File System

Sherif Khattab

http://www.cs.pitt.edu/~skhattab/cs1550

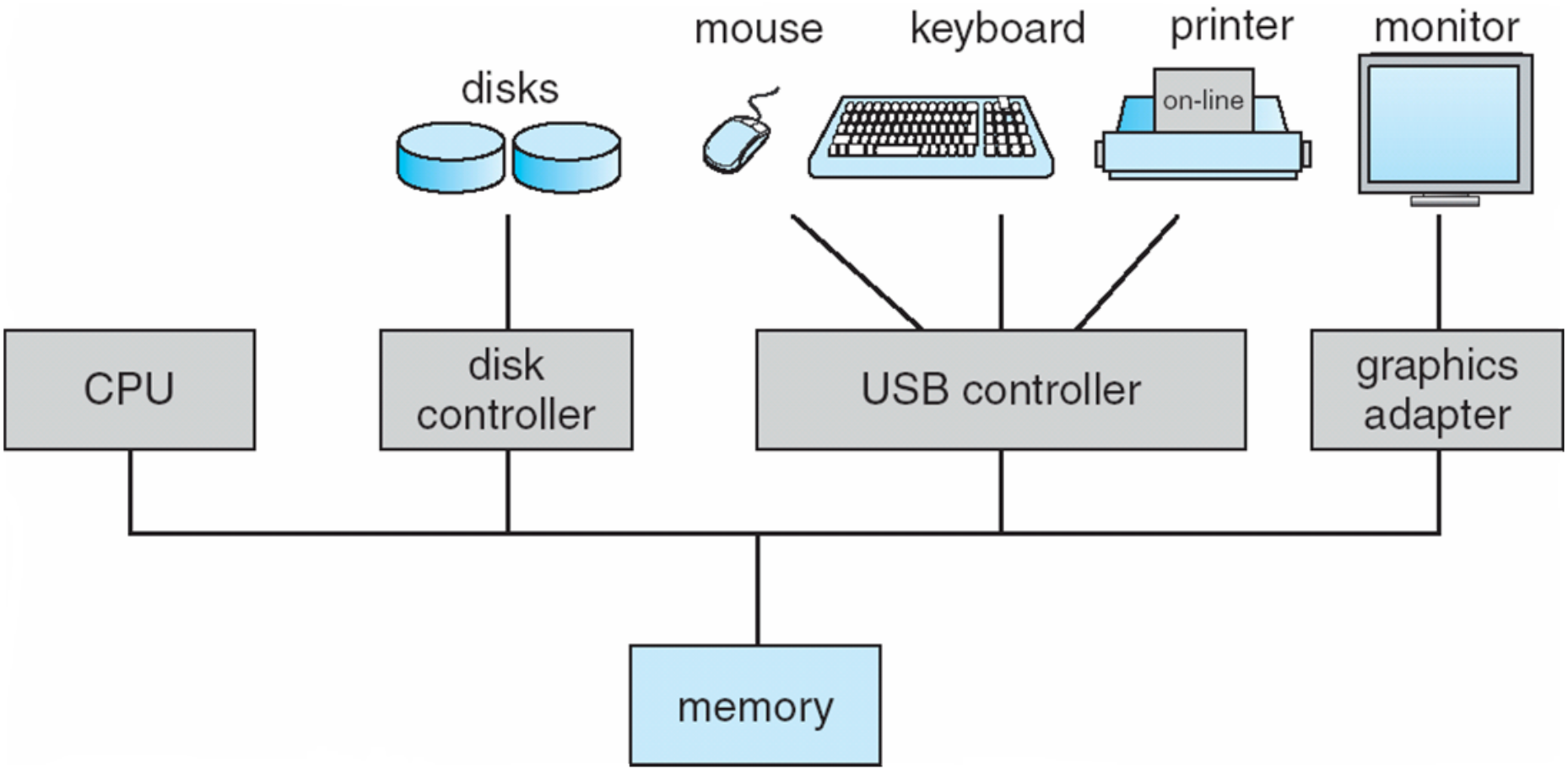
(slides are from **Dr. Ahmed Amer's CS 1550 Slides)**

Outline

- Disk I/O
- File System

How complex is the OS's job?

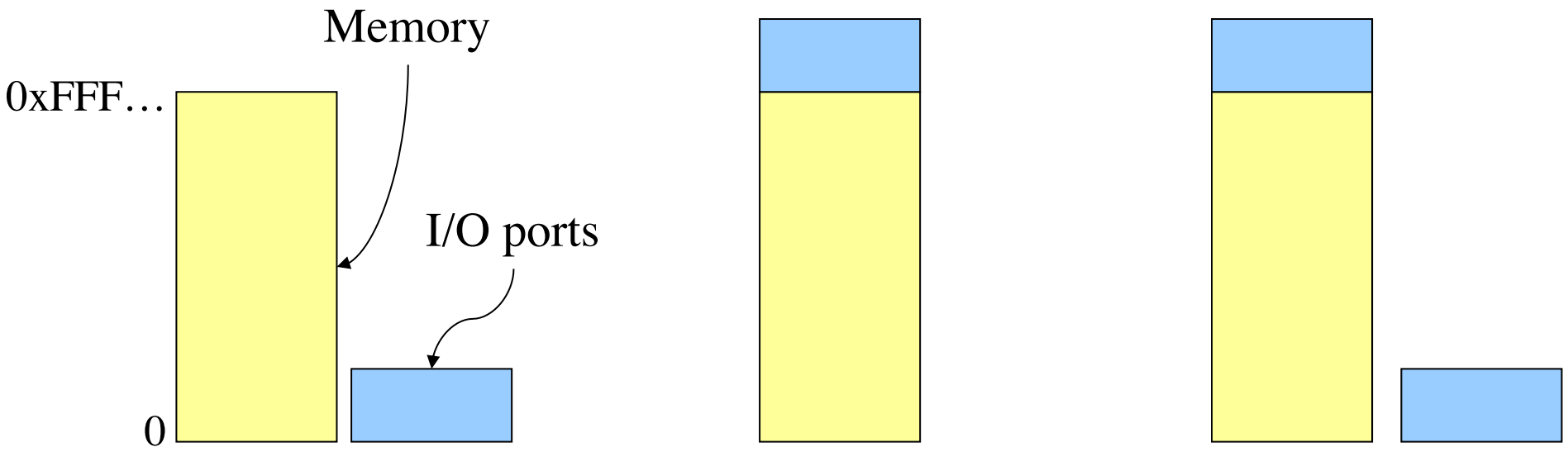
- Let's look at one of the resources managed by the OS: I/O devices



Device controllers

- Electronic component controls the device
  - May be able to handle multiple devices
  - May be more than one controller per mechanical component (example: hard drive)
- Controller's tasks
  - Convert serial bit stream to block of bytes
  - Perform error correction as necessary
  - Make available to main memory
- Block Devices
  - A device that stores data in fixed-sized blocks, each uniquely addressed, and can be randomly accessed
- Character Devices
  - Device that delivers or accepts a stream of characters

Memory-Mapped I/O



Separate I/O & memory space

Memory-mapped I/O

Hybrid: both memory-mapped & separate spaces

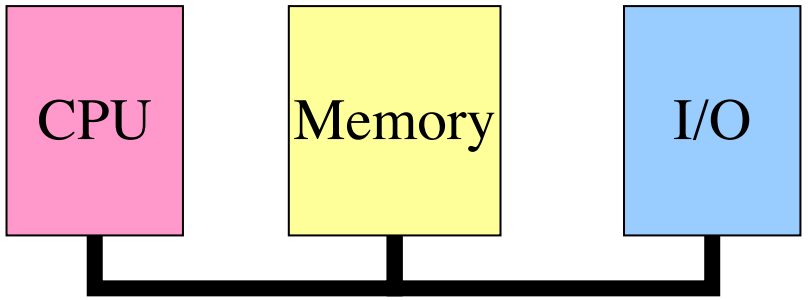
Dynamic Frequency on XScale

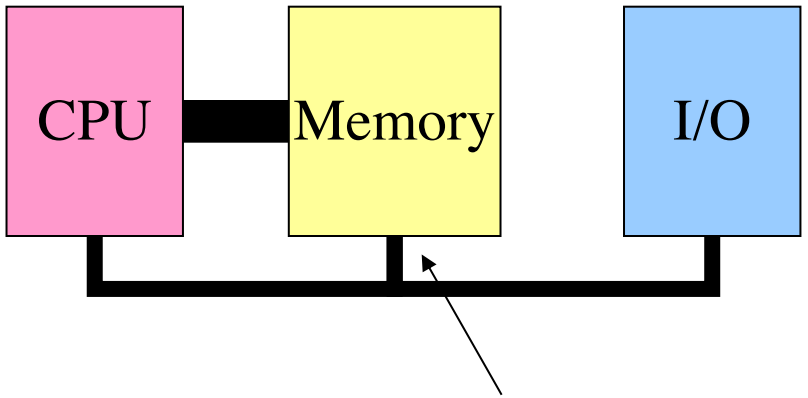
```
// Allocate some space for the virtual reference to CCCR
LPVOID virtCCCR = VirtualAlloc(0, sizeof(DWORD), MEM_RESERVE, PAGE_NOACCESS);

//0x41300000 is the memory-mapped location of the CCCR register
LPVOID CCCR = (LPVOID)(0x41300000 / 256); // shift by 8 bits for ability to address 2^40 bytes

// Map writing the virtual pointer to the physical address of the CCCR register
VirtualCopy((LPVOID)virtCCCR, CCCR, sizeof(DWORD), PAGE_READWRITE | PAGE_NOCACHE | PAGE_PHYSICAL);

// Set the CCCR register with the new speed
*(int *)virtCCCR = new_speed;

// Call the assembly function to actually perform the switch
doSwitch(0x02 | 0x01); //0x02 means turbo mode, 0x01 means the clock is being switched

// Clean up memory by freeing the virtual register.
VirtualFree(virtCCCR, 0, MEM_RELEASE);
virtCCCR = NULL;
```

```
; Coprocessor 14, register C6 (CLKCFG) initiates the changes programmed in CCCR
; when CLKCFG is written.

doSwitch
      MOV r3, r0                ; Move r0, the argument to doSwitch, into register r3
      MCR p14, 0, r3, c6, c0, 0 ; Copy the contents of r3 into register c6 on coprocessor 14.
      MOV pc, lr                ; return execution to where it last left off
```

How is memory-mapped I/O done?

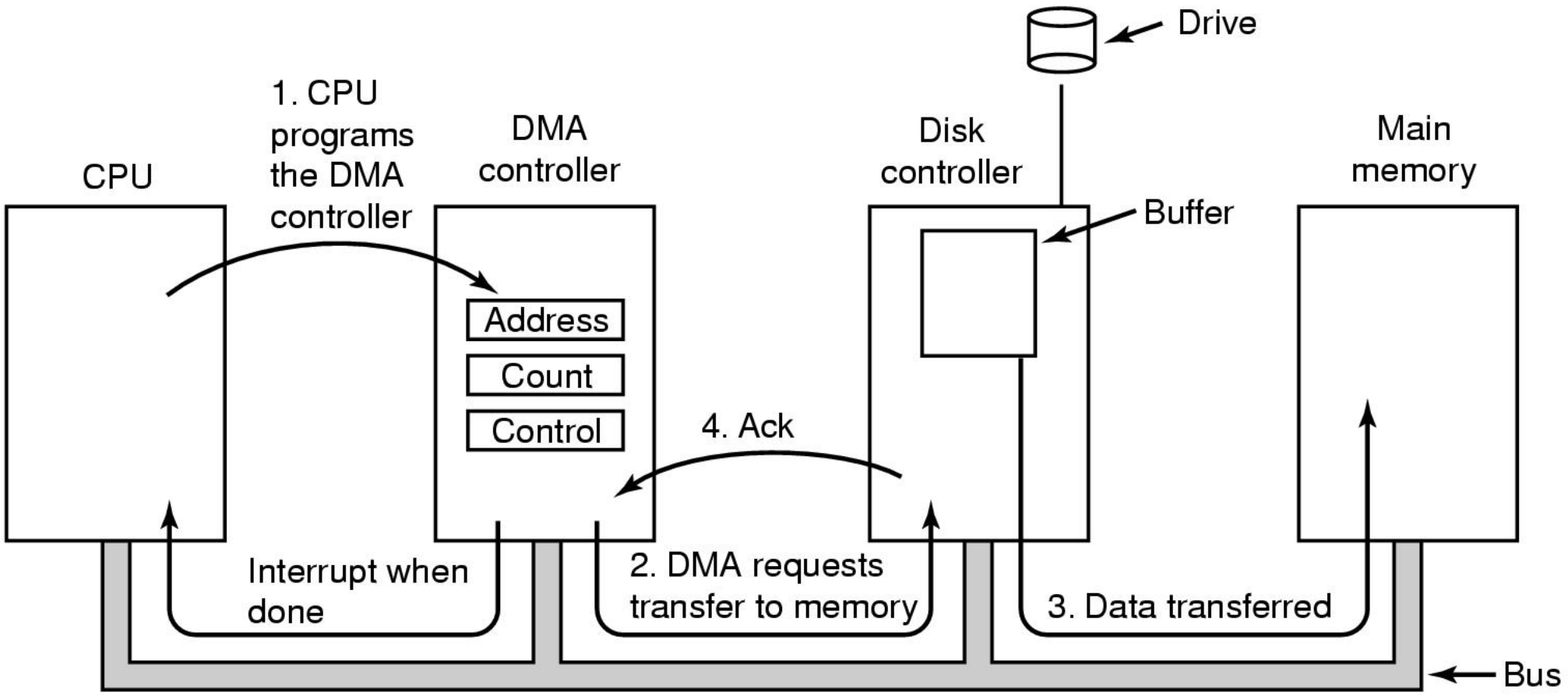
- Single-bus
  - All memory accesses go over a shared bus
  - I/O and RAM accesses compete for bandwidth
- Dual-bus
  - RAM access over high-speed bus
  - I/O access over lower-speed bus
  - Less competition
  - More hardware (more expensive…)



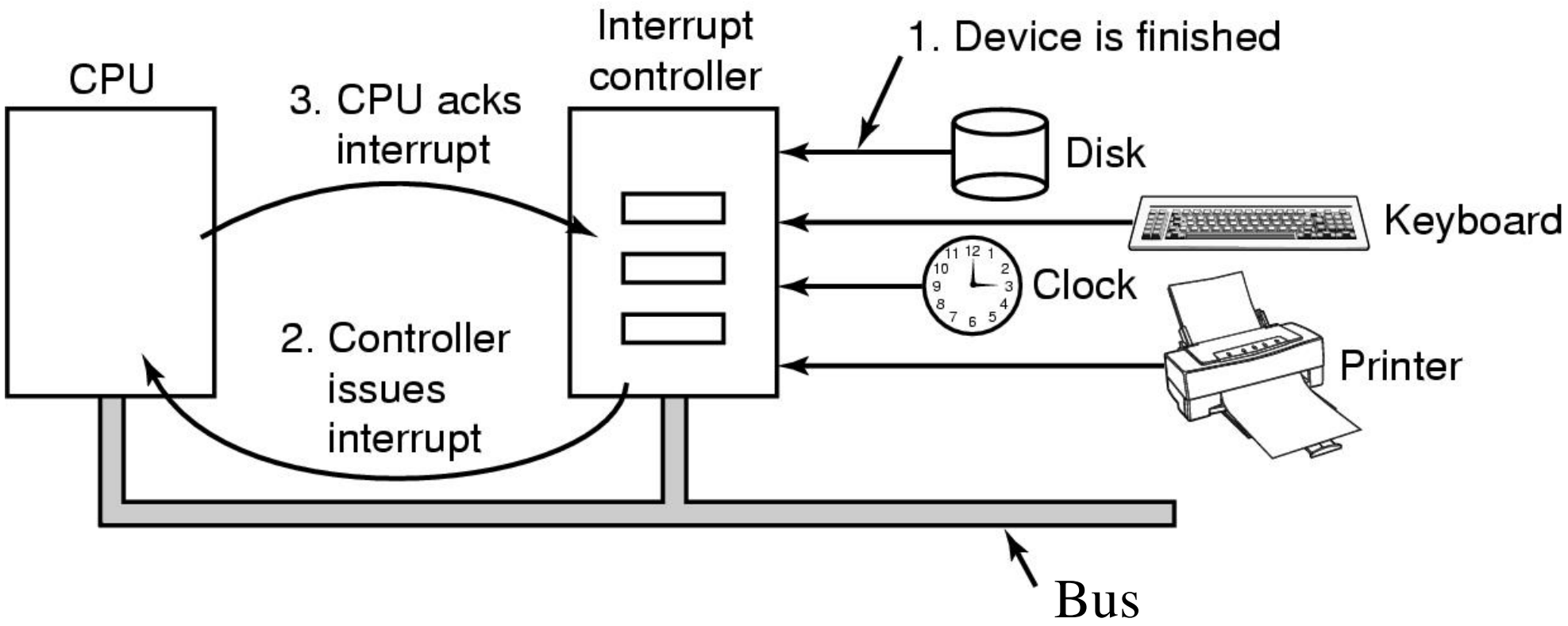


This port allows I/O devices access into memory

Direct Memory Access (DMA) operation



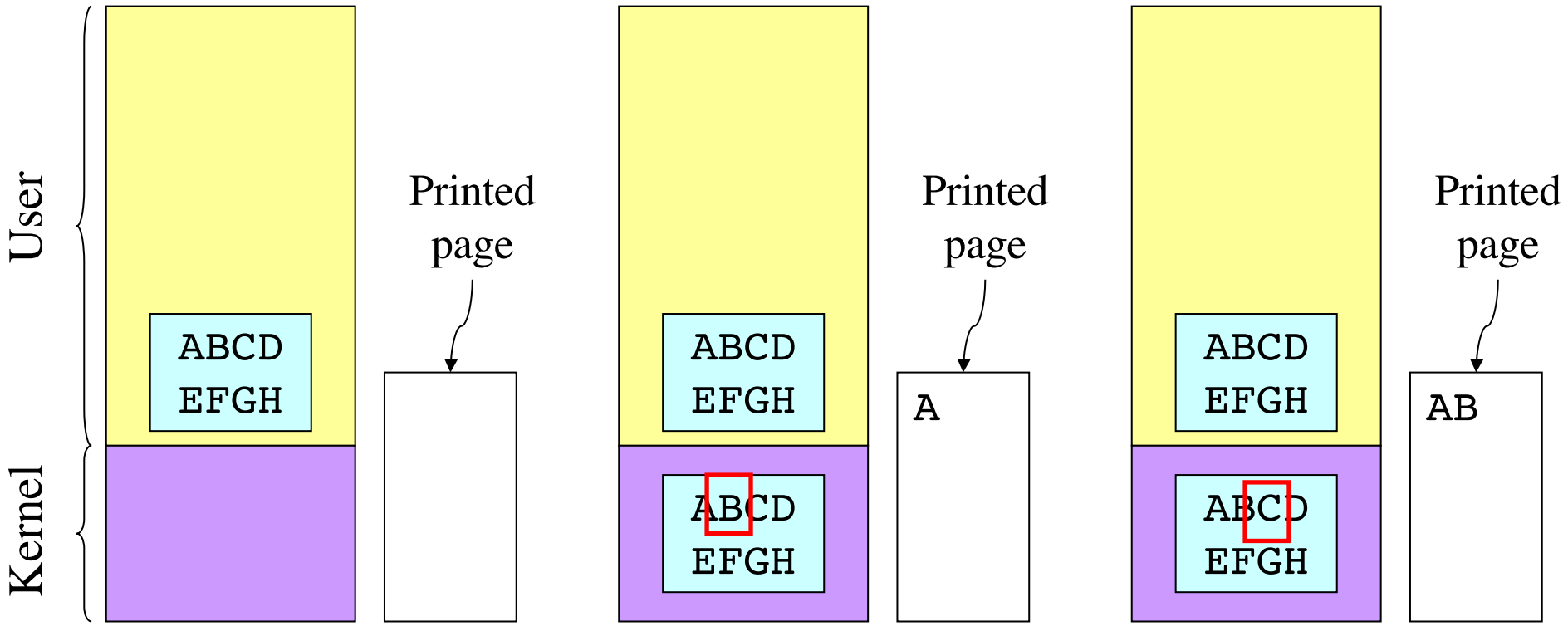
Hardware's view of interrupts



I/O software goals

- Device independence
  - Programs can access any I/O device
  - No need to specify device in advance
- Uniform naming
  - Name of a file or device is a string or an integer
  - Doesn't depend on the machine (underlying hardware)
- Error handling
  - Done as close to the hardware as possible
  - Isolate from higher-level software
- Synchronous vs. asynchronous transfers
  - Blocked transfers vs. interrupt-driven
- Buffering
  - Data coming off a device cannot be stored in final destination
- Sharable vs. dedicated devices

Programmed I/O: printing a page



Polling

```
copy_from_user (buffer, p, count);  // copy into kernel buffer
for (j = 0; j < count; j++) {       // loop for each char
  while (*printer_status_reg != READY)
    ;                               // wait for printer to be ready
  *printer_data_reg = p[j];         // output a single character
}
return_to_user();
```

Interrupt-driven I/O

```
copy_from_user (buffer, p, count);
j = 0;
enable_interrupts();
while (*printer_status_reg != READY)
  ;
*printer_data_reg = p[0];
scheduler(); // and block user
```

Code run by system call

```
if (count == 0) {
  unblock_user();
} else {
  *printer_data_reg = p[j];
  count--;
  j++;
}
acknowledge_interrupt();
return_from_interrupt();
```

Code run at interrupt time
(Interrupt handler)

