Linux kernel map

2.6.36

functionalities
layers

human interface | system | processing | memory | storage | networking

user space interfaces
system calls and system files

HI char devices
cdev_add
input_fops
video_fops
snd_fops
console_fops
sys_capset
fb_fops
sys_syslog

interfaces core
kernel/
System Call Interface
system files
linux/syscalls.h
linux/uaccess.h
copy_from_user
/proc /sysfs /dev
sysfs_ops
register_chrdev
sys_ioctl
cdev_add
/sys/class/
cdev_map
sys_epoll_create
cdev
sys_reboot
sys_init_module
kvm_dev_ioctl

processes
kernel/
fs/exec.c
kernel/signal.c
sys_fork
sys_kill
sys_signal
sys_vfork
sys_execve
do_sigaction
sys_clone
linux_binfmt
sys_gettimeofday
sys_time
sys_times
sys_futex
sys_nanosleep

memory access
mm/
sys_brk
sys_mmap
shm_vm_ops
sys_sysinfo
sys_shmctl
sys_shmat
si_meminfo
sys_mprotect
sys_mincore
/dev/mem
mem_fops
mmap_mem
/proc/meminfo
/proc/self/maps
sys_msync

files & directories access
fs/
sys_chroot
sys_pivot_root
sys_mkdir
sys_chdir
sys_open
sys_read
sys_tee
sys_poll
sys_write
sys_pipe
sys_select
do_path_lookup
sys_getdents
sys_flock
sys_readv
sys_splice
sys_swapon
sys_sendfile
alloc_file
sys_chown
sys_chmod
fd
sys_newfstat
notify_change
fget
sys_fsync
fsnotify_change
sys_sync
sys_mount
inode_setattr
sys_sysfs

sockets access
net/
sys_socketcall
sys_socket
sys_connect
sys_accept
sys_bind
sys_listen
sys_sendmsg
sys_recvmsg
sys_setsockopt
/proc/net/
tcp4_seq_show
sg_proc_seq_show_dev
rt_cache_seq_show
sock_ioctl

virtual

security
security/
security_capset
may_open
linux/security.h
security_socket_create
inode_permission
security_inode_create
security_ops
selinux_ops

Device Model
drivers/base/
driver_init
kobject
linux/kobject.h
kset
bus_register
linux/device.h
bus_type
device_create
device
class
device_type
driver_register
device_driver
probe
kvm
load_module
module
module_param
kernel_param
kobject_uevent_init
kobject_uevent

threads
schedule_work
create_workqueue
INIT_WORK
queue_work
work_struct
workqueue_struct
kthread_create
kernel_thread
current
thread_info
do_fork

virtual memory
vmalloc_init
find_vma_prepare
vmalloc
vfree
vmlist
vm_struct
virt_to_page

Virtual File System
file
file_systems
vfs_fsync
vfs_read
vfs_fstat
vfs_write
vfs_getattr
vfs_create
inode
fsnotify
inode_operations
file_operations
file_system_type
get_sb
super_block
do_splice_direct
generic_file_aio_read

protocol families
inet_init
__sock_create
socket
inet_family_ops
inet_create
unix_family_ops
proto_ops
inet_dgram_ops
inet_stream_ops
socket_file_ops

bridges
cross-functional modules

debugging
sys_ptrace
log_buf
register_kprobe
printk
handle_sysrq
oprofile_start
kgdb_breakpoint
oprofile_init

synchronization
lock_kernel
mutex_lock_interruptible
mutex_unlock
kernel_flag
mutex
wait_for_completion
owner
complete
add_timer
timer_list
down_interruptible
run_timer_softirq
up
semaphore
wait_event
wake_up
msleep
spin_lock_irqsave
spin_unlock_irqrestore

memory mapping
mm/mmap.c
do_mmap
do_mmap_pgoff
kmem_cache_alloc
vma_link
mm_struct
vm_area_struct

page cache
address_space
bdi_writeback_thread
do_writepages

swap
si_swapinfo
swap_info
kswapd
do_swap_page
wakeup_kswapd

networking storage
nfs_file_operations
smb_file_type
cifs_file_ops
iscsi_tcp_transport

socket splice
sock_sendpage
tcp_sendpage
udp_sendpage
sock_splice_read
tcp_splice_read

logical
functions implementations

HI subsystems
oss
alsa
video_device
mousedev_handler
tty

system run
init/
kernel/
boot, shutdown
power management
init/main.c
start_kernel
mount_root
do_initcalls
run_init_process
kernel_restart
kernel_power_off
hibernate
machine_ops

Scheduler
kernel/sched.c
task_struct
schedule_timeout
schedule
setup_timer
process_timeout
activate_task
rq
context_switch

logical memory
physically mapped memory
mm_init
kmalloc
kfree
pgd_t
pmd_t
pte_t

logical file systems
ext4_file_operations
ext4_get_sb
ext4_readdir

protocols
proto
/proc/net/protocols
udp_prot
tcp_prot
udp_recvmsg
udp_sendmsg
tcp_recvmsg
tcp_sendmsg
tcp_v4_rcv
udp_rcv
tcp_transmit_skb
NF_HOOK
nf_hooks
ip_queue_xmit
ip_local_deliver
ip_push_pending_frames
ip_route_input
alloc_skb
ip_rcv
sk_buff
ip_output

device control

drivers/

abstract devices and HID class drivers
drivers/input/
drivers/media/
sound/
console
kbd
fb_ops
mousedev
drm_driver

generic HW access
request_region
pci_driver
request_mem_region
pci_register_driver
pci_request_regions
usb_driver
usb_submit_urb
usb_hcd_giveback_urb
ioremap
usb_hcd

interrupts core
request_irq
tasklet_struct
jiffies_64++
tasklet_action
do_timer
setup_irq
tick_periodic
timer_interrupt
do_softirq
do_IRQ
irq_desc
softirq_init

Page Allocator
/proc/slabinfo
mm/slob.c
mm/slub.c
mm/slab.c
__free_pages
kmem_cache
__free_one_page
kmem_cache_init
kmem_cache_alloc
__get_free_pages
page
__alloc_pages
vm_stat
totalram_pages
try_to_free_pages

block devices
block/
gendisk
block_device_operations
request_queue
init_scsi
scsi_device
scsi_driver
sd_fops
usb_storage_driver
usb_stor_host_template

network interface
linux/netdevice.h
netif_receive_skb
dev_queue_xmit
netif_rx
register_netdev
net_device
dev_ioctl
alloc_netdev_mq
ieee80211_alloc_hw
ether_setup
ieee80211_rx
netif_carrier_on
ieee80211_xmit

hardware interfaces
drivers, registers and interrupts

HI peripherals device drivers
uvc_driver
ac97_driver
vga_con
atkbd_drv
i8042_driver
psmouse
drivers/media/video/

device access and bus drivers
ehci_irq
writew
readw
ehci_urb_enqueue
outw
inw
usb_hcd_irq
pci_read
pci_write

CPU specific
setup_arch
x86_init
trap_init
early_trap_init
start_thread
native_init_IRQ
switch_to
/proc/interrupts
set_intr_gate
system_call
show_regs
interrupt
pt_regs
atomic_t
cli
sti

physical memory operations
arch/x86/mm/
mem_init
get_page_from_freelist
zonelist
zone
free_area
free_list
out_of_memory
die
num_physpages
do_page_fault

disk controller drivers
scsi_host_alloc
libata
Scsi_Host
ahci_pci_driver
aic94xx_init

network device drivers
drivers/net/
usbnet_probe
ipw2100_pci_init_one
zd1201_probe
e1000_xmit_frame
e1000_intr

electronics

user peripherals
keyboard
camera
mouse
graphics card
audio

I/O
I/O mem
I/O
I/O ports
ACPI
USB controller
PCI controller

CPU
registers
APIC
interrupt controller

memory
RAM
DMA
MMU

disk controllers
SCSI
SATA

network controllers
Ethernet
WiFi

File Systems

- Files
- Directories
- Implementation
- Examples

Long-Term Information Storage: Goals

- Must store large amounts of data
- Information must survive process termination
- Multiple processes must have concurrent access to information

What a file is/is not

- File: a named collection of bytes stored on disk
  - contained within its own contiguous logical address space
- User view:
  - collection of records
- OS view:
  - bunch of blocks stored on a device
  - File system packs bytes into disk blocks on write
  - File system unpacks bytes from disk blocks on read

Why files?

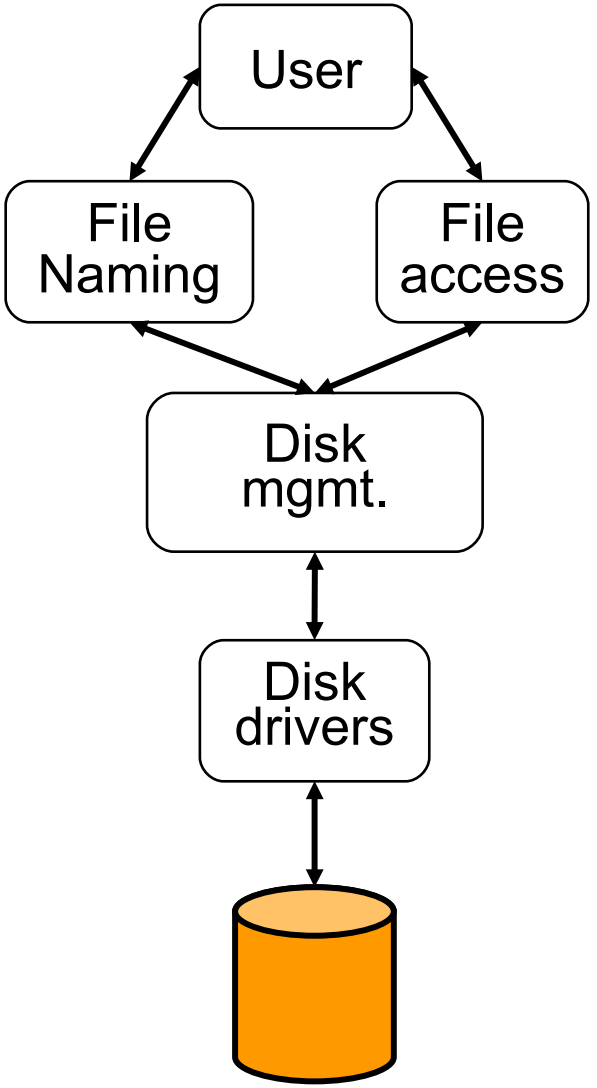
- Physical reality
  - block oriented
  - physical sector numbers
  - no protection between users
  - data corruption due to crashes, etc.
- File system abstraction
  - byte oriented
  - named files
  - users protected
  - some recovery possible after failure

Sounds simple, what's the catch?

- Files grow and shrink
  - sizes span orders of magnitude
- Little/no a-priori knowledge about contents
- Users desire efficiency
- Must overcome disk performance
  - it is awful
- Must cope with failure

High-Level Components/Abstractions

- Disk management
  - arrange collection of disk blocks into files
- Naming
  - allow user to name files, rather than give sector/track numbers, to locate data
- Security
  - keep information secure
- Reliability/durability
  - ensure crashes aren't catastrophic

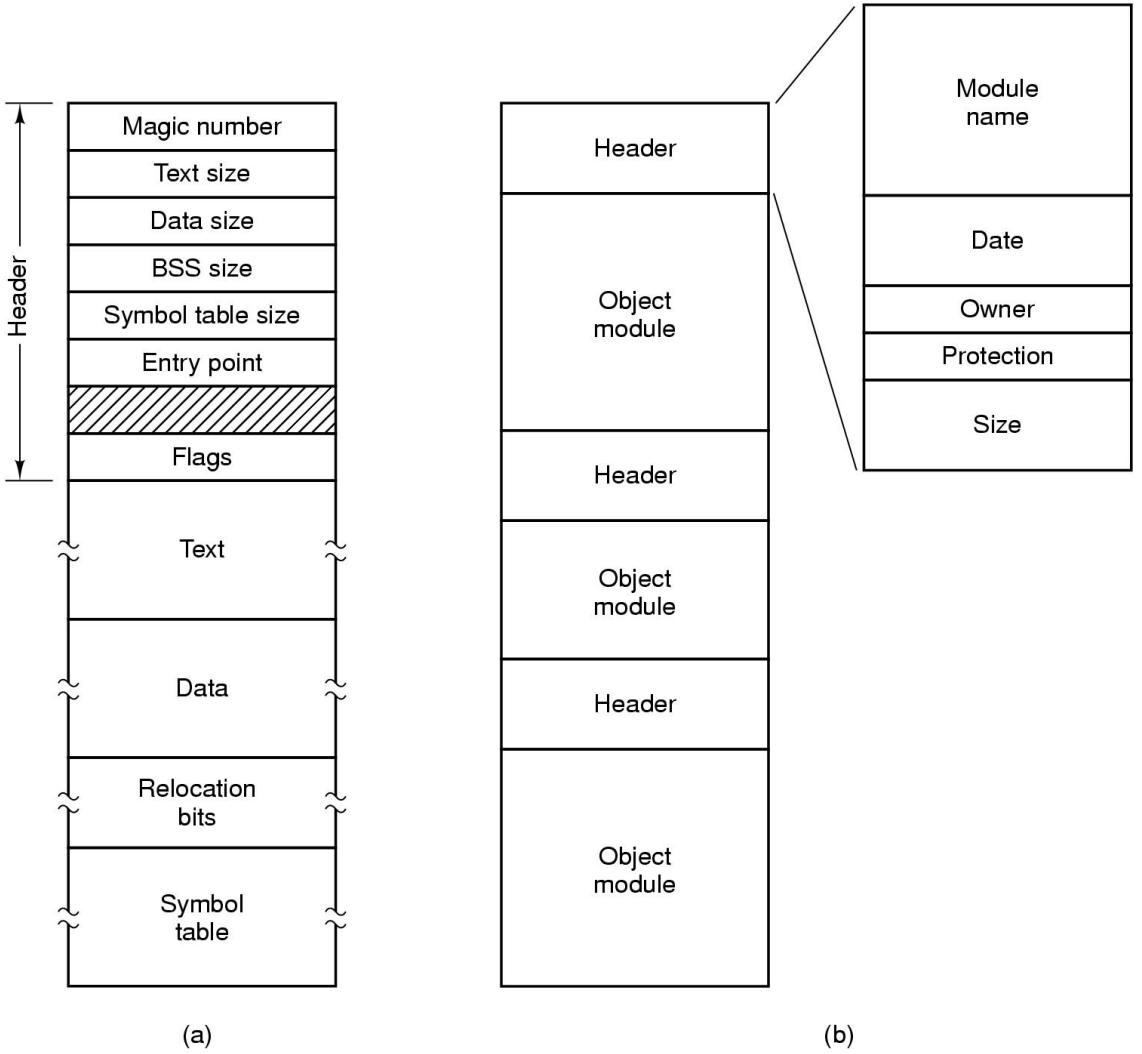


File Structure

- None
  - sequence of words/bytes
- Simple
  - lines
  - fixed/variable length
- Complex
  - formatted document
  - relocatable load file, etc
- Who decides?
  - OS and programs

Examples of structured files

- Executable (a) vs archive (b)



Sampling of file types (by extension)

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

Sampling of 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

List of basic file operations

- Open/close
- Create/delete
- Read/write/append
- Seek
- Get/set attributes
- Rename

FS System Calls example (1)

```
/* 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 */
```

FS System Calls example (2)

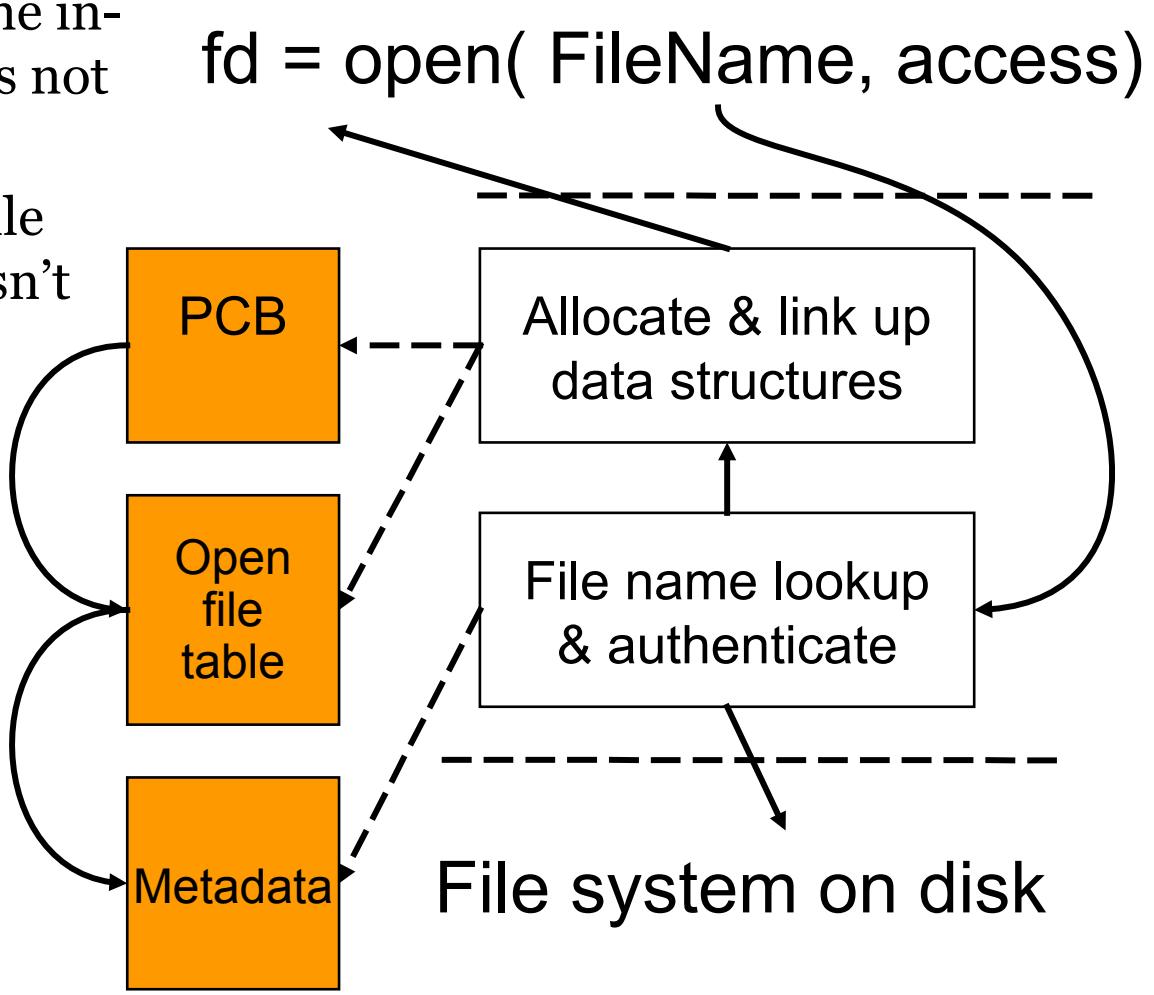
```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);     /* open the source file */
if (in_fd < 0) exit(2);                          /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
if (out_fd < 0) exit(3);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

Open Files

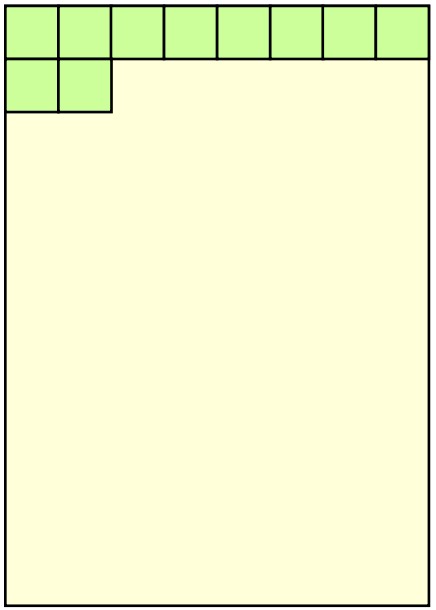
- Need several pieces of information to manage open files
  - file pointer: pointer to last read/write location, for each process which has opened the file
  - file-open count: how many times has this file been opened
    - needed to remove file from open-file table when last process closes it
  - disk location: where is the file?
  - access rights: per-process access mode information
- Some systems allow open file locking
  - mediated access to files ^ limit concurrent access

Opening A File

- File name lookup and authentication
- Copy the file metadata into the in-memory data structure, if it is not in yet
- Create an entry in the open file table (system wide) if there isn't one
- Create an entry in PCB
- Link up the data structures
- Return a pointer to user



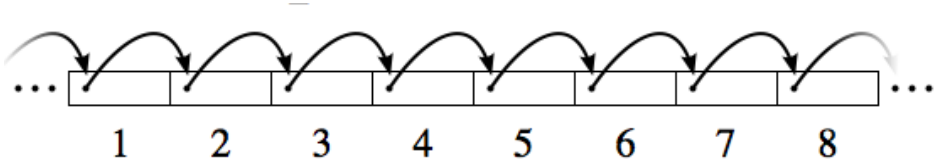
Open / Create / Write Example



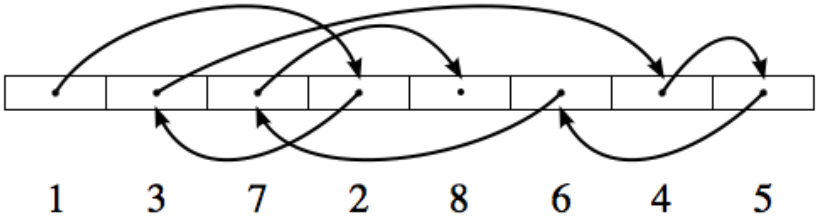

