

TDDDB68 Concurrent Programming and Operating Systems

Lecture: Memory management II + File systems

Mikael Asplund
Real-time Systems Laboratory
Department of Computer and Information Science

Copyright Notice:

Thanks to Christoph Kessler for much of the material behind these slides.

The lecture notes are partly based on Silberschatz's, Galvin's and Gagne's book ("Operating System Concepts", 7th ed., Wiley, 2005). No part of the lecture notes may be reproduced in any form, due to the copyrights reserved by Wiley. These lecture notes should only be used for internal teaching purposes at the Linköping University.

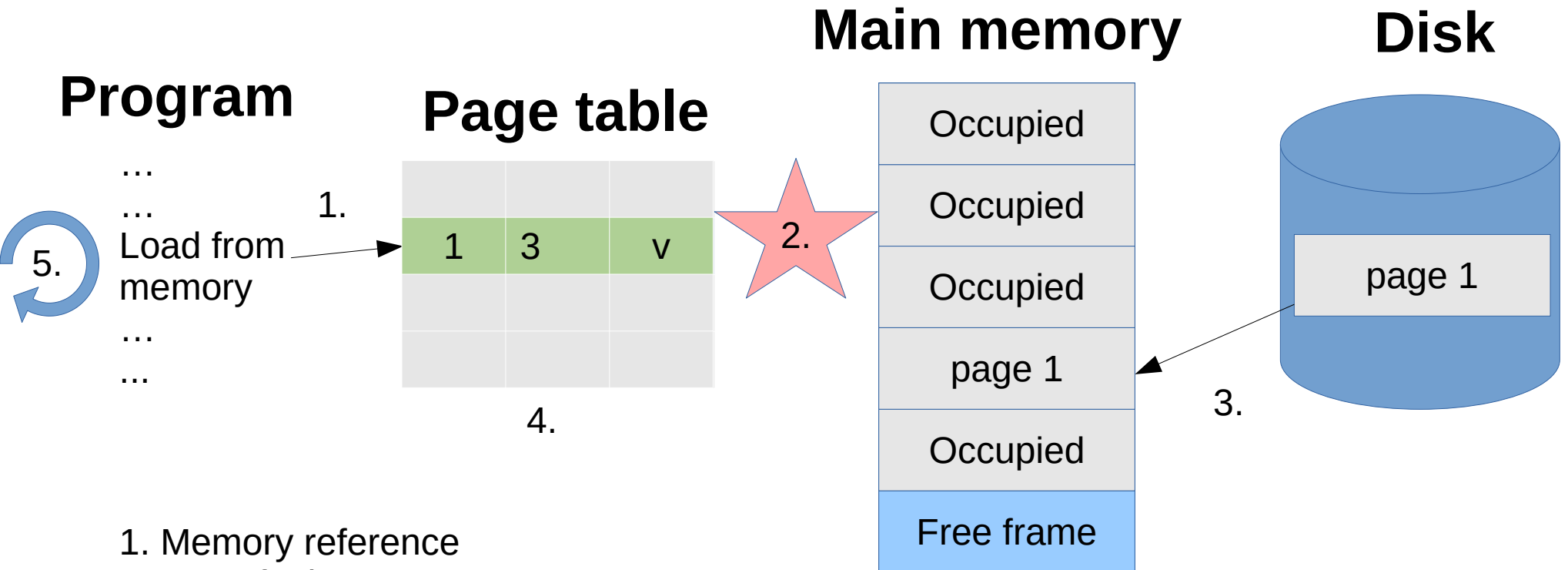
Reading

- 9e:
 - Page replacement: 9.4
 - Thrashing: 9.6
 - File system interface: 11.1 (the rest superficially)
 - File system implementation: 12.1-12.7
- 10e:
 - Page replacement: 10.4,
 - Thrashing: 10.6
 - Memory compression:10.7
 - File system interface: 13.1 (the rest superficially)
 - File system implementation: 14.1-14.7

Page Replacement

Steps in Handling a Page Fault

(Case: a free frame exists)

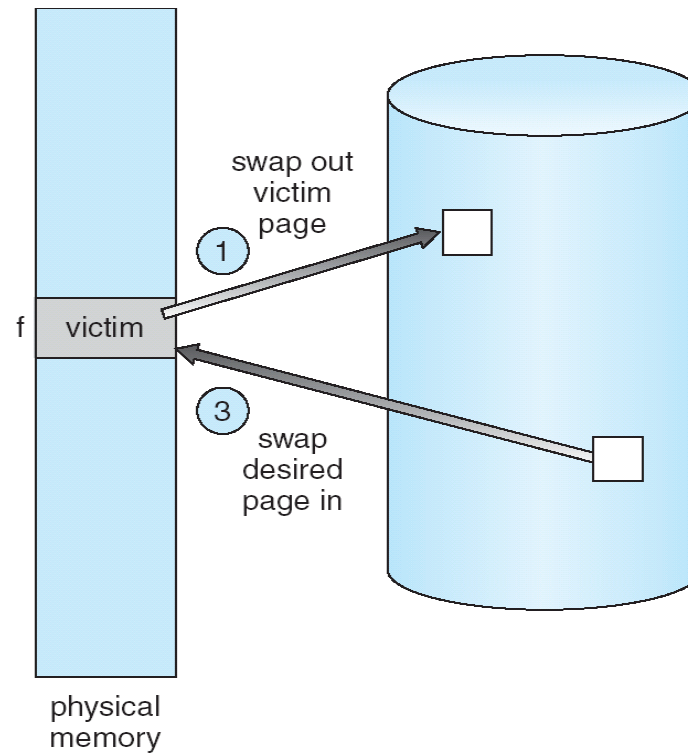
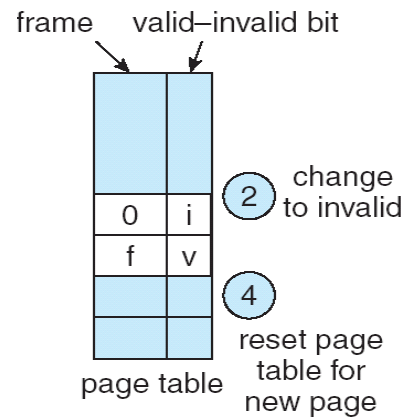


1. Memory reference
2. Page fault! → Interrupt
- 3. OS moves page into memory**
4. Update page table
5. Restart memory access instruction

Page Replacement

- When no free frames, move one page out.
- Use **modify (dirty) bit** to reduce overhead of page transfers
 - only modified pages are written to disk!

Basic Page Replacement












How to compare algorithms for page replacement?

- **Goal:** find algorithm with lowest page-fault rate.
- **Method:** Simulation.
 - Assume initially empty page table
 - Run algorithm on a particular string of memory references (reference string – page numbers only)
 - Count number of page faults on that string.
- In all our examples, the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

First-In-First-Out (FIFO) Algorithm

- Use a time stamp or a queue
- *Victim* is the "oldest page"
- Assume table size = 3 frames / process
(3 pages can be in memory at a time per process)
and reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

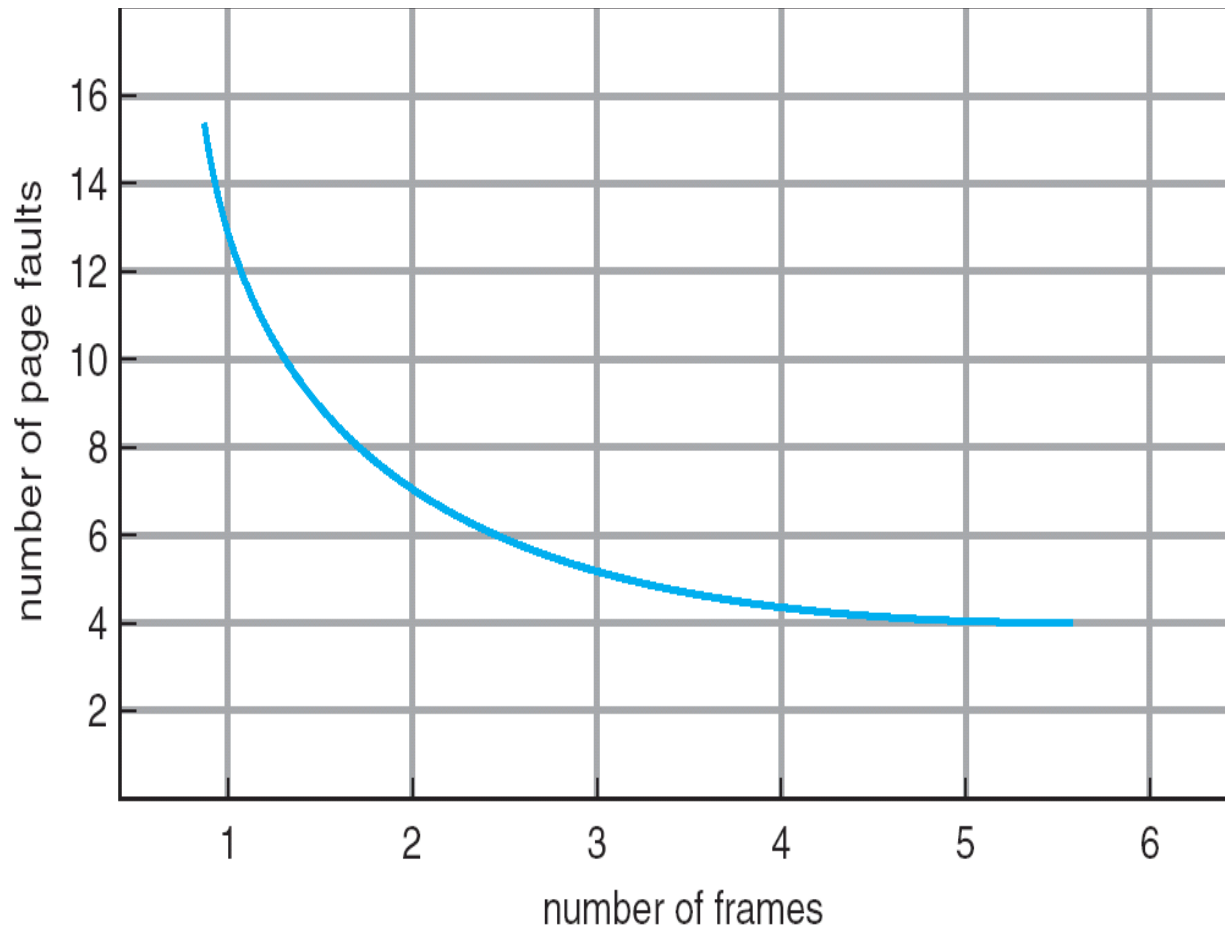
	 1	1	1	 4	4	4	 5	5	5	5	5	5
		 2	2	2	 1	1	1	1	1	 3	3	3
			 3	3	3	 2	2	2	2	2	 4	4

A total of 9 page faults

After page 3 is loaded, page 1 is the oldest of them all (gray box)

The fact that we re-use an existing page does not alter who has been in there the longest...

Expected Page Faults Versus Number of Frames


















Generally, more frames \Rightarrow Less page faults ?

Same FIFO: More frames = better ?

