File-System Implementation

These slides were compiled from the OSC textbook slides (Silberschatz, Galvin, and Gagne) and instructors' class materials.

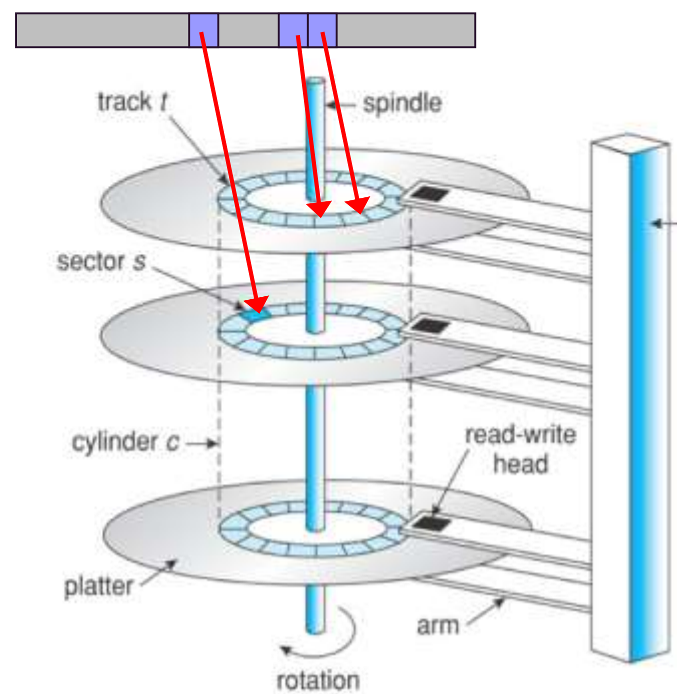
File System Design Challenges

① How should the file system look to the user? (logical view)

② How should algorithms and data structures map the **logical** file system onto the **physical** secondary-storage devices

File System Structure

- File system interface
  - provides applications with various system calls and commands such as open, write, read, seek, etc..
- File system
  - maintains disk space in blocks and allocates available blocks to each stream-oriented file.
- Basic file system
  - maintains data in physical blocks
- Disk driver
  - reads from and writes to disk in a unit of block which consists of one (or more) sector(s).
- Disk
  - maintains data locations with drive#, cylinder#, track# and sector#

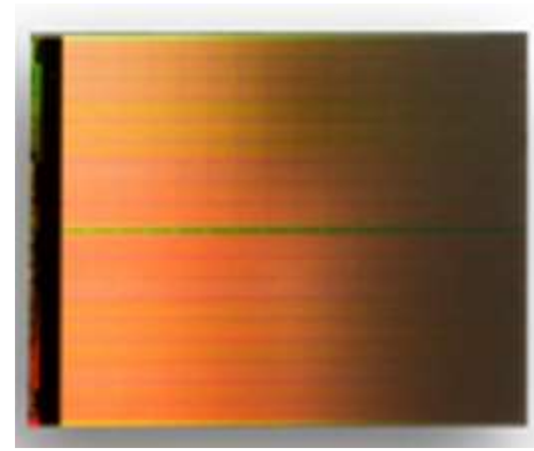


A word on hardware: Magnetic vs. SSD?

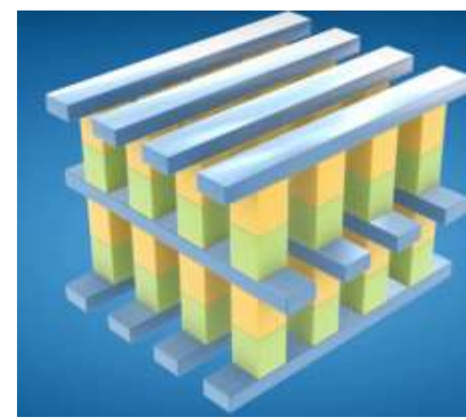
- Structure
  - Don't really have sectors, they have an array of transistors (cells)
  - So how about boot sector 0?
    - Legacy "emulator"
    - New standards use Unified Extensible Firmware Interface (UEFI)

- Physical aspect
  - No moving parts: more energy efficient, more reliable and durable
- Data access
  - Flash memory, not volatile
  - Good for direct access, constant seek time
  - No need to defrag
  - Data is scattered across the device in blocks rather than sectors

Faster than SSD? 3D Xpoint



- "Newer" Technology (2015)
  - New storage standard from Intel and Micron
  - ~1000x faster than flash storage used in SSDs
  - It may also dramatically increase storage capacity
    - 10 times denser than flash memory
- Structure
  - Doesn't use transistors or capacitors
  - Matrix of perpendicular conductors
  - Memory cells can be addressed individually bit-by-bit
  - Reading small data clusters makes Xpoint fast
  - Can current interface technology keep up?



- References:

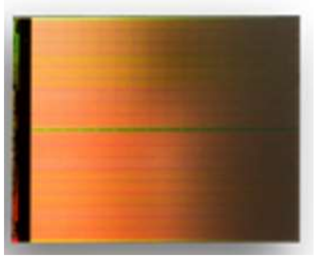
https://www.micron.com/about/emerging-technologies/3d-xpoint-technology

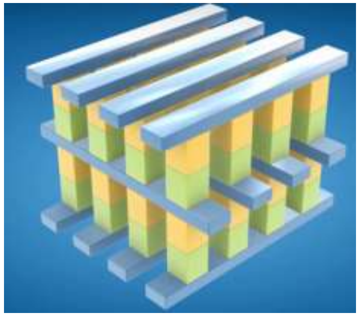
https://youtu.be/Wgk4U4qVpNY

http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-technology-animation.html

The Future?

- "Persist" memory
  - 3D stackable memory can be used for storage





- FPGA (Field-programmable gate array)
  - On-board large memory to buffer data transfers
  - Processing circuitry then can be

File-System (What is kept in Memory?)

- Mount table
  - contains info about each volume
- Directory-structure cache
  - holds directory of recent dir access
- System-wide open-file table
  - contains a copy of the FCB(inode) of each open file and other information
- Per-process open-file table
  - contains a pointer to the FCB(inode) in the system-wide-open-file table and other information
- **Buffers** to hold file-system blocks to/from disk

File-System (What lives on the "disk"?)

- **Boot control block** contains info needed by system to boot OS from that volume
- **Volume control block** contains volume details (superblock in Unix)
- **Directory** structure organizes the files (a file in Unix)
- Per-file **File Control Block** (FCB) contains many details about the file (*inode* in Unix)