I/O using DMA

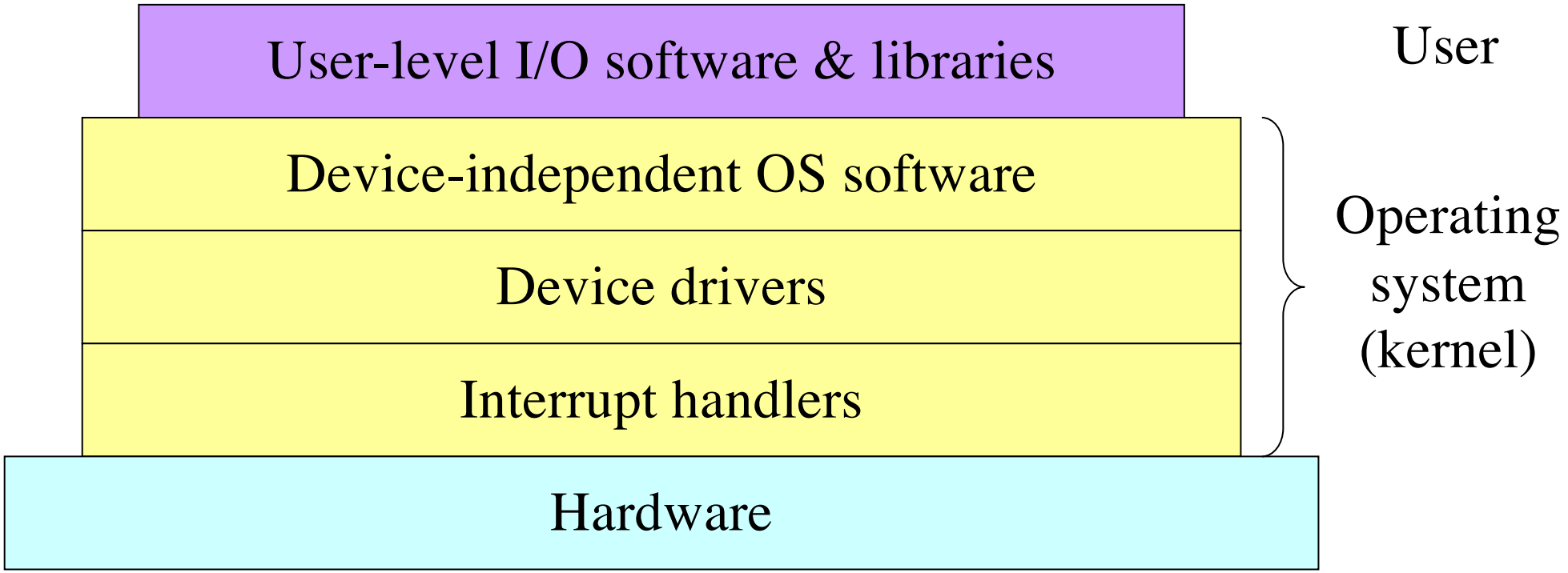
```
copy_from_user (buffer, p, count);
set_up_DMA_controller();
scheduler(); // and block user
```

Code run by system call

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Code run at interrupt time

Layers of I/O software

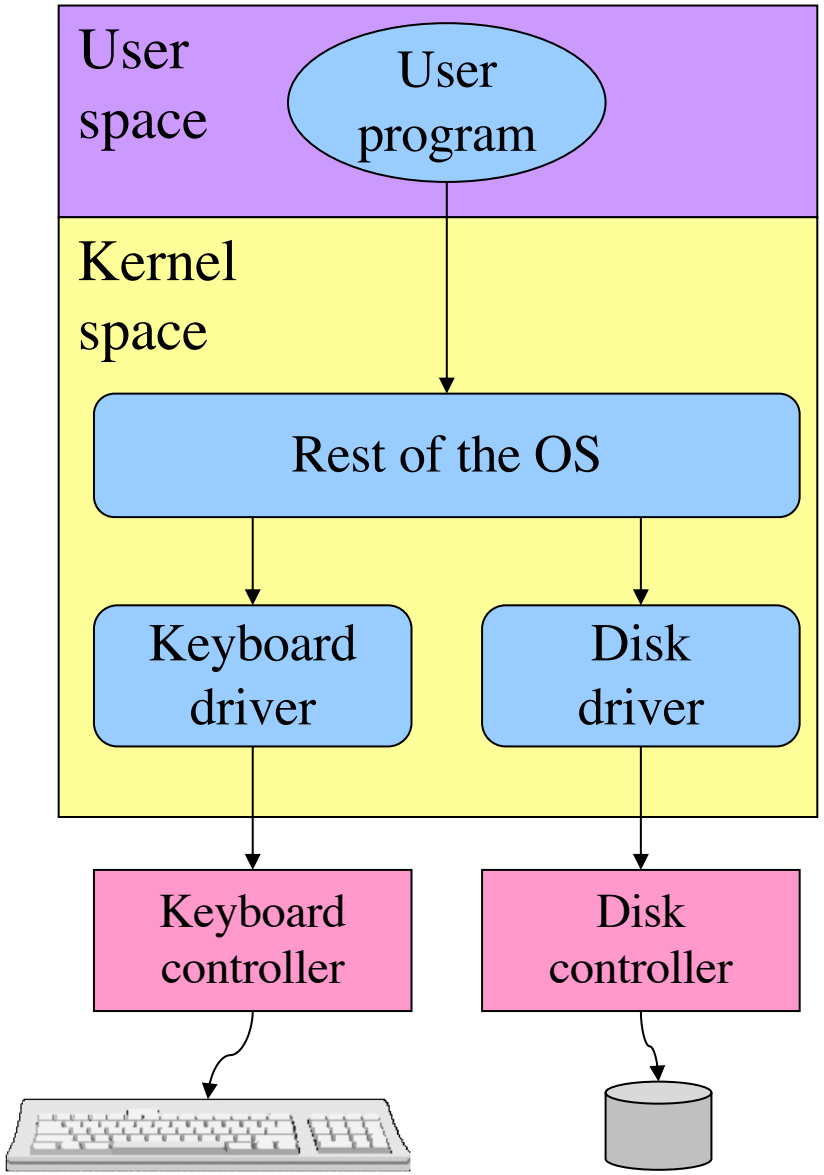


What happens on an interrupt

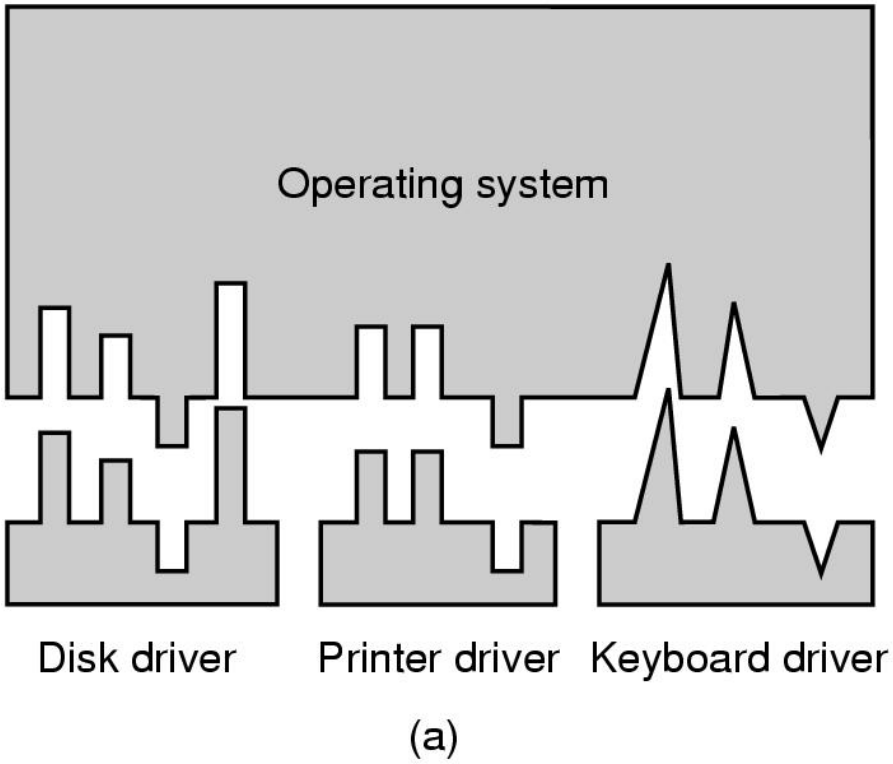
- Set up stack for interrupt service procedure
- Ack interrupt controller, reenable interrupts
- Copy registers from where saved
- Run service procedure
- (optional) Pick a new process to run next
- Set up MMU context for process to run next
- Load new process' registers
- Start running the new process

Device drivers

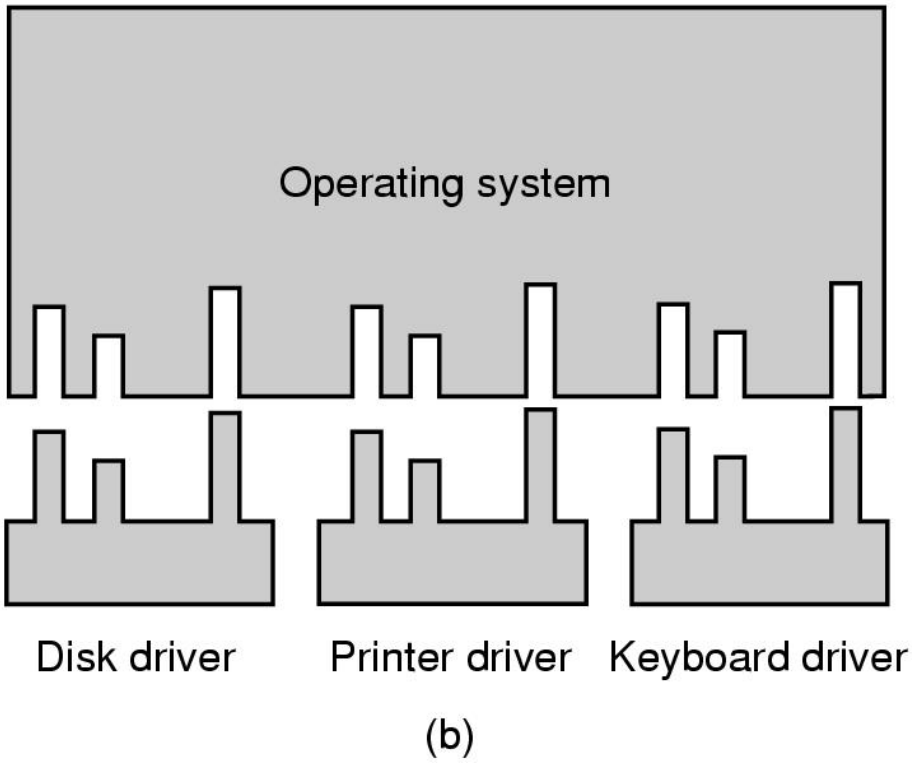
- Device drivers go between device controllers and rest of OS
  - Drivers standardize interface to widely varied devices
- Device drivers communicate with controllers over bus
  - Controllers communicate with devices themselves



Why a standard driver interface?

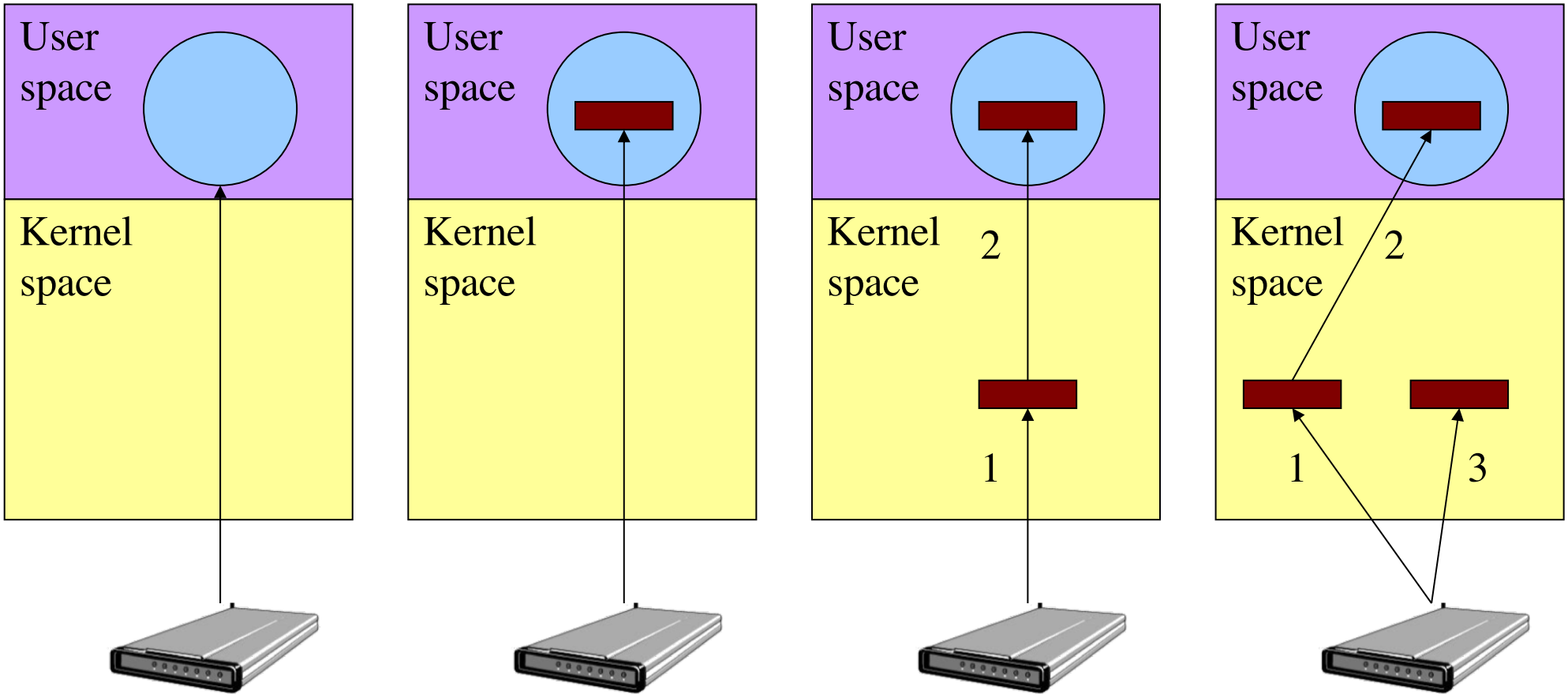


Non-standard driver interfaces



Standard driver interfaces

Buffering device input



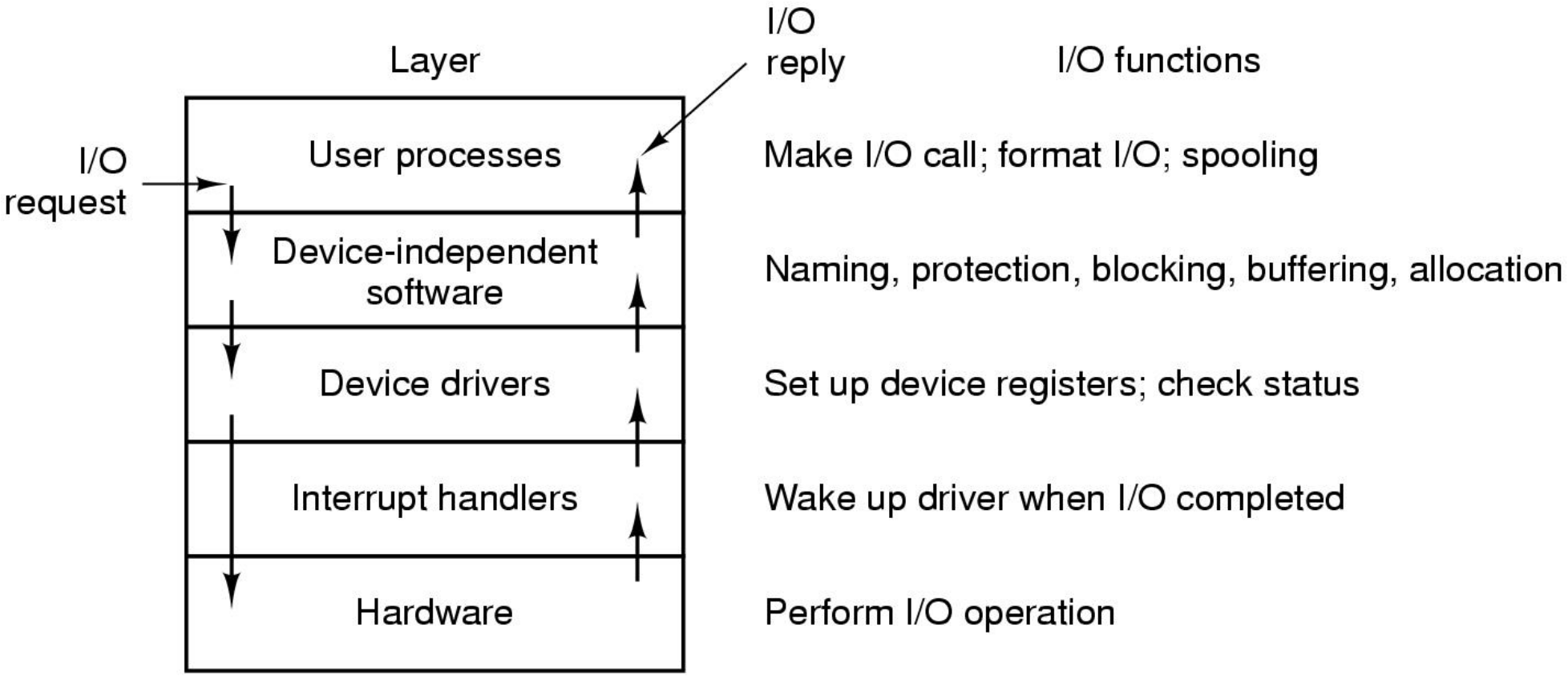
Unbuffered input

Buffering in user space

Buffer in kernel
Copy to user space

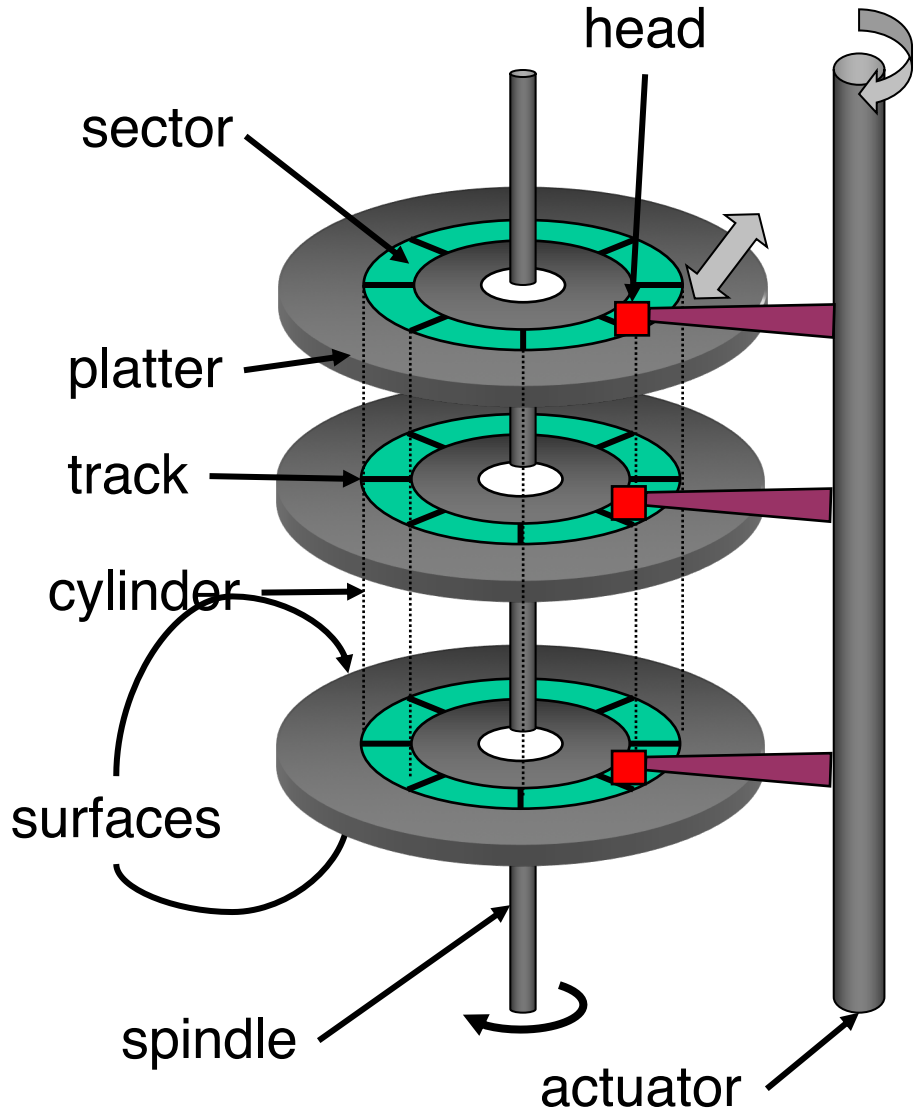
Double buffer in kernel

Anatomy of an I/O request



Disk drive structure

- Data stored on surfaces
  - Up to two surfaces per platter
  - One or more platters per disk
- Data in concentric tracks
  - Tracks broken into sectors
    - 256B-1KB per sector
  - Cylinder: corresponding tracks on all surfaces
- Data read and written by heads
  - Actuator moves heads
  - Heads move in unison