- 4 frames/process
(4 pages can be in memory at a time per process)
- Reference string:

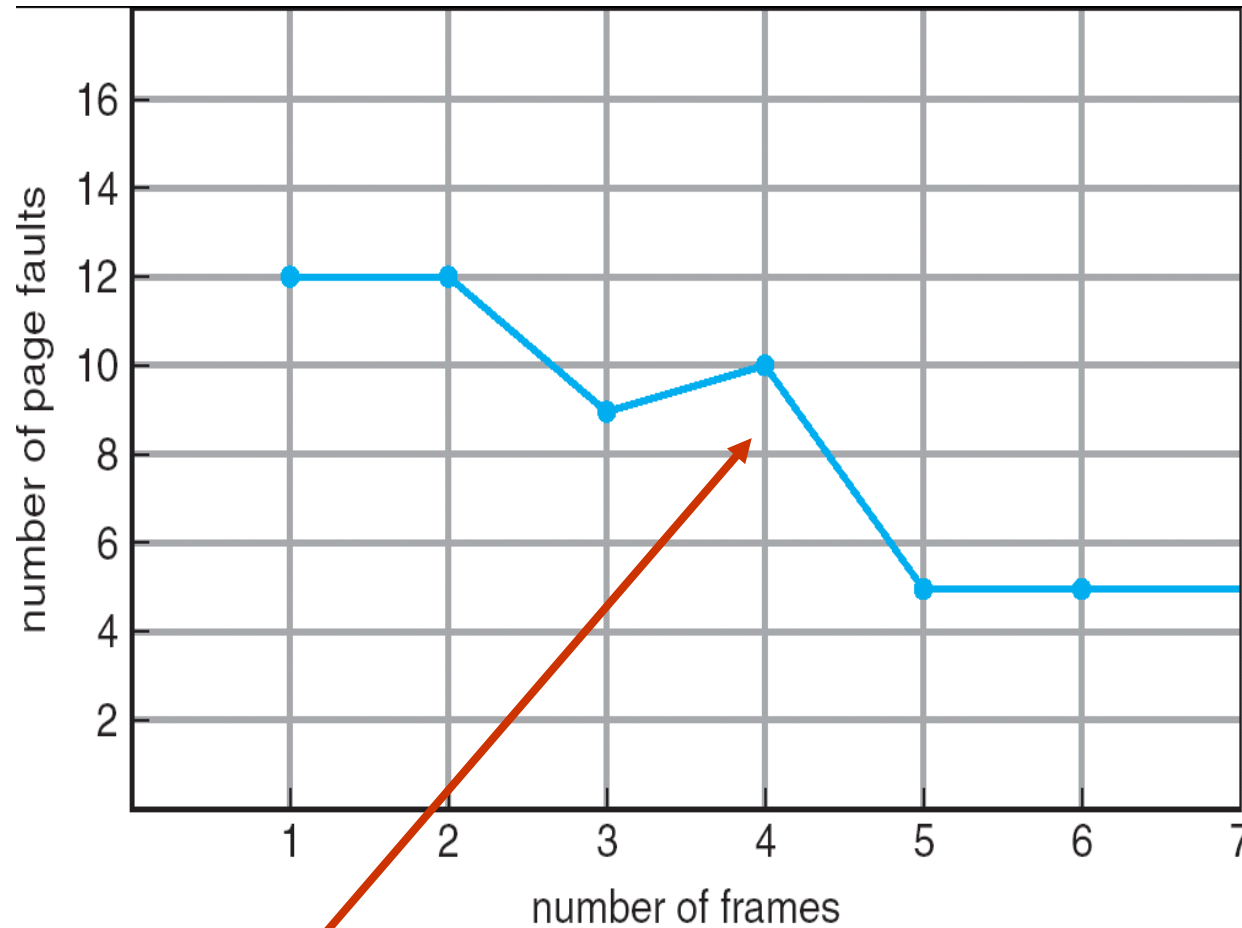
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

	 1	1	1	1	1	1	 5	5	5	5	 4	4
		 2	2	2	2	2	 2	 1	1	1	 1	 5
			 3	3	3	3	3	 3	 2	2	2	 2
				 4	4	4	4	4	 4	 3	3	3

A total
of 10 page
faults

- FIFO Replacement – **Belady's Anomaly**
 - more frames with more page faults – possible!

FIFO illustrating Belady's Anomaly









more frames but more page faults

An Optimal Algorithm

[Belady 1966]

- "optimal":
 - has the lowest possible page-fault rate (NB: still ignoring dirty-ness)
 - does not suffer from Belady's anomaly
- Belady's Algorithm: *Farthest-First, MIN, OPT*
 - Replace page that *will not be used for the longest period of time....*
 - How do you know this?
- Example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

	 1	1	1	1	1	1	1	1	1	1	 4	4
		 2	2	2	2	2	2	2	2	2	2	2
			 3	3	3	3	3	3	3	3	3	3
				 4	4	4	 5	5	5	5	5	5

A total of 6 page faults









Remark: Belady's algorithm is only optimal if there are no dirty write-backs. Otherwise it is just a heuristic algorithm.

We will need frames 1, 2 and 3 before we need frame 4 again, thus throw it out!

Least Recently Used (LRU) Algorithm

- Optimal algorithm not feasible?
....try using recent history as approximation of the future!
- **Algorithm:**
 - Replace the page that *has not been used for the longest period of time*

• Example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

	 1	1	1	1	1	1	1	1	1	1	1	 5
		 2	2	2	2	2	2	2	2	2	2	2
			 3	3	3	3	 5	5	5	5	 4	4
				 4	4	4	4	4	4	 3	3	3

A total
of 8 page
faults

Out of pages 1,2,3 and 4,
page 3 is the one *not* used for
the longest time...

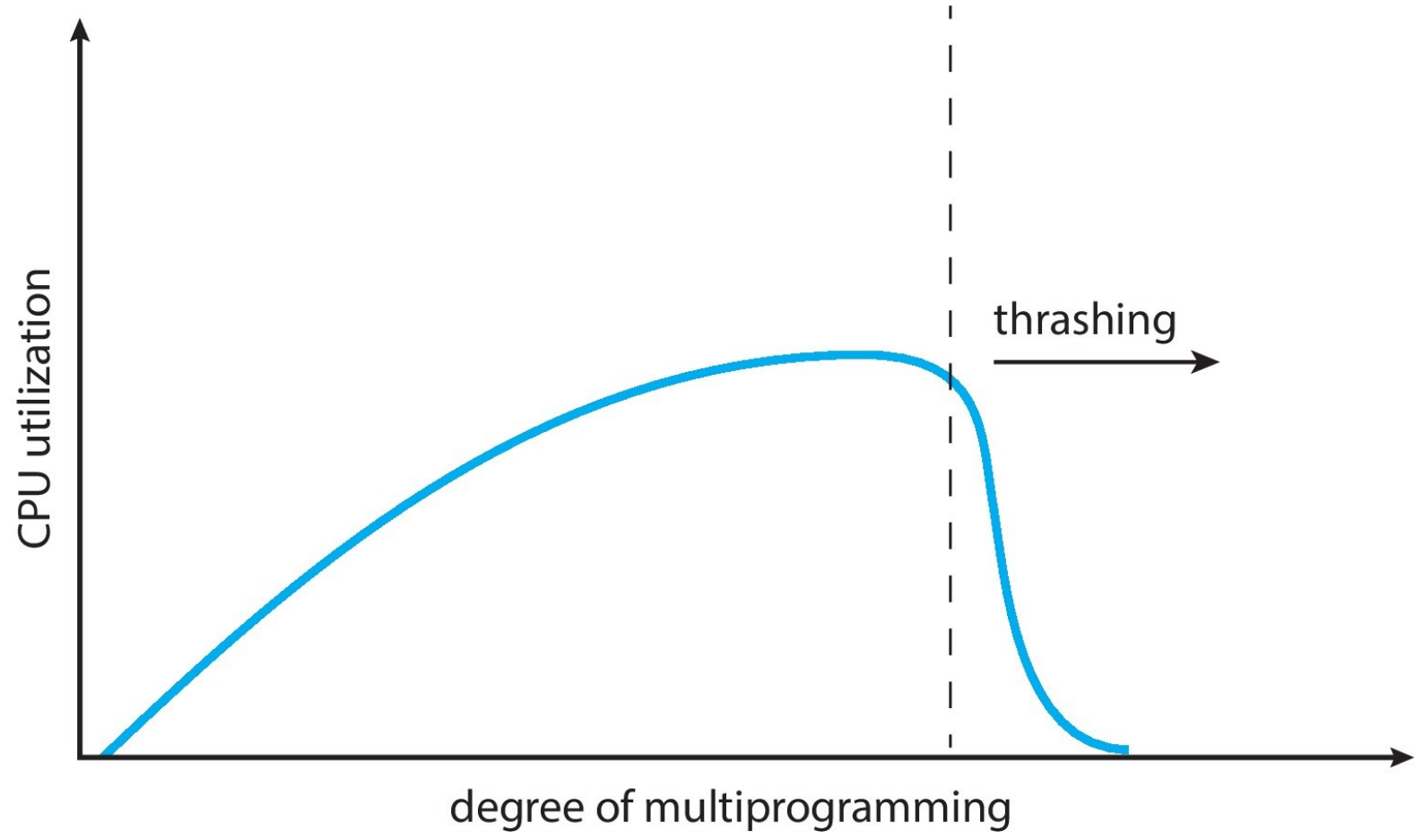
Thrashing

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace existing frame
 - But quickly need replaced frame back

- This leads to:
 - ▶ Low CPU utilization
 - ▶ Operating system thinking that it needs to increase the degree of multiprogramming
 - ▶ Another process added to the system

Effect of thrashing



Demand Paging and Thrashing

- Why does demand paging work?

Locality model

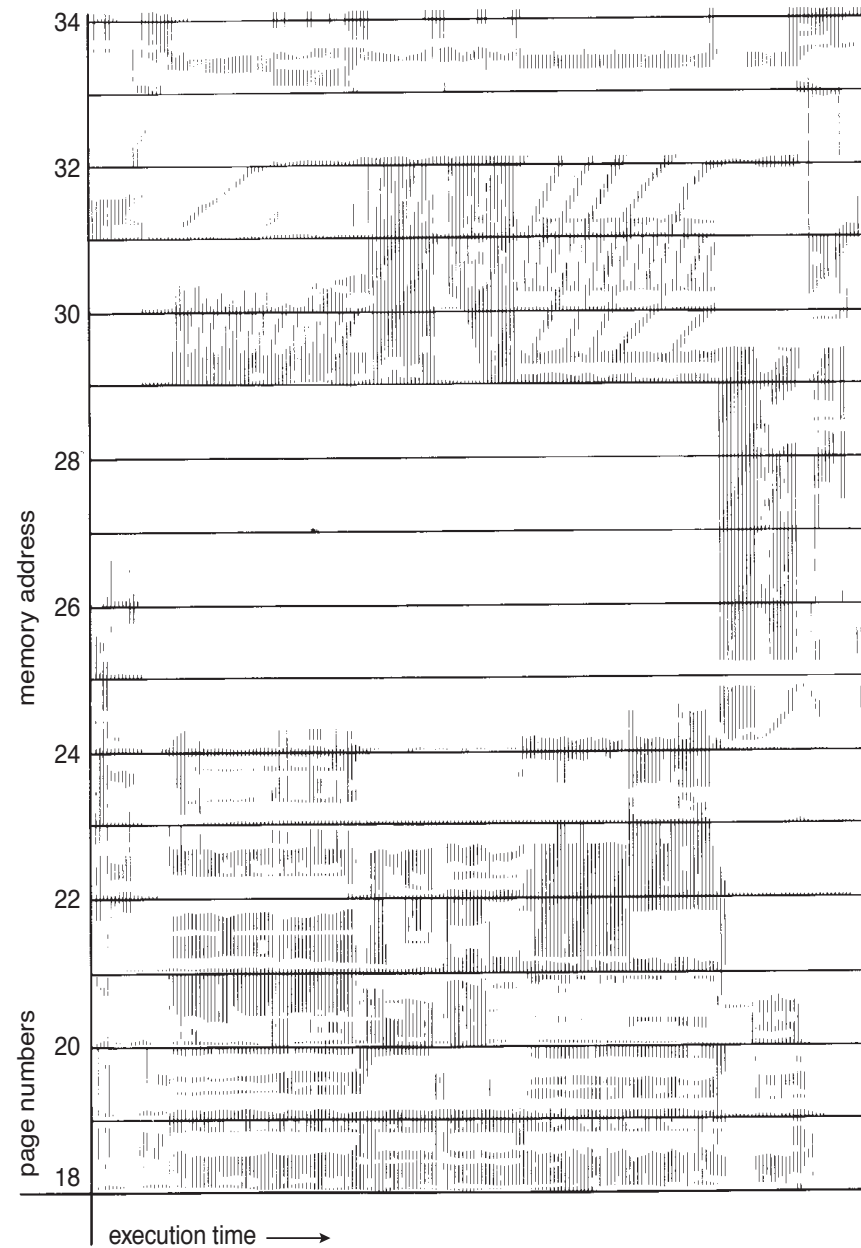
- Process migrates from one locality to another
- Localities may overlap

- Why does thrashing occur?

