Operating Systems
[14. File-System Implementation]

Chung-Wei Lin

cwlin@csie.ntu.edu.tw

CSIE Department

National Taiwan University

From Operating System Concepts (10th Edition)

Objectives

- Describe the details of implementing local file systems and directory structures
- Discuss block allocation and free-block algorithms and trade-offs
- Explore file system efficiency and performance issues
- Look at recovery from file system failures
- Describe the WAFL file system as a concrete example

Outline

- [ ] **File-System Structure**
- [ ] File-System Operations
- [ ] Directory Implementation
- [ ] Allocation Methods
- [ ] Free-Space Management
- [ ] Efficiency and Performance
- [ ] Recovery
- [ ] Example: WAFL File System

File-System Structure

- To improve I/O efficiency, I/O transfers between memory and mass storage are performed in units of **blocks**
  - Each block on a hard disk drive has one or more sectors
  - Depending on the disk drive, sector size is usually 512 or 4,096 bytes
- **File systems** provide access to the storage device by allowing data to be stored, located, and retrieved easily
  - Define a file and its attributes, the operations allowed on a file, and the directory structure for organizing files
    - How the file system should look to the user
  - Create algorithms and data structures to map the logical file system onto the physical secondary-storage devices

Layered File Systems (1/3)

Application Programs
Logical File System
File-Organization Module
Basic File System
I/O control
Devices

- Devices
- I/O control
  - Device drivers and interrupt handlers to transfer information between the main memory and the disk system
  - A device driver, as a translator, usually writes specific bit patterns to special locations in the I/O controller's memory
    - Its input consists of high-level commands, such as "retrieve block 123"
    - Its output consists of low-level, hardware-specific instructions that are used by the hardware controller

Layered File Systems (2/3)

Application Programs
Logical File System
File-Organization Module
Basic File System
I/O control
Devices

- Basic file system (Linux block I/O subsystem)
  - Need only to issue generic commands to the appropriate device driver to read and write blocks on the storage device
    - Issue commands to the drive based on logical block addresses
  - Also manage the memory buffers and caches that hold various filesystem, directory, and data blocks
    - A block in the buffer is allocated before the transfer of a mass storage block can occur
    - When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete
    - Caches are used to hold frequently used file-system metadata to improve performance

Layered File Systems (3/3)

Application Programs
Logical File System
File-Organization Module
Basic File System
I/O control
Devices

- File-organization module
  - Know about files and their logical blocks
    - Each file's logical blocks are numbered from 0 (or 1) through N
  - Also include the free-space manager
    - Track unallocated blocks
    - Provide these blocks to the file-organization module when requested
- Logical file system
  - Manage metadata which includes all of the file-system structure except the actual data (or contents of the files)
  - Manage the directory structure
  - Maintain file structure via **file-control blocks (FCBs)** (**inodes** in UNIX)
    - Information about the file, including ownership, permissions, and location of the file contents
- Application programs

Typical File-Control Block

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

Advantage and Disadvantage

- Advantage of layered file systems
  - Duplication of code is minimized
    - The I/O control and sometimes the basic file-system code can be used by multiple file systems
    - Each file system can then have its own logical file-system and file-organization modules
- Disadvantage of layered file systems
  - More operating-system overhead, which may result in decreased performance
- A major challenge in designing new systems
  - The use of layering, including the decision about how many layers to use and what each layer should do

File Systems in Use

- Most CD-ROMs are written in the ISO 9660 format
- UNIX uses the UNIX file system (UFS), which is based on the Berkeley Fast File System (FFS)
- Windows supports disk file-system formats of FAT, FAT32, and NTFS, as well as CD-ROM and DVD file-system formats
- The standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4
  - Although Linux supports over 130 different file systems
- More are coming
  - ZFS, GoogleFS, Oracle ASM, FUSE

Outline

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

Structures on File Systems (1/2)

- A **boot control block** (per volume) contains information needed to boot an operating system from that volume
  - If the disk does not contain an operating system, this block can be empty
  - It is typically the first block of a volume
  - **Boot block** in UFS and **partition boot sector** in NTFS
- A **volume control block** (per volume) contains volume details
  - The number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers
  - **Superblock** in UFS and stored **master file table** in NTFS

Structures on File Systems (2/2)

