TDDB68 Concurrent Programming and Operating Systems

Lecture 8:
Memory management II + File systems II

Klas Arvidsson
with slides inherited from
Adrian Pop and Mikael Asplund
PELAB / RTSLAB
Department of Computer and Information Science

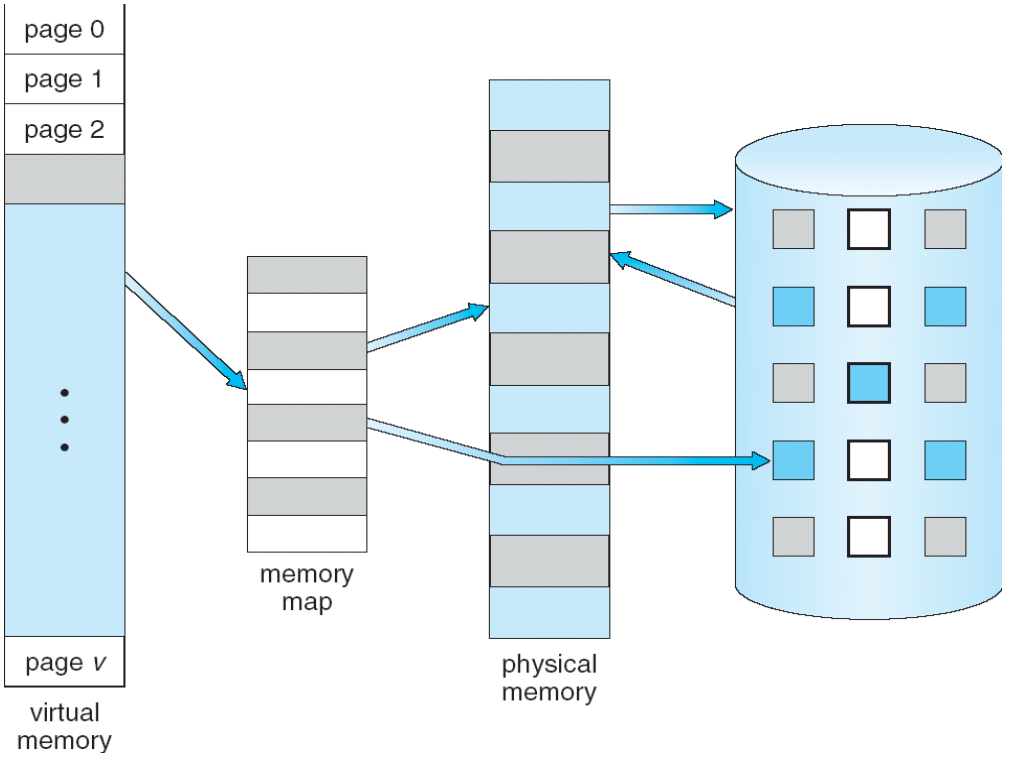
Reading

- 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

[Kilburn *et al.* 1961]

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed if referenced (load/store, data/instructions)
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

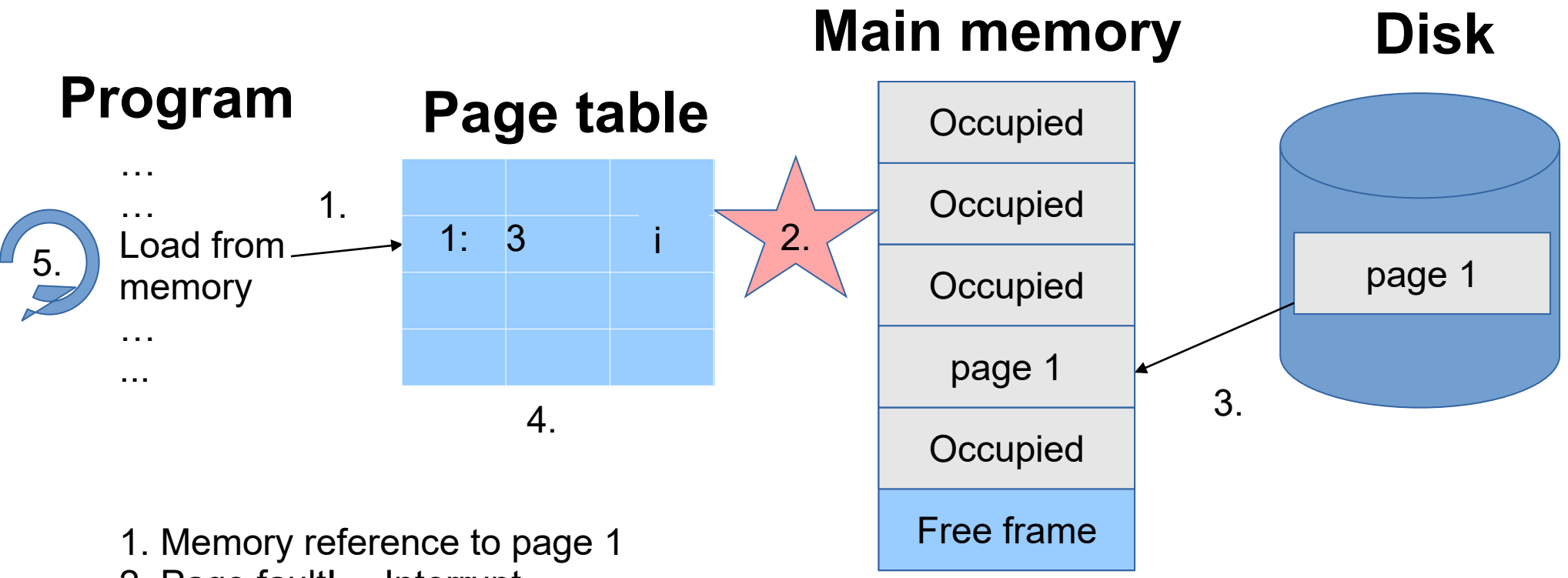


What happens if there is no free frame?

Page Replacement

Steps in Handling a Page Fault

(Case: a free frame exists)



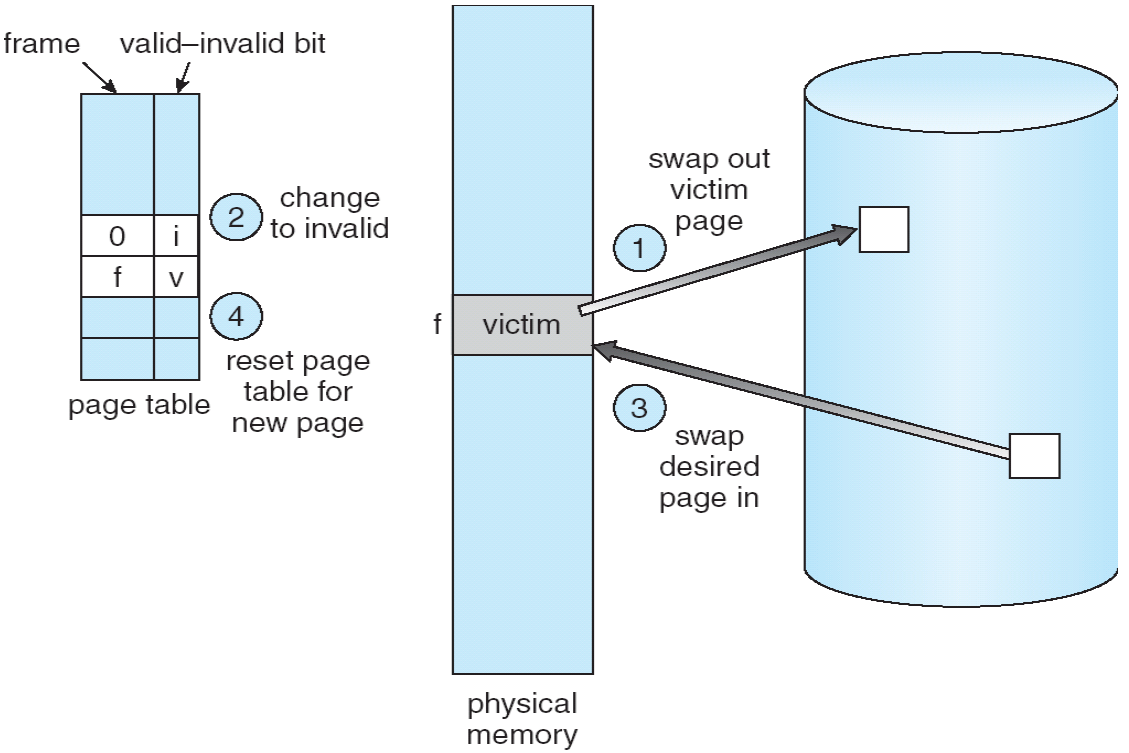
1. Memory reference to page 1
2. Page fault! →Interrupt
3. OS moves page into memory
4. Update page table
5. Restart memory access instruction

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.

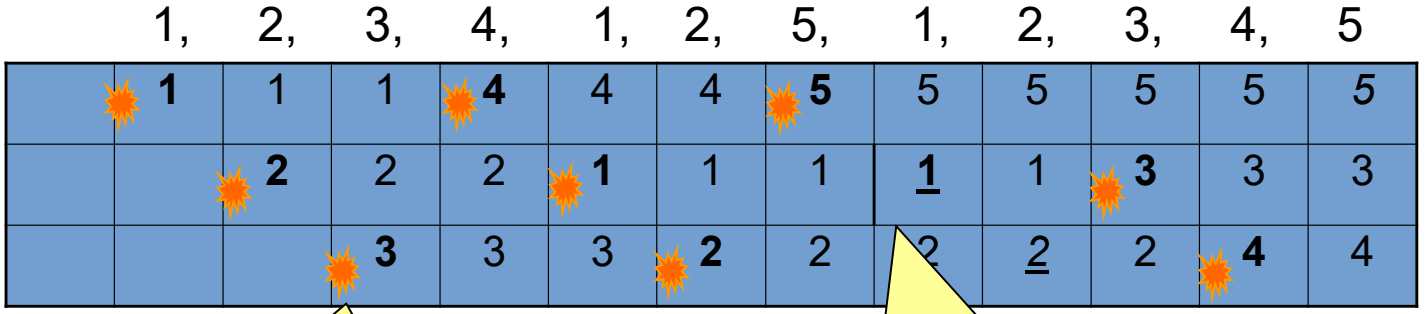
- In all our examples, the reference string is

  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:

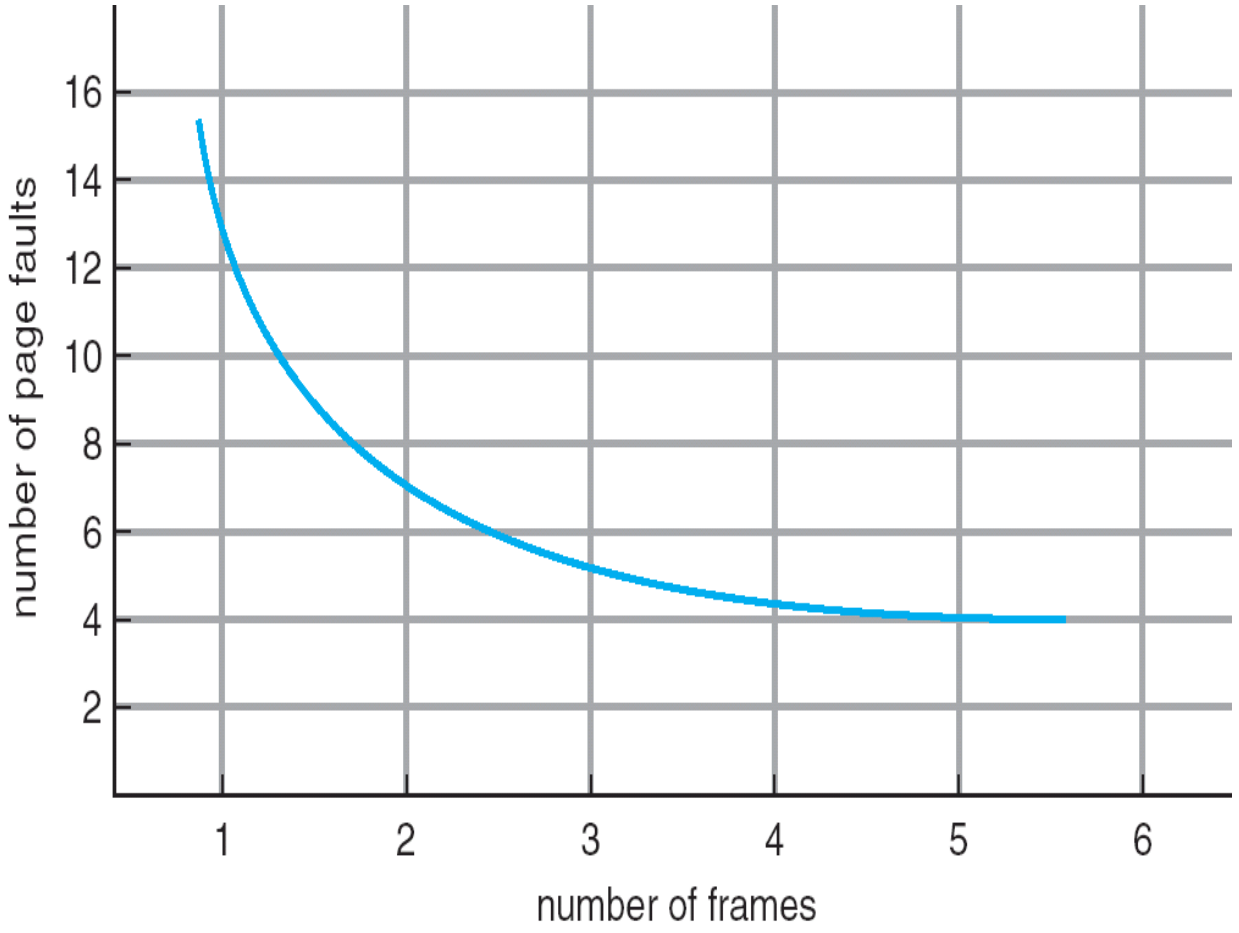


A total of 9 page faults

After page 3 is loaded, page 1 is the oldest of them all

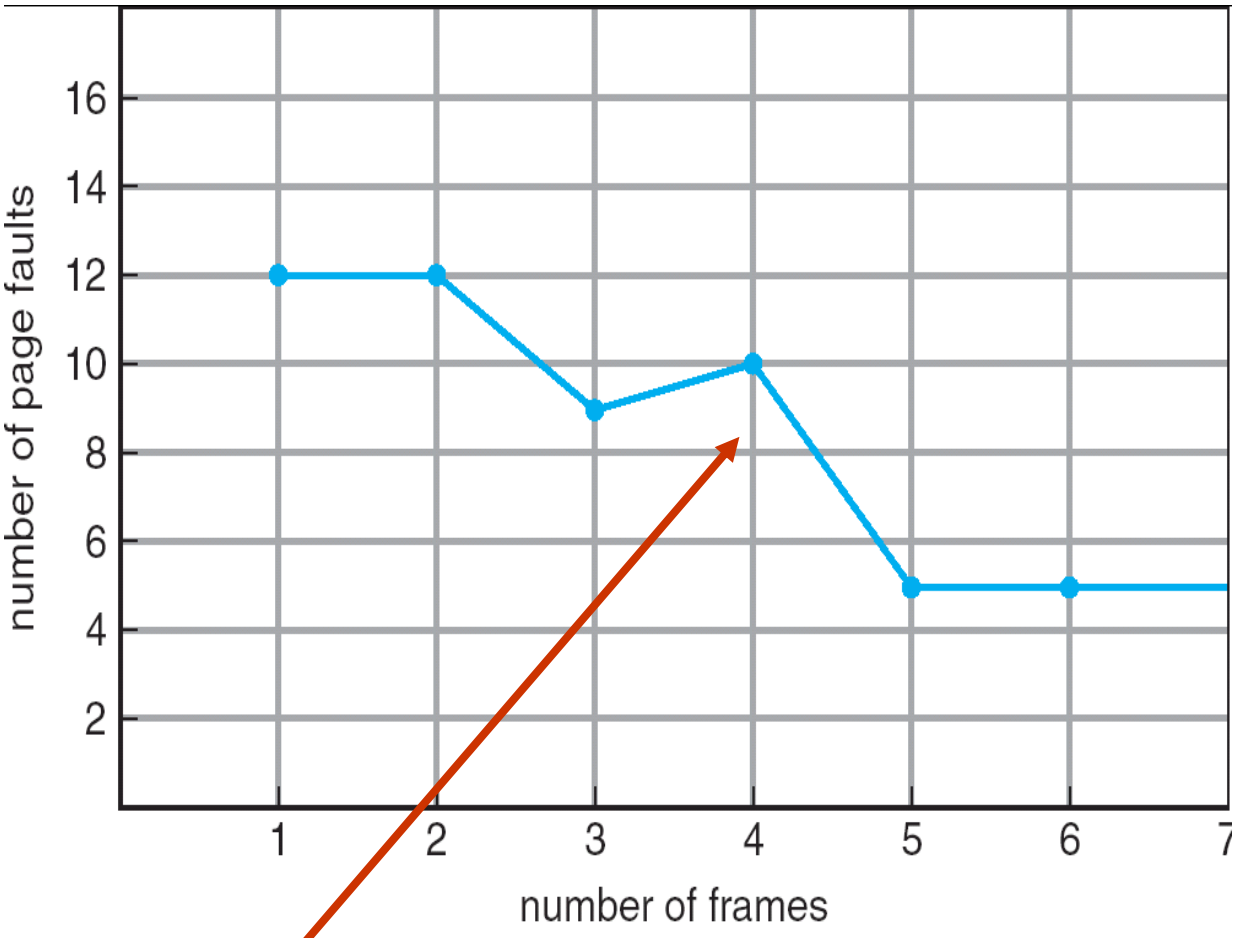
The fact that we re-use an existing page does not alter who has been in there the longest...

Expected Page Faults Versus Number of Frames



Generally, more frames => Less page faults **?**

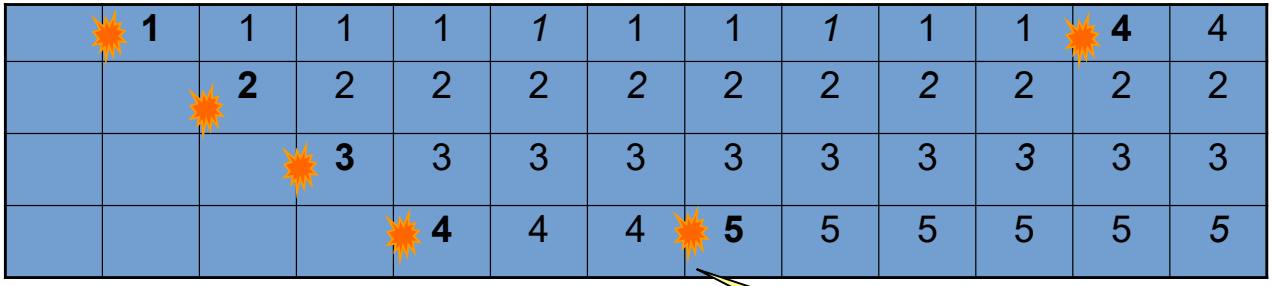
FIFO illustrating Belady's Anomaly



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



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.

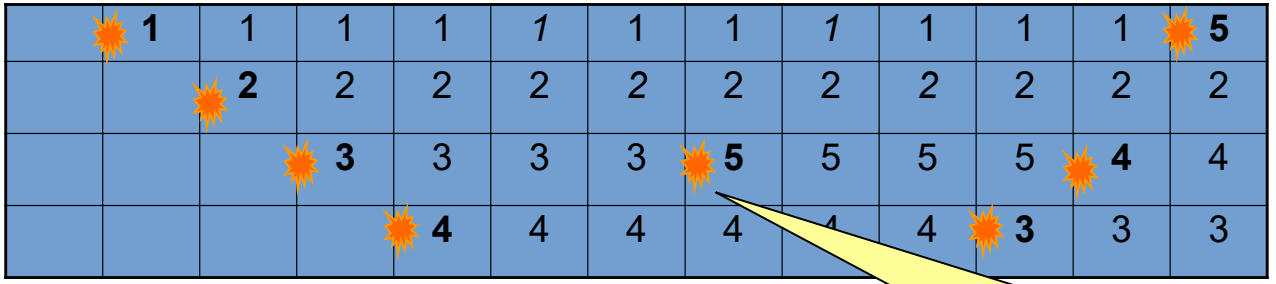
We will need frames 1, 2 and 3 before we need frame 4 again, thus throw it out!

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



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...