FCB = iNode

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

A Typical File Control Block

- NTFS: Master file table entry
- Unix: iNode

Virtual File Systems

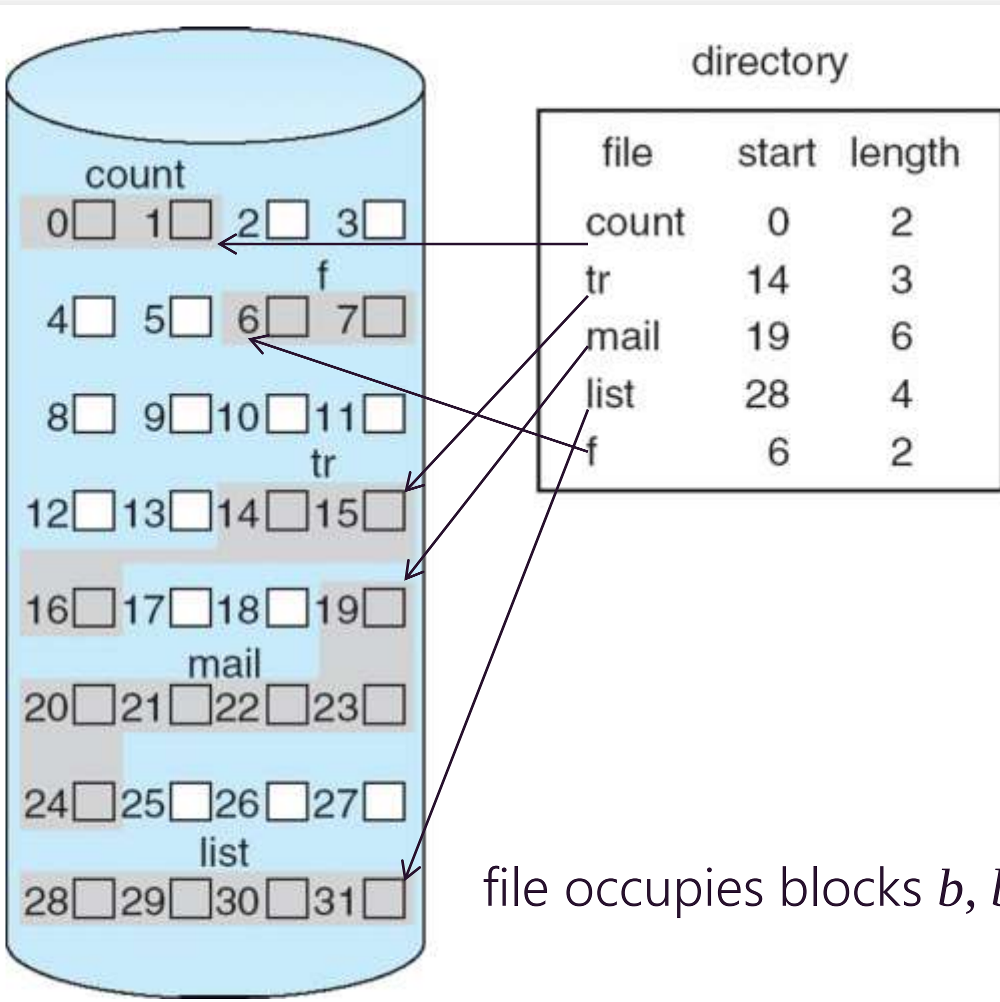
U-read

- Virtual File Systems (VFS) provide an *object-oriented* way of implementing file systems.
  - inode – represents an individual file
  - file object – represents an *open* file
  - superblock – represents enter file system
  - dentry – represents an individual directory entry
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

File Allocation Methods

1. Contiguous
2. Linked
3. Indexed

Contiguous Allocation

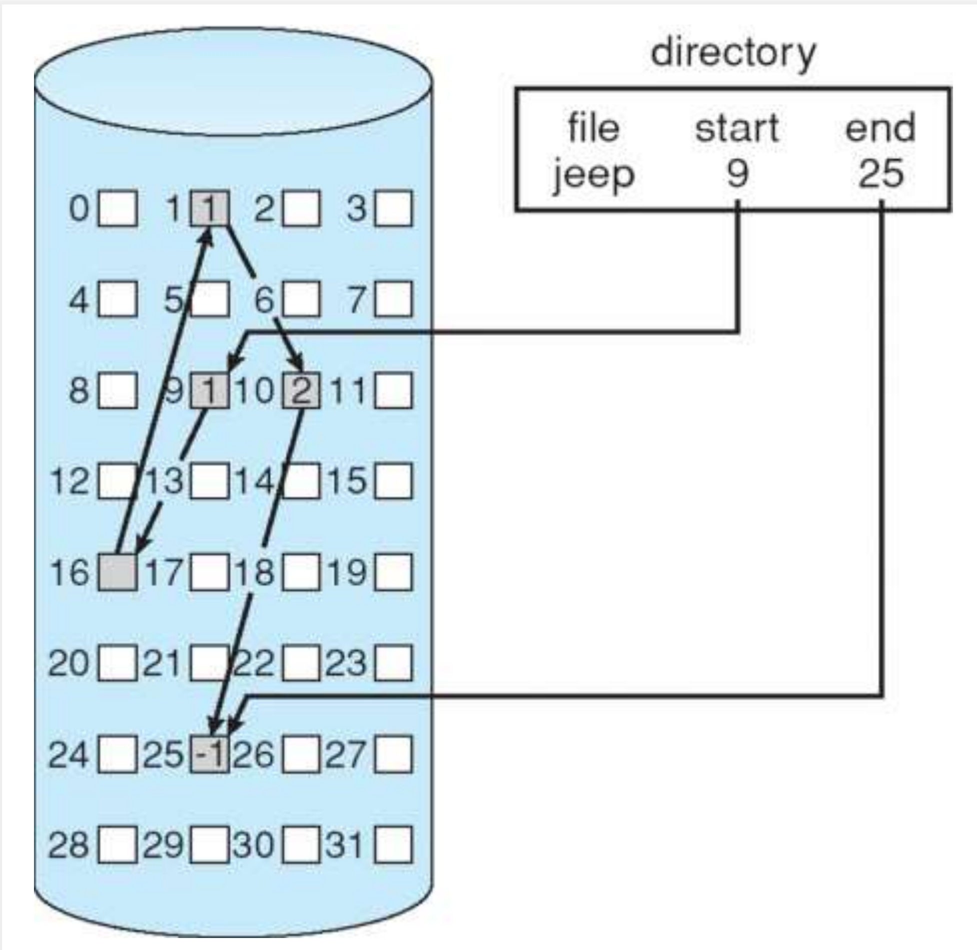


file occupies blocks $b, b + 1, b + 2, \ldots, b + n\text{-}1$

- Pros
  - Good performance (minimal seek time)
  - Example: IBM VM/CMS
- Cons
  - External fragmentation
  - Determining the file space upon its creation

  (Can we predict the size before a file is written?)