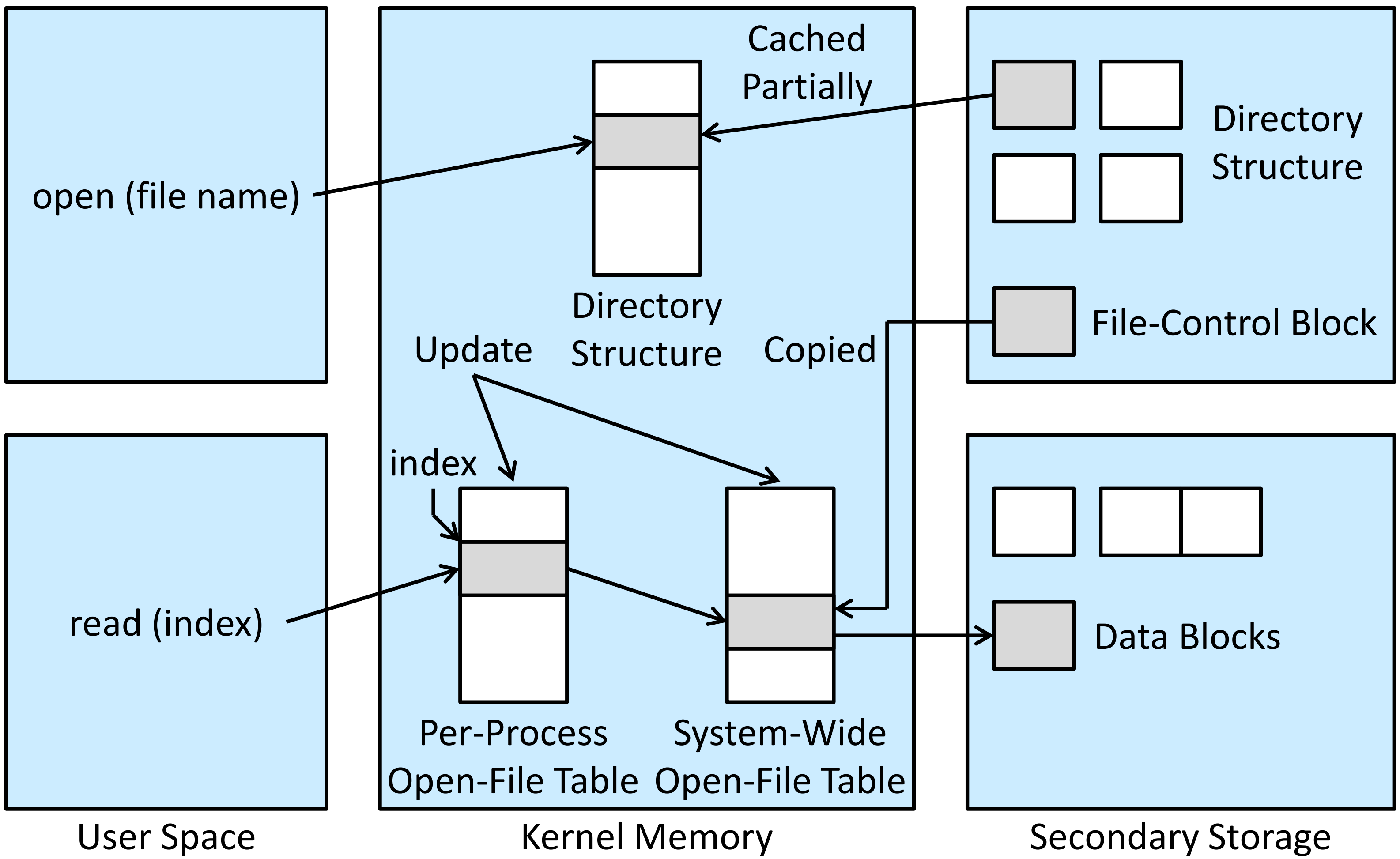
- A directory structure (per file system) organizes the files
  - In UFS, this includes file names and associated inode numbers
  - In NTFS, it is stored in the master file table
- A per-file FCB contains many details about the file
  - It has a unique identifier number to allow association with a directory entry
  - In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file

In-Memory File-System Structures

- An in-memory **mount table** contains information about each mounted volume
- An in-memory directory-structure cache holds the directory information of recently accessed directories
- The **system-wide open-file table** contains a copy of the FCB of each open file
  - As well as other information
- The **per-process open-file table** contains pointers to the appropriate entries in the system-wide open-file table
  - As well as other information
- Buffers hold file-system blocks when they are being read from or written to a file system

File Open and File Read

- **`open()`** returns a **file handle** (**descriptor**) for subsequent use



Outline

- File-System Structure
- File-System Operations
- **<u>Directory Implementation</u>**
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

Directory Implementation

- **Linear list** of file names with pointers to the data blocks
  - Simple to program
  - Time-consuming (linear search) to execute
    - A sorted list allows a binary search and decreases the average search time
- **Hash table**: linear list with a hash data structure
  - Decrease the directory search time
  - Need some provision for **collisions**
    - Situations where two file names hash to the same location
  - Have difficulties with its generally fixed size and the dependence of the hash function on that size
    - Alternatively, we can use a chained-overflow hash table

Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- **Allocation Methods**
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation
  - Performance
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

Allocation Methods

- How to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly

Outline

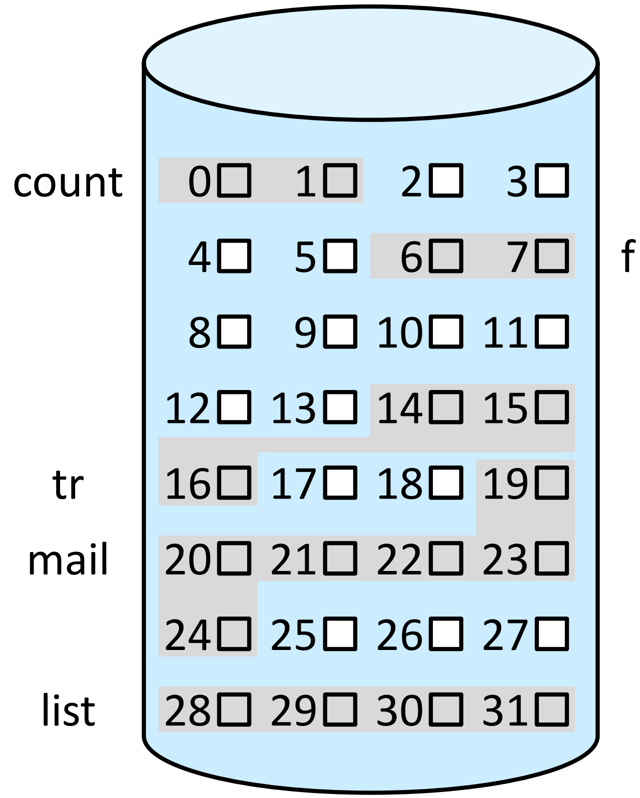
- File-System Structure
- File-System Operations
- Directory Implementation
- **Allocation Methods**
  - **Contiguous Allocation**
  - Linked Allocation
  - Indexed Allocation
  - Performance
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

