
1878
Western
UNIVERSITY · CANADA

Chapter 14 – File-System Implementation

Spring 2023

Overview

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery

File-System Structure

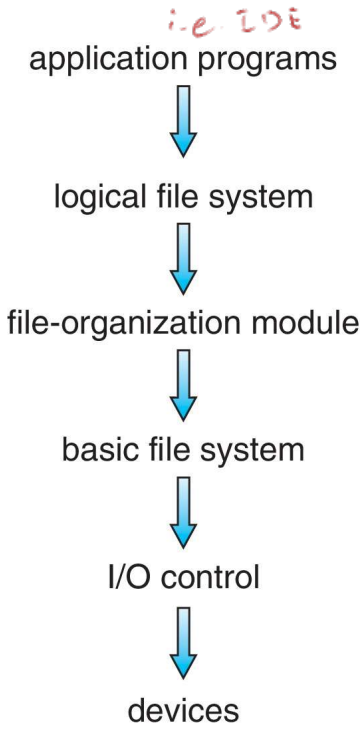
- A **file system** provides an interface between users and files
- The file system manages
  - File structure
  - Allocating storage space
  - Recovering free space
  - Track the location of data on disk (Logical to physical mappings)
  - Interface between the operating system and secondary storage

File-System Structure

- Operating systems provide many different file systems (Linux supports more than 130!)
- Which file system to use depends on need
  - Performance?
  - Reliability?
  - Features?
- E.g. A file system optimized for temporary storage will sacrifice reliability for performance
- New file systems are being created all the time. **FUSE (File system in user space)** provide an interface to easily roll your own file system without editing kernel code

File-System Structure

- File systems are composed of multiple layers
  - Layering reduces complexity and redundancy
  - However, layering adds overhead and decreases performance



File-System Structure

- I/O control layer set of hardware command indicate where data stored.
  - Consists of device drivers and interrupt handlers
  - Manage I/O devices at the I/O control layer
  - Given commands like
    - `read drive1, cylinder 72, track 2, sector 10, into memory location 1060`
  - Outputs low-level hardware specific commands to hardware controller

File-System Structure

- Basic file system layer
  - Uses buffers and cache to store data as it is retrieved or stored
  - Given more generic commands like
    - `retrieve block 123`
  - This layer manages I/O request scheduling (Chapter 11)

File-System Structure

- File organization module layer
  - Understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation

File-System Structure

- Logical file system layer
  - Manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Maintains **file control blocks (FCBs)**
  - Directory management
  - Protection access to file

File-System Operations

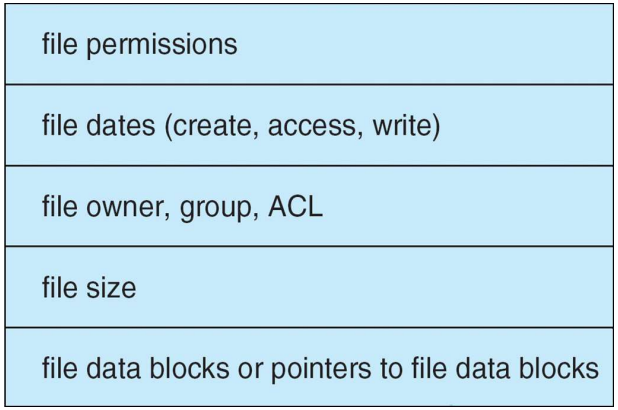
- File systems provide structures and operations to manage files and directories
- File systems use several on-storage and in-memory structures
- Operations are performed against one or more of these structures
- On-storage structures are persistent
- In-memory structures last until file system dismount or system reboot

File-System Operations

- Common on-storage structures
  - **Boot control block** – (1 per volume) contains info needed by system to boot OS from that volume
    - Needed if volume contains OS, usually first block of volume
    - Otherwise, this may be empty
  - **Volume control block** – (1 per volume) contains volume details such as total # of blocks, # of free blocks, block size, free block pointers
  - **Directory structure** – (1 per filesystem) organizes the files

File-System Operations

- Common on-storage structures (continued)
  - **FCBs** – (Usually 1 per file, although NTFS on Windows uses a master file table) Contains details about the file



File-System Operations

- Common in-memory structures
  - **Mount table** – Storing file system mounts, mount points, file system types
  - **Directory structure cache** – Holds recently used directories
  - **System-wide open-file table** – contains a copy of the FCB of each file and other info
  - **Per-process open-file table** - contains pointers to appropriate entries in system-wide open-file table as well as other info

File-System Operations

- Operations
  - E.g. To create a new file, a process calls the *logical file system layer* to create
  - The *logical file system layer* will allocate a new **FCB**, assign the **FCB** and the new file to a directory in the **directory structure**
  - The *file organization module layer* will find logical blocks for the file
  - The *basic file system layer* will map the logical blocks to physical blocks
  - The *I/O control layer* will write the data to disk

File-System Operations

- Operations

cache?

