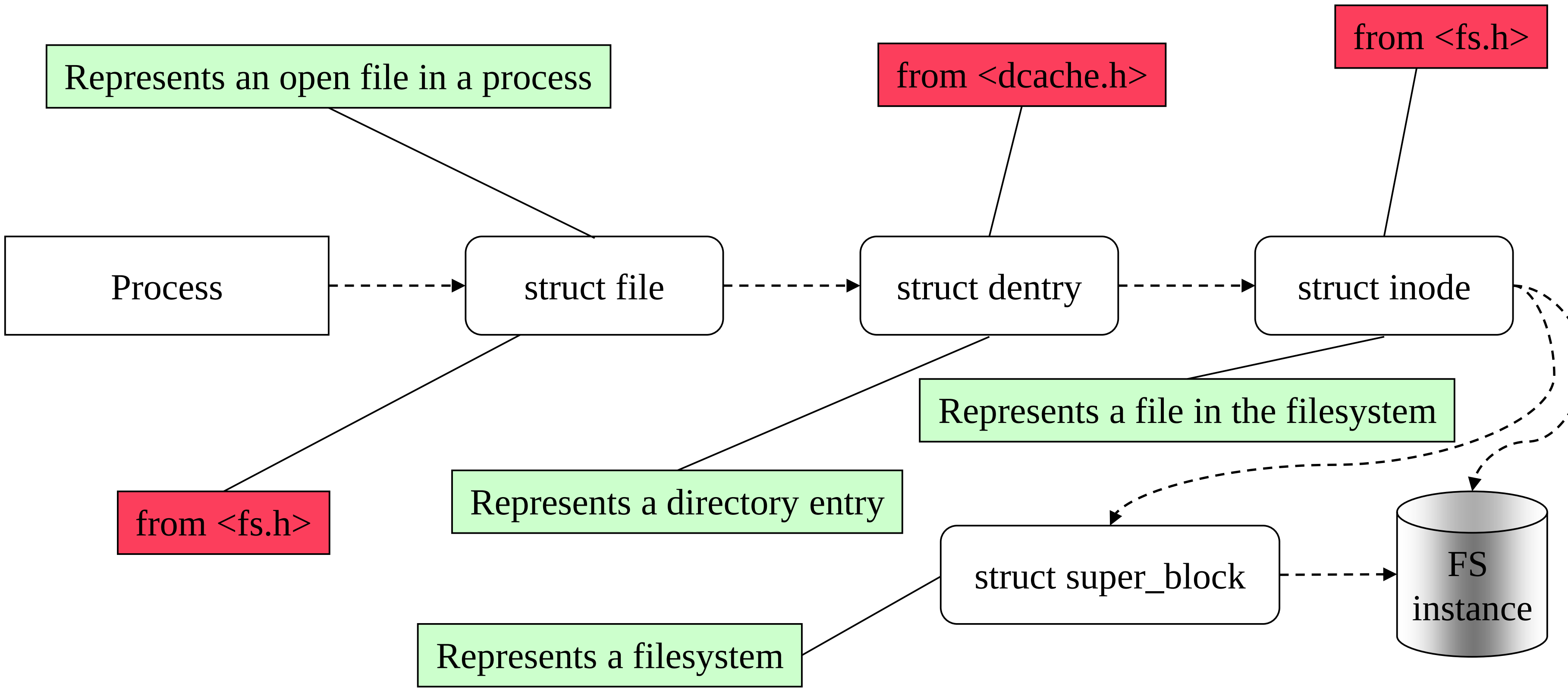
- FSs need to implement VFS's inode operations
- Exercise session will look at structs and operations in details, also for
  - dentry
  - file
  - etc.

```
struct inode_operations {
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        int (*permission) (struct inode *, int, unsigned int);
        int (*check_acl)(struct inode *, int, unsigned int);

        int (*readlink) (struct dentry *, char __user *,int);
        void (*put_link) (struct dentry *, struct nameidata *, void *);

        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        void (*truncate) (struct inode *);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
        void (*truncate_range)(struct inode *, loff_t, loff_t);
        int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                      u64 len);
        struct file *(*open)(struct dentry *, int flags, const struct cred *);
} ____cacheline_aligned;
```

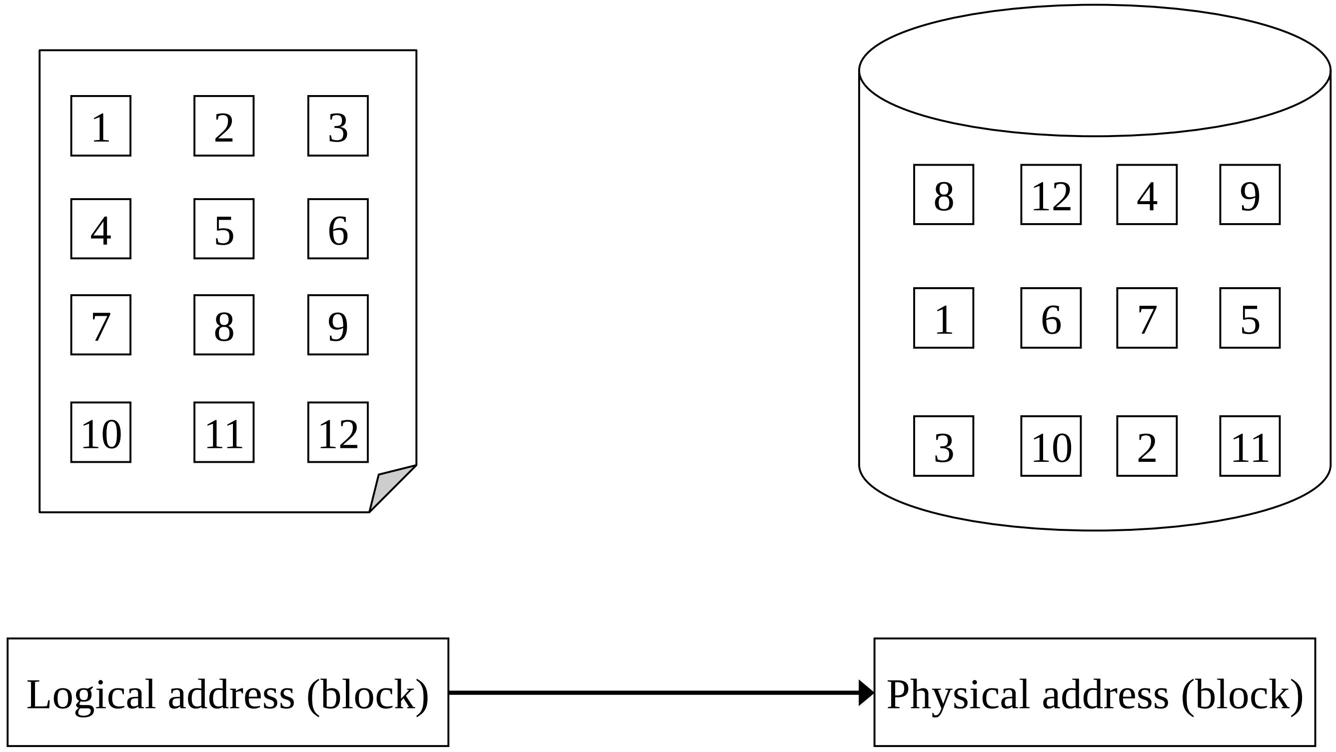
VFS – Conceptual View



Outline

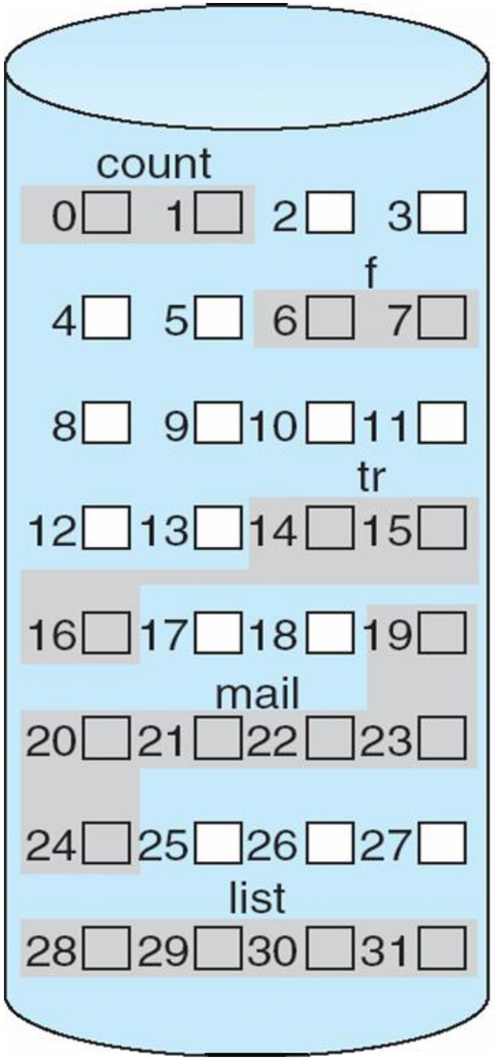
- Today's lecture:
  - Basic FS concepts
    - User perspective
    - OS & programmer perspective
  - FS organization & implementation
  - Examples: ext3 & ext4 FS

Files Consist of Blocks of Data



- Where to store/allocate blocks?
- How to find/access files/blocks?
- What is a good block size?