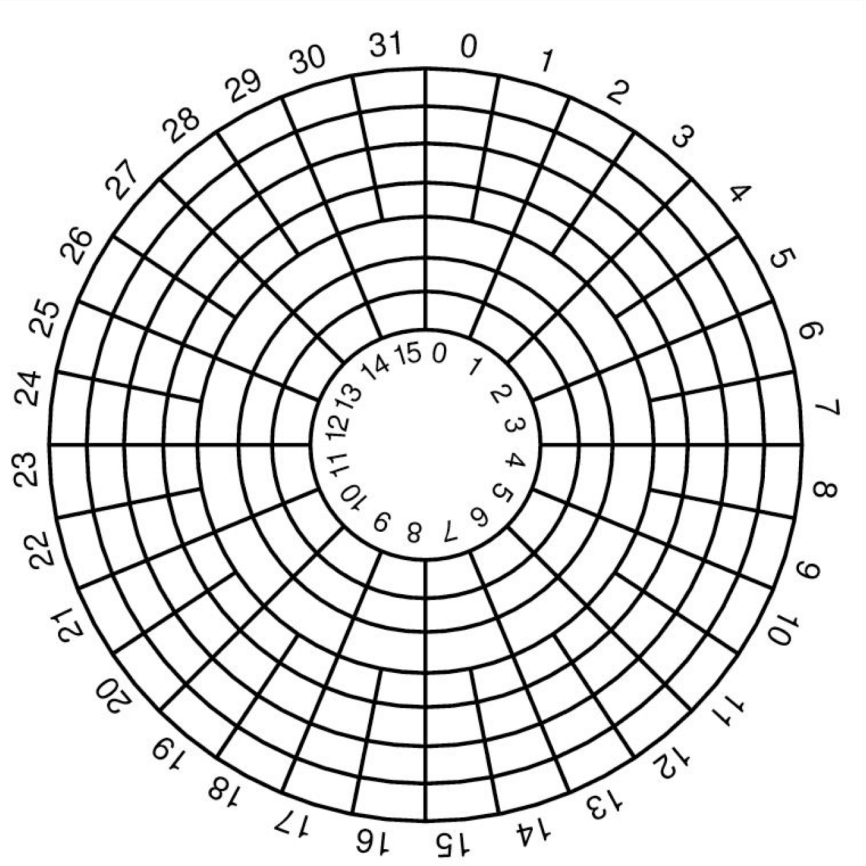


Disk drive specifics

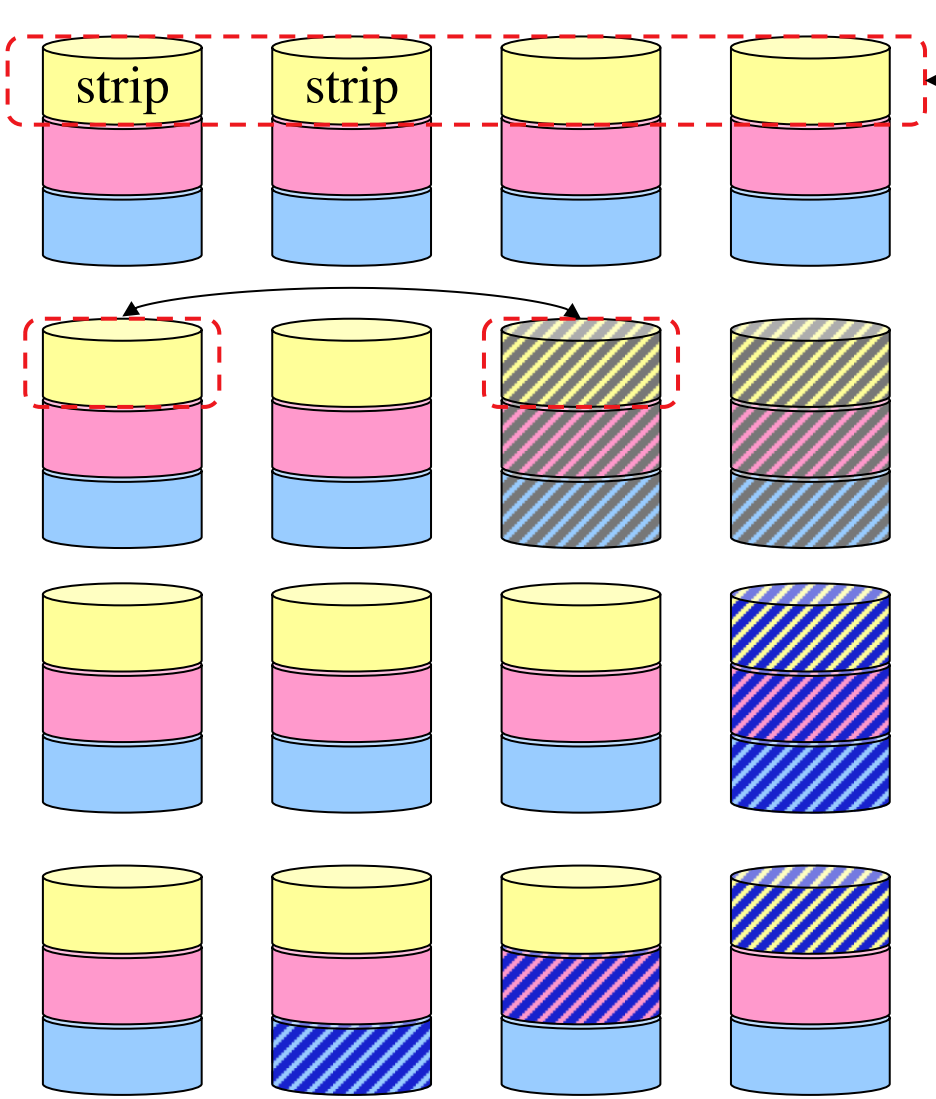
	IBM 360KB floppy	WD 18GB HD
Cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (average)
Sectors per disk	720	35742000
Bytes per sector	512	512
Capacity	360 KB	18.3 GB
Seek time (minimum)	6 ms	0.8 ms
Seek time (average)	77 ms	6.9 ms
Rotation time	200 ms	8.33 ms
Spinup time	250 ms	20 sec
Sector transfer time	22 ms	17 μsec

Disk “zones”

- Outside tracks are longer than inside tracks
- Two options
  - Bits are “bigger”
  - More bits (transfer faster)
- Modern hard drives use second option
  - More data on outer tracks
- Disk divided into “zones”
  - Constant sectors per track in each zone
  - 8–20 (or more) zones on a disk



RAIDs, RAIDs, and more RAIDs



RAID 0
(Redundant Array of Inexpensive Disks

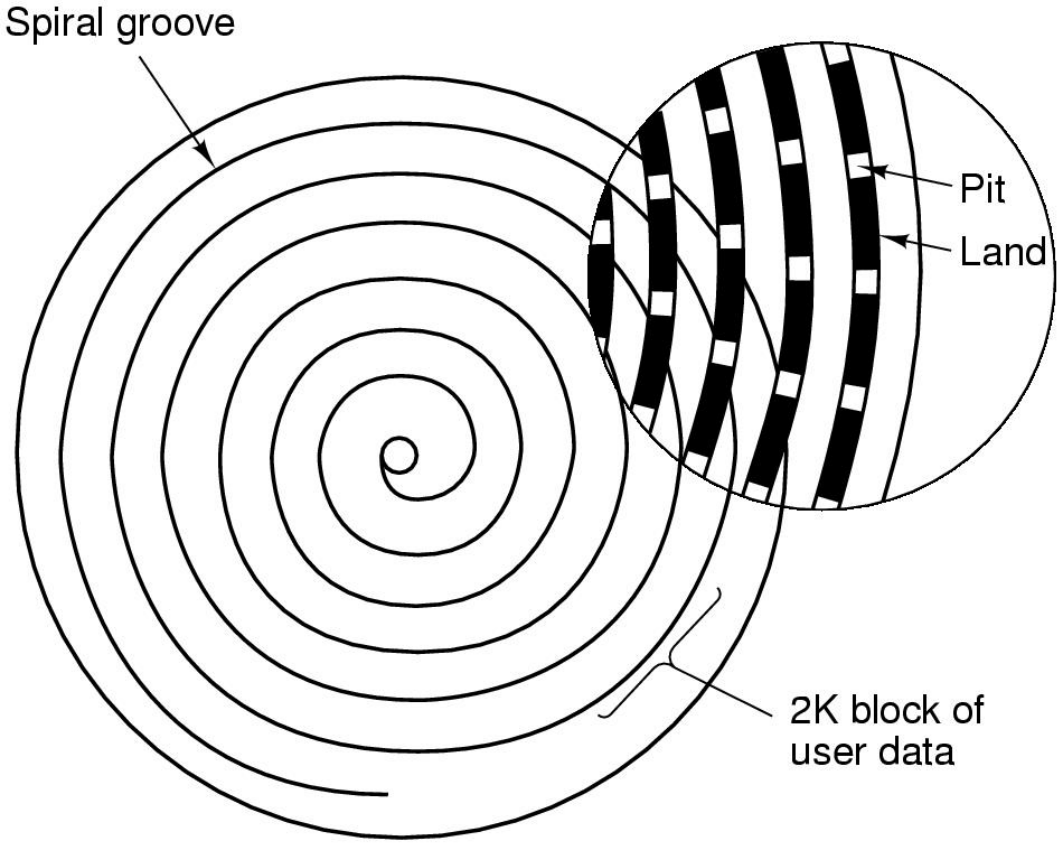
RAID 1
(Mirrored copies)

RAID 4
(Striped with parity)

RAID 5
(Parity rotates through disks)

CD-ROM recording

- CD-ROM has data in a spiral
  - Hard drives have concentric circles of data
- One continuous track: just like vinyl records!
- Pits & lands “simulated” with heat-sensitive material on CD-Rs and CD-RWs

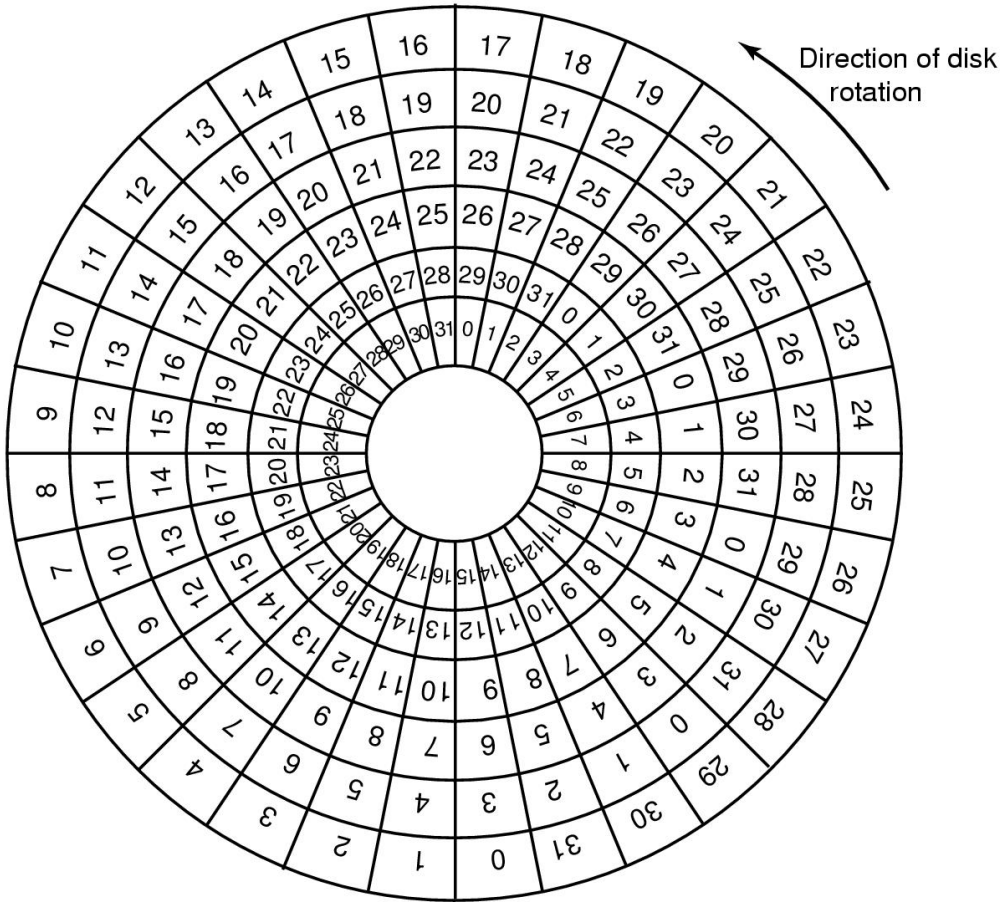


Structure of a disk sector

- Preamble contains information about the sector
  - Sector number & location information
- Data is usually 256, 512, or 1024 bytes
- ECC (Error Correcting Code) is used to detect & correct minor errors in the data

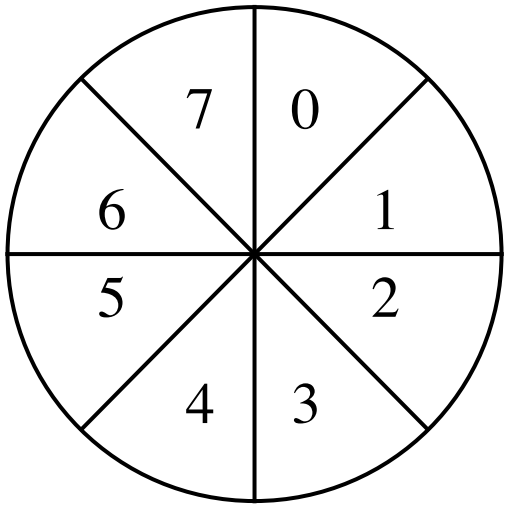
Sector Skew

- Sectors numbered sequentially on each track
- Numbering starts in different place on each track: *sector skew*
  - Allows time for switching head from track to track
- All done to minimize delay in sequential transfers

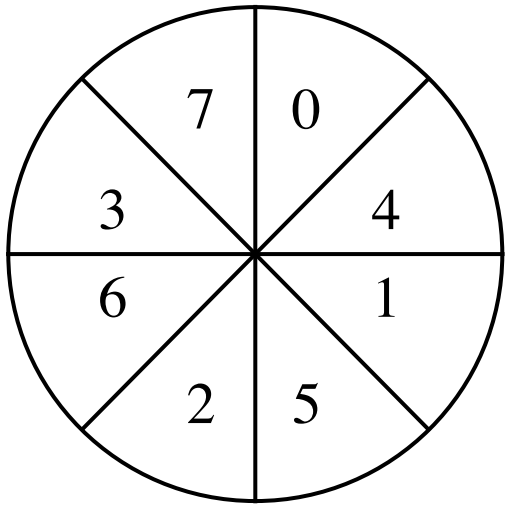


Sector interleaving

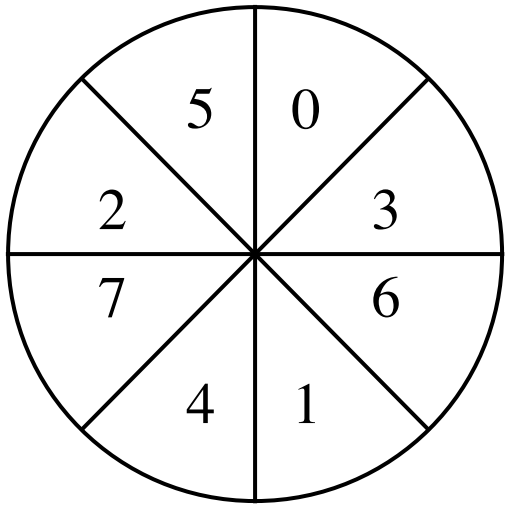
- On older systems, the CPU was slow => time elapsed between reading consecutive sectors
- Solution: leave space between consecutively numbered sectors
- This isn't done much these days…



No interleaving



Skipping 1 sector



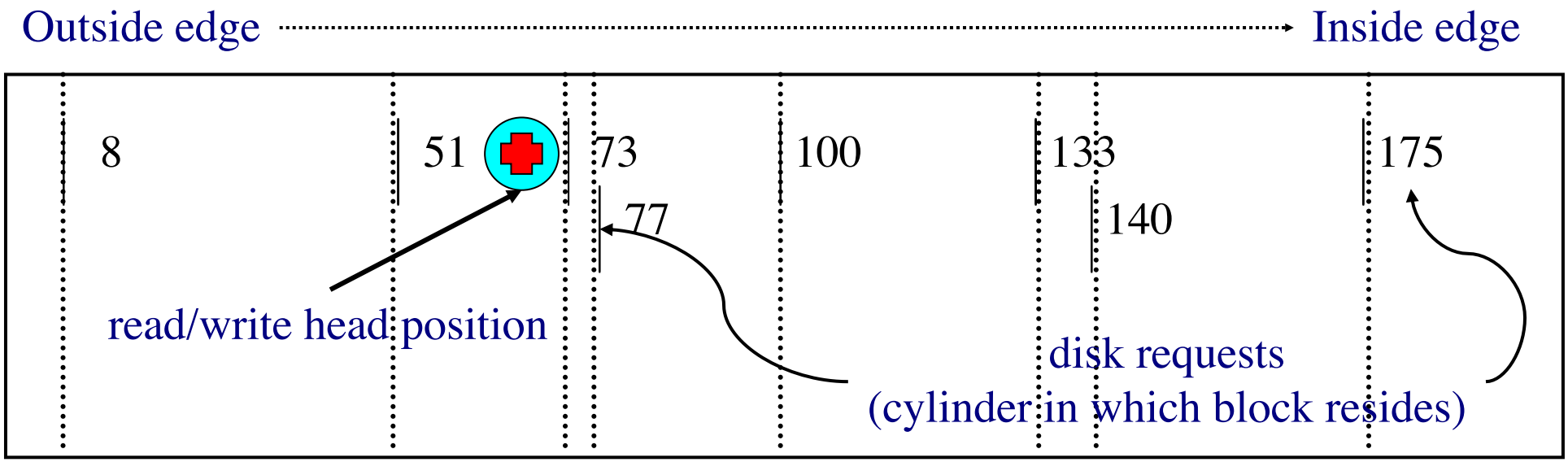
Skipping 2 sectors

What's in a disk request?

- Time required to read or write a disk block determined by 3 factors
  - Seek time
  - Rotational delay
    - Average delay = 1/2 rotation time
    - Example: rotate in 10ms, average rotation delay = 5ms
  - Actual transfer time
    - Transfer time = time to rotate over sector
    - Example: rotate in 10ms, 200 sectors/track => 10/200 ms = 0.05ms transfer time per sector
- Seek time dominates, with rotation time close
- Error checking is done by controllers

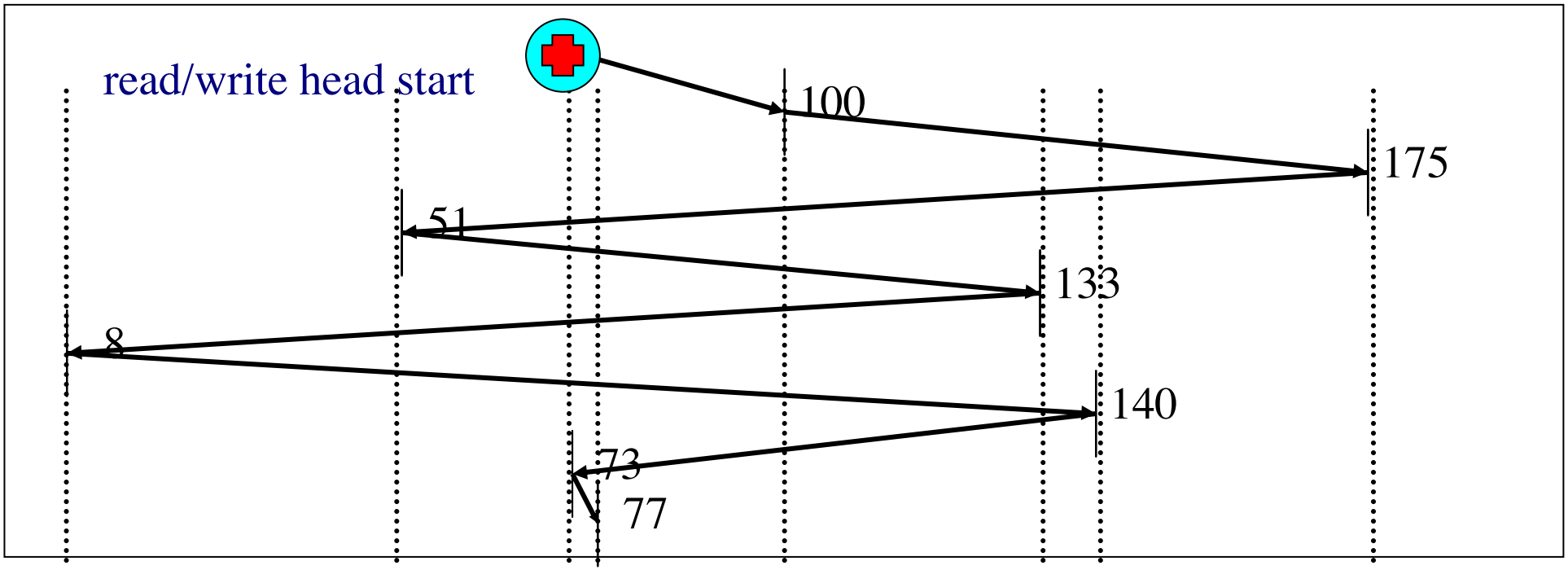
Disk scheduling algorithms

- Schedule disk requests to minimize disk seek time
  - Seek time increases as distance increases (though not linearly)
  - Minimize seek distance -> minimize seek time
- Disk seek algorithm examples assume a request queue & head position (disk has 200 cylinders)
  - Queue = 100, 175, 51, 133, 8, 140, 73, 77
  - Head position = 63



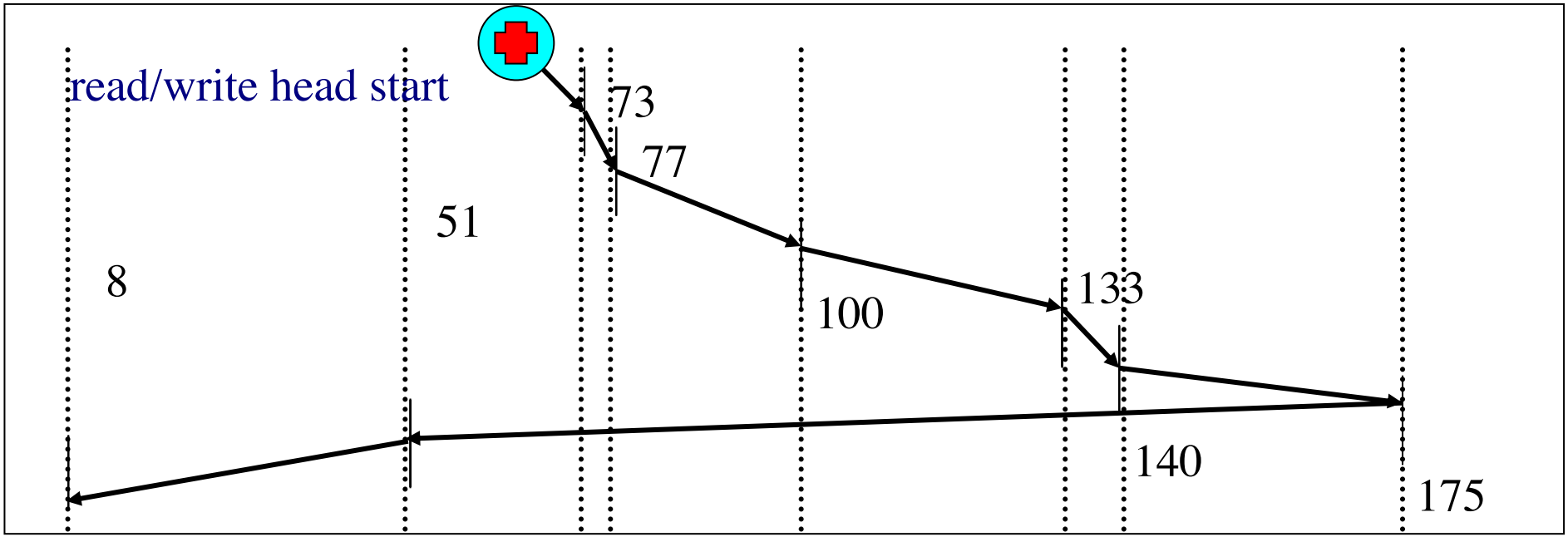
First-Come-First Served (FCFS)

- Requests serviced in the order in which they arrived
  - Easy to implement!
  - May involve lots of unnecessary seek distance
- Seek order = 100, 175, 51, 133, 8, 140, 73, 77
- Seek distance = (100-63) + (175-100) + (175-51) + (133-51) + (133-8) + (140-8) + (140-73) + (77-73) = 646 cylinders