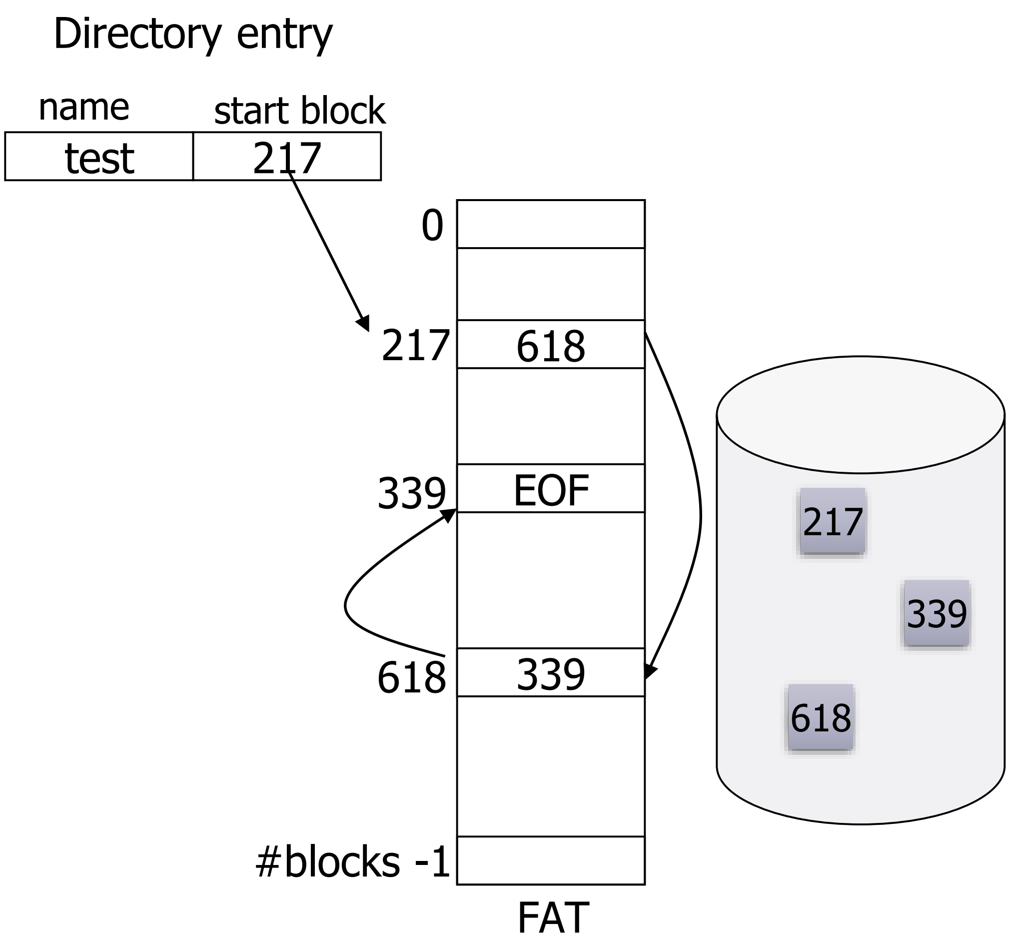
Linked Allocation



- Pros
  - Needs to know only the starting block
  - No external fragmentation
- Cons
  - No sequential access
  - Link information occupying a portion of block
  - File not recovered if its link is broken

- Each disk block contains a pointer to next block
- Blocks must be traversed in order

File Allocation Table (FAT)



- FAT has an entry for each disk block.
- FAT entries rather than blocks themselves are linked.
- Example:
  - MS-DOS and OS/2
- **Pros**
  - Save disk block space
  - Faster random accesses
- **Cons**
  - A significant number of disk head seeks

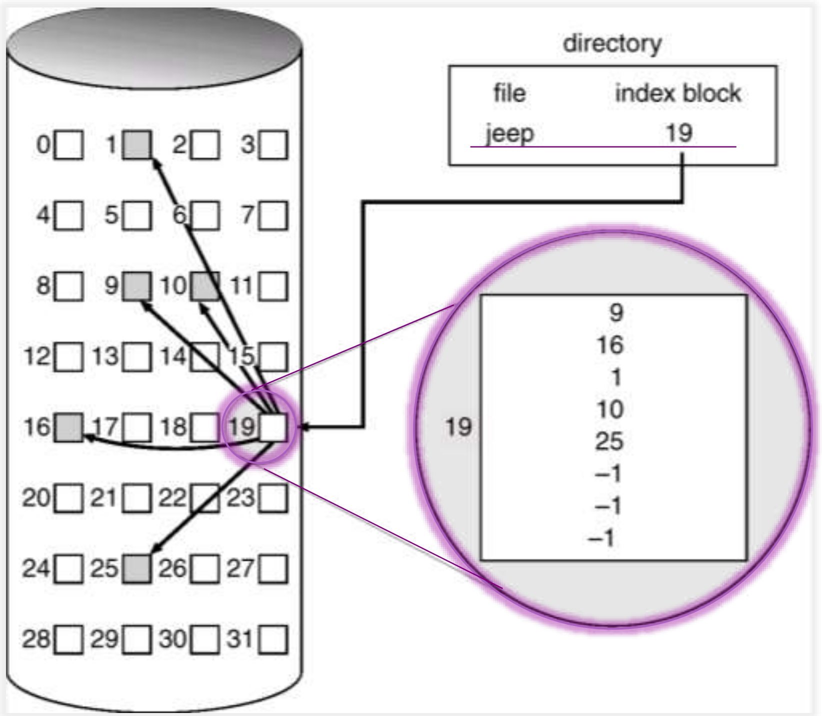
File Allocation Table (FAT)

- FAT12
  - FAT entry size: 12bits
  - #FAT entries: 4K ← (2^12)
  - Disk block (cluster) size: 32KB (depends on each system)
  - **Total disk size: 128MB**

- FAT16
  - FAT entry size: 16bits
  - #FAT entries: 64K ← (2^16)
  - Disk block size: 32KB
  - **Total disk size: 2G**

- FAT32
  - FAT entry size: 32bits, of which 28bits are used to hold blocks
  - #FAT entries: 256M ← (2^28)
  - Disk block size: 32KB
  - **Total disk size: 8T, (but limited to 2T due to the use of sector counts with the 32bit entry)**