  - E.g. To read the new file, a process calls the *logical file system layer* to open and read

  - The *logical file system layer* will search the **system-wide open-file table**

    - If it is already open, a **per-process open-file table** is created pointing to this entry and the entry is incremented

    - If it is not already open, search the **directory structure** for the file and add the **FCB** to the **system-wide open-file table**

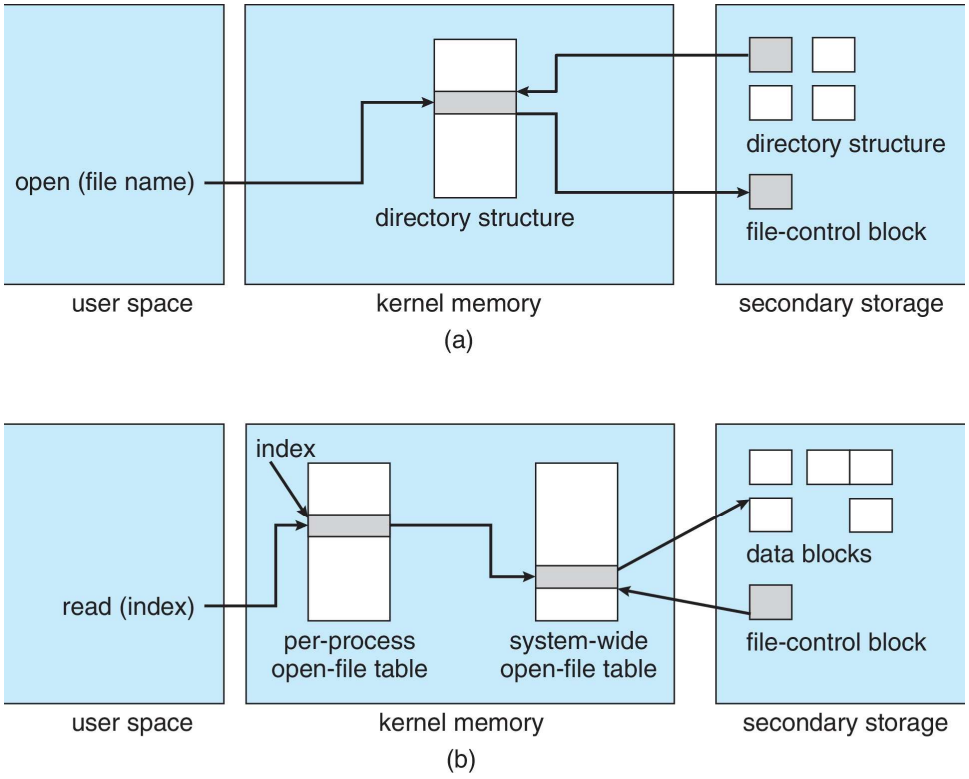
  - Update the **per-process open-file table** to include read location in the file and access mode

File-System Operations

- Operations
  - E.g. To close the new file, a process calls the *logical file system layer* to close
  - The *logical file system layer* will remove the **per-process open-file table**
  - Decrement the **system-wide open-file table** entry
    - If the open count is now 0, write any metadata to the **directory structure** and remove the entry

