CS 370: Operating Systems
[File Systems]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: Shrideep Pallickara

Topics covered in this lecture

- Block Allocations
  - Contiguous allocations
  - Linked allocations
  - Indexed allocations
    - iNodes
- Free space management
- Memory mapped files

Allocation methods: Objective and approaches

- How to allocate space for files such that:
  - Disk space is utilized effectively
  - File is accessed **quickly**

- Major Methods
  - Contiguous
  - Linked
  - Indexed

Contiguous Allocations

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
  - If file is of size **n** blocks and starts at location **b**
    - Occupies blocks **b**, **b+1**, …, **b+n-1**
- Disk head movements
  - None for moving from block **b** to (**b+1**)
  - Only when moving to a different track

Sequential and direct access in contiguous allocations

- Sequential accesses
  - Remember *disk address* of the last referenced block
  - When needed, read the next block
- **Direct access** to block `i` of file that starts at block `b`

  `b + i`

Contiguous allocations suffer from external fragmentation

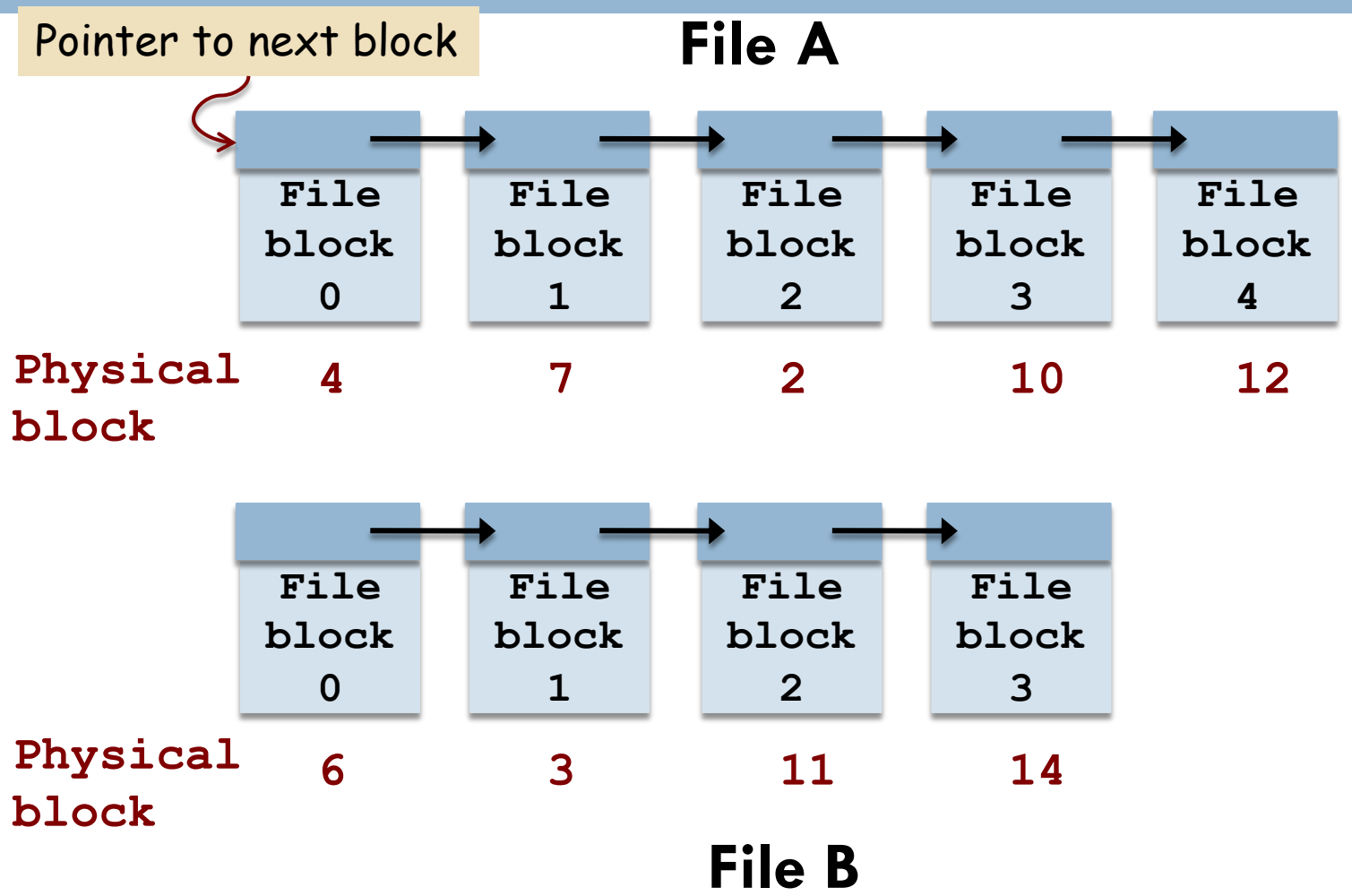
- Free space is broken up into chunks
  - Space is **fragmented** into holes
- Largest continuous chunk may be insufficient for meeting request
- **Compaction** is very slow on large disks
  - Needs several hours

Determining how much space is needed for a file is another problem

- **Preallocate** if eventual size of file is known?
  - Inefficient if file grows very slowly
    - Much of the allocated space is unused for a long time
- Modified contiguous allocation scheme
  - Allocate space in a continuous chunk initially
  - When space runs out allocate another set of chunks (**extent**)

Linked Allocations

Linked Allocation: Each file is a linked list of disk blocks



Linked List Allocations: Advantages

- **Every** disk block can be used
  - No space is lost in external fragmentation
- Sufficient for directory entry to merely store *disk address of first block*
  - Rest can be found starting there

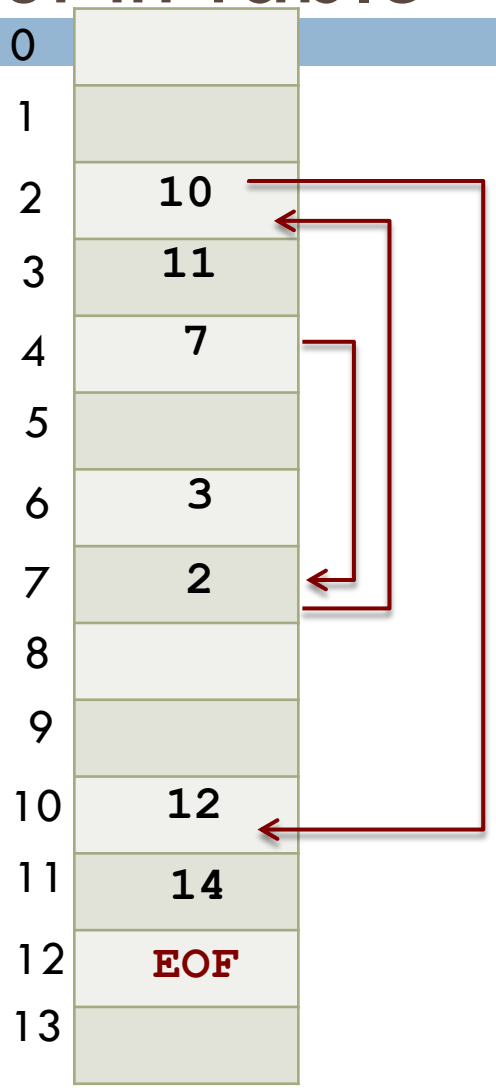
Linked List Allocation: Disadvantages

- Used effectively only for sequential accesses
  - Extremely **slow random access**
- Space in each block set aside for pointers
  - Each file requires *slightly more space*
- Reliability
  - What if a pointer is lost or damaged?

