운영체제의 기초: File System

2023년 6월 13, 15일

홍 성 수
sshong@redwood.snu.ac.kr
SNU RTOSLab 지도 교수
서울대학교 전기정보공학부 교수

Seoul National University
RTOS Lab

Agenda

I. Understanding File System

What is File System? (1)

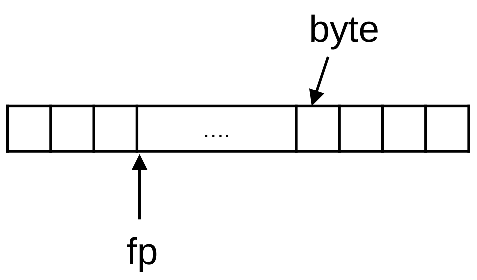
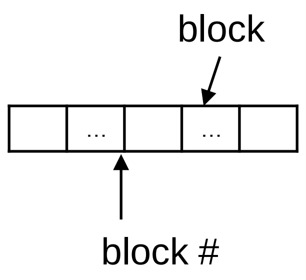
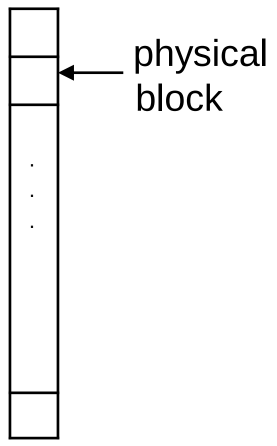
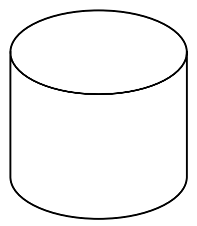
- File system
  - From user's standpoint: *Directory tree*
    - A collection of directories and files organized in a hierarchical manner
  - From OS' standpoint: *Formatted storage volume*
    - A formatted collection of disk blocks in a given storage volume
    - Disk blocks are used for storing
      - File system metadata, AKA superblock
      - File metadata, AKA inode blocks
      - File data, AKA data blocks

What is File System? (2)

- File system (cont'd)
  - Purely software-based
    - Does not require hardware support
      unlike process management and virtual memory
      - Surely exploits the hardware idiosyncrasies, though
  - Offers a *logical name space* for the system
  - Includes in-memory data structures and operations
    to them for enhanced performance

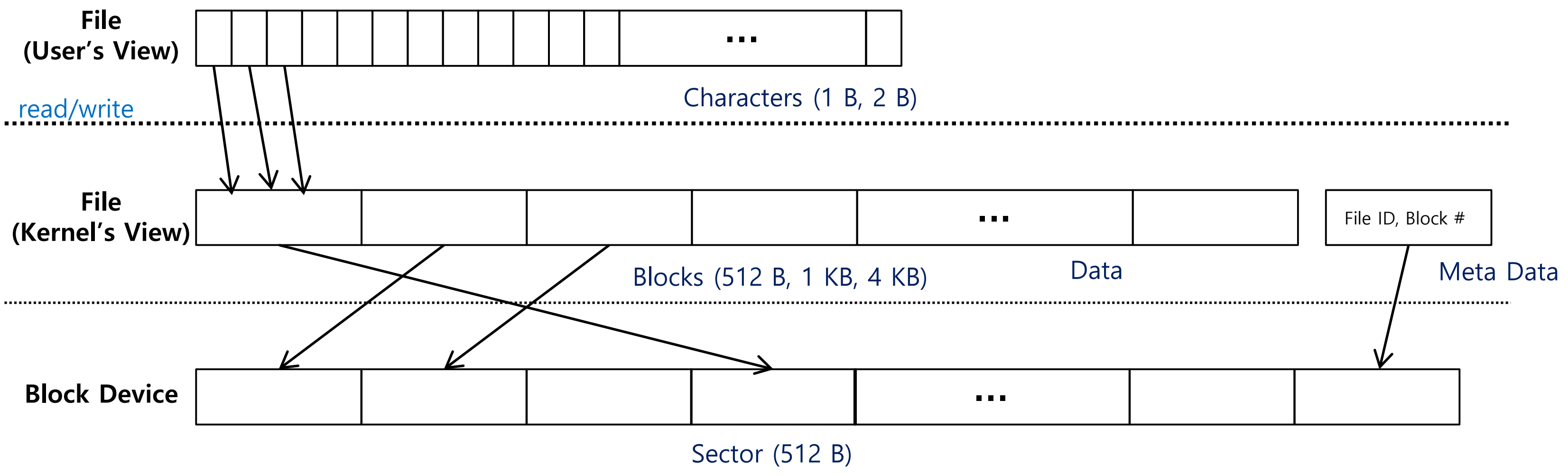
- We're going to delve into *file system implementation*
  - Consider *only* disk drives as storage for simplicity's sake

Different Views on Files (1)

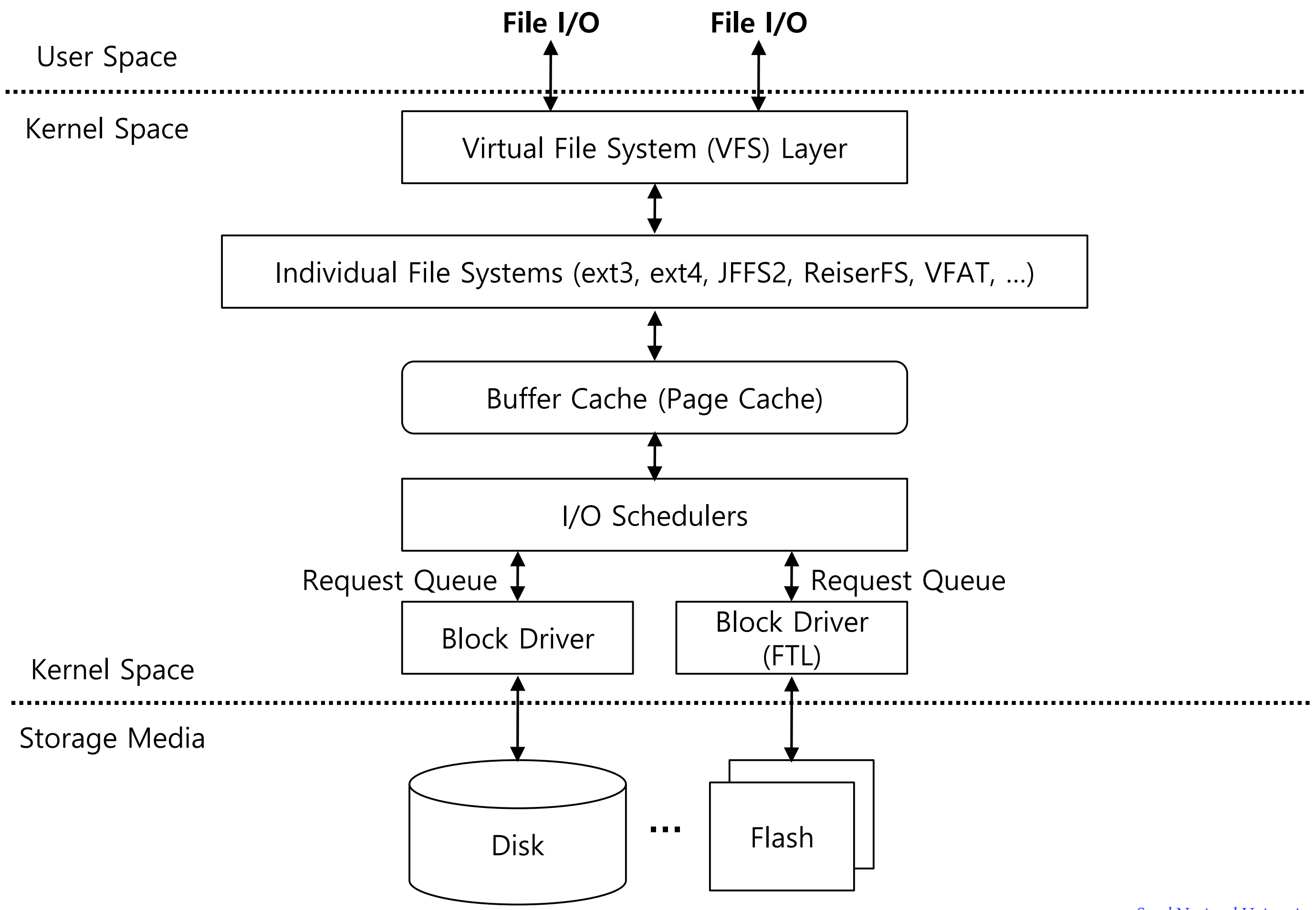
User View	*Kernel View*	*Device Driver View*	*Device View*
file name + byte offset	ino + logical block #	physical block #	drive # cylinder # head # sector #
File 	File 	File System 	Drive 