Contiguous Allocation

- Each file occupy a set of contiguous blocks on the device
  - Easy implementation
  - External fragmentation
    - Dynamic storage-allocation problem (Section 9.2)
    - Free-space management (Section 14.5)
  - One strategy: copying an entire file system onto another device and then coping back
    - Compaction
    - Off-line (during down time)
    - On-line

Directory

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



Extent-Based Systems

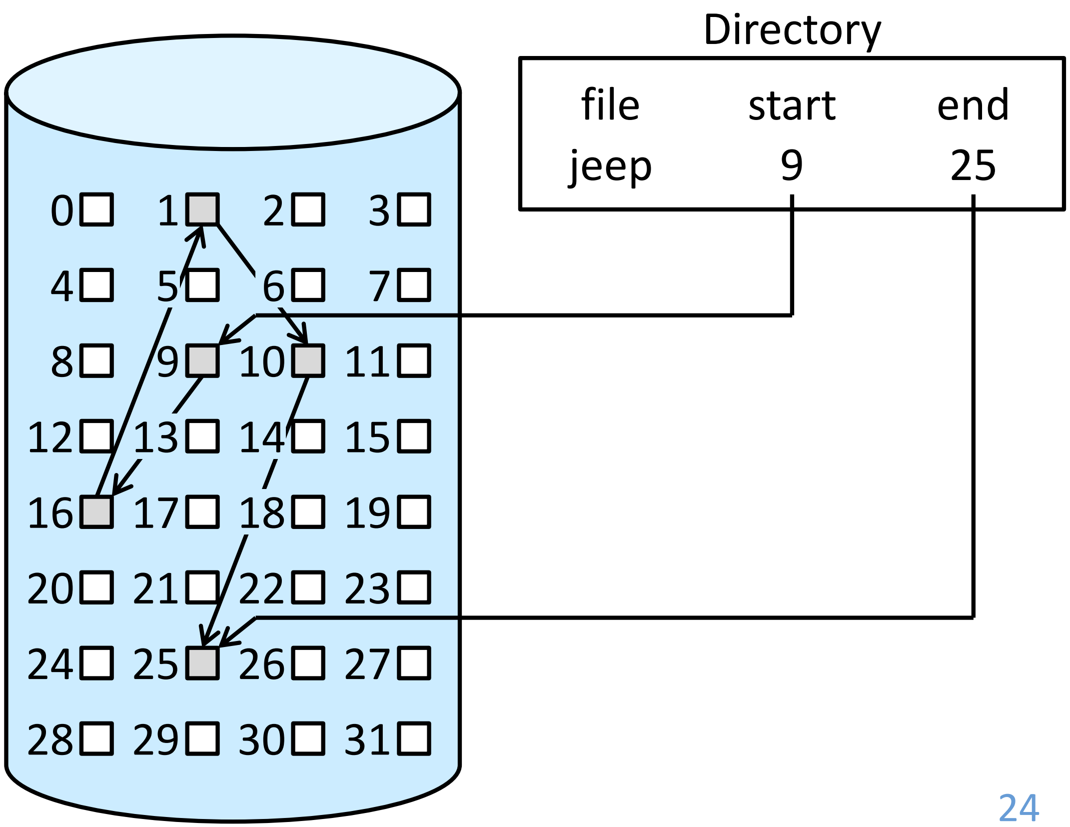
- How much space is needed for a file?
- How to make the file larger in place?
  - Terminate the program, allocate more space, and run the program again
    - These repeated runs may be costly
    - To prevent them, the user will normally overestimate the amount of space needed
  - Find a larger hole, copy the contents of the file to the new space, and release the previous space
- If that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added
  - The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent

Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- **<u>Allocation Methods</u>**
  - Contiguous Allocation
  - **<u>Linked Allocation</u>**
  - Indexed Allocation
  - Performance
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