Σ size of locality > total memory size

- Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



Working-Set Model

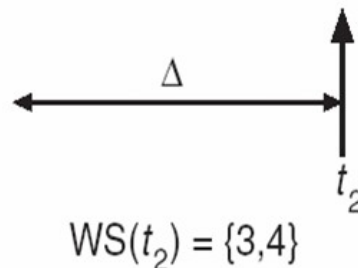
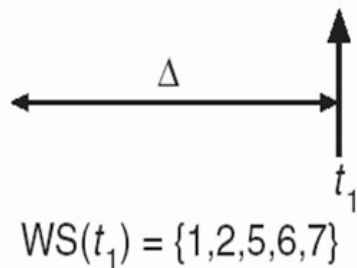
- Δ \equiv working-set window \equiv a fixed number of page references
Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality

Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...

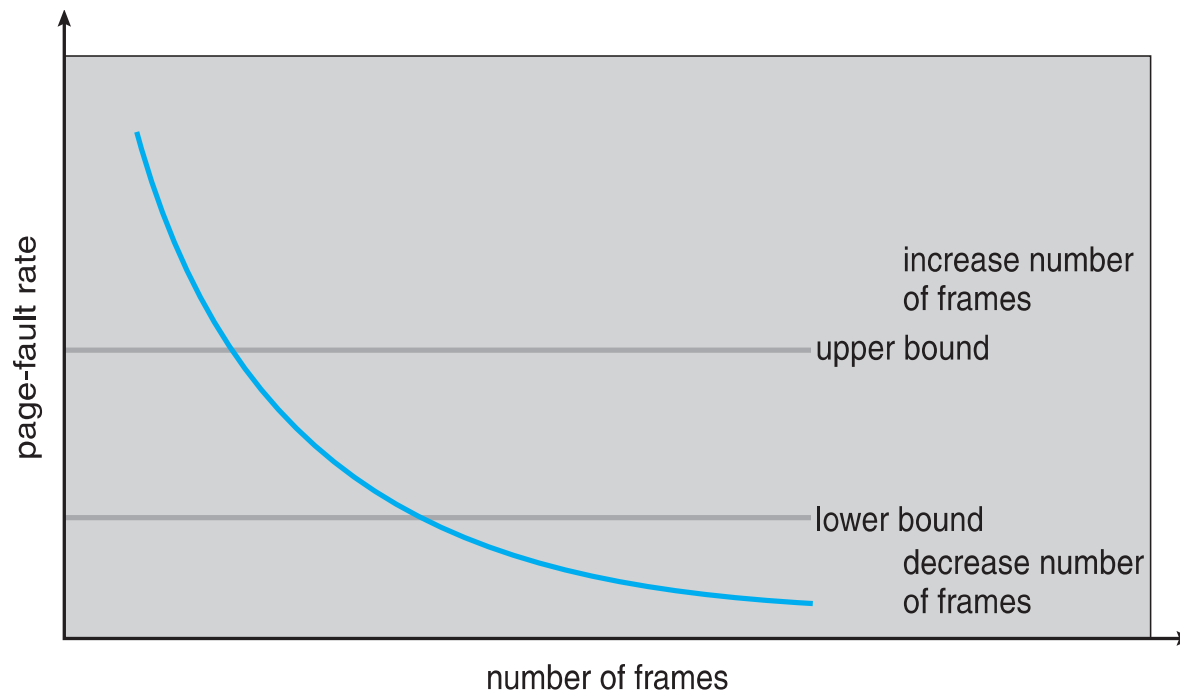


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

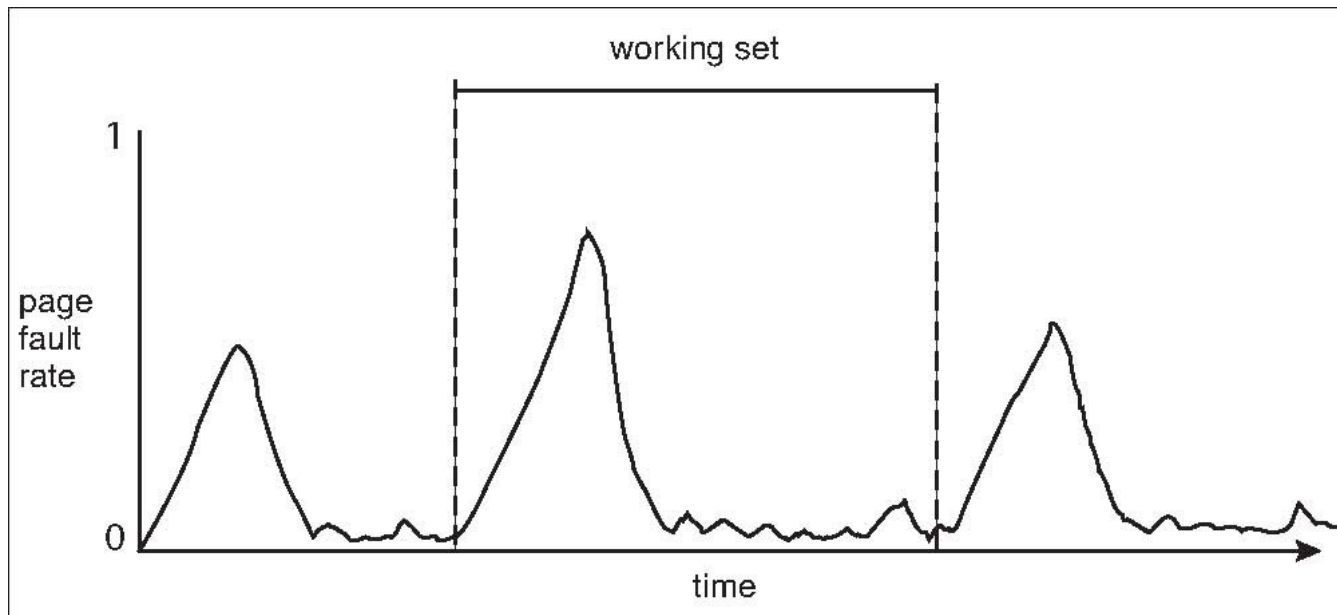
Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Memory compression

Motivation

- Disk access is very slow
- Memory is fast, and CPU is even faster
- Sometimes there is no more disk
- Flash memory degrades from frequent erasures

Memory compression

- Compress memory in RAM
- Possible to do in HW, but mostly done by OS
- Examples:
 - Linux: zram, zswap
 - Windows 10
 - Mac OS X

File systems

File system consists of
interface + implementation

Storing data

- Primary memory is volatile
 - need secondary storage for long-term storage
- A *disk* is essentially a linear sequence of numbered blocks
 - With 2 operations: write block b , read block b
 - Low level of abstraction

The **file** abstraction

- Provided by the OS
- Smallest allotment of secondary storage known to the user
 - Typically, contiguous logical address space
- Organized in a *directory* of files
- Has
 - Attributes (Name, id, size, ...)
 - API (operations on files and directories)

File Attributes

- **Name** – the only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can read, write, execute
- **Time, date, and user identification** – data for protection, security, and usage monitoring

Meta data

Such information *about* files (i.e., **meta-data**) is kept in a **directory structure**, which is maintained on the disk.

Stored in a **File Control Block (FCB)** data structure for each file

Open in unix:

`open ("filename", "mode")`

returns a *file descriptor / handle* = index into a per-process →

table of open files (part of PCB)

(or an error code)

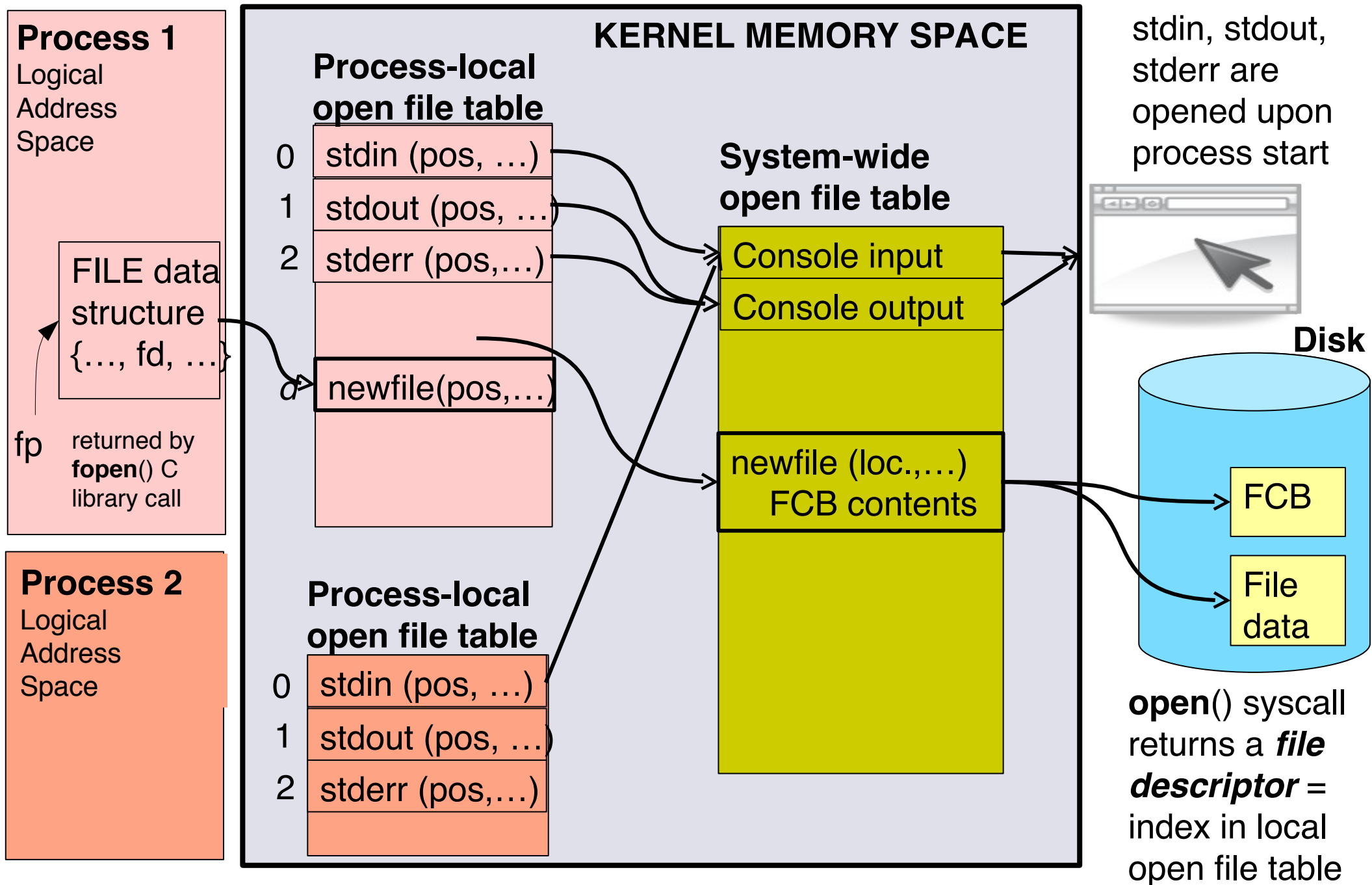
Data to manage open files

- **Disk location of the file** (and other metadata from FCB)
- **File-open count:** count number of times a file is opened – to allow removal of data from open-file table when last process closes it
 - shared by all processes who opened the file
- **File pointer (seekpos):** pointer to next read/write location
 - one for every *open* system call (process)

Storing open file data

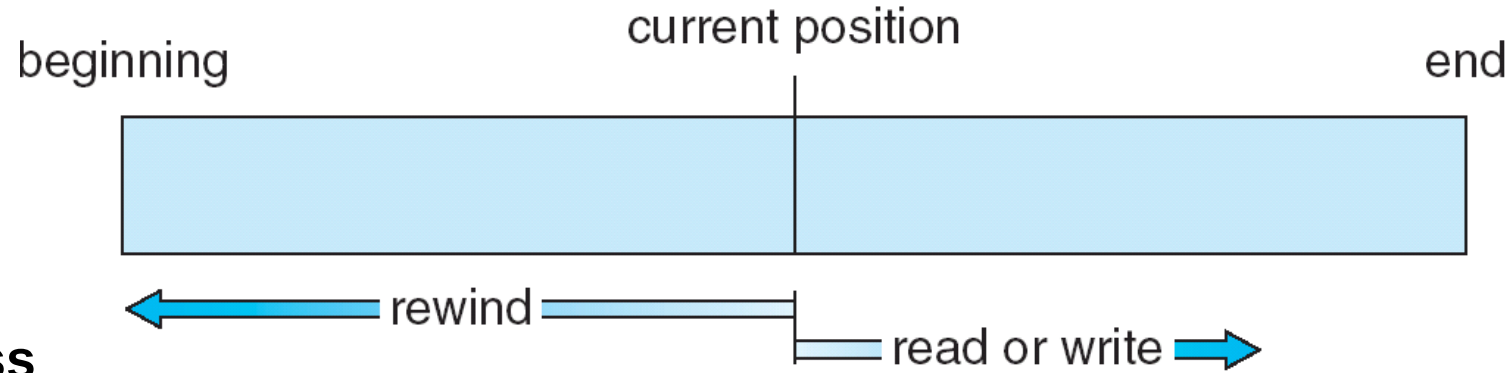
- Collected in a system-wide table of open files and process-local open file tables (part of PCB)
- Process-local open file table entries point to system-wide open file table entries
- Semantics of fork()?