```
ThreadA(){
  FILE* f;
  int i=0;
  CREAT("file");
  f=FOPEN("file");
  for(;i<10;i++){
    FWRITE(f,i);
  }
  FCLOSE(f);
}
```

File access types

- Sequential
  - read all bytes/records from beginning
  - no jumping around



- Random
  - read bytes/records in any order

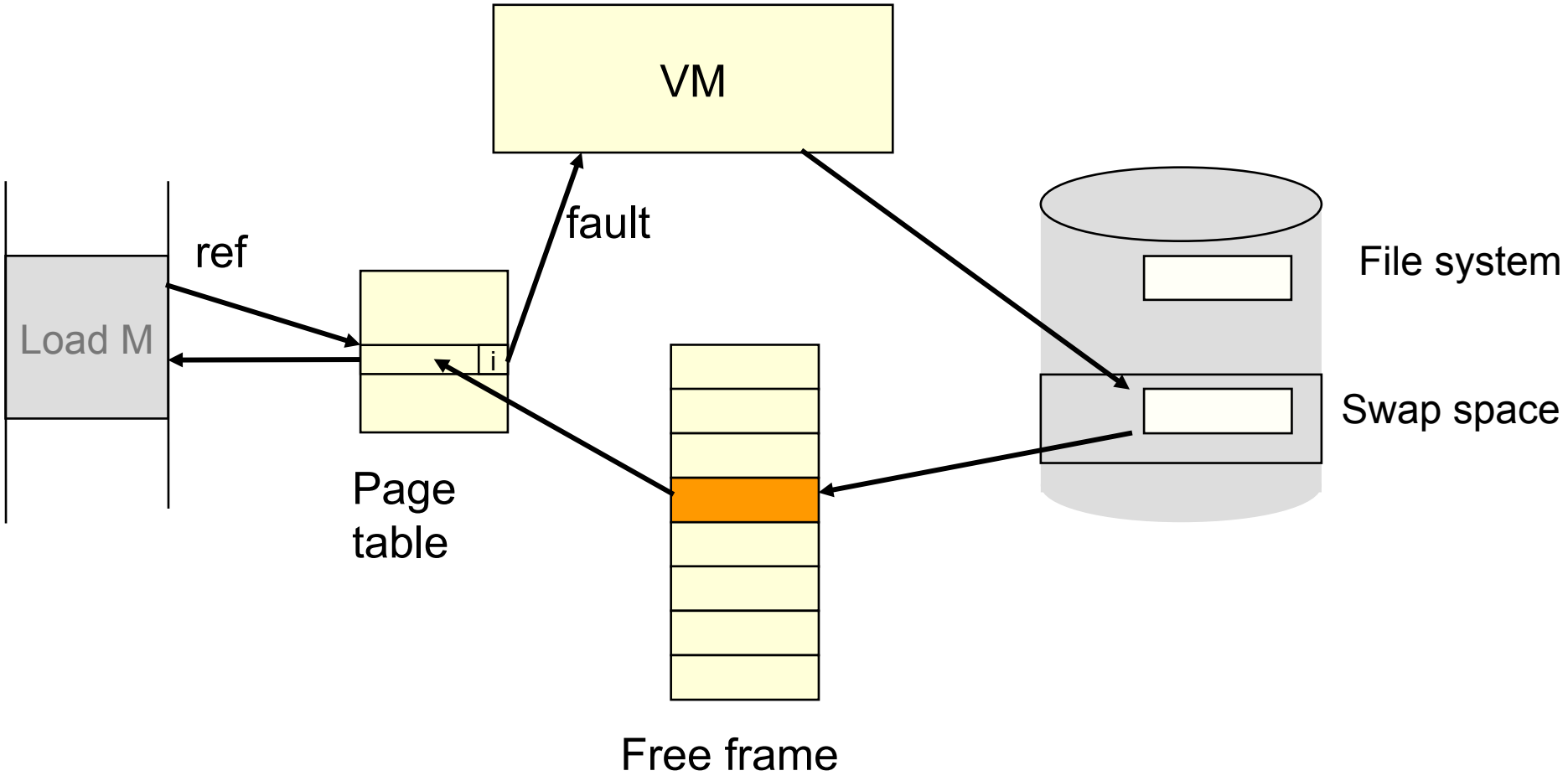


File system interfaces

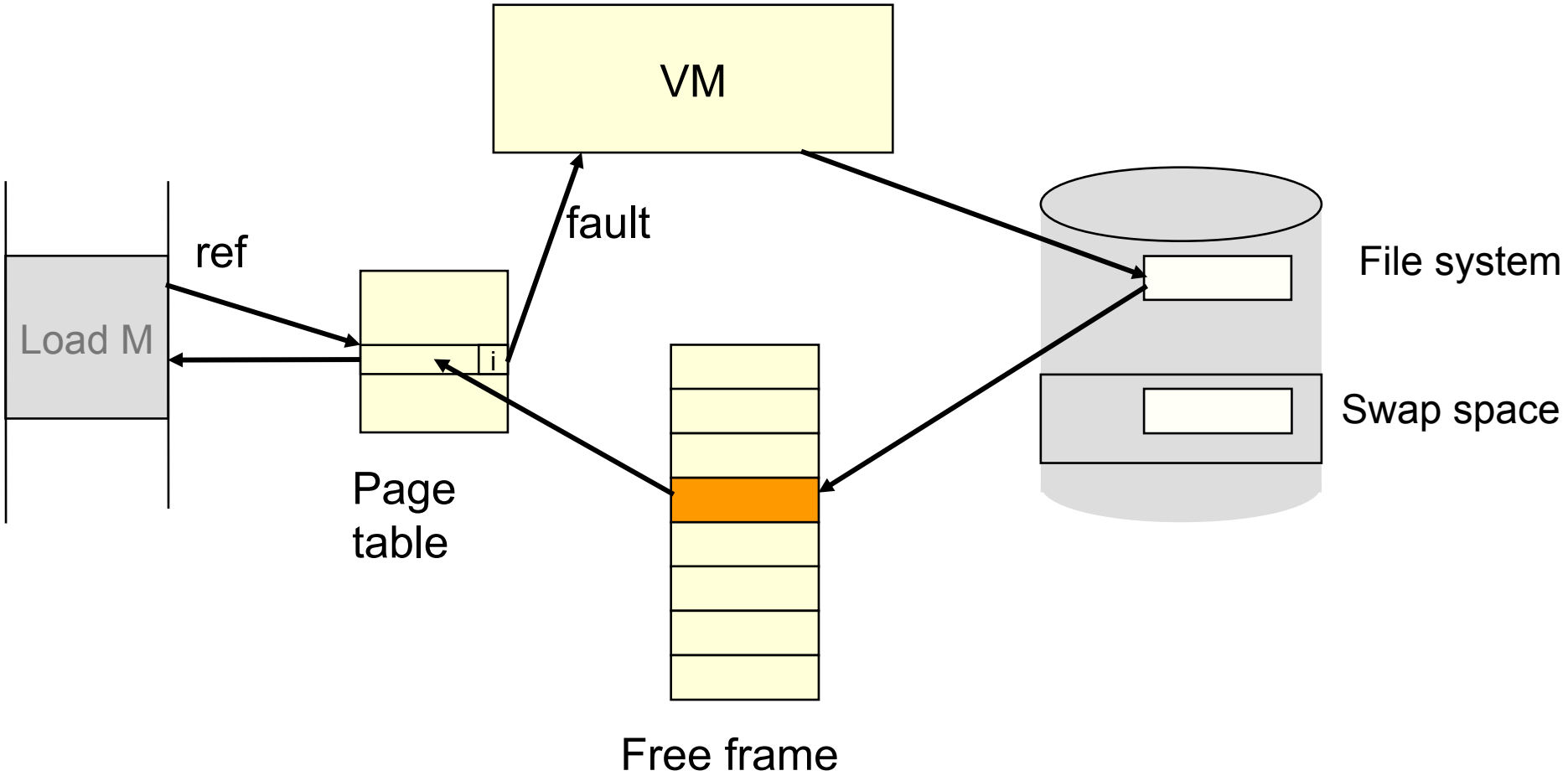
- Typically, explicit read/write operations
  - file data explicitly copied between disk file and process mem.
  - programs do not directly access file data

- Memory-mapped files
  - map files into process virtual memory
  - directly access file data through regular memory access ops.

Remember Demand Paging in VM?



Memory-Mapped Files



Memory-Mapped Files: properties

- File mapped by initializing virtual memory so that the file directly serves as the backing store for a region in the process' address space
  - Hence no explicit copying
  - Hence direct access
- Processes mapping the same file share physical memory that caches file data

Memory-Mapped Files: example

- use mmap()/munmap()

```
fd = open(argv[1], O_RDWR);
fstat(fd, &sb);
addr = mmap(NULL, sb.st_size, PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, 0);
int i;
for (i = 0; i < 10; i++)
  printf("%c", addr[i]);