Different Views on Files (2)

- Requires mappings between two adjacent layers



Layered Implementation in Linux



Design Goal for File System

- No. 1 design goal: "*Speed up data access within file*"
  - Average access time must be low for all kinds of "*data access patterns*"

- Types of data access patterns
  1. Sequential access
  2. Random access
  3. Keyed access

1. Sequential Access

- Behavior
  - Programs read or write data within a file in order, one data block after another
    - To process information sequentially
  - This is by far the most common mode

- Examples
  - Text editor writes out a new file
  - Compiler scans and compiles the source file

2. Random Access

- Behavior
  - Programs access data blocks in a file directly in random order, without passing through its predecessors
    - To process information according to the order determined by the algorithm of the program
    - The address of a block is called the *block index* or *number*

- Examples
  - Page set for demand paging
  - Indexed data structures like binary search tree
  - Databases

3. Keyed Access

- Behavior
  - Search for data blocks with particular values
  - Application-specific
    - Used to be supported by old IBM OS
    - Usually not provided by the modern OS
- Examples
  - Hash table
  - Associative database
  - Dictionary

Topics to be Covered

- File system internals
  - Structure of a file
  - Disk block management
    - Disk block allocation
    - Free block management
  - Performance enhancement
    - Buffer cache, page cache, inode cache
    - Seek optimization
  - Reliability
    - **`fsck`** and journaling for crash consistency (recovery)
  - Evolution of Unix file systems
  - Disk scheduling

II. File Structures

Design Considerations (1)

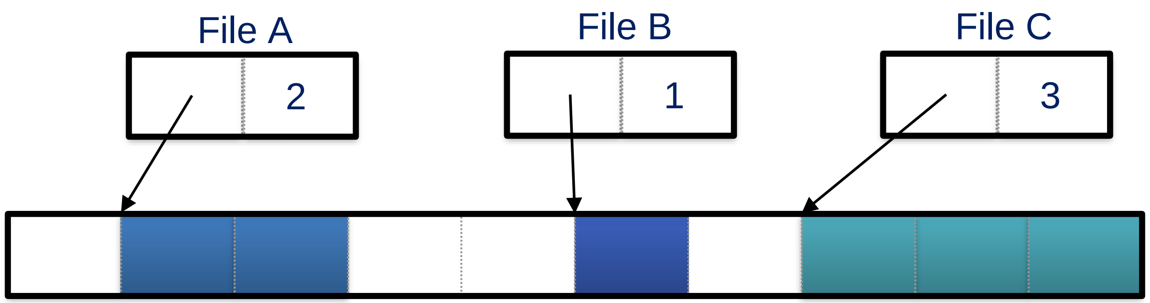
- Things to think about in designing a file structure
  - Primary design goal
    - Average access time must be low
      for both sequential and random accesses
  - Usage patterns
    - Most files are small but large in number
    - Much of the disk is allocated to large files
    - Many of the I/O operations are made to large files
  - Other design goals
    - Per-file cost must be low
    - But large files must have good performance

Design Considerations (2)

- Possible forms of file structure
  1. Contiguous files
  2. Linked files
  3. Indexed files

1. Contiguous Files (1)

- Key structural properties
  - Simply keeps the first block index and file size
- Disk layout example
  - How do you perform file operations (e.g., read block 3)?
  - How do you increase the file's size?



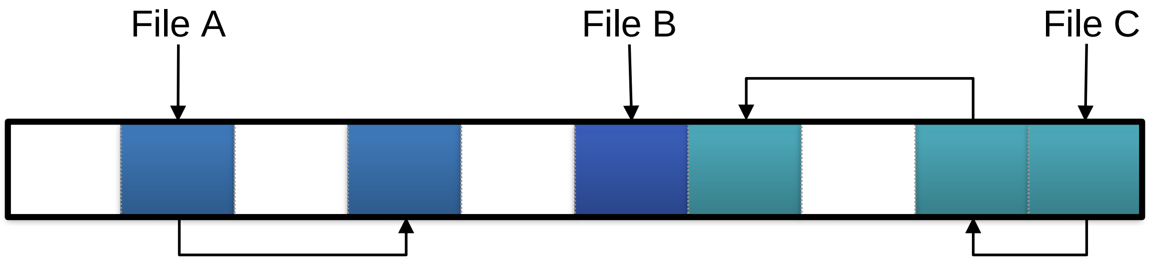
1. Contiguous Files (2)

- Pros
  - Simple file structure
  - Both sequential and random accesses are simple and easy
  - Optimized for seeks (just a few seeks)
- Cons
  - Horrible fragmentation will make large files impossible
  - Hard to predict the needed block size at file creation time
- Example: IBM OS/360

2. Linked Files (1)

- Key structural properties
  - Keeps a linked list of the allocated data blocks in a file
    - Each data block keeps the pointer to the next
    - File descriptor contains the pointer to the first data block

- Disk layout example
  - How do you perform file operations (e.g., read block 3)?
  - How do you figure out the file's size?

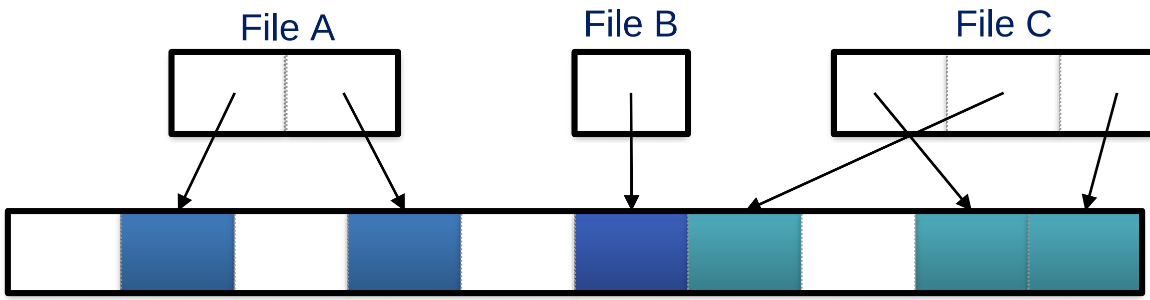


2. Linked Files (2)

- Pros
  - Files can be extended
  - No fragmentation problems
  - Sequential access is easy
    - Just chase the links
- Cons
  - Random access is virtually impossible
  - Lots of seeks, even in sequential access
- Example: TOPS-10, Alto

3. Indexed Files (1)

- Key structural properties
  - Keeps an array of pointers to all the data blocks for each file
    - The "*index array*" is created and stored in separate blocks on file creation
    - Data blocks are not allocated in the beginning
      - Dynamically allocated from a free block pool and their pointers filled in the index array
- Disk layout example