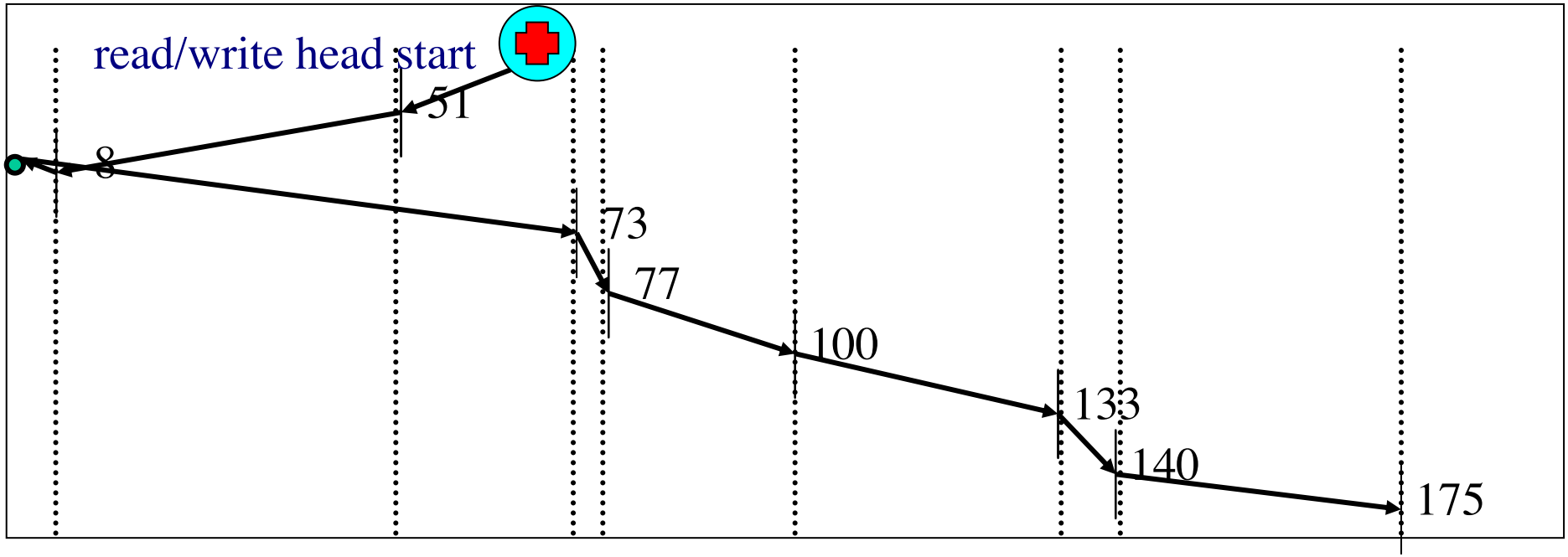
Shortest Seek Time First (SSTF)

- Service the request with the shortest seek time from the current head position
  - Form of SJF scheduling
  - May starve some requests
- Seek order = 73, 77, 100, 133, 140, 175, 51, 8
- Seek distance = 10 + 4 + 23 + 33 + 7 + 35 + 124 + 43 = 279 cylinders



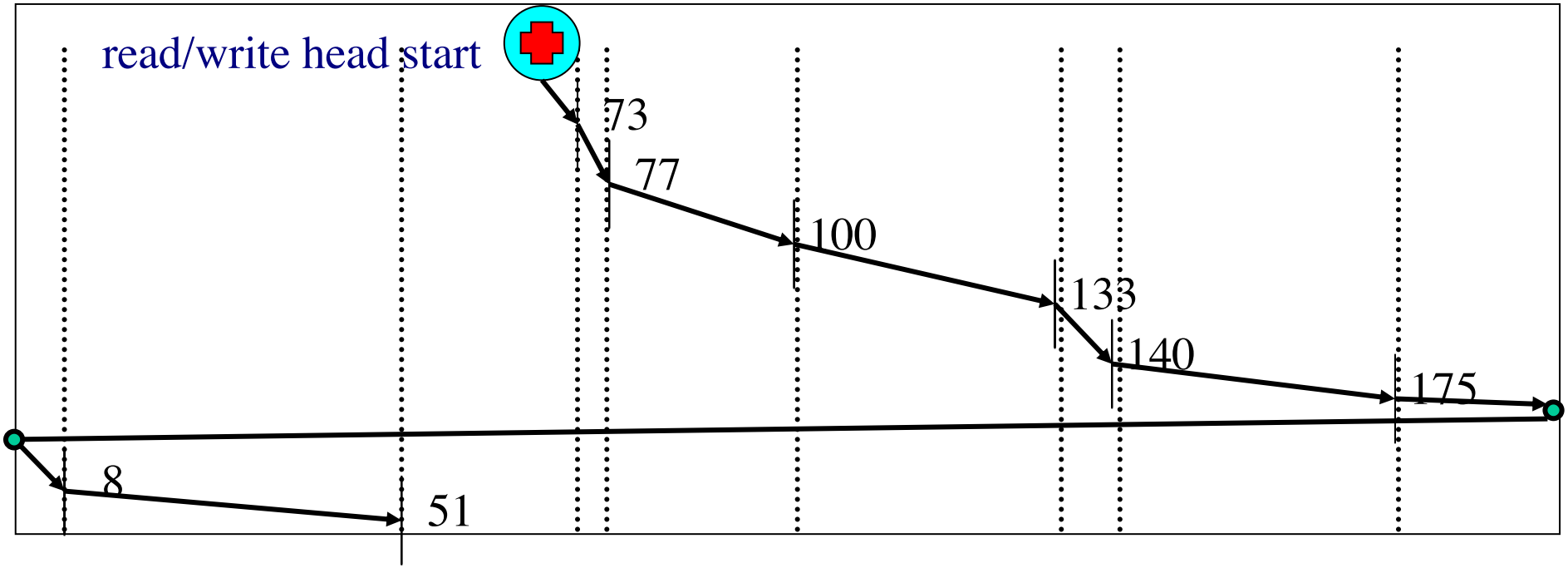
SCAN (elevator algorithm)

- Disk arm starts at one end of the disk and moves towards the other end, servicing requests as it goes
  - Reverses direction when it gets to end of the disk
  - Also known as elevator algorithm
- Seek order = 51, 8, 0 , 73, 77, 100, 133, 140, 175
- Seek distance = 12 + 43 + 8 + 73 + 4 + 23 + 33 + 7 + 35 = 238 cyls



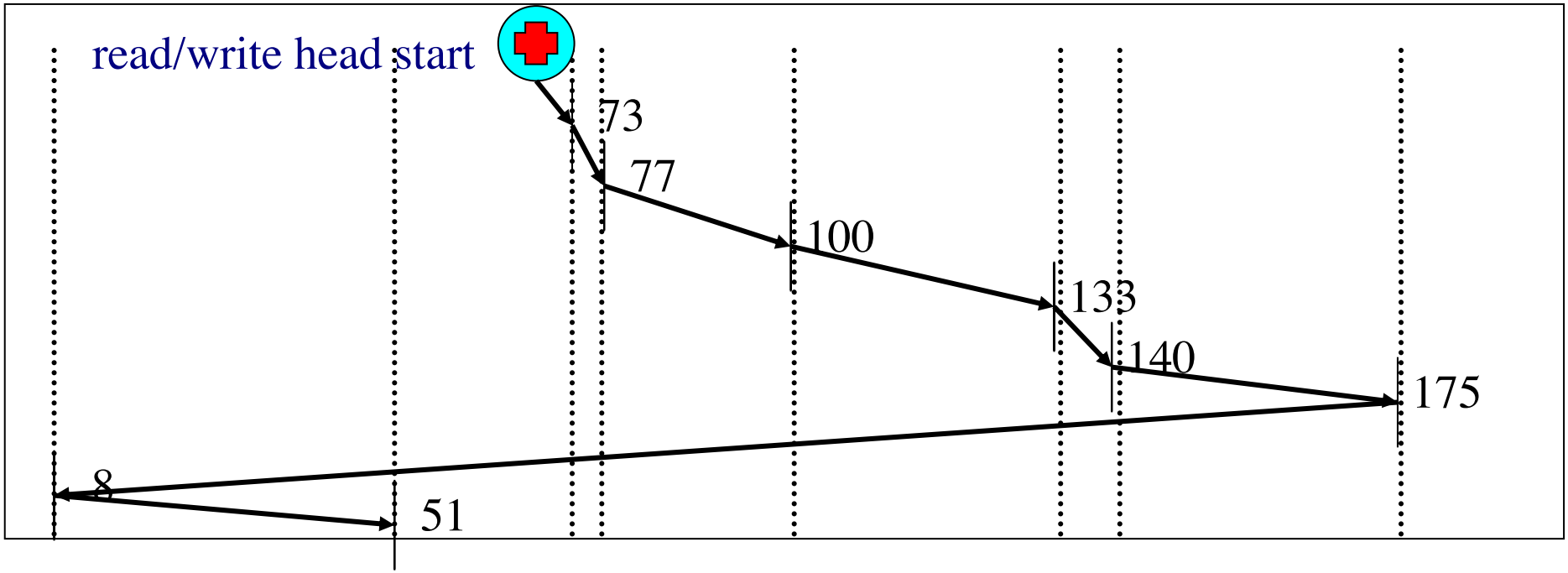
C-SCAN

- Identical to SCAN, except head returns to cylinder 0 when it reaches the end of the disk
  - Treats cylinder list as a circular list that wraps around the disk
  - Waiting time is more uniform for cylinders near the edge of the disk
- Seek order = 73, 77, 100, 133, 140, 175, 199, 0, 8, 51
- Distance = 10 + 4 + 23 + 33 + 7 + 35 + 24 + 199 + 8 + 43 = 386 cyls



C-LOOK

- Identical to C-SCAN, except head only travels as far as the last request in each direction
  - Saves seek time from last sector to end of disk
- Seek order = 73, 77, 100, 133, 140, 175, 8, 51
- Distance = 10 + 4 + 23 + 33 + 7 + 35 + 167 + 43 = 322 cylinders



When good disks go bad…

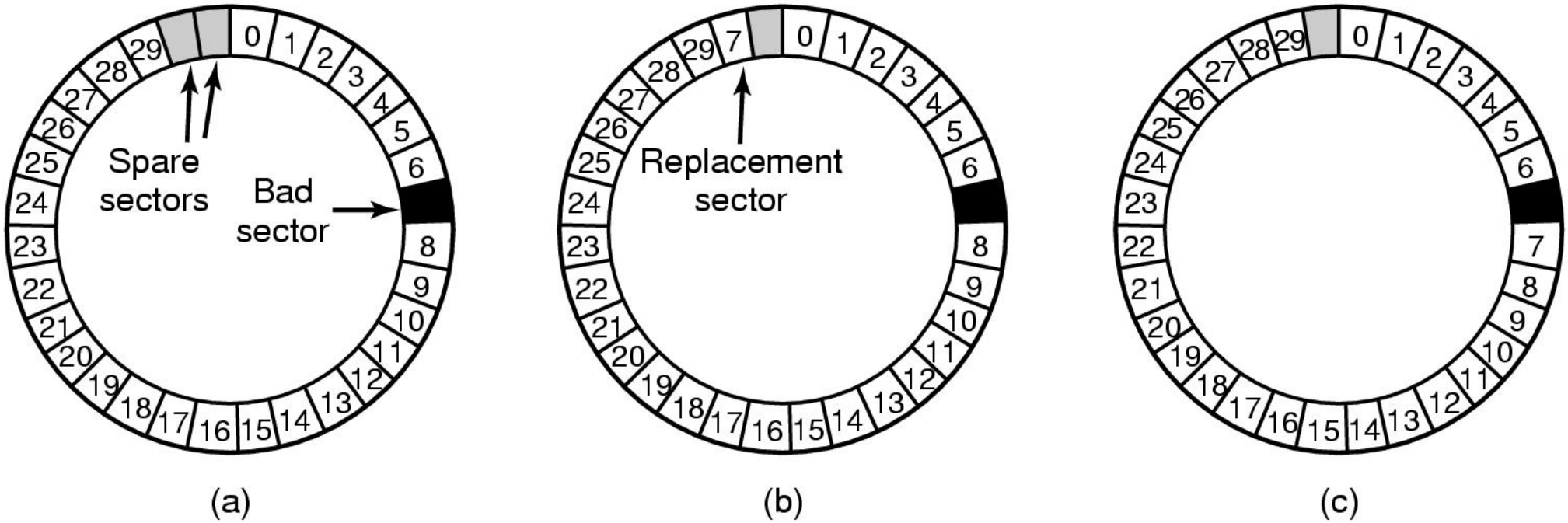
Disks have defects

- In 3M+ sectors, this isn't surprising!

ECC helps with errors, but sometimes this isn't enough

Disks keep spare sectors (normally unused) and remap bad sectors into these spares

- If there's time, the whole track could be reordered…



Long-term information storage

- Must store large amounts of data
  - Gigabytes -> terabytes -> petabytes
- Stored information must survive the termination of the process using it
  - Lifetime can be seconds to years
  - Must have some way of finding it!
- Multiple processes must be able to access the information concurrently

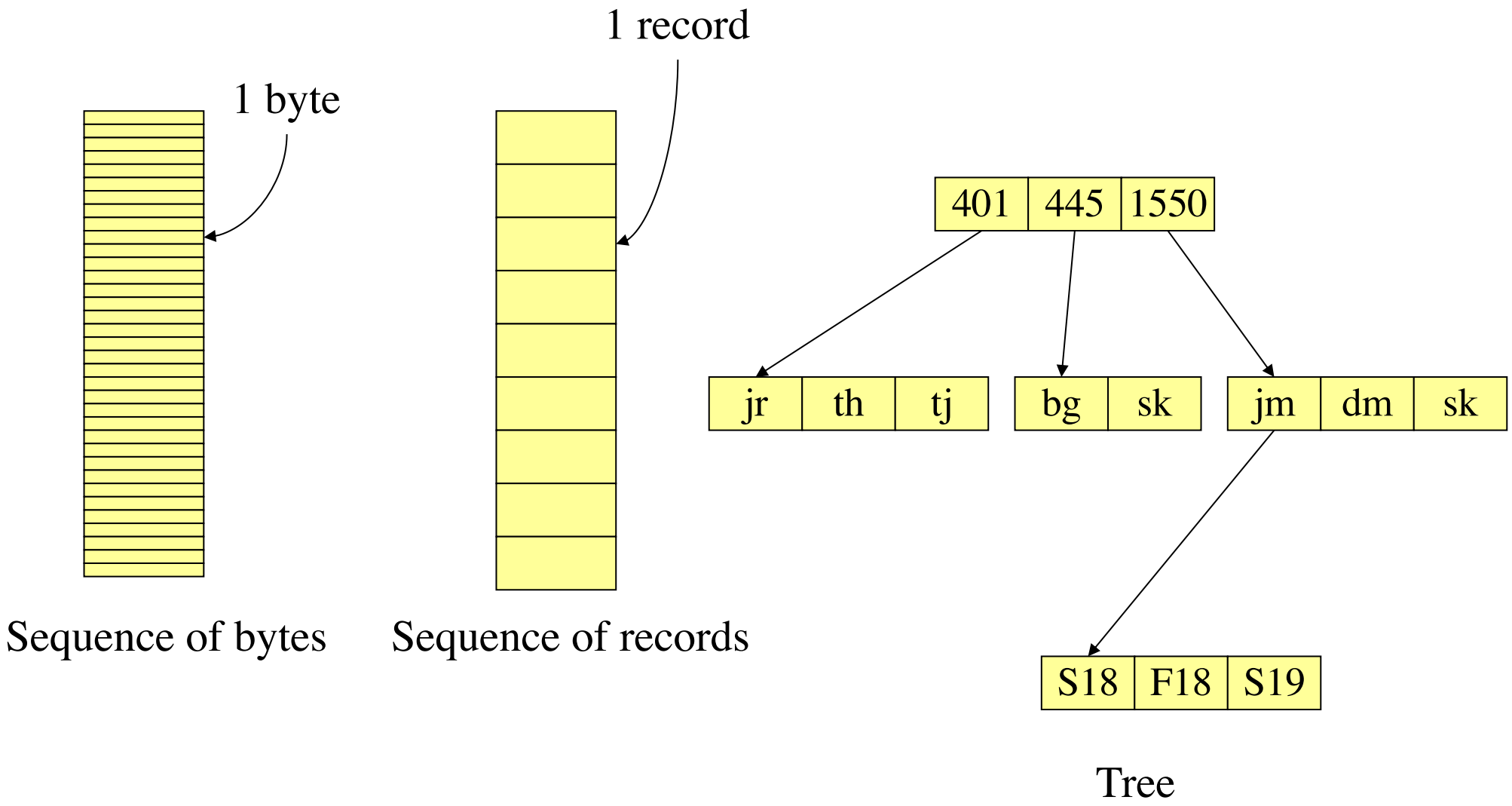
Naming files

- Important to be able to *find* files after they're created
- Every file has at least one name
- Name can be
  - Human-accessible: "foo.c", "my photo", "Go Panthers!", "Go Steelers!"
  - Machine-usable: 4502, 33481
- Case may or may not matter
  - Depends on the file system
- Name may include information about the file's contents
  - Certainly does for the user (the name should make it easy to figure out what's in it!)
  - Computer may use part of the name to determine the file type

Typical file extensions

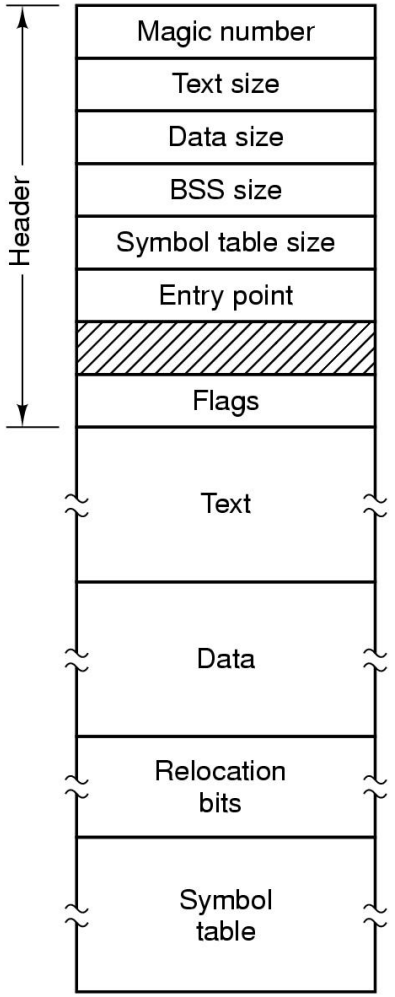
Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

File structures

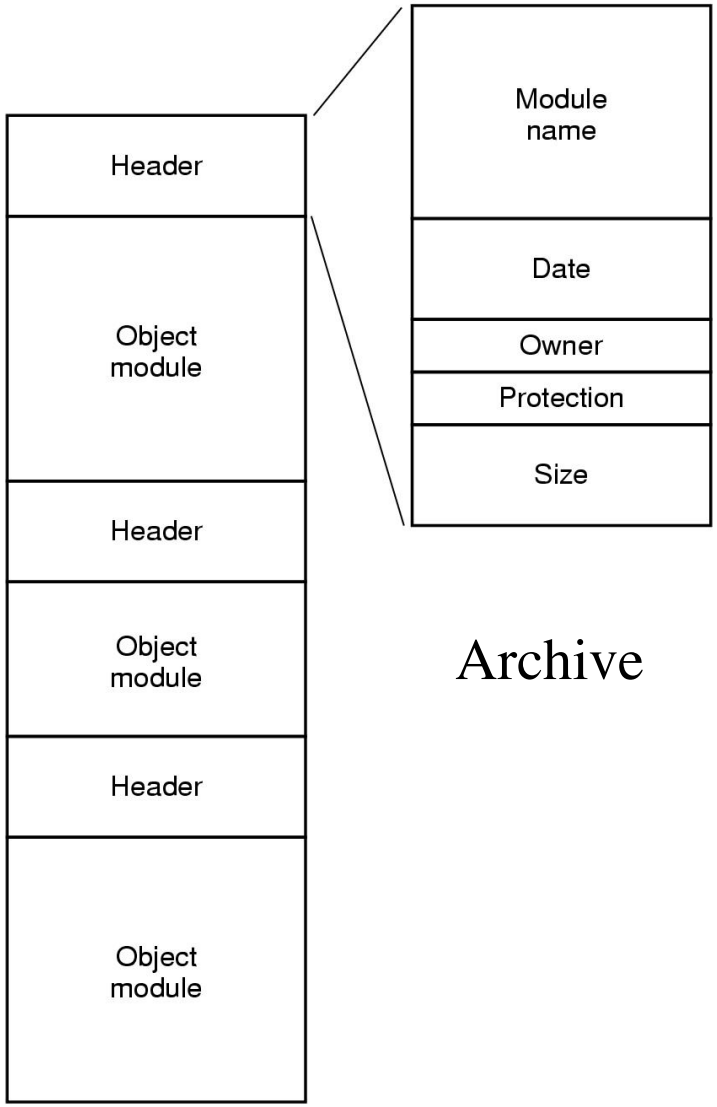


File types

Executable file



(a)



Archive

(b)

File attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File operations

- Create: make a new file
- Delete: remove an existing file
- Open: prepare a file to be accessed
- Close: indicate that a file is no longer being accessed
- Read: get data from a file
- Write: put data to a file
- Append: like write, but only at the end of the file
- Seek: move the "current" pointer elsewhere in the file
- Get attributes: retrieve attribute information
- Set attributes: modify attribute information
- Rename: change a file's name

Using file system calls

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
      int in_fd, out_fd, rd_count, wt_count;
      char buffer[BUF_SIZE];

      if (argc != 3) exit(1);           /* syntax error if argc is not 3 */
```

Using file system calls, continued

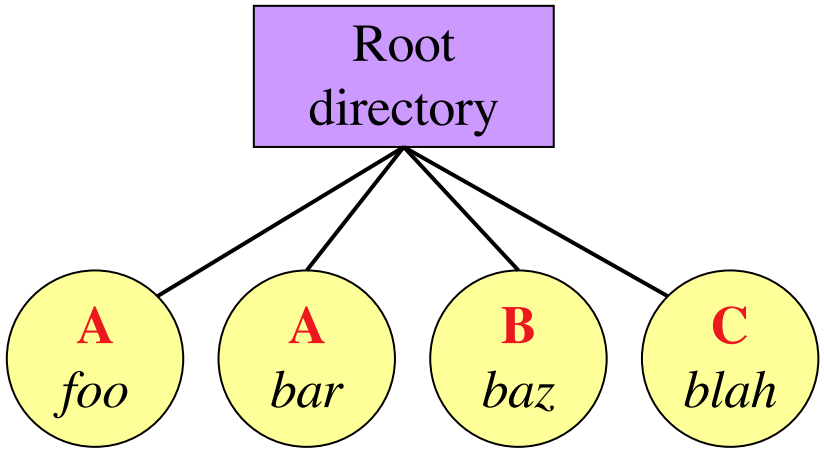
```
    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