File-System Operations

- Operations
  - E.g. (a) Open and (b) Read

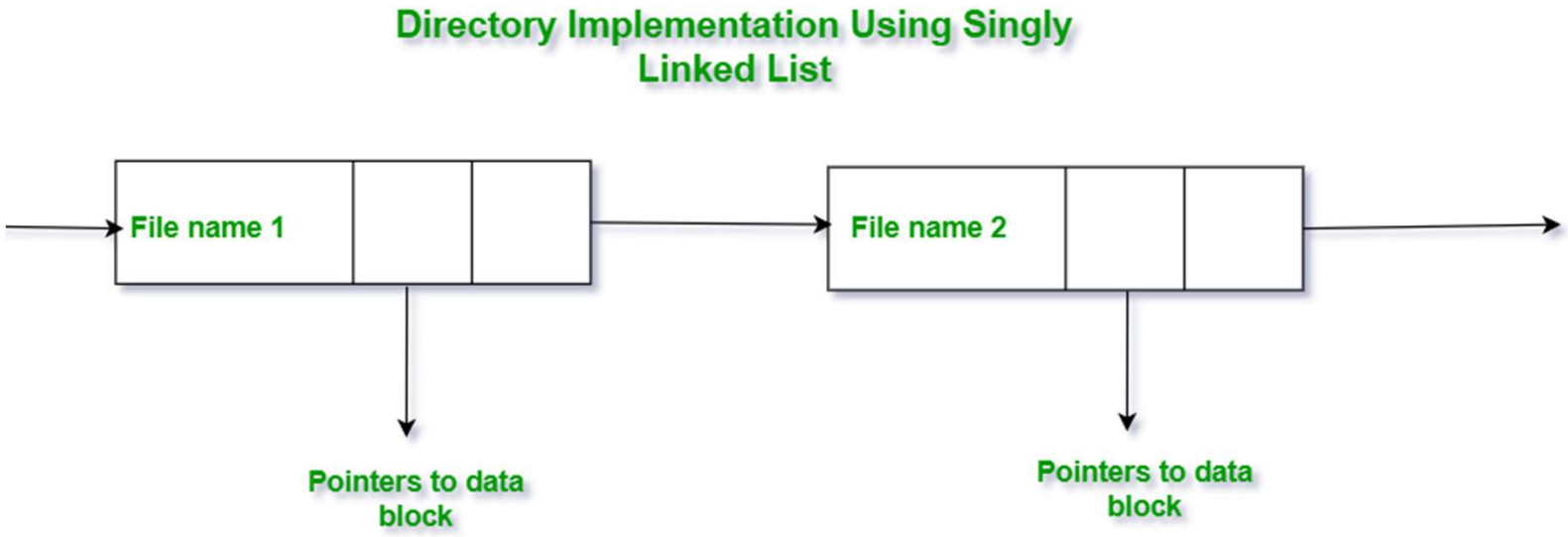


Directory Implementation

- The directory structure implementation affects efficiency, performance, and reliability
- **Linear list** – file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute. Using a cache can help
  - Linear search time
  - Could keep ordered alphabetically via linked list or use B+ tree