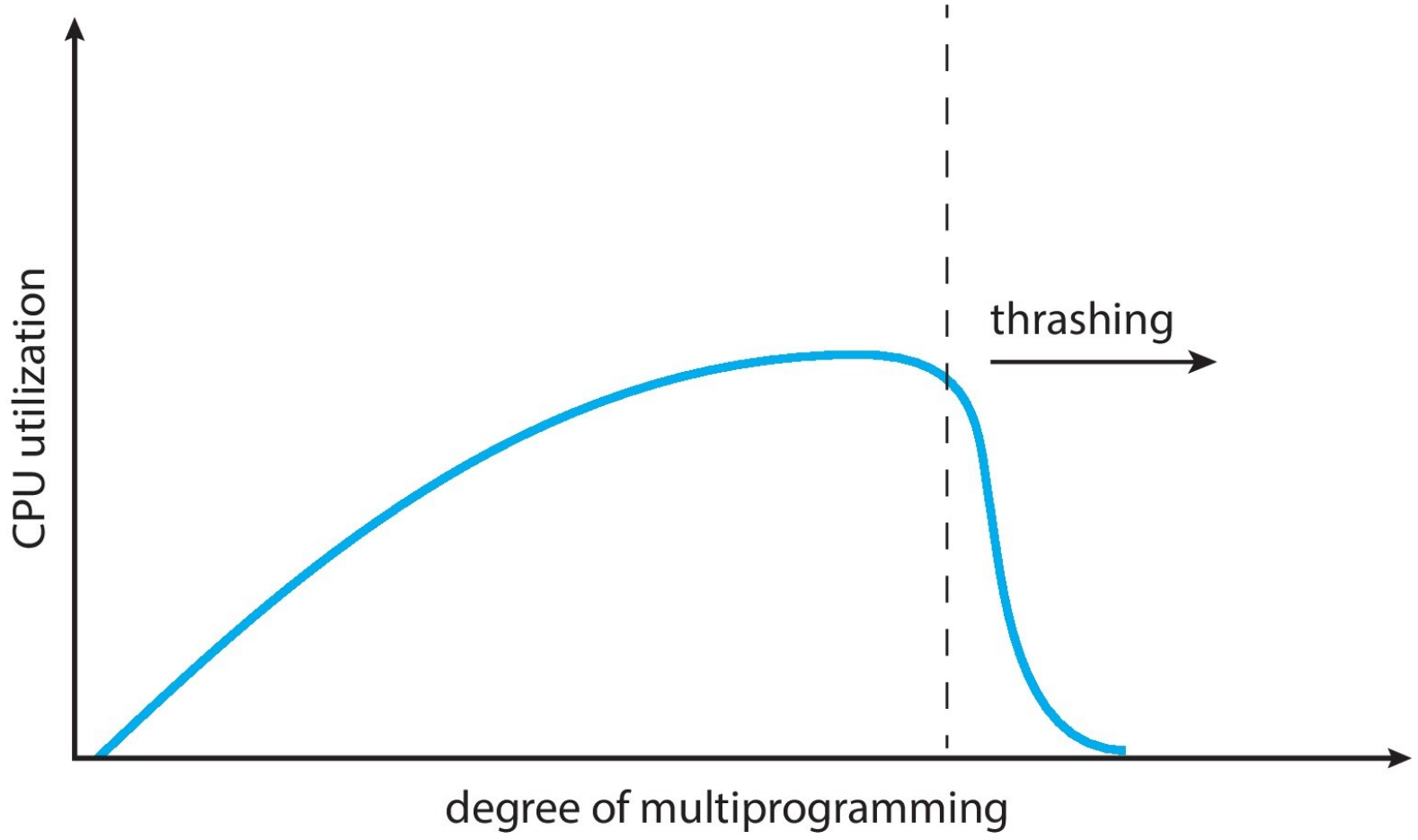
Thrashing

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

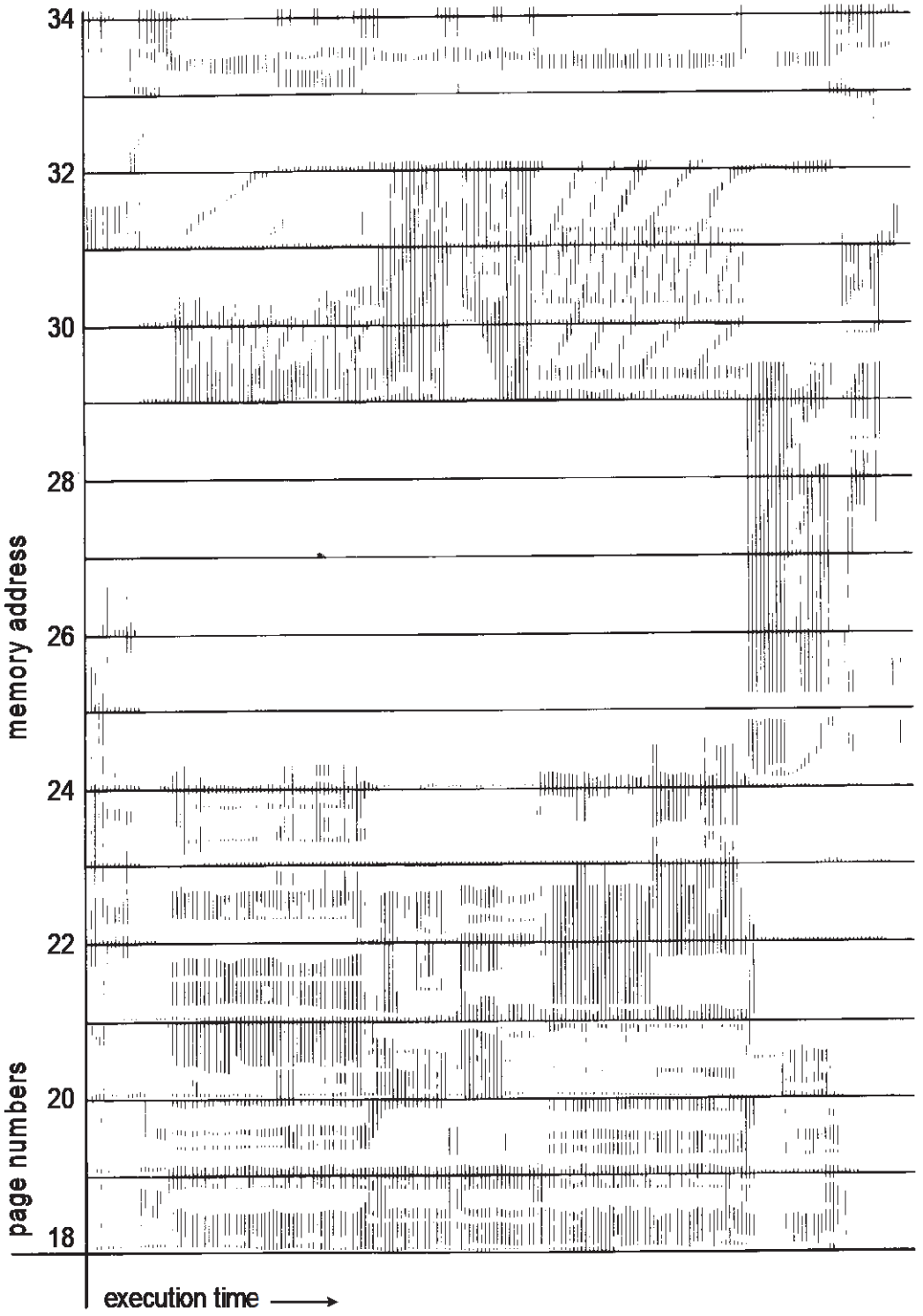
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?

  Σ size of locality > total memory size

- Limit effects by using local or priority page replacement (instead of global)

Locality In A Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ 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

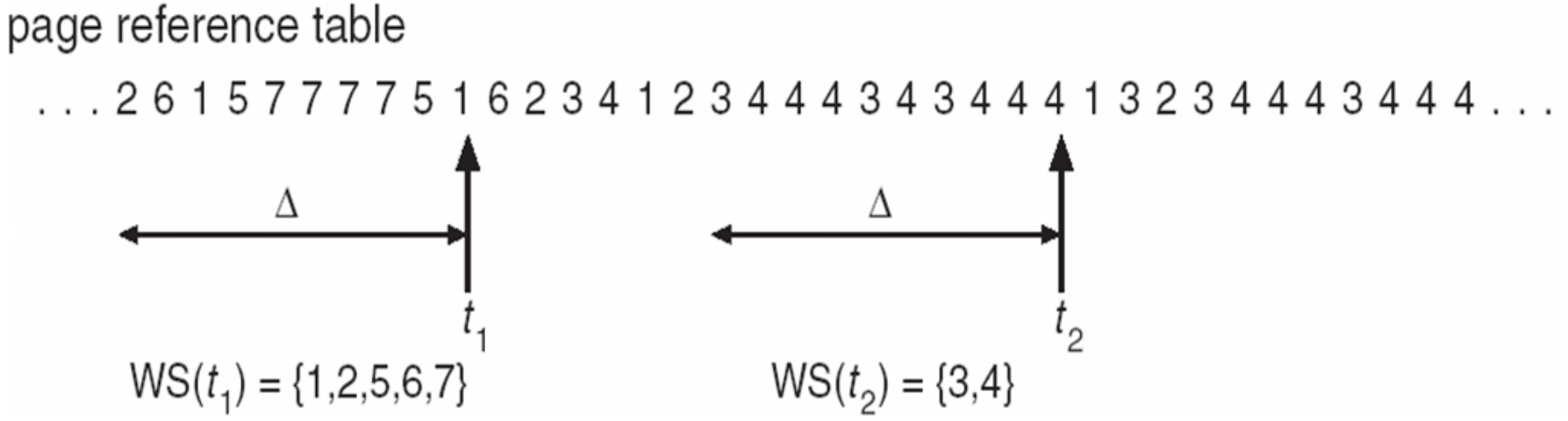
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

- if $D > m \Rightarrow$ Thrashing

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

Working-Set - Example

- WS at two different time points t_1 and t_2



Keeping Track of the Working Set

- Approximate with interval timer + a reference bit

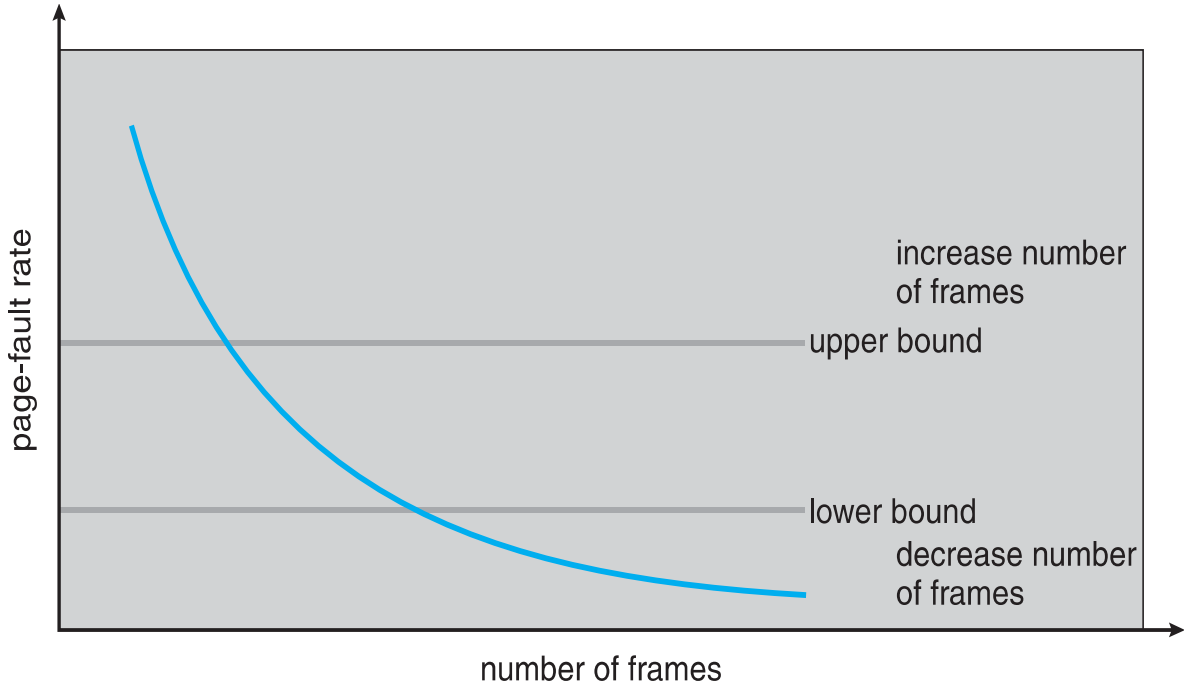
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 ws bits for each page
  - Whenever a timer interrupts shift down ws bits, copy ref bit to high ws bit, set ref bit to 0
  - Set ref bit to 1 when page accessed
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set

- Why is this not completely accurate?