3. Indexed Files (2)

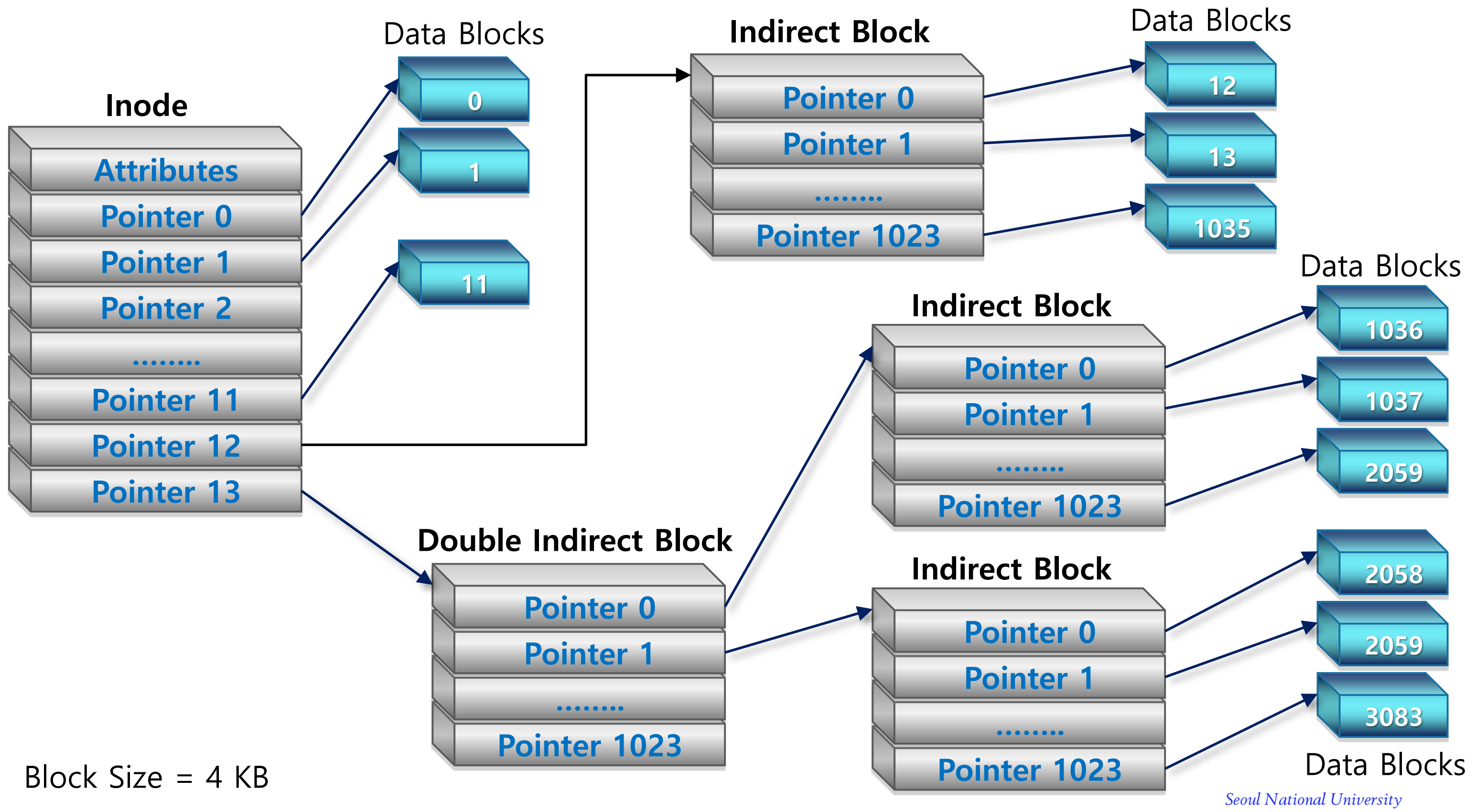
- Pros
  - Not so much space wasted by over-predicting
  - Both sequential and random accesses are easy
- Cons
  - Still have to set the maximum file size
    - Due to the static size limitation of the index array
  - There will be lots of seeks

3. Indexed Files (3)

- Example: Multi-level indexed files in 4.3 BSD
  - Inode contains 14 pointers to data blocks
    - The first 12 point to data blocks
    - The next one to an indirect index block that contains 1024 additional pointers to data blocks
    - The last one to a doubly indirect index block
  - Maximum file length is fixed, but large

3. Indexed Files (4)

- Example: Multi-level indexed files in 4.3 BSD



3. Indexed Files (5)

- Example: Multi-level indexed files in 4.3 BSD
  - Pros
    - Simple, easy to implement
    - Incremental expansion
    - Easy access to small files
  - Cons
    - Indirect mechanism lacks support for very efficient access to large files
      - Up to two extra inode accesses for each real data access
    - Block-by-block organization of the free block pool may mean that file data get spread around the disk
      - A lot of seeks

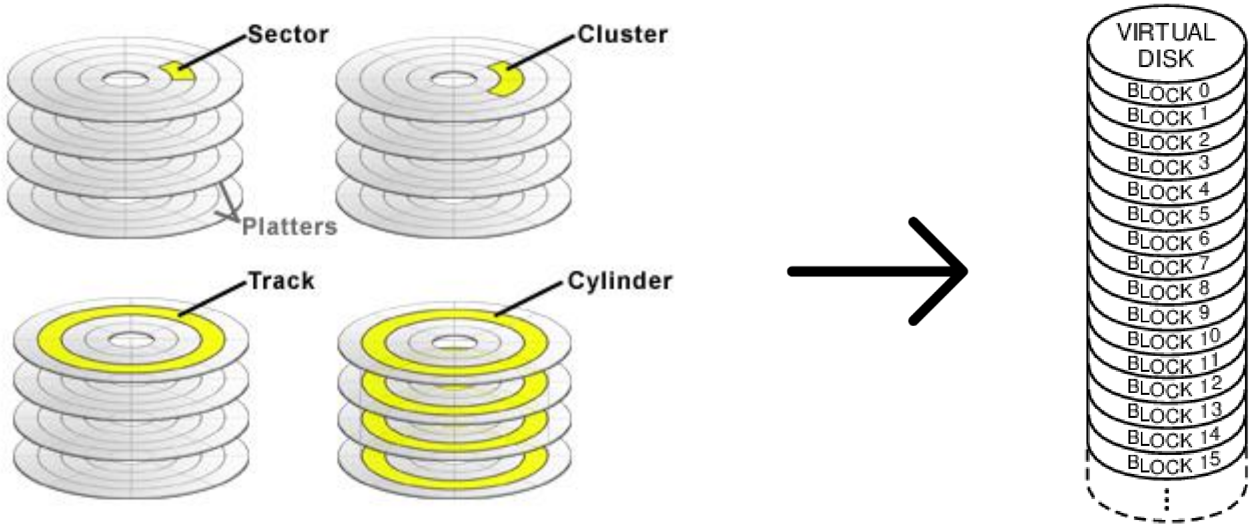
III. Disk Block Management

Design Considerations (1)

- Design goals
  - Performance
    - Average access time must be low for both sequential and random accesses
      - Low overhead in manipulating data structures
      - Less seeks per access
  - Reliability
    - Information must last safely for long period of time

Design Considerations (2)

- Logical view of a disk drive
  - A linear array of blocks
    - Map three-dimensional disk structure to the array of sectors
      - Give each sector a number from 0 up
      - One-to-one mapping from sectors to blocks



Topics to be Covered

- Disk block allocation
- Free block management

Cases for Disk Block Allocation

- Disk blocks are used to store
  - File contents
    - Stored in data blocks
  - File metadata
    - Stored in inode blocks
  - File system metadata
    - Stored in superblocks
  - Boot loader program
    - Stored in block 0, traditionally (often empty)

Data Block Allocation Policies

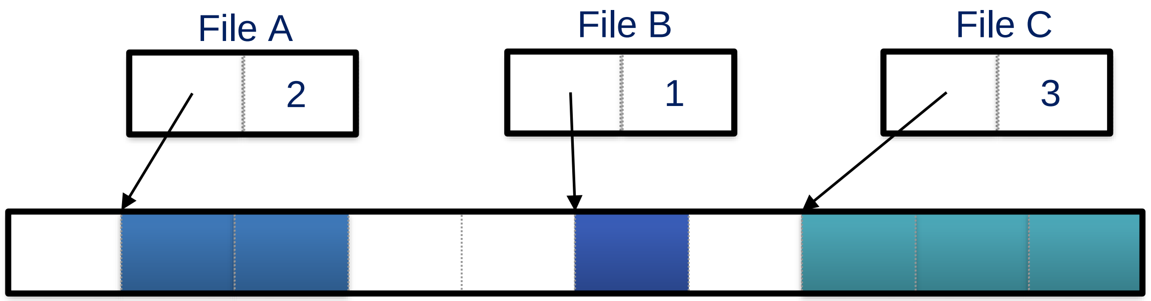
- Three possibilities
  1. Contiguous allocation
  2. Block-based allocation
  3. Extent-based allocation

1. Contiguous Allocation (1)

- Operations
  - Allocate disk space to files like segmented memory
  - Make the user specify its length and allocate all the requested space at once when creating a file
  - Keep a free list of unused areas (free blocks) of the disk
  - File descriptor contains
    - Location and size of the allocated space

1. Contiguous Allocation (2)

- Pros
  - Both sequential and random accesses are simple and easy
  - Optimized for seeks (a few seeks)
- Cons
  - Horrible fragmentation will make large files impossible
  - Hard to predict the needed block size at file creation time
- Example: IBM OS/360