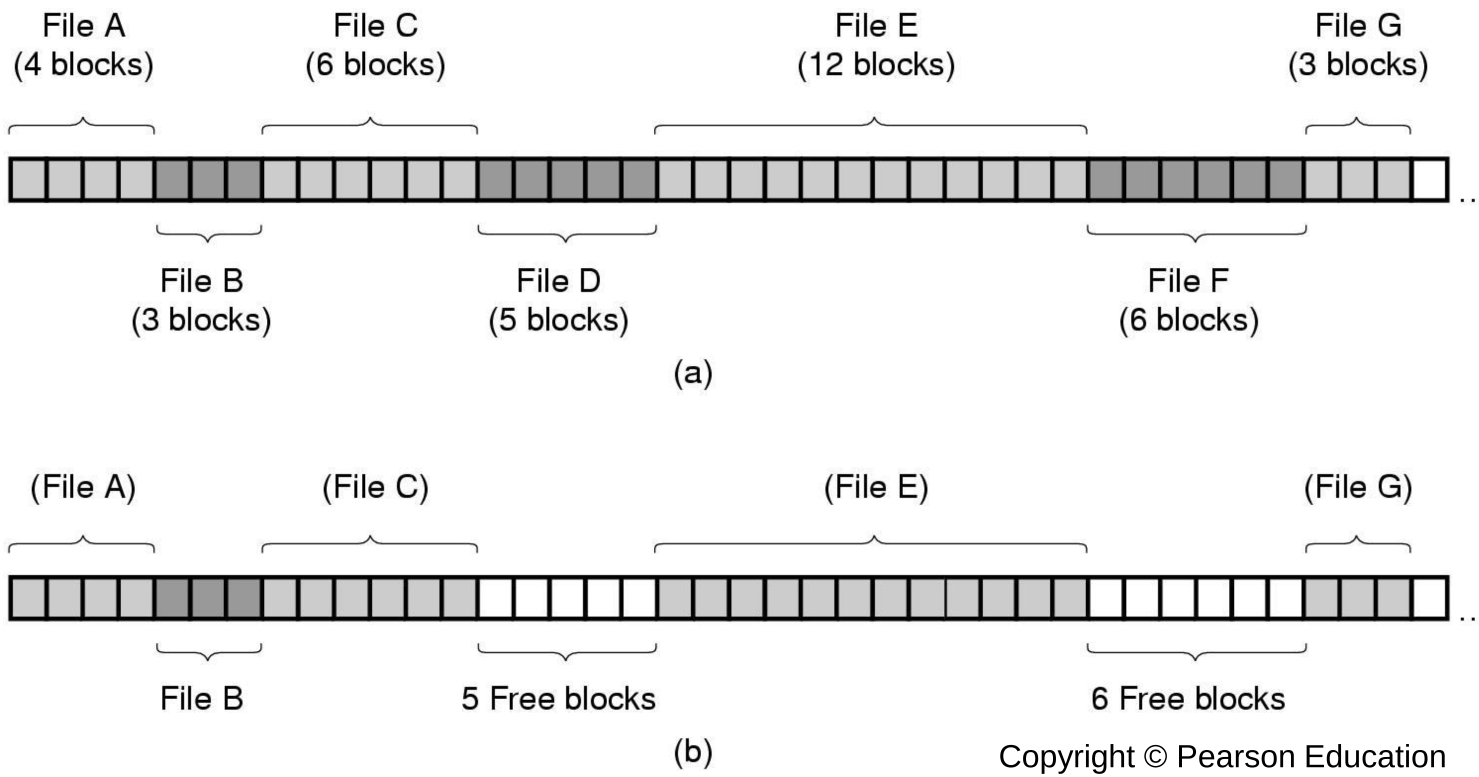
Contiguous Allocation



directory

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

Contiguous Allocation (2)



(a) Contiguous allocation of disk space for 7 files
(b) State of the disk after files *D* and *F* have been removed

Contiguous Allocation (3)

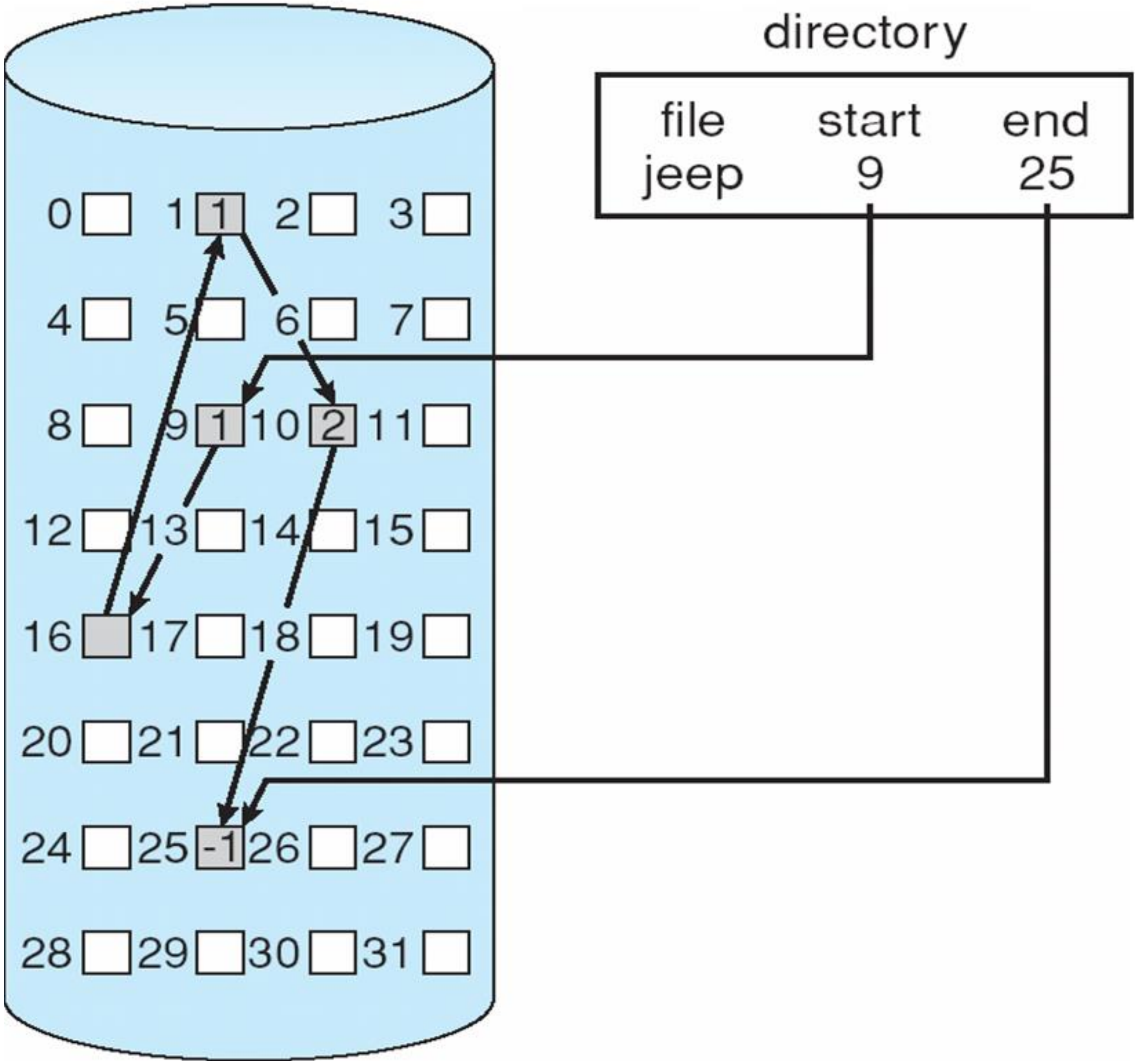
- Finding files/blocks is easy
  - Offset + number of blocks
- Excellent read performance
- Good for both: sequential and direct access

- But: suffers from fragmentation
  - Compaction is expensive
  - Reuse of holes?
  - Need to know max file size when allocating

- Where could this allocation be useful?

- What is the standard alternative to static allocation in computer science (think arrays in C)?