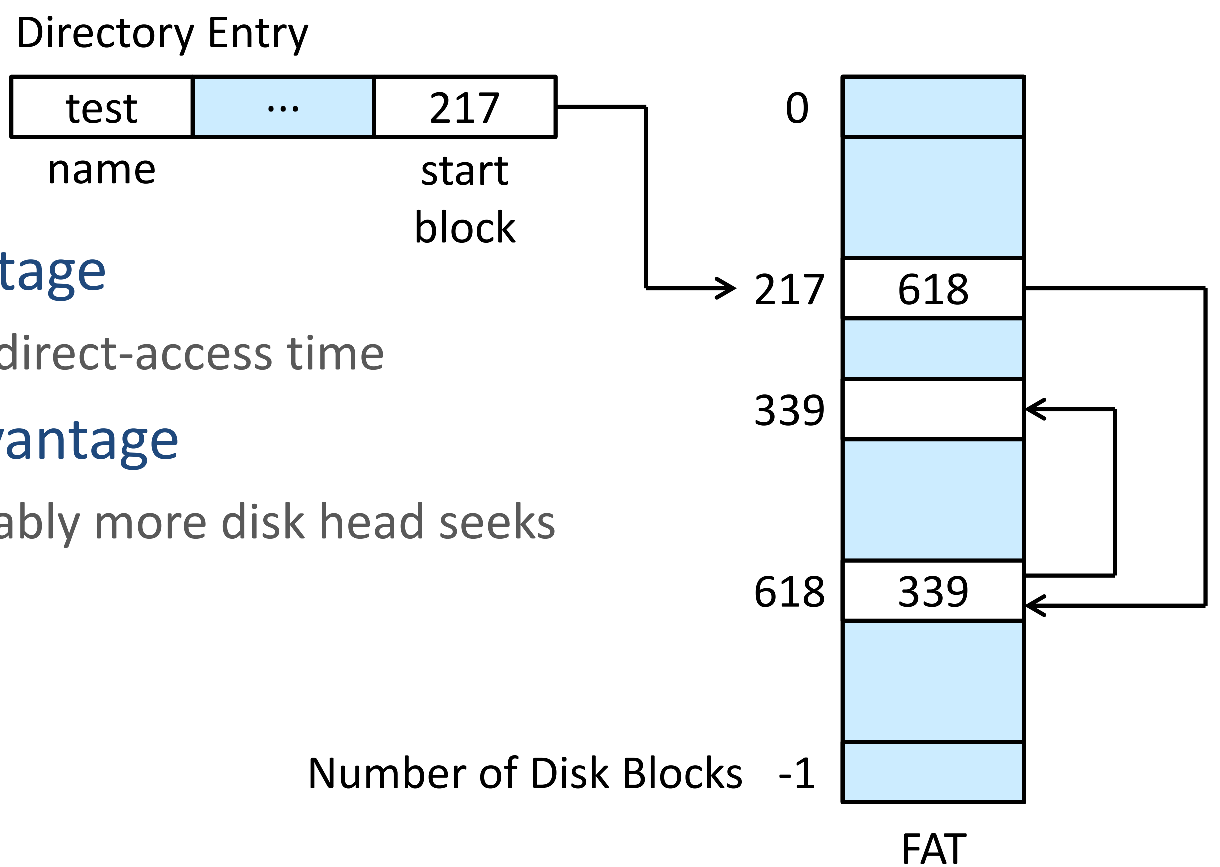
Linked Allocation

- Each file is a linked list of storage blocks which may be scattered anywhere on the device
  - The directory contains a pointer to the first and last blocks of the file
  - Each block contains a pointer to next block
- Advantages
  - No external fragmentation
  - No compaction needed
- Disadvantages
  - Inefficiency of finding the i-th block of a file
  - More space for pointers
    - Solution: clusters
  - Reliability



File-Allocation Table (FAT)

- A section of storage at the beginning of each volume is set aside to contain the table
  - The table has one entry for each block and is indexed by block number



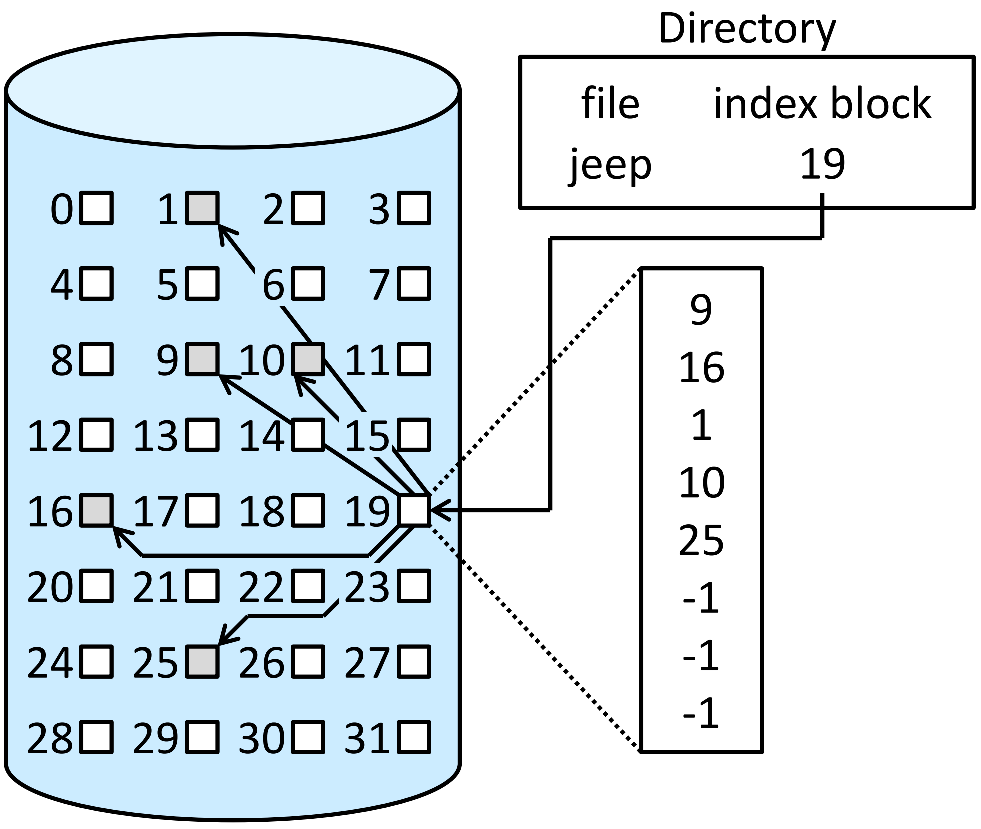
- Advantage
  - Less direct-access time
- Disadvantage
  - Probably more disk head seeks

Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- **Allocation Methods**
  - Contiguous Allocation
  - Linked Allocation
  - **Indexed Allocation**
  - Performance
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

Indexed Allocation

- Bring all the pointers together into the **index block**
  - In the absence of a FAT, linked allocation cannot support efficient direct access
- Advantages
  - No external fragmentation
  - No compaction needed
- Disadvantage
  - More space than linked allocation