Linked List Allocations: Reading and writing files is much less efficient

- Amount of data storage in block is no longer a **power of two**
  - Pointer takes up some space
- **Peculiar size** is less efficient
  - Programs read/write in blocks that is a power of two

Linked list allocation: Take pointers from disk block and put in table



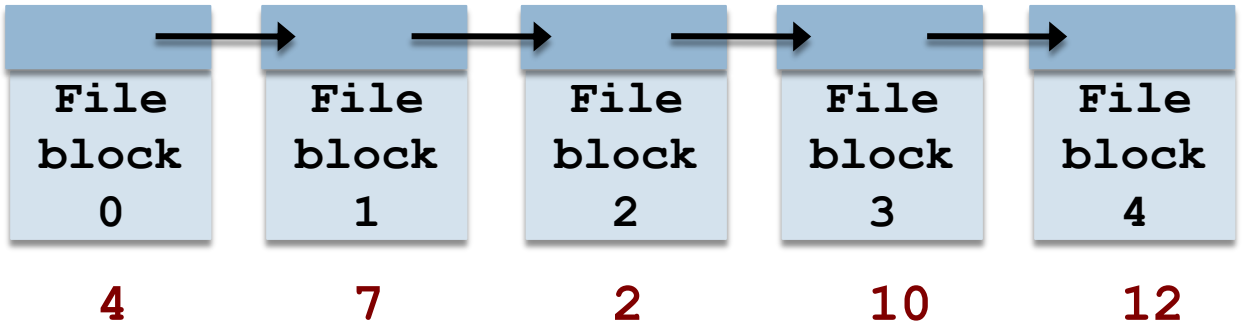


Table tracks **EVERY** disk block in the system

Linked list allocation using an index

- **Entire** disk block is available for data
- Random access is much easier
  - Chain must still be followed
    - But this chain could be *cached in memory*
- MS-DOS and OS/2 operating systems
  - Use such a file allocation table (FAT)

Linked list allocation using an index: Disadvantages

- Table must be cached **in memory** for efficient access
- A large disk will have a large number of data blocks
  - Table consumes a large amount of physical memory

Indexed Allocations

Indexed allocations

- Bring all pointers together into one location
  - **index block**
- Each file has its ***own*** index block
  - **ith** entry points to **ith** block of the file
  - Directory contains address of the index block

Indexed allocation supports direct access without external fragmentation

- Every disk block can be utilized
  - **No external fragmentation**
- Space wasted by pointers *is generally higher* than linked listed allocations
  - Example: File has two blocks
    - Linked listed allocations: 2 pointers are utilized
    - Indexed allocations: Entire index block must be allocated

iNODES

inode

- **Fixed-length** data structure
  - One per file
- Contains information about
  - File **attributes**
    - Size, owner, creation/modification time etc.
  - **Disk addresses**
    - File blocks that comprise file