Linked List Allocation



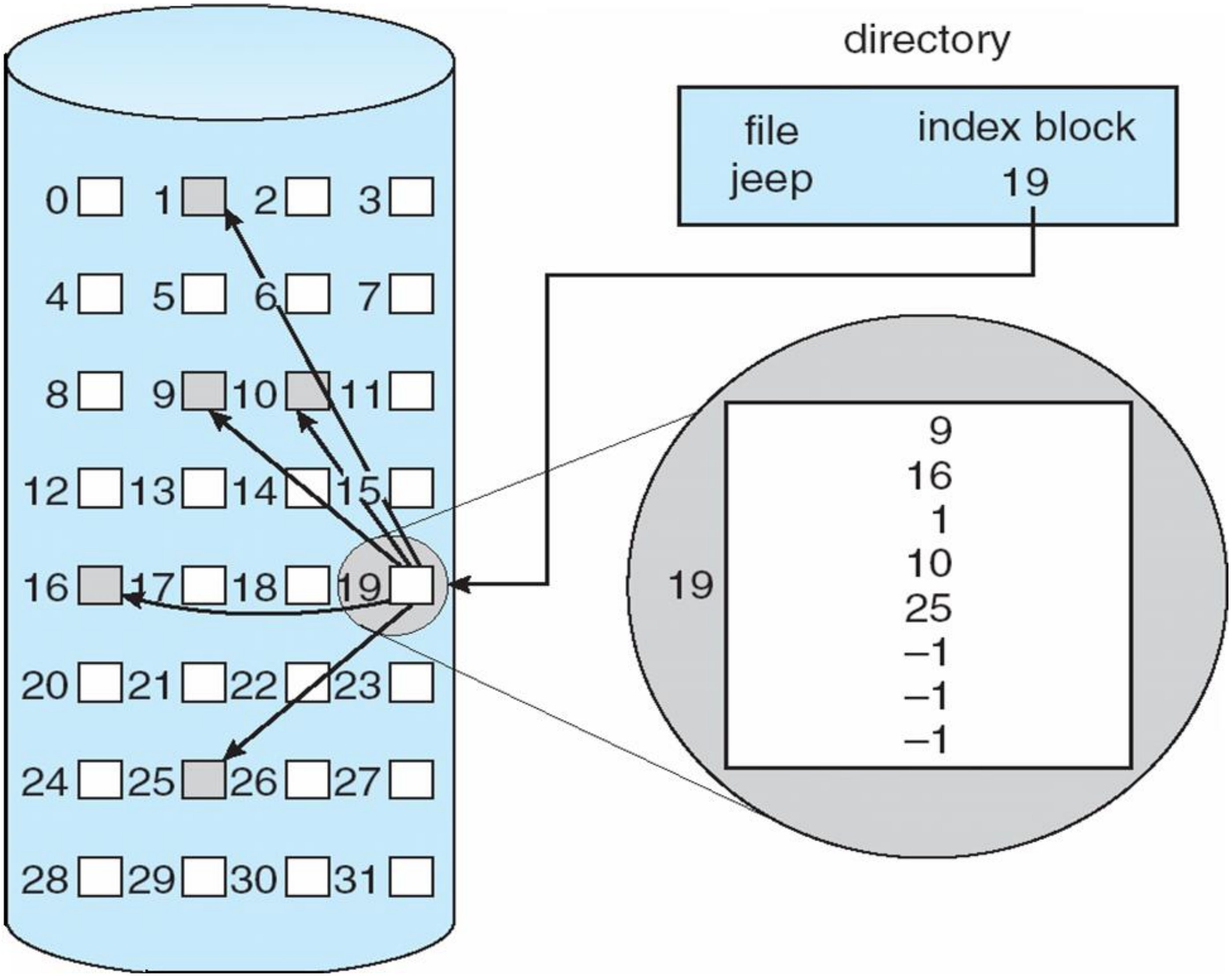
Linked List Allocation (2)

- No holes, no pre-allocation problem
- Only address of first block needs to be stored
- Good for sequential access, suboptimal for direct access

- Finding block n is expensive
  - Need to read all *n-1* blocks prior to block *n*
- Size of data block is not 2^x
  - Due to pointer overhead

- Both disadvantages can be removed using a new data structure, which?

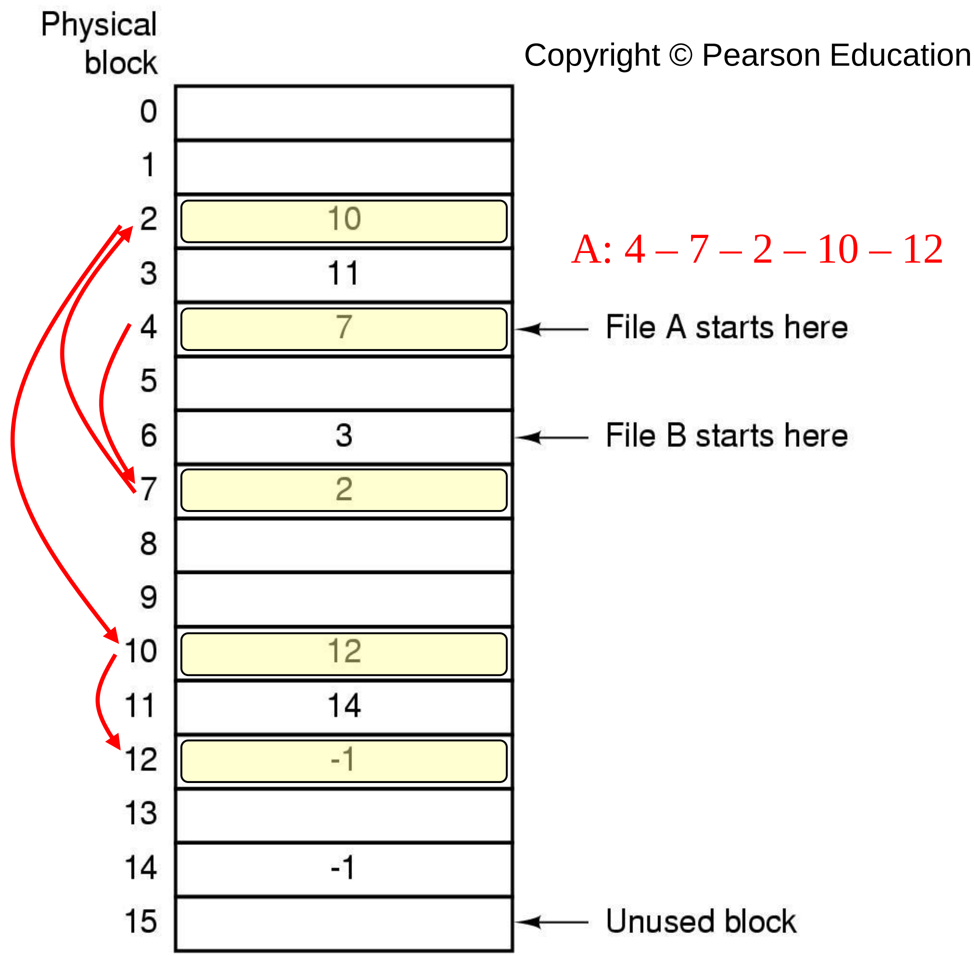
Indexed Allocation



Indexed Allocation (2)

- Requires an index table
- Random access, allows for good sequential and indexed access
- No external fragmentation
  - What about internal fragmentation?

Implementing Files - FAT



Idea: store the pointers in a table

- Fast random access
  - Table can be stored in RAM
- Full 2^x block size

This method is called FAT
(*File Allocation Table*)

Disadvantage: table size

20 GB, block size 1 KB → 20M blocks → 80 MB (4-byte entries) or 60MB (3-byte entries)

What can we do to reduce the storage requirement?

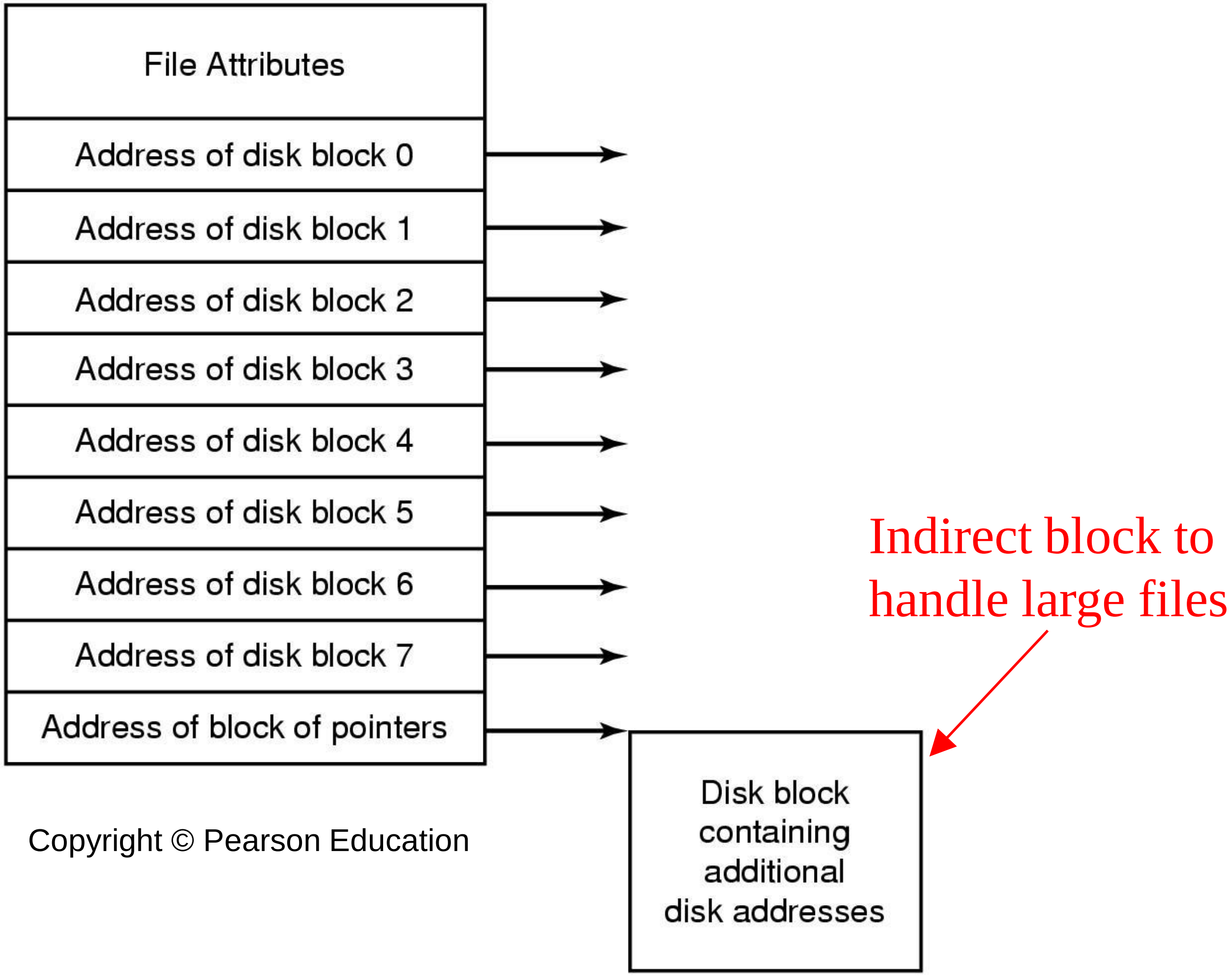
FAT → i-nodes

- Do we actually need to have the whole table in memory all the time?
  - table size proportional to disk size!