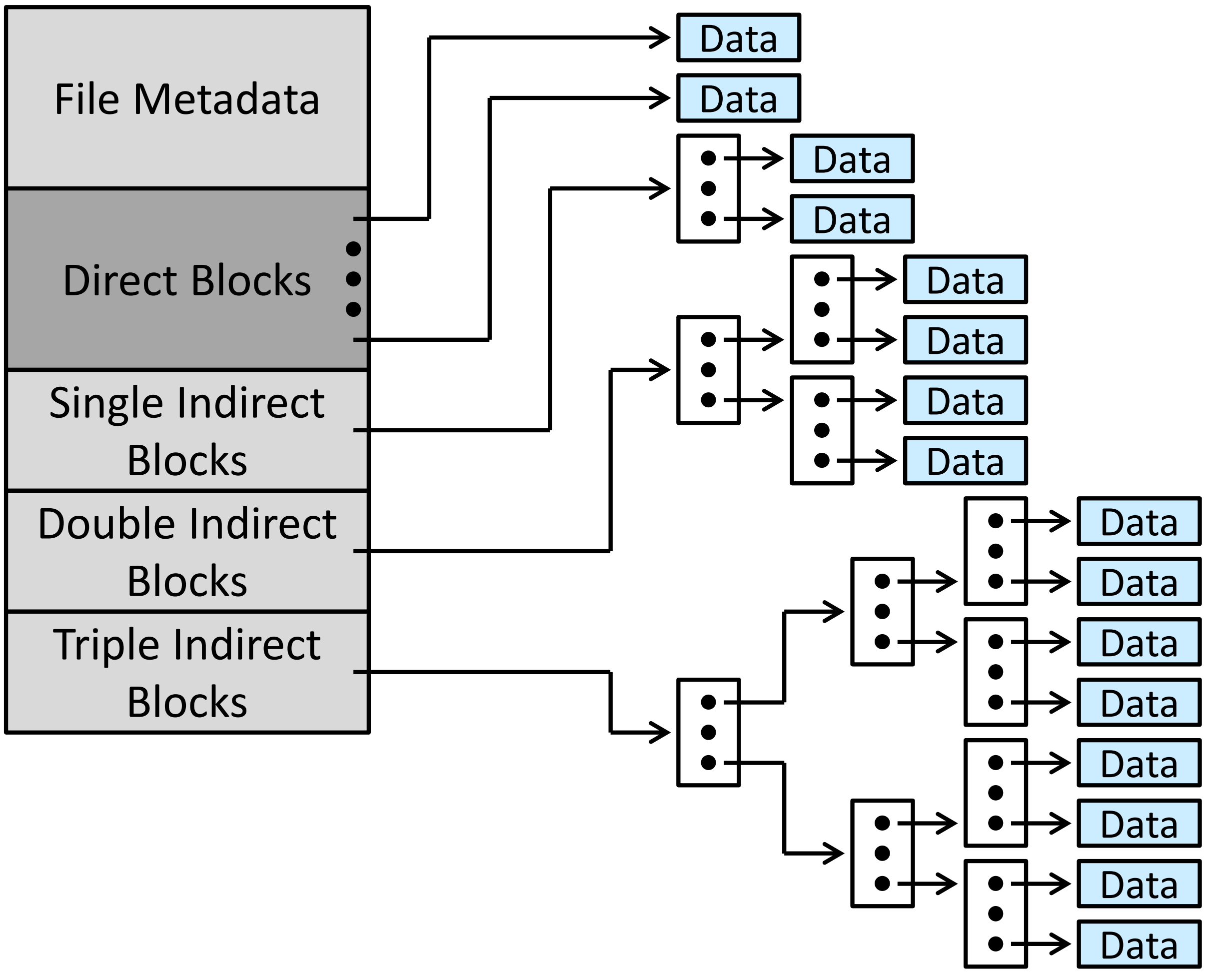


Schemes of Index Blocks (1/2)

- Linked scheme
  - An index block is normally one storage block
  - To allow for large files, we can link together several index blocks
- Multilevel index
  - A first-level index block points to a set of second-level index blocks, which in turn point to the file blocks
  - This approach could be continued to a third or fourth level, depending on the desired maximum file size
  - Example
    - With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block
    - Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB

Schemes of Index Blocks (2/2)

- Combined scheme (in UNIX-based file systems)



Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- **<u>Allocation Methods</u>**
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation
  - **<u>Performance</u>**
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

Performance

- Goals: storage efficiency and data-block access times
- A system with mostly sequential access should not use the same method as a system with mostly direct (random) access
  - Contiguous allocation requires only one access to get a block, for any type of access
  - Linked allocation should not be used for direct access
- Indexed allocation is more complex
  - Depend on the index structure, the file size, and the block position
- Mix of contiguous, linked, and indexed allocations
- For NVM, different algorithms and optimizations are needed
  - Reduce the instruction count and overall path between the storage device and application access to the data

Outline

- File-System Structure, File-System Operations
- Directory Implementation, Allocation Methods
- **Free-Space Management**
  - Bit Vector
  - Linked List
  - Grouping
  - Counting
  - Space Maps
  - TRIMing Unused Blocks
- Efficiency and Performance
- Recovery
- Example: WAFL File System