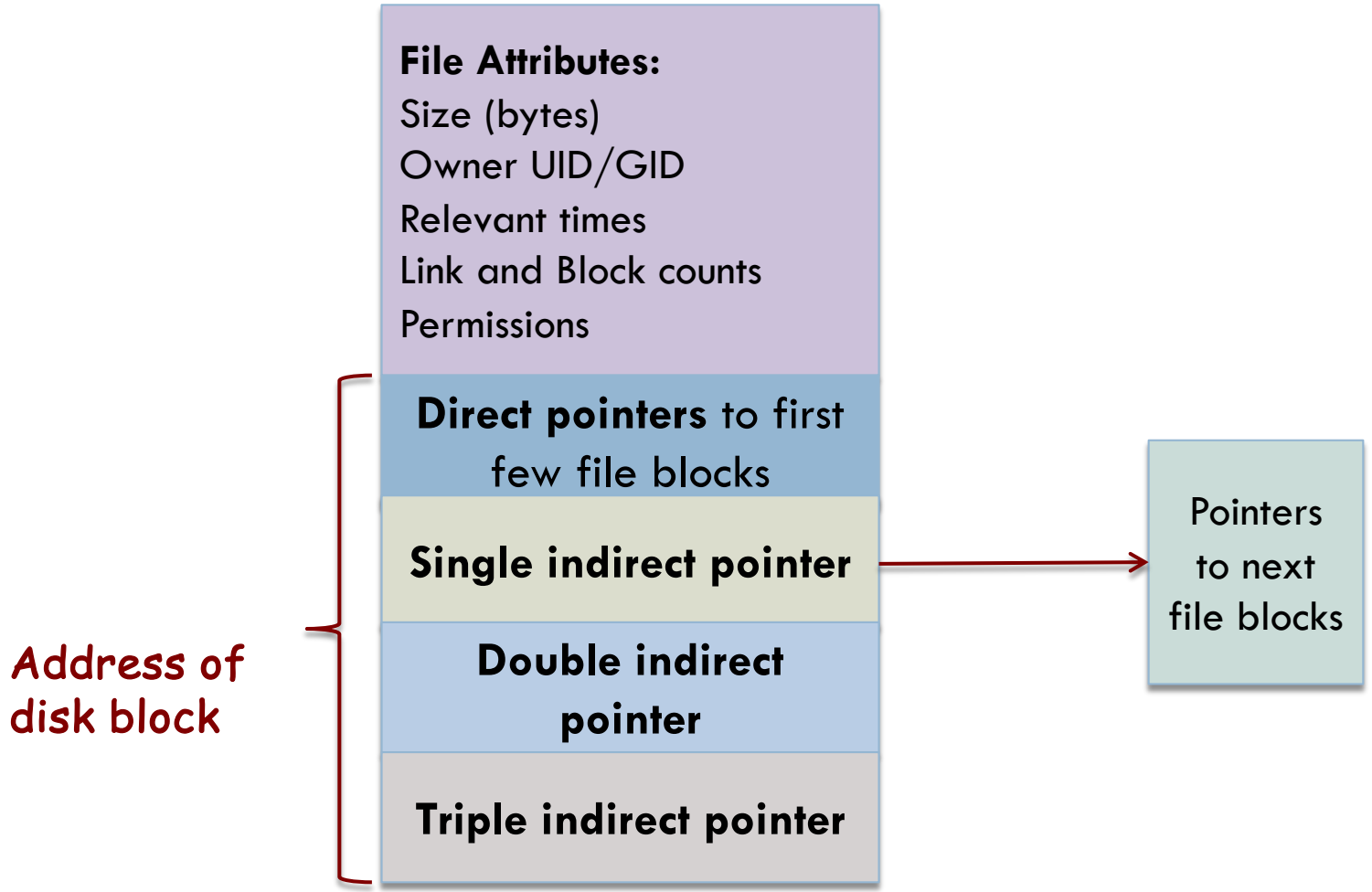
inode

- The inode is used to encapsulate information about a large number of file blocks.
- For e.g.
  - Block size = 8 KB, and file size = 8 GB
  - There would be a million file-blocks
    - inode will store info about the **pointers to these blocks**
  - inode allows us to access info for *all* these blocks
    - Storing pointers to these file blocks also takes up storage

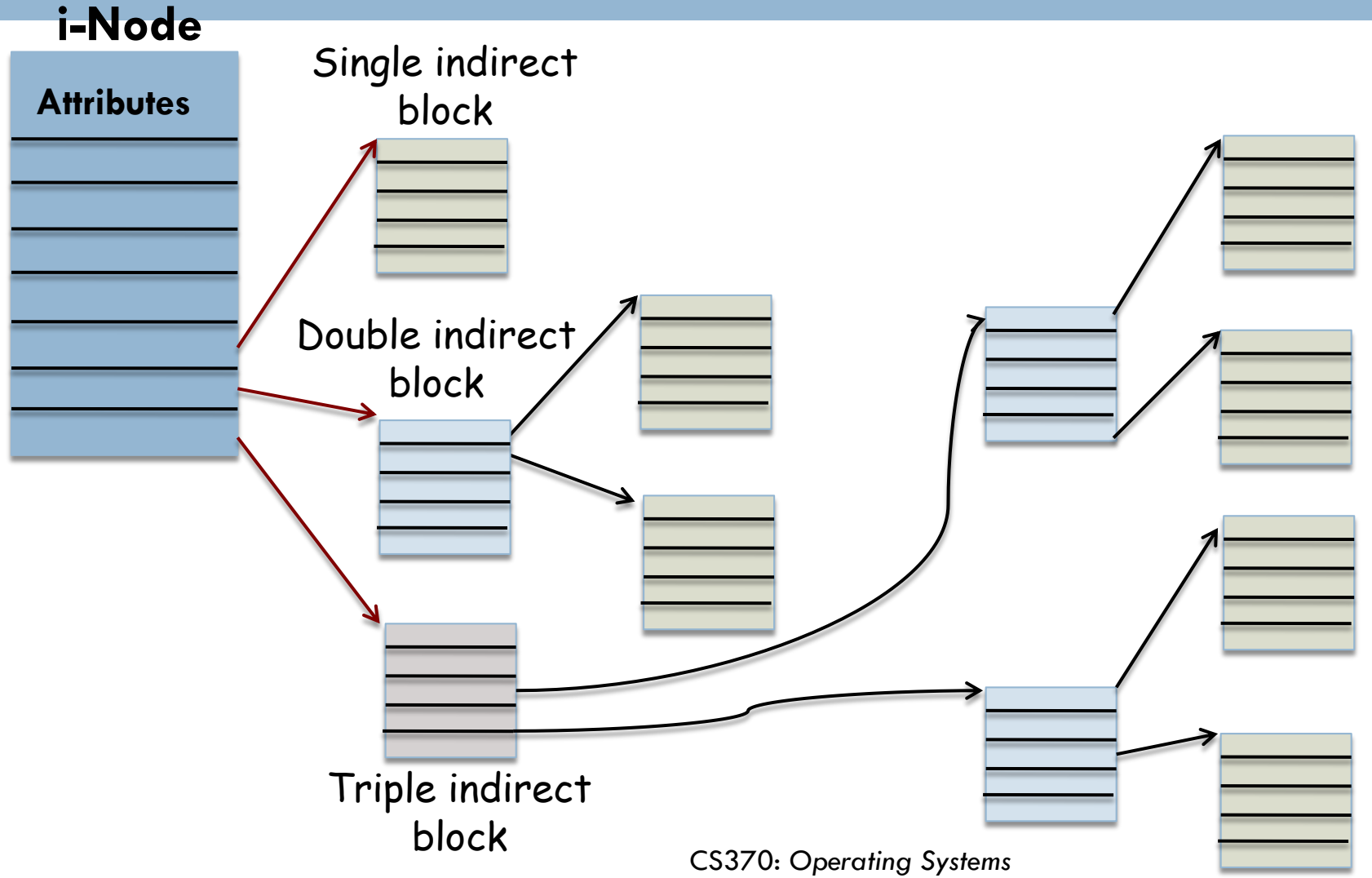
Managing information about data blocks in the inode

- First few data blocks of the file stored in the inode
- If the file is large: **Indirect** pointer
  - To a block of pointers that point to additional data blocks
- If the file is larger: **Double indirect** pointer
  - Pointer to a block of indirect pointers
- If the file is huge: **Triple indirect** pointer
  - Pointer to a block of double-indirect pointers

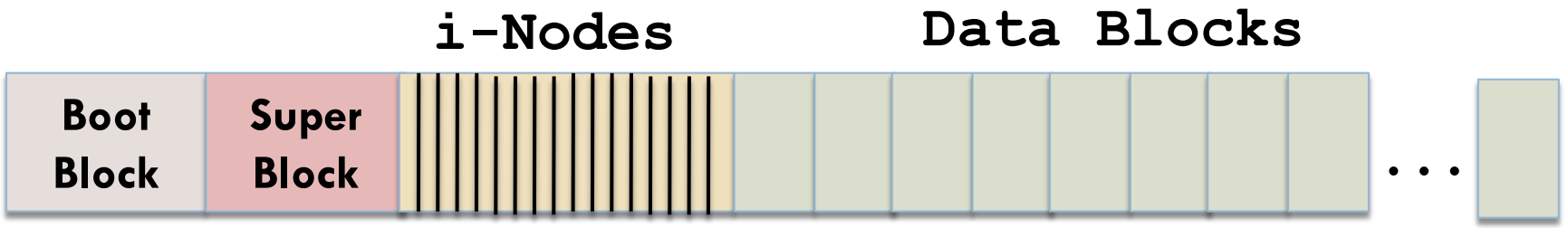
Schematic structure of the inode



i-Node: How the pointers to the file blocks are organized



Disk Layout in traditional UNIX systems



An integral number of inodes fit in a single data block

Super Block describes the state of the file system

- Total size of partition
- Block size and number of disk blocks
- Number of inodes
- List of free blocks
- inode number of the root directory