- Improvement = 10 bits and interrupt every 1000 time units

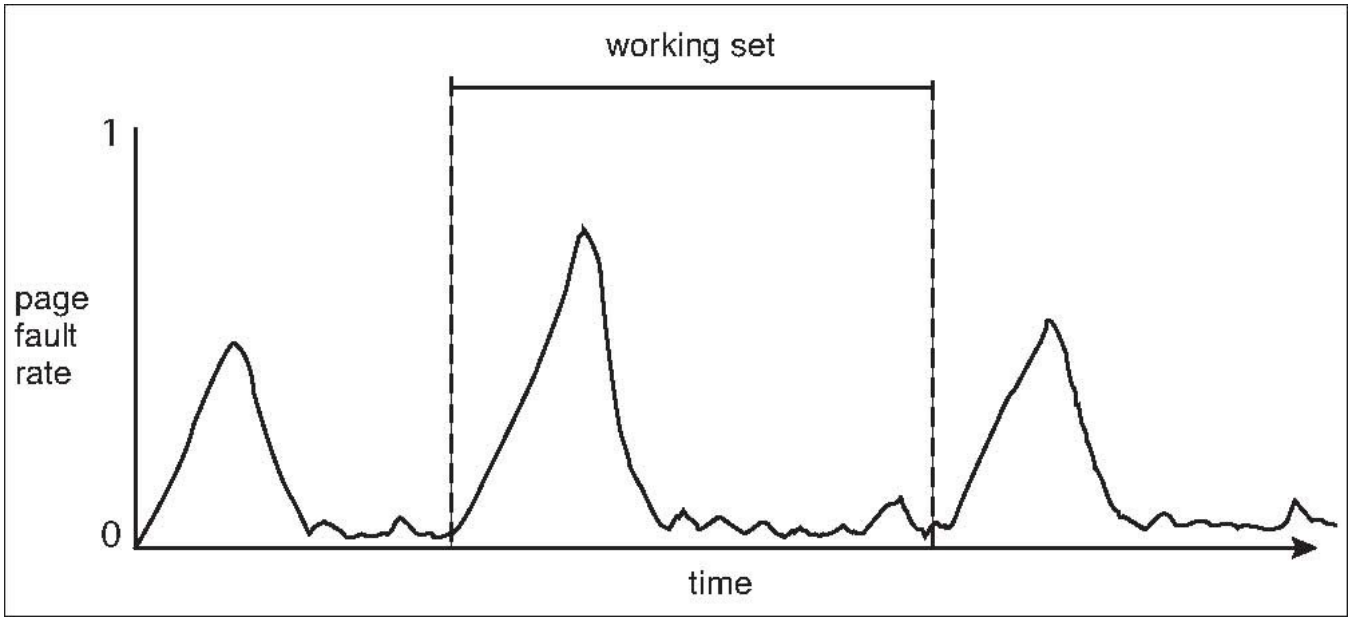
Page-Fault Frequency

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



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

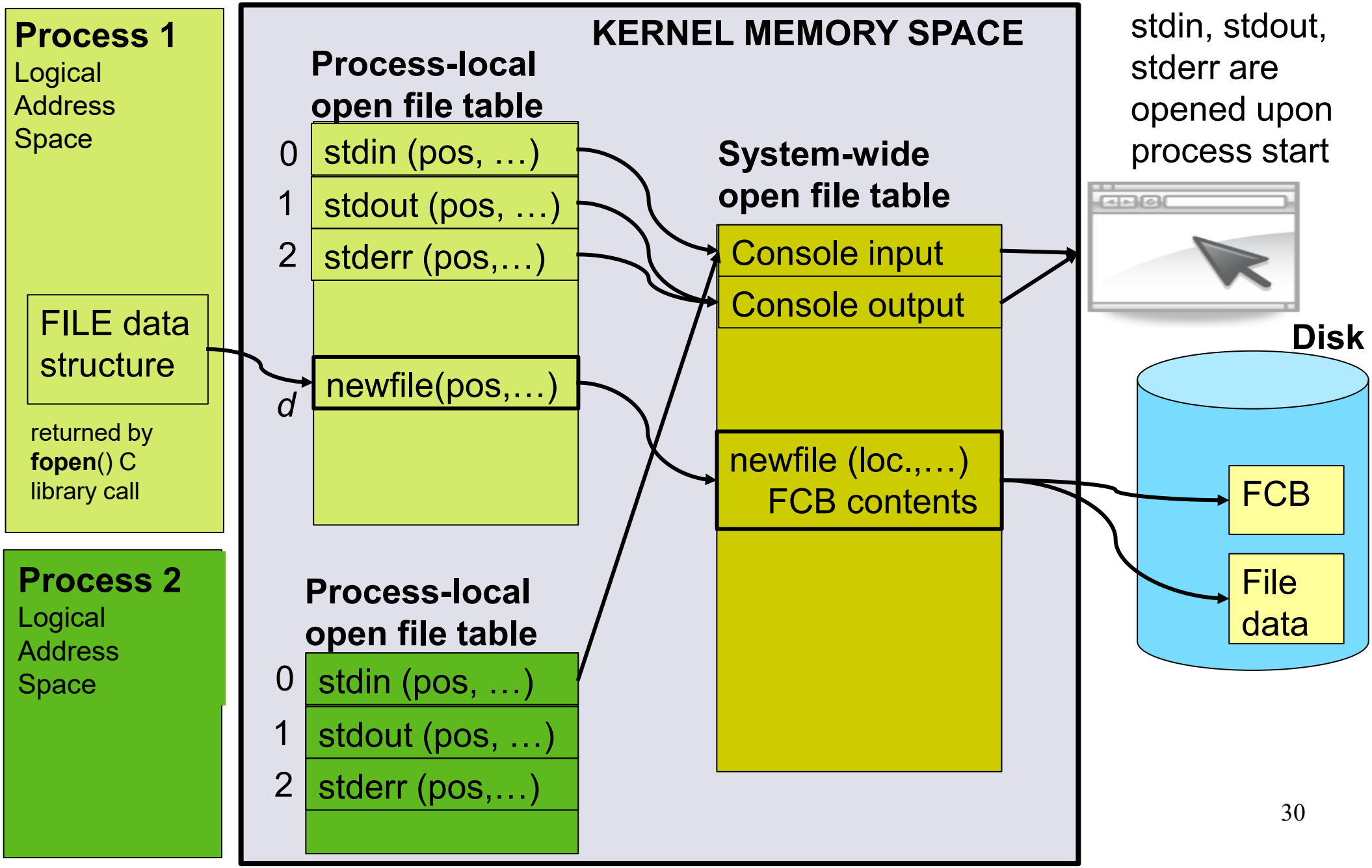
- Compress memory in RAM
- Possible to do in HW, but mostly done by OS
- Examples:
  - Linux: zram, zswap
  - Windows 10
  - Mac OS X
  - Android & iOS

File systems (II)

Refresh

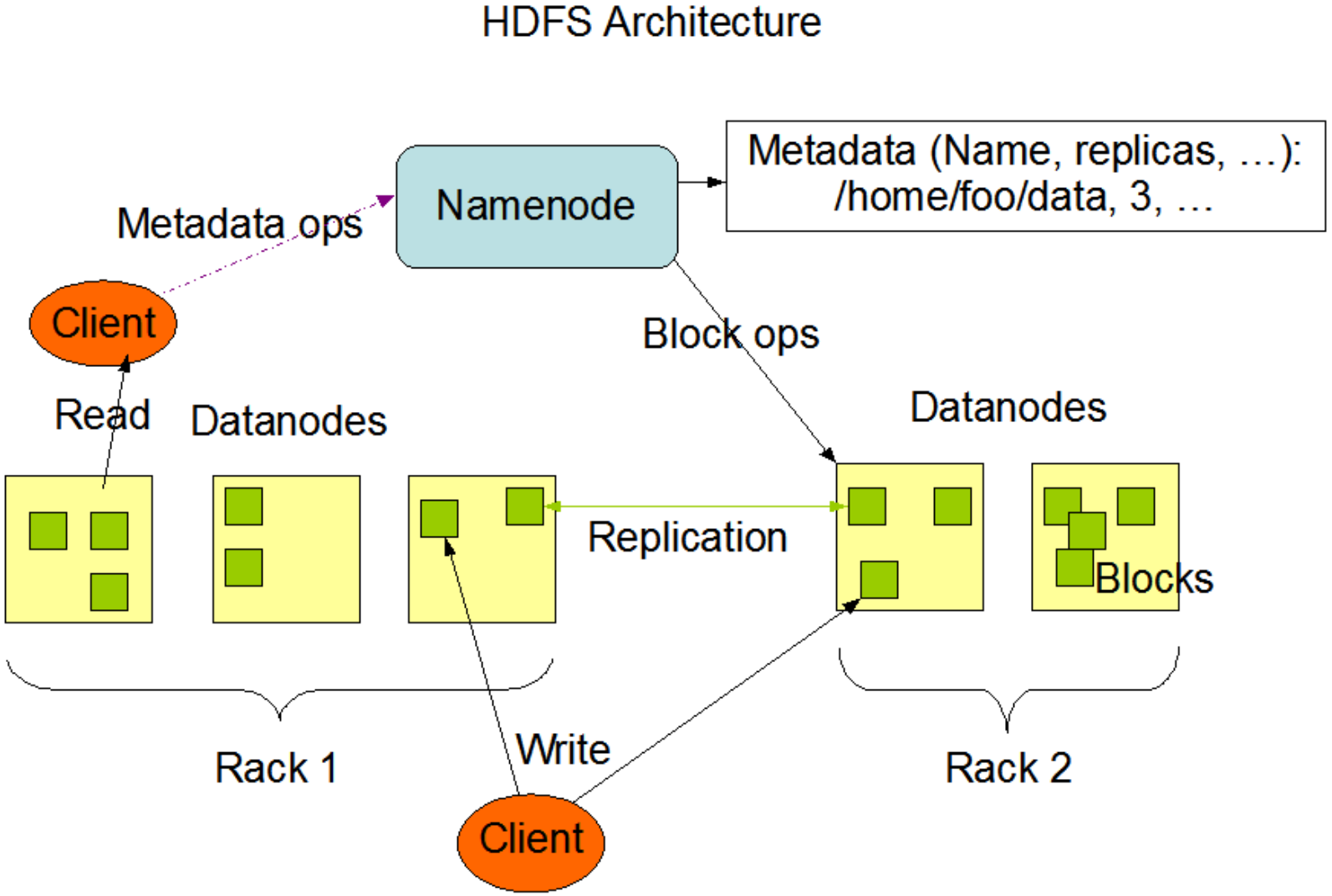
- File systems
  - Interface + Implementation
- Disk – linear sequence of numbered blocks
  - Write block b
  - Read block b
- File abstraction allows to structure data
  - Contiguous logical address space (vs Physical address space)
  - Directory structures
  - Attributes
  - API (create, open, close)
  - FCB for each file (metadata about the file)