Free-Space Management

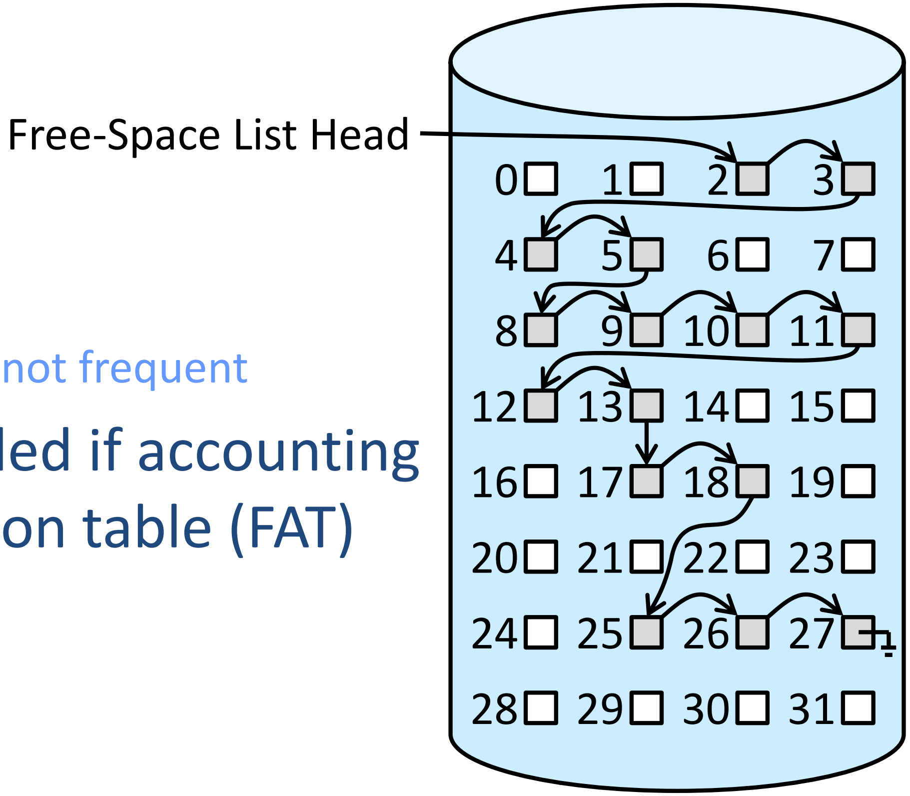
- To keep track of free disk space, the system maintains a <u>**free-space list**</u> which records all free device blocks
  - To create a file, we search the free-space list for the required amount of space and allocate that space to the new file
    - This space is then removed from the free-space list
  - When a file is deleted, its space is added to the free-space list
- The free-space list, despite its name, is not necessarily implemented as a "list"

Bit Vector (or Bitmap)

- If a block is free, its bit is 1; if the block is allocated, its bit is 0
- Calculation of the first free block
  - (# of bits per word) * (# of first 0-value words) + offset of first 1 bit
    - Bit-manipulation instructions can be used effectively
- Advantage
  - Simple
- Disadvantage
  - Inefficient unless the entire vector is kept in main memory
  - Example
    - 1-TB (2^{40} bytes) disk with 4-KB (2^{12} bytes) blocks
    - The number of blocks = $2^{40}/2^{12} = 2^{28}$
    - The size of bit vector = 2^{28} bits (32 MB)
    - If we use 4-block clusters instead of blocks, we still need 8 MB

Linked List

- Link together all the free blocks
- Keep a pointer to the first free block in a special location in the file system and cache it in memory
- Advantage
  - No waste of space
- Disadvantage
  - Inefficient to traverse the list
    - Fortunately, traversing the list is not frequent
- No separate method is needed if accounting free-block into a file-allocation table (FAT)



Grouping

- A modification of the linked-list (free-list) approach
  - Stores the addresses of n free blocks in the first free block
    - The first n-1 of these blocks are actually free
    - The last block contains the addresses of another n free blocks, and so on
- Advantage
  - The addresses of a large number of free blocks can be found faster than the standard linked-list approach

Counting

- Each entry consists of an address and a count
  - Keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block
    - Not keeping a list of n free block addresses
    - Similar to the extent method of allocating blocks
- Advantage
  - Shorter list, as long as the count is generally greater than 1
    - Several contiguous blocks may be allocated or freed simultaneously, particularly with the contiguous-allocation algorithm or through clustering
- These entries can be stored in a balanced tree for efficient lookup, insertion, and deletion
  - Rather than a linked list

Space Maps

- Oracle's **ZFS** file system was designed to encompass huge numbers of files, directories, and even file systems
  - On these scales, metadata I/O can have a large performance impact
    - Need to control the size of data structures and minimize the metadata I/O
- Divide the space into **metaslabs** of manageable size
  - A given volume may contain hundreds of metaslabs
  - Each metaslab has an associated space map
- Use log-structured file-system techniques
  - A space map is a log of all block activity, in time order, in counting format
  - When ZFS decides to allocate or free space from a metaslab
    - Load the space map into memory in a balanced-tree structure indexed by offset
    - Replay the log into that structure

TRIMing Unused Blocks

- Some storage devices (e.g., HDD) allowing blocks to be overwritten need only the free list for managing free space
  - A block does not need to be treated specially when freed
    - A freed block typically keeps its data until the data are overwritten
- Some storage devices (e.g., NVM) must be erased before they can again be written to
  - A new mechanism is needed to allow the file system to inform the storage device that a page is free and can be considered for erasure
    - For ATA-attached drives, it is TRIM
    - For NVMe-based storage, it is the **`unallocate`** command
  - With them, the garbage collection and erase steps can occur before the device is nearly full

Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- **<u>Efficiency and Performance</u>**
  - Efficiency
  - Performance
- Recovery
- Example: WAFL File System

Efficiency and Performance

- Disks tend to be a major bottleneck in system performance
  - They are the slowest main computer component
- Even NVM devices are slow compared with CPU and main memory