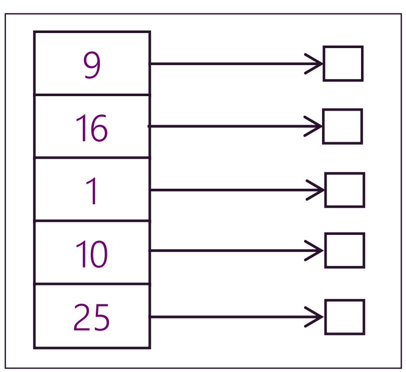
Indexed Allocation



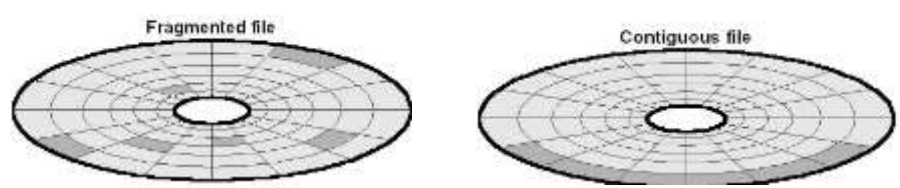
Logical view

Pointer Disk Block

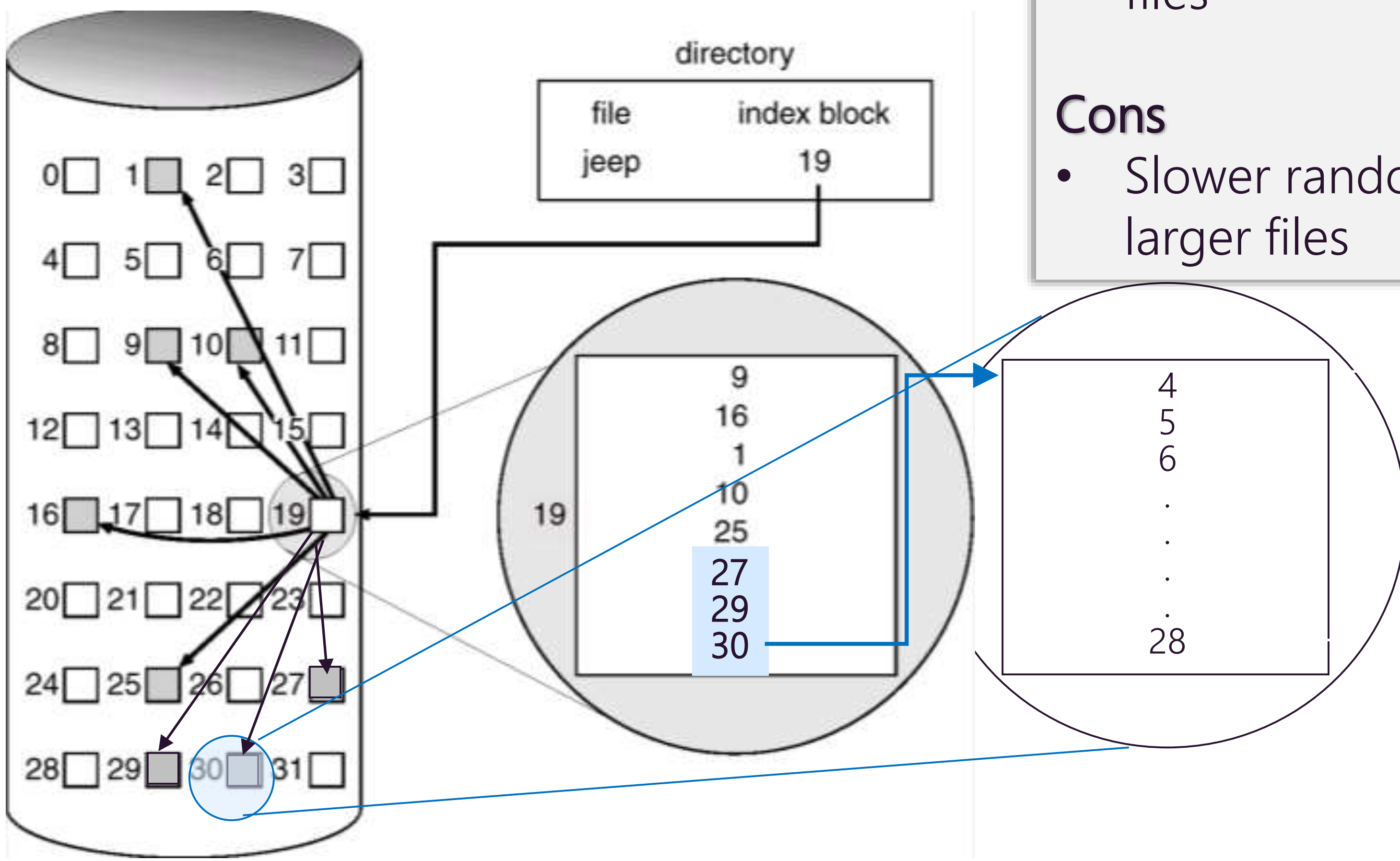
Index Table



- Brings all pointers together into the index block
- Pros
  - Efficient directory access
- Cons
  - Internal fragmentation
  - Uncertainty in the index block size
    - Too small: cannot hold a large file
    - Too large: waste disk space



Linked Scheme in Indexed Allocation



Pros

- Adjustable to any size of files

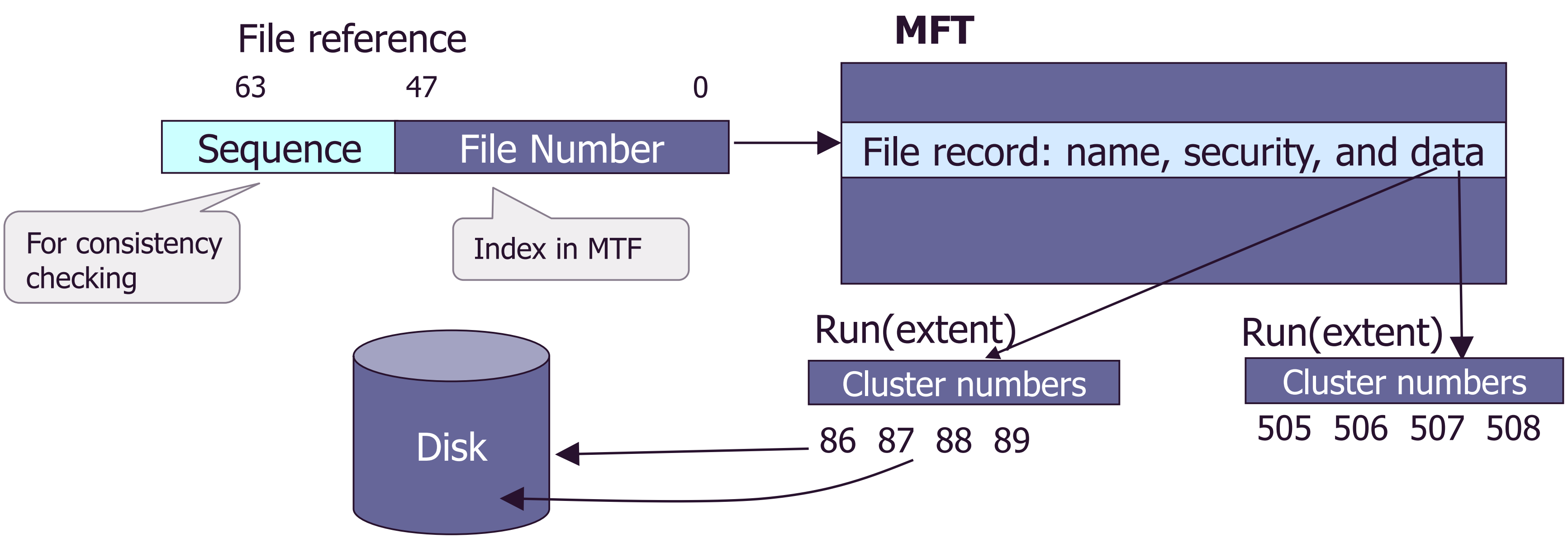
Cons

- Slower random accesses for larger files

NTFS

U-read!
(may be in exam)

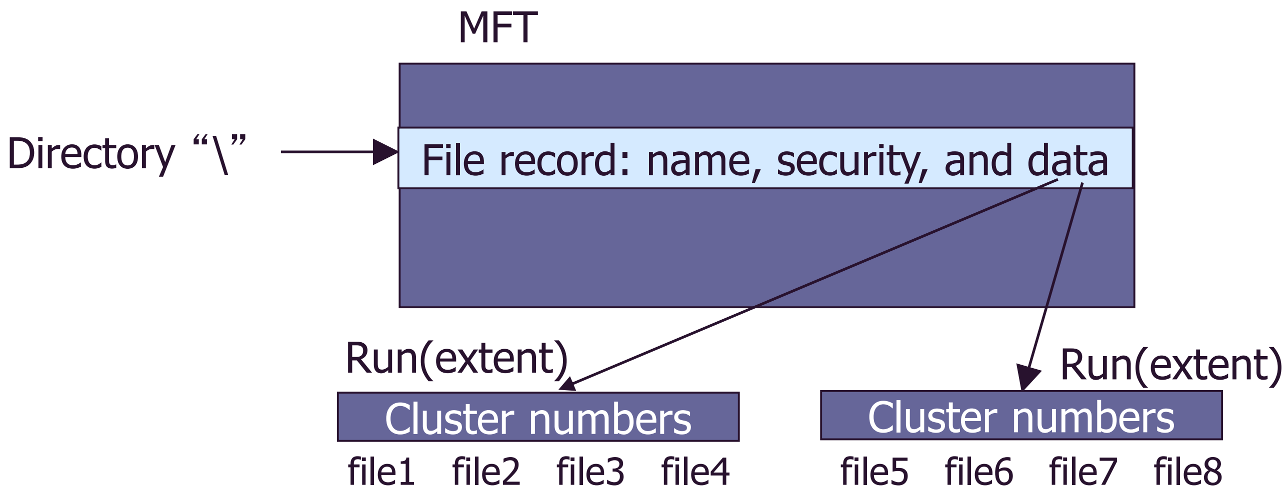
- MFT (Master File Table):
  - An array of records, each holding the attributes for each different file
- Adjustable cluster size (4KB for modern disks)
- Reliable, secure (encryption), recoverable (redundancy)
- User quotas enabled