- Actually, only open files need to be there…

- Split the table into per-file tables, called *i-nodes* (index node)

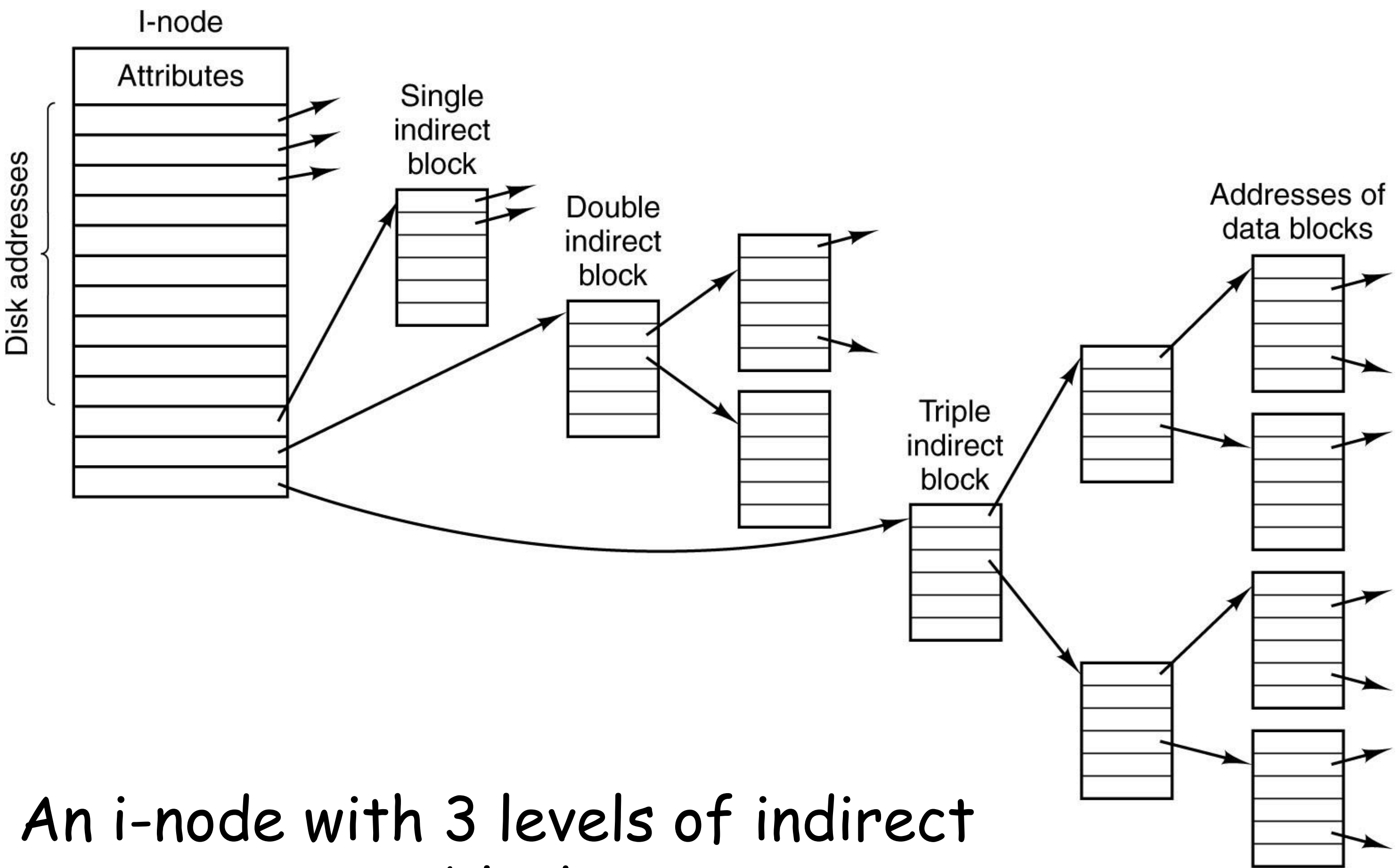
i-node Example



Copyright © Pearson Education

An example i-node

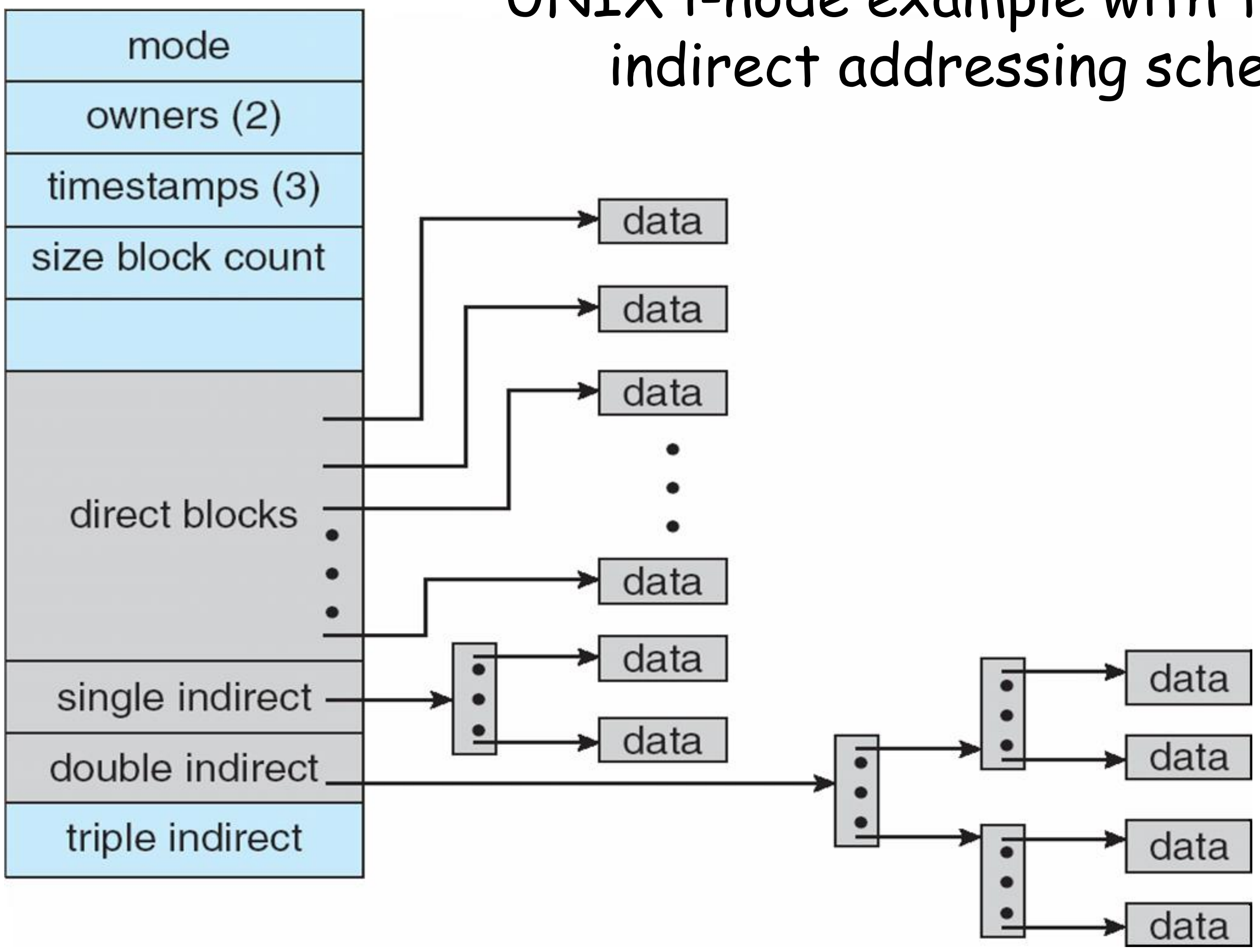
Indirect Addressing



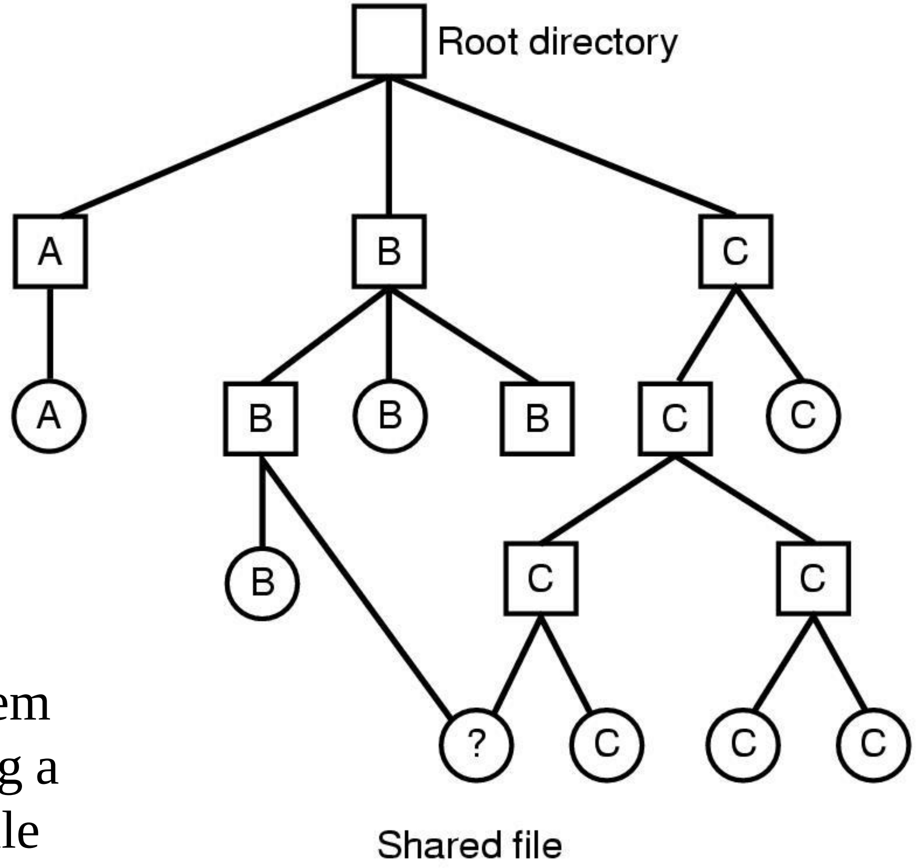
An i-node with 3 levels of indirect blocks