Some Factors

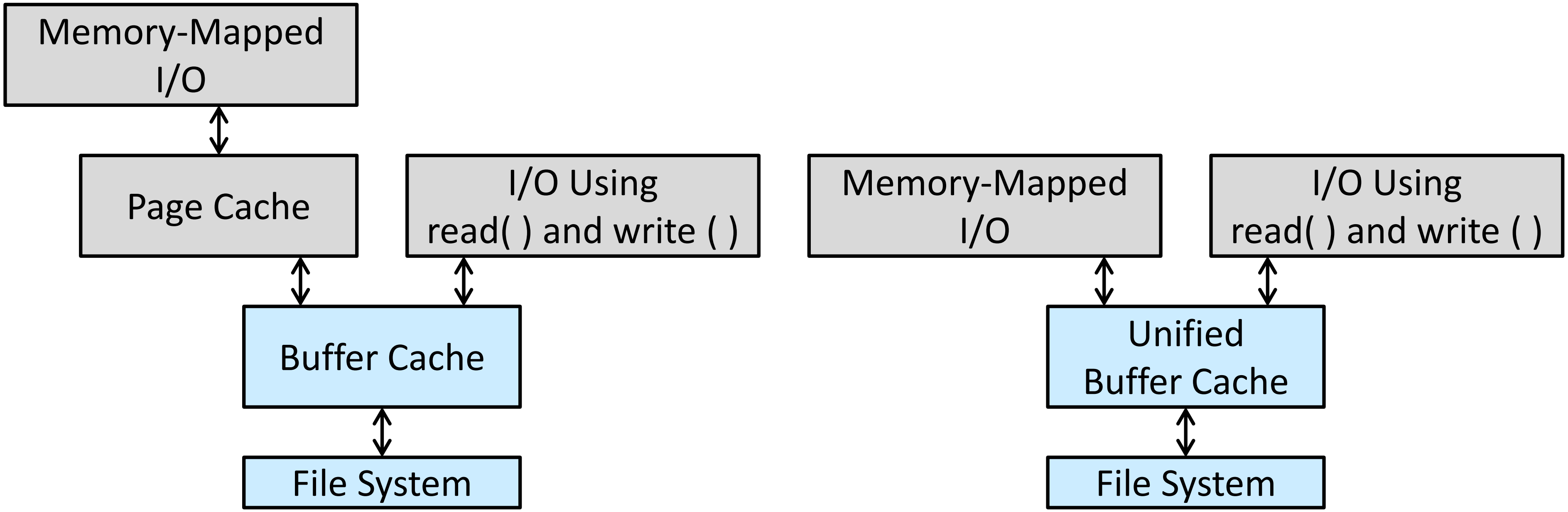
- Allocation and directory algorithms
  - Example: UNIX tries to keep a file's data blocks near that file's inode block to reduce seek time
- Types of data kept in a file's directory
  - What if keeping the "last access date" of a file?
- Size of the pointers used to access data
- Fixed-size or varying-size (dynamically-allocated) data structures

Buffer Cache and Page Cache

- Some systems maintain a separate section of main memory for a **buffer cache**
  - Keep blocks which are assumed to be used again shortly
- Other systems cache file data using a **page cache**
  - Use virtual memory techniques to cache file data as pages rather than as file-system-oriented blocks
    - More efficient than caching through physical disk blocks
  - Memory-mapped I/O uses a page cache
  - Routine I/O through the file system uses a buffer cache
- **Unified virtual memory**
  - Several systems, including Solaris, Linux, and Windows, use page caching to cache both process pages and file data

Unified Buffer Cache

- Use the same page cache for both memory-mapped I/O mapping and file system I/O to avoid **double caching**
  - Double caching
    - Waste memory
    - Waste significant CPU and I/O cycles
    - Result in corrupt files with inconsistencies between the two caches



Synchronous and Asynchronous Writes

- Synchronous write
  - The writes are not buffered
  - The calling routine must wait for the data to reach the drive
- Asynchronous write
  - The writes are stored in the cache
  - The calling routine proceeds
- Most writes are asynchronous
  - However, metadata writes can be synchronous
- Operating systems frequently allows a process to request writes to be synchronous
  - Example: databases use this feature for atomic transactions to assure that data reach stable storage in the required order

Free-Behind and Read-Ahead

- A file being read or written sequentially should not have its pages replaced in least recently used (LRU) order
  - The most recently used page will be used last, or perhaps never again
- Sequential access can be optimized
  - **Free-behind** removes a page from the buffer as soon as the next page is requested
    - The previous pages are not likely to be used again and waste buffer space
  - **Read-ahead** reads and caches a requested page and several subsequent pages
    - These pages are likely to be requested after the current page is processed

Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- **Recovery**
  - Consistency Checking
  - Log-Structured File Systems
  - Other Solutions
  - Backup an Restore
- Example: WAFL File System

Consistency Checking

- Consistency checking
  - Compare the data in the directory structure and other metadata with the state on storage
  - Try to fix any inconsistency it finds
- The allocation and free-space-management algorithms dictate
  - What types of problems the checker can find
  - How successful it will be in fixing them

Log-Structured File Systems

- **Log-based transaction-oriented** (or **journaling**) file systems
  - All metadata changes are written sequentially to a log
    - The log may be in a separate section of the file system or even on a separate storage device
  - Once the changes are written (committed) to this log, the system call can return to the user process
  - These log entries are replayed across the actual file-system structures
  - When the file-system structures are modified, the **transaction** is removed from the log
    - Transaction: each set of operations for performing a specific task
- If the system crashes, any transactions which were not completed to the file system must now be completed
  - The only problem occurs when a transaction was not committed before the system crashed