File descriptors and open file tables

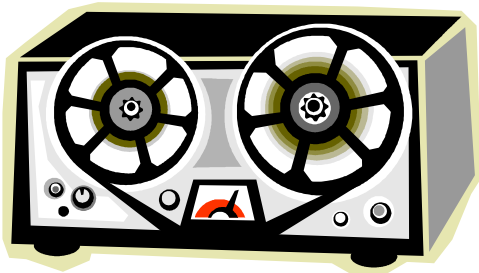


What about the Pintos file table?

Access Methods



- **Sequential Access**



read next block
write next block
reset (rewind)

- **Direct Access**

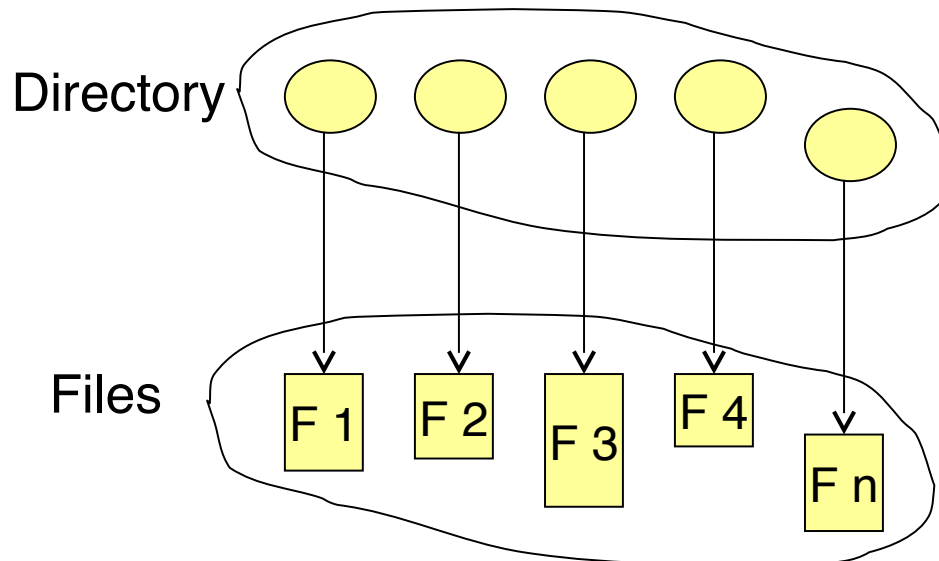


read block n
write block n
position to n
 read next block
 write next block

n = relative block number from beginning of file)

Directory Structure

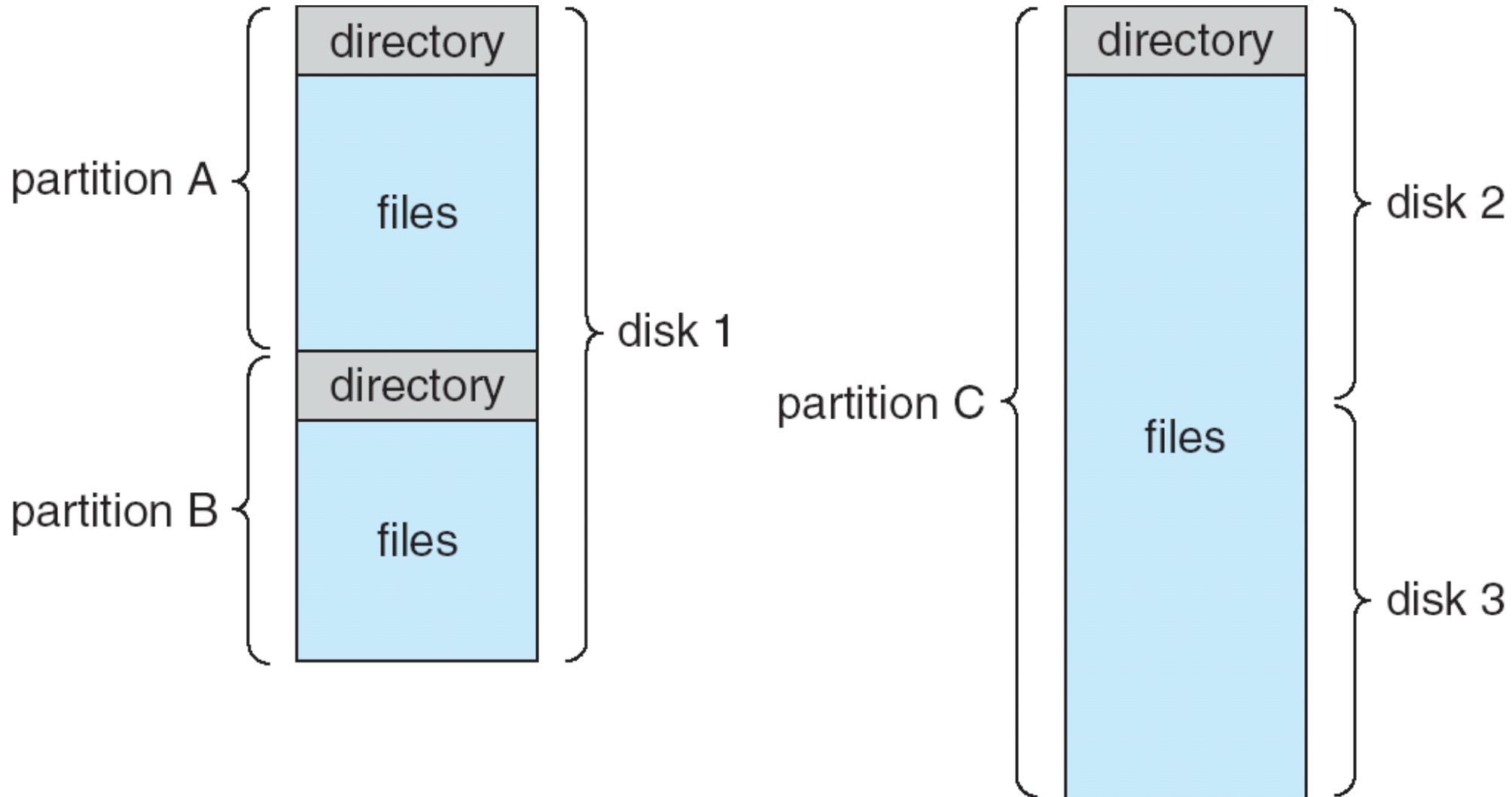
- Files in a system organised in **directories**
 - A collection of *nodes* containing information about all files
- Both the directory structure and the files reside on disk.