NTFS Continued

U-read!
(may be in exam)

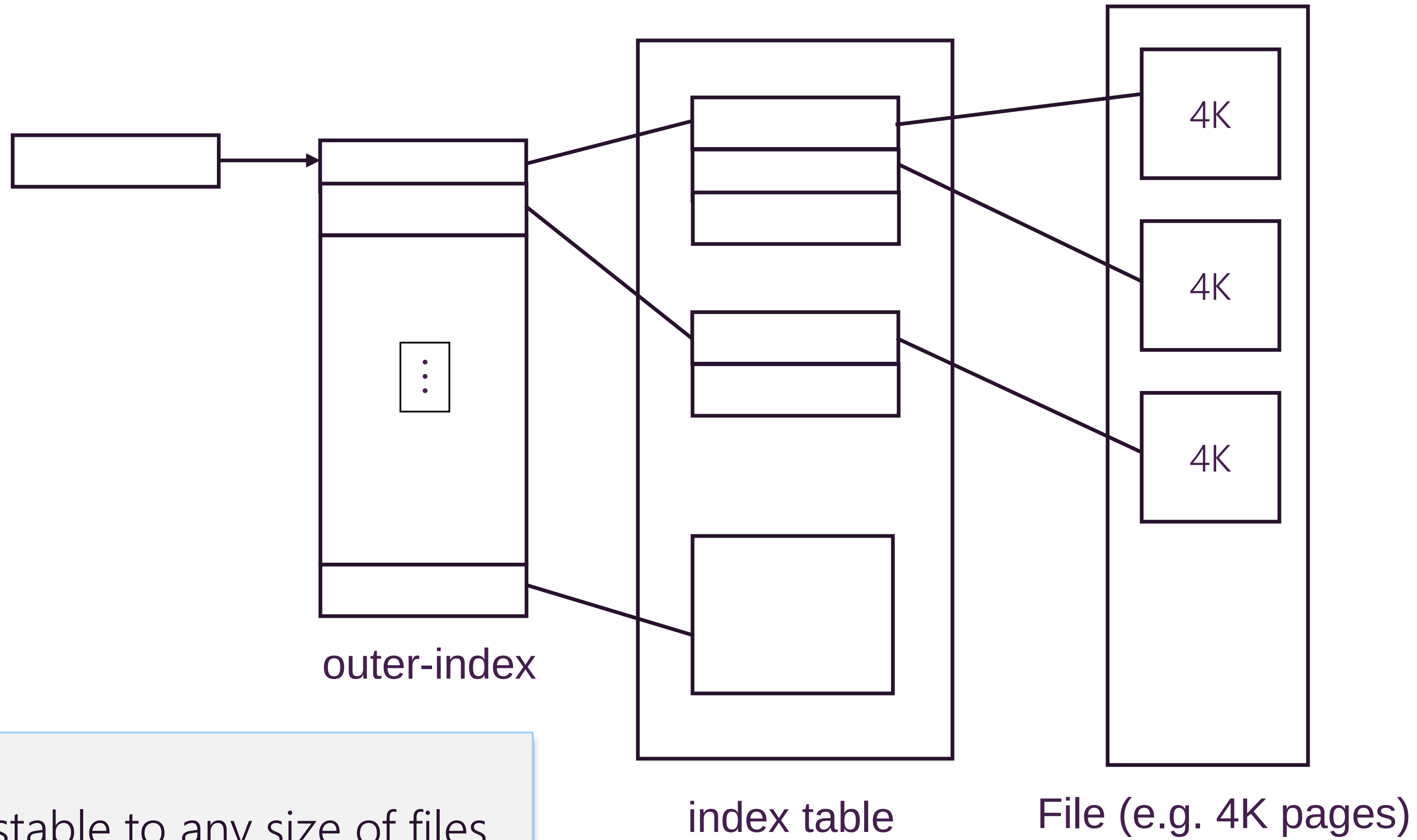
- Directory:
  - Includes all the file references that belong to the directory in its *runs*



Large File Problem?

- **Linked** scheme – linked blocks normally one disk block
- **Multilevel** index – first level index blocks point to other level index blocks, which in turn point to file blocks
- **Combined** scheme (Inode) – both direct index blocks and indirect index blocks