2. Block-based Allocation (1)

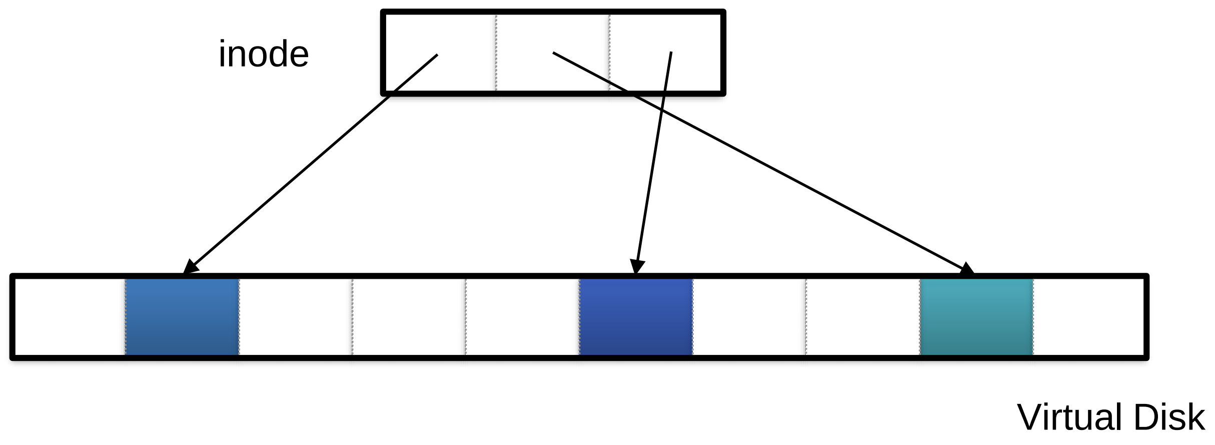
- Operations
  - Disk blocks are allocated as they are used
    - Minimal number of blocks are allocated to a file in an attempt to conserve storage space
  - When a file is extended, blocks are allocated from a free block map
    - Blocks are allocated in a random order
  - Each file must contain and maintain block allocation information, often called metadata
    - File system metadata is written synchronously to disk
      - File size changes must wait for each metadata operation to complete
      - Metadata operations significantly slow down the overall file system performance

2. Block-based Allocation (2)

- Pros
  - Efficient disk space usage
- Cons
  - Excessive seeks for sequential accesses
  - Extra disk IO to read and write the metadata

2. Block-based Allocation (3)

- Example: Traditional Unix file system (UFS)



3. Extent-based Allocation (1)

- Extent
  - A *large contiguous* area of storage reserved for a file
    - Represented as a range of block numbers
      - File stores each range compactly as a starting address-size pair, instead of storing every block number in the range

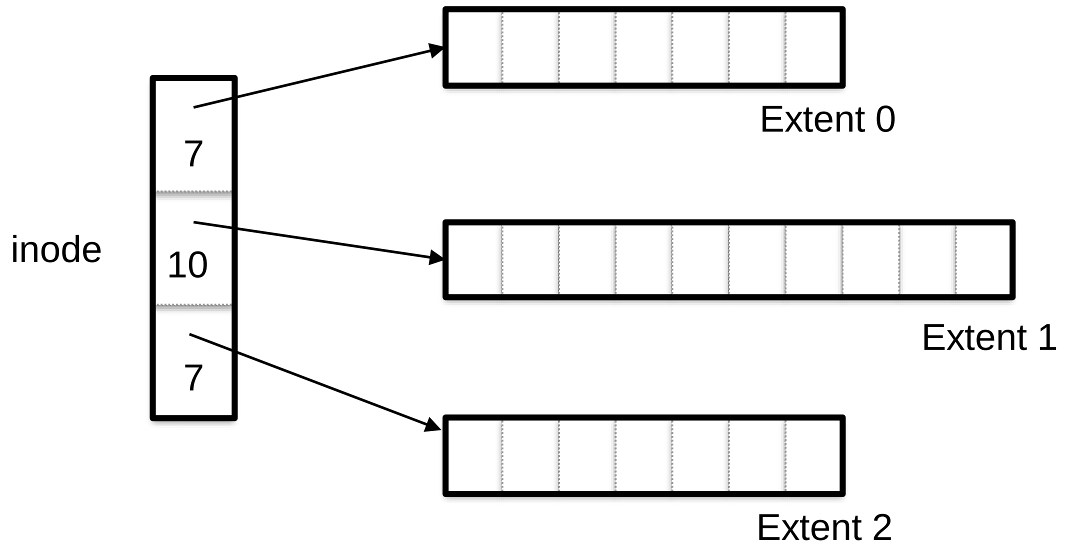
- Operations
  - Allocate a file an extent when it is created
  - Allocate a new extent each time the file exhausts the space available in its last extent
  - File metadata is written only when a new extent is allocated
    - Subsequent data writes do not require additional metadata writes until the next extent is allocated

3. Extent-based Allocation (2)

- Pros
  - Optimized disk seek patterns
    - Efficient for sequential accesses
    - Grouping block-writes into extents allows the file system to issue larger physical disk writes
  - The amount of file metadata gets smaller
    - Files with on a few very large extents require only a small amount of metadata
- Cons
  - Disk space fragmentation may make it hard to find consecutive blocks for large extents

3. Extent-based Allocation (3)

- Examples: Linux ext4, VxFS, QFS



Free Block Management

- Need to efficiently keep track of free blocks
  1. Free list
  2. Bitmap (AKA bit vector)

1. Free List (1)

- Key idea
  - Free disk blocks are linked together
    - A free block contains a pointer to the next free block
    - The block number of the very first free disk block is stored at a separate location on disk and is also cached in memory
    - The last free block would contain a null pointer indicating the end of free list
  - Used in old Unix file system in 1974
    - Major reason behind the poor performance of the file system

Source: https://www.geeksforgeeks.org/free-space-management-in-operating-system

1. Free List (2)

- Pros
  - Simple to implement
  - Space-efficiency
    - No need for extra bookkeeping space
- Cons
  - Poor scalability
    - Finding free space becomes slower as the disk fills
  - Fragmentation
    - Due to frequent file creation and deletion
    - Slows down
      - Finding free space and sequential reads due to excessive seeks

Source: https://www.geeksforgeeks.org/free-space-management-in-operating-system

2. Bitmap (1)

- Key idea
  - Just an array of bits, one per block
    - “0” means block allocated, “1” means block free
    - For a disk drive with 8 KB block, one-to-64 K compression ratio
    - Usually keeps entire bitmap in memory most of the time
  - Used in enhanced form in most file systems of Linux

Source: https://en.wikipedia.org/wiki/Free_space_bitmap

2. Bitmap (2)

- Pros
  - Simple to implement
    - Most CPUs have fast bitmap processing hardware logic
  - Excellent way to avoid fragmentation since it's easy to find a large chunk of free space and allocate it to a file
    - Fast random access allocation check
      - Simply scanning words in a bitmap for a non-zero word
    - Fast deletion
      - No need for overwriting on deletion: just flip the bits
  - Fixed cost
    - 2 TB drive could be fully represented with only 64 MB bitmap

Source: https://en.wikipedia.org/wiki/Free_space_bitmap

2. Bitmap (3)

- Cons
  - Wasteful on larger disks
  - Poor scalability
    - Finding free space becomes slower as the disk fills
    - Performance drops precipitously on all operations if bitmap is larger than available memory
  - Fragmentation
    - Due to frequent file creation and deletion
    - Slows down file operations
      - Finding free space and sequential reads due to excessive seeks

Source: https://en.wikipedia.org/wiki/Free_space_bitmap

2. Bitmap (4)

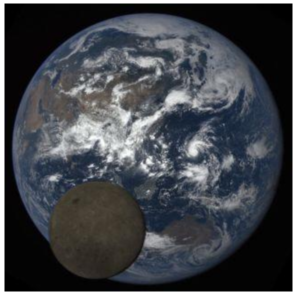
- Enhancements for ever-increasing drive size
  - Split bitmap into chunks
    - Separate array stores the number of free blocks in each chunk
    - Chunks with insufficient space are skipped over
    - Finding free space entails searching the summary array first, then searching the associated bitmap chunk for the exact blocks available

Source: https://en.wikipedia.org/wiki/Free_space_bitmap

IV. Performance

Why File System Performance?

- Huge gap between DRAM and disk performance
  - Representative access time numbers
    - Register: 1 cycle (~1 ns)
    - Cache: 5~15 ns
    - DRAM: 50~150 ns
    - SSD: 20~100 μs
    - Disk: 5~10 ms
  - Performance gap: 1,000~100,000 times
    - Distance between the earth and the moon
      - 363,104 km
    - *Try to save trips to the moon as much as you can!*



Source: The NASA/NOAA DSCOVR spacecraft captured this in July 2016. (Image credit: NASA/NOAA)

How to Improve Performance?