munmap(addr, sb.st_size);
fclose(fd);
```

Memory-Mapped Files: overview

- Benefits
  - explicit file I/O instructions not needed
  - simple to program
    - with possibly different semantics
      e.g. read() copies data ^ local modifications are private
      mmap() does not ^ must explicitly copy to make modifications private
- Caveat
  - mmap() only works on page-size granularity
    - can't really append to files this way (OS doesn't know where the file ends, only that it uses a given page)

File Systems

- Files
- Directories
- Implementation
- Examples

Directory Organization

- Directory = system file
  - keeps track of other files
- Possible approaches to directories
  - each place in directory holds info about a file and pointer to address where file data is stored
  - each place in directory holds a file name and pointer to a data structure, which then holds info about the file and pointers to data location

Directory Operations

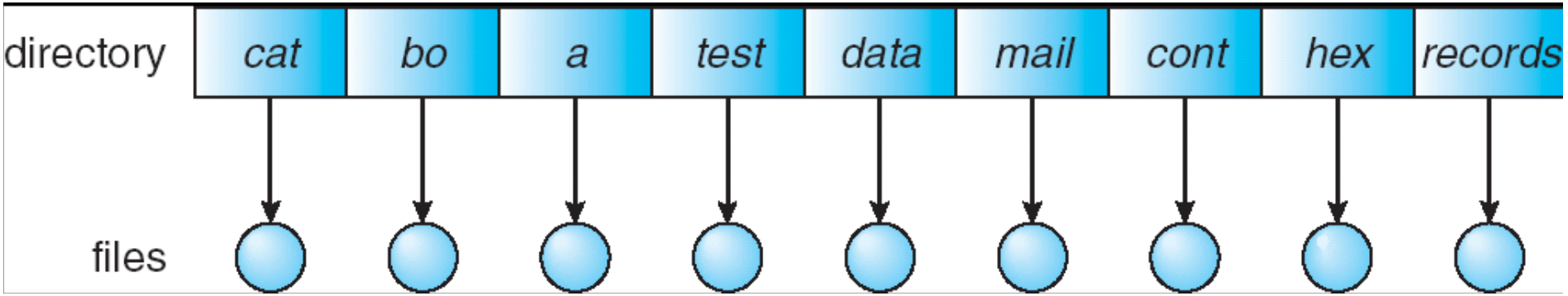
- Create/delete
- Open/close
- Read
- Rename
- Link/unlink

Directory Organization: goals

- Efficiency
  - can you locate a file quickly?
- Naming
  - user convenience
  - users can have different names for same file/same names for different file
- Grouping
  - allows for grouping of files by some properties (code, utilities, games, questionable pictures, etc.)

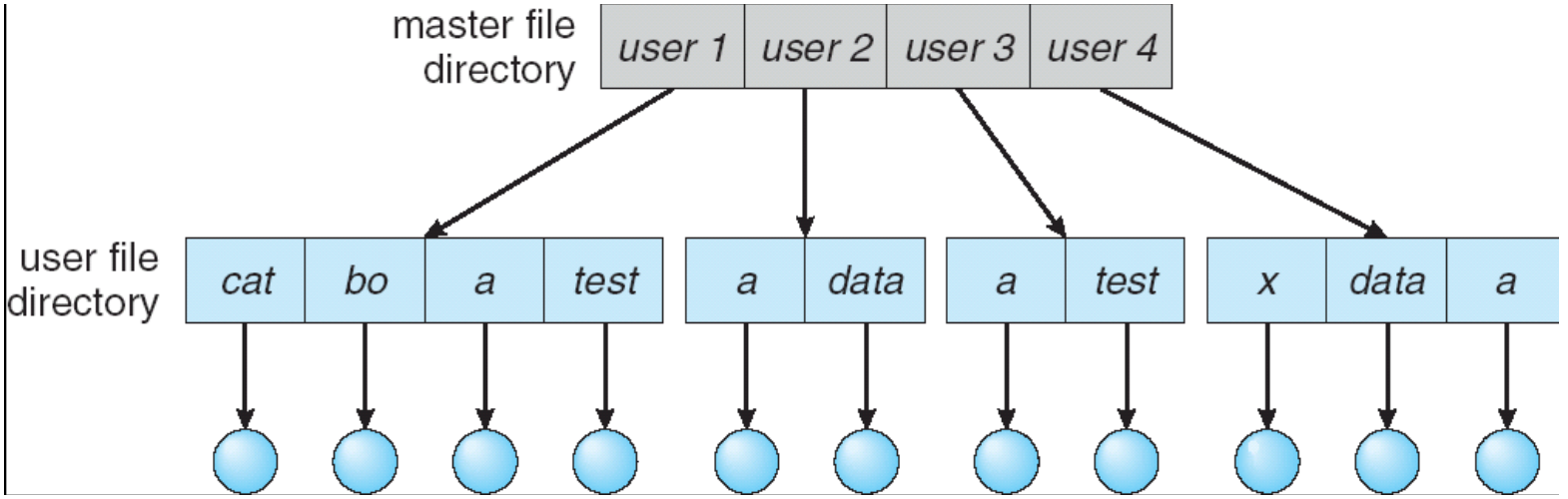
Single-Level Organization

- All users see single directory
  - naming is inconvenient
  - grouping is a problem
  - but, if there are no users, who cares?



Two-Level Organization

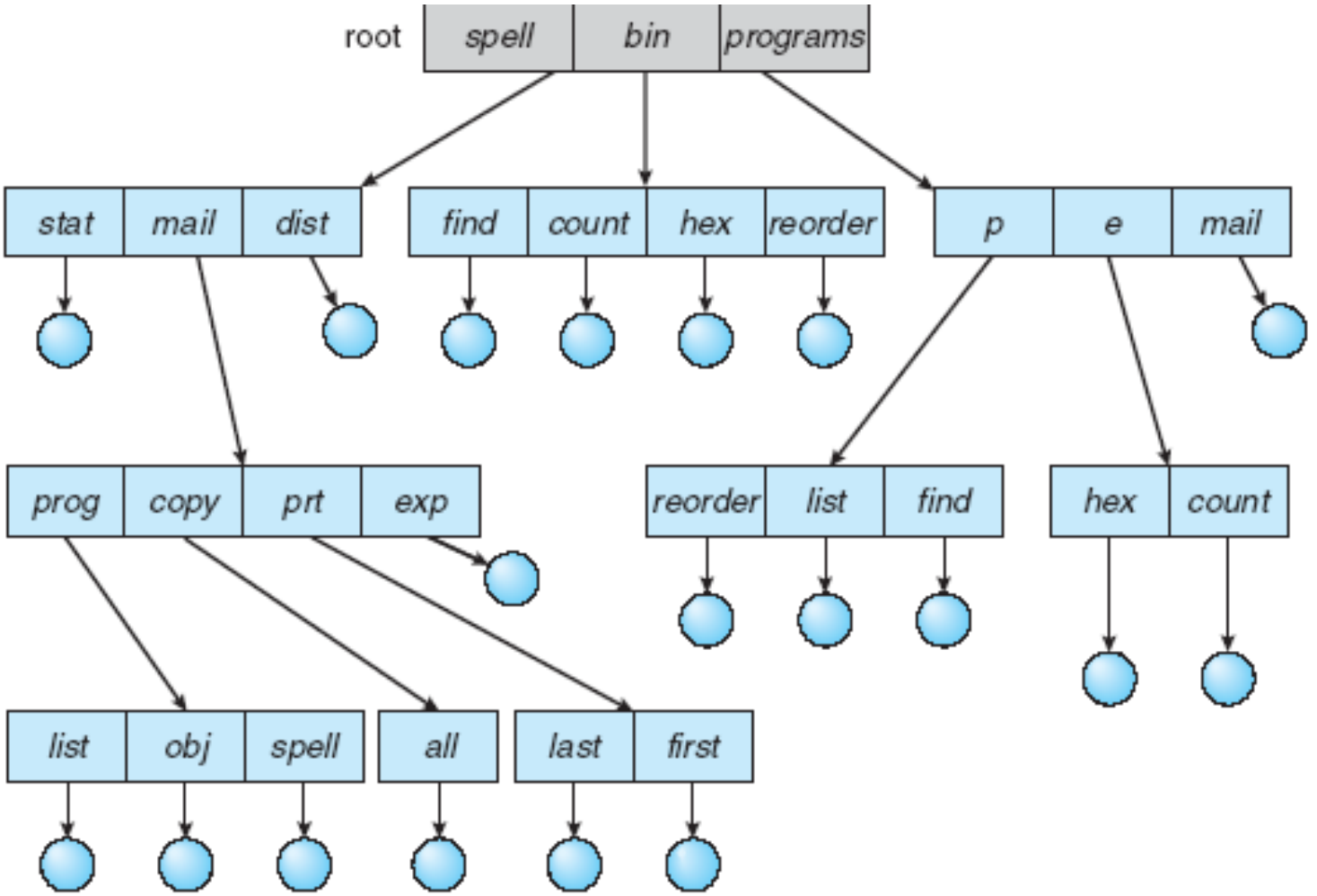
- Separate directory for each user
  - can have same/different filenames for different/same files
  - searching is (more) efficient
  - single user's files are still in one directory



Tree-Structured Organization

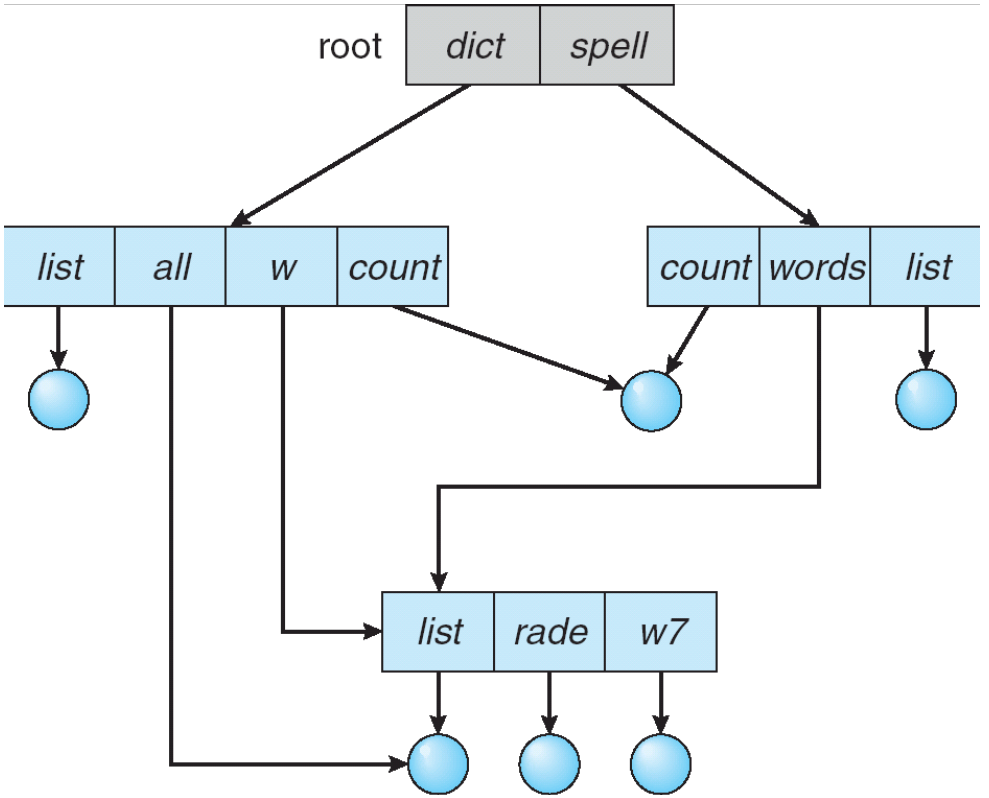
- Directories
  - contain files, directories
  - arbitrary grouping ^ flexibility
  - easily expand to hold large number of files
  - drawback: more work to find stuff
- File naming looks like: /dir1/dir2/dir3/filename
  - that's an absolute pathname… describes exact location starting at root
  - relative pathname:
    - if you are “in” dir2, then dir3/filename is location of file
    - note no leading '/'

Tree-Structured Organization: example



Linking

- Link: another name/pointer to existing file
- Resolving a link: following pointer to locate the file
- Linking
  - saves space
  - makes updates easier
- Hard links
  - links file into a directory structure
- Soft links
  - just a pointer to a file (e.g. shortcut in windows)

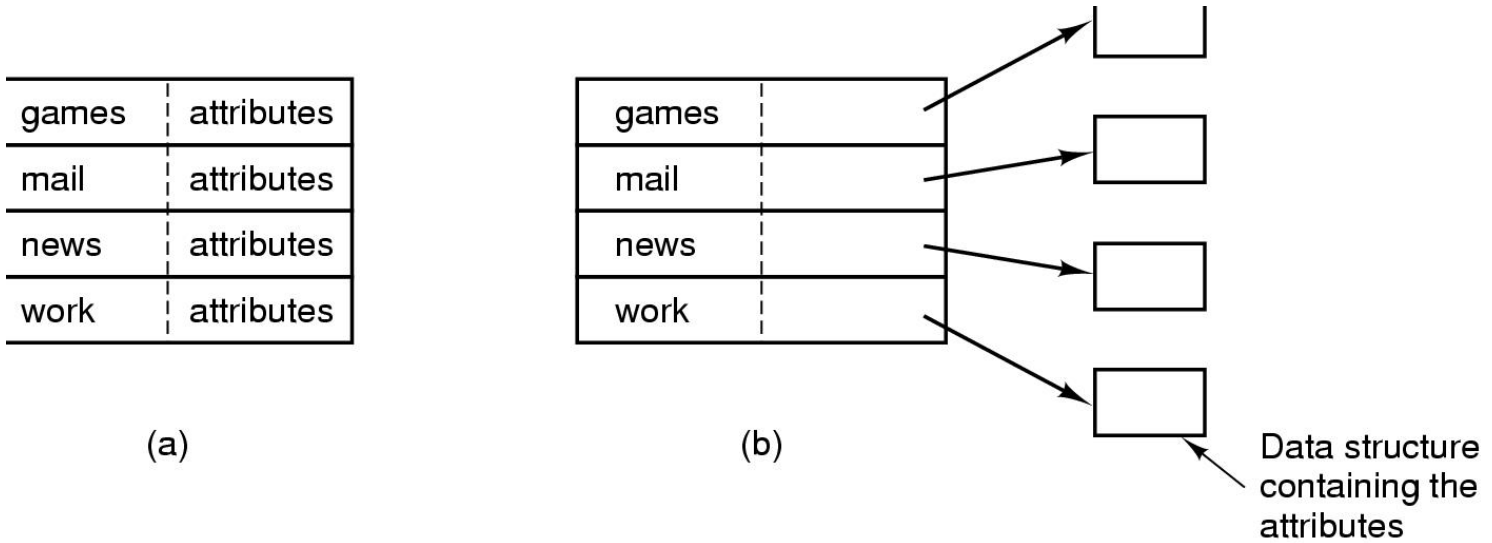


Implementing directories

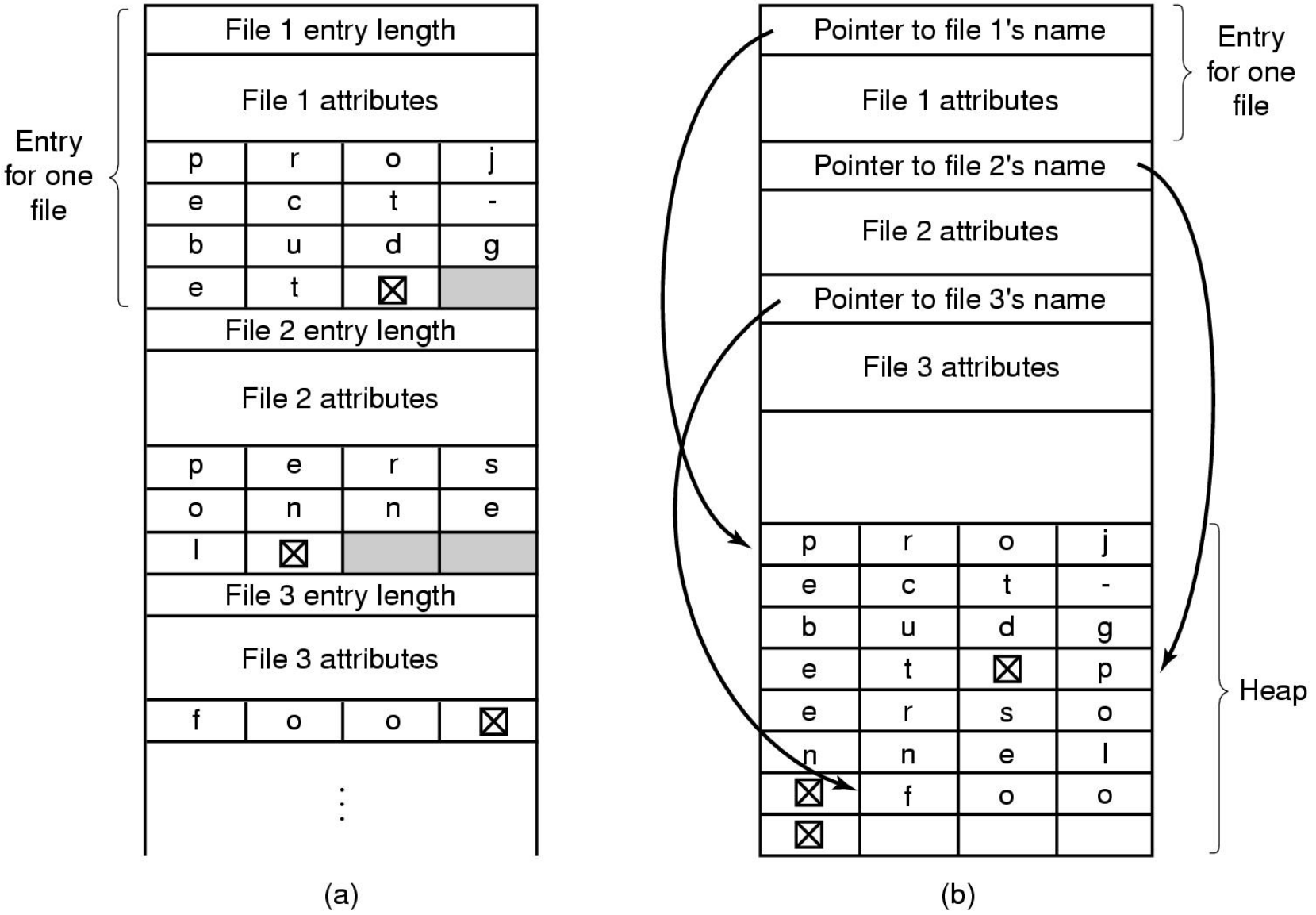
- Linux ^ specially formatted file
  - can read it just like a file
  - it's a directory only because we treat it that way
- Contains names of files
  - <name, fd_index> pairs in no particular order
  - fd_index may be another directory
  - root has no name

Possible approaches redux

- Possible approaches to directories
  - each place in directory holds info about a file and pointer to address where file data is stored
  - each place in directory holds a file name and pointer to a data structure, which then holds info about the file and pointers to data location



Filename handling



So, how might this work…

- Finding '/dir1/dir2/dir3/readme.txt':
  - Fetch root inode
    - Start loading root directory data blocks
    - Walk directory data until you find dir1
    - Get inode # for dir1 from directory file
  - Fetch dir1's inode
    - Start loading dir1's directory data blocks
    - Walk directory data until you find dir2
    - Get inode # for dir2 from directory file
    - …
  - Repeat process until you have inode # for file

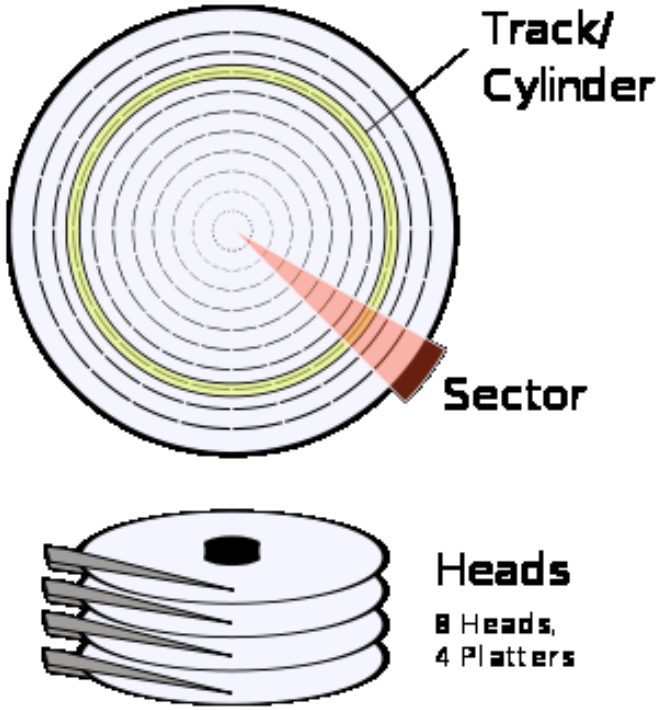
Observations

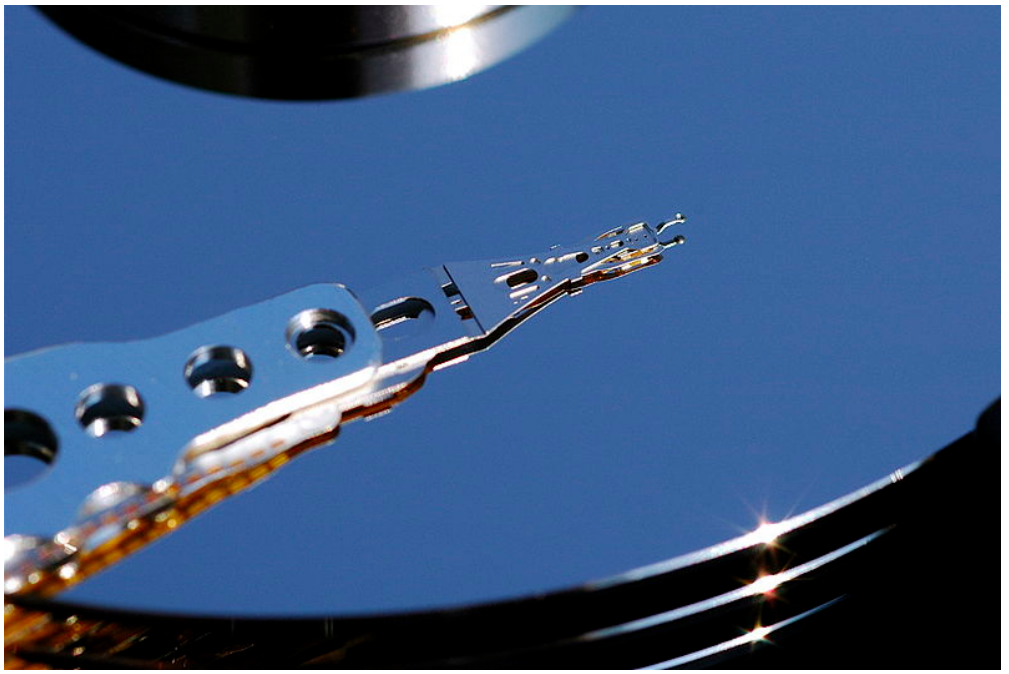
- Name lookup is slow
  - lots of CPU time
  - lots of fetches (inodes + data blocks)
  - but… also lots of repetition
- So
  - cache name lookups
  - cache recently used inodes and directory blocks
  - reserve some kernel memory for this

File Systems

- Files
- Directories
- Implementation
- Examples

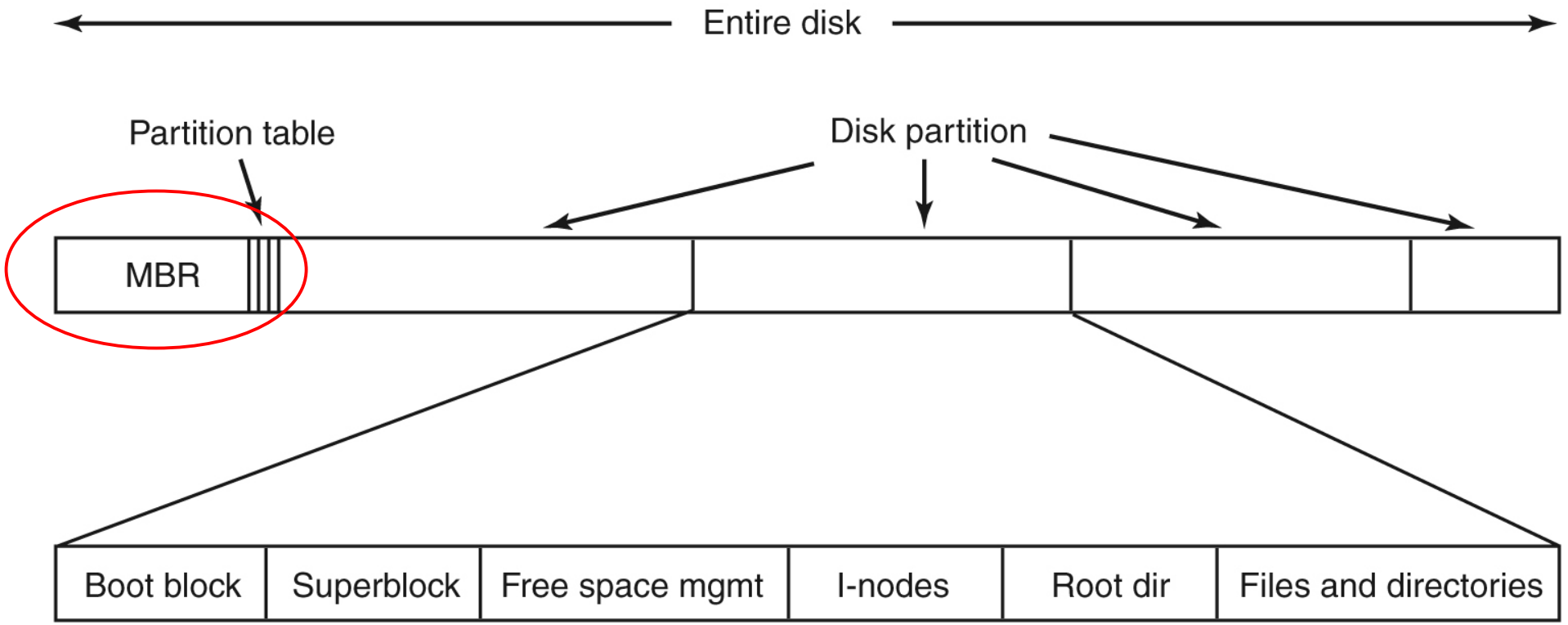