Indirect Addressing

UNIX i-node example with triple indirect addressing scheme



Shared Files



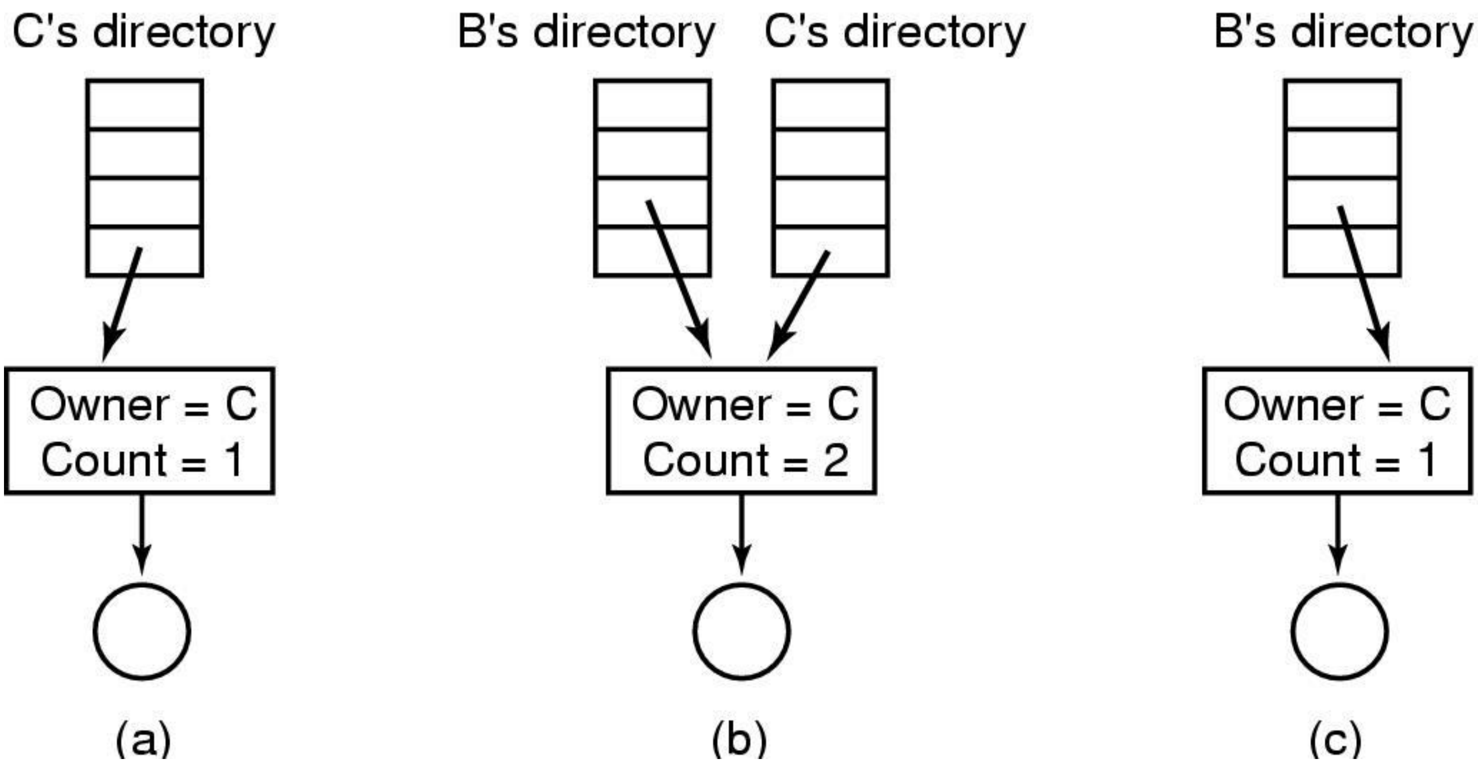
File system containing a shared file

Storing attributes in i-node simplifies sharing

Hard/Symbolic Links

- Hard links are actually the same file
  - share the same i-node
  - will be seen as the same file everywhere
    - same owner
    - same contents
    - same permissions
    - keeps counter
- Symbolic links are dereferenced
  - a special file
    - different owners/permissions
    - can cross filesystem boundaries
  - short cuts in Windows, alias in Mac

Shared Files



(a) Situation prior to linking

(b) After the link is created

(c) After the original owner removes the file

Check this under Linux..

Execute as u1=user1, u2=user2
(make sure that u2 has write permissions)

1. u1: `echo Hi > file-u1`
2. u2: `ln file-u1 file-u2`
3. u2: `ln -s file-u1 file-u2-s`
4. u2: `cat file-u2`
5. u2: `cat file-u2-s`
6. u1: `echo again >> file-u1`
7. u1: `rm file-u1`
8. u2: `cat file-u2`
9. u2: `cat file-u2-s`

What is the output of line 4, 5 & 8, 9? Why?

ext3

- Basic features
  - Native to Linux
  - Variable block size
  - "Related" blocks stored in *Block Groups*
  - Pre-allocates blocks to allow file growth
  - Supports fast symlinks
  - Journaling
  - Online FS growth

Disk vs. Memory Structs

- There needs to be a mapping
  - VFS ↔ disk structures
  - inode ↔ ext3_inode
  - superblock ↔ ext3_super_block
- Most structures are stored in page cache
- Some operations are generic VFS and some ext3-specific

ext3 - inode

```
struct ext3_inode {
        __le16 i_mode;                    /* File mode */
        __le16 i_uid;                     /* Low 16 bits of Owner Uid */
        __le32          i_size;           /* Size in bytes */
        __le32          i_atime;          /* Access time */
        __le32          i_ctime;          /* Creation time */
        __le32          i_mtime;          /* Modification time */
        __le32          i_dtime;          /* Deletion Time */
        __le16 i_gid;                     /* Low 16 bits of Group Id */
        __le16 i_links_count;             /* Links count */
        __le32          i_blocks;         /* Blocks count */
        __le32          i_flags;          /* File flags */
        union { struct { __u32  l_i_reserved1; } linux1;
                struct { __u32  h_i_translator;} hurd1;
                struct { __u32  m_i_reserved1;} masix1; } osd1;      /* OS dependent 1 */
        __le32          i_block[EXT3_N_BLOCKS];/* Pointers to blocks */
        __le32          i_generation;/* File version (for NFS) */
        __le32          i_file_acl;       /* File ACL */
        __le32          i_dir_acl;        /* Directory ACL */
        __le32          i_faddr;          /* Fragment address */
        union { struct {        __u8      l_i_frag;      /* Fragment number */
                                __u8      l_i_fsize;     /* Fragment size */
                                __u16     i_pad1;
                                __le16    l_i_uid_high;/* these 2 fields    */
                                __le16    l_i_gid_high;/* were reserved2[0] */
                                __u32     l_i_reserved2;} linux2;
                struct {        __u8      h_i_frag;      /* Fragment number */
                                __u8      h_i_fsize;     /* Fragment size */
                                __u16     h_i_mode_high;
                                __u16     h_i_uid_high;
                                __u16     h_i_gid_high;
                                __u32     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 */
        __le16 i_extra_isize;
        __le16 i_pad1; };
```

Effective length of file

#blocks allocated to file

Pointer to the blocks

ext3

- ext3 supports the following file types
  - Unknown
  - Regular
  - Directory
    - Stores names and inode numbers in data blocks
  - Character and block devices, FIFOs and sockets
    - Use no data blocks
  - Symbolic links
    - Stores filenames < 60 characters in inode, else in data block
    - Uses the `i_block[EXT3_N_BLOCKS]` field

ext3.h

```
#define EXT3_FT_UNKNOWN       0
#define EXT3_FT_REG_FILE      1
#define EXT3_FT_DIR           2
#define EXT3_FT_CHRDEV        3
#define EXT3_FT_BLKDEV        4
#define EXT3_FT_FIFO          5
#define EXT3_FT_SOCK          6
#define EXT3_FT_SYMLINK       7
```

ext3 – How to Find a Block

Finding the block number within a file is simple:

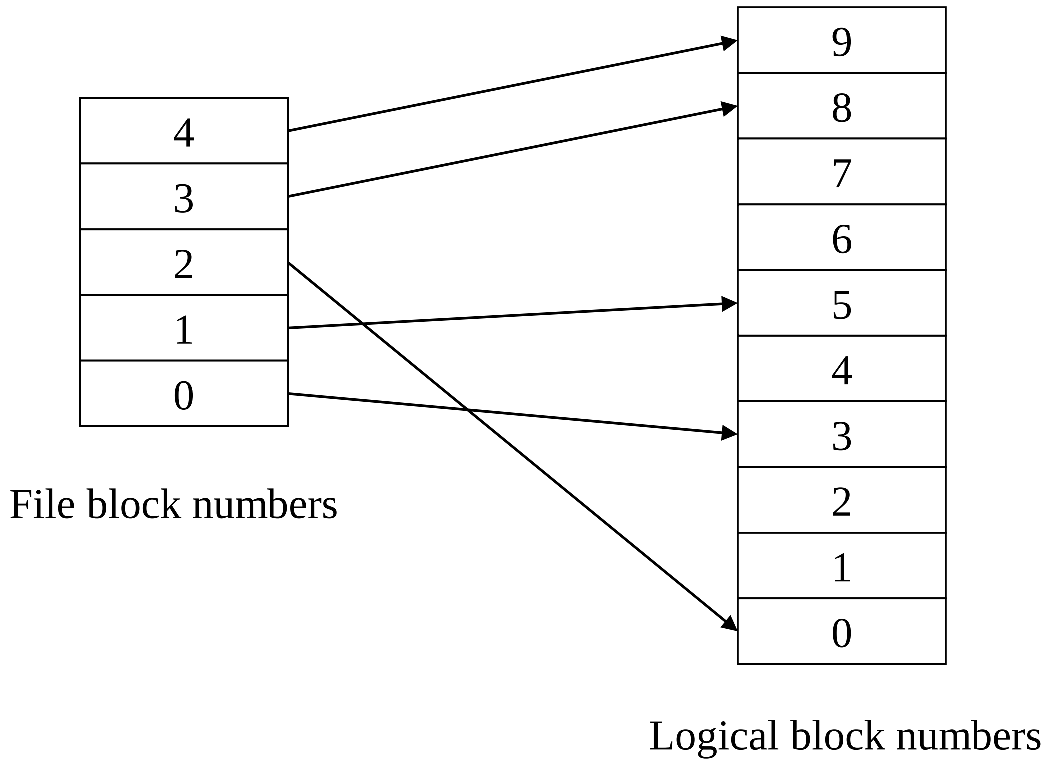
f **div** b

where b is the block size, f is the position in the file

The f^{th} character is in the f **mod** b position in the block

ext3 – How to Find a Block

- Blocks on disk and blocks in files are not the same



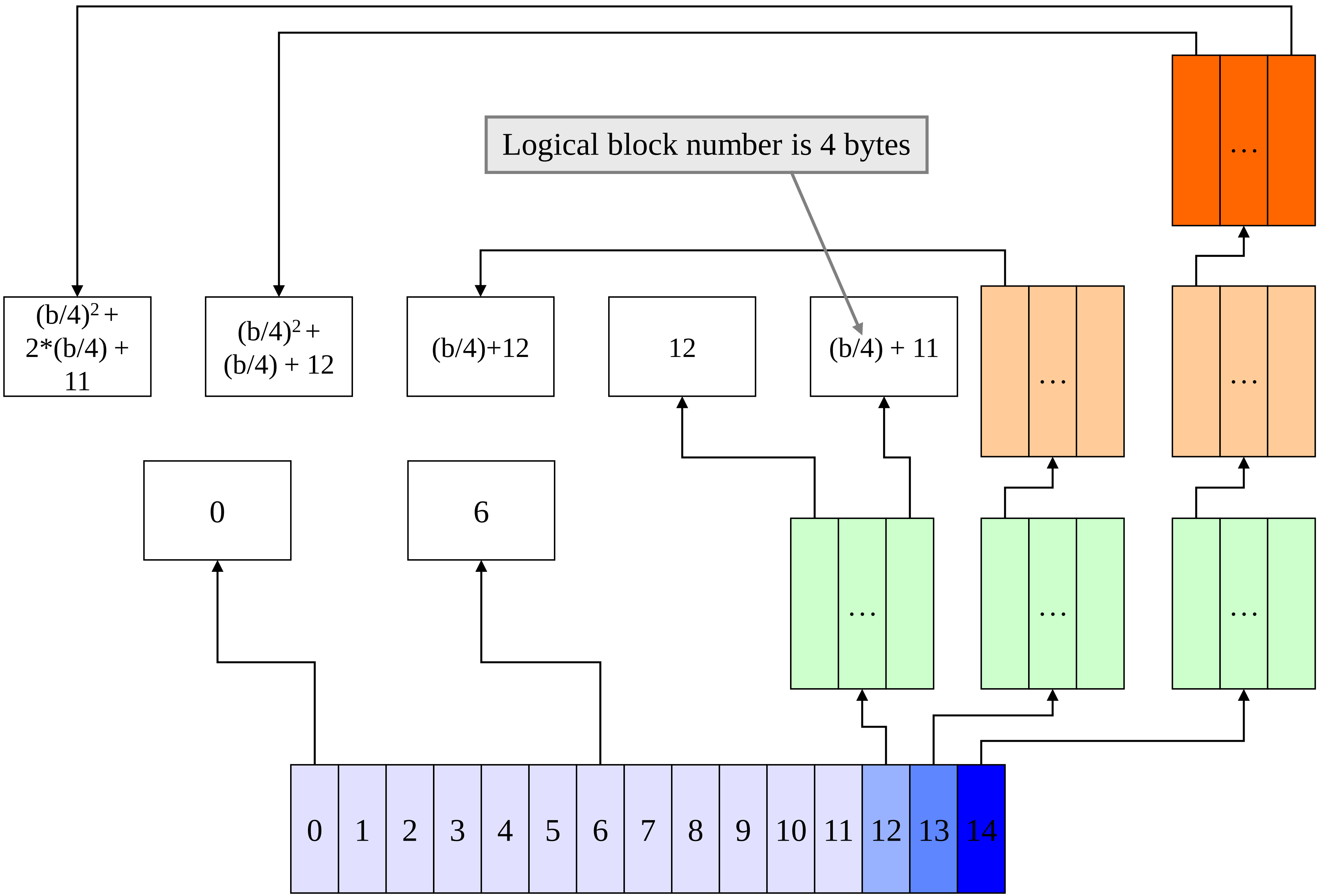
ext3 – How to Find a Block (2)

- File block → disk block mapping is stored in the inode
- Remember the pointer array

```
__u32 i_block[EXT3_N_BLOCKS]; /* Pointers to blocks */
```

- Usually fifteen 32-bits words used as indexes

ext3 – How to Find a Block



ext4 Extents (1)

- Indirect addressing inefficient for large files
  - E.g., multimedia content or large databases
  - Deletion takes long, due to reading (and freeing) all the single/double/triple indirect blocks
  - Real life tells: strictly increasing block numbers are very common

- Superior approach: extents
  - Used in ext4, btrfs, NTFS, HPFS, others
  - An extent is a contiguous amount of blocks that's reserved after file creation
  - Less metadata than with indirect addressing
  - Performance boost for sequential data access
  - Individual extents form a tree, see next slide

ext4 Extents (2)

- ext4_extent structure: 12 bytes
  - max FS size 1 EiB (48 bit physical block number)
  - max extent 128 MiB (16 bit extent length, 15 bits for address, MSB used as initialization flag)
  - max file size 16 TiB (32 bit logical number)
- Up to four ext4_extent data structures stored directly in inode
  - More than four: b-tree is created, see below

node header

eh_depth > 0 → ext4_extent_idx

eh_depth = 0 → ext4_extent

```
struct ext4_extent_header {
        __le16      eh_magic;       /* probably will support different formats */
        __le16      eh_entries;     /* number of valid entries */
        __le16      eh_max;         /* capacity of store in entries */
        __le16      eh_depth;       /* has tree real underlying blocks? */
        __le32      eh_generation;/* generation of the tree */
};
```

points to more headers

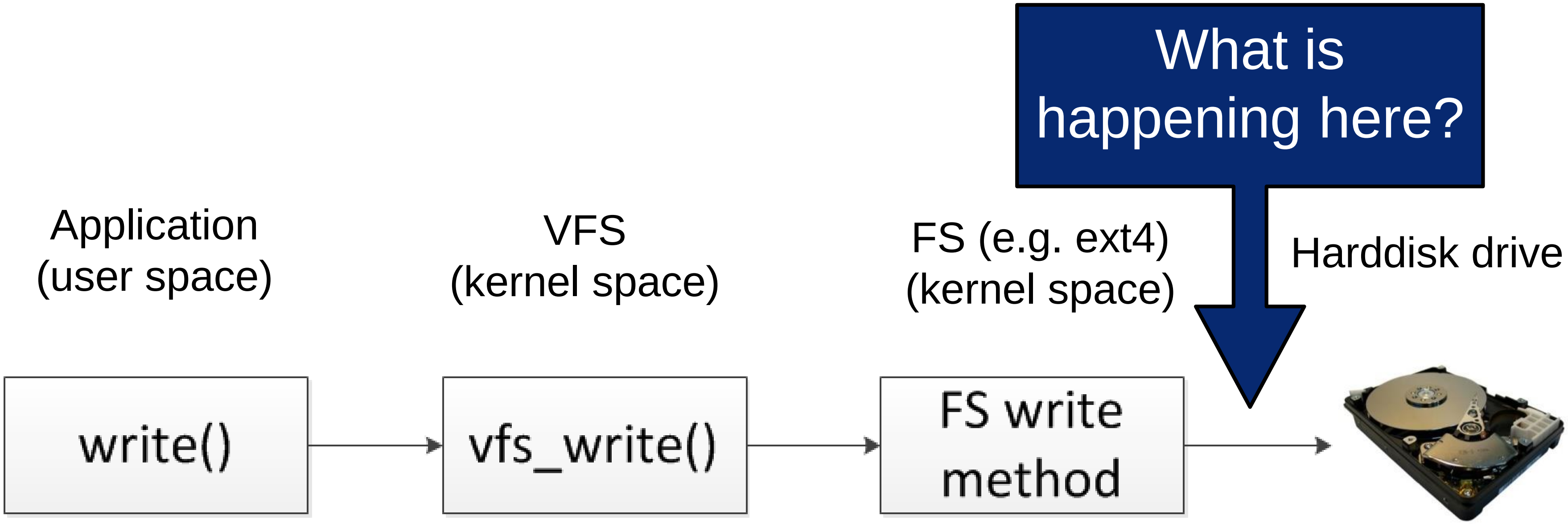
```
struct ext4_extent_idx {
        __le32      ei_block;       /* index covers logical blocks from 'block' */
        __le32      ei_leaf_lo;     /* pointer to the physical block of the next level. leaf or next index could be there */
        __le16      ei_leaf_hi;     /* high 16 bits of physical block */
        __u16       ei_unused;
};
```