Directories

- Naming is nice, but limited
- Humans like to group things together for convenience
- File systems allow this to be done with *directories* (sometimes called *folders*)
- Grouping makes it easier to
  - Find files in the first place: remember the enclosing directories for the file
  - Locate related files (or just determine which files are related)

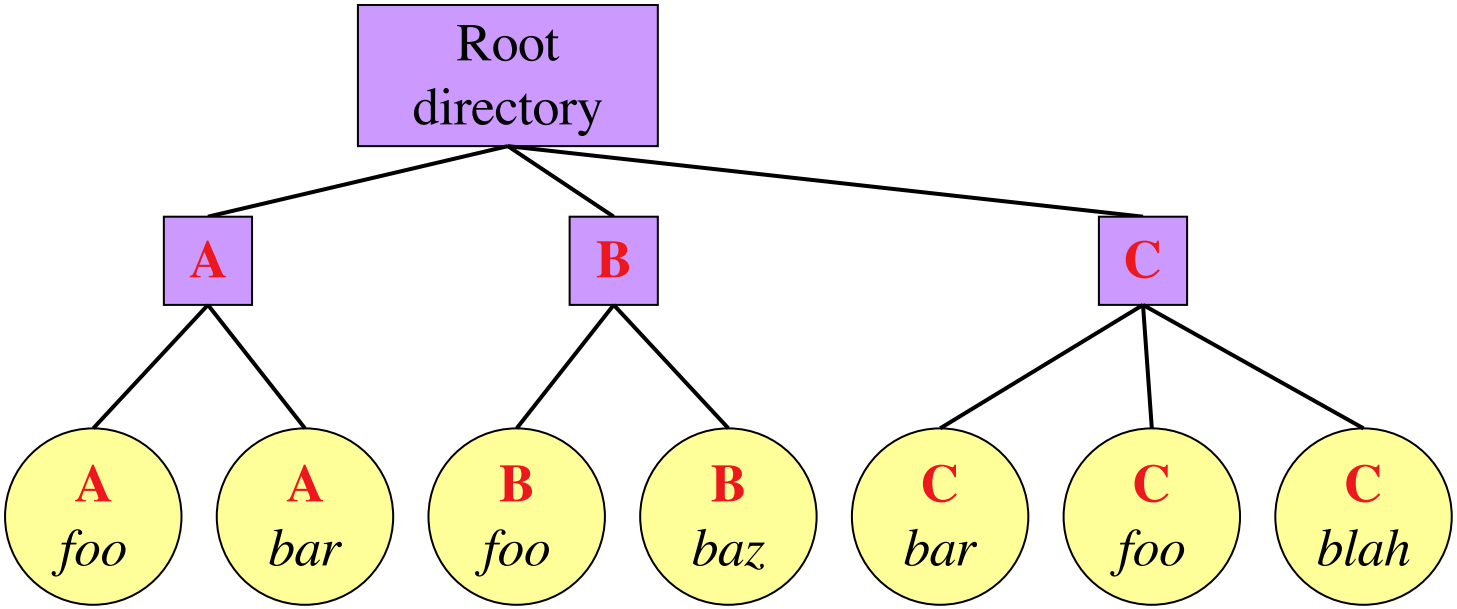
Single-level directory systems

- One directory in the file system
- Example directory
  - Contains 4 files (*foo*, *bar*, *baz*, *blah*)
  - owned by 3 different people: A, B, and C (owners shown in red)
- Problem: what if user B wants to create a file called *foo*?

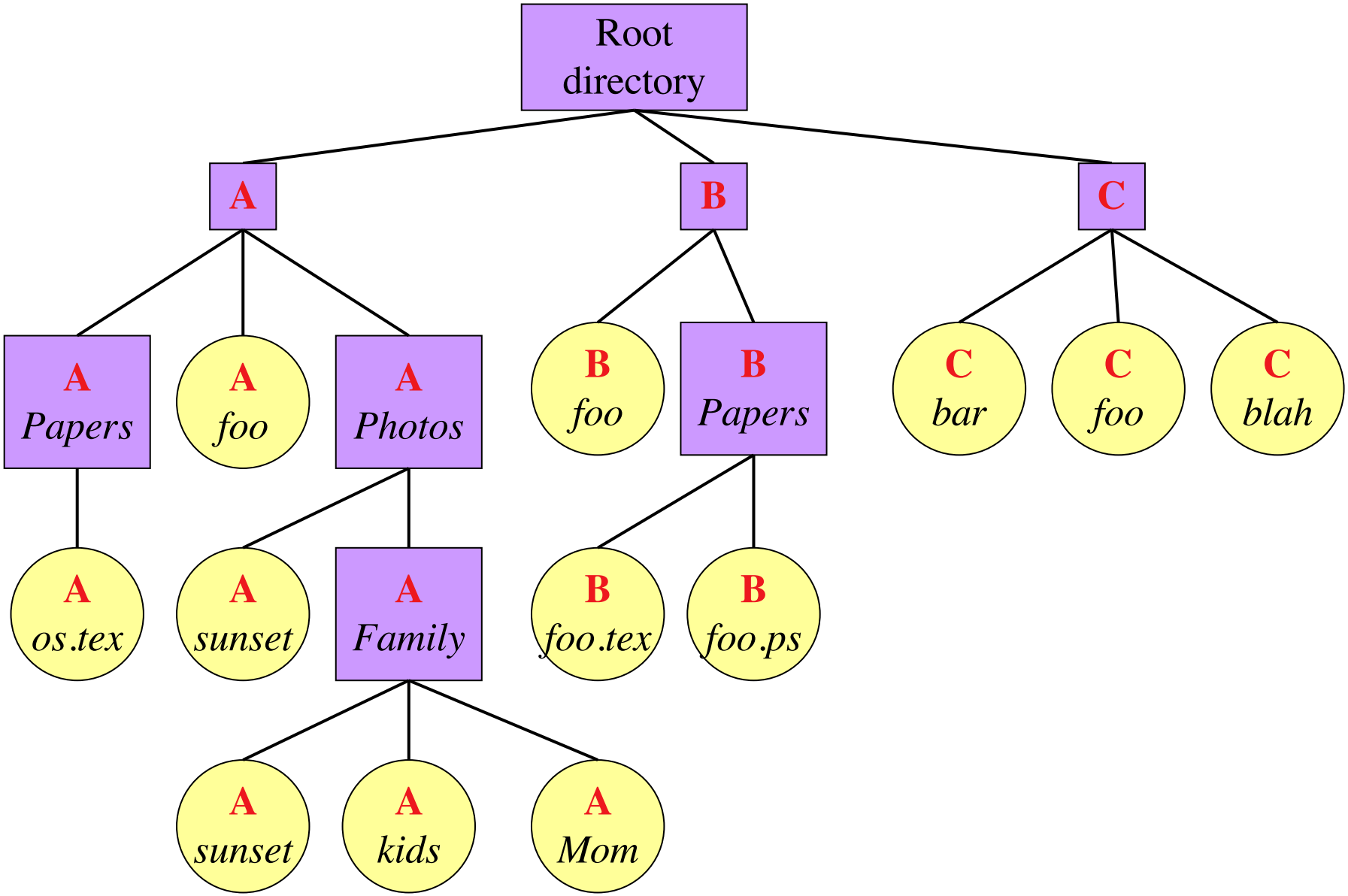


Two-level directory system

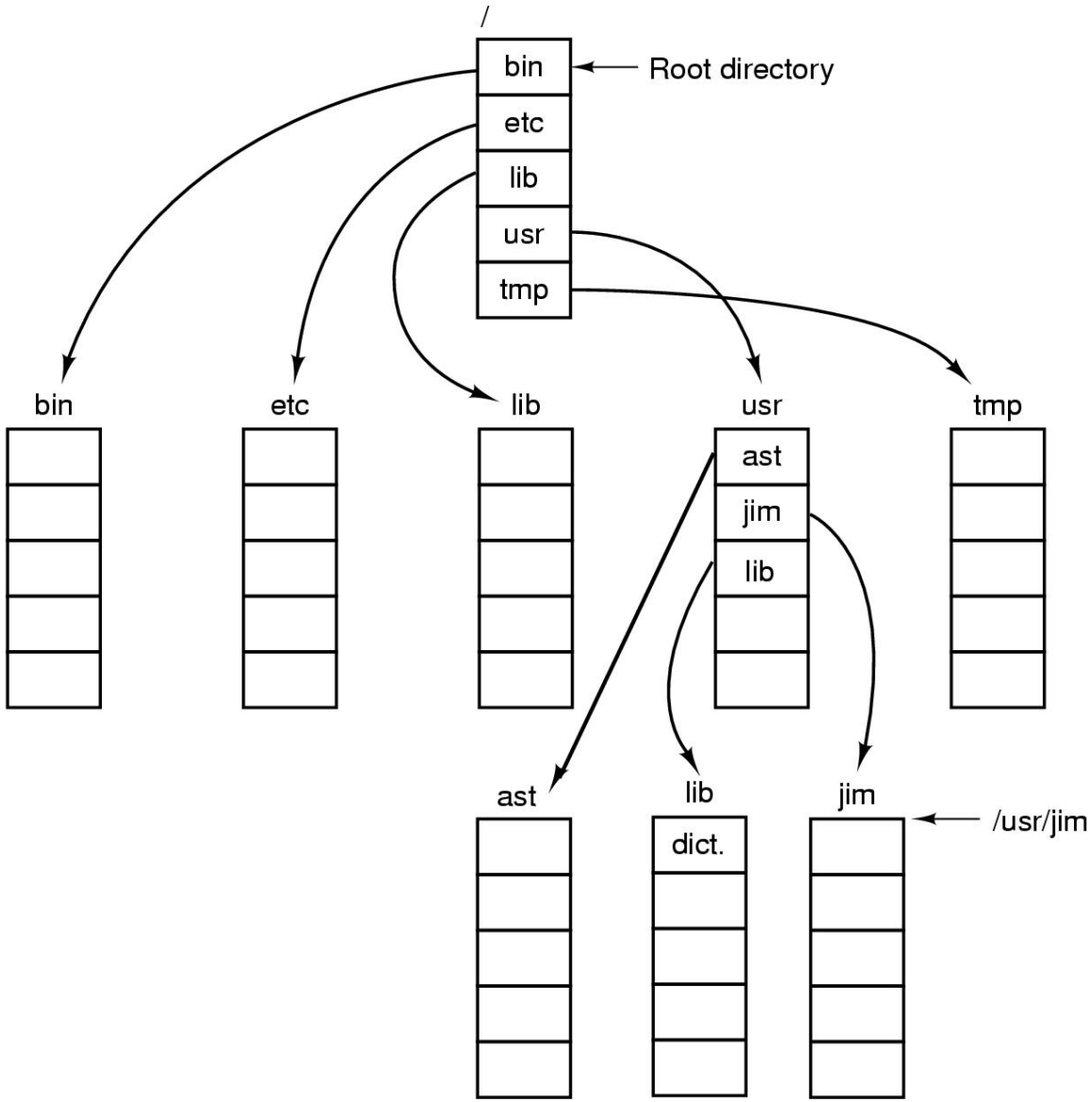
- Solves naming problem: each user has her own directory
- Multiple users can use the same file name
- By default, users access files in their own directories
- Extension: allow users to access files in others' directories



Hierarchical directory system



Unix directory tree



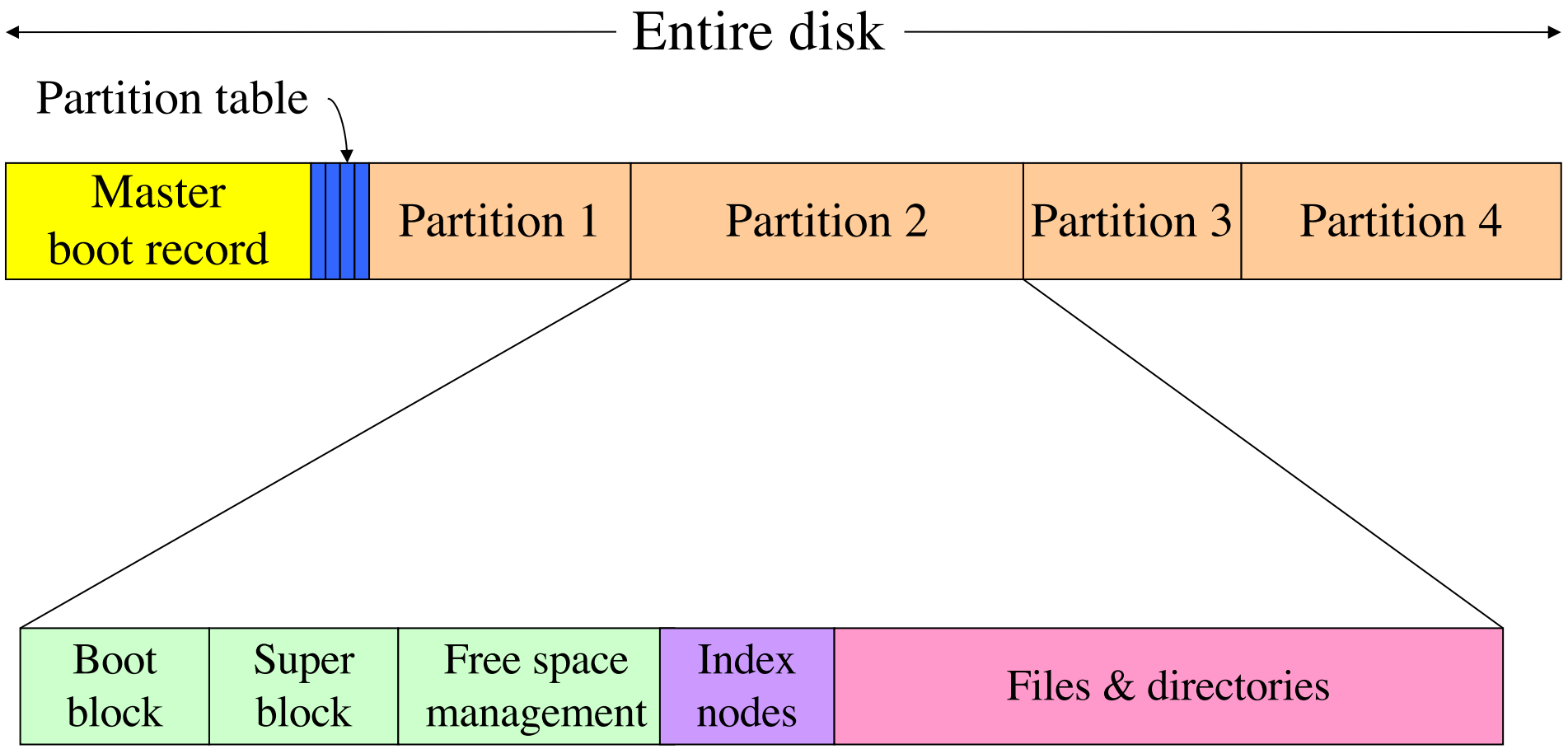
Operations on directories

- Create: make a new directory
- Delete: remove a directory (usually must be empty)
- Opendir: open a directory to allow searching it
- Closedir: close a directory (done searching)
- Readdir: read a directory entry
- Rename: change the name of a directory
  - Similar to renaming a file
- Link: create a new entry in a directory to link to an existing file
- Unlink: remove an entry in a directory
  - Remove the file if this is the last link to this file

File system implementation issues

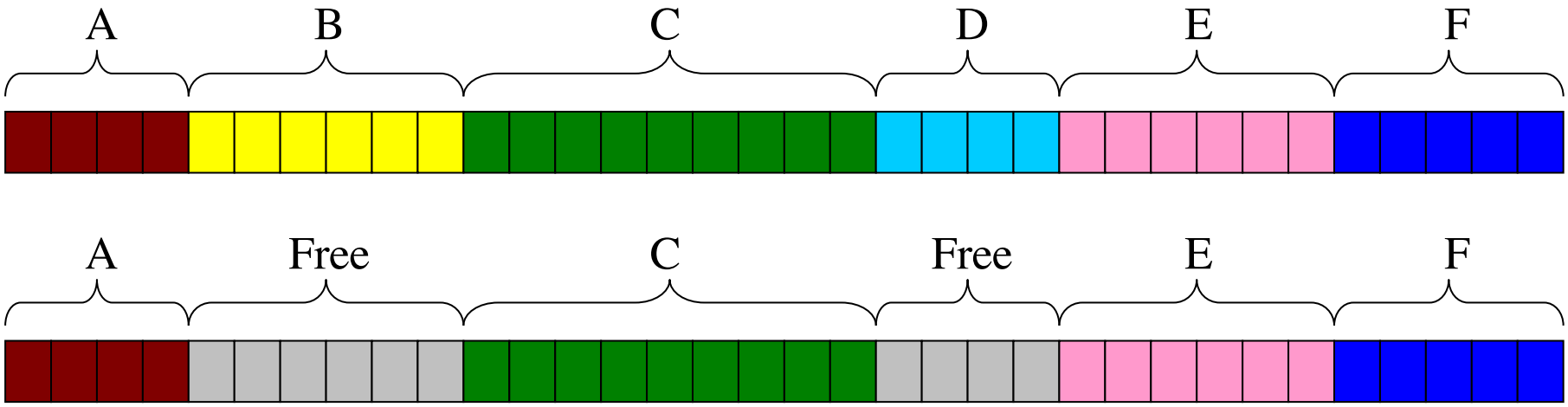
- How are disks divided up into file systems?
- How does the file system allocate blocks to files?
- How does the file system manage free space?
- How are directories handled?
- How can the file system improve…
  - Performance?
  - Reliability?

Carving up the disk



Contiguous allocation for file blocks

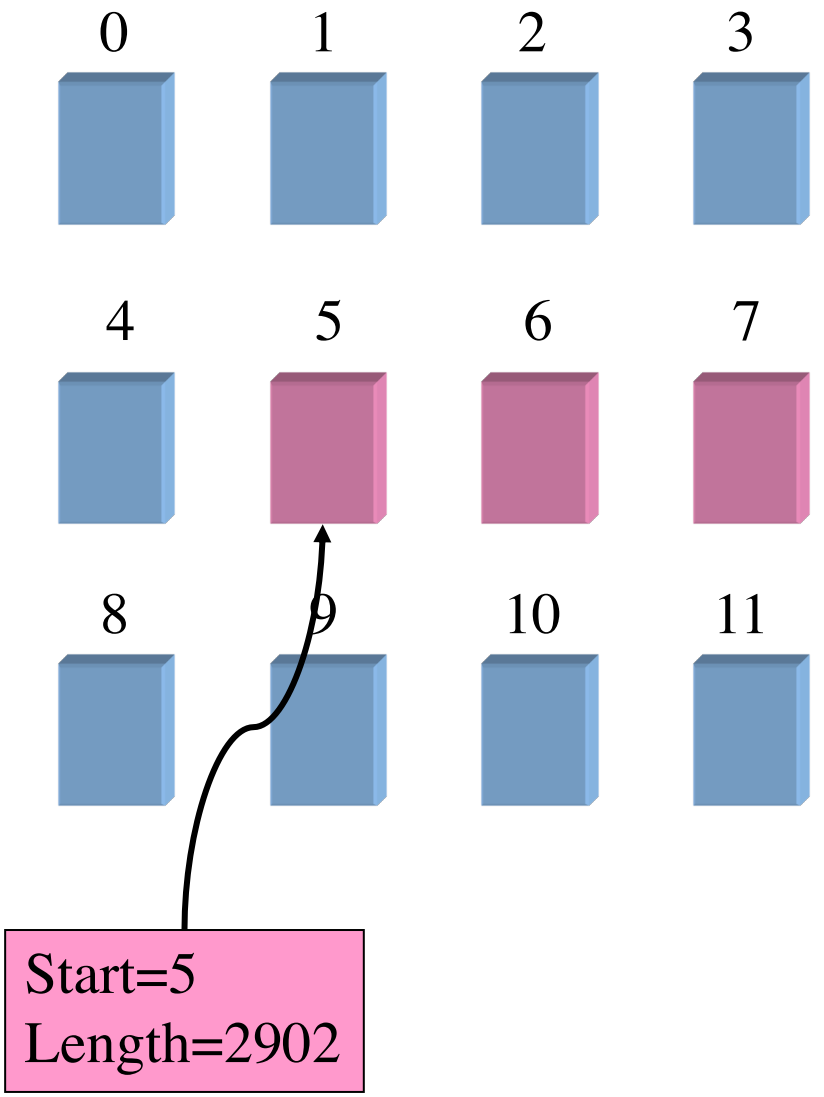
- Contiguous allocation requires all blocks of a file to be consecutive on disk
- Problem: deleting files leaves “holes”
  - Similar to memory allocation issues
  - Compacting the disk can be a very slow procedure…