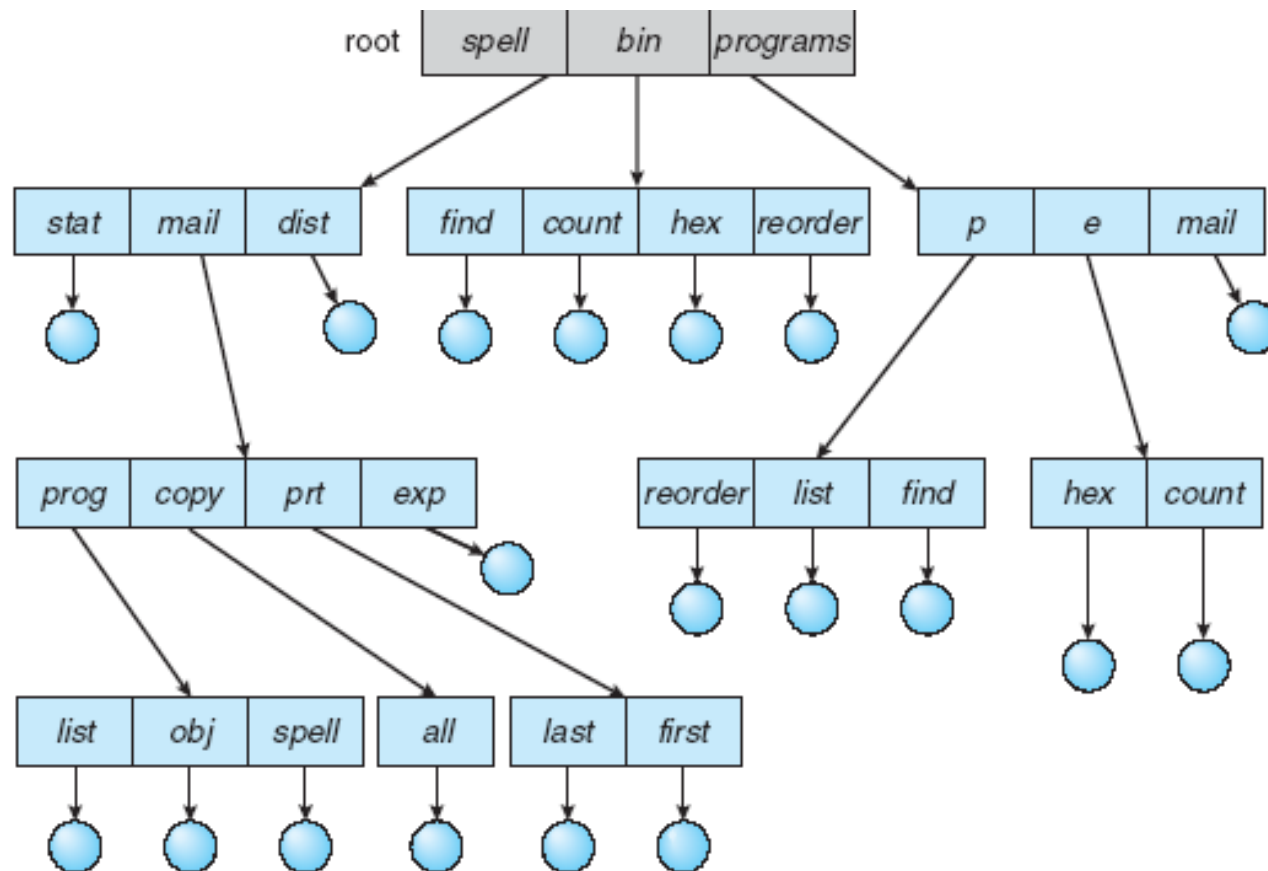
Directory API

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

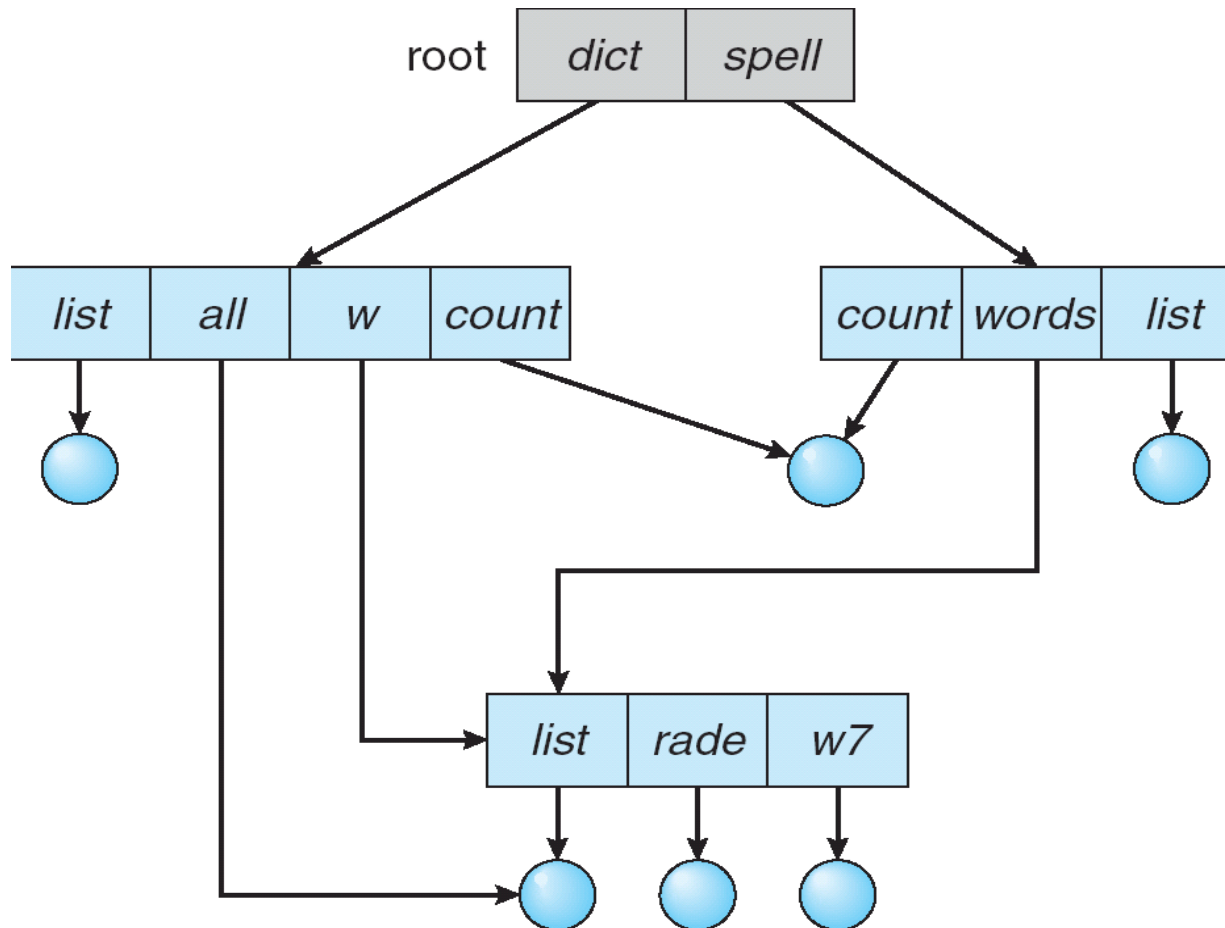
Examples of File-system Organization



Tree-Structured Directories



Acyclic-Graph Directories



Hard links

- Direct pointer (block address) to a directory or file
- Cannot span partition boundaries
- Need be updated when file moves on disk
- Unix: **ln** <filename> <linkname>

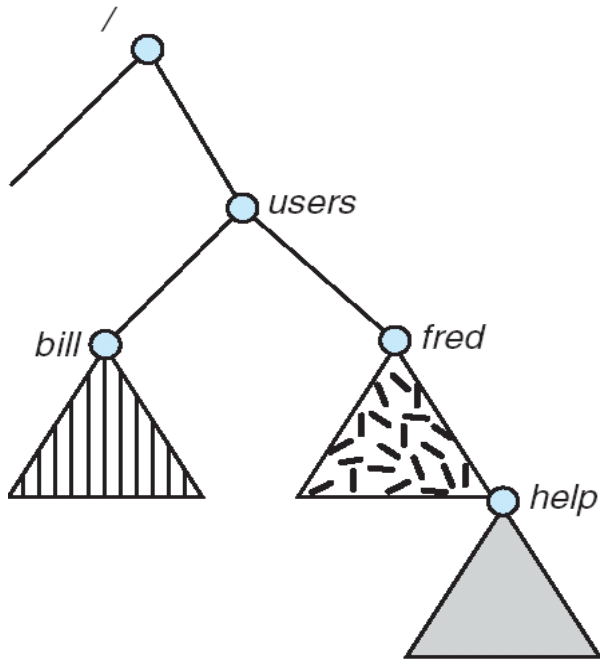
Soft links

- **Soft links** (symbolic links, "shortcut", "alias")
 - files containing the actual (full) file name
 - still valid if file moves on disk
 - no longer valid if file name (or path) changes
 - Not as efficient as hard links (one extra block read)
 - Unix: **ln -s** <filename> <linkname>

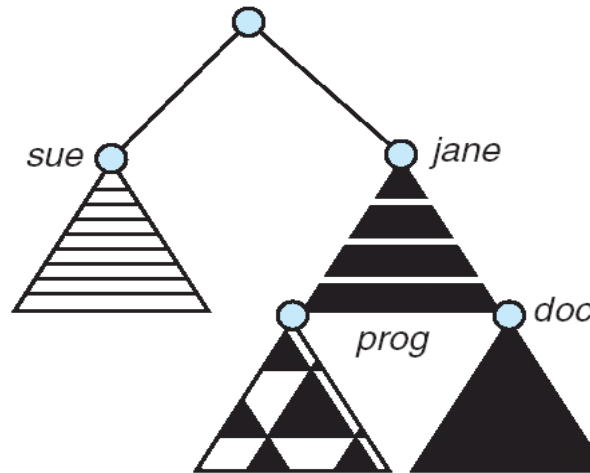
File System Mounting

- A file system must be **mounted** before it can be accessed
- Mounting combines multiple file systems in one namespace
- An unmounted file system is mounted at a **mount point**
- In Windows, mount points are given names C:, D:, ...

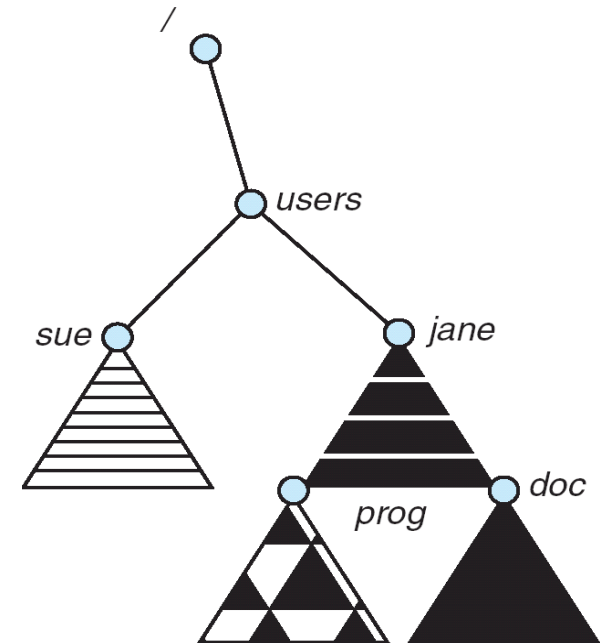
Example



(a)
Existing
file
system



(b)
Unmounted volume
residing on
/device/dsk



Mount point:
Mounted /device/dsk
over *users*

File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- In order to have a protection scheme, the system should have
 - **User IDs** - identify users, allowing permissions and protections to be per-user
 - **Group IDs** - allow users to be in groups, permitting group access rights

Protection

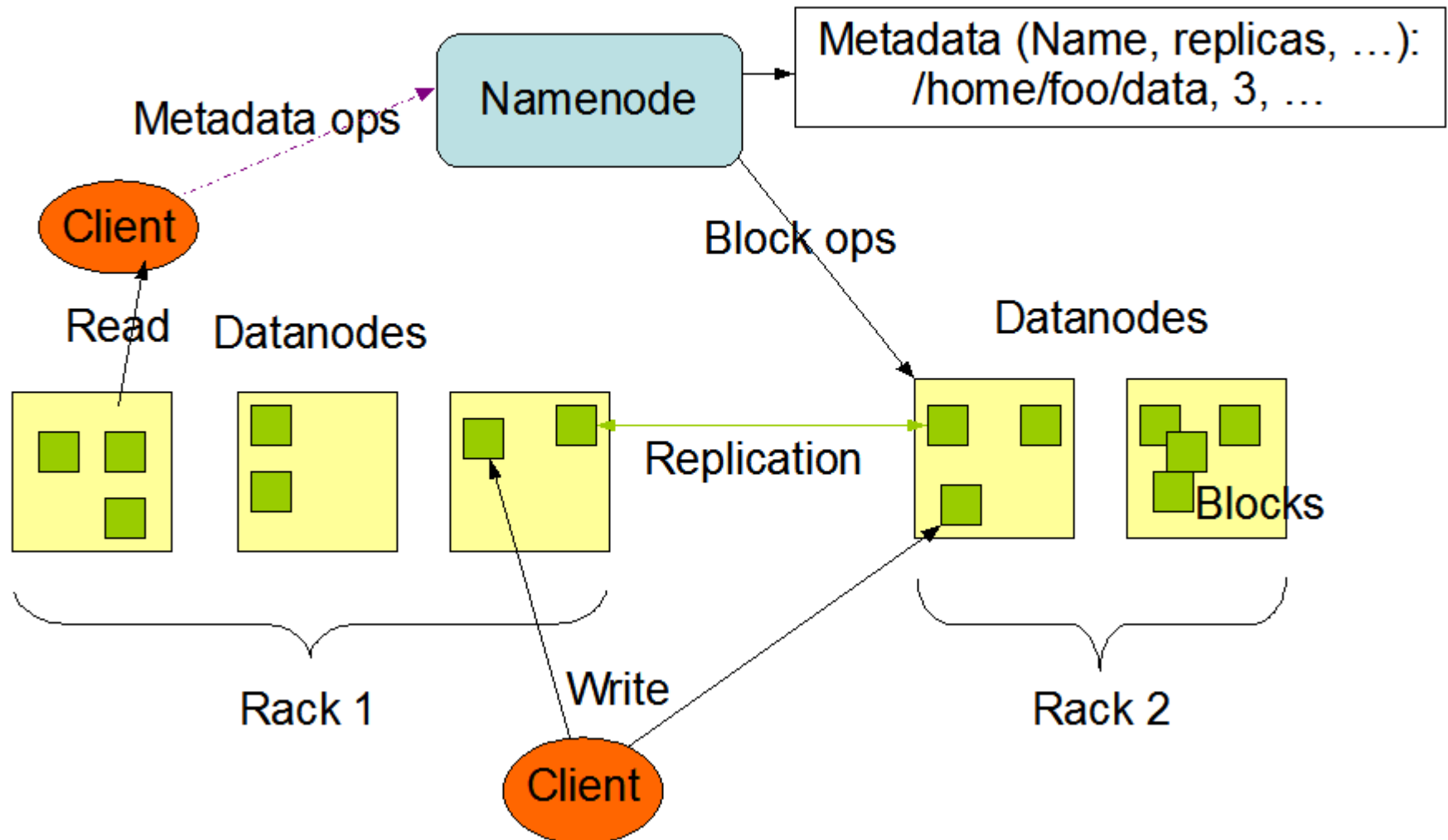
- File owner/creator should be able to control:
 - what can be done
 - by whom
- Types of access
 - **Read**
 - **Write**
 - **Execute**
 - **Append**
 - **Delete**
 - **List**
- Each OS has its own policies and mechanisms for protection

Sharing across a network

- Distributed system
- Network File System (NFS) is a common distributed file-sharing method
- SMB (Windows shares) is another
- Protection is a challenge!