B+ tree / hash table

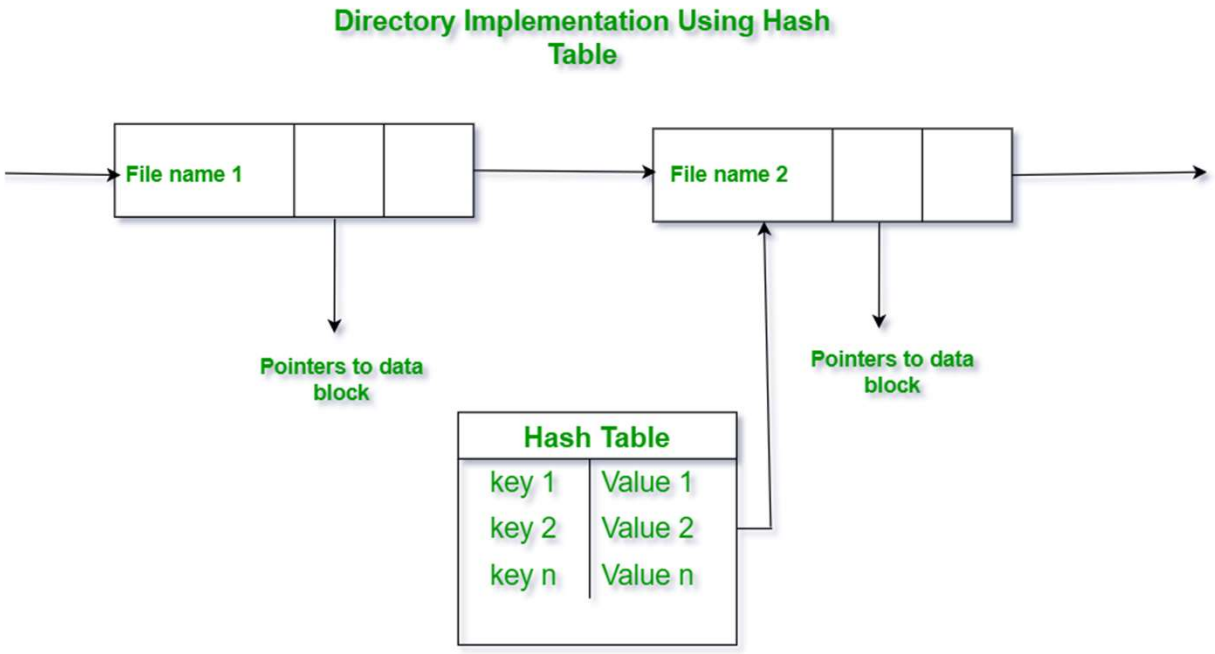
Directory Implementation



Directory Implementation

- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
    - Expand the hash table space when a collision occurs or
    - Use a chained-overflow hash table

Directory Implementation



Allocation methods

- The file system assigns multiple contiguous sectors on physical disk to a logical block
- Files consist of one or more blocks
- Free space consists of free or unused blocks
- The file system must keep track of these blocks effectively and efficiently
- Three approaches to assigning blocks to files
  - Contiguous allocation
  - Linked list allocation
  - Indexed allocation

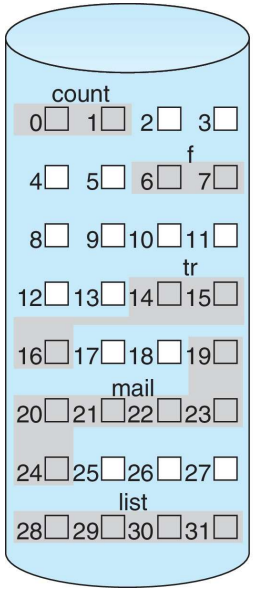
Allocation methods

- Contiguous allocation - Each file occupies a contiguous set of blocks
  - Pro
    - Usually has the best performance (very little head movement)
    - Simple to implement. You just need the starting block and the number of blocks
  - Con
    - Harder to find space on disk
    - The entire file size needs to be known ahead of time
    - Fragmentation and compaction issues again (see Chapter 9)

for HDD only? or also for SSD?

Allocation methods

- Contiguous allocation
  - The `start` and `length` are stored in the FCB



directory

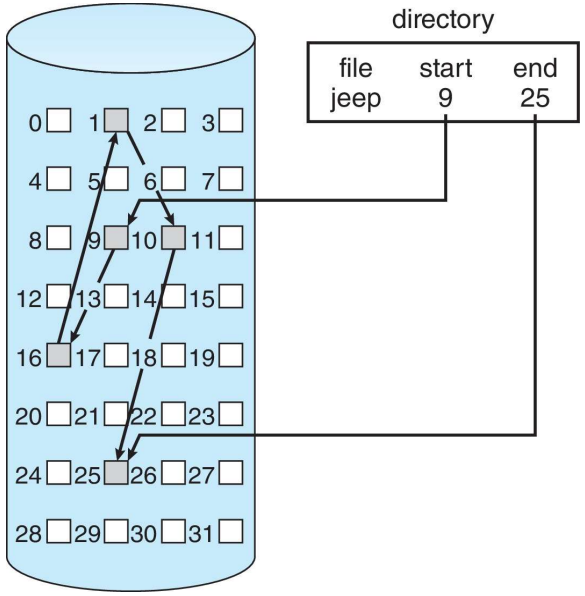
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation methods

- Linked list allocation – A linked list of blocks
  - Pro
    - No external fragmentation or compaction
    - The entire file size does not need to be known ahead of time
  - Con
    - A small portion of the block needs to be reserved for the pointer to the next block. **Clustering** blocks together and pointing to the next cluster can help.
    - Random access is slow. To read a portion of a file, you need to traverse the chain from the start until the block you want

Allocation methods

- Linked list allocation – A linked list of blocks
  - The `start` and `end` are stored in the FCB