File descriptors and open file tables



Specialized file systems

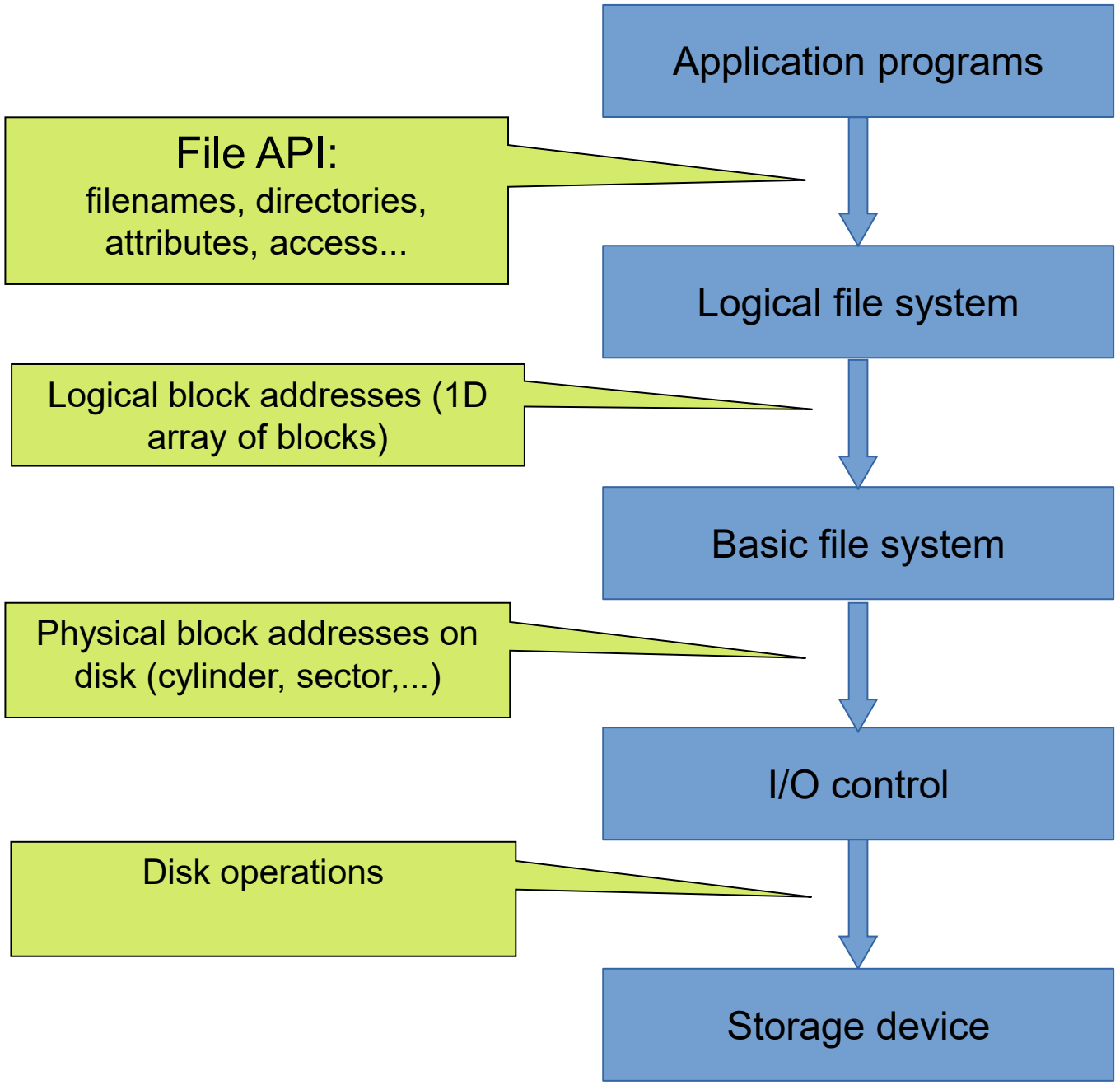
Hadoop Distributed File System (HDFS)



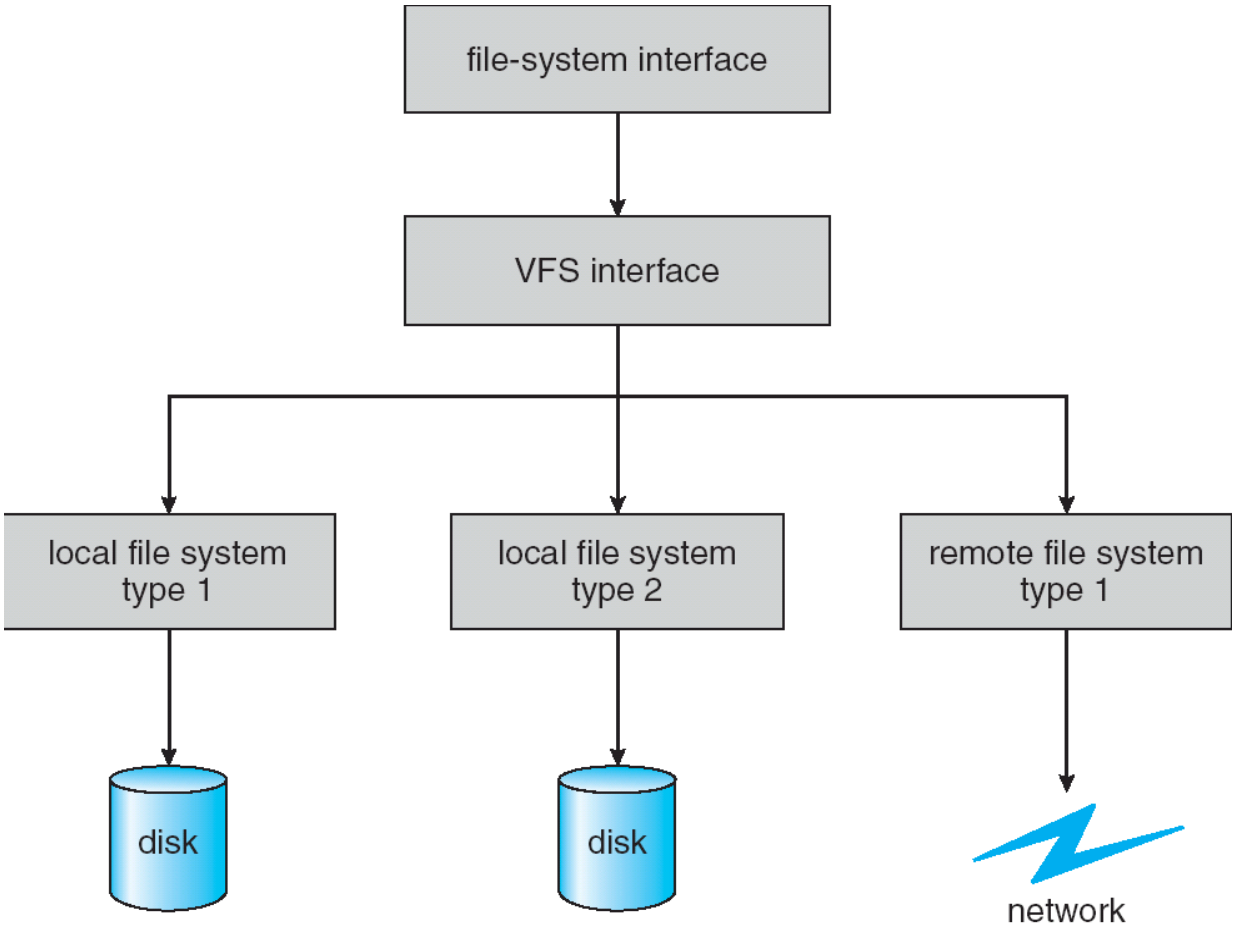
https://cwiki.apache.org/confluence/display/hadoop2/PoweredBy
https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

File system implementation

File-System Layers

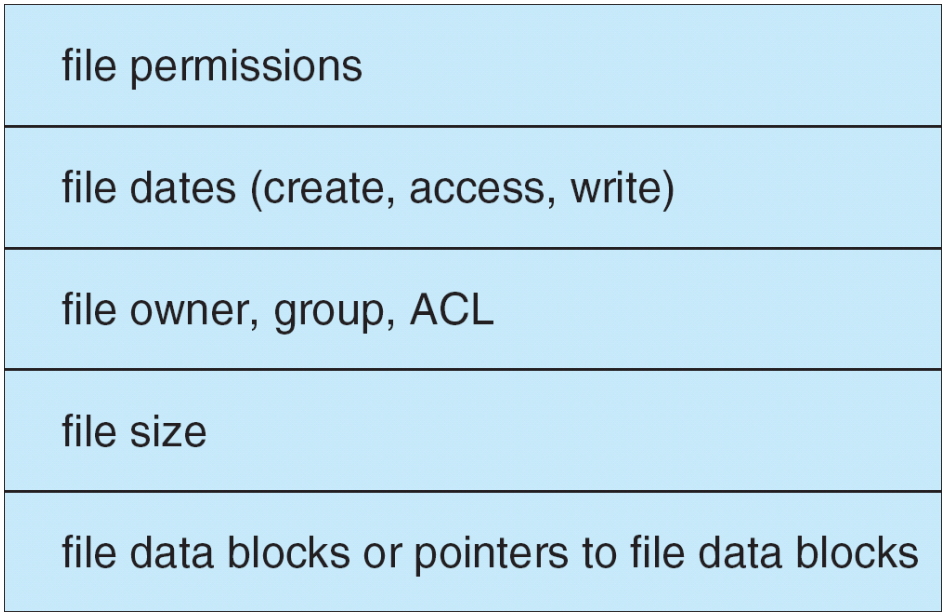


Virtual File System (VFS)