Contiguous allocation

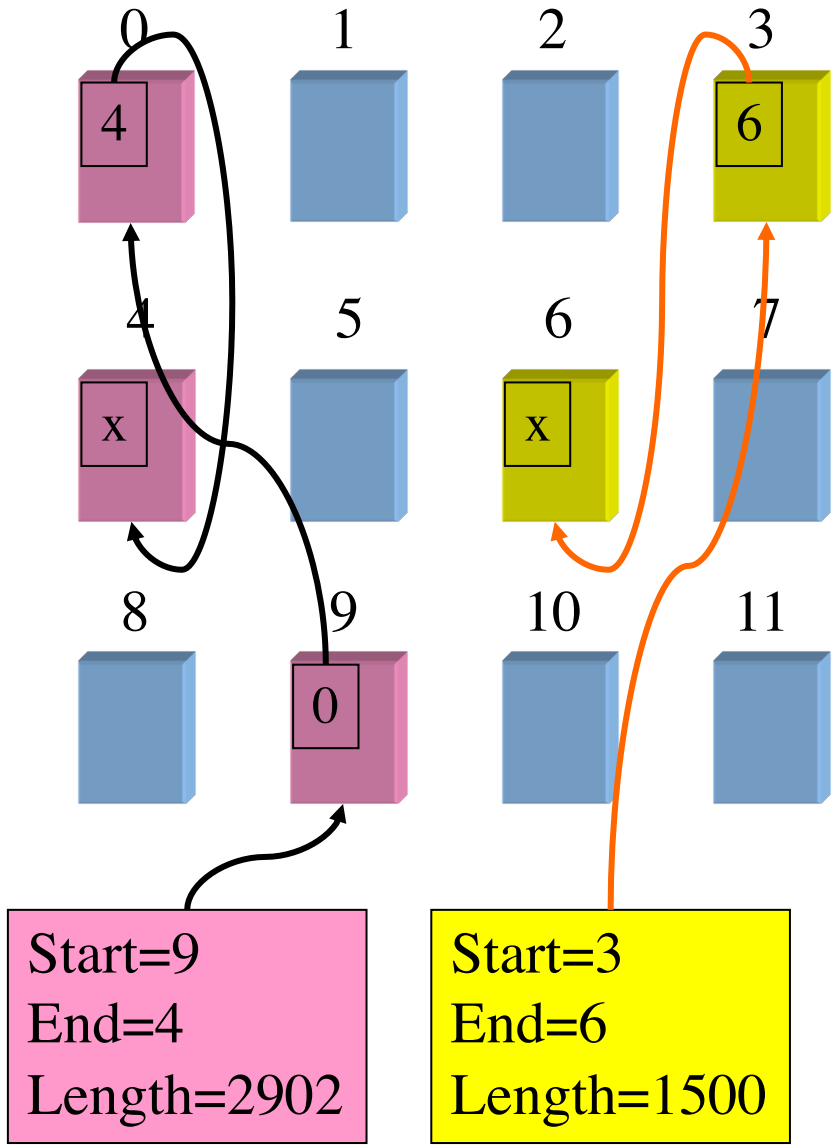
- Data in each file is stored in consecutive blocks on disk
- Simple & efficient indexing
  - Starting location (block #) on disk (`start`)
  - Length of the file in blocks (`length`)
- Random access well-supported
- Difficult to grow files
  - Must pre-allocate all needed space
  - Wasteful of storage if file isn't using all of the space
- Logical to physical mapping is easy

```
blocknum = (pos / 1024) + start;
offset_in_block = pos % 1024;
```



Linked allocation

- File is a linked list of disk blocks
  - Blocks may be scattered around the disk drive
  - Block contains both pointer to next block and data
  - Files may be as long as needed
- New blocks are allocated as needed
  - Linked into list of blocks in file
  - Removed from list (bitmap) of free blocks



Finding blocks with linked allocation

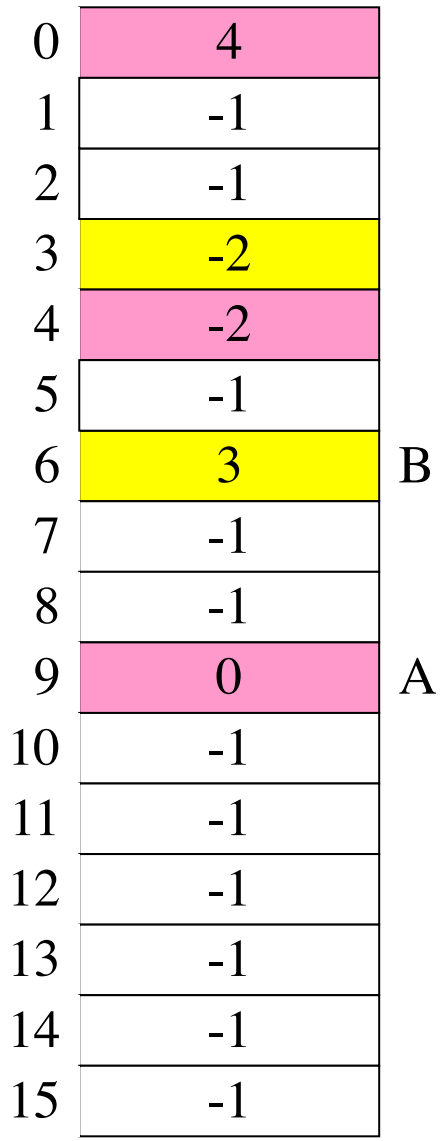
- Directory structure is simple
  - Starting address looked up from directory
  - Directory only keeps track of first block (not others)
- No wasted blocks - all blocks can be used
- Random access is difficult: must always start at first block!
- Logical to physical mapping is done by

```
block = start;
offset_in_block = pos % 1020;
for (j = 0; j < pos / 1020; j++) {
  block = block->next;
}
```

  - Assumes that *next* pointer is stored at end of block
  - May require a long time for seek to random location in file

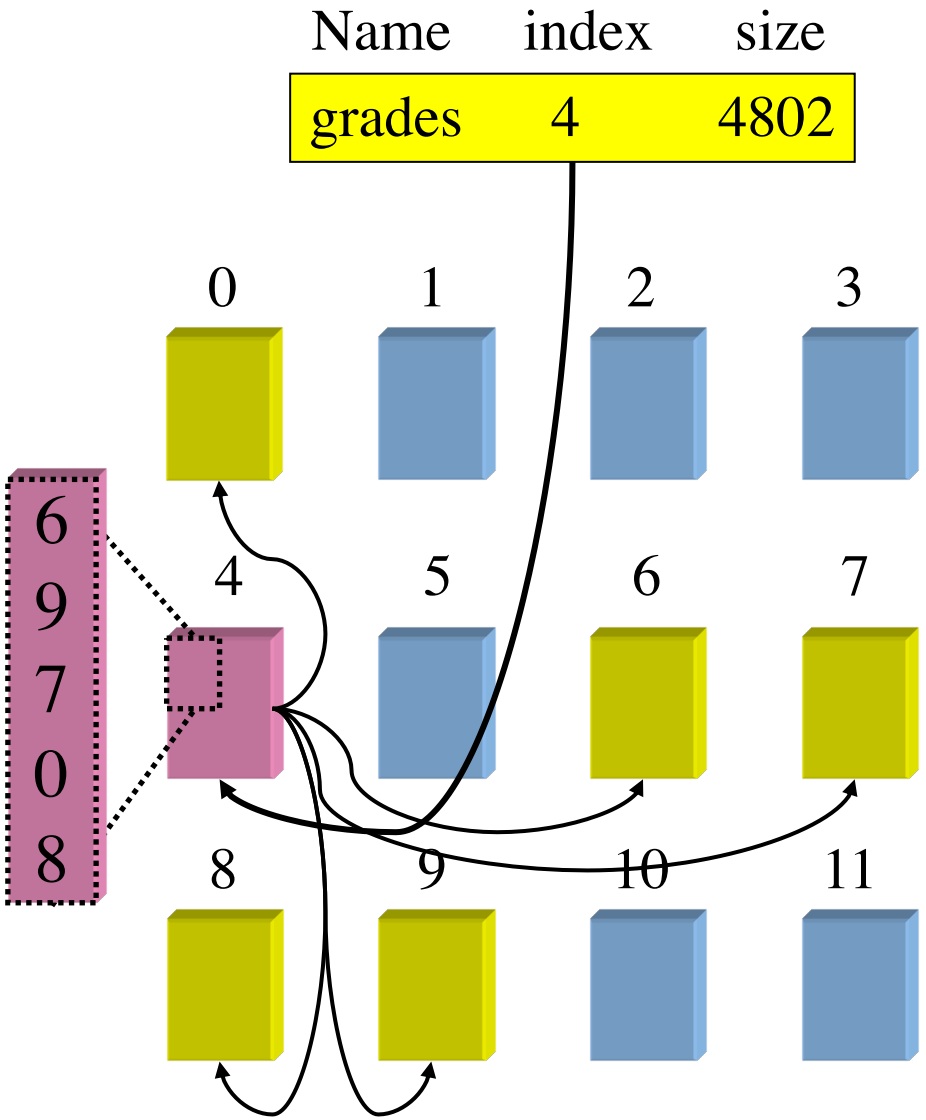
File Allocation Table (FAT)

- Links on disk are slow
- Keep linked list in memory
- Advantage: faster
- Disadvantages
  - Have to copy it to disk at some point
  - Have to keep in-memory and on-disk copy consistent



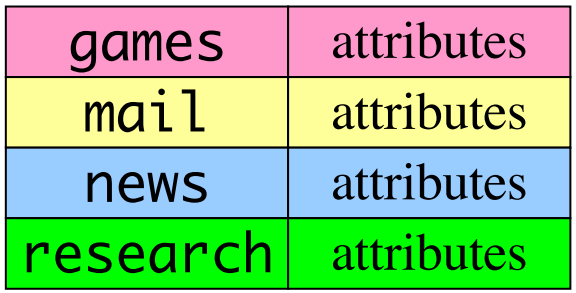
Using a block index for allocation

- Store file block addresses in an array
  - Array itself is stored in a disk block
  - Directory has a pointer to this disk block
  - Non-existent blocks indicated by -1
- Random access easy
- Limit on file size?

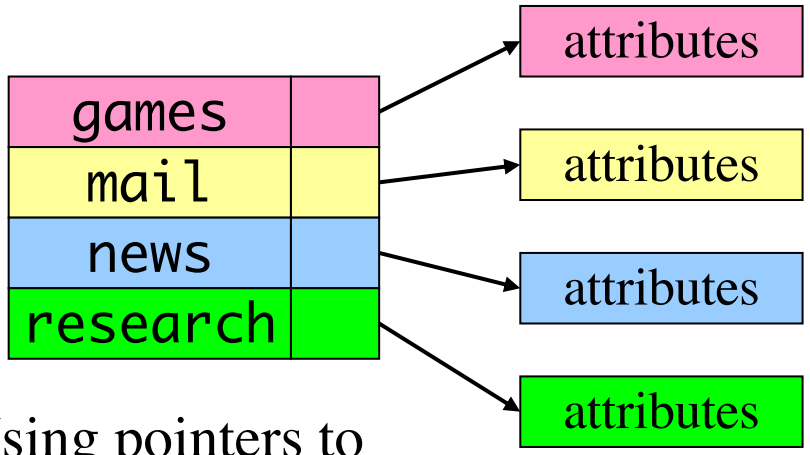


What's in a directory?

- Two types of information
  - File names
  - File metadata (size, timestamps, etc.)
- Basic choices for directory information
  - Store all information in directory
    - Fixed size entries
    - Disk addresses and attributes in directory entry
  - Store names & pointers to index nodes (i-nodes)

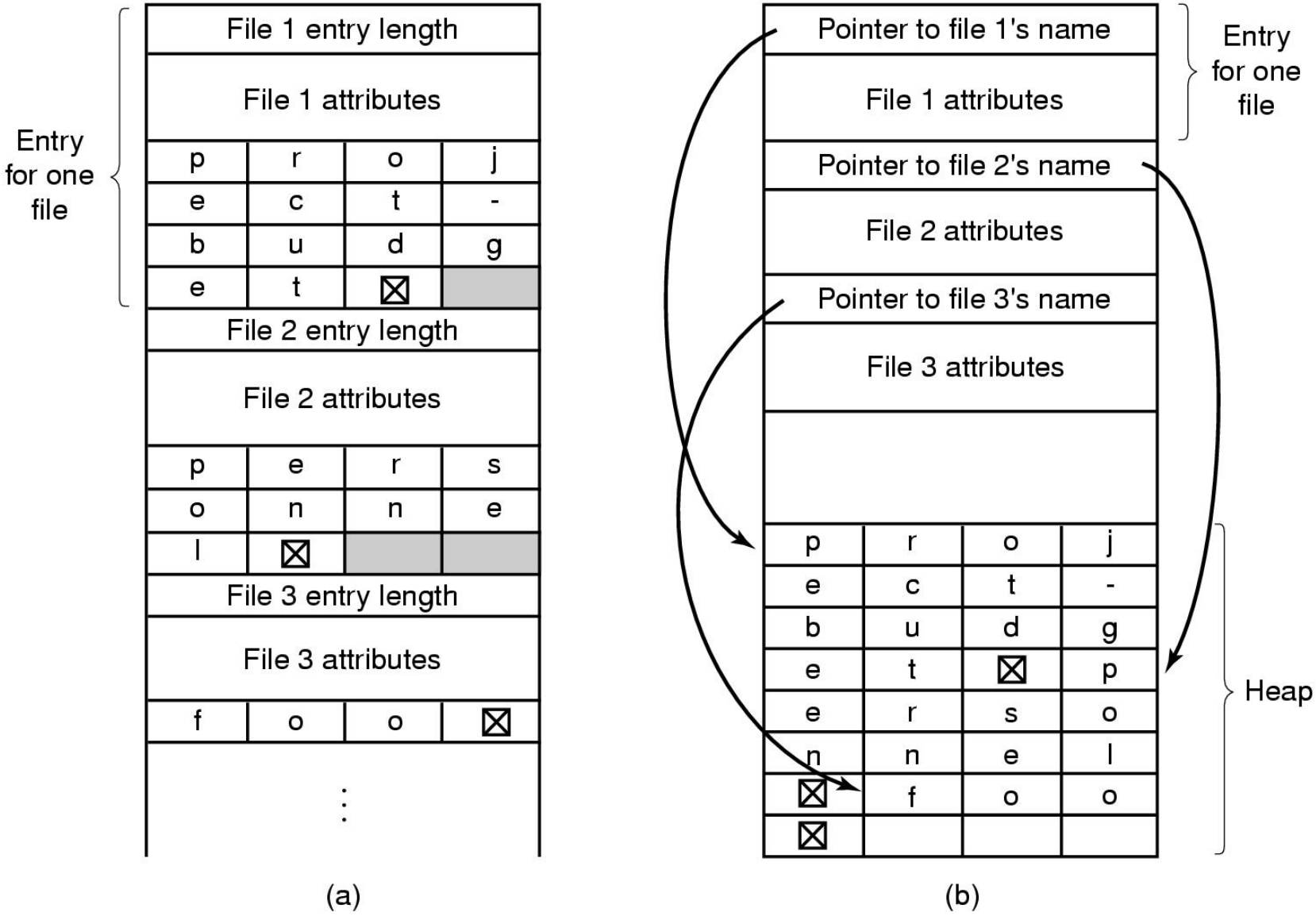


Storing all information in the directory

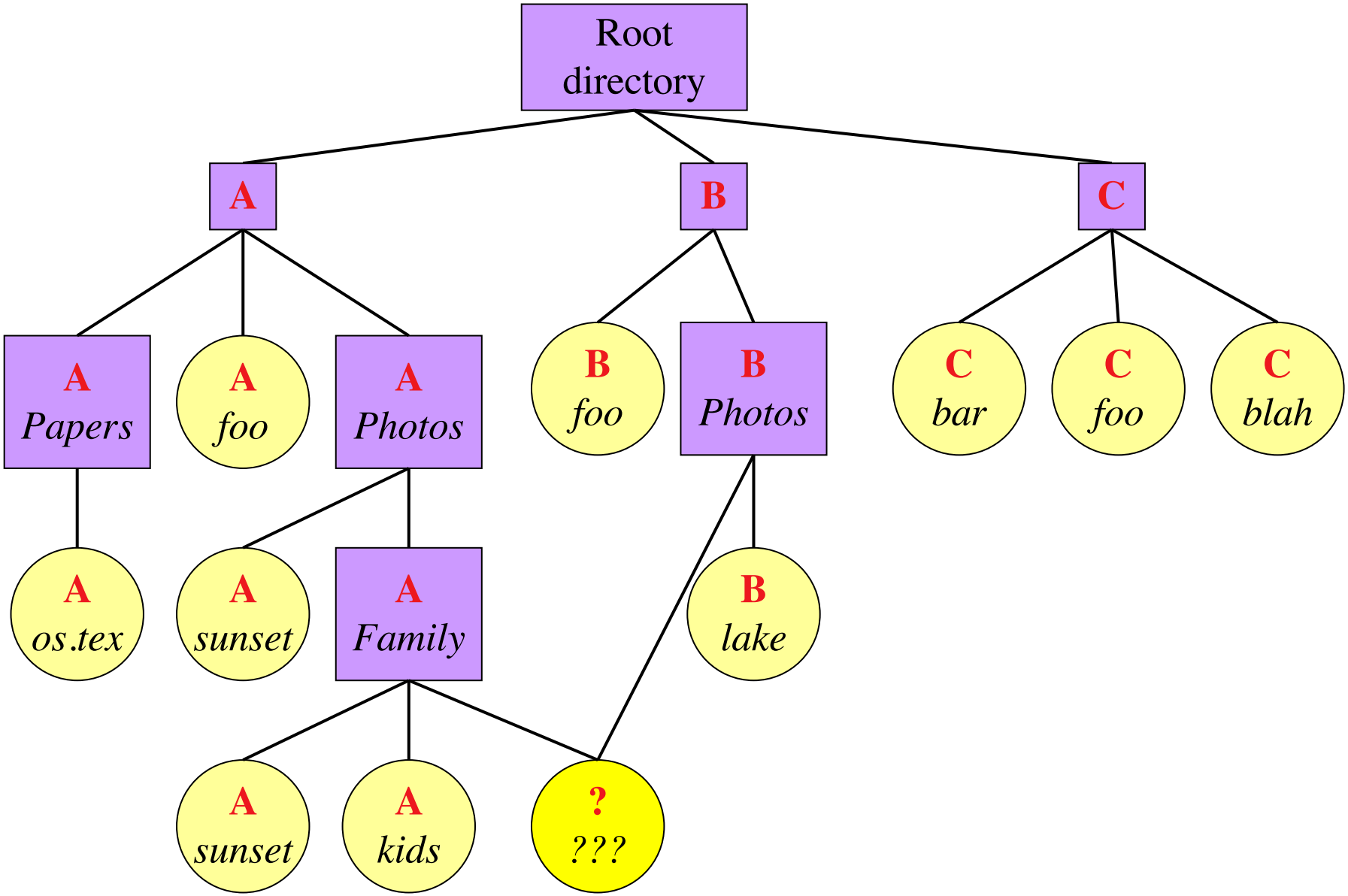


Using pointers to index nodes

Handling long file names in a directory

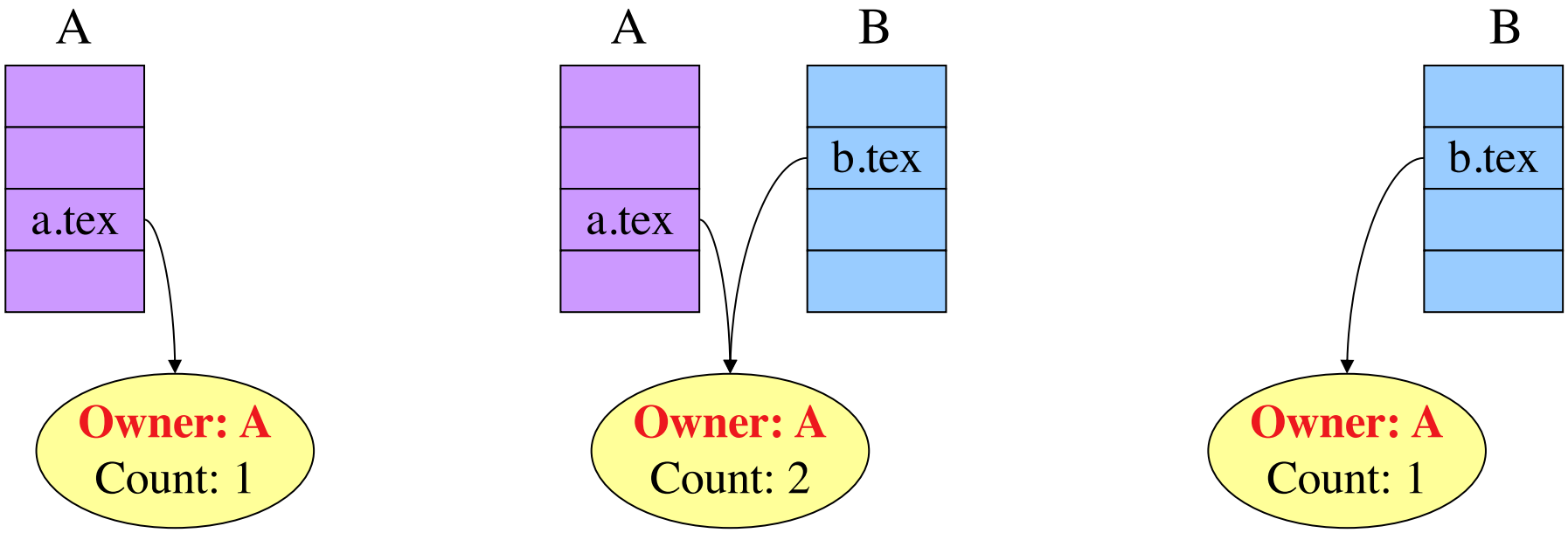


Sharing files

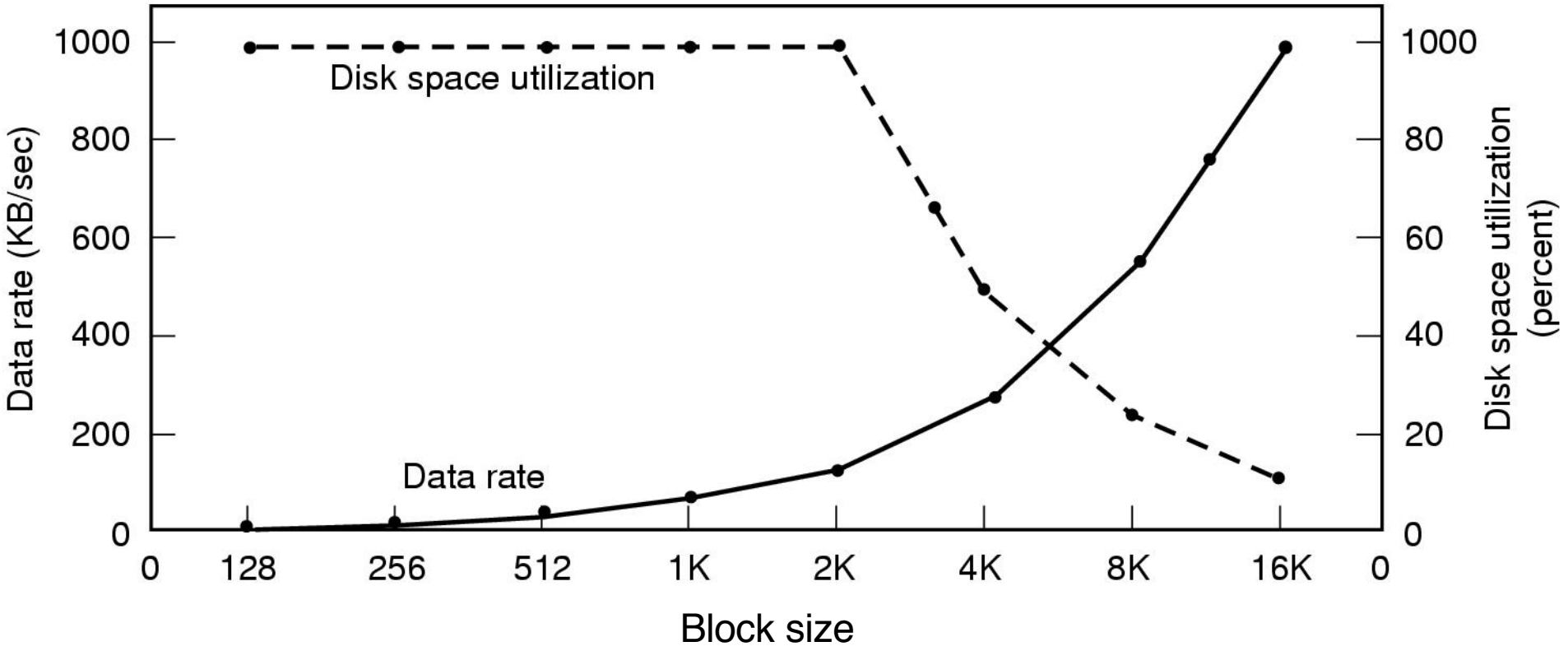


Solution: use links

- A creates a file, and inserts into her directory
- B shares the file by creating a link to it
- A unlinks the file
  - B still links to the file
  - Owner is still A (unless B explicitly changes it)