- Destruction of super block?
  - Will render file system unreadable

A linear array of inodes follows the data block

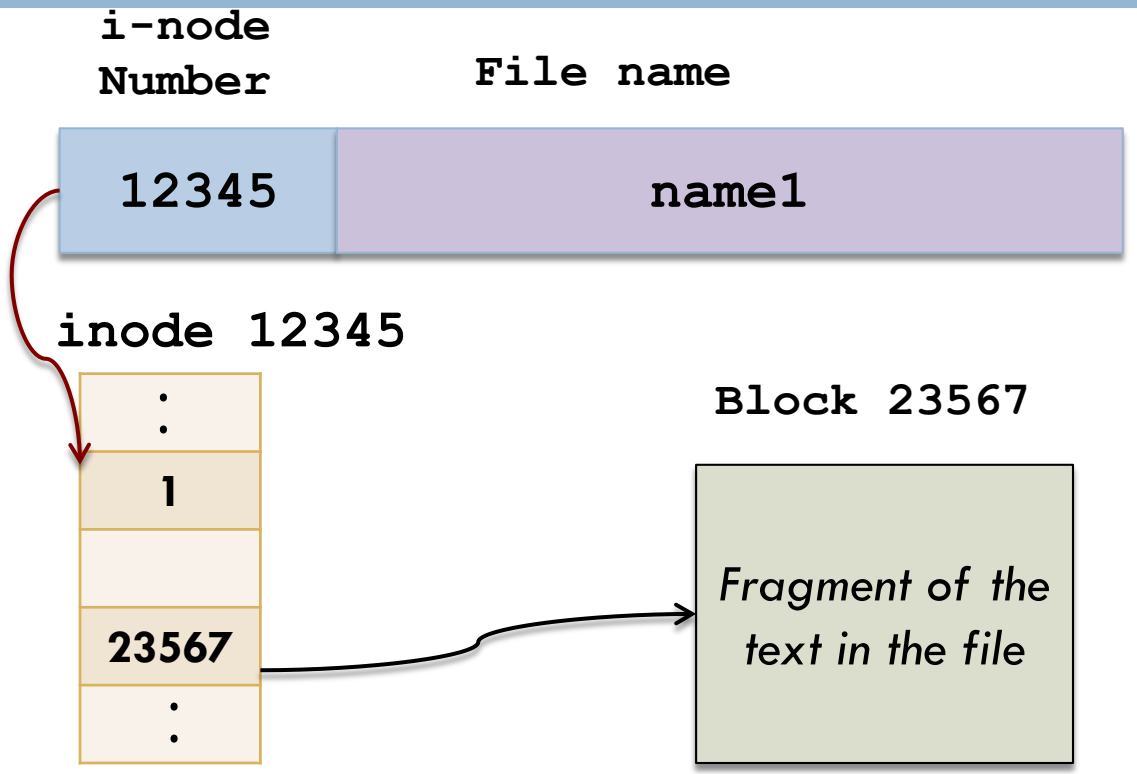
- inodes are numbered from **1** to some **`max`**
- Each inode is identified by its inode number
  - inode number contains info needed to **locate** inode on the disk
  - Users think of files as filenames
  - UNIX thinks of files in terms of inodes

UNIX directory structure

- Contains only file names and the corresponding inode numbers

- Use `ls -i` to retrieve inode numbers of the files in the directory

Directory entry, inode and data block for a simple file



Looking up path names in UNIX
Example: /usr/tom/mbox

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up usr yields i-node 6

i-node 6 is for /usr

Mode, size .. attributes
132

i-node 6 says that /usr is in block 132

Block 132 is /usr directory

6	.
1	..
19	dick
30	eve
51	jim
26	tom
45	zac

/usr/tom is in i-node 26

i-node 26 is /usr/tom

Mode, size .. attributes
406

i-node 26 says that /usr/tom is in block 406

Block 406 is /usr/tom dir

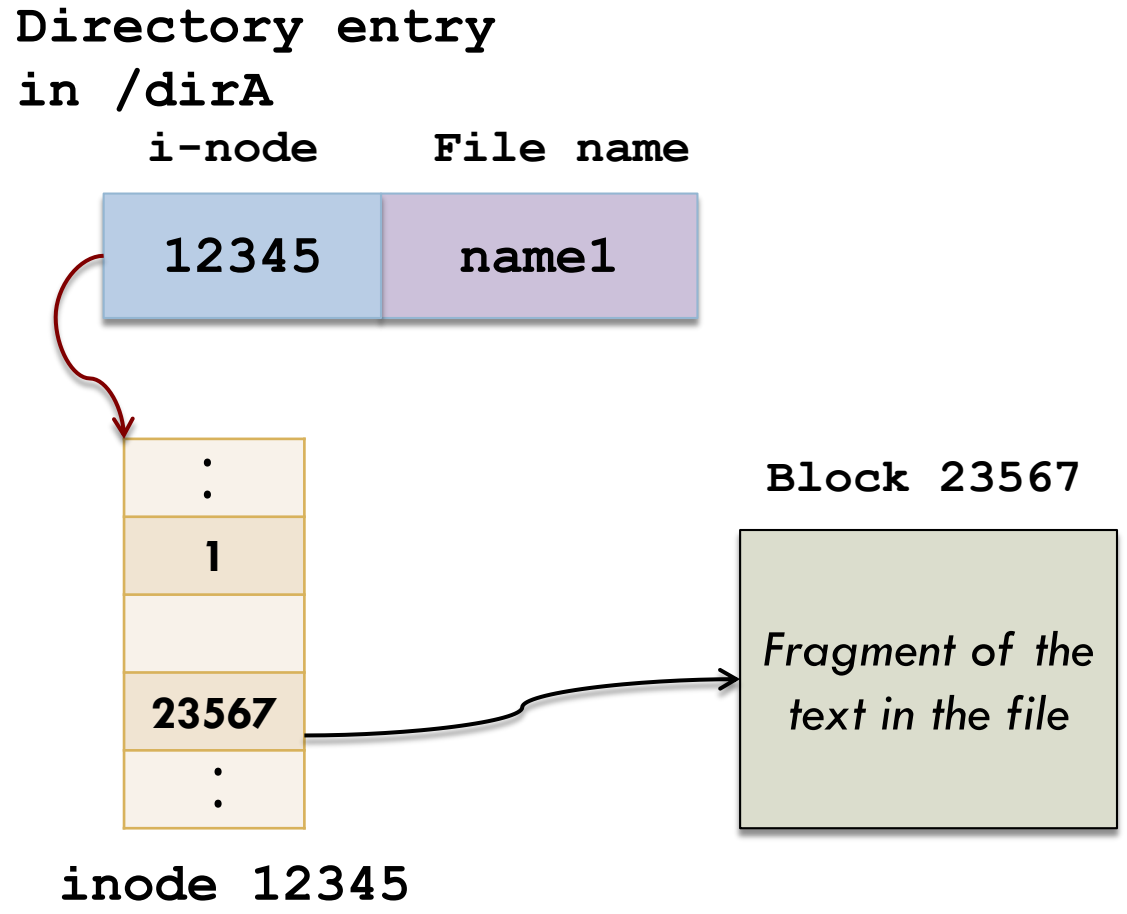
26	.
6	..
64	grants
92	dev
60	mbox
81	docs
17	src