- Clues
  - Lowering the number of real disk accesses
    - “*Buffer cache*”, “*page cache*” for lowering data block access
    - “*Inode cache*” for lowering inode block access
  - Reducing average data access time
    - Seek optimization to reduces the number of seeks

1. Buffer Cache

- Definition
  - A pool of recently-accessed disk blocks kept in main memory
- Key idea
  - If the same blocks are referenced over and over, there's no need ever to read them from disk
  - This solves the problem of slow access to large files
  - *How does a block cache compare with virtual memory?*

2. Page Cache (1)

- Definition
  - A transparent cache for the pages originating from a disk
- Key idea
  - OS keeps it in otherwise unused portions of the main memory
    - Resulting in quicker access to the contents of cached pages
    - Implemented in kernels with the paging memory management
    - Mostly transparent to applications

Source: https://en.wikipedia.org/wiki/Page_cache

2. Page Cache (2)

- Rationale
  - OS uses for the page cache *physical memory not directly allocated to applications*
    - The memory would otherwise be idle and is easily reclaimed whenever applications request it
      - No associated performance penalty
      - OS might even report such memory as "free" or "available"

Source: https://en.wikipedia.org/wiki/Page_cache

3. Inode Cache

- Definition
  - A transparent cache for the inodes originating from a disk
- Key idea
  - OS implements it as a hash table
    - Points to lists of in-memory inodes with the same hash value
    - Calculates hash value from the inumber and the device ID
  - In-memory inode has a reference counter
    - Inode with 0 reference can be a candidate for replacement
    - Important inodes such as "`/`" don't have 0 reference

4. Seek Optimization

- Allocate disk blocks in a way to reduce seeks
  - Try to allocate next block close to previous blocks of a file
    - If disk isn’t full, this will usually work well
    - If disk becomes full, this becomes VERY expensive, and doesn’t get much in the way of adjacency
    - Solution
      - Keep a reserve (e.g., 10% of disk) and don’t even tell users about it
      - Never let the disk get more than 90% full
  - With multiple surfaces on disk, there are multiple optimal next blocks in terms of “*seek time*”
    - With 10% of disk free, can almost always use one of them

V. Reliability

Why File System Reliability? (1)

- Data persistency – unique property of storage device
  - File system data structures must persist despite power loss or system crash
  - “*Crash consistency*”
    - Updating persistent data structures on disk *atomically* despite the presence of a power loss or system crash
    - Otherwise, on-disk data structure inconsistency accrues
  - “*Data integrity*” or “*data protection*”
    - Ensuring the data put into the disk drive remains the same as the data returned from the disk, given the unreliable nature of disk drives

Why File System Reliability? (2)

- Crash inconsistency case 1:
  - Suppose you have to update two on-disk data structures, A and B, to complete a particular operation
    - Ex: Inode and data bitmap
  - Either one of these requests will reach the disk first
    - Disk only services a single request at a time
  - If the system crashes or loses power after one write completes but before the other does, the on-disk data structure will be left in an inconsistent state

Why File System Reliability? (3)

- Crash inconsistency case 2:
  - Metadata updates must be performed *atomically*
    - Deleting a file in a directory
      - Remove the directory entry of the file (data block of the directory)
      - Free the inode of the file (inode bitmap)
      - Free the disk blocks used by the file (data bitmap)
    - What will happen if the system crashes after freeing the inode and before removing the directory entry?
    - In traditional file systems, such atomicity is achieved through synchronous writes – a serious performance penalty
      - Not a solution for a power loss, though

Why File System Reliability? (4)

- Crash inconsistency case 3:
  - Metadata inconsistency
    - Need to read in inode from disk to access data blocks
    - Often maintains in-memory cache of inodes to avoid performance degradation
    - Data inconsistency may occur between in-memory and on-disk metadata in case of a system crash
      - Causes on-disk data structure inconsistency
    - Need to scan entire disk blocks to reconstruct on-disk inodes during system boot after a system crash
      - Serious problem for modern disk drives with huge volume
      - Lazy approach to file system inconsistency

Tools for File System Reliability

- Crash consistency
    1. File system checker (`fsck`) of old Unix
    2. Journaling
- Data integrity (data protection)
    - RAID (redundant array of inexpensive disks)
    - Checksumming

1. File System Checker: `fsck` (1)

- Lazy approach to crash recovery
  - Let inconsistencies happen and then fix them later, usually when booting
  - Implemented by a tool called **`fsck`** in Unix
    - Run before the file system is mounted
  - Can’t fix all problems, though

1. File System Checker: `fsck` (2)

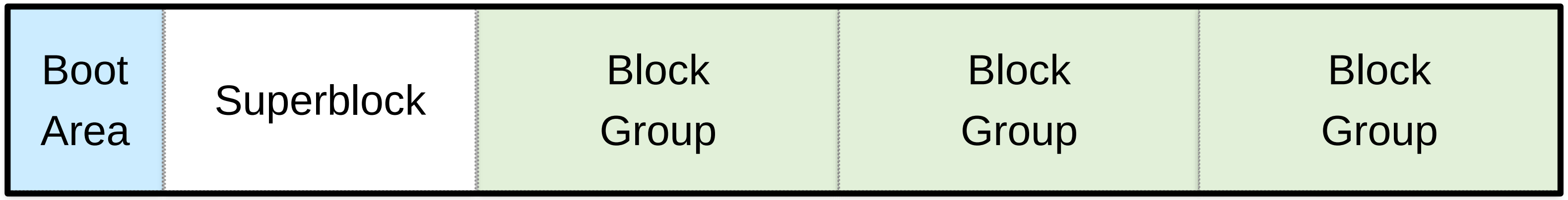
- Operations performed by `fsck`
  1. Reads and checks all inodes and builds a bitmap of used blocks
  2. Records inumbers and block addresses of all directories
  3. Validates the structure of the directory tree, making sure that all links are accounted for
  4. Validates directory contents to account for all the files
  5. If any directories could not be attached to the directory tree in phase 3, puts them into the `lost+found` directory
  6. If any file could not be attached to a directory, puts it into the `lost+found` directory
  7. Checks the bitmaps and summary counts of each cylinder group

2. Journaling (1)

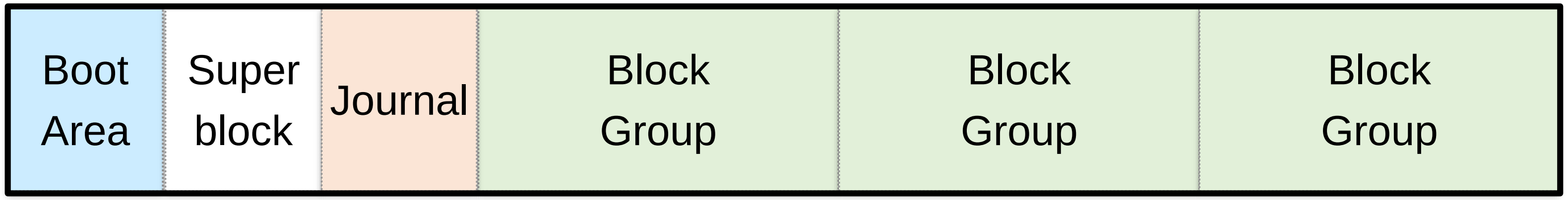
- Is “*write-ahead logging*”
  - An approach taken from the DBMS world
    - When updating on-disk data structures, first writes down a “*log*” somewhere else on the disk, in a well-known location
      - Log is a little note describing what you are about to do
      - Writing this log is the “write-ahead” part
    - Writing the log to disk guarantees the update even in the presence of a crash
      - Can go back and look at the log and know exactly what to fix (and how to fix it) after a crash
    - By design, journaling adds a bit of work during updates
      - No need to scan the entire disk
      - Greatly reduces the amount of work required during recovery

2. Journaling (2)

- Representative file systems utilizing journaling
  - Linux ext3, ext4, ReiserFS, IBM’s JFS, SGI’s XFS, NTFS
    - ext2 lacks journaling



    - ext3 supports journaling



VI. Evolution of Unix File Systems

Unix File Systems

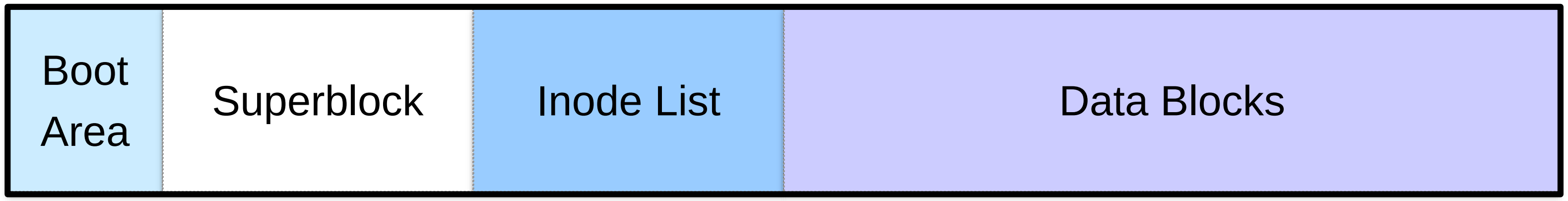
- Unix file systems have evolved
  1. Old Unix file system (FS or s5fs) in 1974
  2. Berkeley fast file system (FFS, often called UFS) in 1984
  3. Log-structured file system (LFS) in 1991
  4. ext2, ext3 file system in 1993~present
  5. Sun’s network file system (NFS)
  6. Sun’s virtual file system (VFS)
  7. And many more