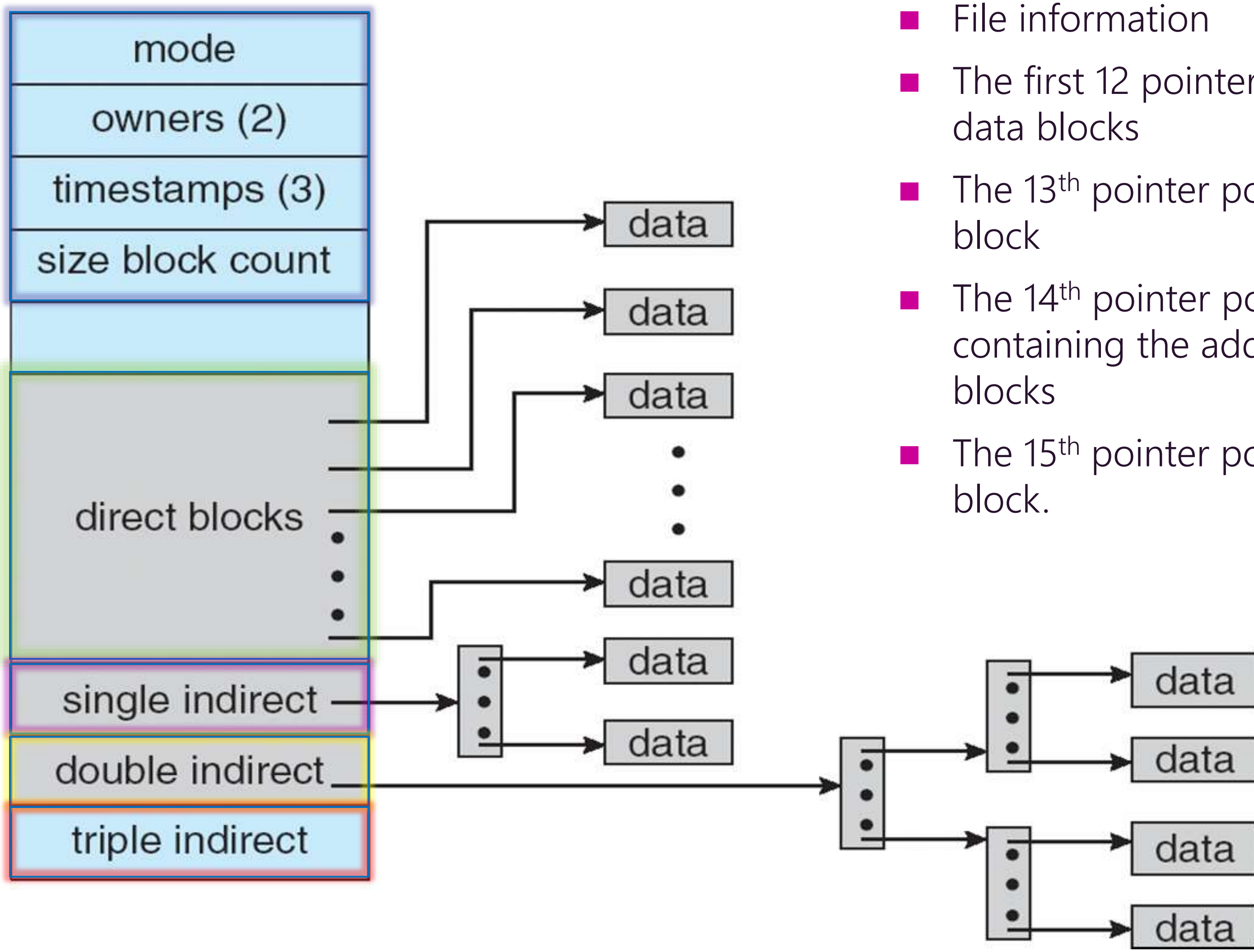
Indexed Allocation - Mapping



Pros
- Adjustable to any size of files

Cons
- Multiple table accesses

Combined Scheme:
Linux – (UFS) 4K bytes per block

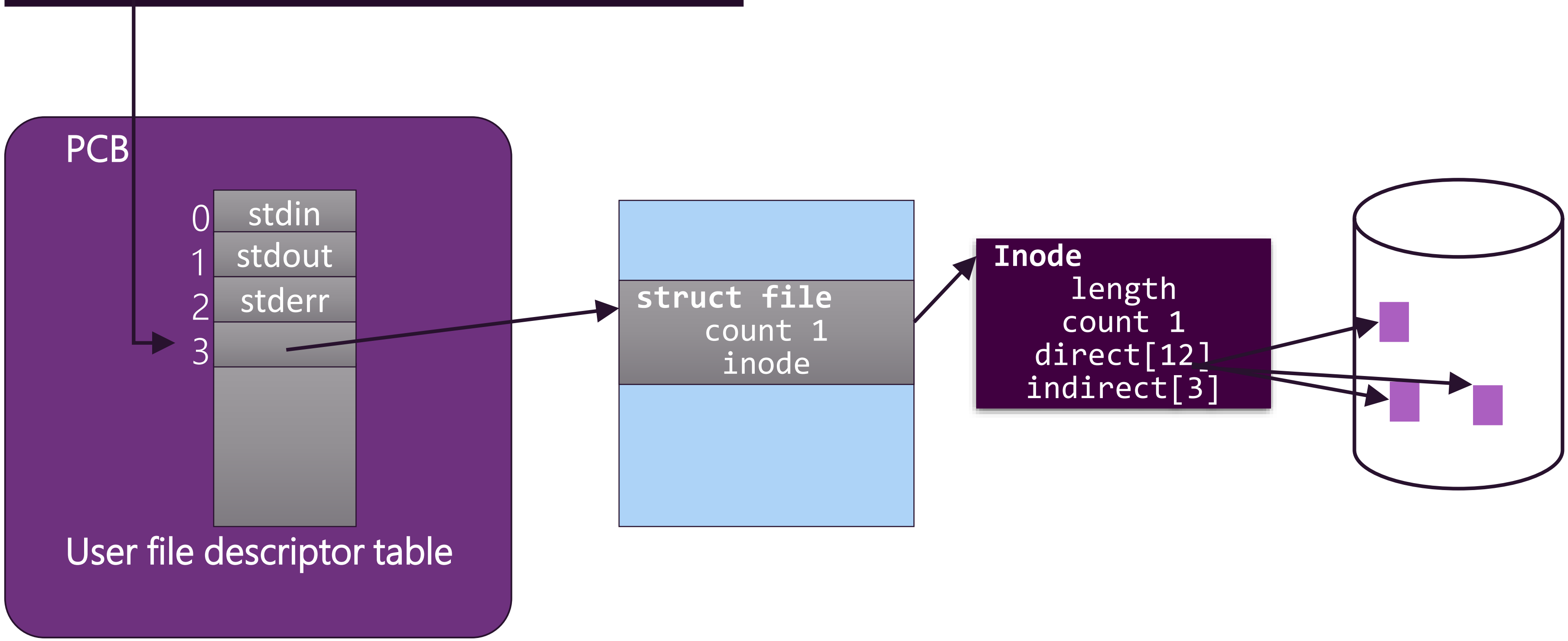


- Inode
  - File information
  - The first 12 pointers point directly to data blocks
  - The 13th pointer points to an index block
  - The 14th pointer points to a block containing the addresses of index blocks
  - The 15th pointer points to a triple index block.

Consider:
How much space?
Big files
Small files

Linux File System Structure

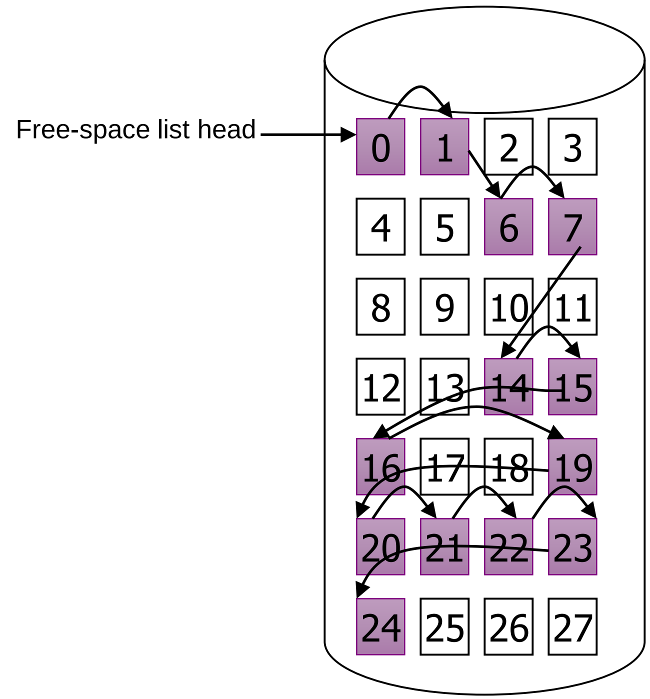
```
int fd = open("fileA", flags);
read(fd, …);
```



Free-Space Management

Free-Space Management

- Free-space list records all free disk blocks—those not allocated to some file or directory.
  1) Bit vector (*n* blocks)
  2) Linked free space list



$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

Pros

- **Bit Vector**: Easy to find contiguous space
- **Linked List**: No space waste in disk

Cons

- **Bit Vector**: Wasted space in memory (in old systems)
- **Linked List**: Difficult to find contiguous space

Free-Space Bit Vector Example

- Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free (0) and the rest of the blocks are allocated (1).