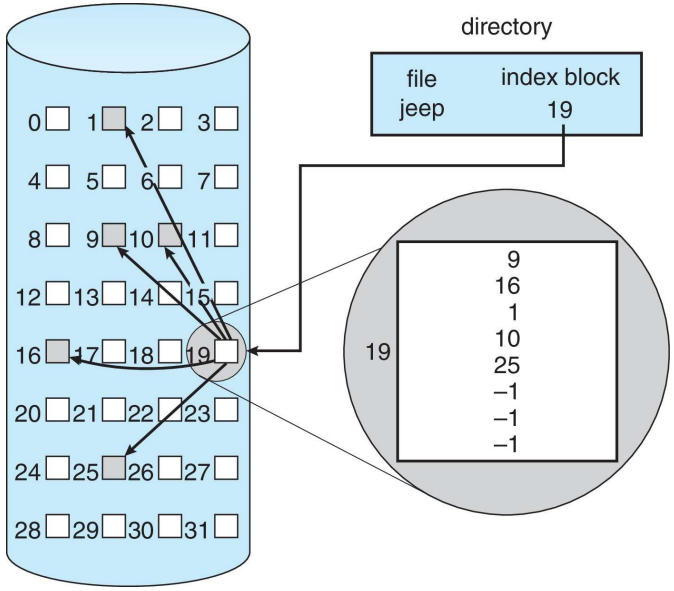
data - regular data
data - pointer to heap
data structure
(small proportion)

Allocation methods

- Indexed allocation – Each file has an index block. The index points to all the other blocks
  - Pro
    - No external fragmentation or compaction
    - Random access is very fast
  - Con
    - If the number of indexes exceeds the size of the index block, you need to employ multiple layers of index blocks. A 4KB index block with 4 byte pointers can store 1,024 pointers or a file size of 4MB. Two layers can hold 1,048,576 pointers or 4GB.

Allocation methods

- Indexed allocation – Each file has an index block. The index points to all the other blocks
  - The `index block` is stored in the FCB

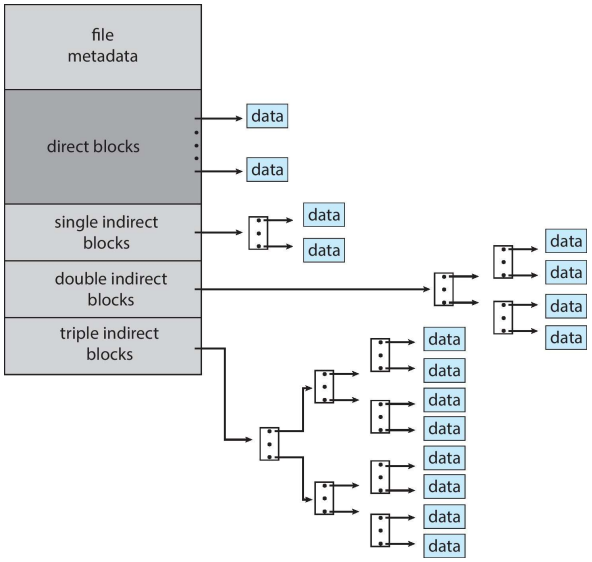


Allocation methods

- Indexed allocation – Each file has an index block. The index points to all the other blocks
  - Some operating systems (e.g. UNIX/Linux) will combine direct and indirect blocks.
    - Small files will not use an index block. Large files will use multiple layers
- Increasing the size of the block pointers raises the maximum file size
  - Using 4 byte (32-bit) pointers, the maximum file size without layering is 2^{32} or 4GB
  - Most UNIX/Linux file systems support 8 byte (64-bit) pointers. `ZFS` supports 16 bytes (128-bit)