A Bit About Hard Drives





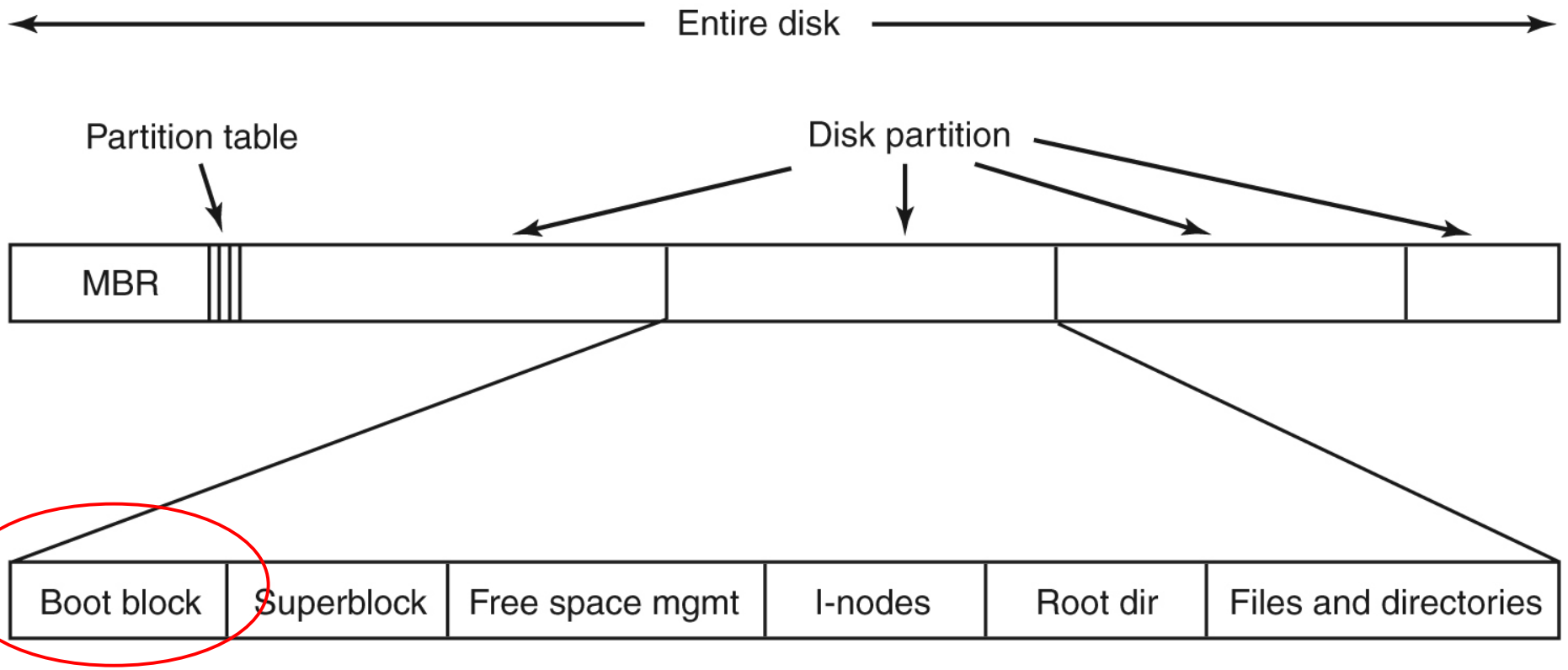
- Physical performance determined by seek time and rotational latency
- To be efficient, file systems must play to hard disk strengths and weaknesses

File System Layout



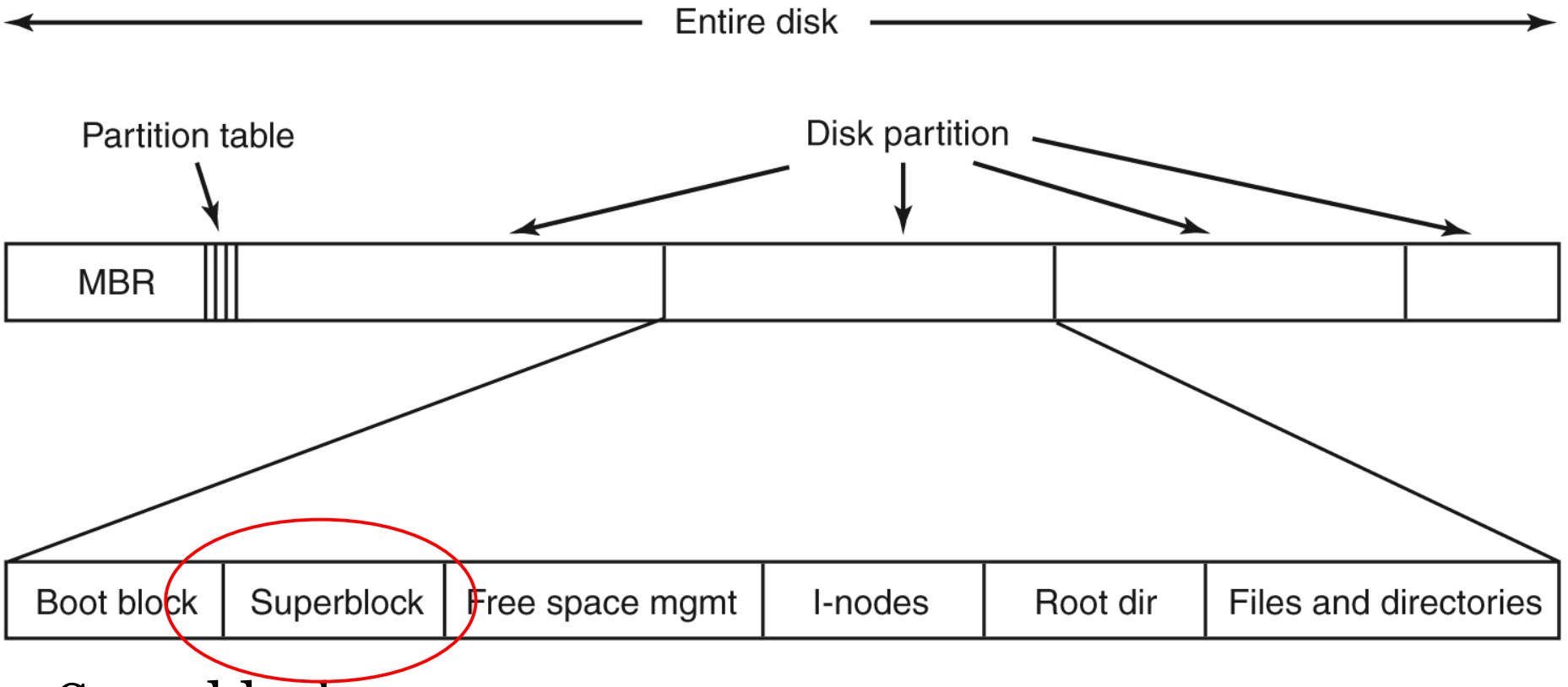
- MBR
  - Sector 0 of disk
  - Contains partition table (start, end, active)
  - Boot from active partition

File System Layout



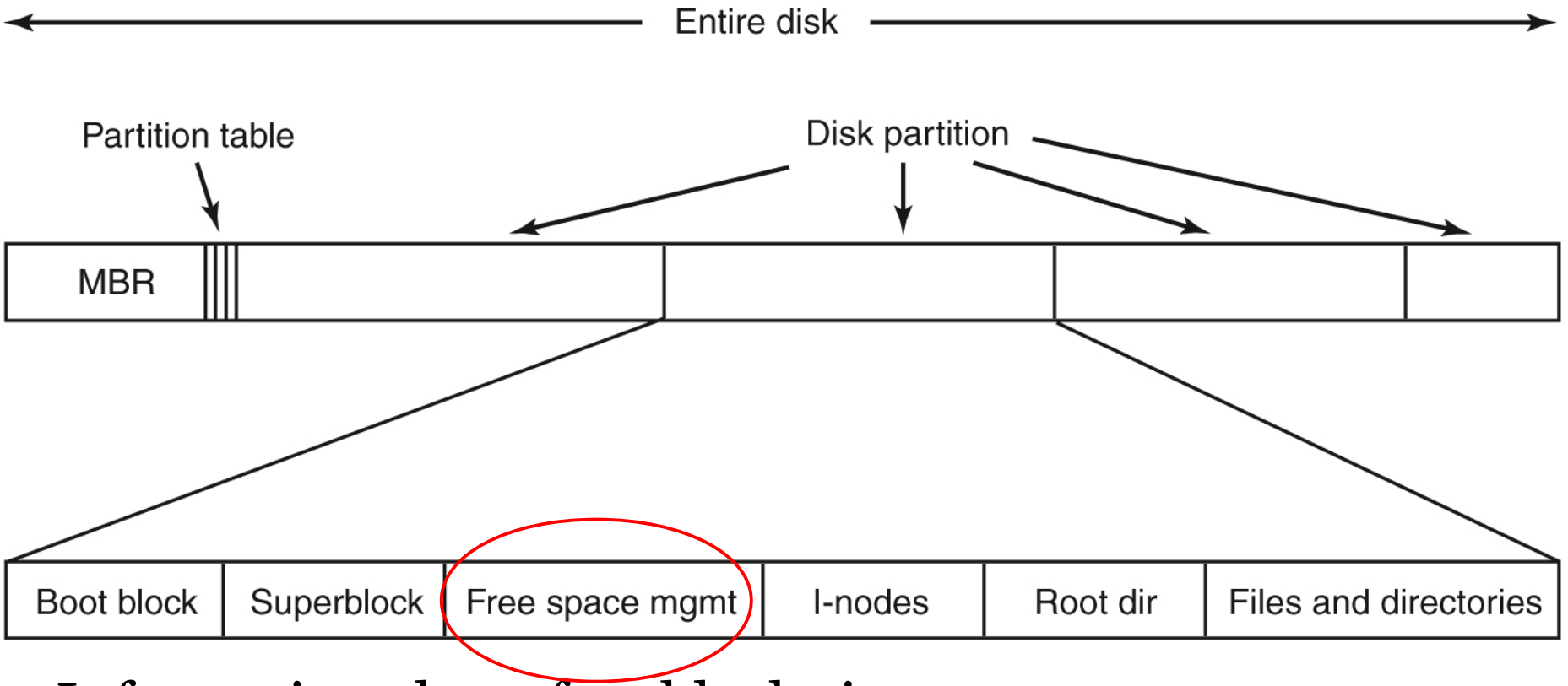
- Boot block
  - Loader for OS in that partition

File System Layout



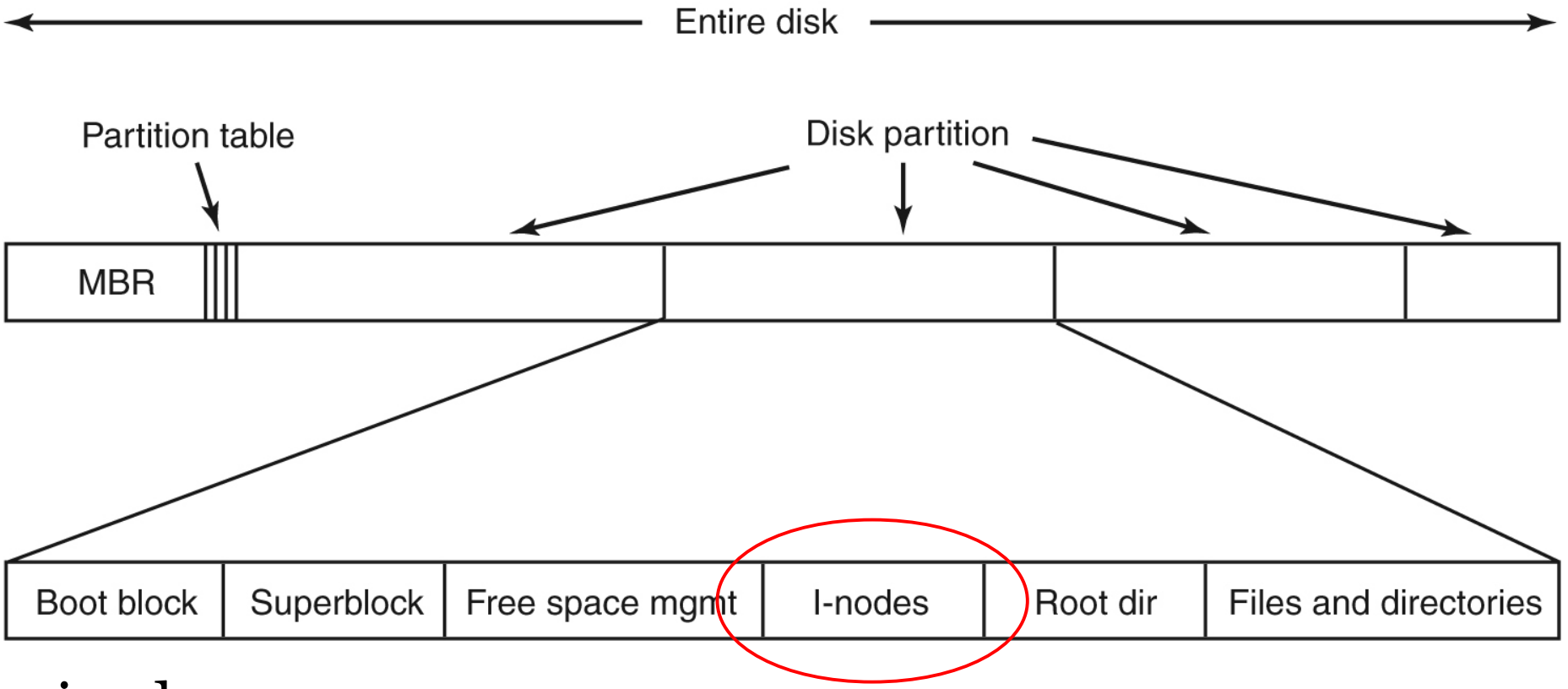
- Superblock
  - File system parameters (magic number, number of blocks in file system, etc)

File System Layout



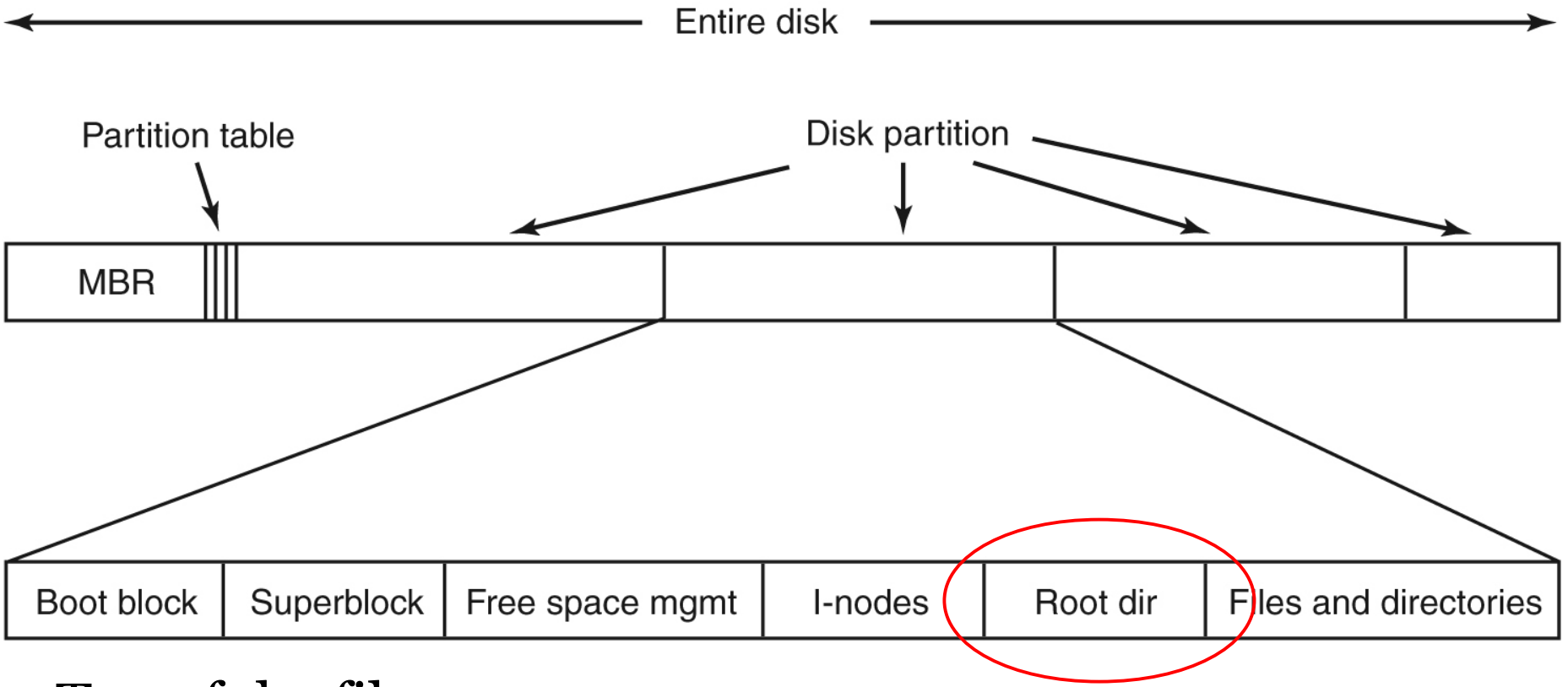
- Information about free blocks in system
  - Bitmap
  - List of pointers

File System Layout



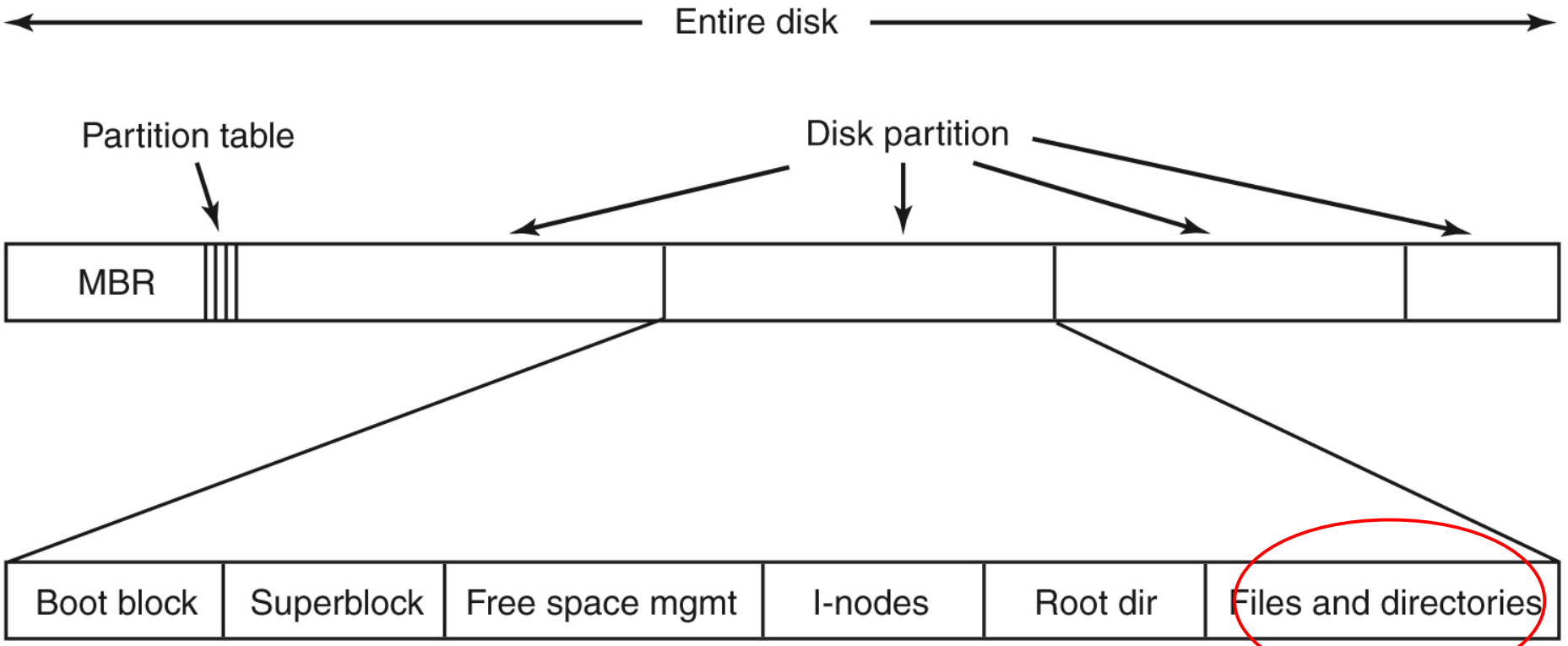
- inodes
  - Array of structures describing the file

File System Layout



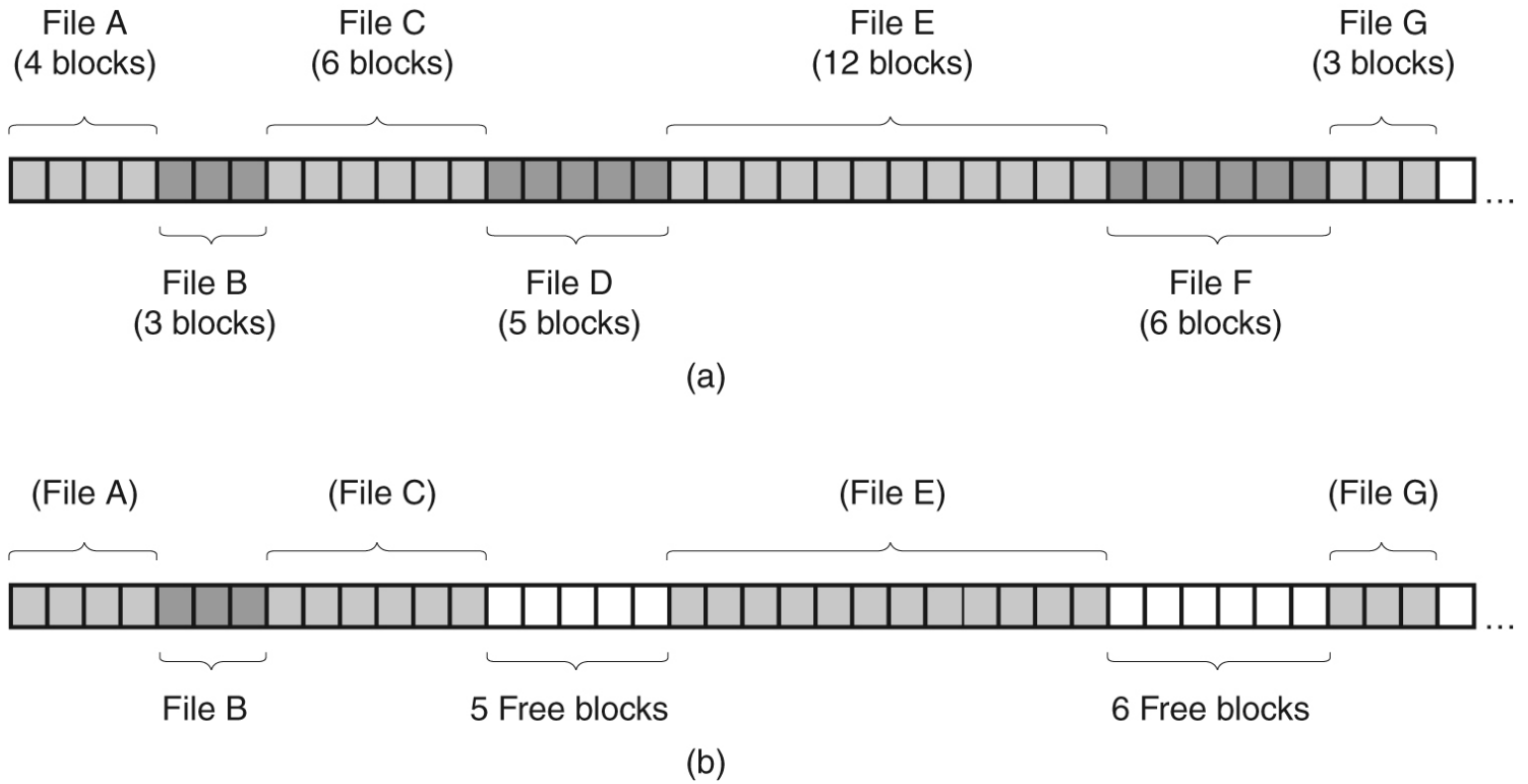
- Top of the file system

File System Layout



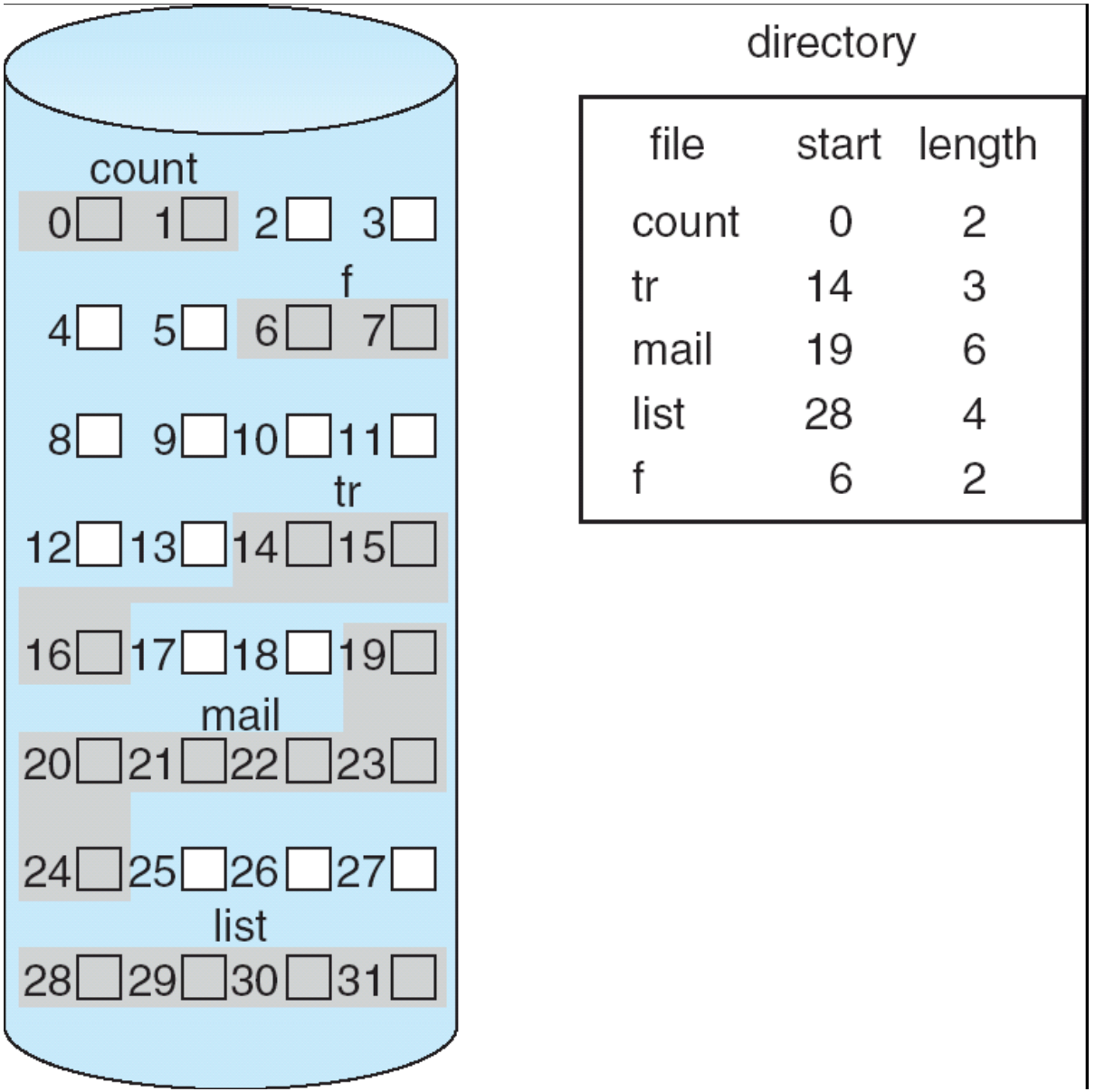
- All other directories and files

Files (Contiguous Allocation)



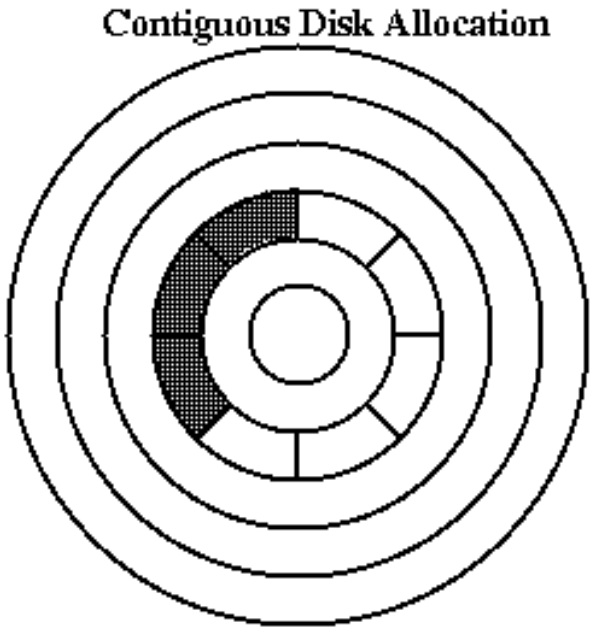
- Files begin at start of new block
- Simple: keep starting block and # of blocks
- Read performance excellent (few seeks)
- Fragmentation, have to know length of files

Contiguous Allocation of Disk Space

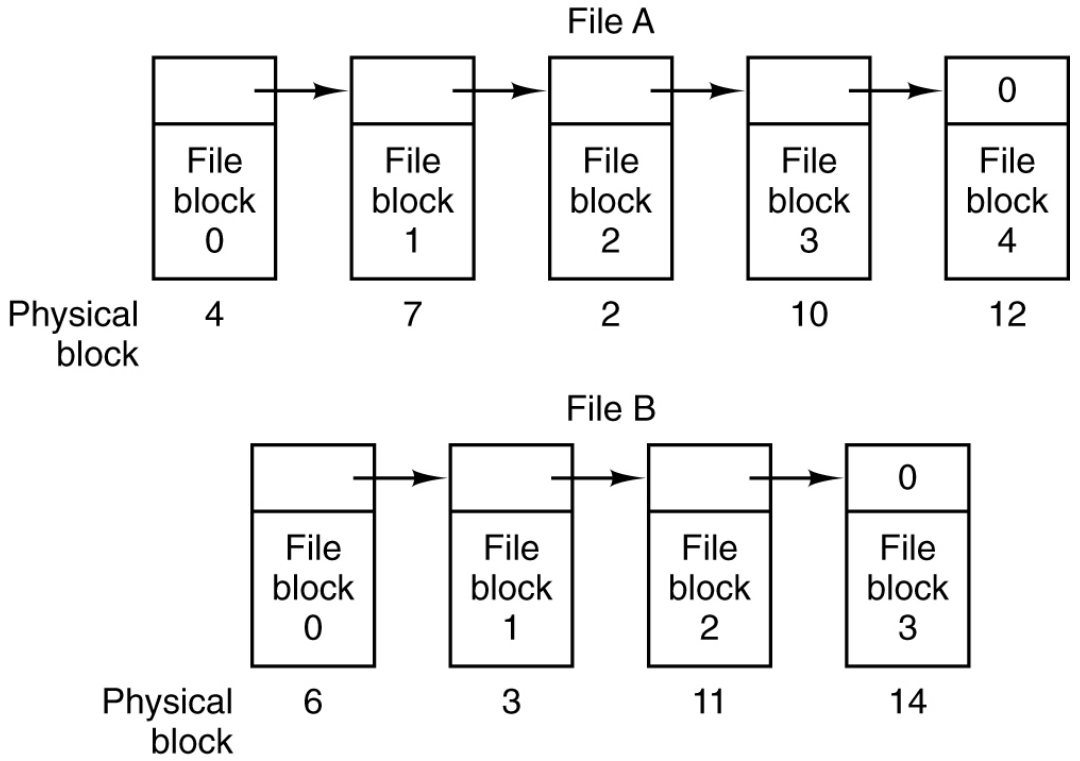


directory

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

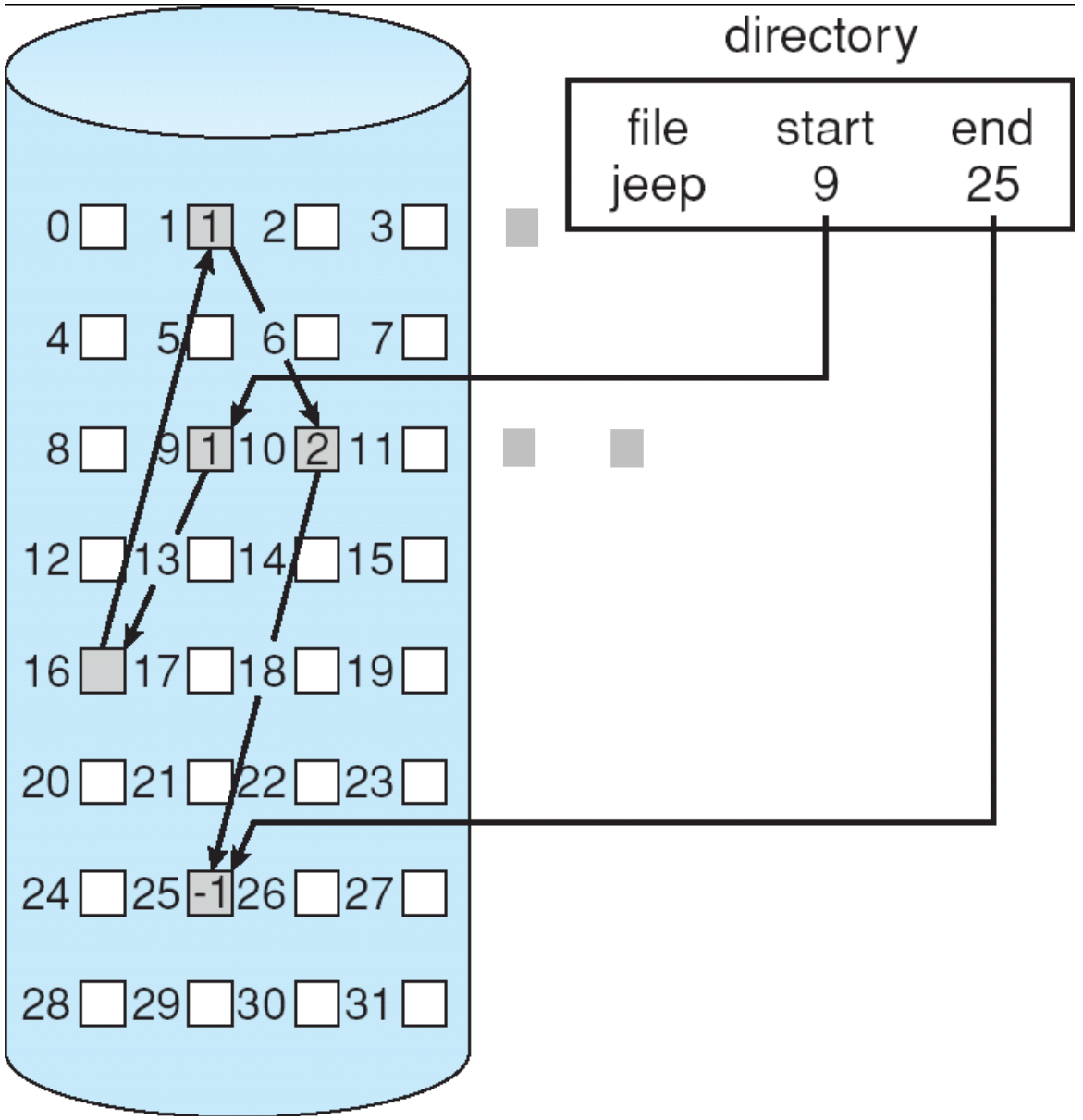


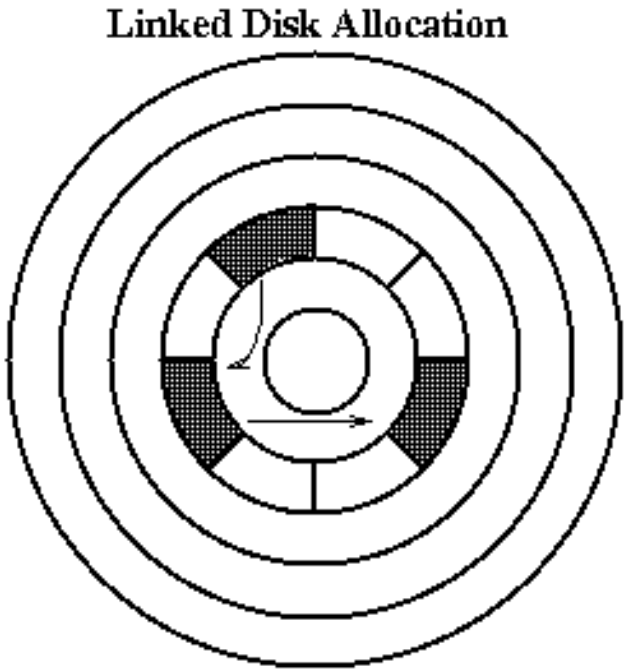
Files (Linked-List Allocation)



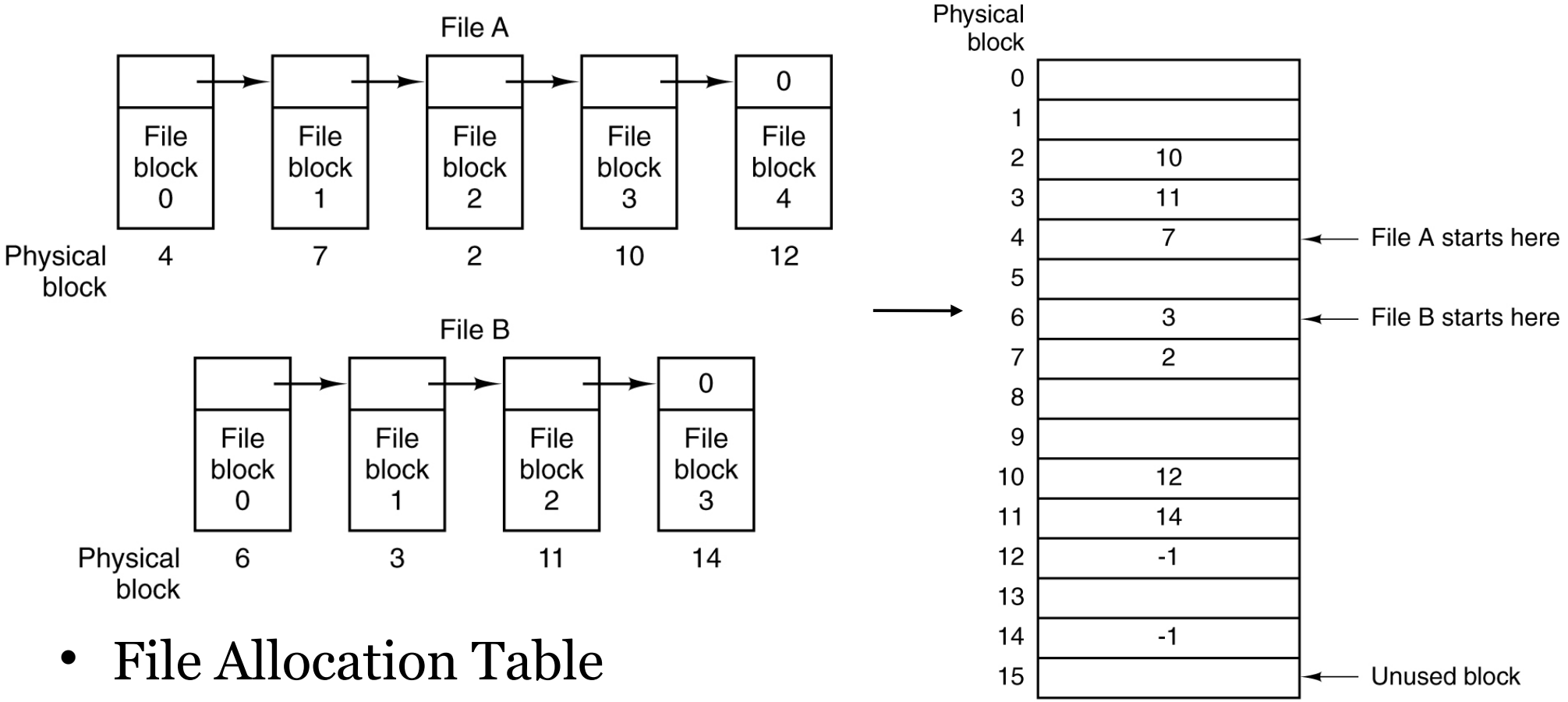
- First word in block points to next block
  - No external fragmentation
  - Random access is slow, lots of seeks
  - Reading block size of data requires 2 blocks

Linked Allocation of Disk Space



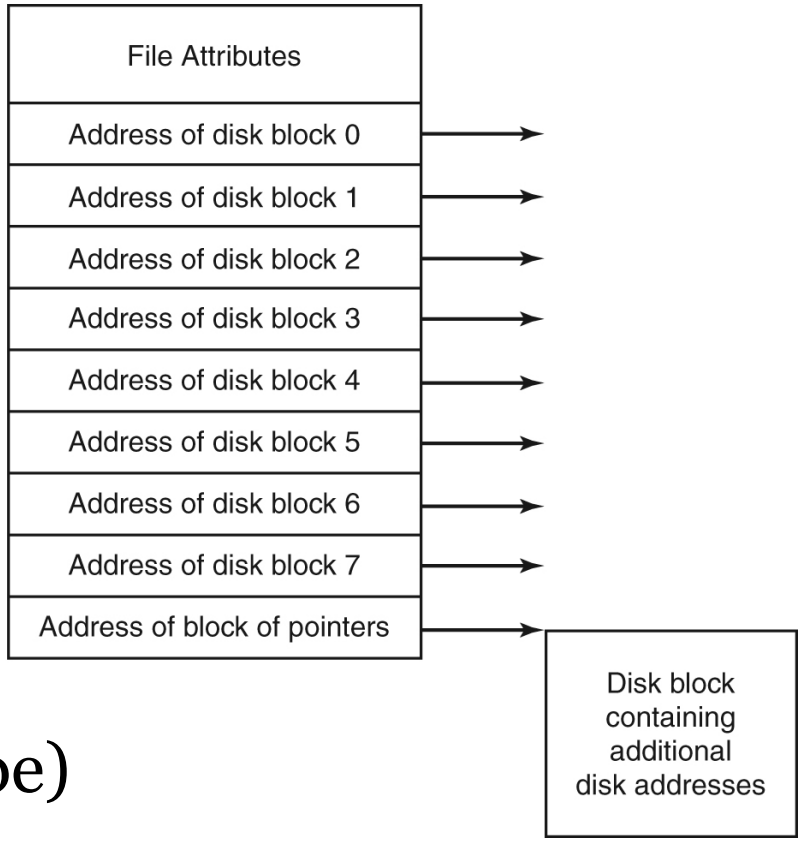


Files (FAT)



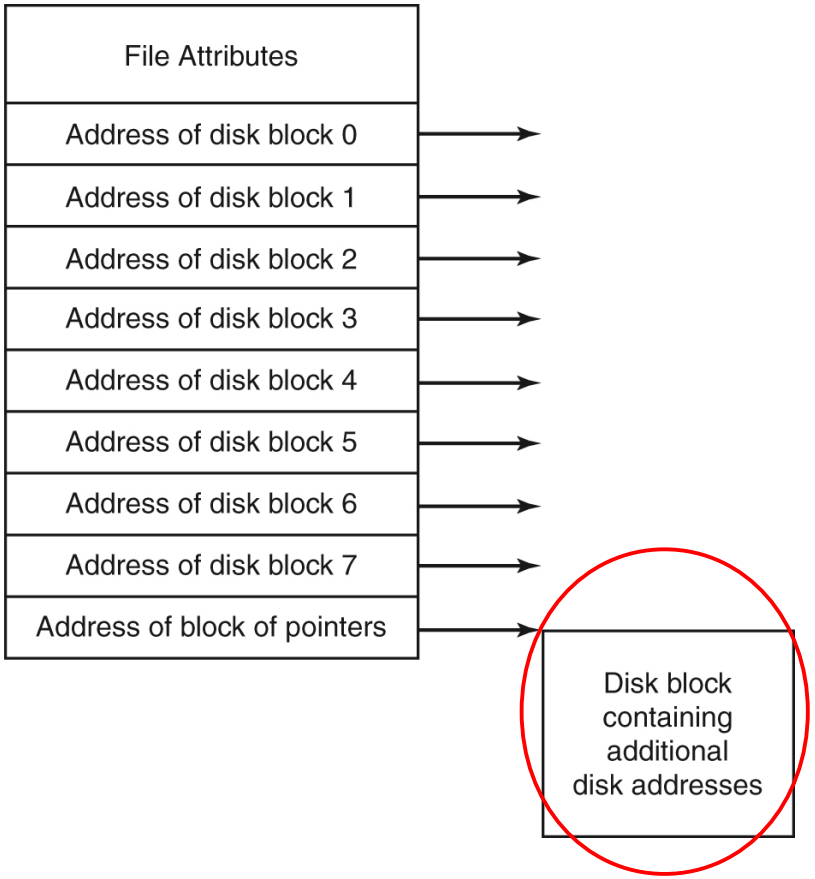
- File Allocation Table
  - Solves problem of wasting block space for pointer
  - Can still locate all blocks from starting block
  - Entire table in memory (e.g. 800MB for 200GB disk)

Files (inodes)



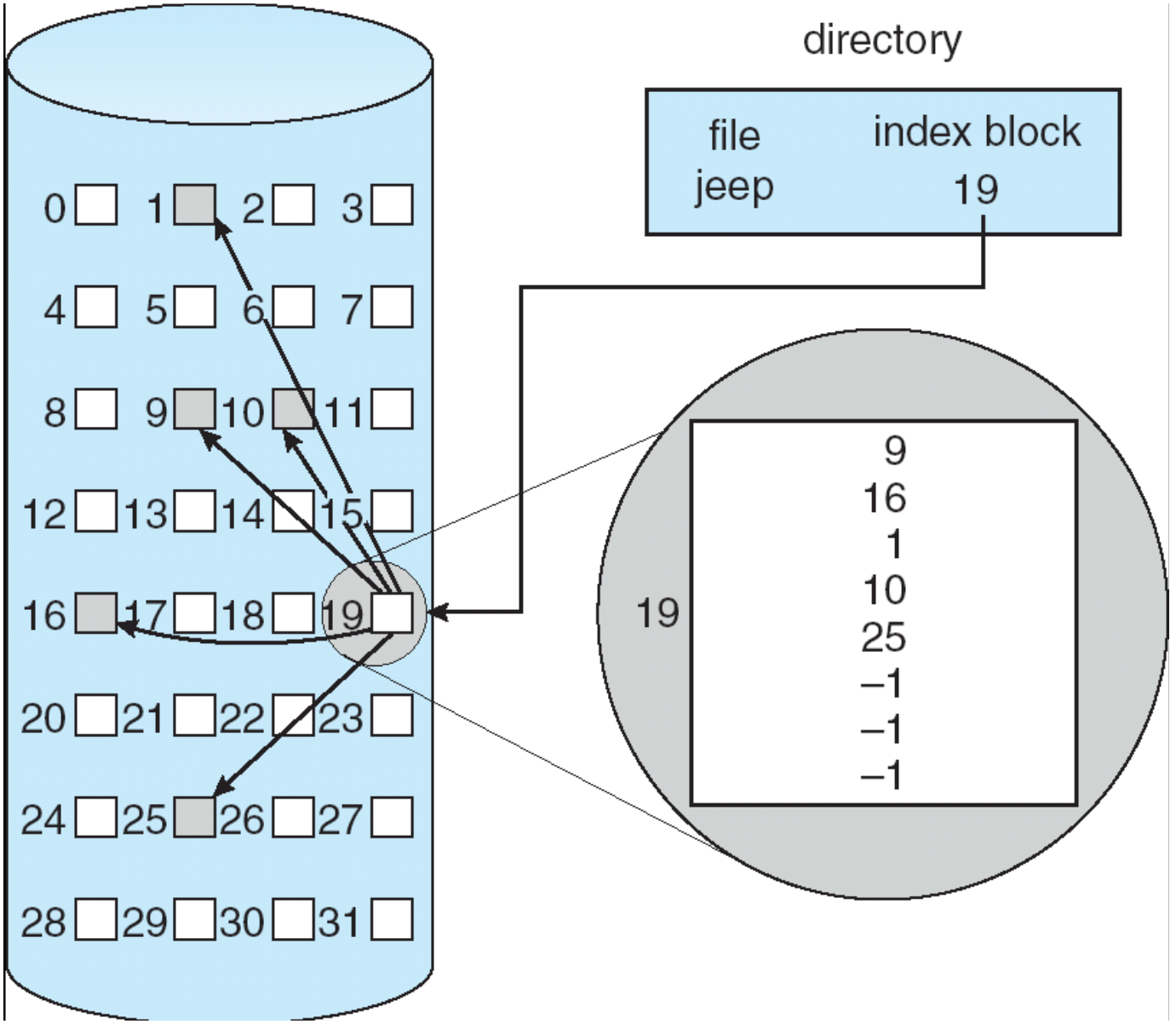
- Index-node? (maybe)
  - Lists attributes and disk addresses of all blocks
  - File's inode only in memory when file is used

Files (inodes)

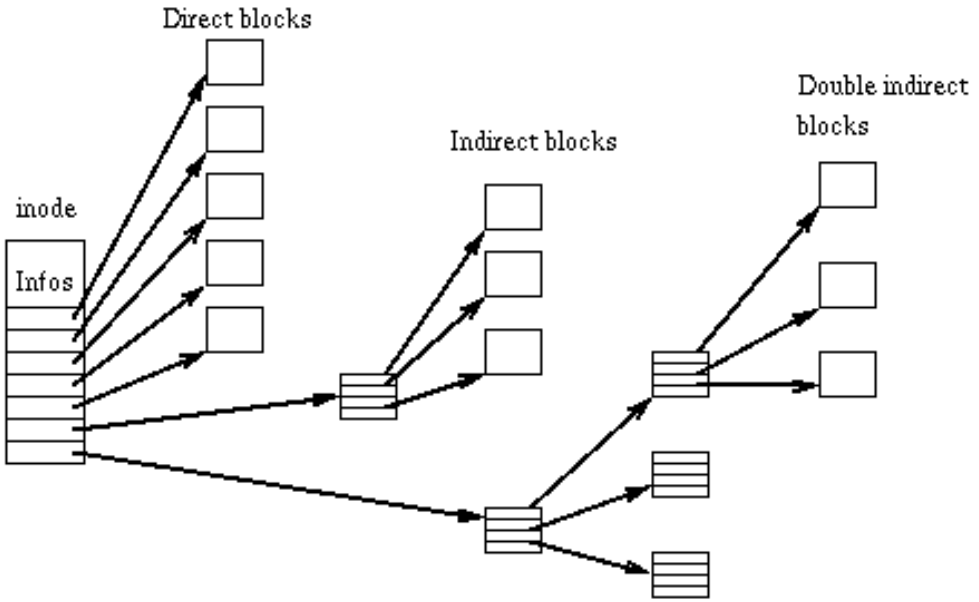


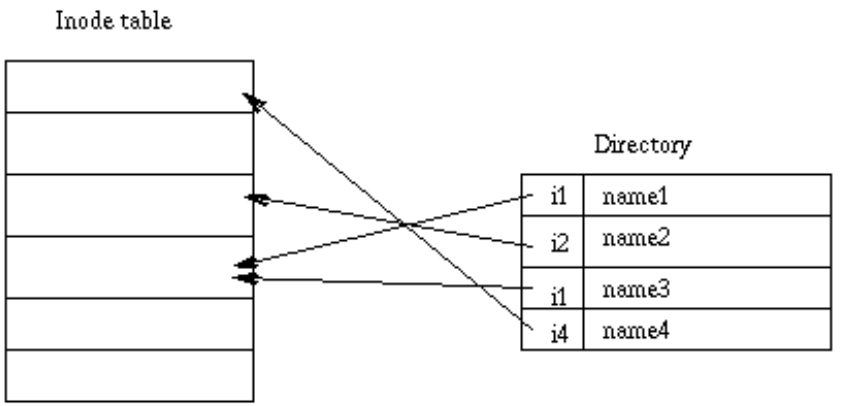
- Why this?

Example of Indexed Allocation



Summary so far...





FS Management and Optimization

- Files usually stored as collections of fixed-size blocks
  - Need not be stored adjacently on disk
- Ok, but what's a good block size?
  - Depends…

Choosing a Block Size

- Whatever is best fit

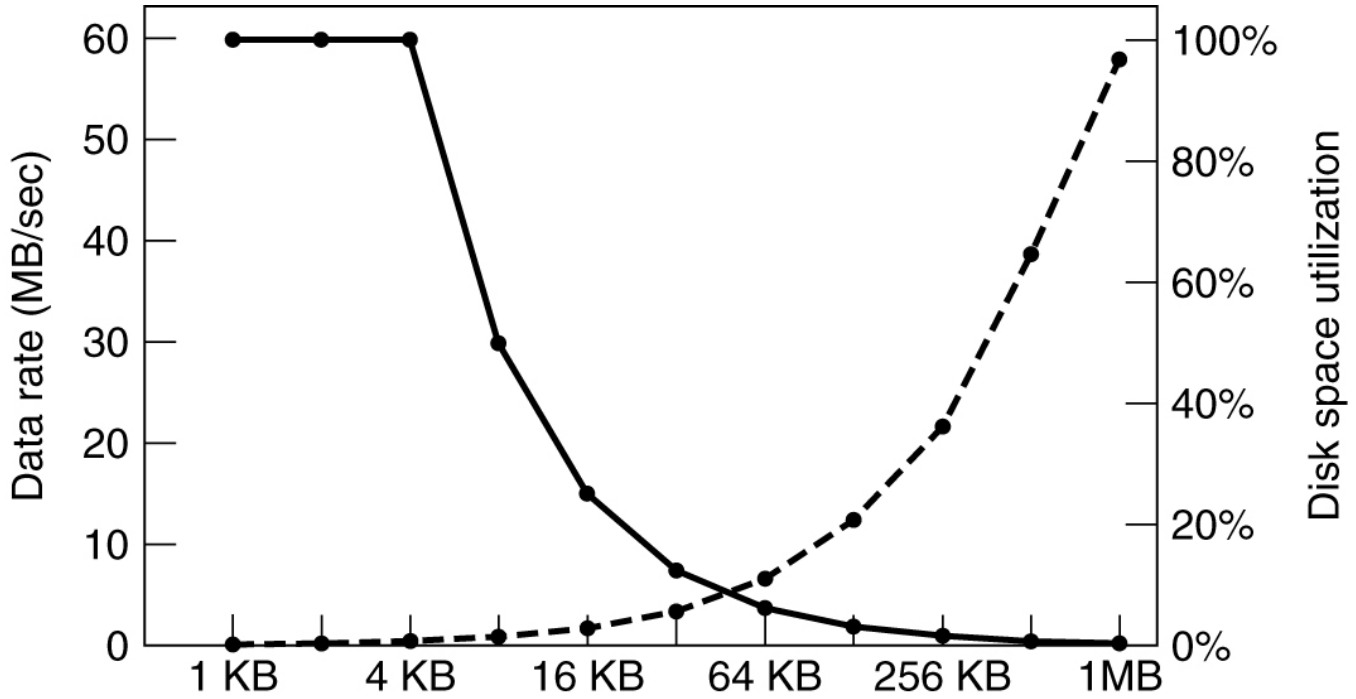
Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

Choosing a Block Size

- Keep in mind:
  - Smaller blocks ^ more seeks
  - Bigger blocks ^ more wasted space
  - Time vs space tradeoff

The Tradeoff



- Data rate increases with block size
  - Access time for a block dominated by seek and rotational latency
- Space wastage increases with block size

I've Chosen a Block Size, Now What

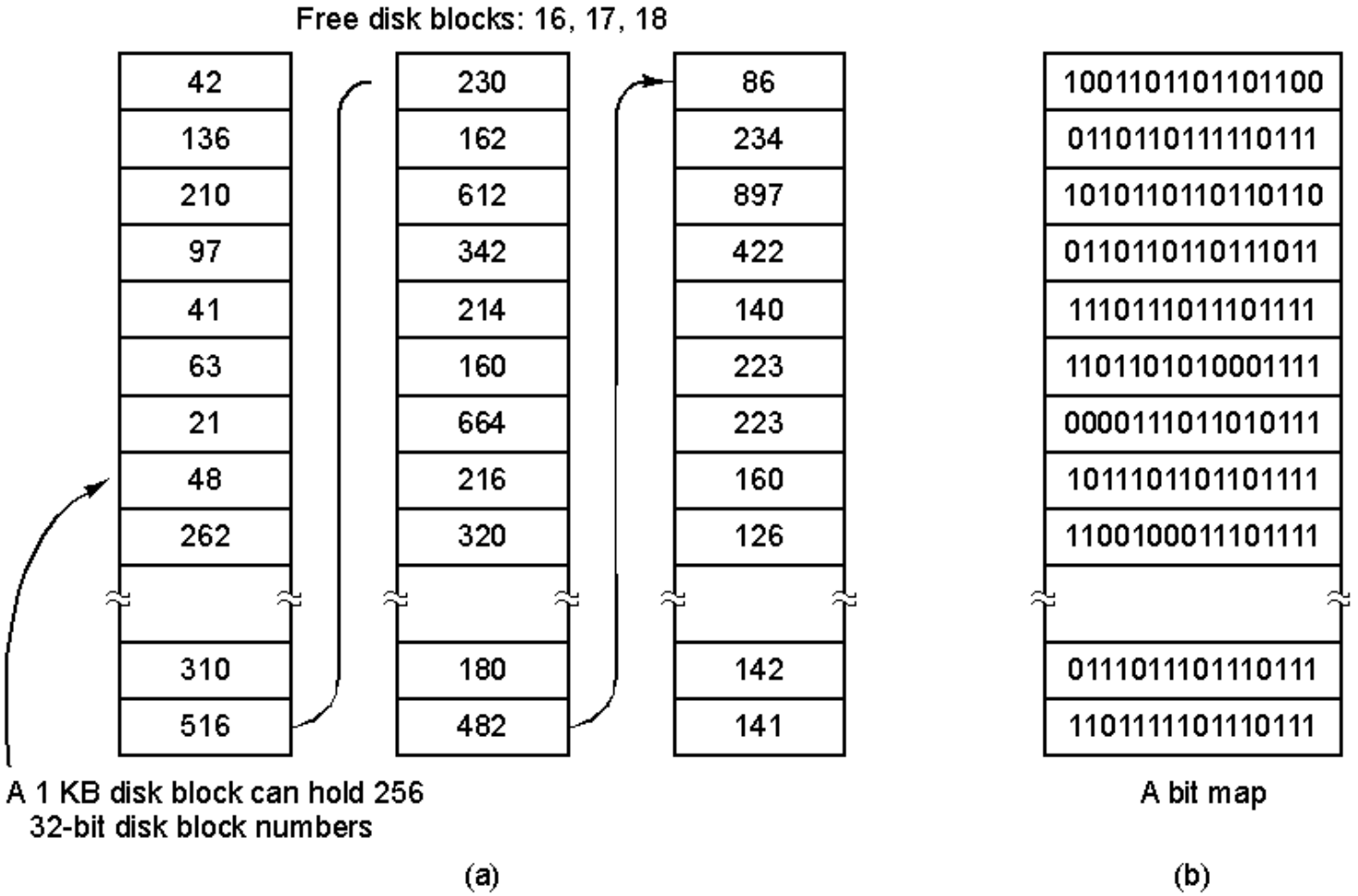
- Need to keep track of free blocks

Free disk space management

- Linked list
  - A linked-list list of unnused blocks is used to keep track of free space
  - 32 bits per list entry
  - Many entries per list in empty blocks (linked)

- Bitmap
  - A bitmap of all blocks keeps track of which blocks are in use and which are free
  - Better than linked list if the entire bitmap can be kept in main memory at all times

List vs. Bit Map



(a) Storing the free list on a linked list
(b) A bit map

Performance

- How do users access files?
  - Sequential: bytes read/written in order
  - Random: read/write blocks in middle of file
  - May access whole or partial file
- How are files used?
  - Most are small
  - Large files take up most of disk space
  - Large files account for most bytes transferred
- Everything needs to be efficient

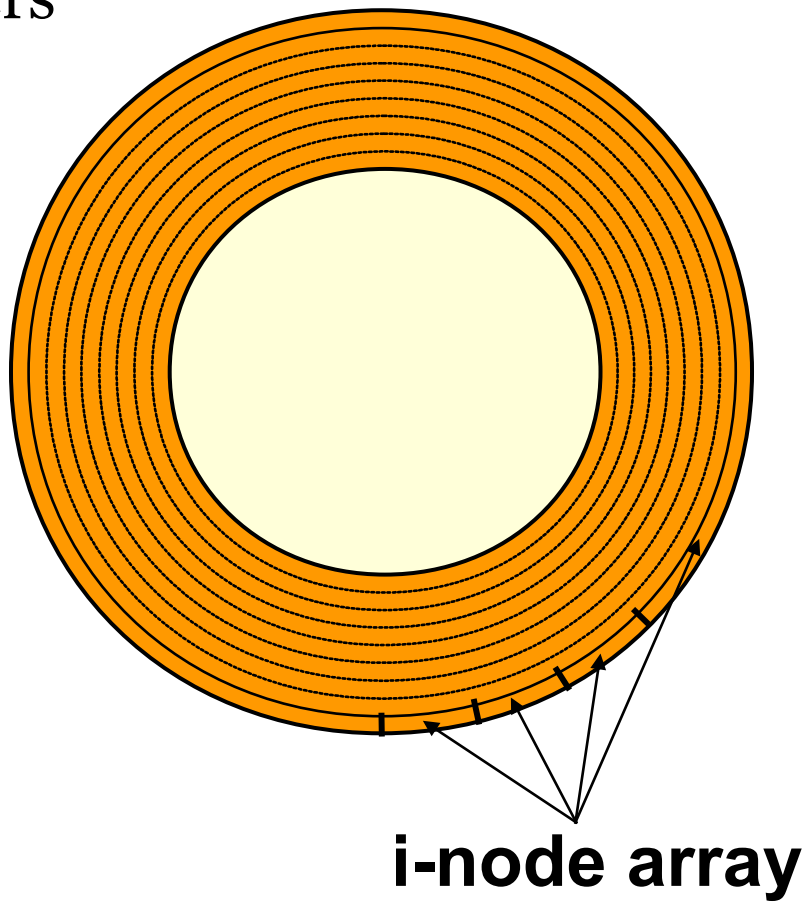
Performance

- Hard drive
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Data/metadata layout

- Operating system