1. S5FS (1)

- Developed by Ken Thomson in 1972
  - When Unix was first introduced
- File system structure
  - File system resides on a single logical disk (partition)
  - Each file system is self-contained
  - A partition is a linear array of disk blocks
  - The size of a block is 512 bytes multiplied by some power of two (512, 1024, or 2048 bytes)
  - The physical block number – an index into this array – uniquely identifies a block on a given disk partition

1. S5FS (2)

- On-disk layout
  - Boot area: may be empty
  - Superblock: holds the metadata of the file system itself
  - Inode list: fixed sized array of inodes
  - Data blocks: holds files, directories, and indirect blocks



1. S5FS (3)

- Superblock
  - Size in blocks of the file system
  - Size in blocks of the inode list
  - Number of free blocks and indoes
  - Free block list
  - Free inode list

1. S5FS (4)

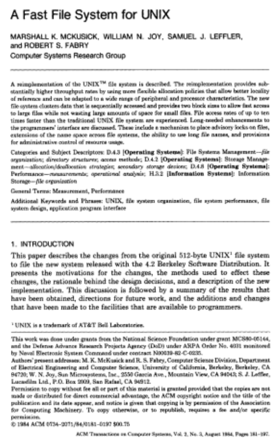
- Drawbacks
  - Reliability concern on superblock
    - Each file system contains a single copy of its superblock
  - Low performance
    - Performance started off bad and got worse over time, to the point where delivering only 2% of overall disk bandwidth
    - Accessing a file requires reading the inode and the file data
    - Inode is allocated far away from its data
    - No attempt to group related inodes (files in the same directory)
      - Example: `ls -l`
  - Suboptimal disk allocation
    - After heavy use, the order of blocks in the free block list is completely random

1. S5FS (5)

- Drawbacks (cont'd)
  - Limitations on functionality
    - Max 14 characters for file names
    - Max 65535 inodes per file system
  - Reason for the poor performance
    - Treated disk like a random-access memory
      - Data was spread all over the place without regard to the fact that the medium holding the data was a disk
      - Thus, had real and expensive positioning costs

2. FFS (1)

- Background
  - Developed by a group at U. C. Berkeley in 1984
    - To overcome the limitations of FS
  - Key idea
    - Disk awareness
      - Design the file system structures and allocation policies considering that the underlying storage device is a disk drive



A Fast File System for UNIX

MARSHALL K. MCKUSICK, WILLIAM N. JOY, SAMUEL J. LEFFLER, and ROBERT S. FABRY
Computer Systems Research Group

A reimplementation of the UNIX™ file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies that allow better locality of reference and can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access to large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long-needed enhancements to the programmers' interface are discussed. These include a mechanism to place advisory locks on files, extensions of the name space across file systems, the ability to use long file names, and provisions for administrative control of resource usage.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*file organization; directory structures; access methods*; D.4.2 [**Operating Systems**]: Storage Management—*allocation/deallocation strategies; secondary storage devices*; D.4.8 [**Operating Systems**]: Performance—*measurements; operational analysis*; H.3.2 [**Information Systems**]: Information Storage—*file organization*

General Terms: Measurement, Performance

Additional Keywords and Phrases: UNIX, file system organization, file system performance, file system design, application program interface

1. INTRODUCTION

This paper describes the changes from the original 512-byte UNIX[1] file system to file the new system released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to effect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the facilities that are available to programmers.

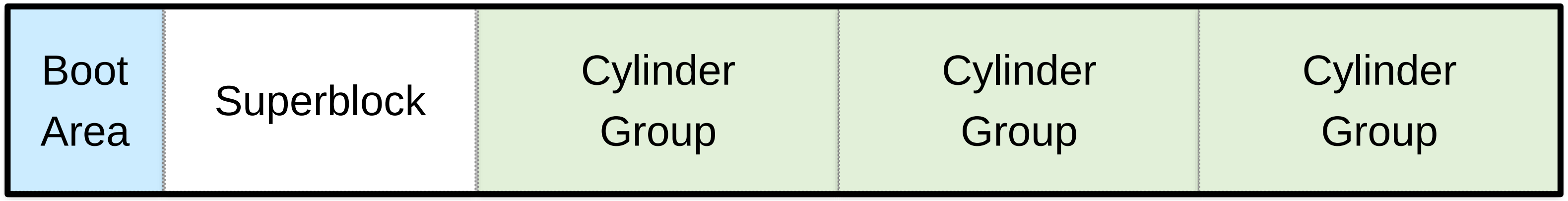
[1] UNIX is a trademark of AT&T Bell Laboratories.

This work was done under grants from the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under ARPA Order No. 4031 monitored by Naval Electronic System Command under contract N00039-82-C-0235.
Authors' present addresses: M. K. McKusick and R. S. Fabry, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA 94720; W. N. Joy, Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043; S. J. Leffler, Lucasfilm Ltd., P.O. Box 2009, San Rafael, CA 94912.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1984 ACM 0734-2071/84/0181-0197 $00.75

ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, Pages 181–197.

2. FFS (2)

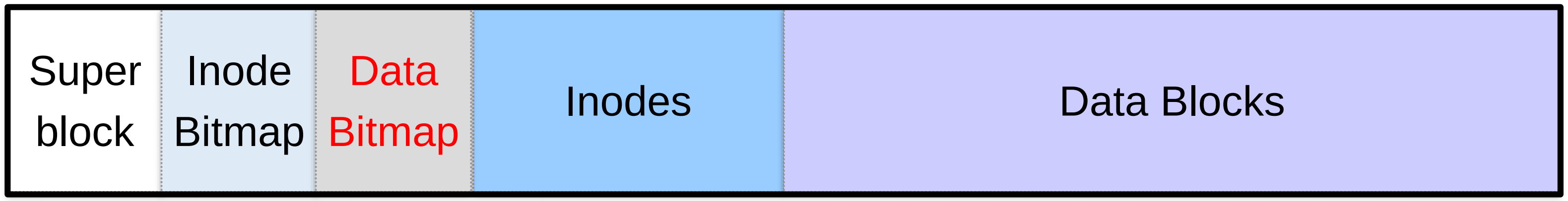
- On-disk organization
  - A formatted partition holds a self-contained file system
  - FFS divides the partition into one or more "*cylinder groups*"



  - Superblock is divided into two structures
    - FFS superblock
      - No., sizes and locations of cylinder groups
      - Block size, total no of blocks and inodes
      - Never change unless the file system is rebuilt
    - Per-cylinder group superblock

2. FFS (3)

- On-disk organization (cont'd)
  - Cylinder group
    - Contains a small set of consecutive cylinders
      - Allows FFS to store related files in the same cylinder group, thus minimizing disk head movements
    - Has the following formatted structure



    - "*Keeps related stuff together!*" – How to find related stuff?
      - Allocates the data blocks and inode of a file in the same group
      - Places all files in a directory along with the directory itself in the cylinder group

2. FFS (4)

- On-disk organization (cont'd)
  - Cylinder group (cont'd)
    - Per-cylinder group superblock
      - Data structure for the particular cylinder group
      - Duplicate copy of the superblock
        - FFS maintains these duplicates at different offsets in each cylinder group in such a way that no single track, cylinder, or platter contains all copies of the superblock

2. FFS (5)

- On-disk organization (cont'd)
  - FFS used 4 KB blocks
    - Many files were small in size, smaller than 2 KB
      - Could incur *internal fragmentation*
  - Blocks were subdivided into sub-blocks called "*fragments*"
  - Fragments
    - Blocks can be broken optionally into 2, 4, 8 fragments
    - Lower bound on the fragment size is the disk sector size
    - Addressable
    - Unit of allocation and allocated in a contiguous manner
    - If a block is fragmented, the fragments of the block can be allocated to different files

2. FFS (6)

- On-disk organization (cont'd)
  - Suppose you create a small file of 1 KB in size
    - The file occupies two fragments of 512 bytes
    - As the file grows, FFS continues allocating fragments to it until it acquires a full 4-KB of data
    - FFS finds a 4-KB block, copies the fragments into it and free the fragments for future use