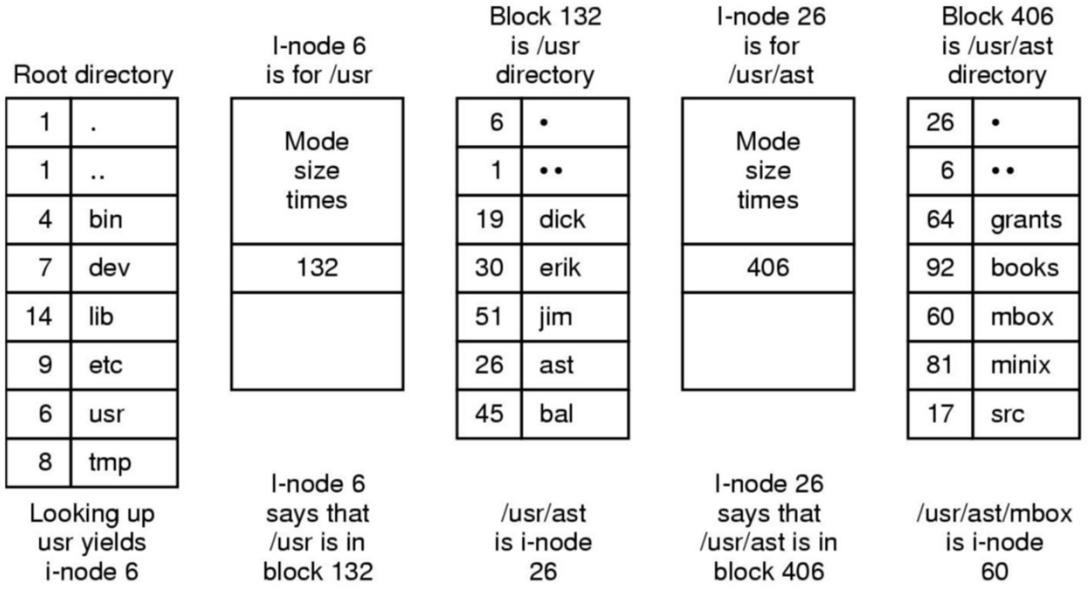
Allocation methods

- Indexed allocation – Each file has an index block. The index points to all the other blocks
  - A sample UNIX/Linux inode structure



Allocation methods

- E.g. Looking up `/usr/ast/mbox` with Unix inodes



Allocation methods

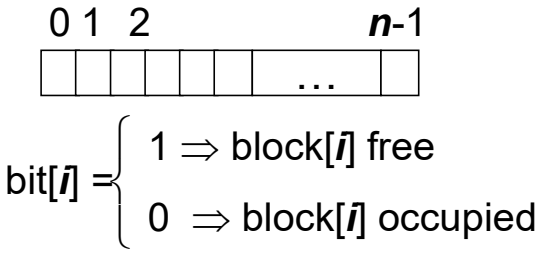
- Performance
  - If the workload against a file is known, the allocation type can be chosen at file creation time to optimize for sequential or random access. Files can be converted from one type to another
  - Keeping index blocks in memory can increase performance
  - Employing strategies to reduce disk head movement can pay dividends (Chapter 11)
    - NVMs have no moving parts so optimizing differently for NVM drives is important

Free-Space Management

- The file system must maintain a **free-space list** to track available blocks/clusters
- When a new file is created, blocks from the free-space list are allocated and removed from the list
- When a file is deleted, blocks are added back to the free-space list
- Note: The free-space "list" may be implemented as a bitmap or a true list

Free-Space Management

- Bitmap
  - Simply flag every block in the system as 0 (in-use) or 1 (free)
  - E.g. if blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, 27 are free, represent this as:
    - `00111100111111000110000011100000`

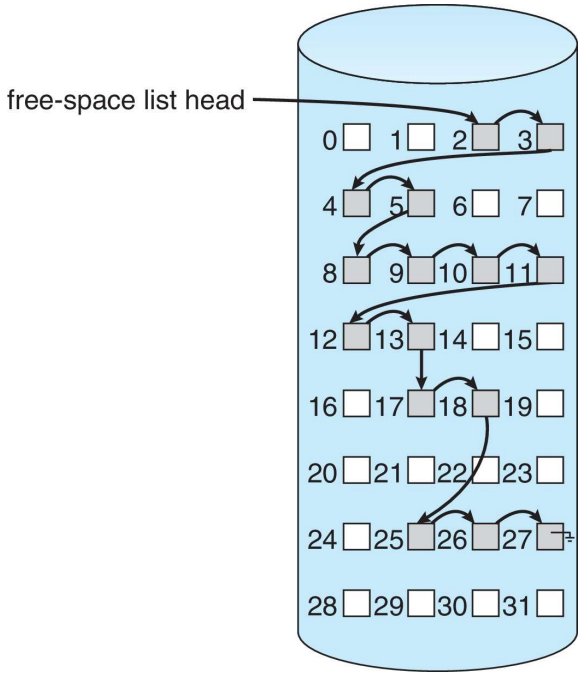


Free-Space Management

- Bitmap
  - Pros
    - Finding a free block (or set of blocks) is simple and efficient
    - Easy to implement bitwise CPU instructions to find free blocks
    - Easy to find contiguous blocks if that is required
  - Cons
    - A copy of the bitmap must be held in memory. 1TB disk with 4KB blocks requires 32MB in memory