  - Disk cache – separate section of main memory for frequently used blocks
  - Delayed-writes – aggregation / higher priority to reads
  - Read-ahead – Prefetching of sequential blocks
  - Section of memory as virtual disk (RAM disk)

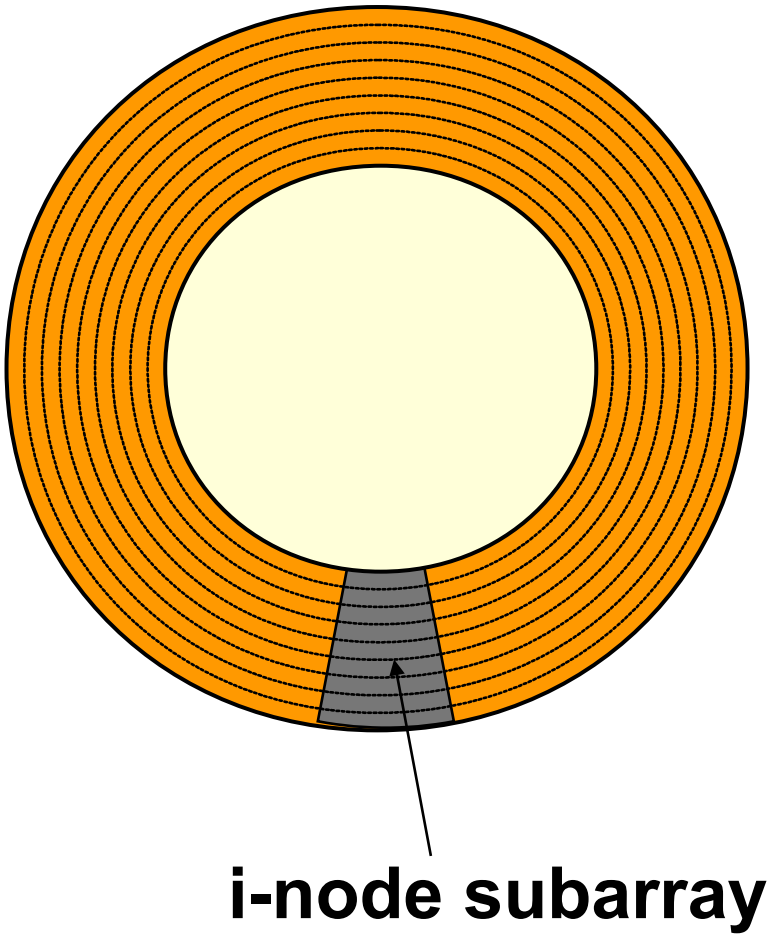
Early Unix Disk Layout

- Array of inodes in outermost cylinders
- inodes are far away from data
- at least 2 seeks per I/O



Modern Disk Layout (very broadly)

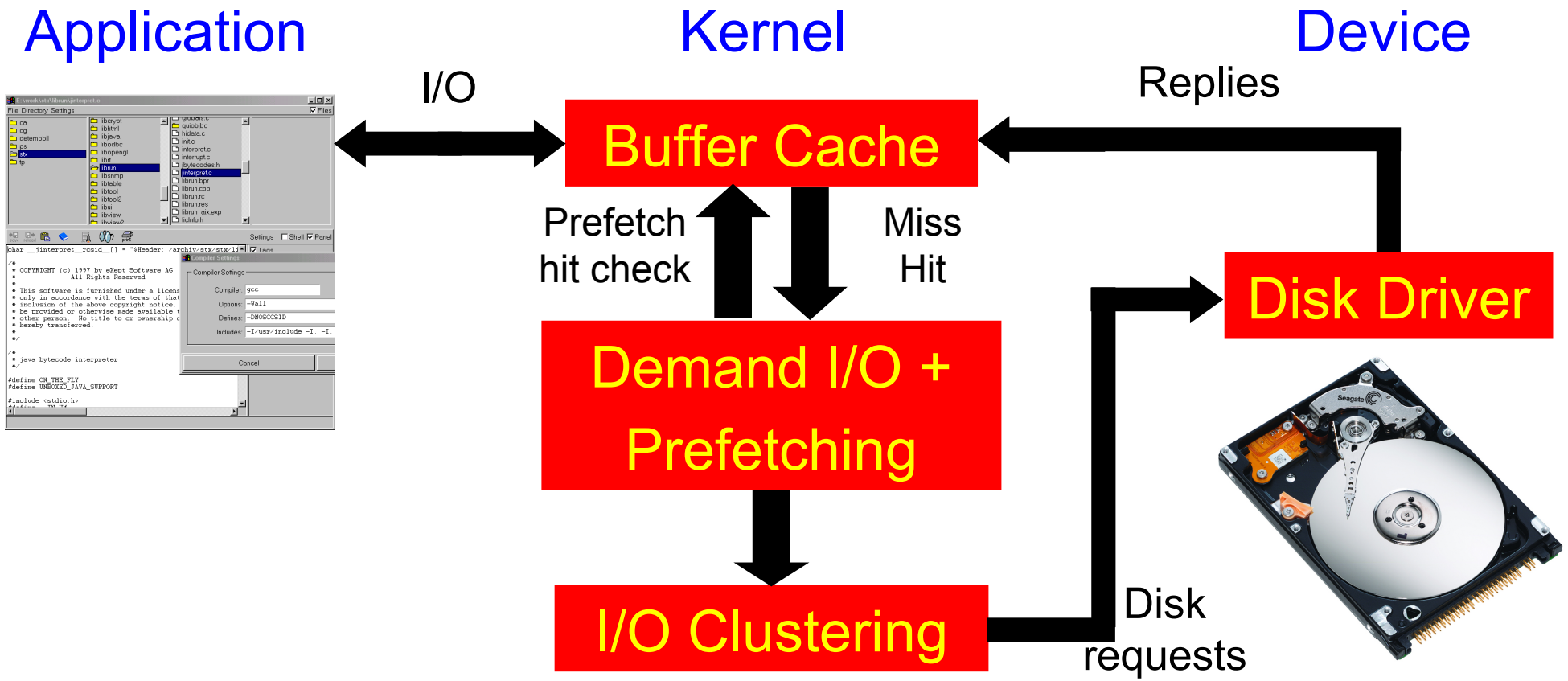
- Portion of inode array on each cylinder
- Multiple inodes on same block
- One seek to access inode AND data



File Caching

- Locality of reference in file accesses
  - Yet another application of the principle of locality
  - What were the earlier instances?
- Keep a number of disk blocks in the much faster memory
  - when accessing disk, check the cache first!

I/O path in modern OS

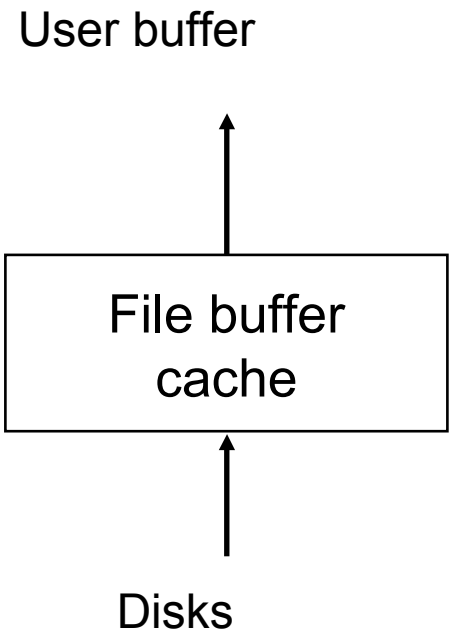


- Variable portion of main memory (in Linux); processes and file system compete for physical memory

Handling Reads

- read(fd, buf, n)

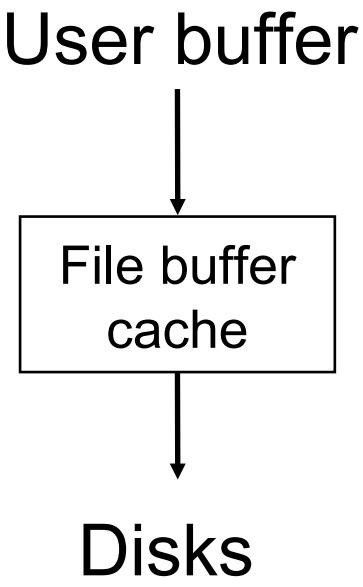
- On a hit
  - copy from the buffer cache to a user buffer
- On a miss
  - replacement when full
  - read a file into the buffer cache



Handling Writes

- write(fd, buffer, n)

- On a hit
  - write to buffer cache
- On a miss
  - read the file into buffer cache (possible replacement)
  - write to buffer cache



File prefetching

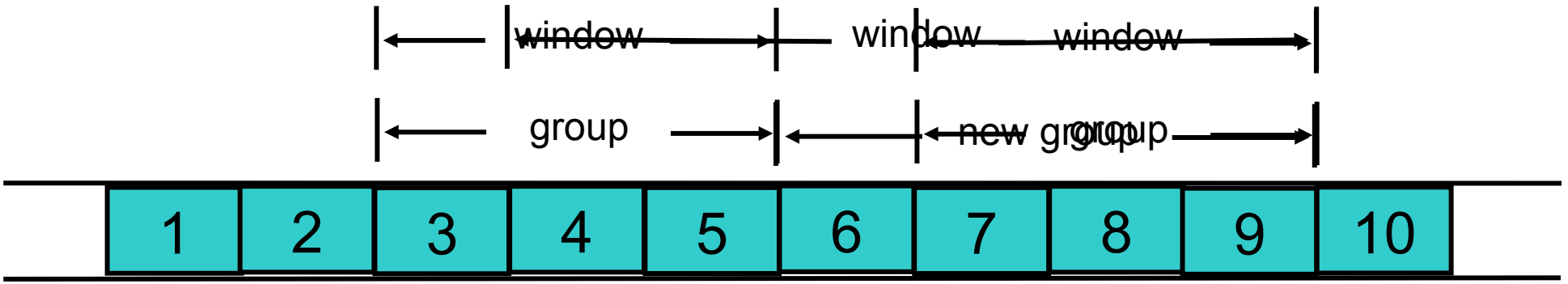
- Read-ahead:
  - For sequential access, read the requested block and the following N blocks together (why is this a good idea?)

- Drawbacks:
- Potential for cache pollution
  - Data prefetched that will not be accessed
  - Useful data replaced

- Wrong prefetches can saturate disk I/O bandwidth

Prefetching in Linux Kernel

- Targets consecutive blocks
- Read-ahead group
  - Consecutive blocks fetched (demand + prefetched)
- Read-ahead window
  - Consecutive blocks in the previous and current read-ahead groups



Observations

- Kernel prefetching can:
  - Narrow the performance gap of replacement algorithms
  - Change the relative performance benefits of algorithms

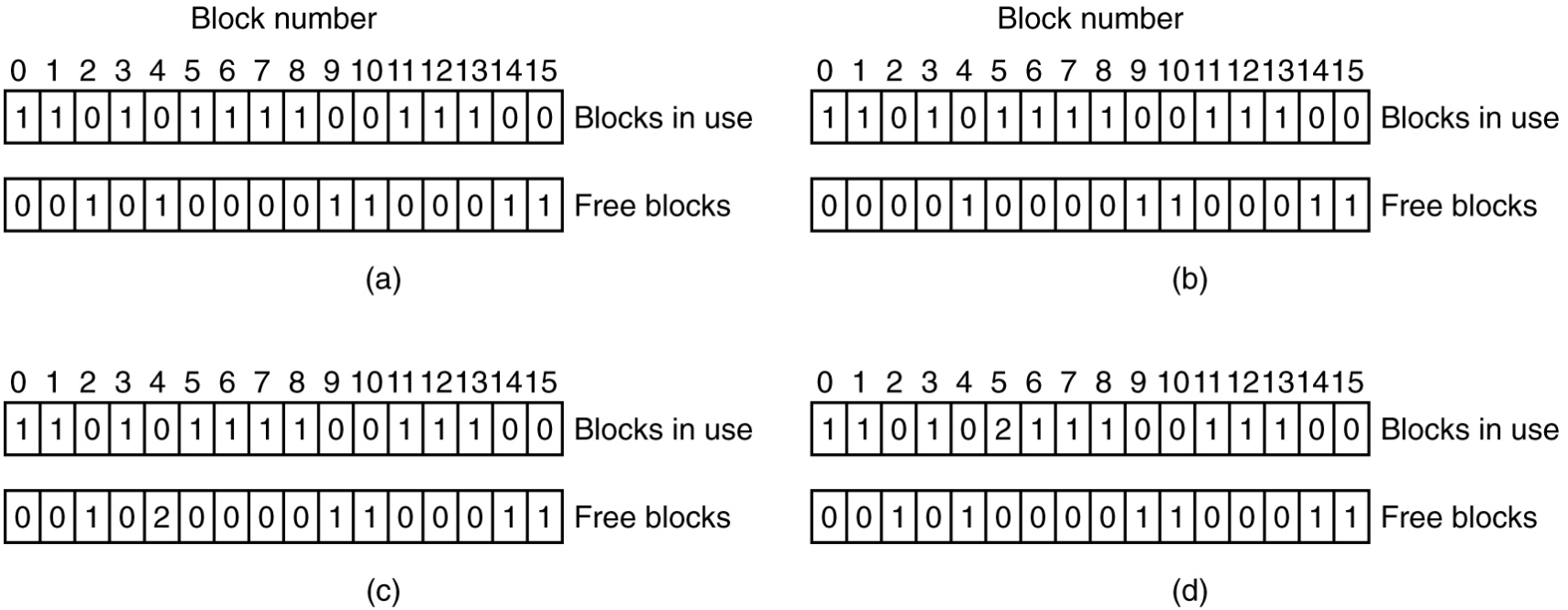
- Implication: Execution time is the ultimate metric

File System Consistency

- System crashes
  - Are all blocks written out?
  - Can lead to inconsistent state
- Utilities exist that check for consistency
  - UNIX: fsck
  - Win: scandisk, chkdsk
- Block consistency
  - Count blocks used
  - Count blocks in free list
  - Consistency means a block is either used or it is not

Consistency Checks

- System crashes can leave the filesystem in a number of inconsistent states



Consistency Checks

- Also need to check directories for inode references
  - Walk the directory tree, keep counts of references
  - Compare reference counts to those stored in inodes

Some File Systems

- Media based
  - Ext3/4 - Linux native
  - ufs – BSD
  - fat - DOS FS
  - vfat - win 95
  - hpfs – OS/2
  - minix – early Linux
  - Isofs – CDROM
  - sysv - Sysv Unix
  - hfs – Macintosh
  - affs - Amiga Fast FS
  - NTFS - NT's FS
  - adfs - Acorn-strongarm
- Network
  - nfs
  - Coda
  - AFS - Andrew FS
  - smbfs – LanManager
  - ncpfs - Novell
- Special ones
  - procfs -/proc
  - umsdos - Unix in DOS
  - userfs - redirector to user

File Systems

Files

Directories

Implementation

Examples

- Virtual File System

Linux File Systems

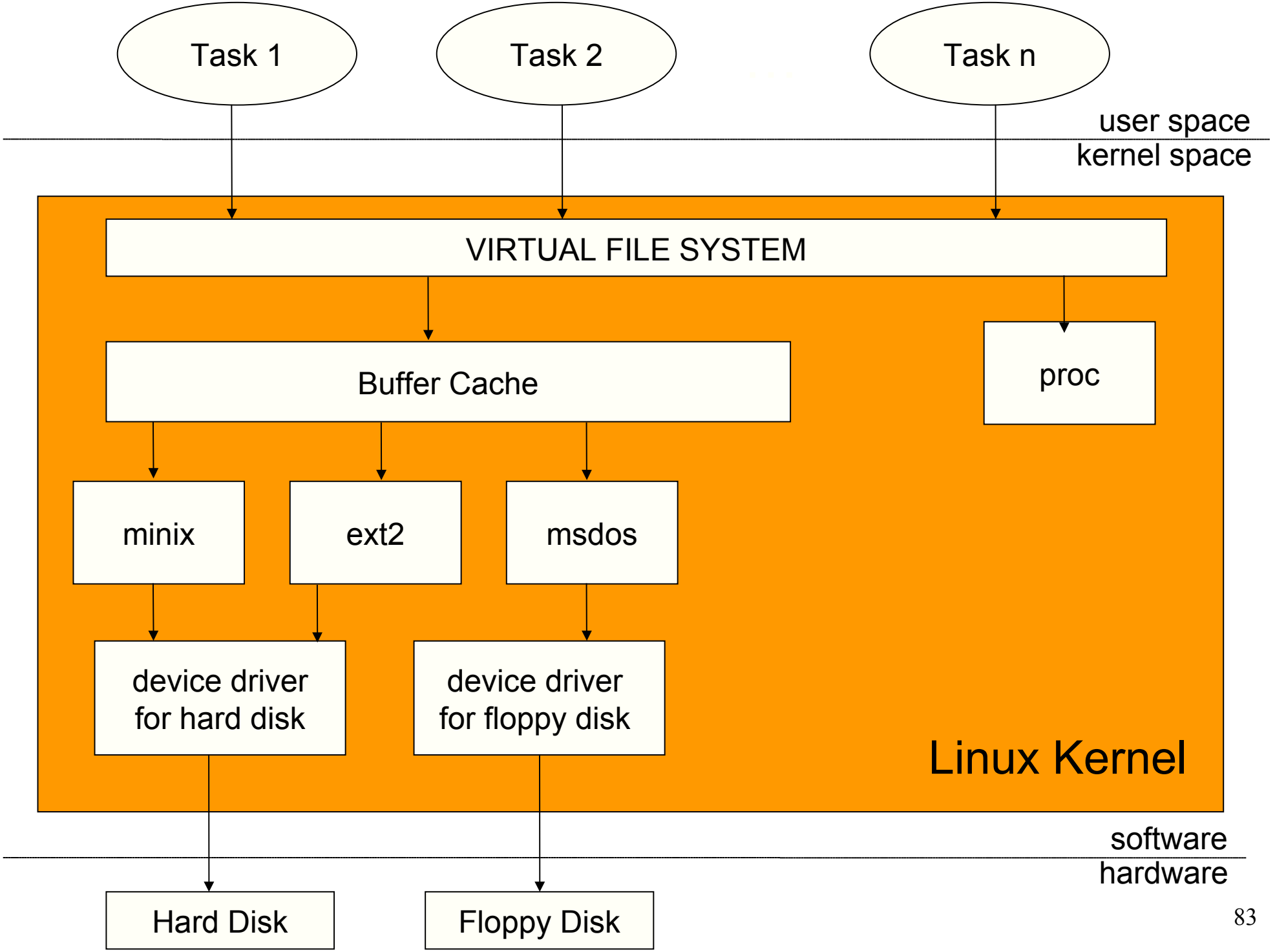
- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the *virtual file system (VFS)*

Linux File Systems

- The Linux VFS is designed around object-oriented principles and is composed of two components:
  - A set of definitions that define what a file object is allowed to look like
    - The *inode-object* and the *file-object* structures represent individual files
    - the *file system object* represents an entire file system
  - A layer of software to manipulate those objects

Virtual File Systems

- Goal
  - Allow one machine to use multiple file system types
    - Unix FFS
    - MS-DOS FAT
    - CD-ROM ISO9660
    - Remote/distributed: NFS/AFS
  - Standard system calls should work transparently
  - Share common concepts and high-level operations
- Solution