File control block (FCB)

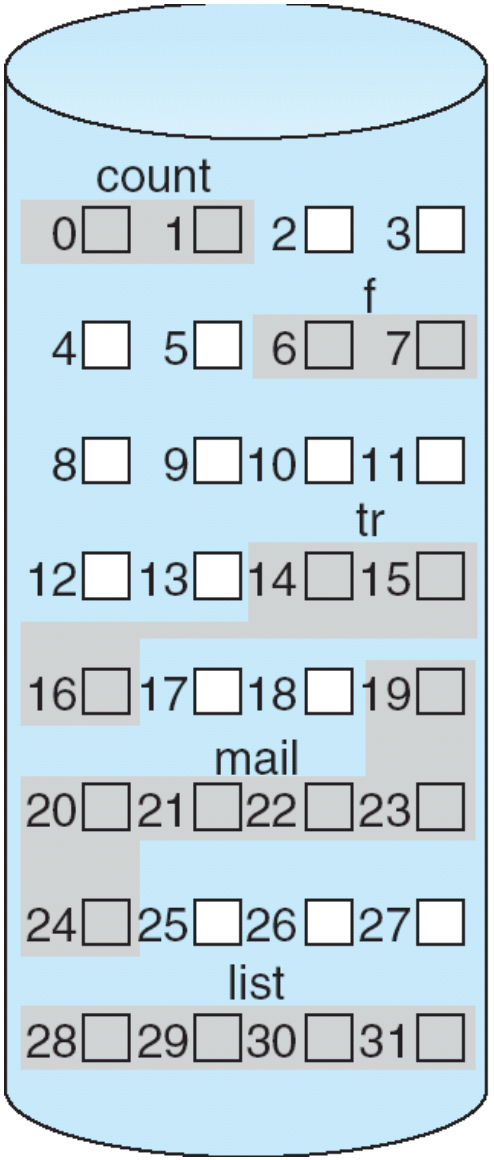
- Resides at the logical FS layer
- Storage structure consisting of information about a file



Allocation Methods

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

Contiguous Allocation



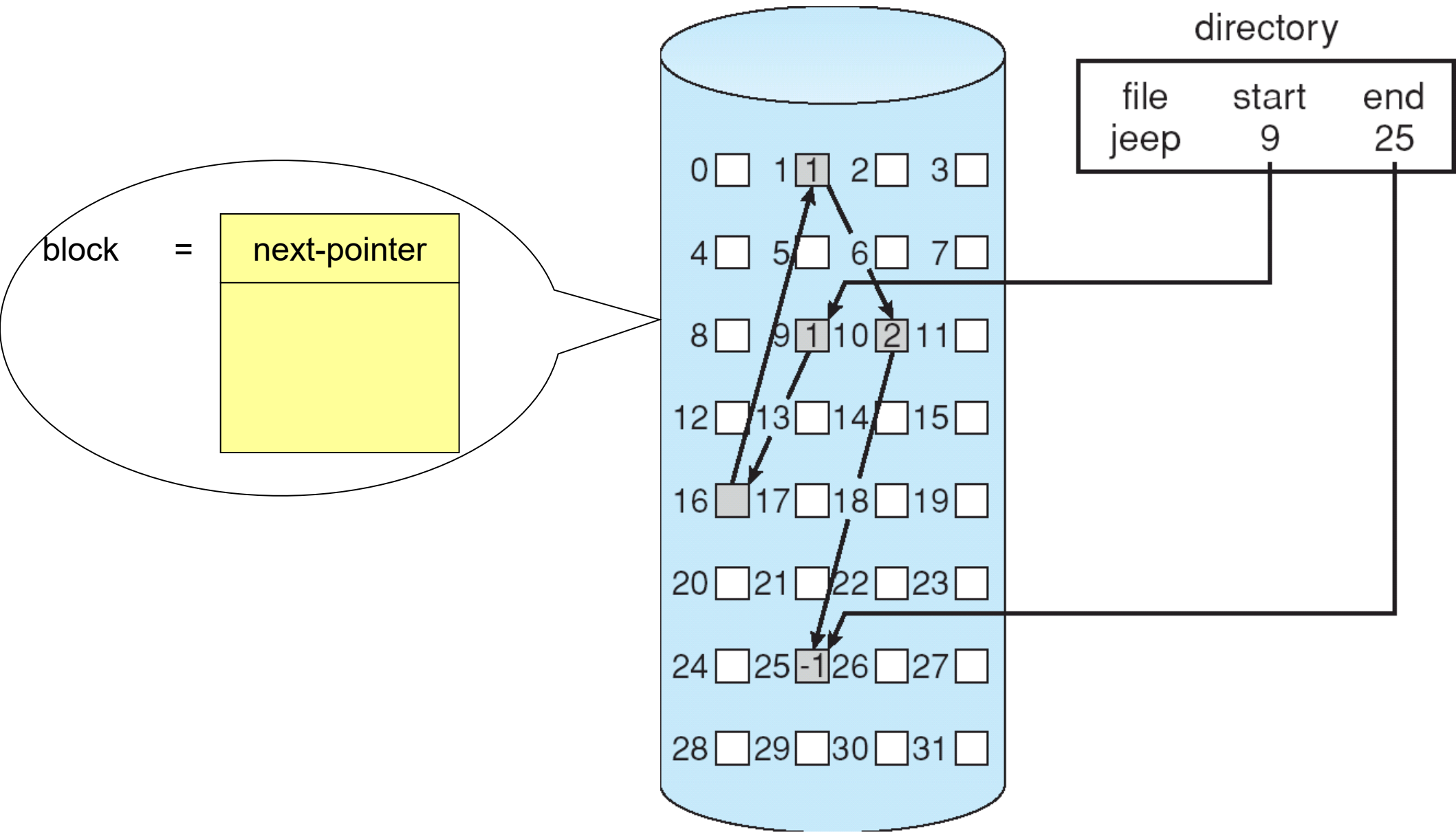
directory

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 read-only media

Linked Allocation

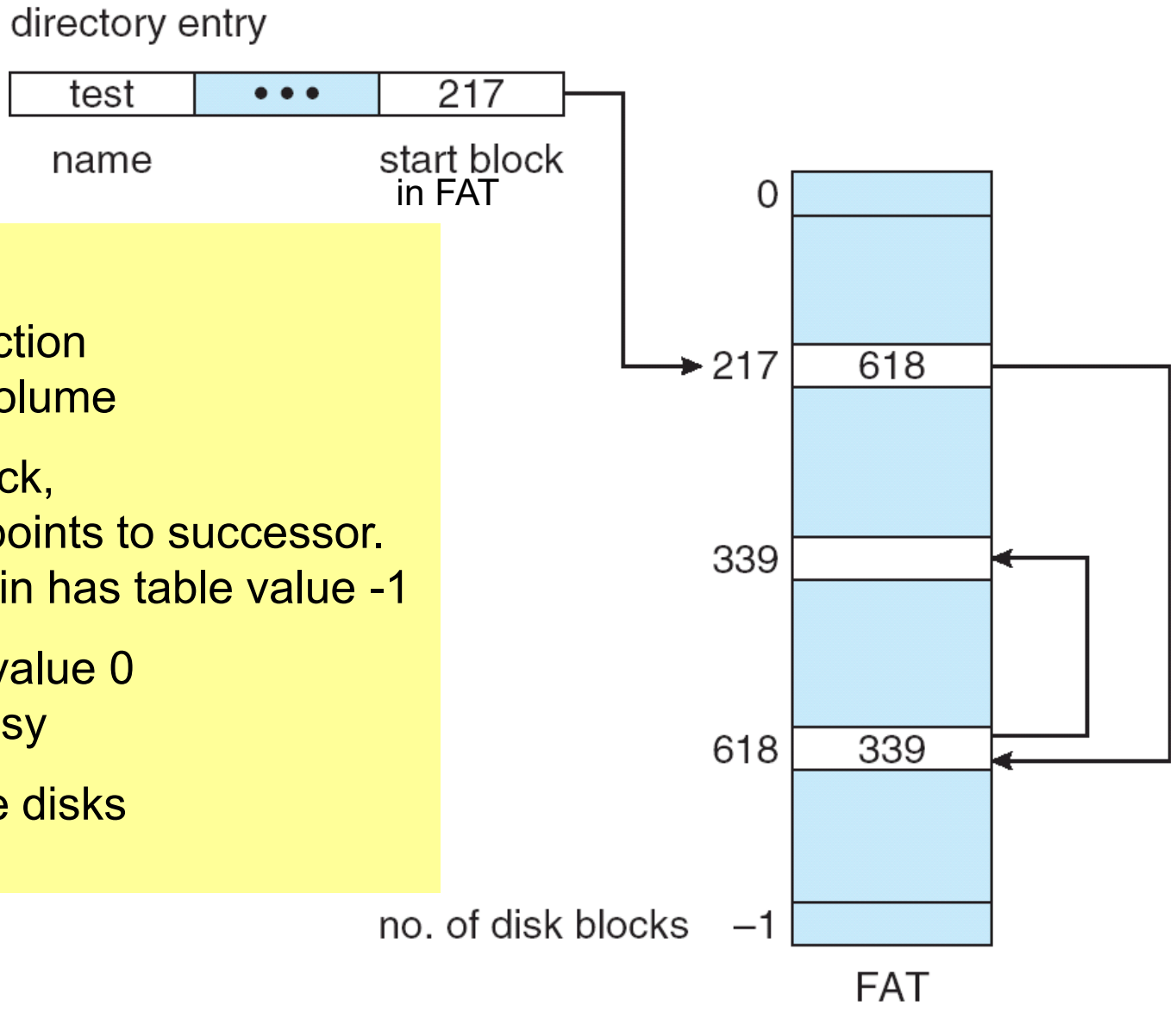


Linked Allocation

- 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



Variant of linked allocation:

FAT resides in reserved section
at beginning of each disk volume

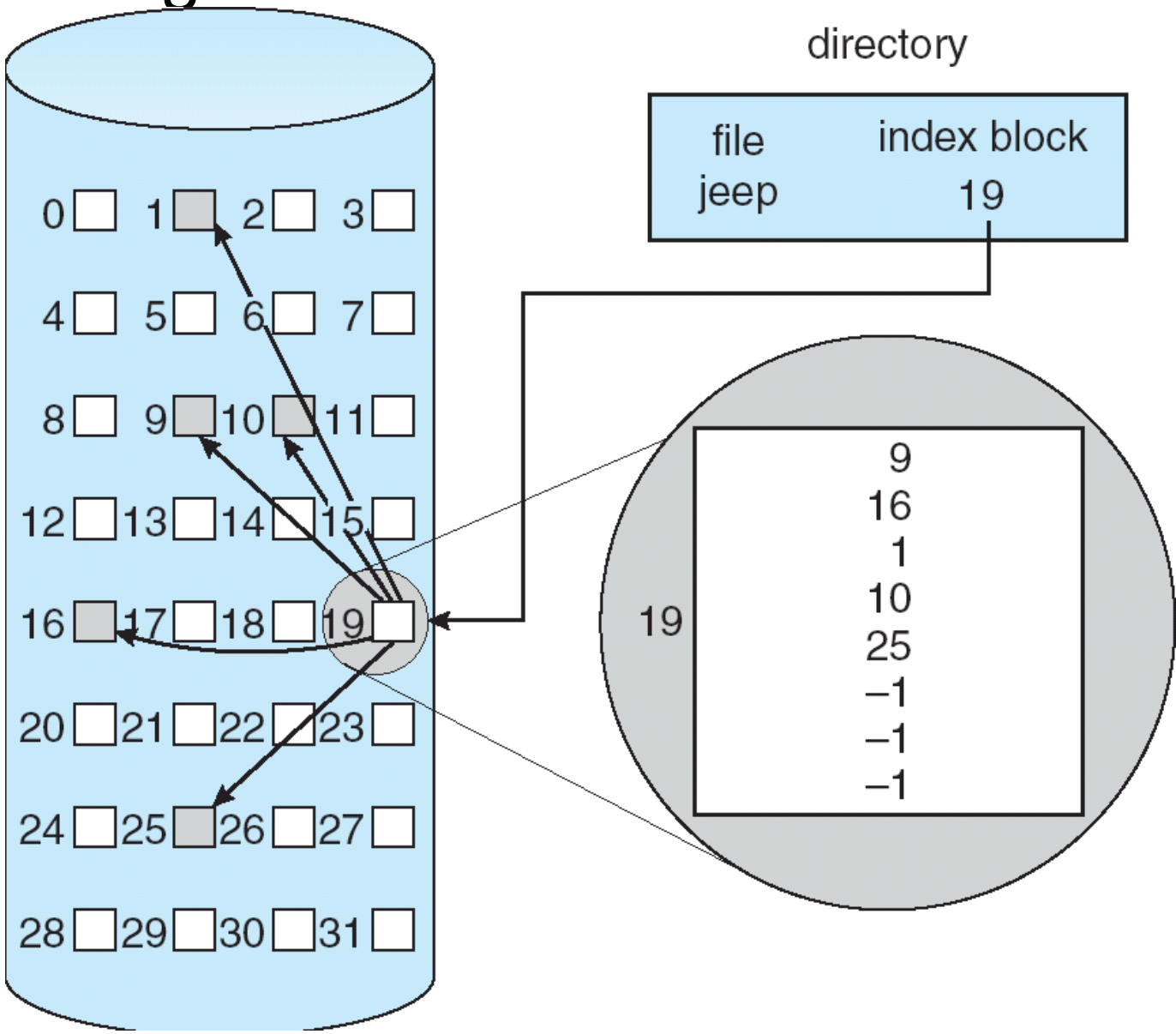
One entry for each disk block,
indexed by block number, points to successor.
Entry for last block in a chain has table value -1

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***

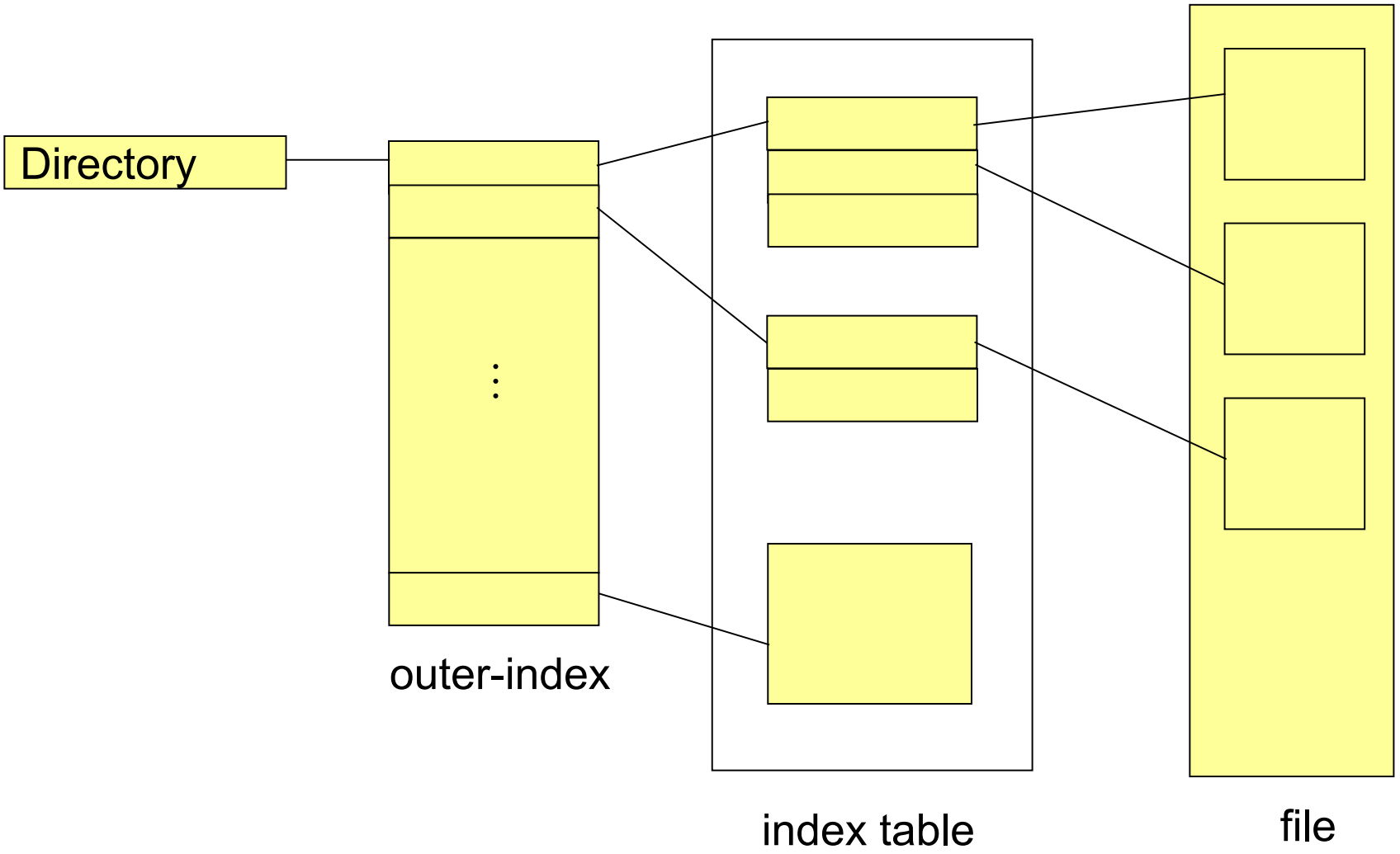


Indexed Allocation (Cont.)

- 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 →

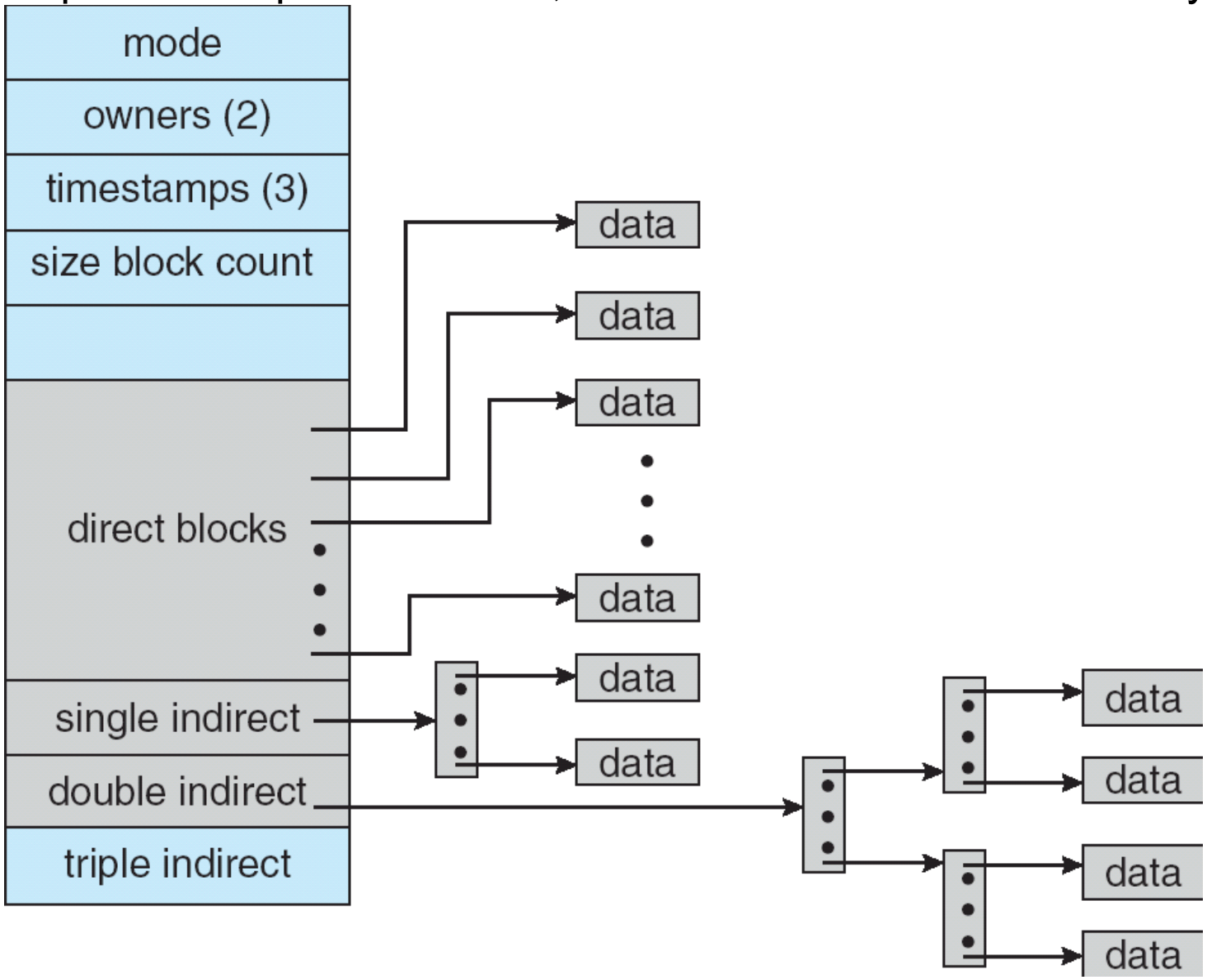
Multilevel-indexed allocation



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

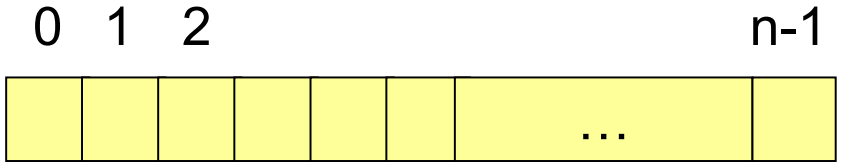
- Self-balancing tree
- Efficient operations O(log n)
- Popular in newer (less old) filesystems
  - NTFS
  - Ext4
  - HFS+