2. FFS (7)

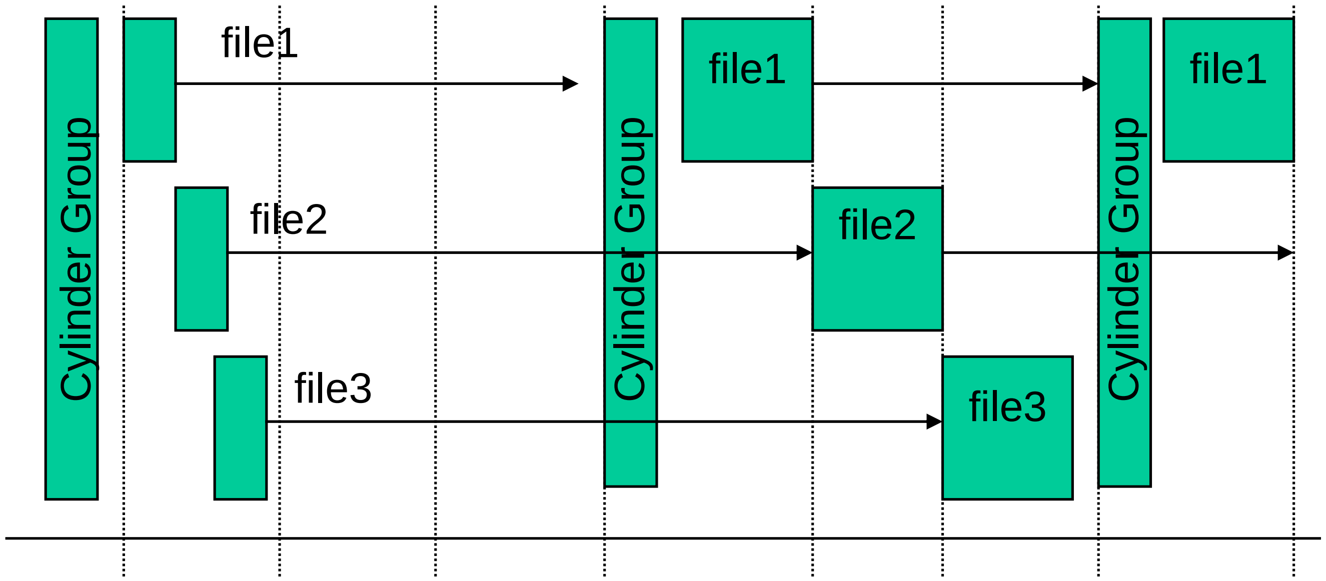
- Heuristics for allocating ordinary files
  - Attempts to place the inodes of all files of a single directory in the same cylinder group
  - Creates each new directory in a different cylinder group from its parent, so as to distribute data uniformly over the disk
  - Tries to place the data blocks of a file in the same cylinder group as the inode

2. FFS (8)

- Heuristics for allocating large files
  - Without an exception to the basic FFS heuristics,
    - A large file would entirely fill the first cylinder group
      - Undesirable, as it prevents subsequent “related” files from being placed within the cylinder group
  - Allocates some number of blocks into the first cylinder group
    - E.g., 12 blocks, or the number of direct pointers within an inode
  - Places the next “large” chunk in another cylinder group
    - E.g., those pointed to by the first indirect block
      - Perhaps chosen for its low utilization
  - Places the next chunk in yet another different cylinder group

2. FFS (9)

- Heuristics for allocating large files (cont'd)
  - Changes the cylinder group when the file size reaches 48 KB and again at every megabyte
    - The 48 KB mark was chosen because the inode's direct block entries describe the first 48 KB (12×4 KB blocks)

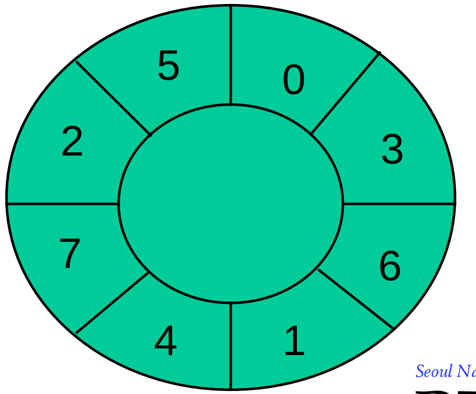


2. FFS (10)

- Allocate sequential blocks of a file at rotationally optimal position
  - The **`rotdelay`** factor or disk's **`interleave`**
    - The time it takes for the kernel to issue the next read and computes the number of sectors the disk head passes over in that time
  - Blocks are interleaved on disk such that consecutive logical blocks are separated by **`rotdelay`** blocks on that track

8 sectors/track

`rotdelay` = 2



2. FFS (11)

- Functionality enhancements
  - Long file names
    - The max size of the filename is 256 characters
  - Symbolic links
  - Disk quota
    - Applies to both inodes and disk blocks
    - Have both a soft limit and a hard limit

2. FFS (12)

- Comparison with S5FS
  - Performance gain
    - Read throughput:
      - 29 KB/s (1 KB blocks in S5FS) vs.
        221 KB/s (4 KB blocks and 1 KB fragments in FFS)
    - Write throughput
      - 48 KB/s vs. 142 KB/s
  - Disk space wastage
    - With free space reserve of 5%, the percentage of waste in S5FS with 1 KB blocks approximately equals that in FFS with 4 KB blocks and 512 B fragments

2. FFS (13)

- Limitations
  - Performance
    - Problem in sequential reads
    - Predominance of writes
  - Slow crash recovery
    - “**`fsck`**” causes unacceptable crash recovery time
  - Limited security policies and mechanisms
  - Size restrictions

3. LFS (1)

- Background
  - Developed by John Ousterhout and Mendel Rosenblum at U. C. Berkeley in 1991
    - To address the new trends in computer systems
      - Memory sizes were growing
      - A large and growing gap between random I/O performance and sequential I/O performance
      - Existing file systems perform poorly on many common workloads
  - Key idea
    - First buffers all updates in an in-memory segment
    - Writes the segment to disk in one, long, sequential transfer to an unused part of the disk when the segment is full
      - Never overwrites existing data, but rather writes segments to free locations

3. LFS (2)

- Basic concepts
  - Record all file system changes in an append-only log file
  - The log is written sequentially, in large chunks at a time
    - Results in efficient disk utilization and high performance
      - Making all writes sequential writes
    - After a crash, only the tail of the log needs to be examined
    - Results in quick recovery and high reliability

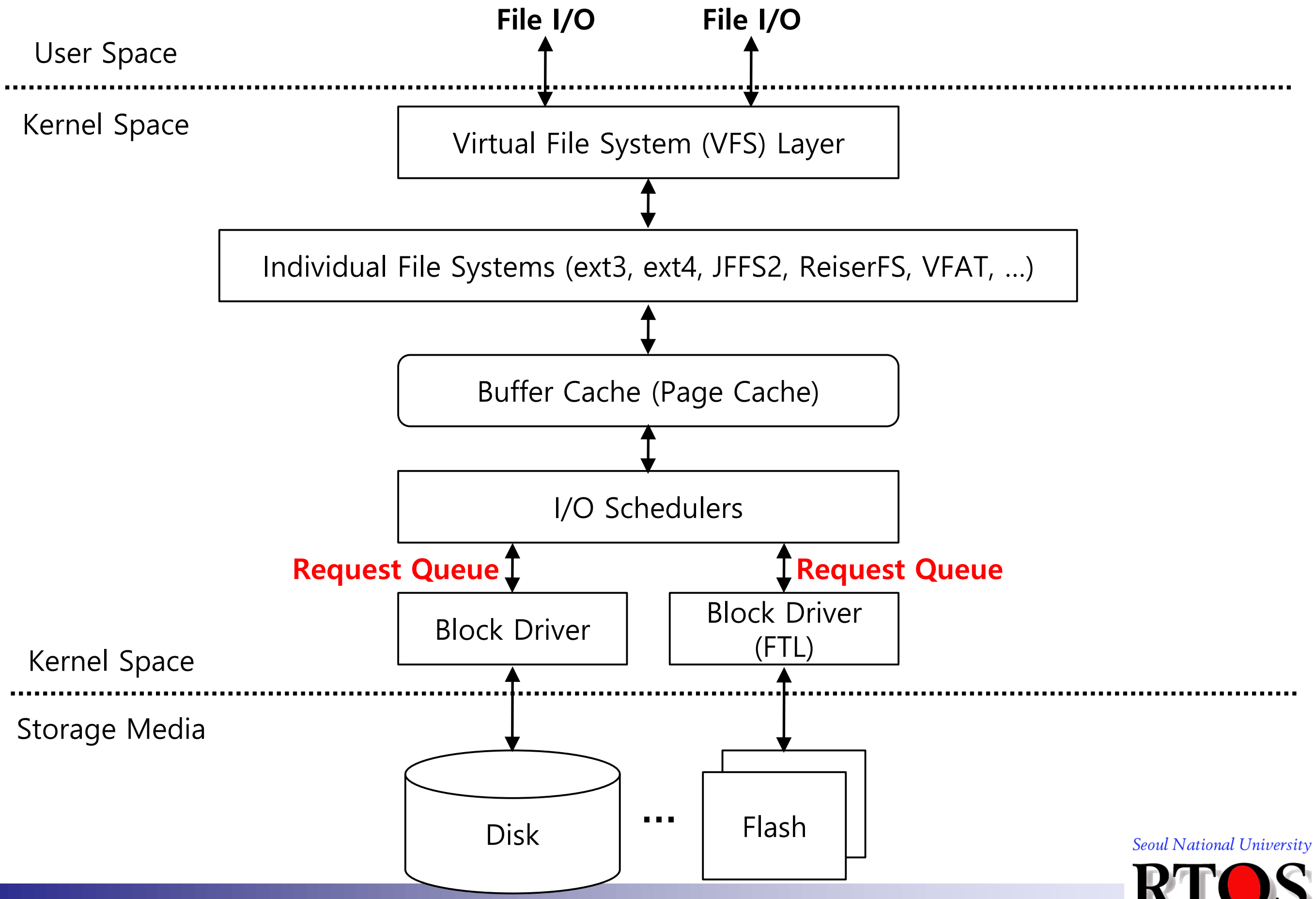
3. LFS (3)