/usr/tom/mbox is in i-node 60

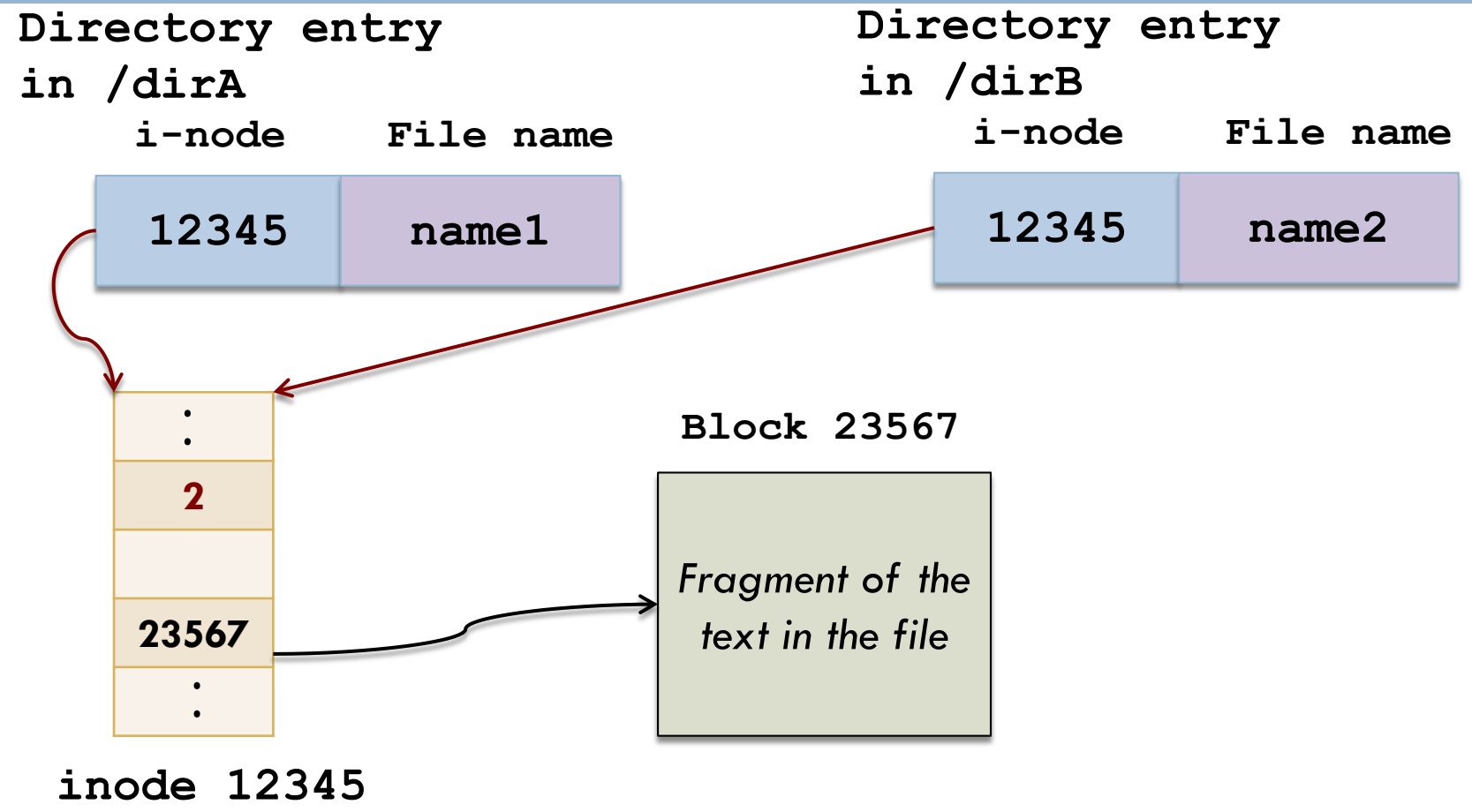
Advantages of directory entries that have name and inode information

- Changing filename only requires changing the directory entry
- Only 1 physical copy of file needs to be on disk
  - File may have several names (or the same name) in different directories
- Directory entries are small
  - Most file info is kept in the inode