  - Insert a level of indirection (VFS)!
  - Applications interact with this VFS
  - Kernel translates abstract-to-actual


Task 1
Task 2
Task n
user space
kernel space
VIRTUAL FILE SYSTEM
Buffer Cache
proc
minix
ext2
msdos
device driver
for hard disk
device driver
for floppy disk
Linux Kernel
software
hardware
Hard Disk
Floppy Disk

File Systems

Files

Directories

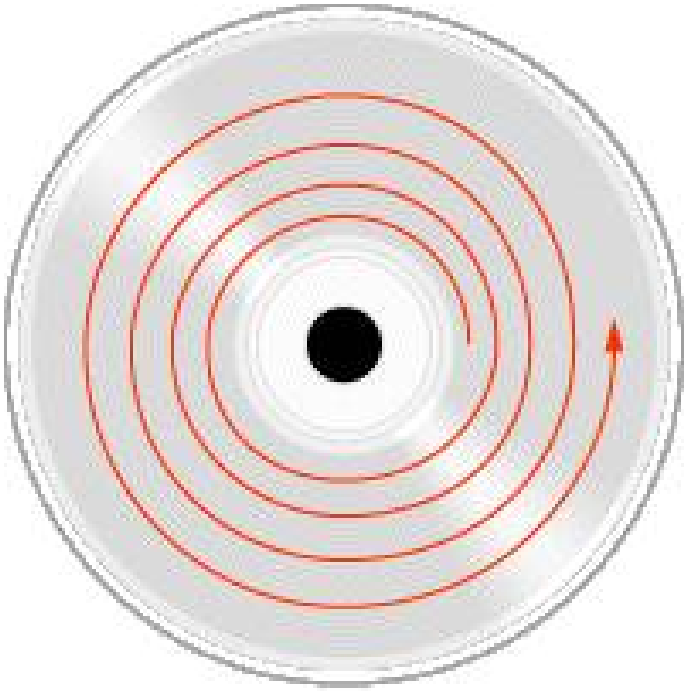
Implementation

Examples

- Virtual File System
- CD-Rom FS (ISO 9660)

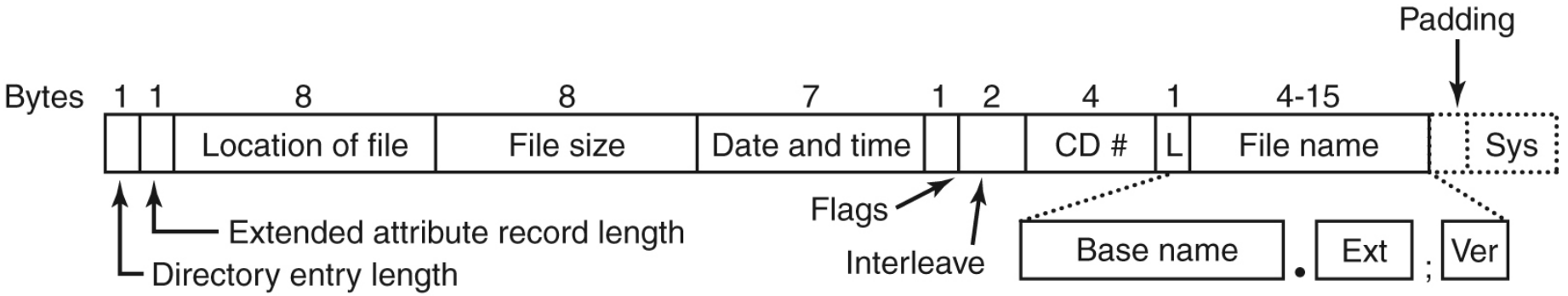
CD-ROM

- Write-once, append-only
- No tracking of free blocks
- Free space at end of CD
- Spiral contains bits in linear sequence
- 2352 byte blocks
- 2048 byte payload



CD-ROM

- 16 blocks undefined
- (block 17) Primary volume descriptor
  - Ids: system, volume, publisher, data preparer, etc
  - Root directory entry (where is it on disc?)
- Can locate rest of file system from root dir
- Directory entries:



File Systems

Files

Directories

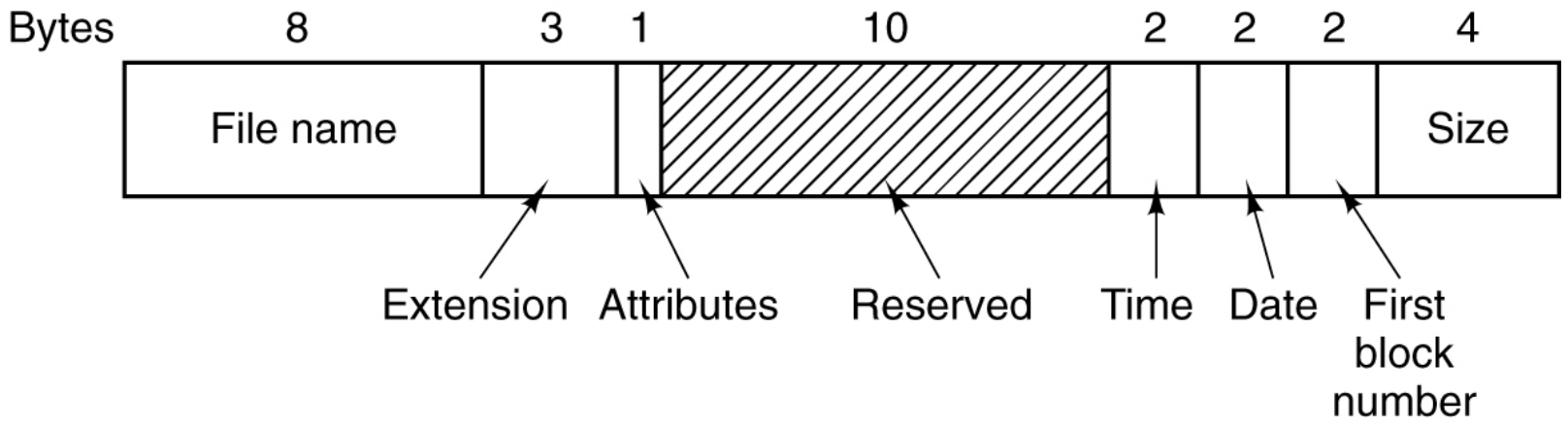
Implementation

Examples

- Virtual File System
- CD-Rom FS (ISO 9660)
- MS-DOS

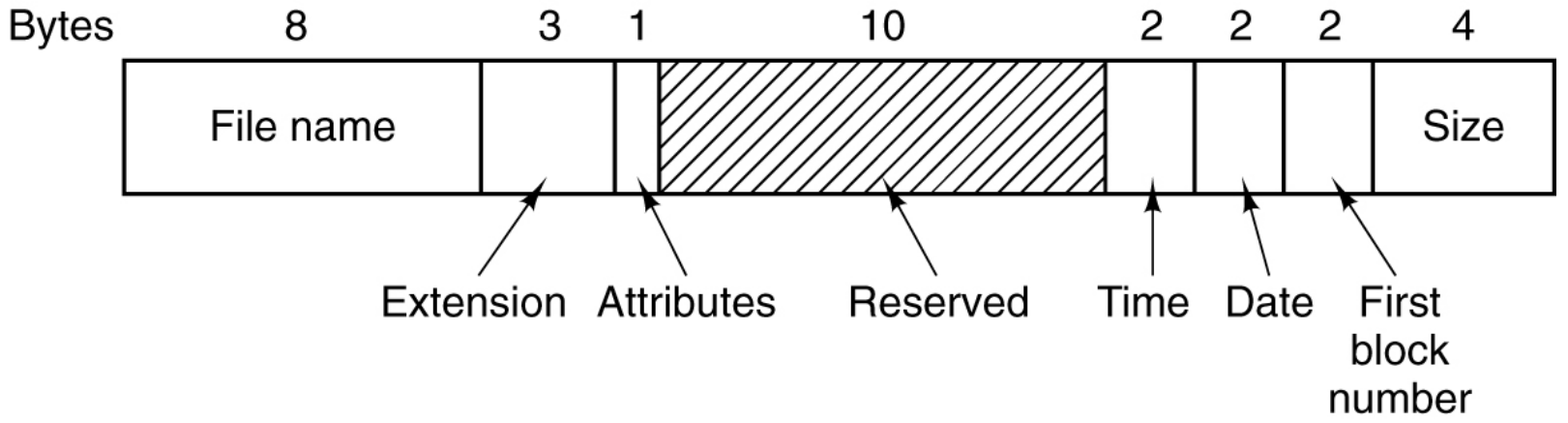
MS-DOS FS

- Used in: Windows 98-ME, floppy disks, embedded systems (FAT32), iPod
- Supported by: Windows 2000, XP, Vista, Linux
- Fixed-size dir entries:



MS-DOS FS

- Time field limited to 2 bytes
  - 65,536 unique values, but day has 86,400 seconds
- Year maxes out at 2107
  - Y2108
- File size ^ 32 bits (theoretical max 4GB)



MS-DOS FS

- Flavors: FAT-12, FAT-16, FAT-32
  - Number refers to number of bits used in disk address
  - Not all bits used for address: e.g. FAT-32 might be better called FAT-28
- Disk blocks: 512 bytes
- So…

  FAT-12 w/ 512 byte blocks

  Maximum partition size?

  2^12*512 = 2MB

MS-DOS FS

- Maximum partition sizes w/ different block sizes

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

- So, if I have a 8GB disk and FAT-32, at most how many partitions do I need?
- How about FAT-16?

File Systems

Files

Directories

Implementation

Examples

- Virtual File System
- CD-Rom FS (ISO 9660)
- MS-DOS
- Ext2/Ext3

Design Goals

- No performance loss
  - should have a performance gain!
- Backward compatibility
- Reliability through
  - preservation: stable data not affected by crashes
  - predictability: known failure modes
  - atomicity: each operation fully completes or is undone after recovery

JFS

- Atomic updates
- Old and new versions of data held on disk until commit
- Undo logging
  - copy old data to log
  - write new data to disk
  - crash during update? copy old data from log
- Redo logging
  - write new data to log
  - old data remains on disk
  - crash during update? copy new data from log

JFS (2)

- Speedy recovery after a crash
  - fsck is slow on large disks
  - JFS rereads journal after a crash, from last checkpoint
- Journal:
  - contains three types of blocks
    - metadata: contents of single fs block metadata as updated by transaction
    - descriptor: describe other journal metadata blocks
    - header: head and tail of journal, sequence number
  - circular structure

JFS vs LSFS

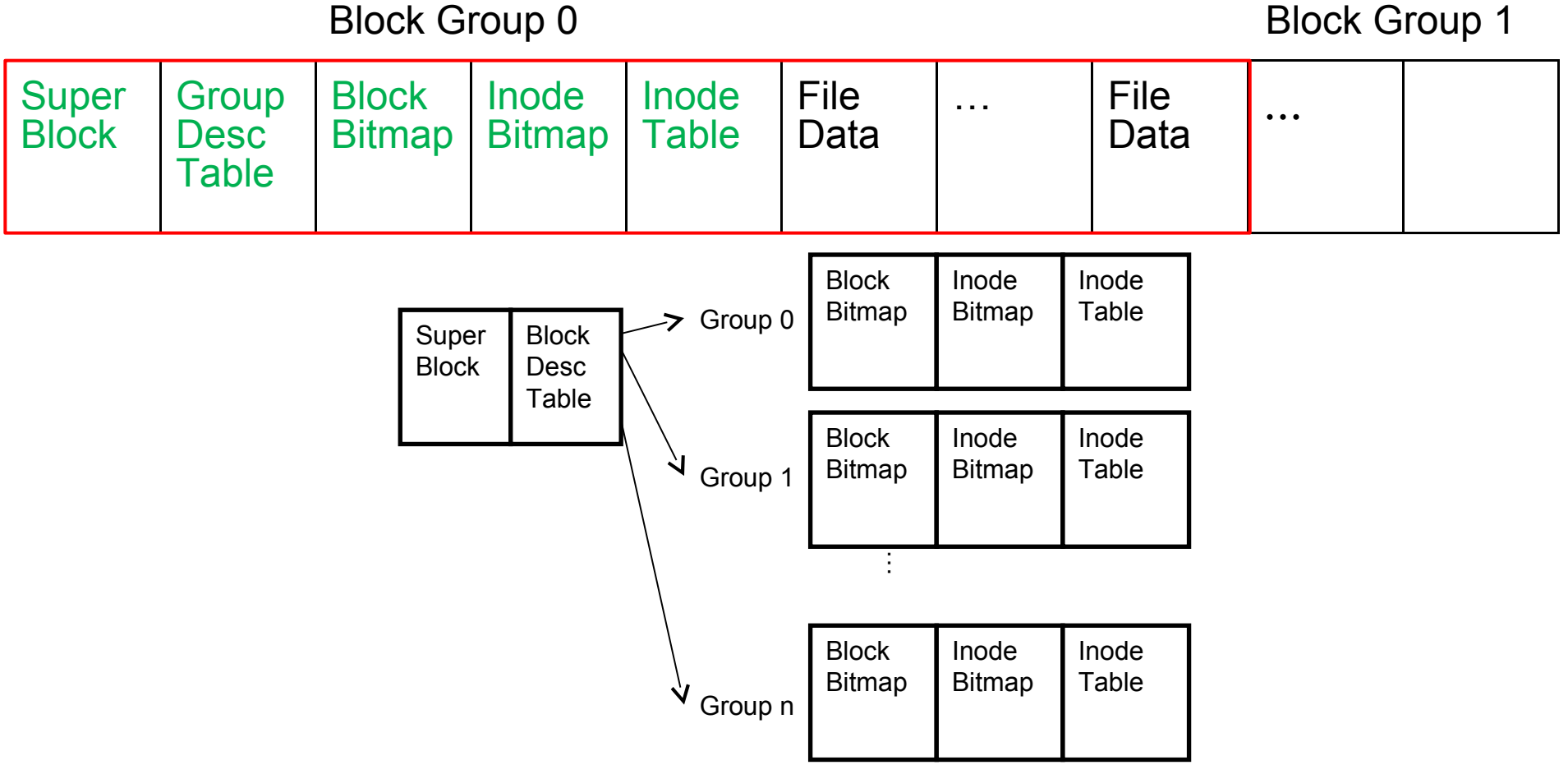
- Log structured FS ONLY contains a log
  - everything written to end
- LSFS dictates how data is stored on disk
- LFS does not

ext3 and JFS

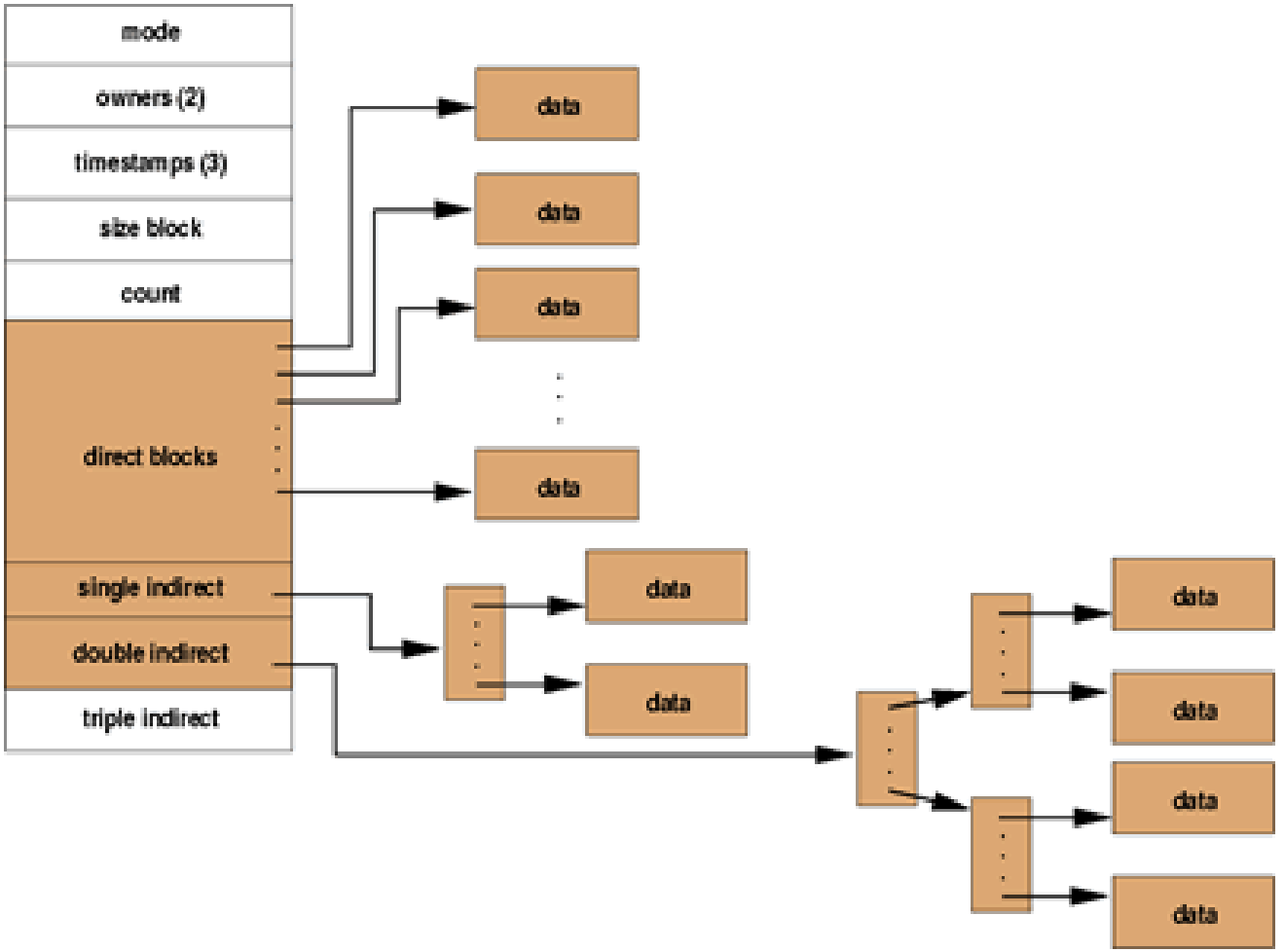
- ext3 grew out of ext2 (same code base)
  - backward compatible
- Two separate layers
  - /fs/ext3 ^ just the FS with transactions
  - /fs/jdb ^ just the journalling stuff
- ext3 calls JFS as needed
  - start/stop transaction
  - journal recovery after unclean reboot

Internals

- Physically organized into block groups
  - each group contains redundant copies of system control blocks



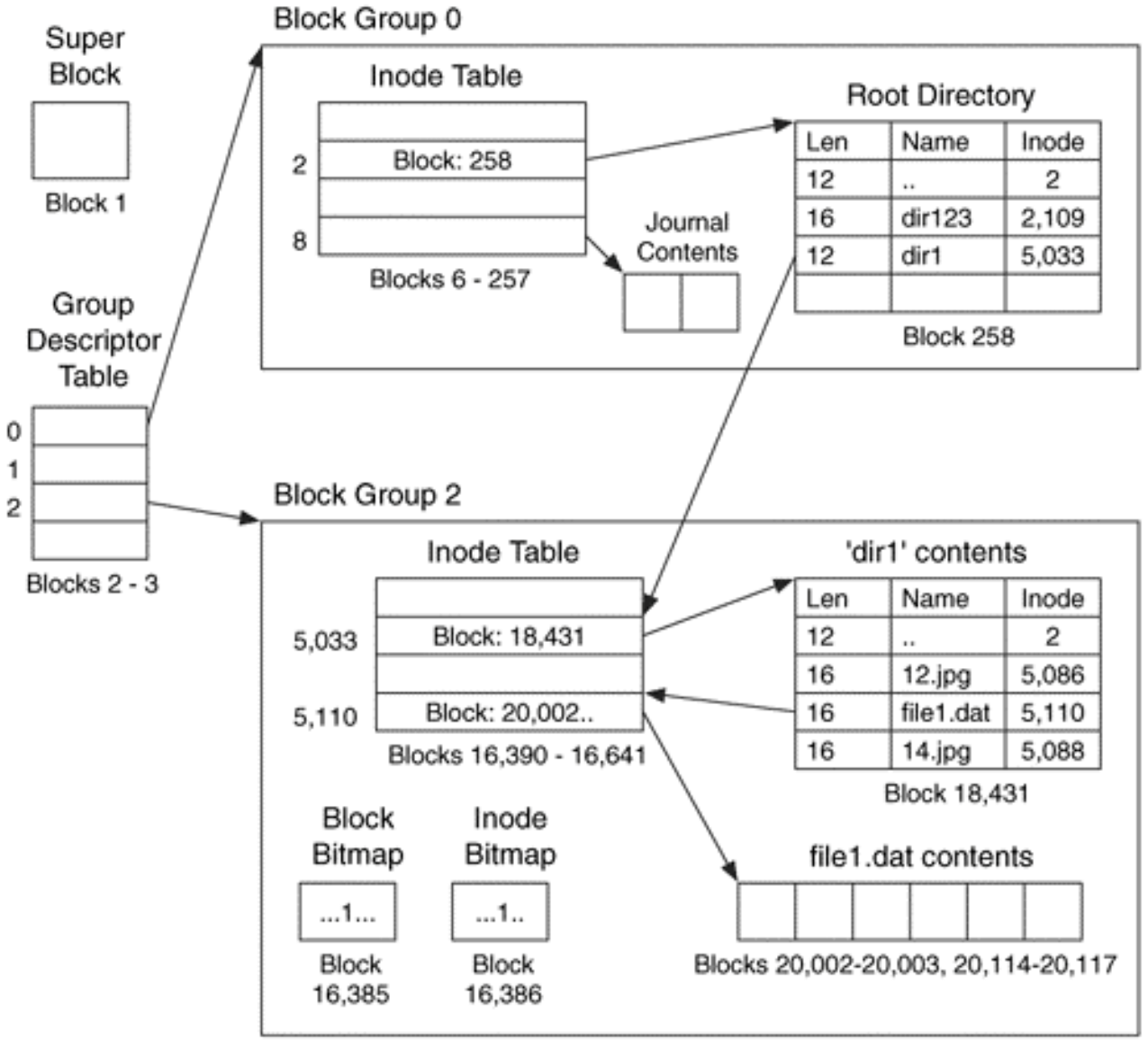
inodes


mode
owners (2)
timestamps (3)
size block
count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

inodes

- Allocation
  - if a new inode is for a non-directory file, allocate inode in the same block group as parent directory
    - unless there is no free inode or block
      - search groups to find one whose free inodes or blocks are less than the average of free inodes and blocks per group