points to actual data

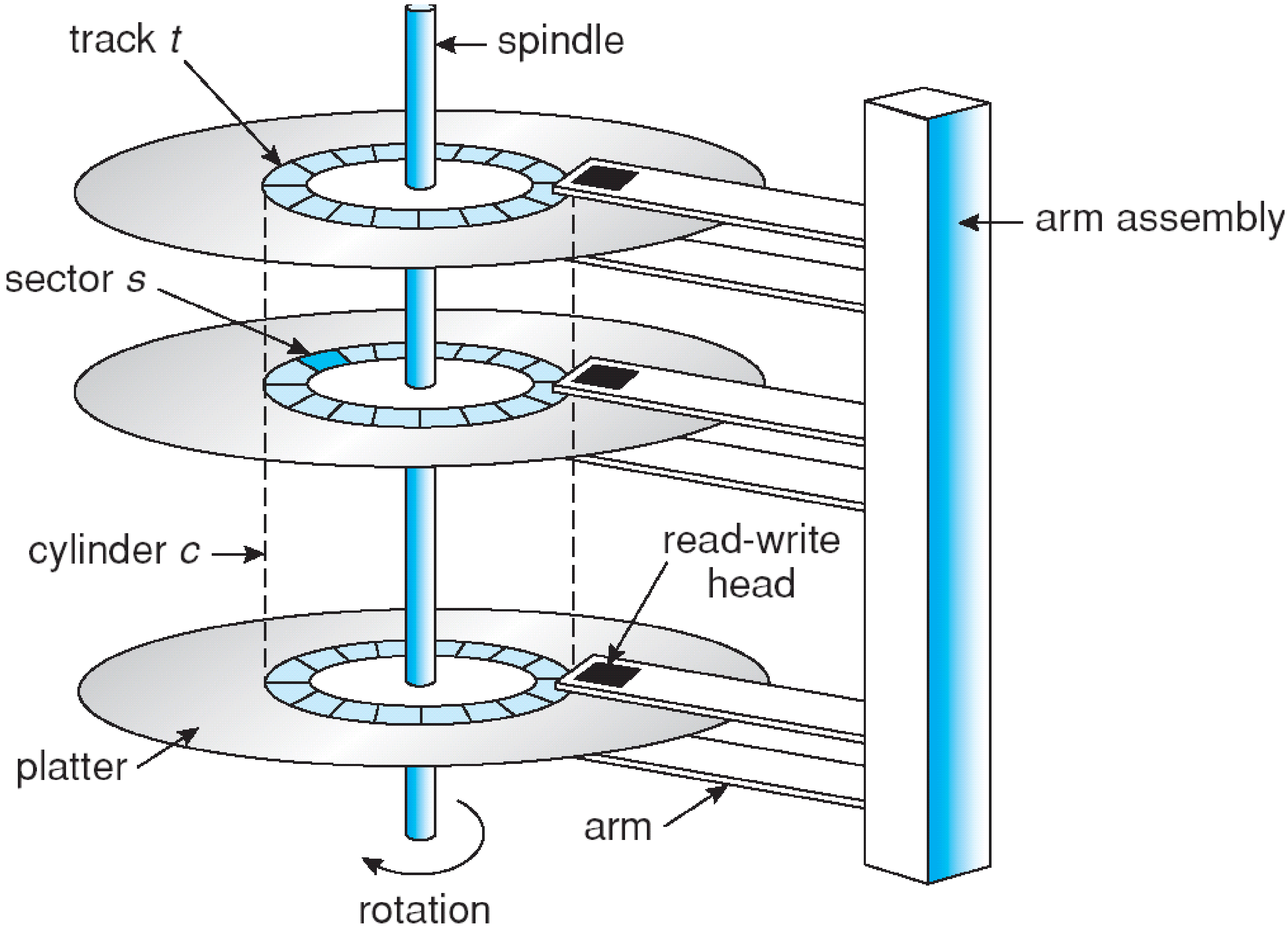
```
struct ext4_extent {
        __le32      ee_block;       /* first logical block extent covers */
        __le16      ee_len;         /* number of blocks covered by extent */
        __le16      ee_start_hi;    /* high 16 bits of physical block */
        __le32      ee_start_lo;    /* low 32 bits of physical block */
};
```

Disk Scheduling

- OS's responsibility to schedule hardware efficiently
- Efficiency characteristics for disk I/O
  - Fast access time
  - Large disk bandwidth



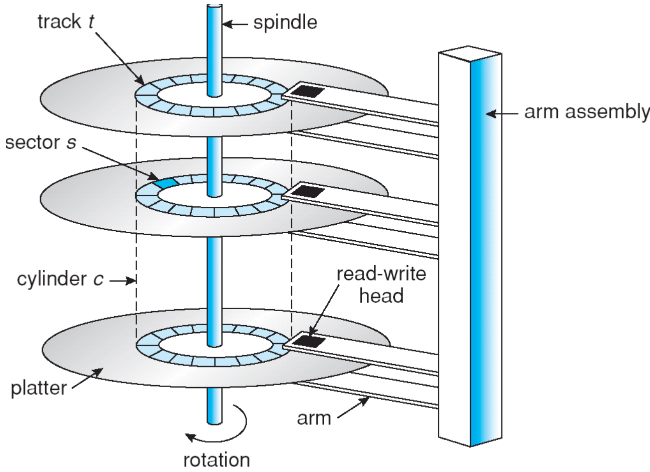
Disk Scheduling (2)



Disk Scheduling (3)

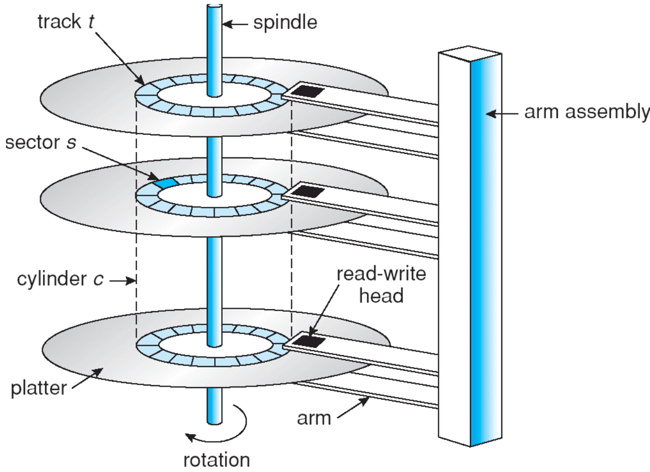
- Access time has two major components
  - Seek time – time for disk arm to move heads to the cylinder containing the desired sector (to be minimized)
  - Rotational latency – additional time for the disk to rotate the desired sector to the disk head
- Disk bandwidth
  - Total number of bytes (not blocks!) transferred, divided by the total time between the first request for service and completion of last transfer

- Disk I/O request of a process:
  - Input or output
  - Disk address
  - Memory address for transfer
  - Number of sectors to be transferred



Disk Scheduling (4)

- Servicing requests
  - Disk drive and controller available? → service immediately
  - Multiprogramming, multi-user systems?
    - → Disk scheduling algorithms



Disk Scheduling (5)