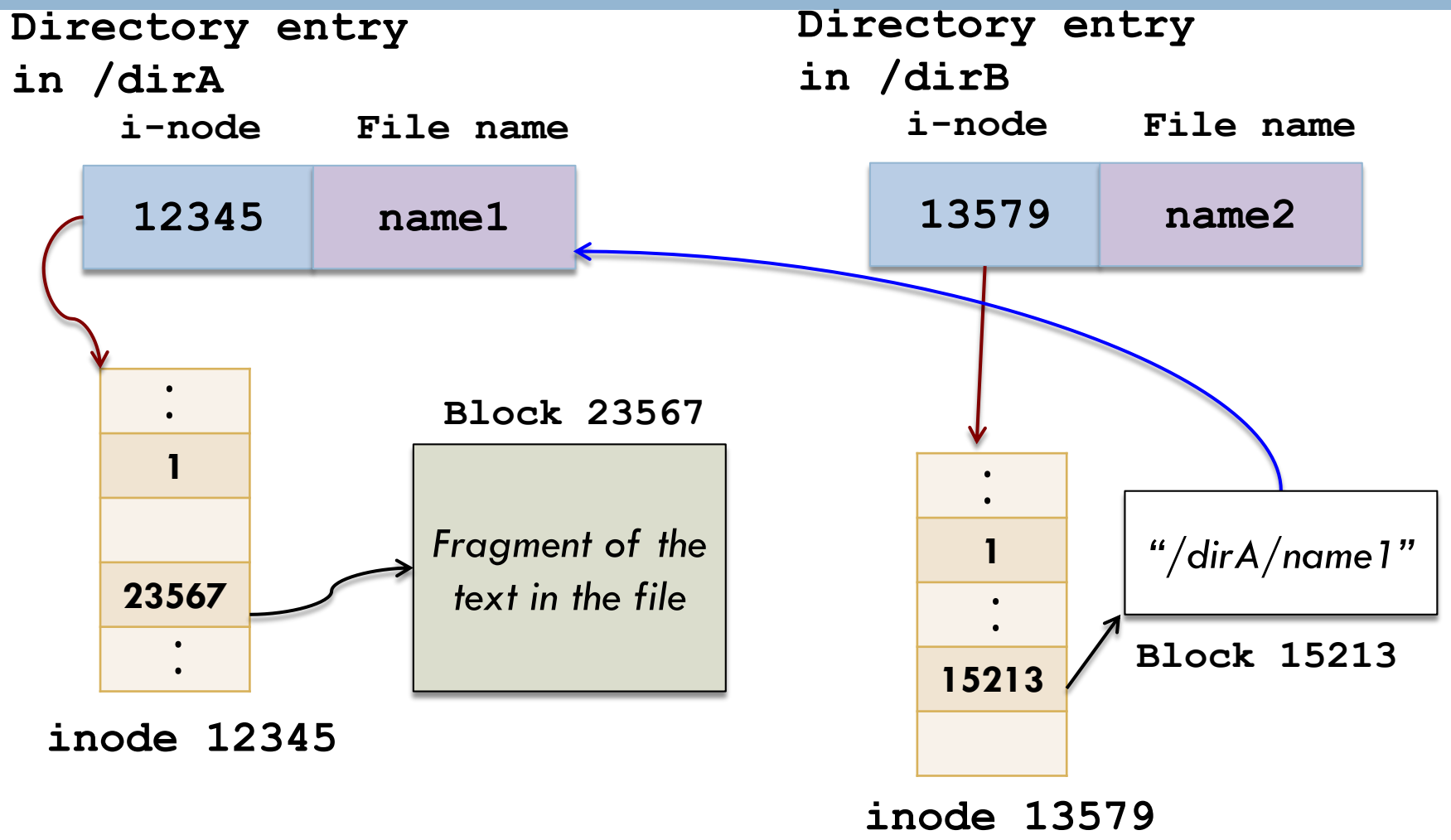
Two hard links to the same file



Two hard links to the same file



File with a symbolic link



Maximum size of your hard disk (8 KB blocks and 32-bit pointers)

- 32-bit pointers can address 2^{32} blocks
- At 8 KB per-block
  - Hard disk can be $2^{13} \times 2^{32} = 2^{45}$ bytes (32 TB)

The case for larger block sizes

- Larger partitions for a fixed pointer size
- Retrieval is more efficient
  - Better system throughput
- Problem
  - Internal fragmentation

Limitations of a file system based on inodes

- File **must fit** in a single disk partition
- Partition size and number of files are **fixed** when system is set up

inode preallocation and distribution

- inodes are **preallocated** on a volume
  - Even on empty disks % of space lost to inodes
- Preallocating inodes and spreading them
  - Improves performance
- Keep file's data block **close** to its inode
  - Reduce seek times

Checking up on the iNodes: The df –i command (*disk free*)

- inode statistics for a given set of file systems
  - Total, free and used inodes

df -i /s/bach/*

Filesystem	Inodes	IUsed	IFree	IUse%
/dev/cciss/c0d1p1	12746752	948362	11798390	8%
/dev/cciss/c0d2p1	10240000	150436	10089564	2%
/dev/cciss/c0d3p1	10240000	812727	9427273	8%
/dev/cciss/c0d4p1	10240000	930080	9309920	10%
/dev/cciss/c0d5p1	10240000	496744	9743256	5%
/dev/cciss/c0d6p1	10240000	167900	10072100	2%
/dev/cciss/c0d7p1	10240000	748709	9491291	8%
/dev/cciss/c0d8p1	12681216	760002	11921214	6%
/dev/cciss/c0d9p1	12681216	394165	12287051	4%

Free Space Management

Free space management

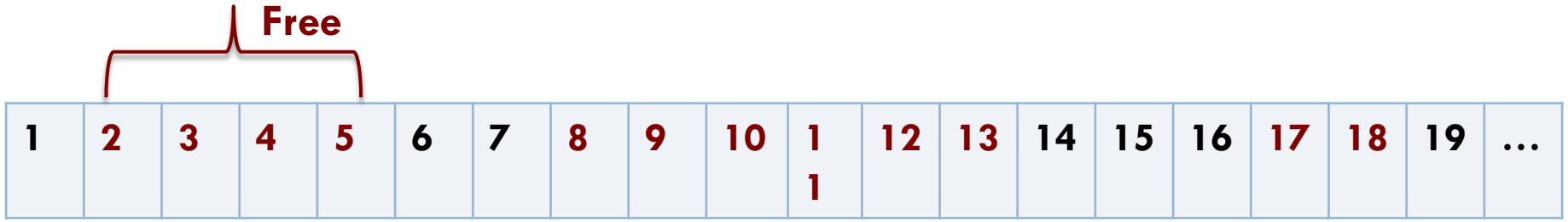
- Disk space is limited
  - **Reuse** space from deleted files
- Keep track of free disk space
  - Maintain **free-space list**
  - Record all free disk blocks
- Metadata I/O can impact performance

Free space management using the free-space list

- Creating a new file
  - **Search** free-space list for requisite space
  - Allocate that to the file
- Deletion of a file
  - Add file blocks of deleted file to the free-space list

Using bit vectors to implement the free-space list

- Each file block is represented with a bit
  - Block is free: bit is **1**
  - Block is allocated: bit is **0**
- A HDD with **n** blocks requires an **n**-bit vector



Bit Vector: 0**1111**00**1111**00**11**0…

Advantages of using the bit-vector

- **Simplicity**
- **Efficiency** in finding first free block
  - Or **n** consecutive free blocks
- Most CPUs have bit manipulation operators
  - Allows us to compute free blocks fairly fast

Finding free blocks using the bit vector

- A **0** valued word represents *allocated* blocks
- First **non-0** word is scanned for first 1-bit
  - This is the location of the first free block
- Free Block number

$$Bits_{Per\text{-}Word} \times Num_{\mathbf{0}\text{-value words}} + Offset_{First\ \mathbf{1}\text{-bit}}$$