      - if that search fails, find group with smallest number of directories
  - why is all this necessary?

Example


Super Block
Block 1
Group Descriptor Table
0
1
2
Blocks 2 - 3
Block Group 0
Inode Table
2
Block: 258
8
Blocks 6 - 257
Journal Contents
Root Directory
Len
Name
Inode
12
..
2
16
dir123
2,109
12
dir1
5,033
Block 258
Block Group 2
Inode Table
5,033
Block: 18,431
5,110
Block: 20,002..
Blocks 16,390 - 16,641
'dir1' contents
Len
Name
Inode
12
..
2
16
12.jpg
5,086
16
file1.dat
5,110
16
14.jpg
5,088
Block 18,431
Block Bitmap
...1...
Block 16,385
Inode Bitmap
...1..
Block 16,386
file1.dat contents
Blocks 20,002-20,003, 20,114-20,117

File Systems

Files

Directories

Implementation

Examples

- Virtual File System
- CD-Rom FS (ISO 9660)
- MS-DOS
- NTFS
- Ext2
- Ext3
- Distributed File Systems

What Distributed File Systems Provide

- Access to data stored at servers using file system interfaces

- What are the file system interfaces?
  - Open a file, check status of a file, close a file
  - Read data from a file
  - Write data to a file
  - Lock a file or part of a file
  - List files in a directory, create/delete a directory
  - Delete a file, rename a file, add a symlink to a file
  - etc

Why DFSs are Useful

- Data sharing among multiple users
- User mobility
- Location transparency
- Backups and centralized management

Distributed systems

- Key differences from centralized systems
  - No shared memory
  - Communication: delays, unreliable
  - No common clock
  - Independent node failure modes
  - Hardware/software heterogeneity

DFS Structure

- Service – software entity running on one or more machines and providing a particular type of function to a priori unknown clients
- Server – service software running on a single machine
- Client – process that can invoke a service using a set of operations that forms its client interface
  - A client interface for a file service is formed by a set of primitive file operations (create, delete, read, write)
  - Client interface of a DFS should be transparent, i.e., not distinguish between local and remote files

Naming Properties

- Naming – mapping between logical and physical objects
- Location transparency:
  - Name of the file does not reveal the file's physical storage location
- Location independence:
  - Name of file does not need to be changed when file's physical location changes
- A file may be replicated in several sites
  - Mapping hides both the existence of multiple copies and their location

Two naming schemes

- Files named by combination of their host name and local name; guarantees a unique system-wide name
  - Neither location transparent
  - Nor location independent

- “Attach” remote directories to local directories, giving the appearance of a coherent directory tree
  - NFS

What is NFS?

- First commercially successful network file system:
  - Developed by Sun Microsystems for their diskless workstations
  - Designed for robustness and “adequate performance”
  - Sun published all protocol specifications
  - Many many implementations
  - Widely used today

➢ http://nfs.sourceforge.net/

NFS Design Objectives

- Machine and Operating System Independence
  - Could be implemented on low-end machines of the mid-80's
- Transparent Access
  - Remote files should be accessed in exactly the same way as local files
- Fast Crash Recovery
  - No global synchronization
- "Reasonable" performance
  - Robustness and preservation of UNIX semantics were much more important

NFS Protocol

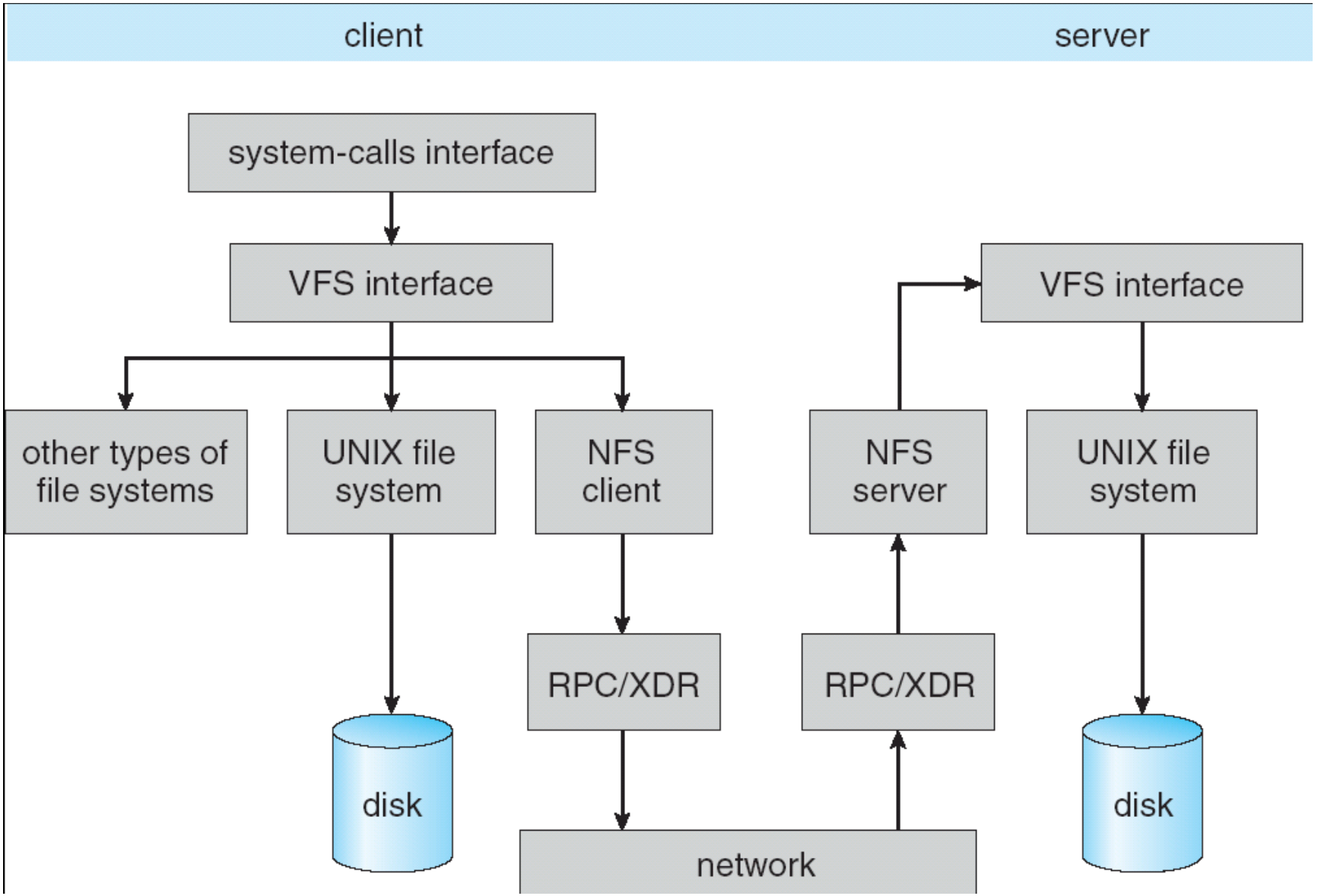
- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)

Three Major Layers in NFS

- Linux file-system interface (based on the open, read, write, and close calls, and file descriptors)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture (implements the NFS protocol)

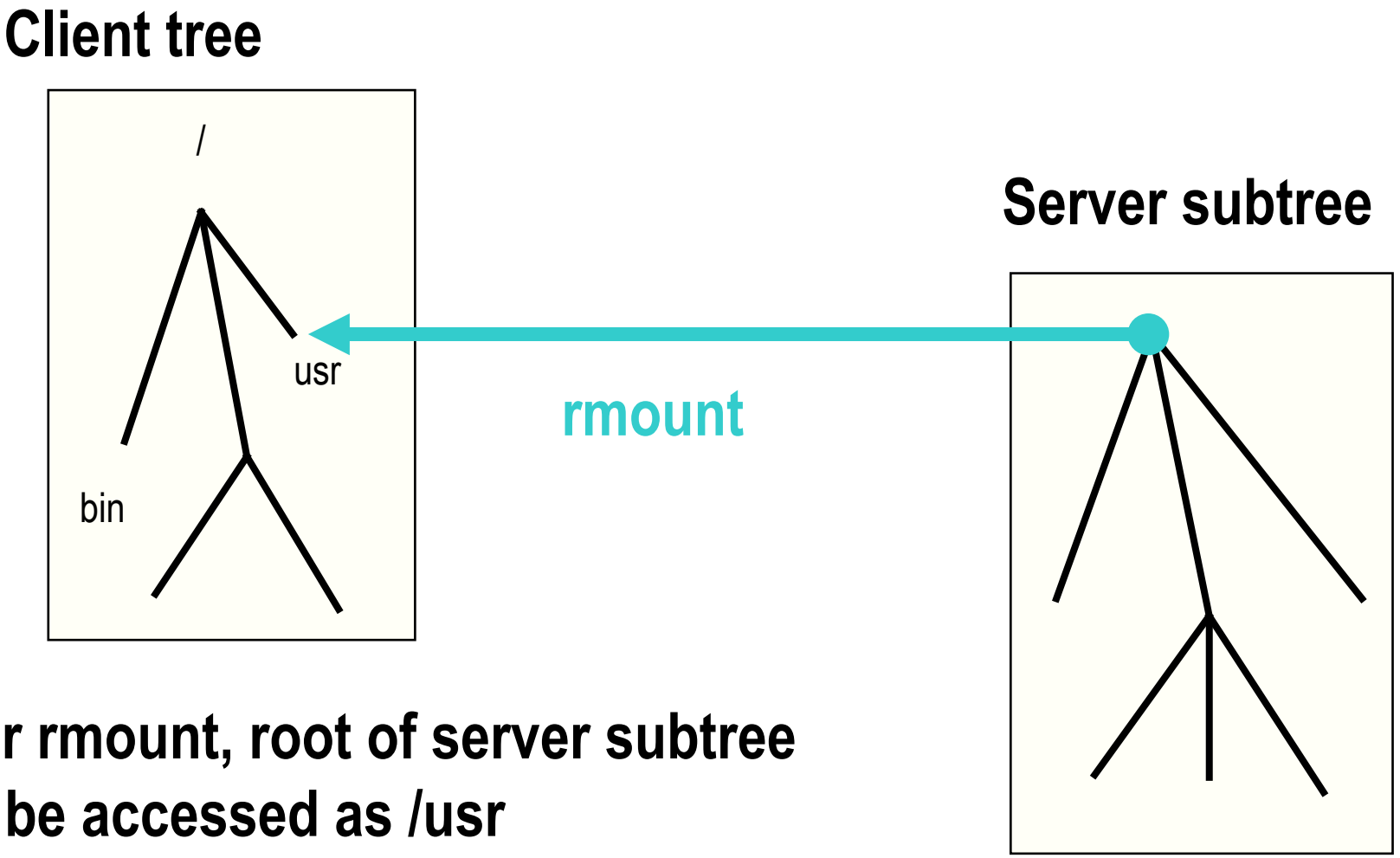
NFS Architecture



Client Side

- Transparent interface to NFS achieved by
  - Mapping between remote file names and remote file addresses via remote mount
  - Extension of UNIX mount
  - Specified in a mount table
  - Makes a remote subtree appear part of a local subtree

Client side: Remote Mount



After rmount, root of server subtree can be accessed as /usr

Mount Table on os