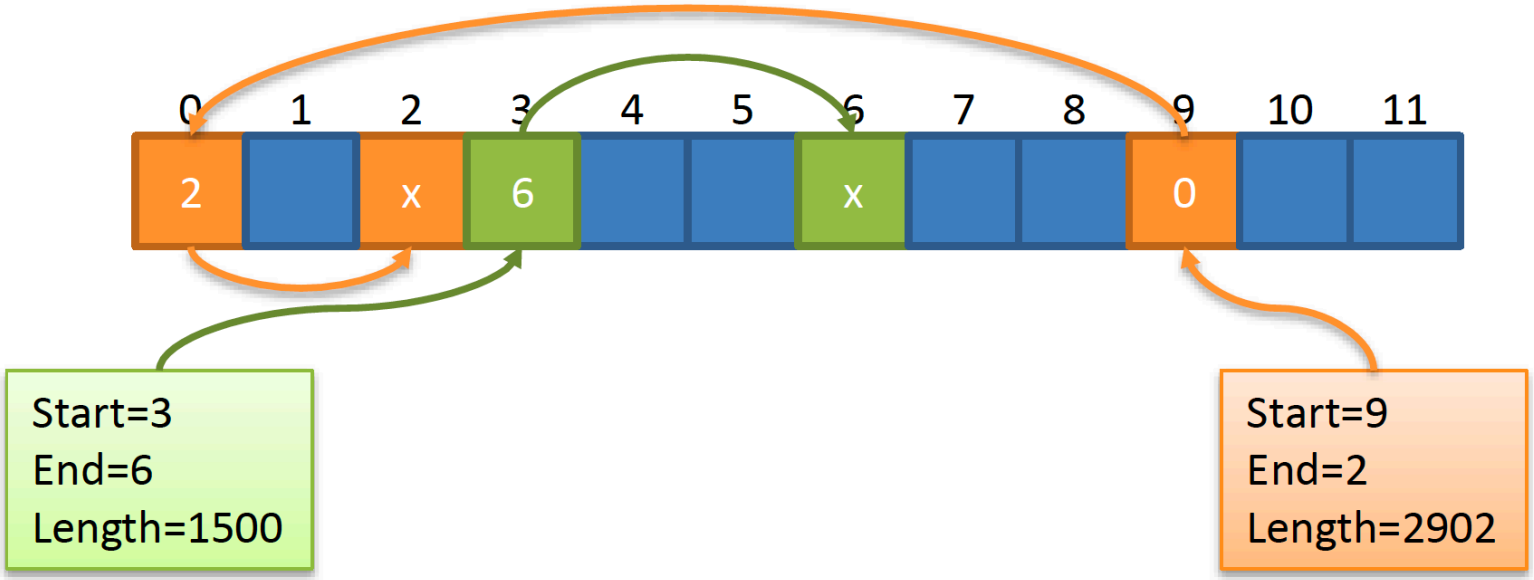


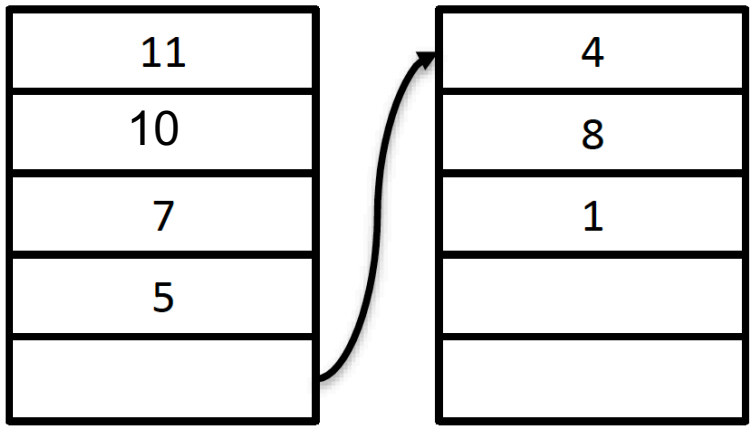
Managing disk space



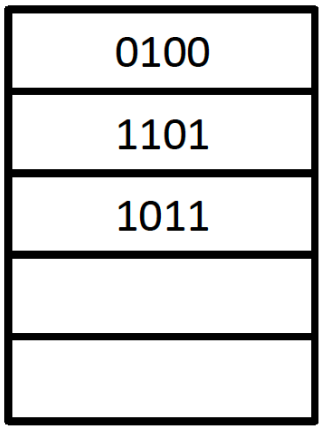
- Dark line (left hand scale) gives data rate of a disk
- Dotted line (right hand scale) gives disk space efficiency
- All files 2KB

Free Block Tracking



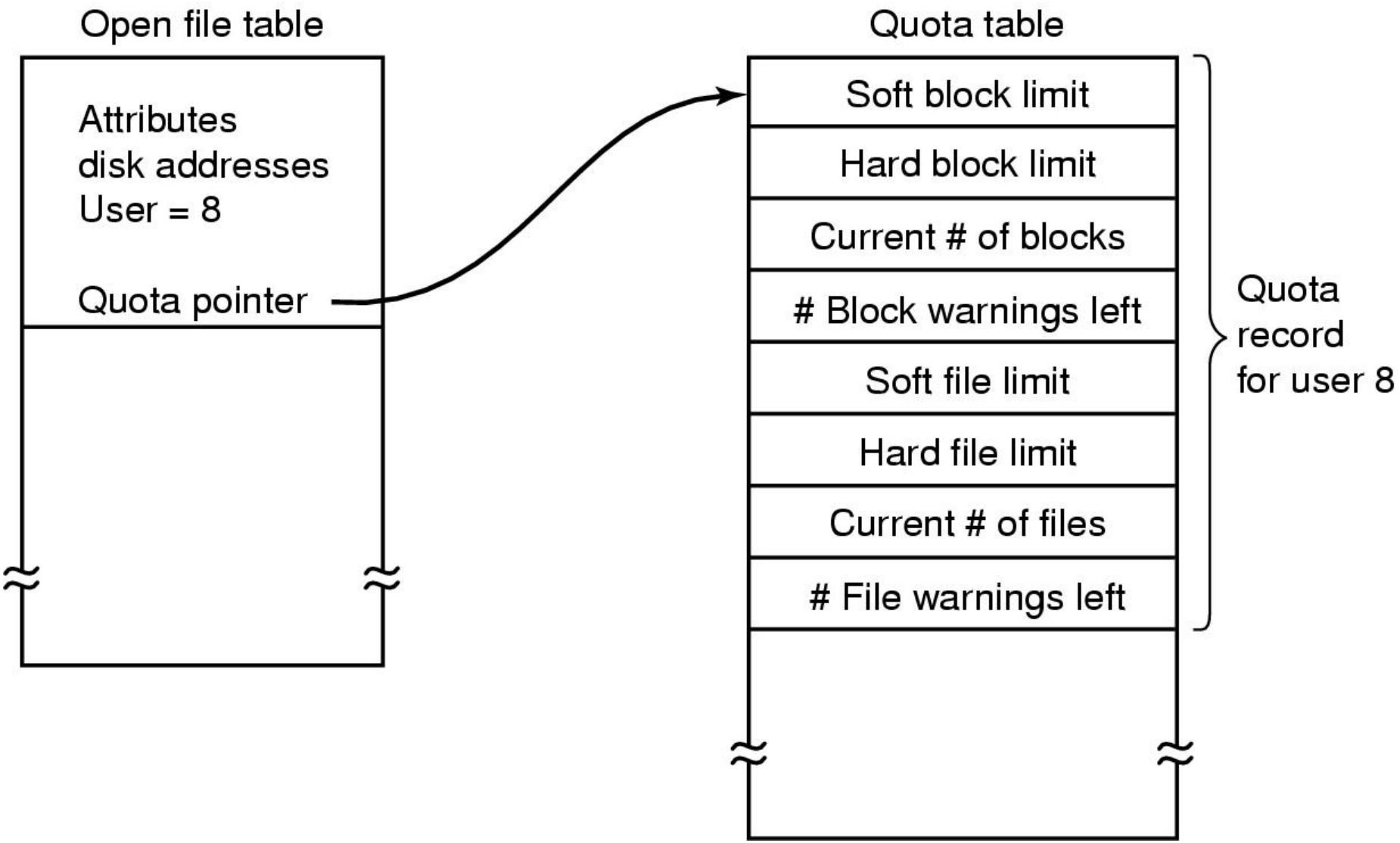


Linked List



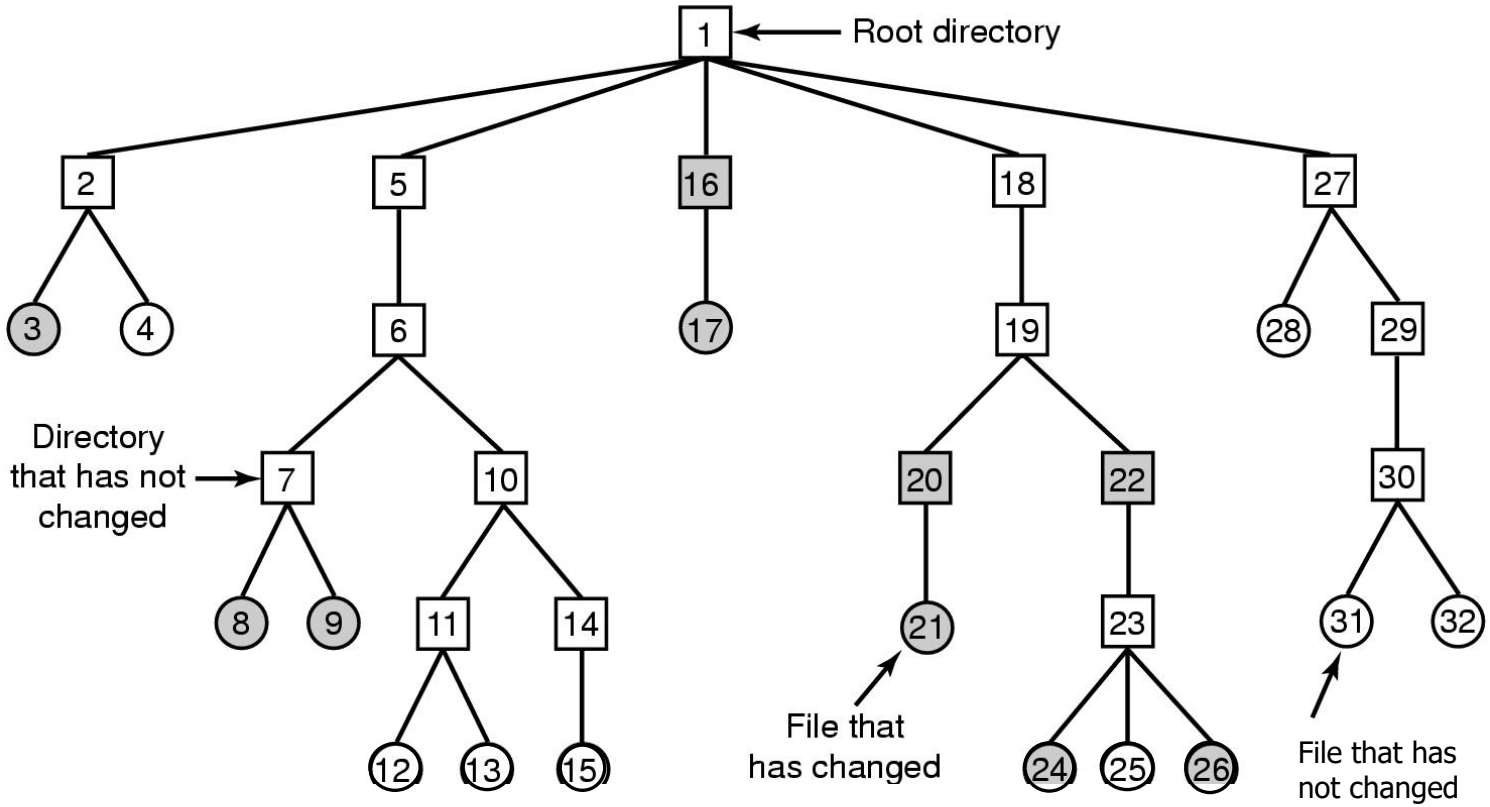
Bitmap

Disk quotas

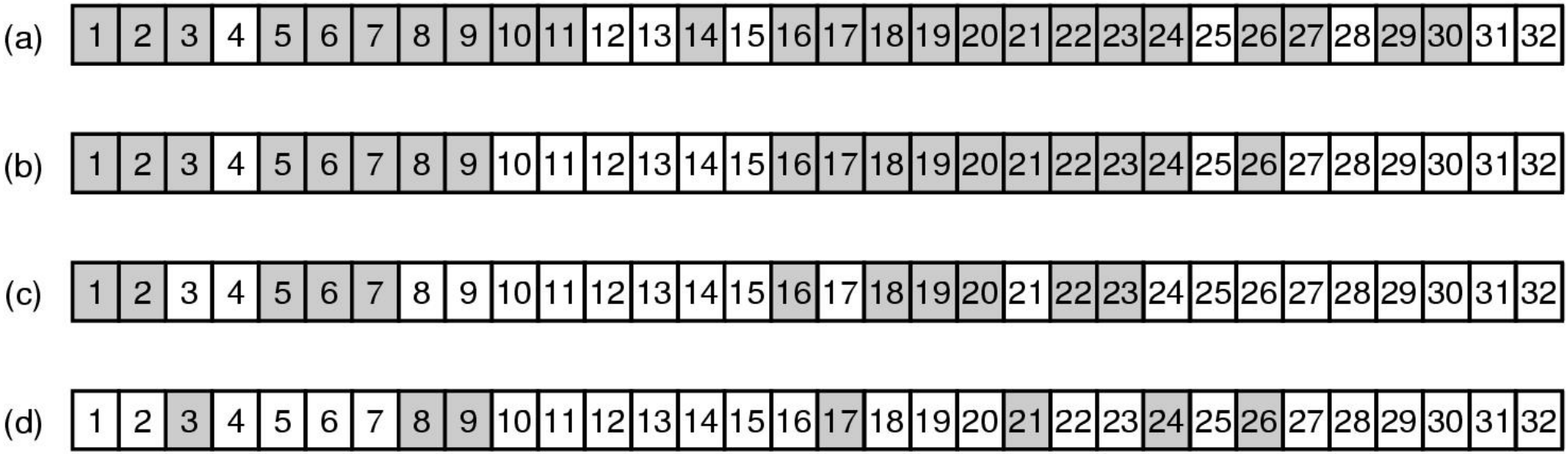

Open file table
Attributes
disk addresses
User = 8
Quota pointer
Quota table
Soft block limit
Hard block limit
Current # of blocks
Block warnings left
Soft file limit
Hard file limit
Current # of files
File warnings left
Quota
record
for user 8

Backing up a file system

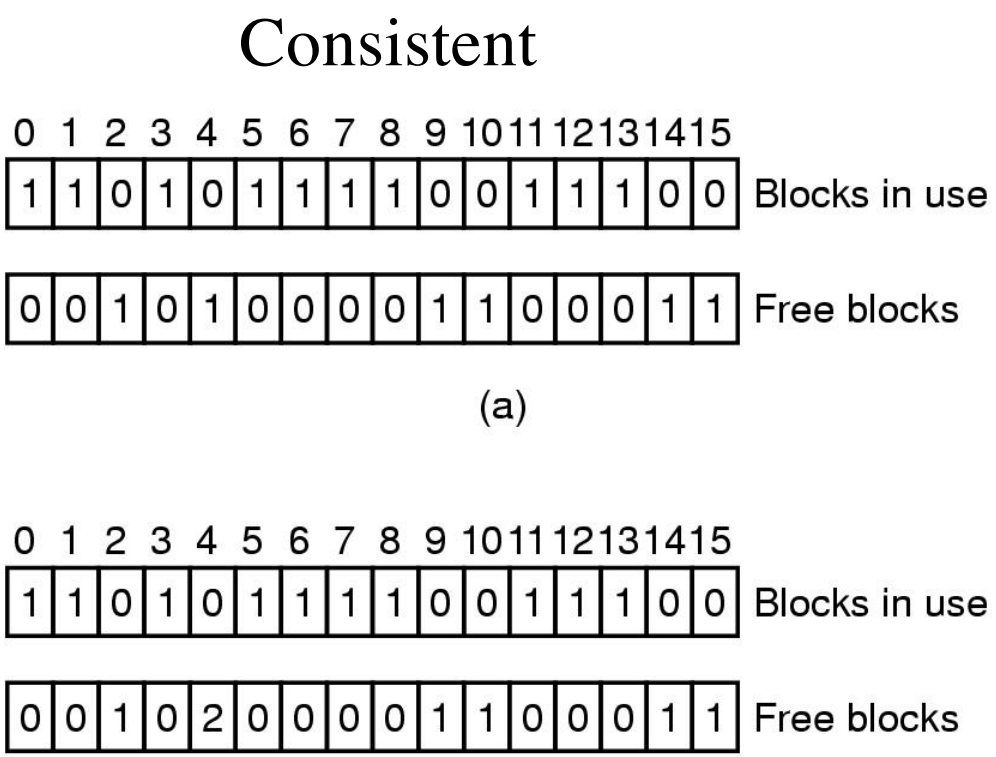
- A file system to be dumped
  - Squares are directories, circles are files
  - Shaded items, modified since last dump
  - Each directory & file labeled by i-node number