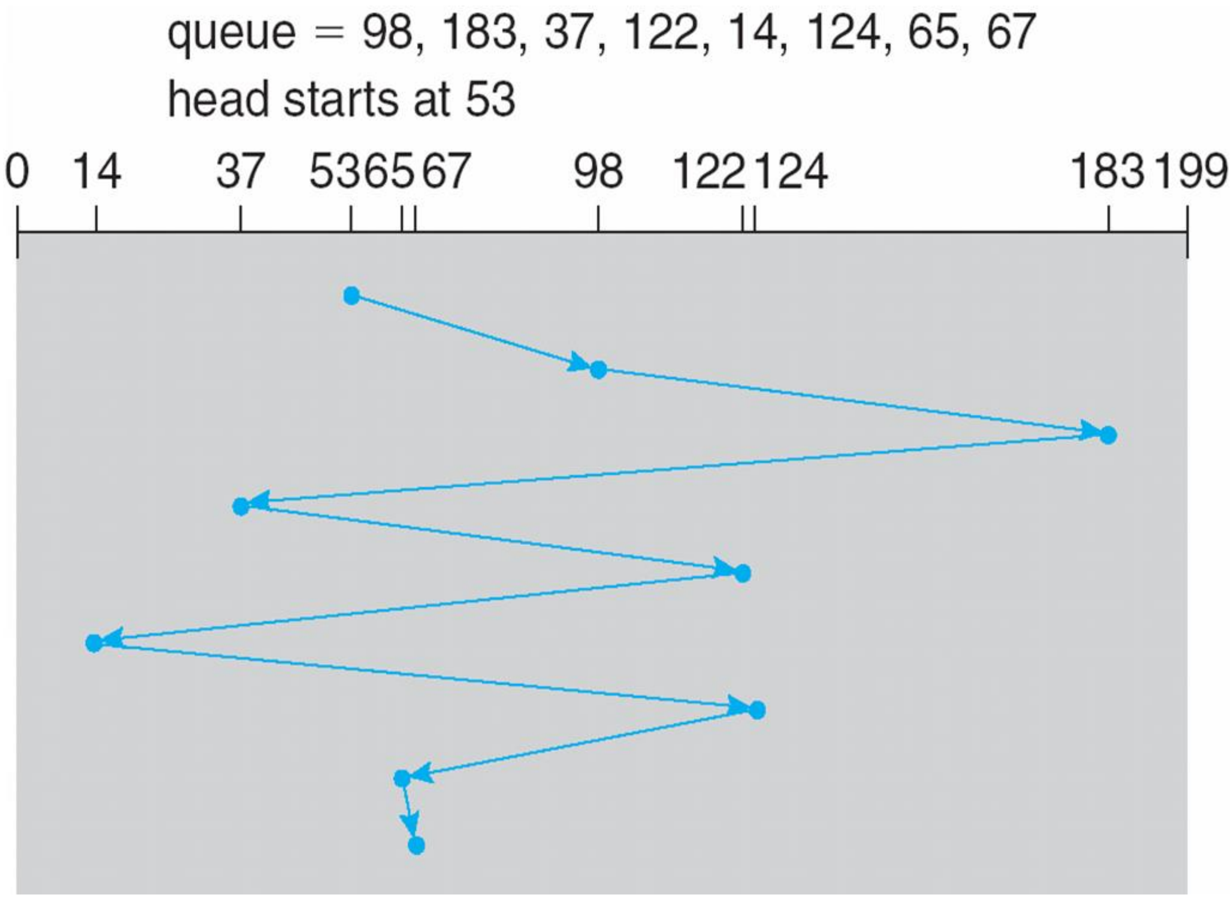
- Several algorithms exist to schedule the servicing of disk I/O requests.
- We illustrate them with a request queue (0-199).

98, 183, 37, 122, 14, 124, 65, 67

Head pointer 53

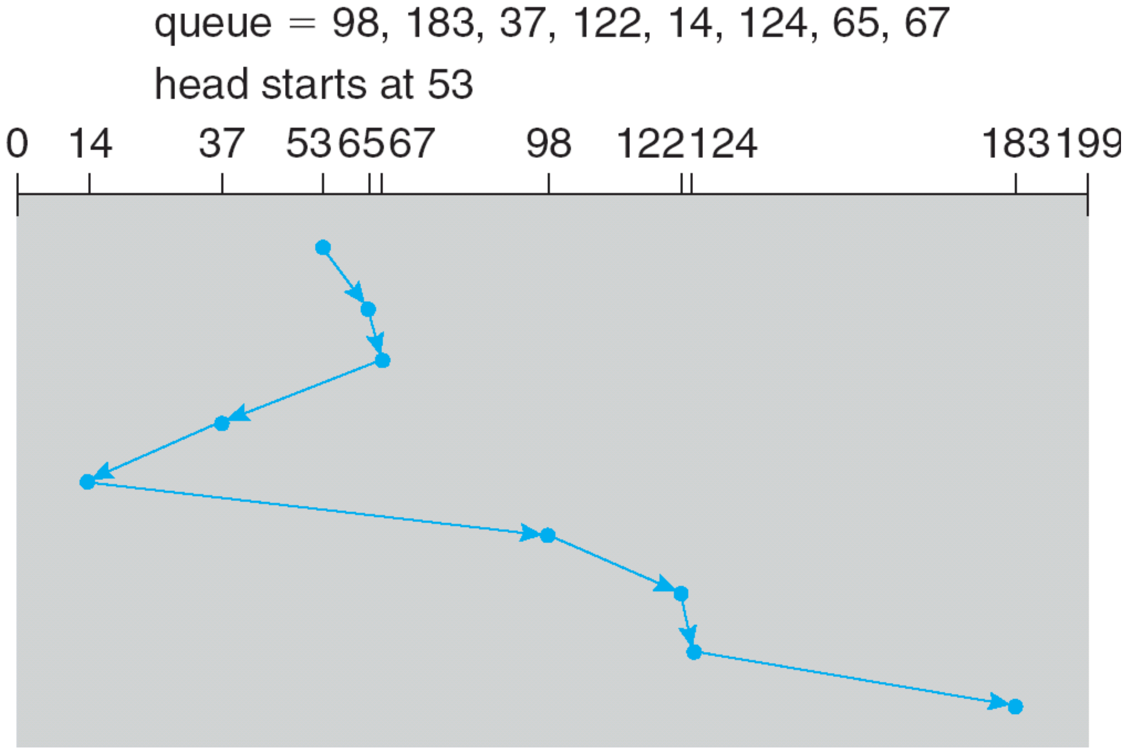
Disk Scheduling: FCFS

Illustration shows total head movement of 640 cylinders.



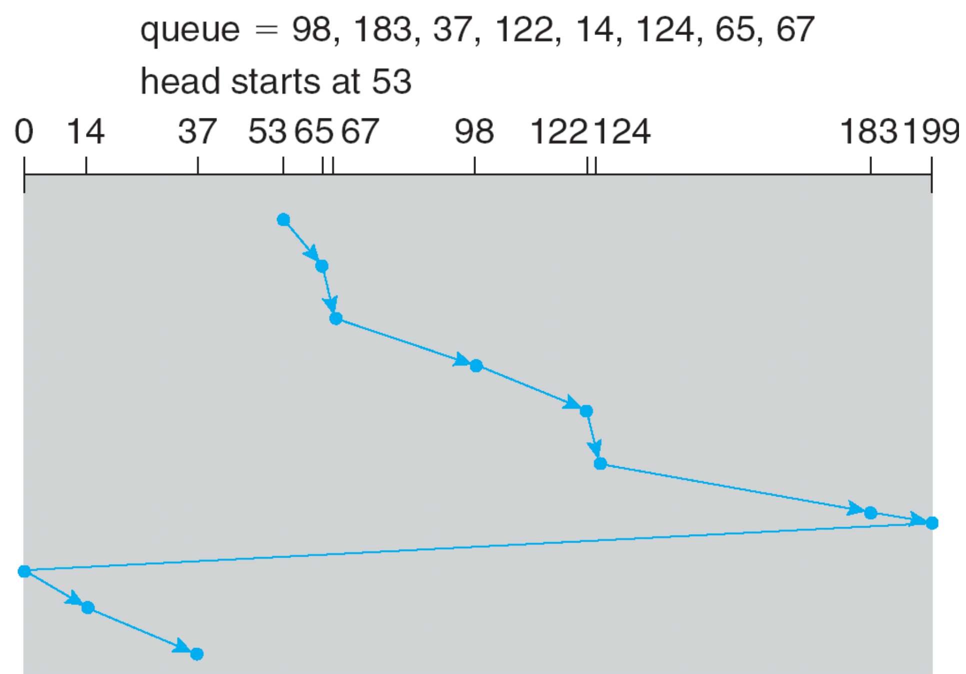
Disk Scheduling: SSTF

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Illustration shows total head movement of 236 cylinders.



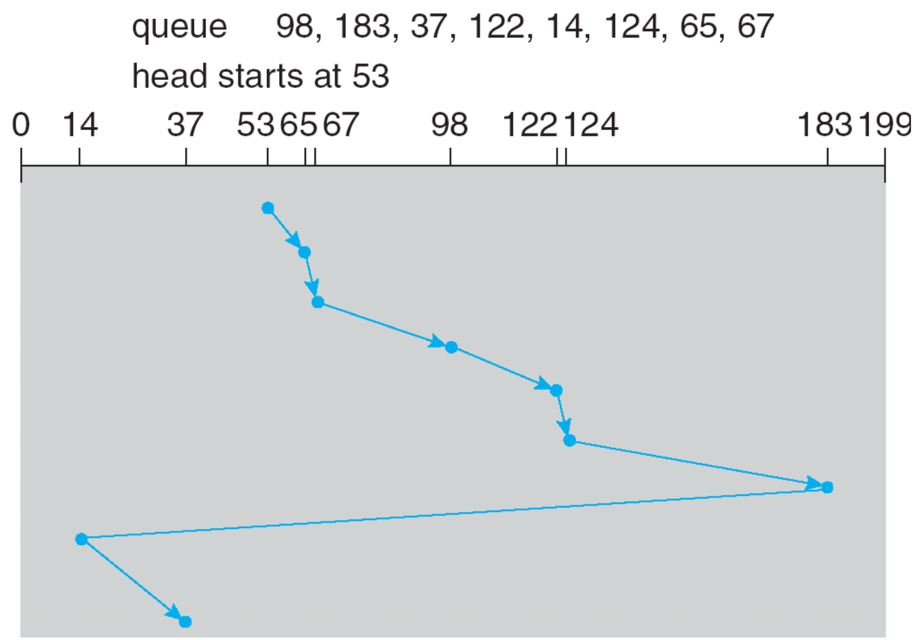
Disk Scheduling: SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the *elevator algorithm*.
- Illustration shows total head movement of 208 cylinders.



Disk Scheduling: C-LOOK

- Version of SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.



EOF ;-)

Tomorrow: exercise session on I/O and file systems!