Free-Space Management

- Linked List



Free-Space Management

- Linked list
  - Pros
    - A free block can be found at the head of the list
  - Cons
    - Traversing the list can be costly but a full traversal is rarely done

Free-Space Management

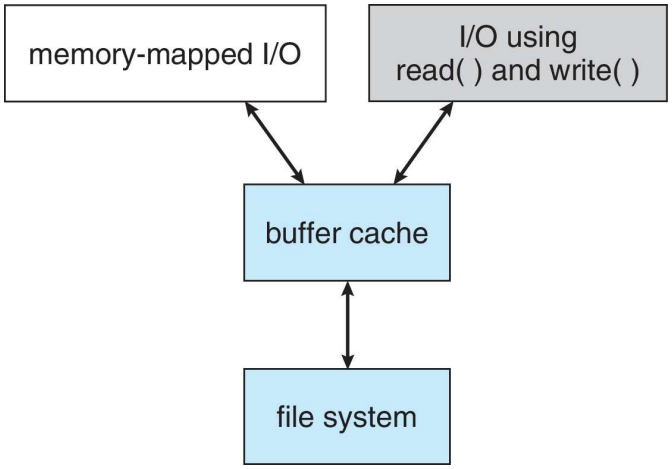
- Linked list – Some efficient modifications
  - Grouping – A block stores a list of `n` addresses. The first n-1 blocks are free. The last block has a pointer to another group of `n` addresses
  - Counting – A block stores a value `n` to signify "the next n blocks are free"
- Freeing blocks generally does not require erasing the existing data. This would use I/O unnecessarily
  - File systems will generally inform the device the block is free. HDDs and NVMs will erase the data either immediately, during a quiet period, or before the block is used again.

Efficiency and Performance

- Disks represent the most significant bottleneck in system performance (even when using NVMs). The following are important considerations.
  - Unix spreads inodes across the volume. Files are stored close to their directory's inode. This reduces seek time
  - Keeping "last modified" and "last accessed" dates can tell the OS if the file needs to be loaded into memory or to be backed up to tertiary storage
  - Pointer size – 32-bit pointers use less space but limits file size to 4GB. 64-bit pointers use more space but allow for larger files.
    - (A device capable of holding 2^{128} bytes at the atomic level should weigh about 272 trillion kilograms)

Efficiency and Performance

- Disks represent the most significant bottleneck in system performance (even when using NVMs). The following are important considerations.
  - Use a cache both on the disk and in memory. Most operating systems combine process and file data cache in a **unified virtual memory**



Recovery

- Crashes and bugs can create inconsistencies on disk. A file system must detect and correct these issues
- A file system can scan the metadata on each file to verify it makes sense.
- This can take minutes to hours and should only occur during boot time
- A status bit can indicate whether or not the entire file system is in flux or up to date. If the system reboots suddenly, the file system is still marked as "in flux" triggering a full **consistency checker** such as the `fsck` tool in UNIX/Linux

Recovery

- Drawing from database theory, some file systems are **log-based transaction-oriented** (or **journaling**) – Maintain a running log of unconfirmed "transactions" that can be re-played in the event of an interruption
- The transaction log may be stored in a separate section of the file system or on a different device entirely
- A log also allows the operating system to write data asynchronously. The operating system does not need to sit and wait for writes to be confirmed before moving on to the next task

Recovery

- Regularly scheduled backups can protect the file system
- A typical backup schedule (e.g. weekly or monthly)
  - Day 1 – Copy all files to a backup medium. This is a **full backup**
  - Day 2 – Compare a file's last backup date with the last modified date. If the file has changed, back it up. This is an **incremental backup**
  - Continue this process up to day N. Then run a full backup again
  - Watch out for daylight savings time

Recovery

- Regularly scheduled backups can protect the file system
- To restore a file system
  - Restore the files from the full backup
  - Restore each incremental backup from every day
- Instead of an incremental backup, it is possible to do a **differential (or cumulative) backup**. This is backing up any file that differs from the full backup. When a restore is required, you must only restore the full backup and the latest differential backup
- **Cold storage backup** – Keeping a long-term backup of your data off-site and disconnected. Useful to protect against environmental hazards or malicious attacks


1878
Western
UNIVERSITY · CANADA