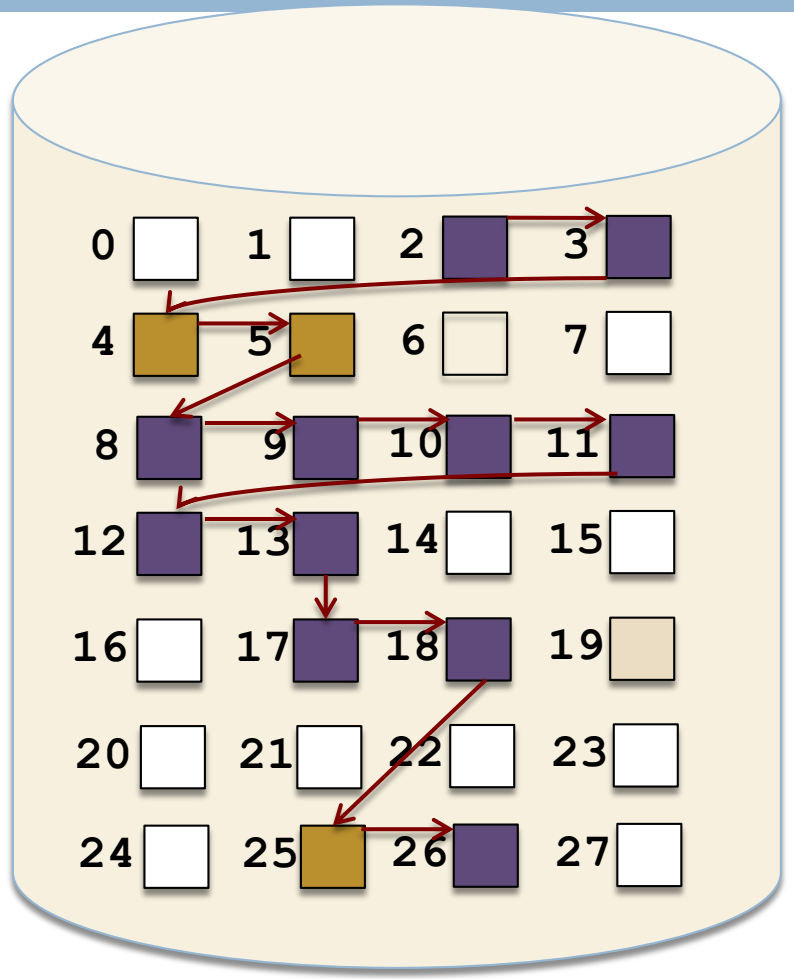
Problems with the bit vector approach

- For efficiency purposes, bit vector must be memory-resident
  - Difficult for larger disks
  - 1 TB hard disk with 4 KB blocks
    - Bit Vector = 32 MB
  - 1 PB disk = 32 GB bit vector
- Freeing 1GB of data on a 1 TB disk
  - Thousands of blocks of bit maps need to be updated
    - Blocks could be scattered all over disk

Linked list approach to free space management

- **Link** free-blocks
- Pointer to *first free* block
  - Stored in a special location on disk
  - Cached in memory

Tracking free space using the linked list approach



Problems with this approach

- To traverse list we must read each block
  - Substantial I/O

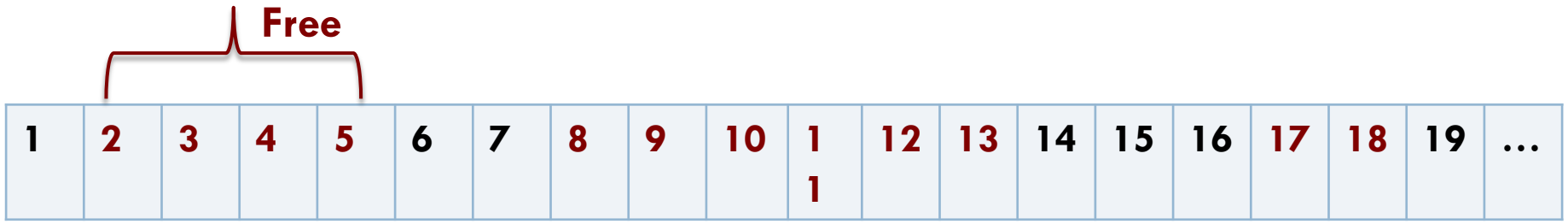
- Finding large number of free blocks is expensive

Grouping to augment the linked list approach

- Set aside one block for tracking **portion** of chain
  - First **n-1** entries are free blocks
  - Last entry points to *another set* of free blocks
- If tracker-block has 512 entries
  - After linked list block is loaded in memory
    - Next 510 blocks do not need I/O operations
  - 512th entry points to another tracker-block

Free space management using Counting

- Several contiguous blocks are free or allocated simultaneously
- Keep address of first free block AND
  - Number of contiguous blocks that follow it



{2, 3} {8, 5} {17, 1} ....

Space Maps used in ZFS (Zettabyte File System)

- 1 ZB = 2^{70} bytes
- Controls size of I/O data structures
  - Minimize I/O needed to manage them
- **Metaslab**s divide space on disk into chunks
  - A volume has 100s of metaslabs
- A metaslab has a space-map
  - *Counting algorithm* to store info on space maps

ZFS free space management

- Does not write I/O metadata directly to disk
  - Free-space list updated on disk using transactional techniques
- When space is (de)allocated from metaslab
  - Corresponding space map is loaded into memory
    - B-Tree structure indexed by block offsets

Memory Mapped Files

Memory mapped files

- `open(), read(), write()`
  - Requires system calls and disk access
- Allow part of the virtual address space to be logically associated with the file
  - **Memory mapping**

Memory-mapping maps a disk block to a page (or pages) in memory

- Manipulate files through memory
  - Multiple processes may **map** file concurrently
    - Enables data sharing
  - Since JVM 1.4, Java supports memory-mapped files
    - FileChannel
- Writes to files in memory are not necessarily immediate

Memory mapped files: Things to watch for

- Make sure that two processes do not see **inconsistent views** of the same file
- File may be **larger** than the entire virtual address space!
  - Map portions of the file