Hadoop Distributed File System (HDFS)

HDFS Architecture

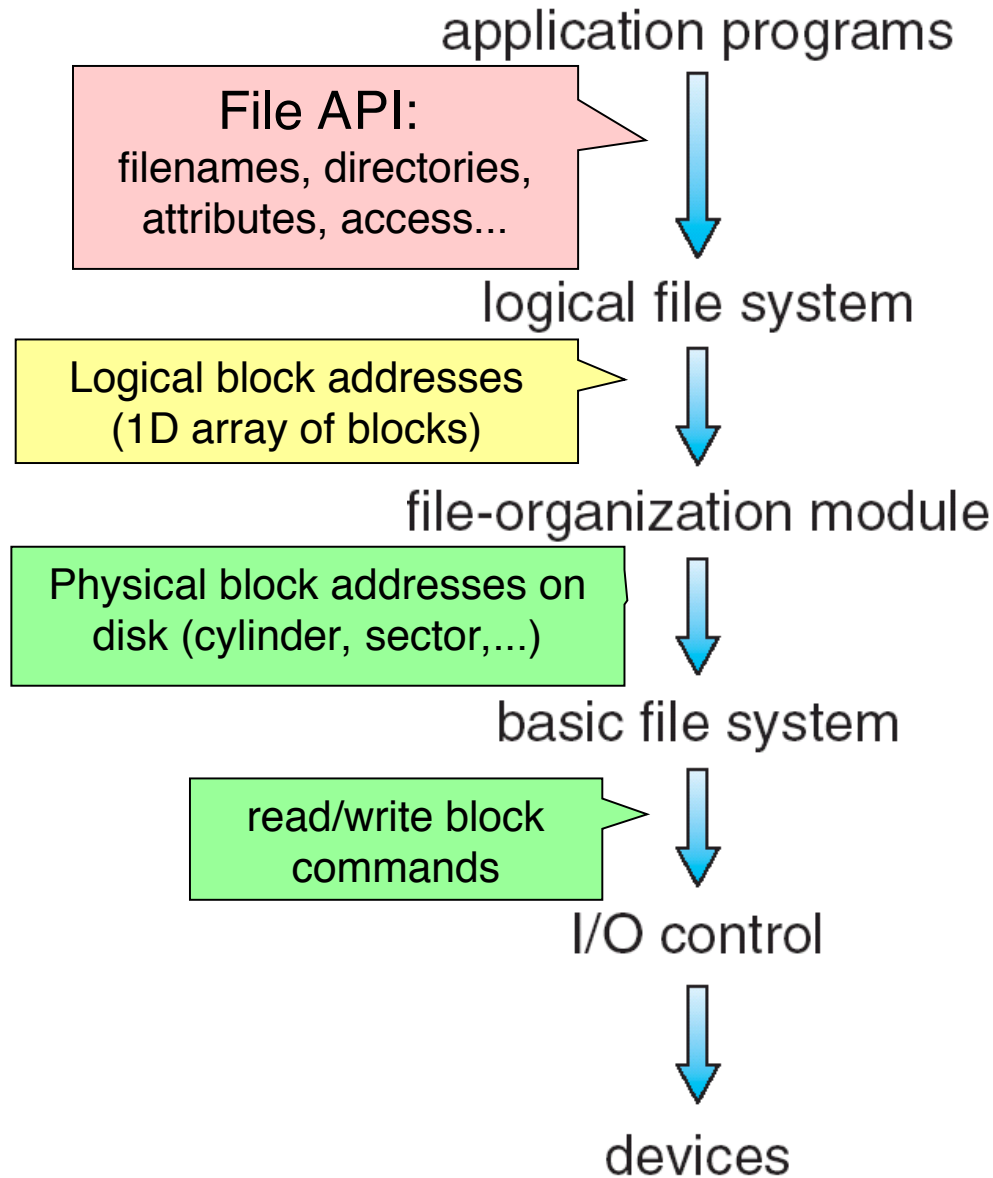


File system implementation

File-System Structure

- File system resides on secondary storage (disks)
- File system organized into layers

File-System Layers

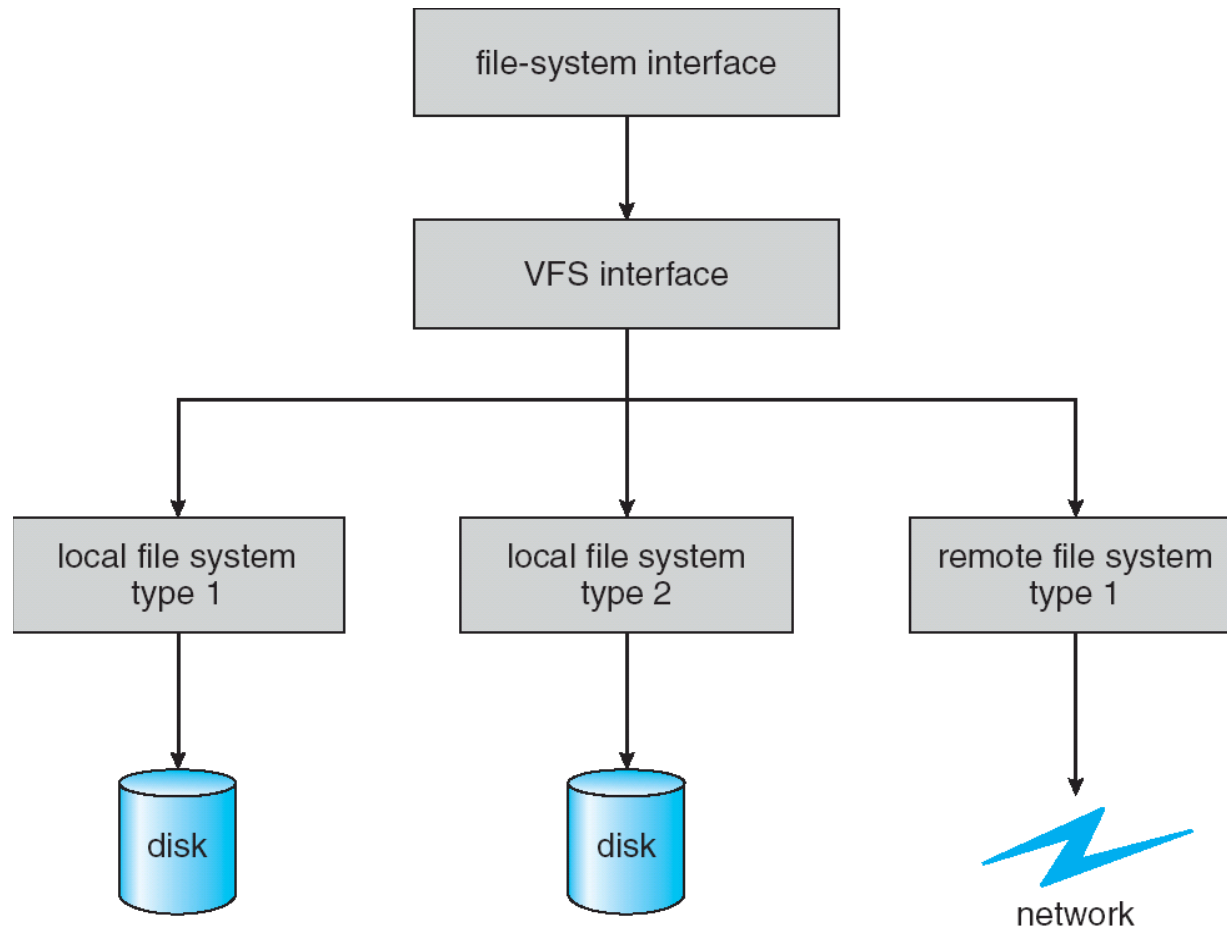


File control block (FCB)

- Resides at the logical FS layer
- Storage structure consisting of information about a file

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

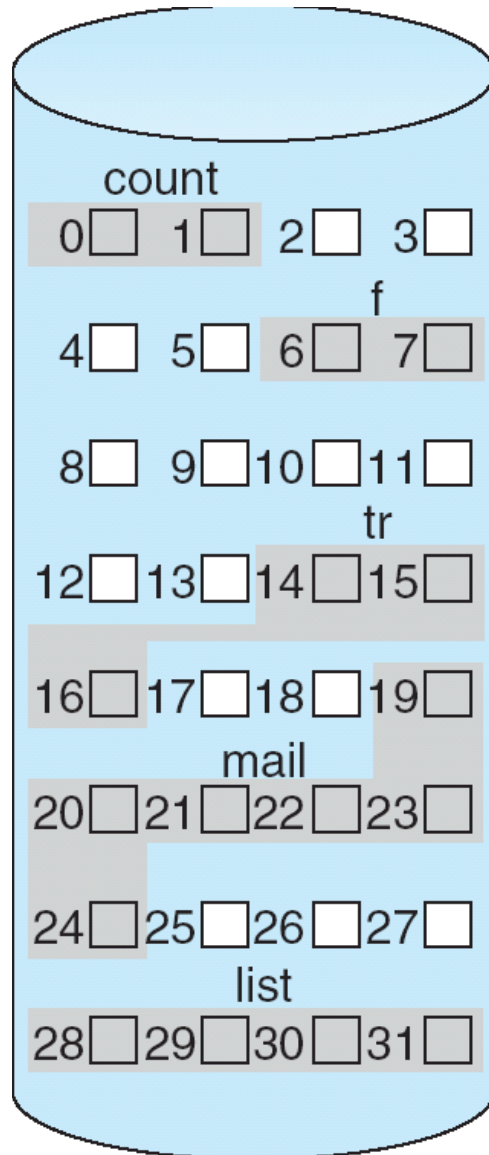
Virtual File System (VFS)



Allocation Methods

- An **allocation method** refers to how disk blocks are allocated for files
 - **Contiguous allocation**
 - **Linked allocation**
 - **Indexed allocation**

Contiguous Allocation

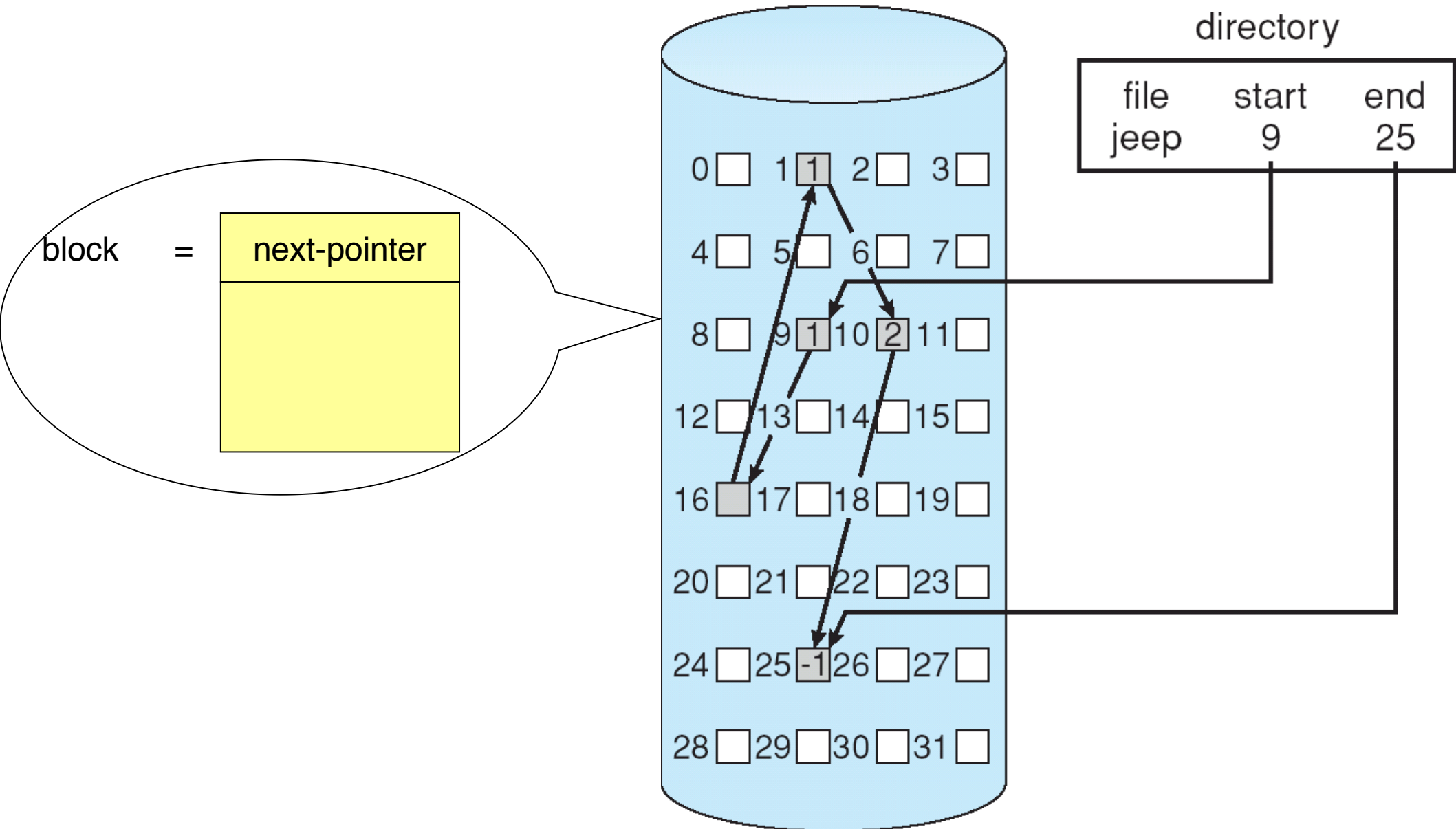


directory		
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation

- Pros:
 - Simple
 - Allows random access
- Cons:
 - Wasteful
 - Files cannot grow easily
- Works well on CD-ROM