```
[icrk@os /]$ cat /etc/mtab
rootfs / rootfs rw 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
devtmpfs /dev devtmpfs rw,nosuid,size=499808k,nr_inodes=124952,mode=755 0 0
devpts /dev/pts devpts rw,relatime,gid=5,mode=620,ptmxmode=000 0 0
tmpfs /dev/shm tmpfs rw,relatime 0 0
tmpfs /run tmpfs rw,nosuid,nodev,mode=755 0 0
/dev/vda1 / ext4 rw,relatime,data=ordered 0 0
tmpfs /sys/fs/cgroup tmpfs rw,nosuid,nodev,noexec,mode=755 0 0
cgroup /sys/fs/cgroup/systemd cgroup rw,nosuid,nodev,noexec,relatime,release_agent=/lib/systemd/systemd-cgroups-agent,name=systemd 0 0
cgroup /sys/fs/cgroup/cpuset cgroup rw,nosuid,nodev,noexec,relatime,cpuset 0 0
cgroup /sys/fs/cgroup/cpu,cpuacct cgroup rw,nosuid,nodev,noexec,relatime,cpuacct,cpu 0 0
cgroup /sys/fs/cgroup/memory cgroup rw,nosuid,nodev,noexec,relatime,memory 0 0
cgroup /sys/fs/cgroup/devices cgroup rw,nosuid,nodev,noexec,relatime,devices 0 0
cgroup /sys/fs/cgroup/freezer cgroup rw,nosuid,nodev,noexec,relatime,freezer 0 0
cgroup /sys/fs/cgroup/net_cls cgroup rw,nosuid,nodev,noexec,relatime,net_cls 0 0
cgroup /sys/fs/cgroup/blkio cgroup rw,nosuid,nodev,noexec,relatime,blkio 0 0
cgroup /sys/fs/cgroup/perf_event cgroup rw,nosuid,nodev,noexec,relatime,perf_event 0 0
systemd-1 /proc/sys/fs/binfmt_misc autofs rw,relatime,fd=25,pgrp=1,timeout=300,minproto=5,maxproto=5,direct 0 0
tmpfs /var/run tmpfs rw,nosuid,nodev,mode=755 0 0
tmpfs /var/lock tmpfs rw,nosuid,nodev,mode=755 0 0
securityfs /sys/kernel/security securityfs rw,relatime 0 0
configfs /sys/kernel/config configfs rw,relatime 0 0
mqueue /dev/mqueue mqueue rw,relatime 0 0
tmpfs /media tmpfs rw,nosuid,nodev,noexec,relatime,mode=755 0 0
debugfs /sys/kernel/debug debugfs rw,relatime 0 0
hugetlbfs /dev/hugepages hugetlbfs rw,relatime 0 0
sunrpc /var/lib/nfs/rpc_pipefs rpc_pipefs rw,relatime 0 0
nfs.cs.siue.edu:/home/ /home nfs4 rw,relatime,vers=4.0,rsize=1048576,wsize=1048576,namlen=255,hard,proto=tcp,port=0,timeo=600,retrans=2,sec=sys,clientaddr=146.163.150.17,lo
cal_lock=none,addr=146.163.150.227 0 0
binfmt_misc /proc/sys/fs/binfmt_misc binfmt_misc rw,relatime 0 0
```

NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC, forwarded to mount server running on server
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses

NFS Mount Protocol (2)

- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

Stateful File Service

- Mechanism
  - Client opens a file
  - Server fetches information about the file from its disk, stores it in its memory, and gives the client a connection id unique to the client and the open file
  - Identifier is used for subsequent accesses until the session ends
  - Server must reclaim the main-memory space used by clients who are no longer active

Stateless File Server

- Avoids state information by making each request self-contained
- Each request identifies the file and position in the file
- No need to establish and terminate a connection by open and close operations

Stateful vs stateless protocols

- Stateful:
  - fd = open(“/aaa/bbb/ccc”, …);
  - read(fd, …)

- Stateless:
  - read(“/aaa/bbb/ccc”, offset, …);
  - Each procedure call contains all the information necessary to complete the call
  - Server maintains no “between call” information

Failure Recovery in Stateful

- A stateful server loses all its volatile state in a crash
  - Restore state by recovery protocol based on a dialog with clients, or abort operations that were underway when the crash occurred
  - Server needs to be aware of client failures in order to reclaim space allocated to record the state of crashed client processes (orphan detection and elimination)

Advantages of stateless

- Crash recovery is very easy:
  - When a server crashes, client just resends request until it gets an answer from the rebooted server
  - Client cannot tell difference between a server that has crashed and recovered and a slow server

- Simplifies the protocol
  - Client can always repeat any request

Consequences of stateless

- read and write RPCs must specify offset
  - Server does not keep track of current position in the file
  - Longer request messages
  - Slower request processing

- But user still uses conventional Linux APIs

Performance Optimization: Cache

- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally
  - Remote-service mechanism is one transfer approach
  - If needed data not already cached, a copy of data is brought from the server to the user
  - Accesses are performed on the cached copy
  - Cache-consistency problem – keeping the cached copies consistent with the master file

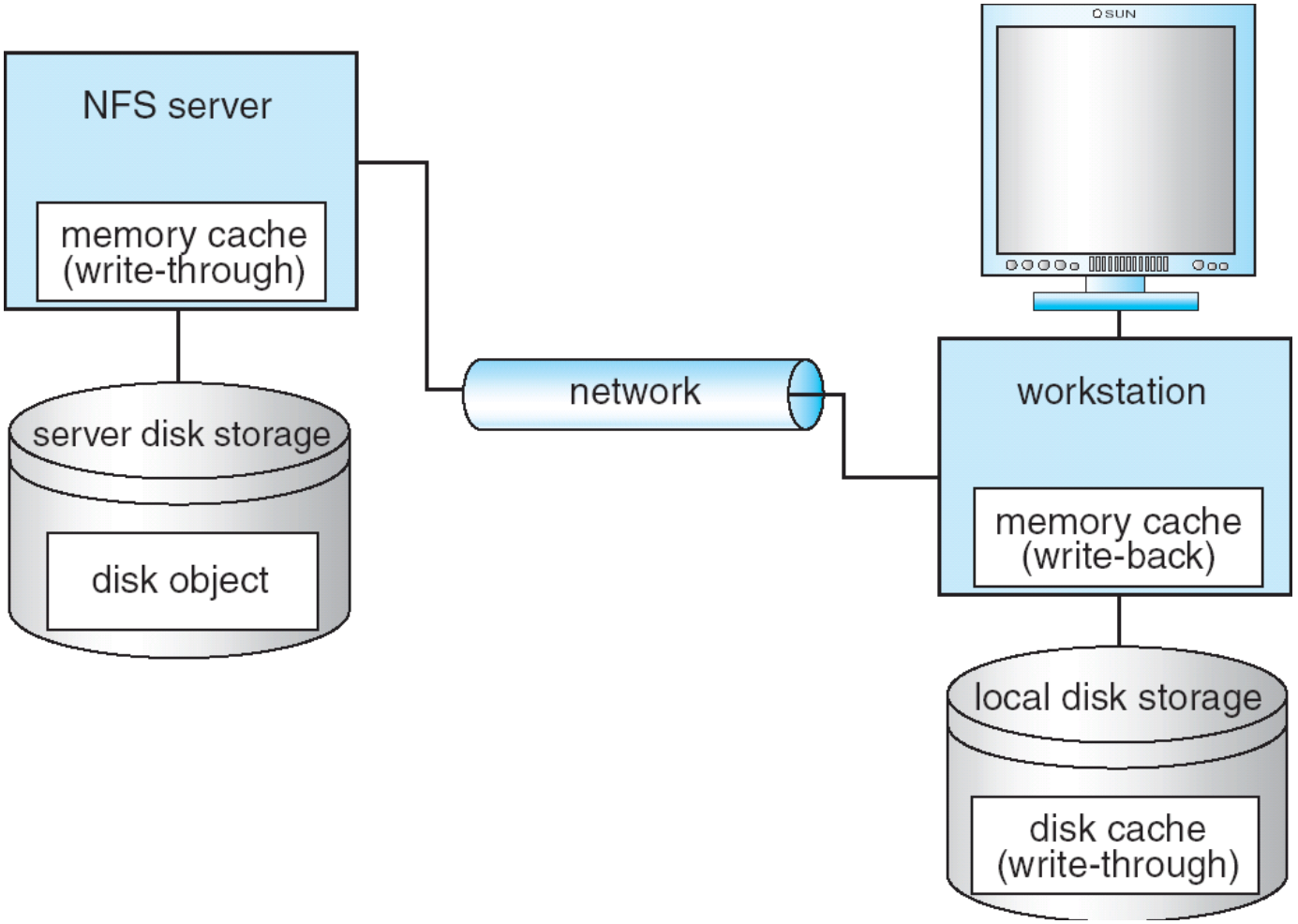
Cache Update Policy

- Write-through – write data through to master's copy as soon as they are placed on any cache
  - Reliable
  - Only reads benefit
  - Poor performance
    - Network traffic increase
    - Reads stuck behind writes
    - Higher server load

Cache Update Policy

- Write-back (delayed write)– modifications written to the cache and written through to the server later
  - Data may be overwritten before written back, and so aggregated
  - Poor reliability
    - Unwritten data will be lost whenever client crashes
  - Standard OS mechanisms limit data loss
    - Scan cache at regular intervals and flush dirty blocks since the last scan
    - Write-on-close, writes data back to the server when the file is closed.
- Big benefit for files that are open for long periods and frequently modified

Cache in NFS


NFS server
memory cache
(write-through)
server disk storage
disk object
network
SUN
workstation
memory cache
(write-back)
local disk storage
disk cache
(write-through)