Other Solutions

- **Snapshot**: a view of the file system at a specific point in time
  - Some systems let a transaction write all data and metadata changes to new blocks
    - They never overwrite blocks with new data
  - When the transaction is complete, the metadata structures that pointed to the old blocks are updated to point to the new blocks
  - The file system can then remove the old pointers and the old blocks and make them available for reuse
    - If the old pointers and the old blocks are kept, a snapshot is created
- ZFS further provides checksumming of all metadata and data blocks

Backup an Restore

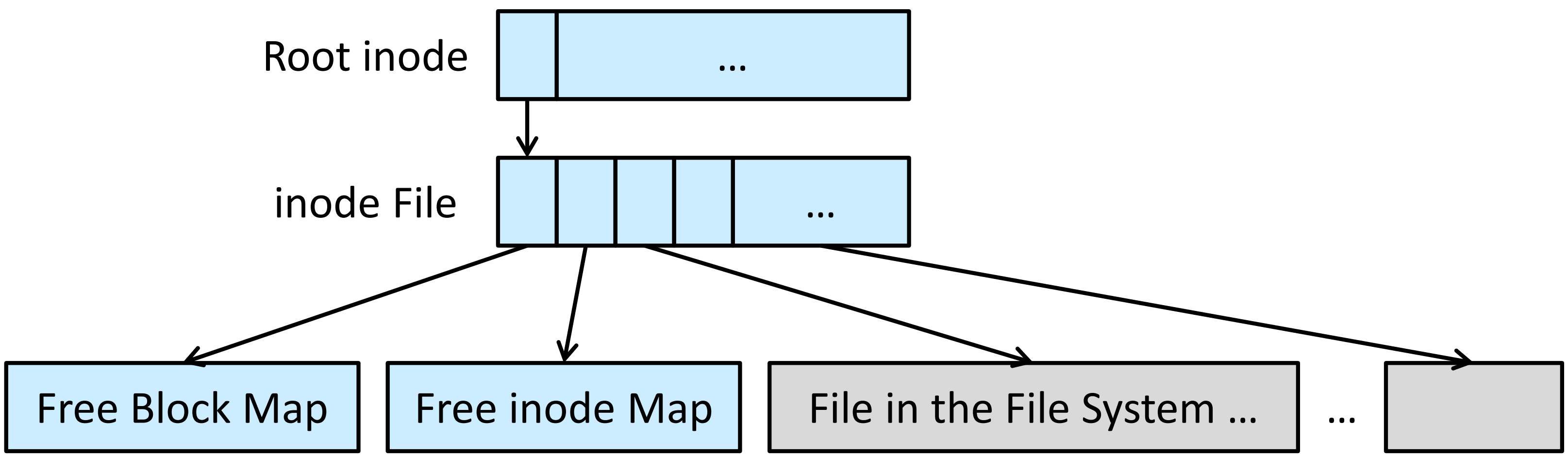
- System programs can be used to **back up** data from one storage device to another
- Recovery from the loss of an individual file or an entire device may then be a matter of **restoring** the data from backup
- A typical backup schedule
  - A **full backup** + multiple **incremental backup**

Outline

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

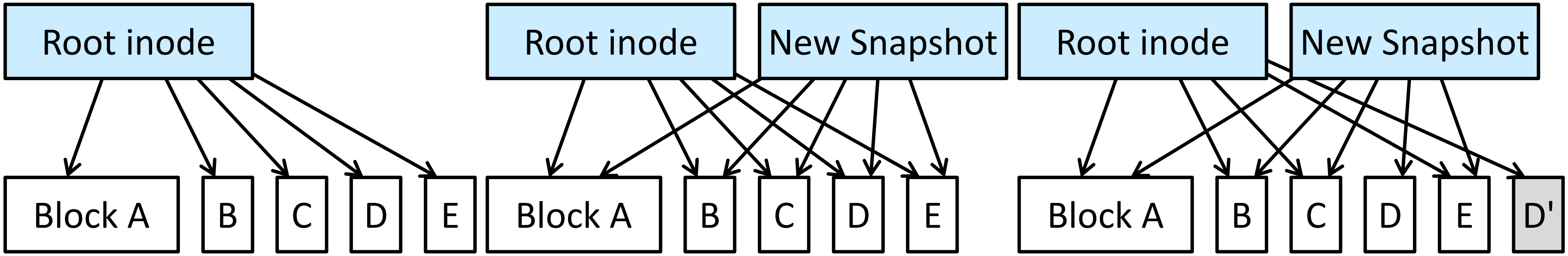
Example: WAFL File System

- The **write-anywhere file layout** (WAFL) is a file system optimized for random writes
  - Used exclusively on network file servers produced by NetApp
  - Meant for use as a distributed file system
- A file server may see a very large demand for random reads and an even larger demand for random writes
  - The NFS and CIFS protocols cache data from read operations, so writes are of the greatest concern to file-server creators



Snapshots in WAFL

- **Many snapshots can exist simultaneously**
  - The snapshot facility is useful for backups, testing, versioning
  - The snapshot facility is also very efficient
    - Not even require that copy-on-write copies be taken before the block is modified
- **Other features**
  - **Clone**: read-write snapshots
  - **Replication**: the duplication and synchronization of a set of data over a network to another system



Before a snapshot | After a snapshot and before any block changes | After block D has changed to D'

Objectives

- ❑ Describe the details of implementing local file systems and directory structures
- ❑ Discuss block allocation and free-block algorithms and trade-offs
- ❑ Explore file system efficiency and performance issues
- ❑ Look at recovery from file system failures
- ❑ Describe the WAFL file system as a concrete example

Q&A