Linked Allocation



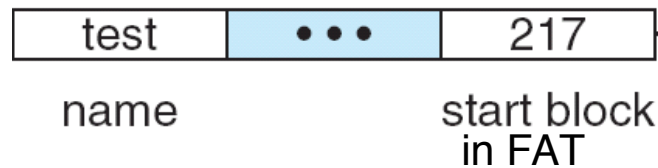
Linked Allocation

- Pros:
 - Simple – need only starting address
 - Free-space management
 - No external fragmentation
- Cons:
 - No random access
 - Overhead (space and time)
 - Reliability

File-Allocation Table (FAT)

File-allocation table (FAT) – disk-space allocation used by older Windows

directory entry



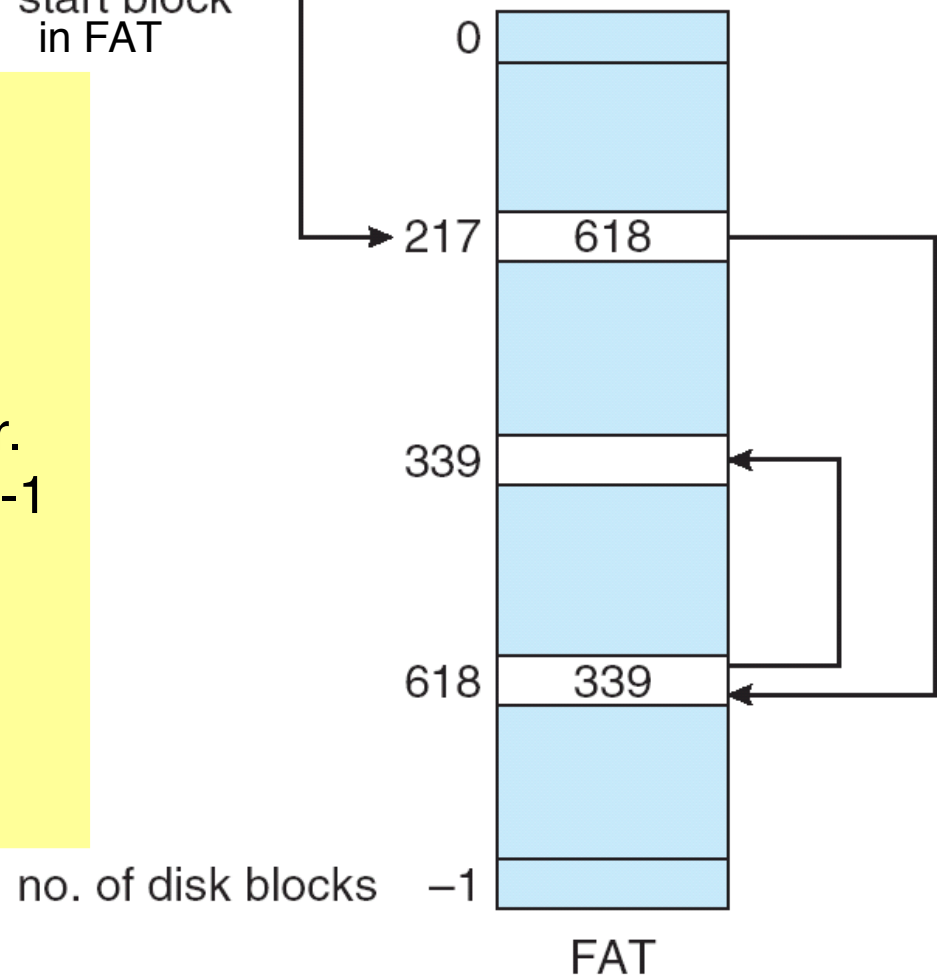
Variant of linked allocation:

FAT resides in reserved section at beginning of each disk volume

One entry for each disk block, indexed by block number, points to successor. Entry for last block in a chain has table value -1

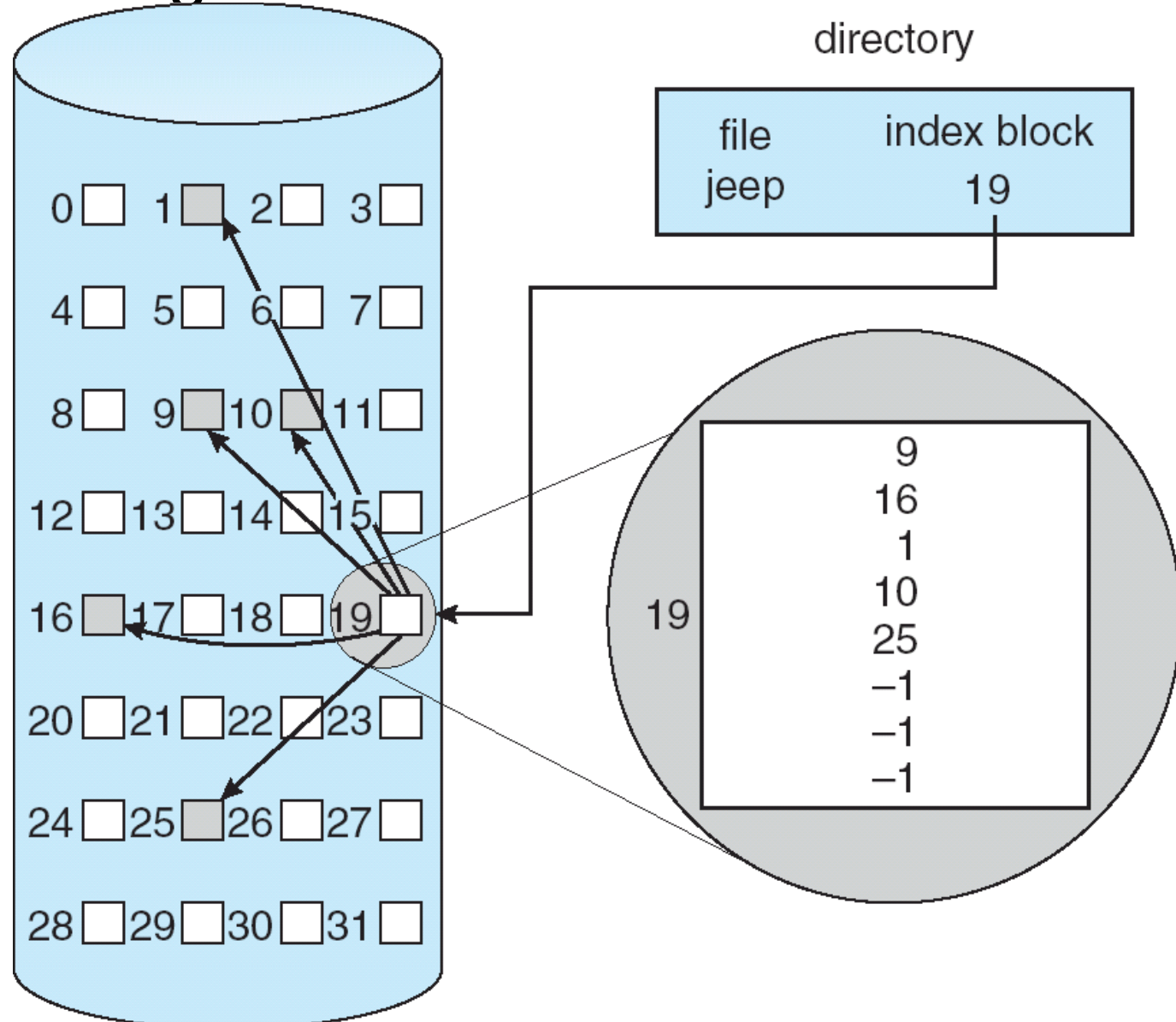
Unused blocks have table value 0
→ Finding free blocks is easy

Does not scale well to large disks or small block sizes



Indexed Allocation

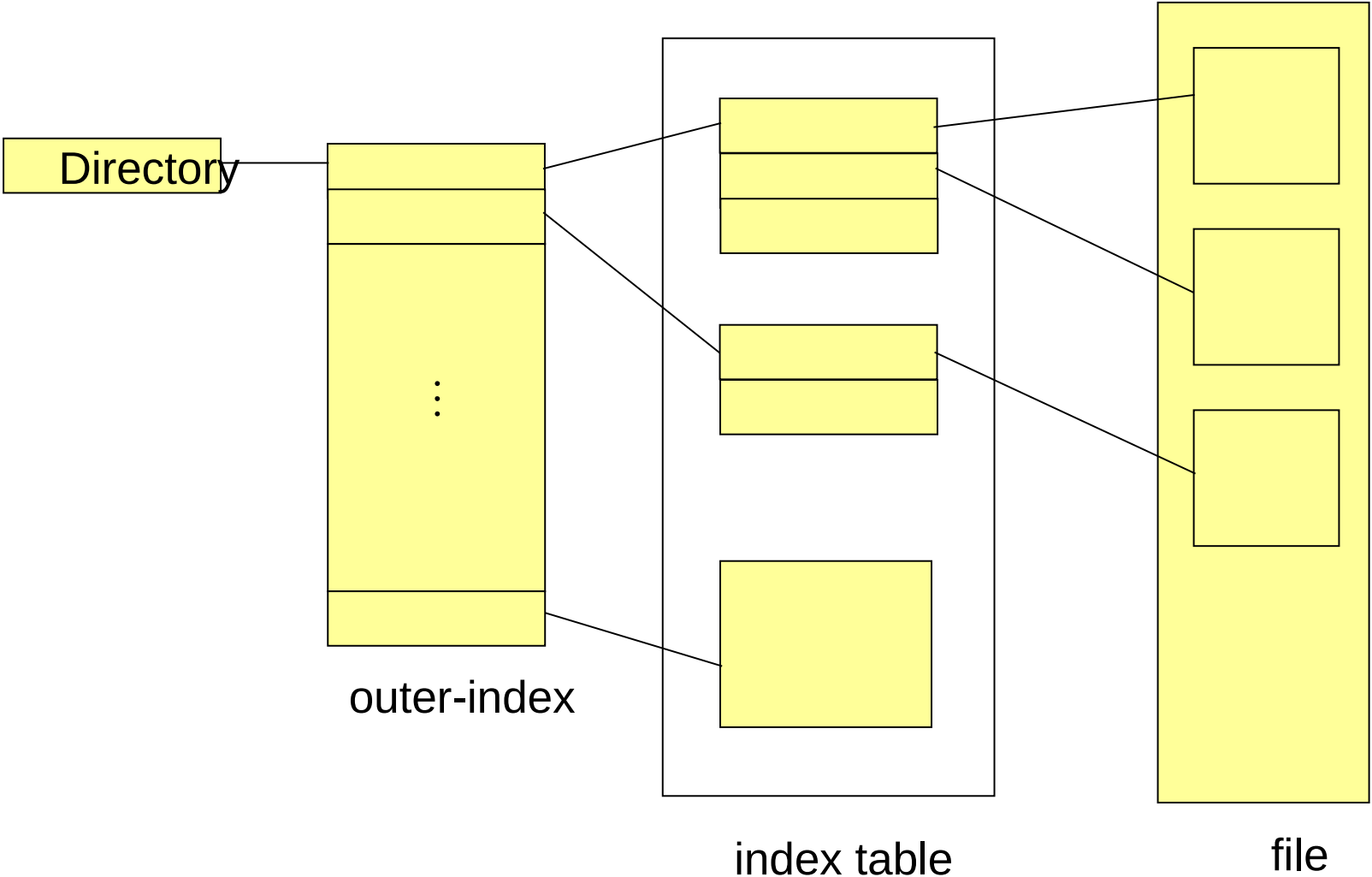
- Brings all pointers together into an *index block*



Indexed Allocation (Cont.)