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



bit[*i*] = 1 ⇒ block[*i*] free
0 ⇒ block[*i*] occupied

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

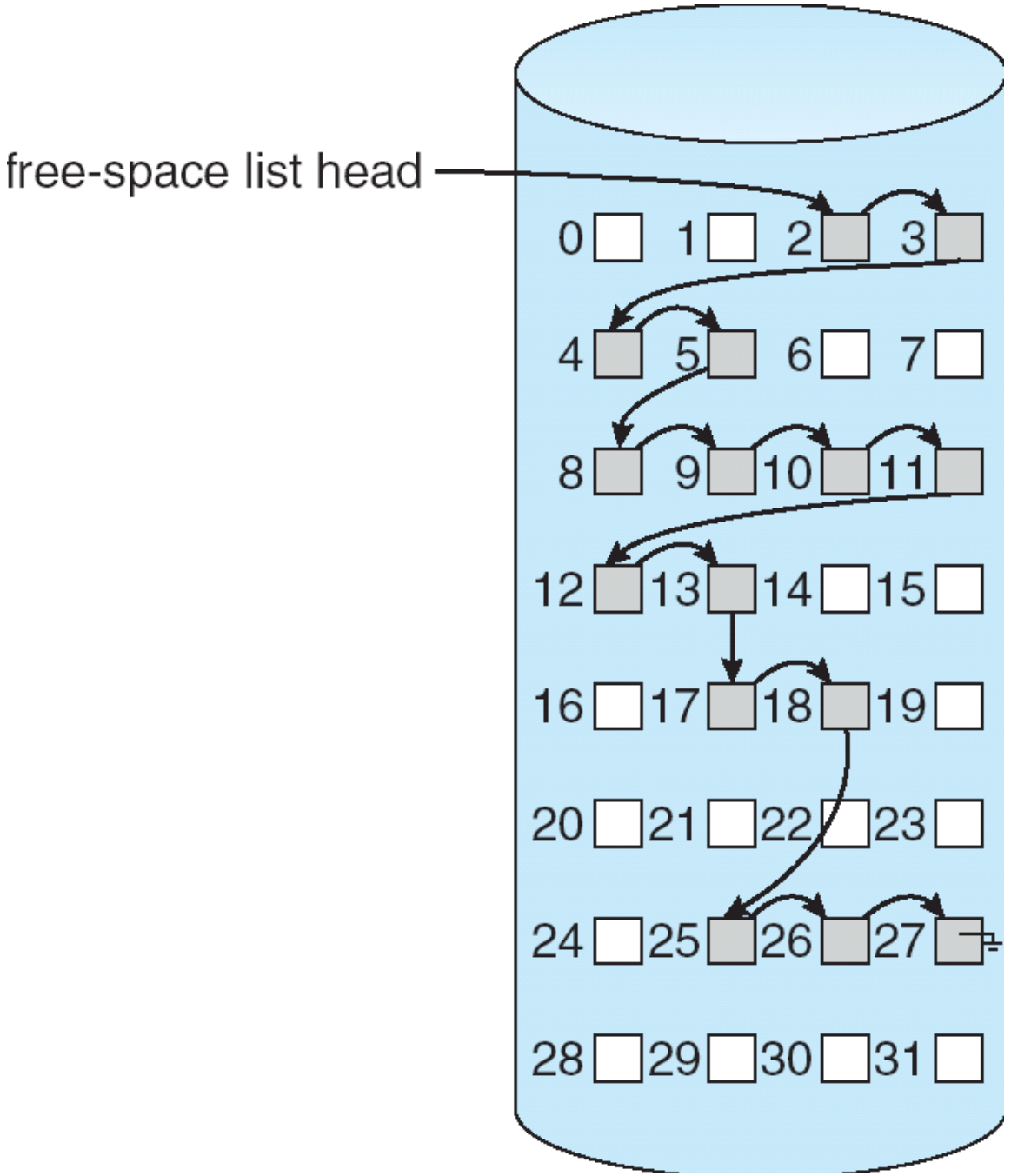
- Example: block size = 1 KB = 2^{10} bytes

  disk size = 68 GB ~ 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


free-space list head
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

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

Fact #1

File systems contain multiple data structures

Fact #2

These data structures have inter-depencencies

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

Journalling file systems

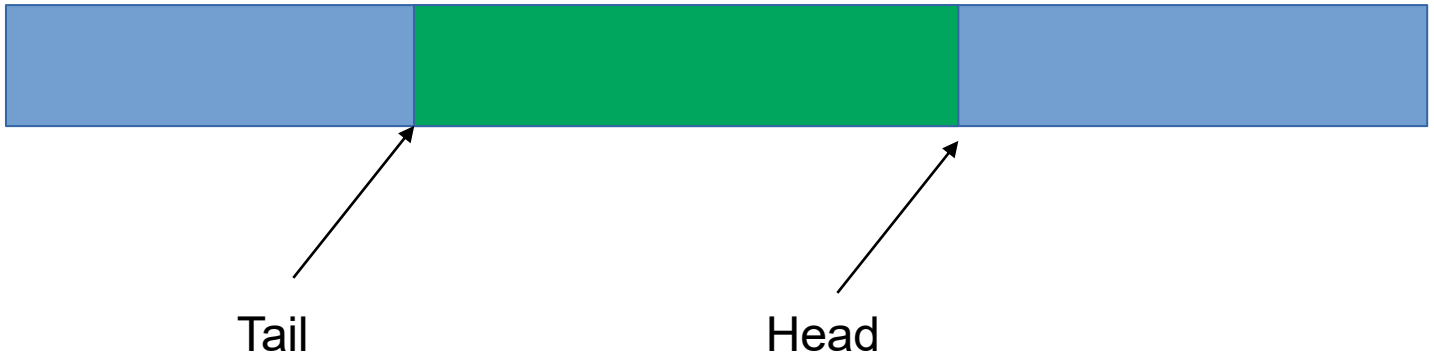
- Each modification is made as a transaction
- Keep a journal (log) of all pending transactions
- Interrupted transaction can be rolled-back
- Every update requires two writes
- Examples: NTFS, ext4

Snapshot-based

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

Log-structured

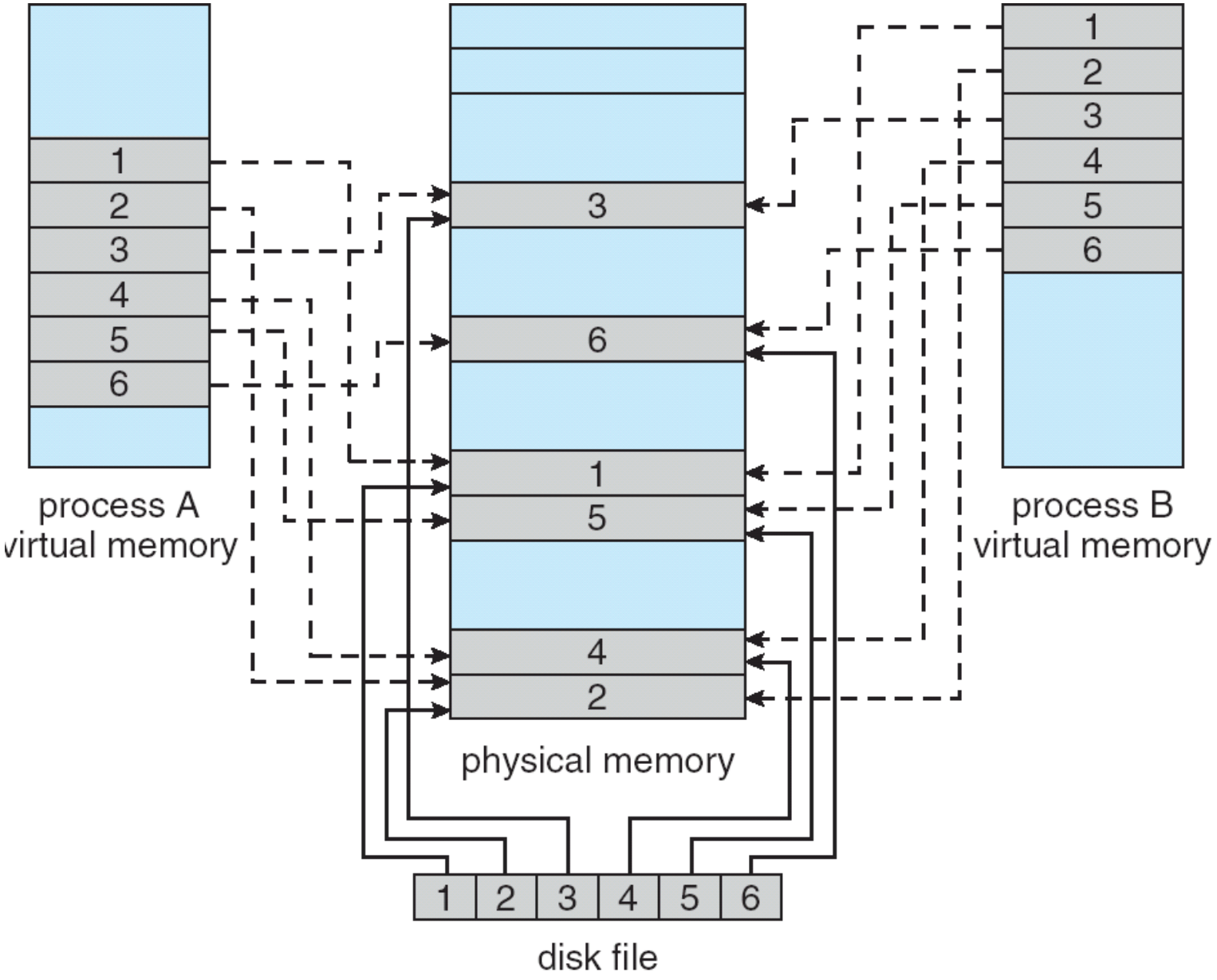
- Idea from ’88 - exploit sequential nature of disks
- Skip the structure just write where free
- Now revived!
  - Per inode logs in NOVA filesystem for non-volatile main memory (NVMM)



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



Next on

- Virtualization + Synchronization II
- Reading 10ed: Ch. 2.8, Ch. 18