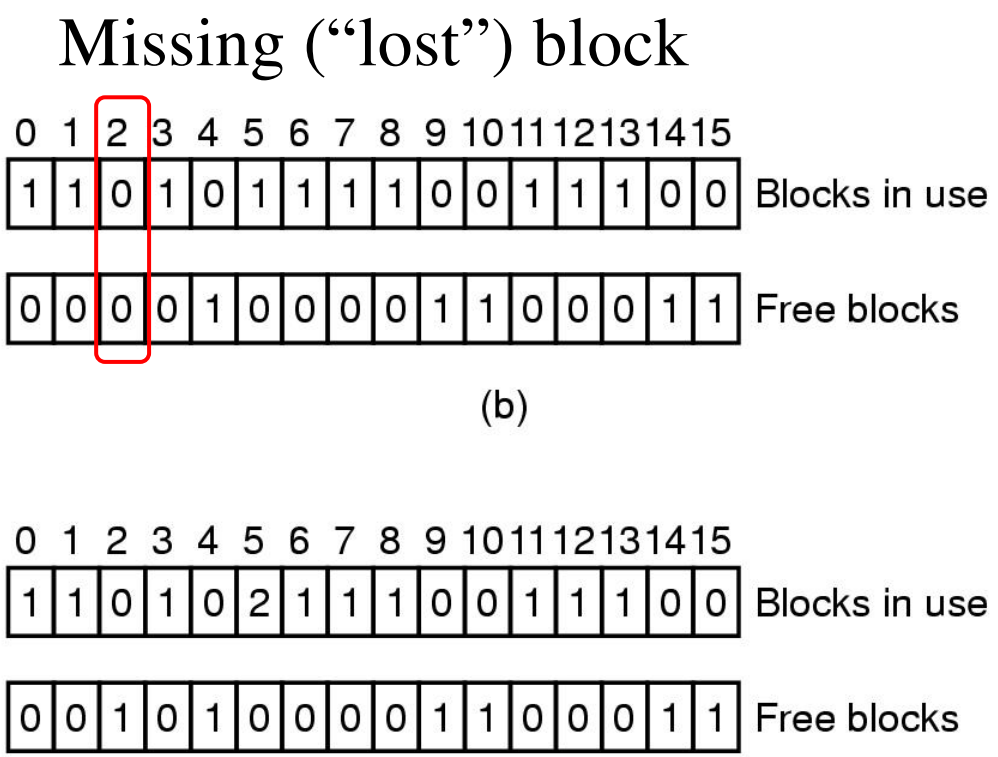


Bitmaps used in a file system dump



Checking the file system for consistency

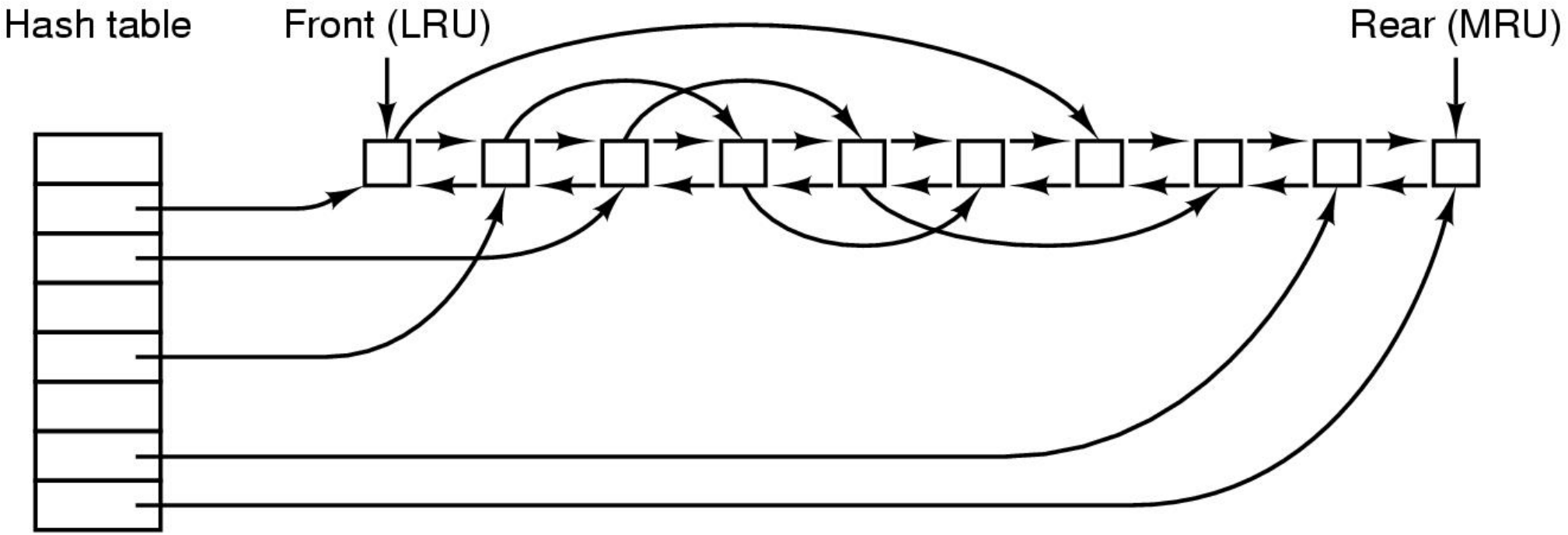




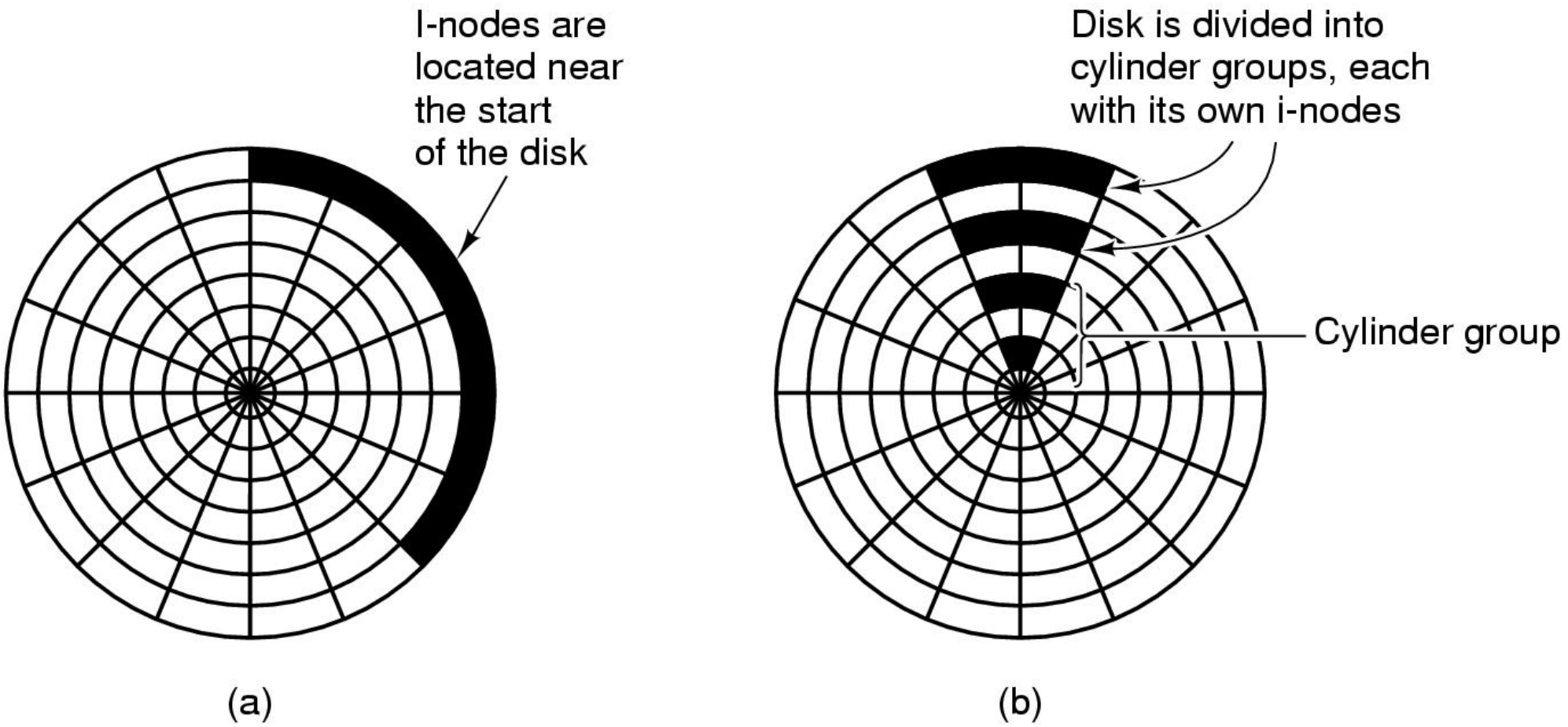
Duplicate block in free list

Duplicate block in two files

File block cache data structures



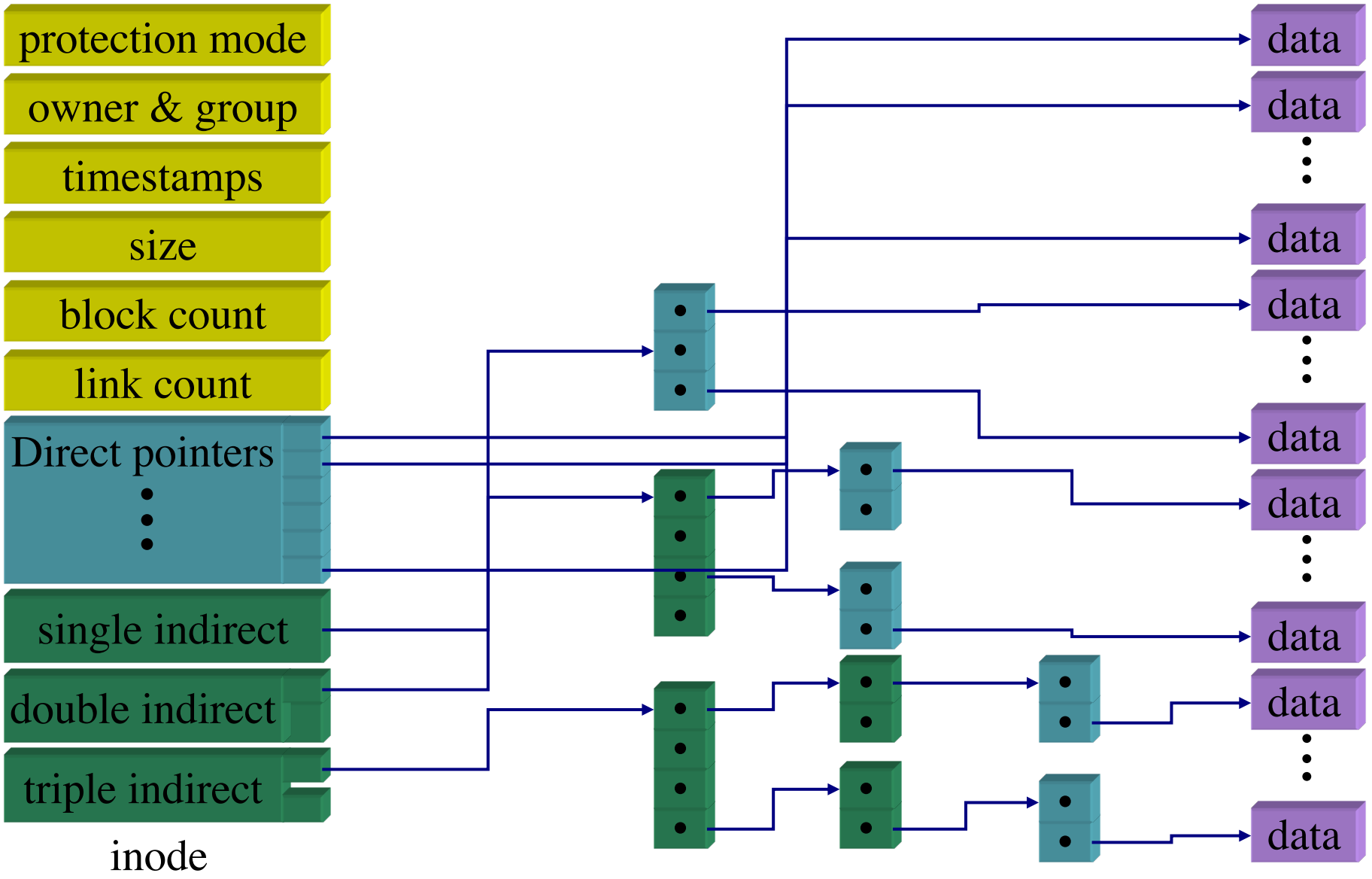
Grouping data on disk



Log-structured file systems

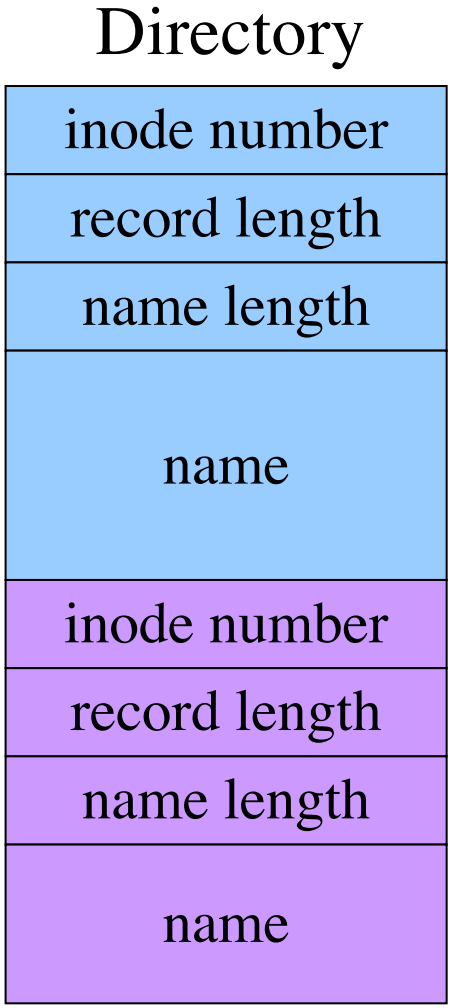
- Trends in disk & memory
  - Faster CPUs
  - Larger memories
- Result
  - More memory -> disk caches can also be larger
  - Increasing number of read requests can come from cache
  - Thus, most disk accesses will be writes
- LFS structures entire disk as a log
  - All writes initially buffered in memory
  - Periodically write these to the end of the disk log
  - When file opened, locate i-node, then find blocks
- Issue: what happens when blocks are deleted?

Unix Fast File System indexing scheme

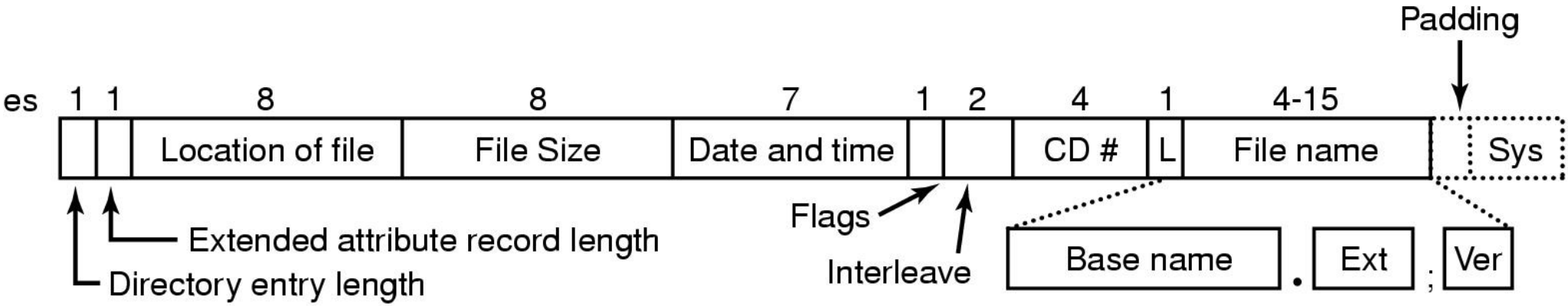


Directories in FFS

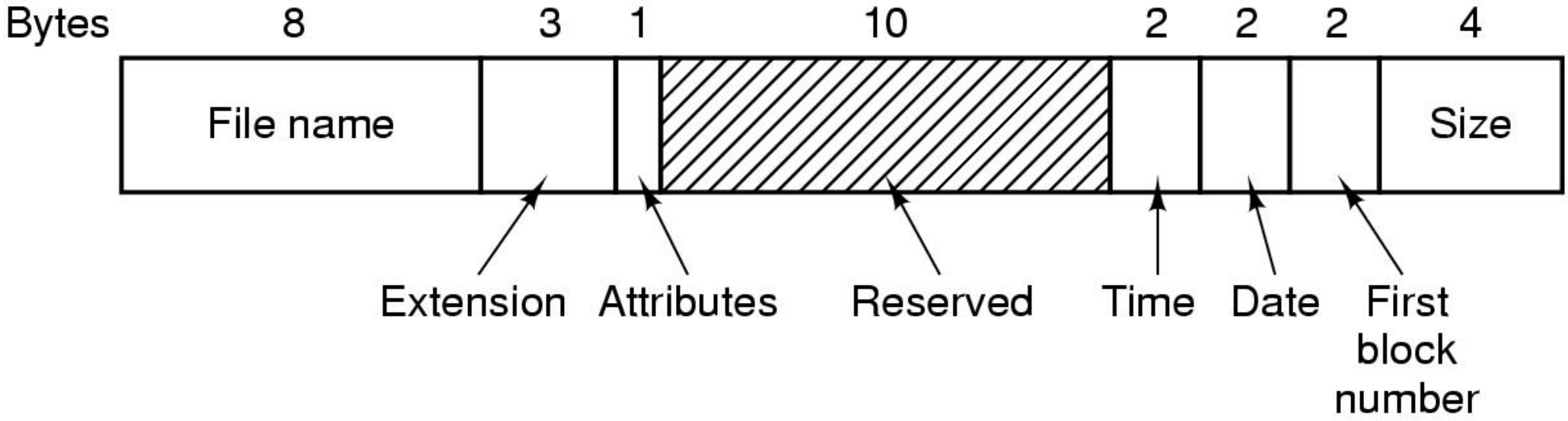
- Directories in FFS are just special files
  - Same basic mechanisms
  - Different internal structure
- Directory entries contain
  - File name
  - I-node number
- Other Unix file systems have more complex schemes
  - Not always simple files…



CD-ROM file system



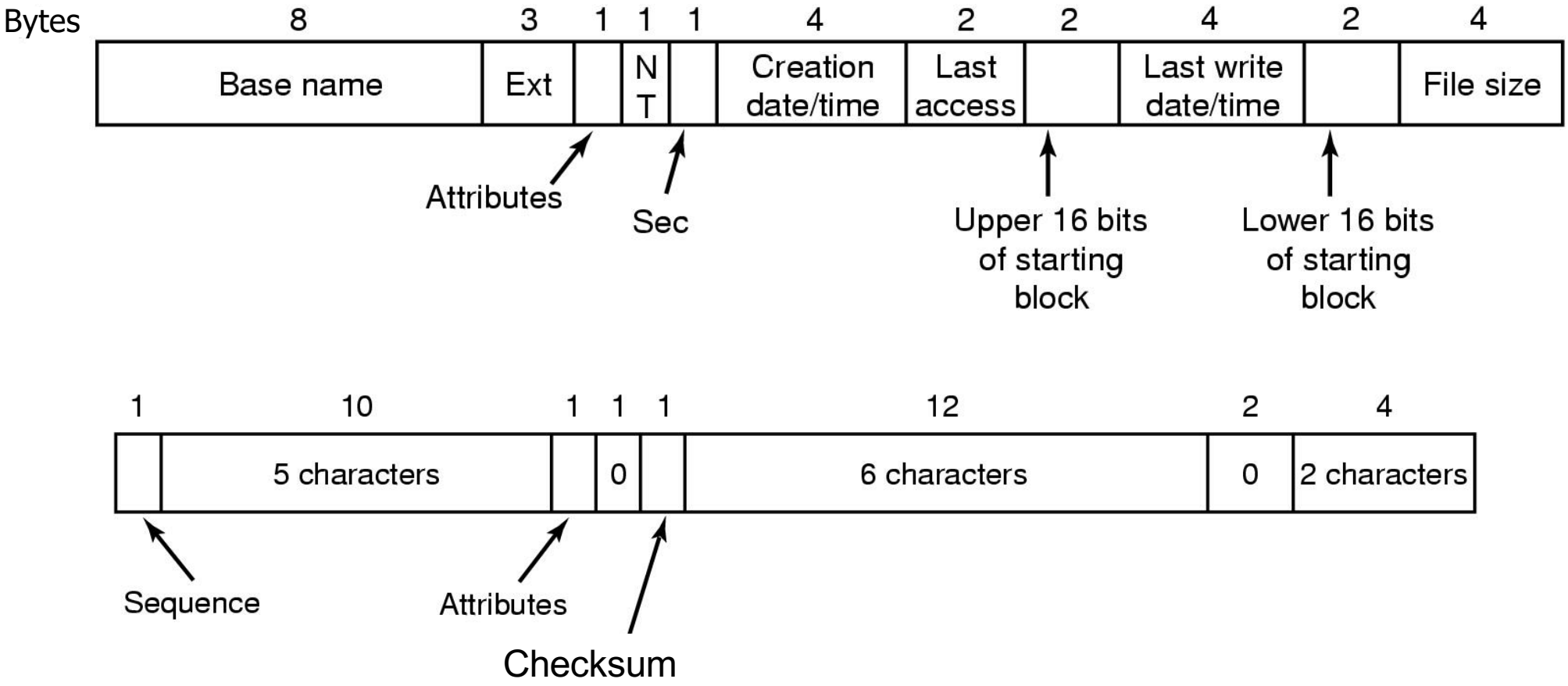
Directory entry in MS-DOS



MS-DOS File Allocation Table

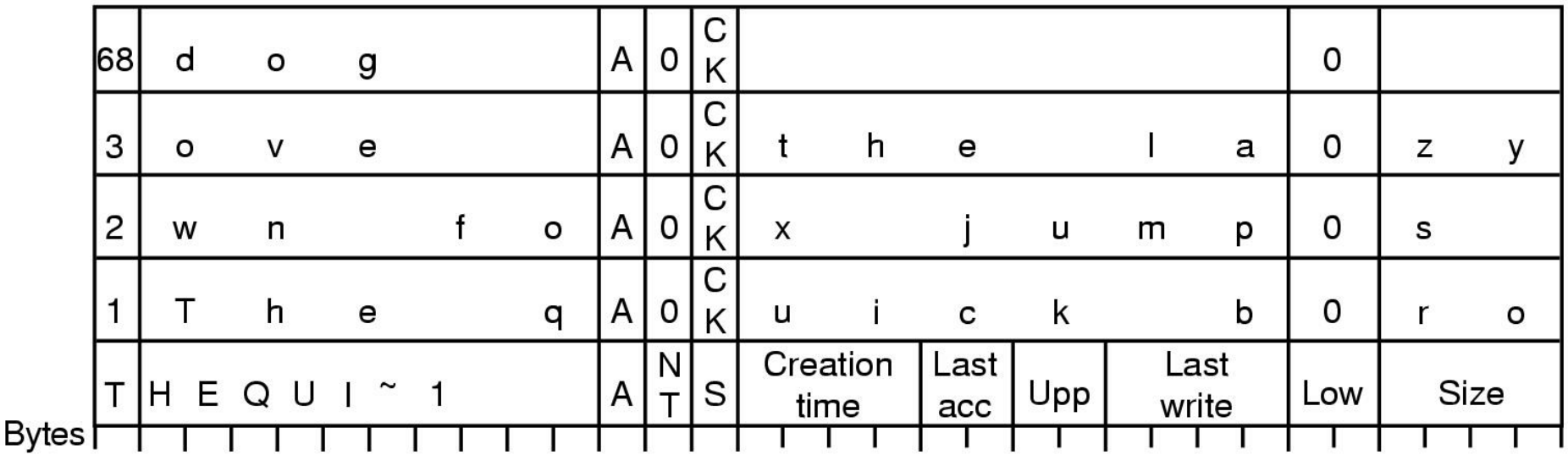
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

FAT32 Directory and Filename

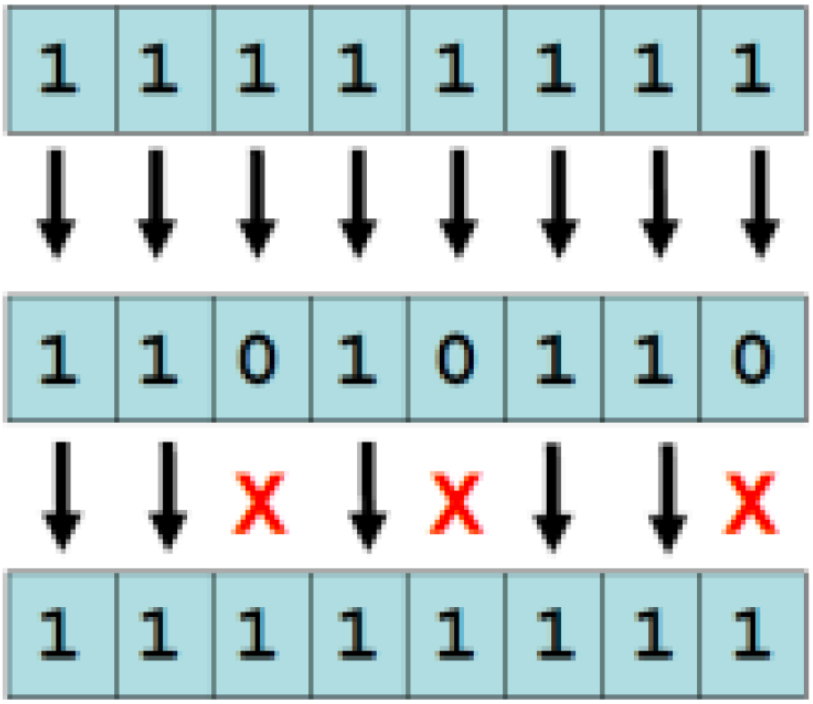


Storing a long name in FAT32

- Long name stored in Windows 98 so that it's backwards compatible with short names
  - Short name in "real" directory entry
  - Long name in "fake" directory entries: ignored by older systems
- OS designers will go to great lengths to make new systems work with older systems…



Flash File System



Start off with all 1's

Programming from 1's to 0's is allowed

Programming from 0's to 1's is not allowed

Wear Leveling

Count total writes per flash sector and attempt to balance across the whole disk