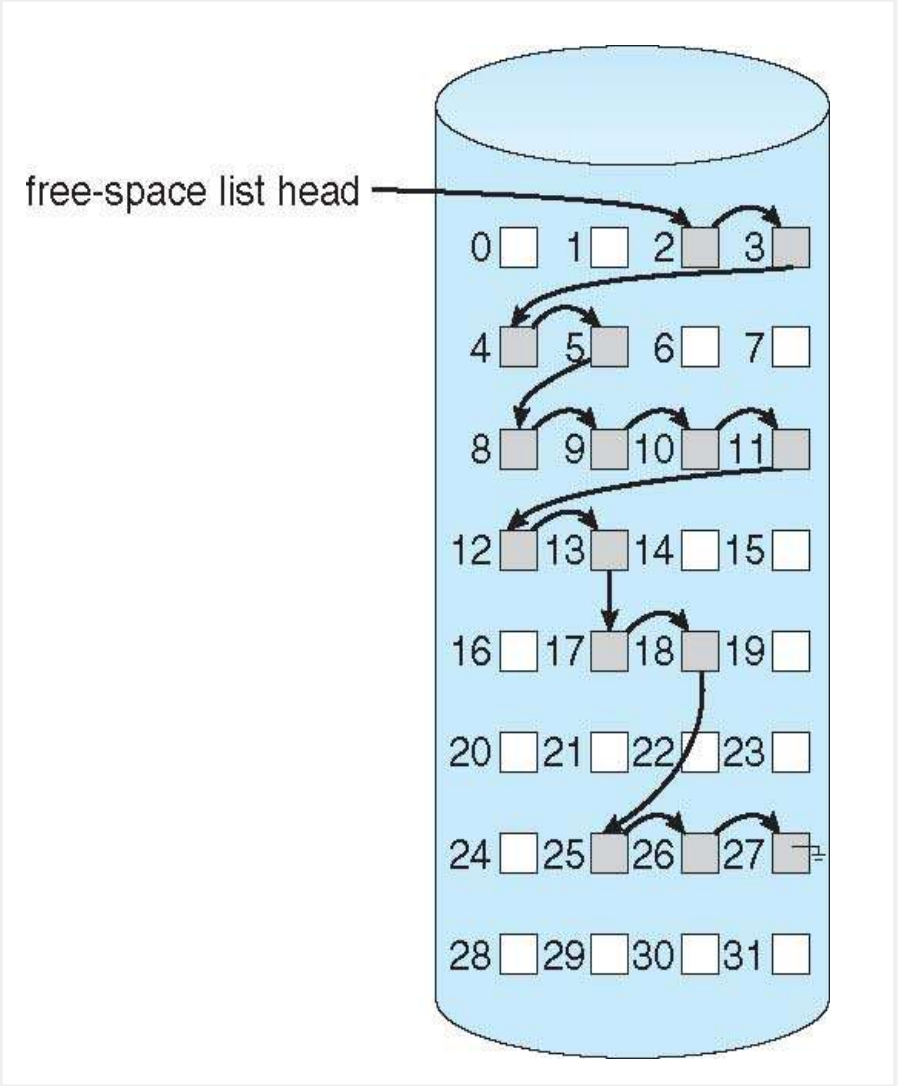
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	1	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	0	0	0	0	1	1
F1								F9								C0								C3							

- Bit vectors are inefficient unless entire vector is kept in main memory
  - Possible for smaller disks
  - Clustering blocks in groups (e.g. 4 blocks/group) can help
  - Not as convenient/feasible for larger disks
    - How much space would a bit-vector use for a 1-TB disk, need given that block size is 4KB? 32MB
    - Divide disk size by block size $2^{40}/2^{12}=2^{28}$

      since each B has 8 bits we need $2^{28}/2^{3} = 2^{25}$ or 32MB to store a bit map

Linked Free Space List on Disk

- Difficult to traverse for large number of free-blocks
- Efficient if the OS just needs a free-block.
- Grouping and/or counting free-blocks can improve efficiency

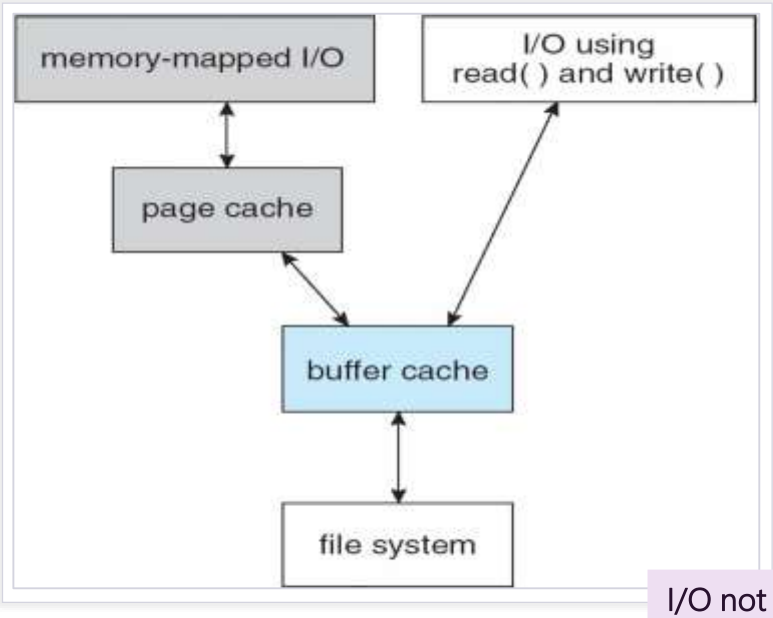


Efficiency and Performance

- Efficiency depends on
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
- Performance improvements
  - **disk cache** – separate section of main memory for frequently used blocks
  - **free-behind and read-ahead** – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as ***virtual disk***, or RAM disk

File System Caches

Page Cache

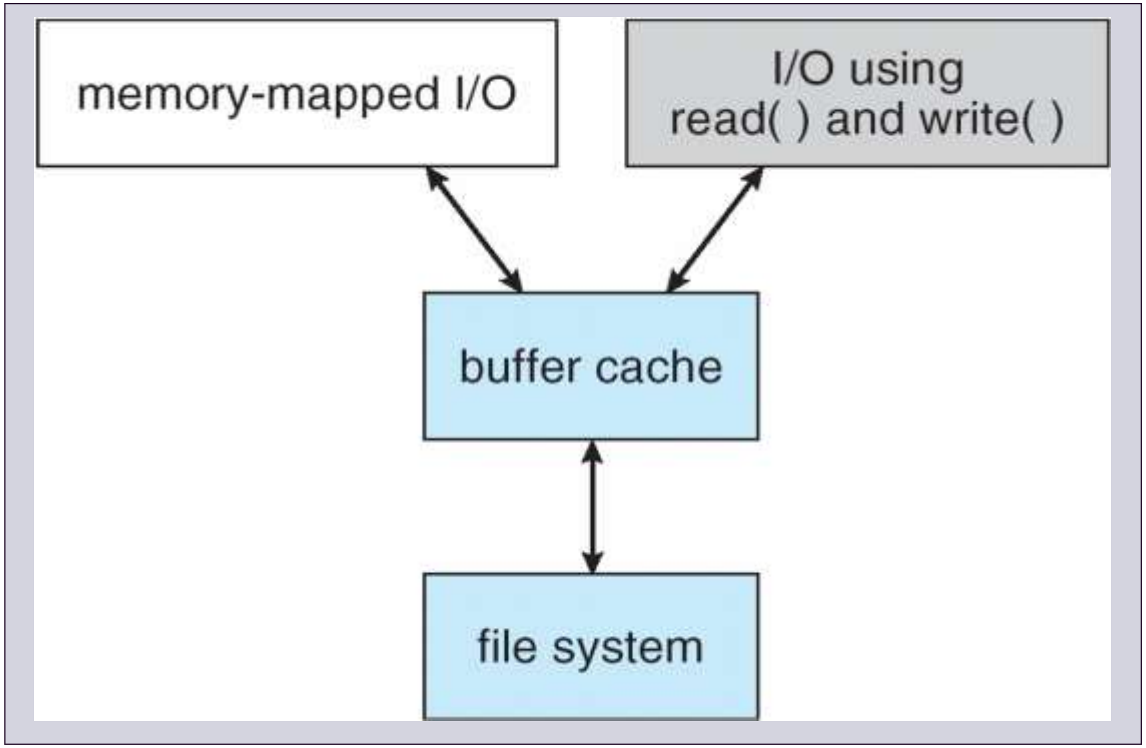


I/O not using unified buffer cache

- **Page cache:** caches *pages* instead of disk blocks
- Used in virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the **buffer (disk) cache**

I/O Using a Unified Buffer Cache

A unified buffer cache (UBC) uses the *same* page cache to cache both memory-mapped pages and ordinary file system I/O



Key benefit is UBC allows the virtual memory system to manage file-system data which removes the double caching performance consistency issues with the former method.