- Direct access once index block is loaded
 - without external fragmentation,
 - but overhead of index block.
- All block pointers of a file must fit into the index block
 - How large should an index block be?
 - Small – Limits file size
 - Large – Wastes space for small files
 - Solution: Multi-level indexed allocation →

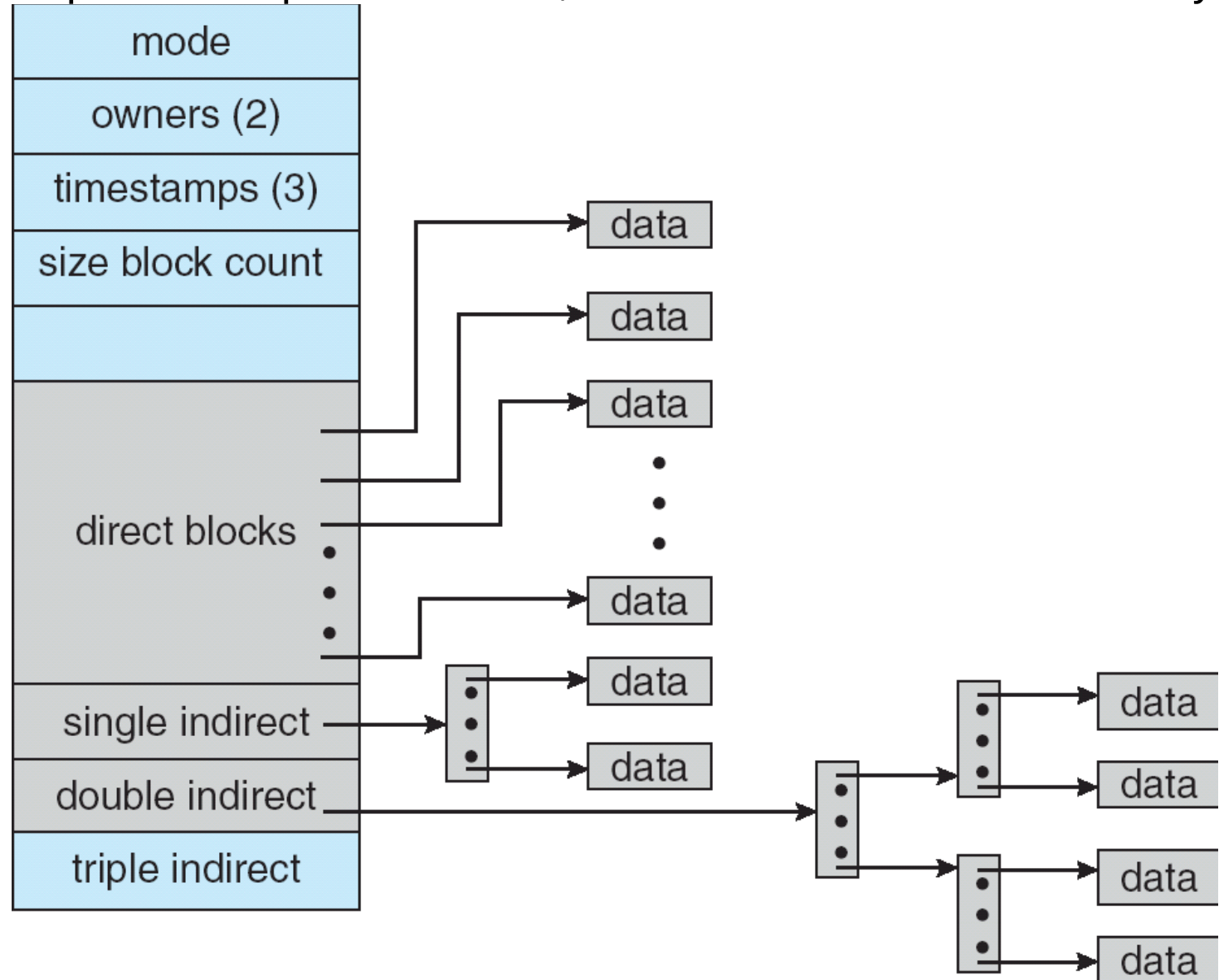
Multilevel-indexed allocation



Combined Scheme: UNIX inode

Block size 4 KB

-> With 12 direct block pointers kept in the inode, 48 KB can be addressed directly.



☺ Small overhead for small files

☺ Still allows large files

B-trees

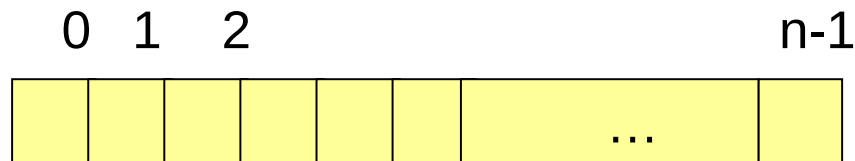
- Self-balancing tree
- Efficient operations $O(\log n)$
- Popular in newer (less old) filesystems
 - NTFS
 - Ext4
 - HFS+

Free-Space Management

- Where is there free space on the disk?
 - A free-space list
- Two basic approaches
 - Free-space map (bit vector)
 - Linked list

Bit vector

- Each block represented by one bit



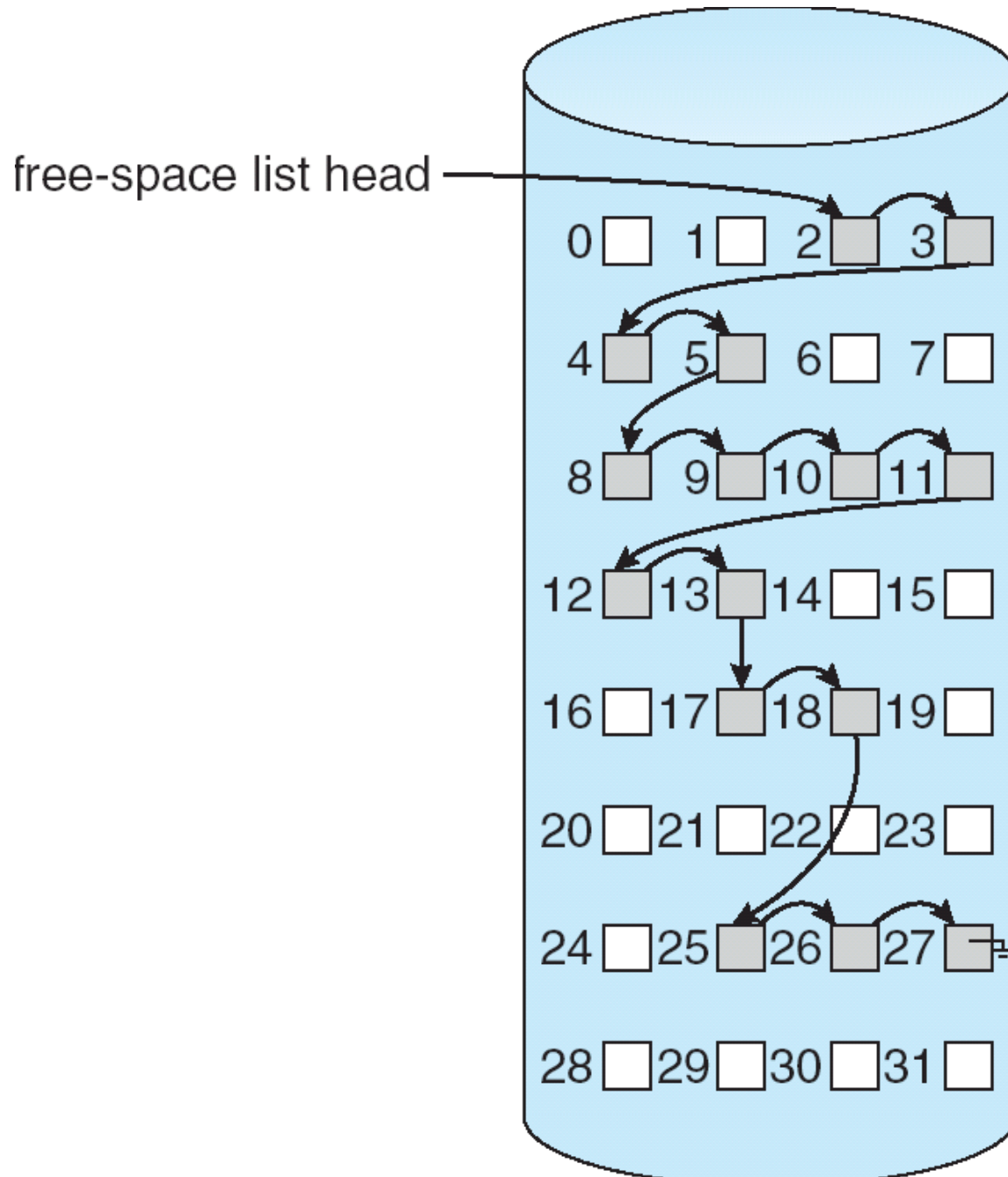
$$\text{bit}[j] = \begin{cases} 1 \Rightarrow \text{block}[j] \text{ free} \\ 0 \Rightarrow \text{block}[j] \text{ occupied} \end{cases}$$

First free block: number of bits per word) * (number of 0-value words) + offset of first 1 bit

Bit vector

- Easy to get contiguous files
- Bit map requires extra space
- Example: block size = 1 KB = 2^{10} bytes
 disk size = 68 GB $\sim 2^{36}$ bytes
 $n = 2^{36} / 2^{10} = 2^{26}$ bits (or 67 MB)
- Inefficient unless entire bit vector is kept in main memory

Linked list



Linked list

- Only need to store the pointer to the first free block
- Finding k free blocks means reading in k blocks from disk
- No waste of space

Counting

- Often, multiple subsequent blocks are allocated/freed together
- For sequences of free blocks located subsequently on disk, keep only reference to first one and length of sequence

Fact #1

File systems contain multiple data structures

Fact #2

These data structures have inter-dependencies

Conclusion:

Modification of the file system should be atomic

What happens if the computer is suddenly turned off?

File system repair

- For each block
 - Find which files use the block
 - Check if the block is marked as free
- The block is used by 1 file xor is free – OK
- Two files use the same block – BAD: duplicate the block and give one to each file
- The block is both used and is marked free – BAD: remove from free list
- The block is neither free nor used – Wasted block: mark as free

Modern alternatives

- Journalling
- Snapshot-based
- Log-structured

Journaling file systems

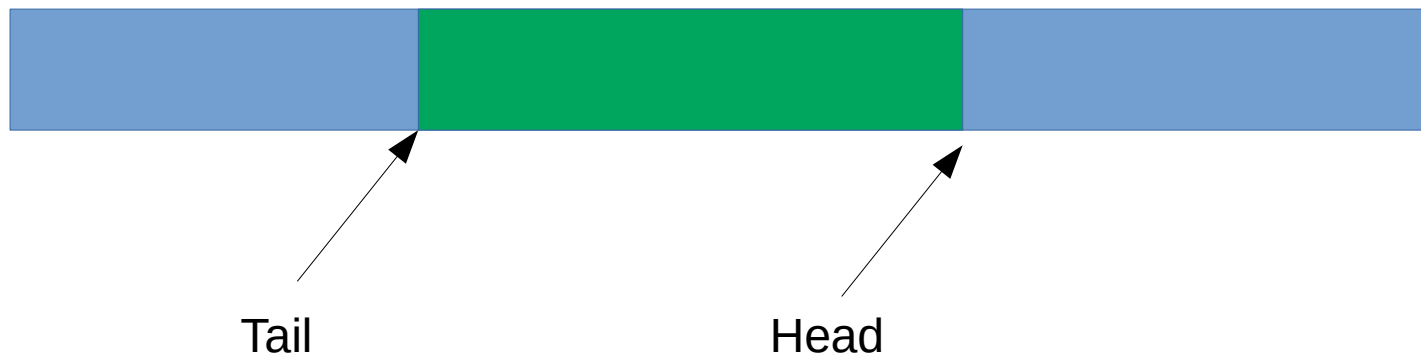
- Each modification is made as a transaction
- Keep a journal (log) of all pending transactions
- Interrupted transaction can be rolled-back
- Every update requires two writes
- Examples: NTFS, ext4

Snapshot-based

- Copy-on-write
- Often combined with checksums
- Atomicity an result in cascading updates
- Examples: ZFS, Btrfs, APFS

Log-structured

- Idea from 88 - exploit sequential nature of disks
- Skip the structure just write where free
- Now revived!
 - Per inode logs in NOVA filesystem for non-volatile main memory (NVMM)



Memory-Mapped Files

- **Mapping** a disk block to a page in memory
- A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read()** / **write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory-Mapped Files