PERFORMANCE

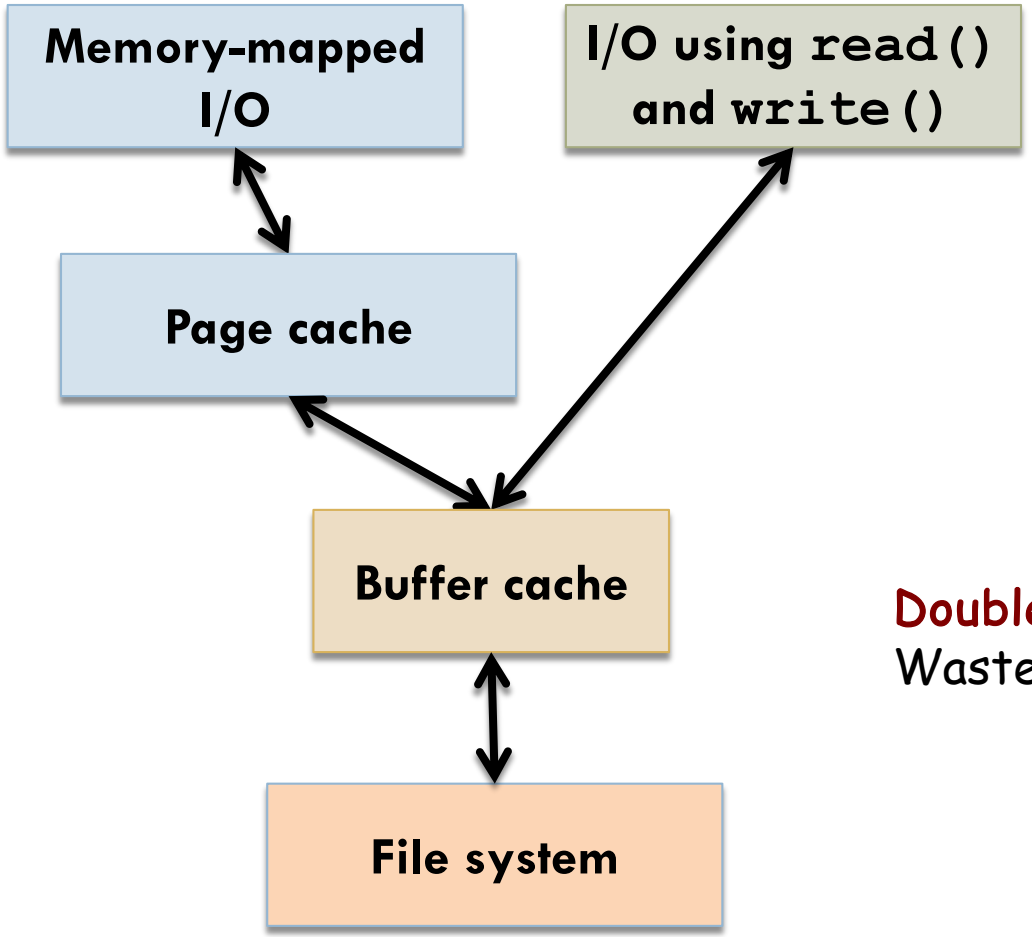
Improving reads from the disk

- Disk controllers have onboard **cache** that can store multiple tracks
- When a seek is performed
  - Track is read into the disk cache
  - Starting at the sector under disk head
    - Reduces latency time
- Controller then transfers sectors to the OS

Buffering data read from disks since they may be used again

- Maintain a separate section of memory for the **buffer cache**
- Cache file data using a **page cache**
  - Use virtual memory techniques
  - Cache as pages rather than file blocks
  - Access interfaces with virtual memory
    - Not the file system

I/O without a unified buffer cache: Double caching

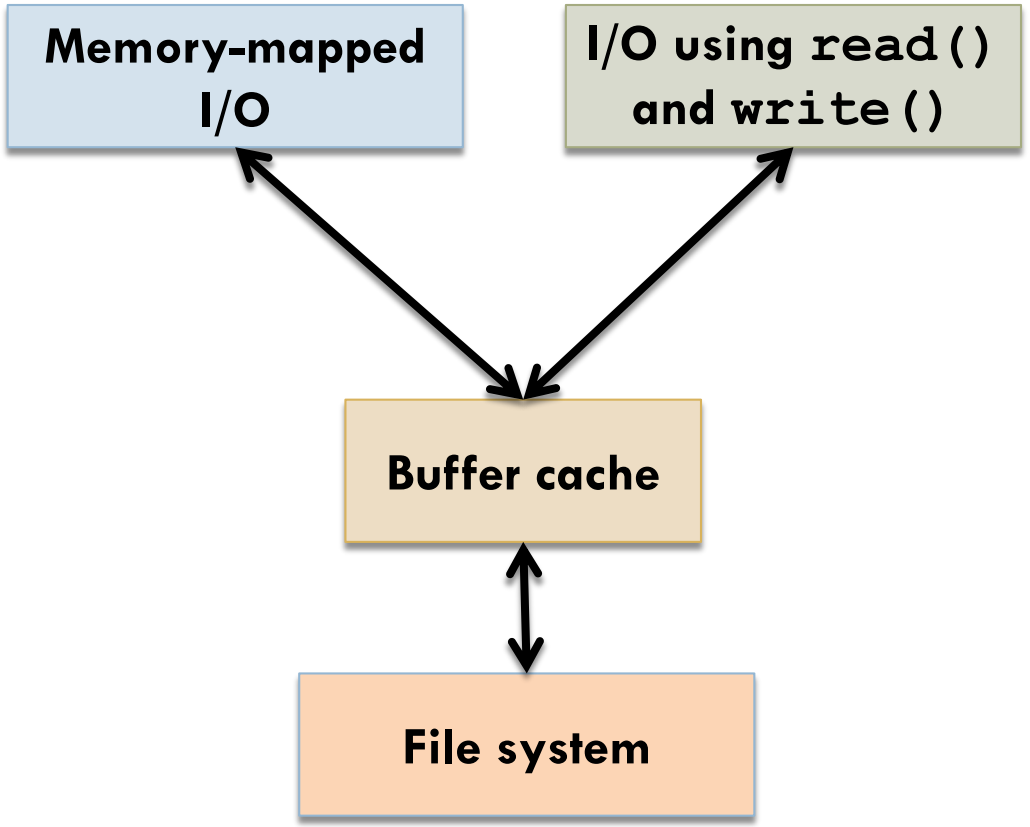


Double Caching:
Wastes memory and CPU cycles

Unified buffer cache

- Memory mapping and `read()`/`write()` system calls use the **same** page cache
- Allows virtual memory to manage the file system data

I/O using a unified buffer cache



Need to make a distinction between pages allocated to processes and page cache, else …

- Processes performing I/O will use most of the memory set aside for caching pages
- Pages may be reclaimed from processes
- Solution: **Fixed** limit for process pages and the file-system page cache
  - Prevent one from forcing out the other
  - Solaris 8

Synchronous writes

- Writes are **not buffered**
- Occurs in the **order** that the disk receives them
- Calling routine waits for data to reach disk
  - Blocking call
- Metadata writes tend to be synchronous
- Databases use this for atomic transactions

Asynchronous writes

- Data stored in the cache
- Control returns to the caller immediately
- Done majority of the time

Page cache, disk drivers and asynchronous disk writes

- When data is written to disk
  - Pages are buffered in the cache
- Disk driver **sorts** its output queue based on disk addresses
  - Minimize disk head seeks
  - Write at times optimized for disk rotations

Page cache and page replacement algorithms

- Replacement algo depends on file access type
- File being read/written sequentially
  - Pages should not be replaced in LRU order
    - Most recently used page may never be used again
- **Free-behind**
  - Remove page from buffer when next one requested
- **Read-ahead**
  - Requested page and subsequent pages are cached

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 11]
- *Andrew S Tanenbaum. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 4]*
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.* [Chapter 4]