- Design considerations
  - What to log
    - Operations or values
  - Redo and undo logs
  - Garbage collection
  - Group commit
  - Retrieval

VII. Disk Scheduling

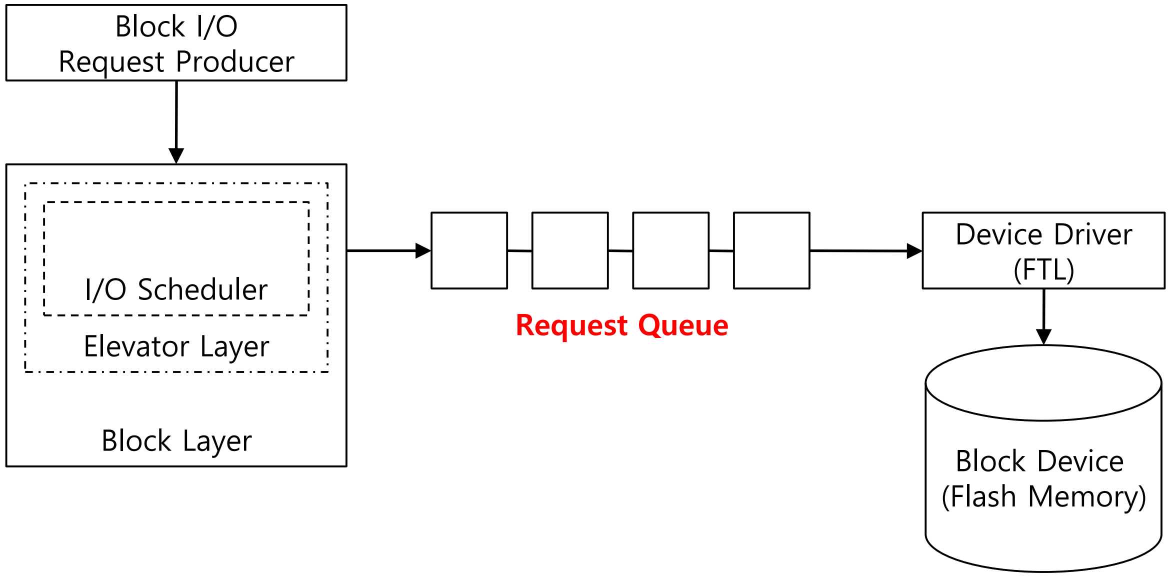
File System inside the Kernel

- Recall the file system structure



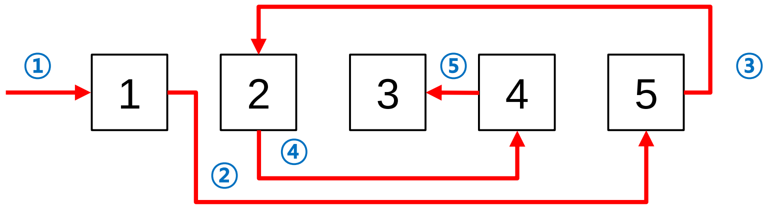
Why Disk Scheduling? (1)

- Block I/O scheduling
  - Several outstanding disk I/O requests frequently exist at the same time in time-sharing OS
  - Gives rise to the "*block I/O*" or "*disk I/O*" scheduler



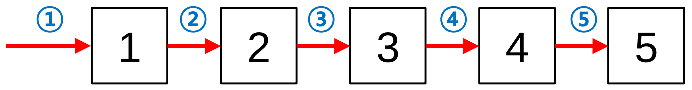
Why Disk Scheduling? (2)

- Ordering of disk I/O requests in the queue affects file system performance
  - When sector read requests arrive at "1, 5, 2, 4, 3" order



Total 10 seeks

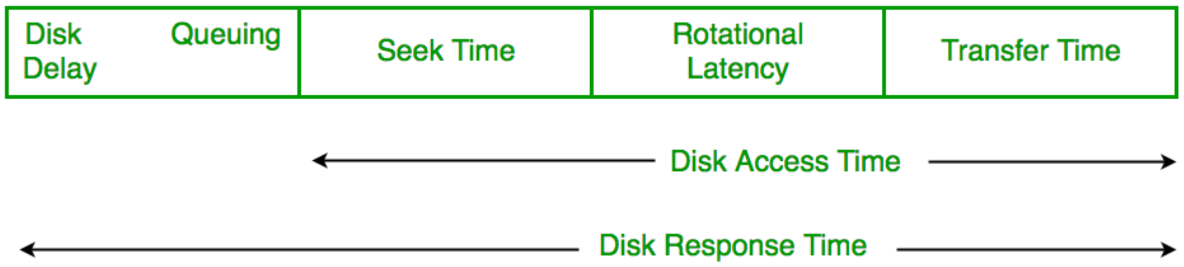
  - When the requests are reordered as "1, 2, 3, 4, 5"



Total 4 seeks

Why Disk Scheduling? (3)

- Disk scheduling algorithms
  - Goals
    - Performance
      - Disk response time must be low
      - Access throughput must be high
    - Fairness
      - Must guarantee free of starvation



Source: https://www.geeksforgeeks.org/disk-scheduling-algorithms

Scheduling Algorithms (1)

1. First come first served (FCFS, FIFO)
   - May result in a lot of unnecessary disk arm motion under heavy loads

2. Shortest seek time first (SSTF)
   - Handle nearest request first
   - Can reduce arm movement and result in greater overall disk efficiency, but some requests may have to wait a long time
     - Severe starvation
   - It is cumbersome to use track number in I/O scheduling; use block number, instead
     - Nearest block first (NBF)

Scheduling Algorithms (2)

3. Scan (SCAN)
   - Moves arm back and forth, handling requests as they are passed (like an elevator)
   - Doesn't get hung up in any place for very long
   - Works well under heavy load, but not as well in the middle (about ½ the time it won't get the shortest seek)

4. Circular scan (C-SCAN)
   - Consider the both ends of the linear sequence of tracks are connected, thus forming a circle
   - Moves arm in one direction in the circle
   - Attempts to achieve fairness for boundary tracks

Scheduling Algorithms (3)

- Example
  - Requested cylinder numbers 0, 53, 14, 27, 2, 31, 85, 30 when head is at 1 initially
  - FCFS: 0, 53, 14, 27, 2, 31, 85, 30
    - Total seek distance: 269
  - SSTF: 0, 2, 14, 27, 30, 31, 53, 85
    - Total seek distance: 86
  - SCAN: 2, 14, 27, 30, 31, 53, 85, 0
    - Total seek distance: 169
  - C-SCAN: 2, 14, 27, 30, 31, 53, 85, 0
    - Total seek distance: 169

Scheduling Algorithms (4)

- Further issues
  - Many more possible scheduling algorithms
    - LOOK algorithm – a variant of C-SCAN
      - Instead of going to the end, goes only to the last request to be serviced in front of head and then reverses its direction from there
  - Newer disks require looking at rotational latencies as well
    - Shortest positioning time first (SPTF)
      AKA shortest access time first (SATF)
  - Most of the time there are not very many disk requests in the queue, so this is not a terribly important decision
  - New storage devices have emerged
    - Flash drives need new scheduling strategies very different from those of disk drives