Network File System (NFS)

U-read!

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs).

NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner.
- A remote directory is mounted over a local file system directory.
  - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory.
- Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided.
  - Files in the remote directory can then be accessed in a transparent manner.
- Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory.

NFS (Cont.)

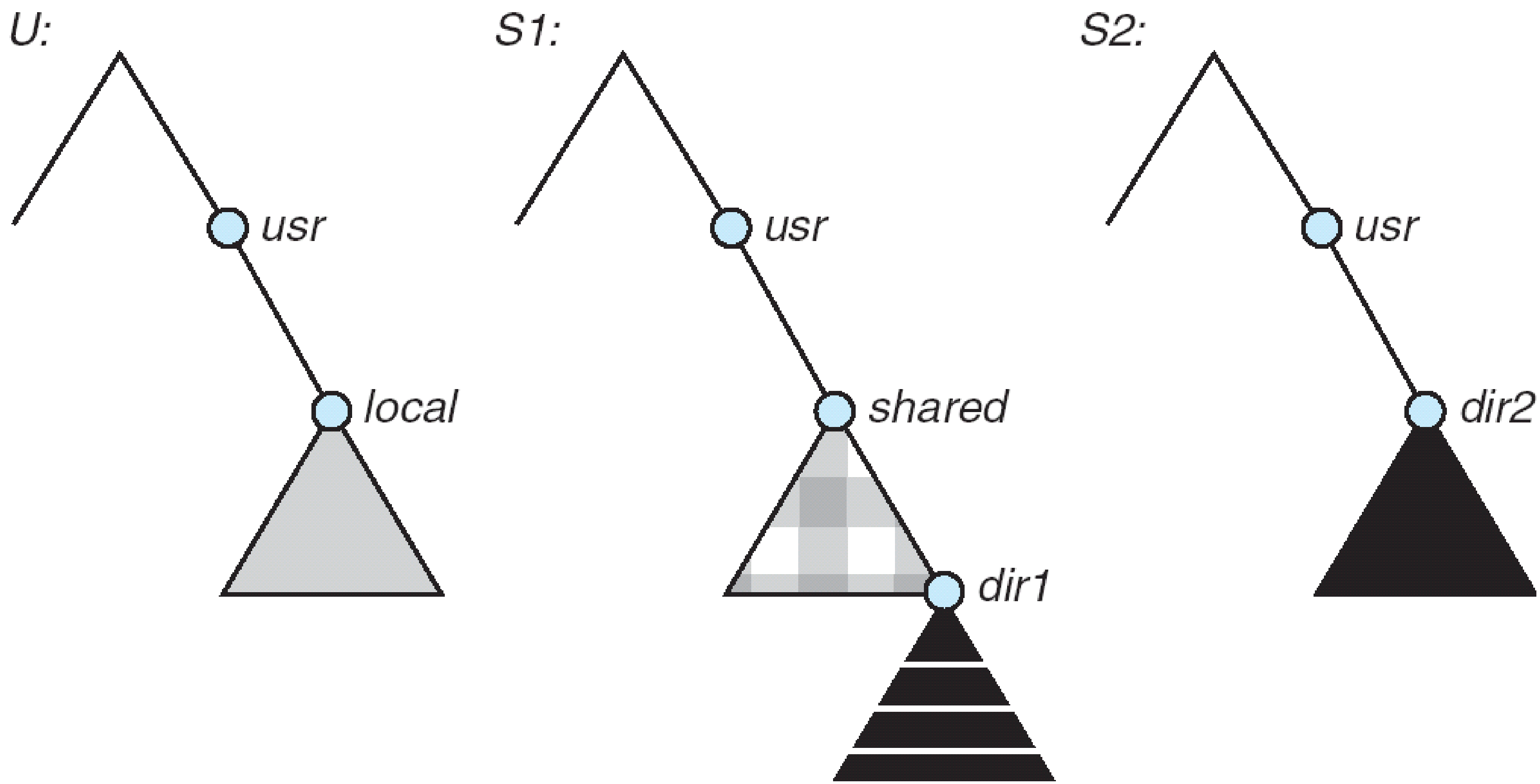
- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures

- NFS specifications are independent of these media.

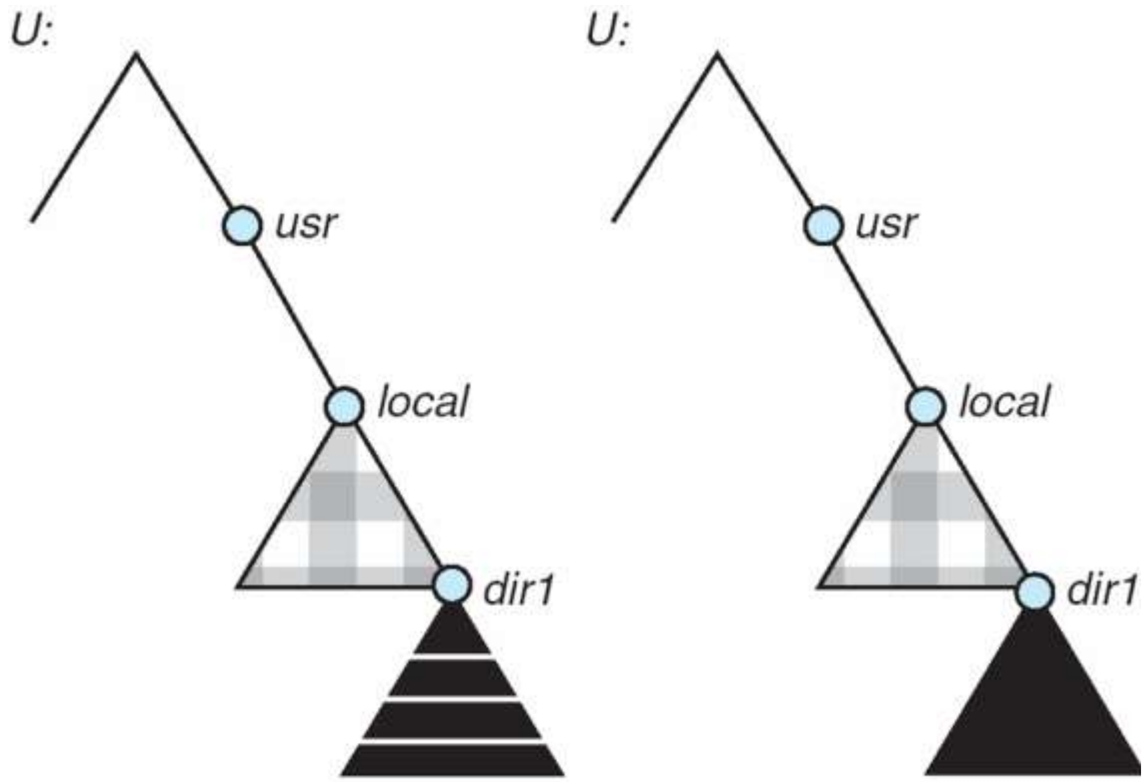
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces.

- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services.

Three Independent File Systems



Mounting in NFS



(a) (b)

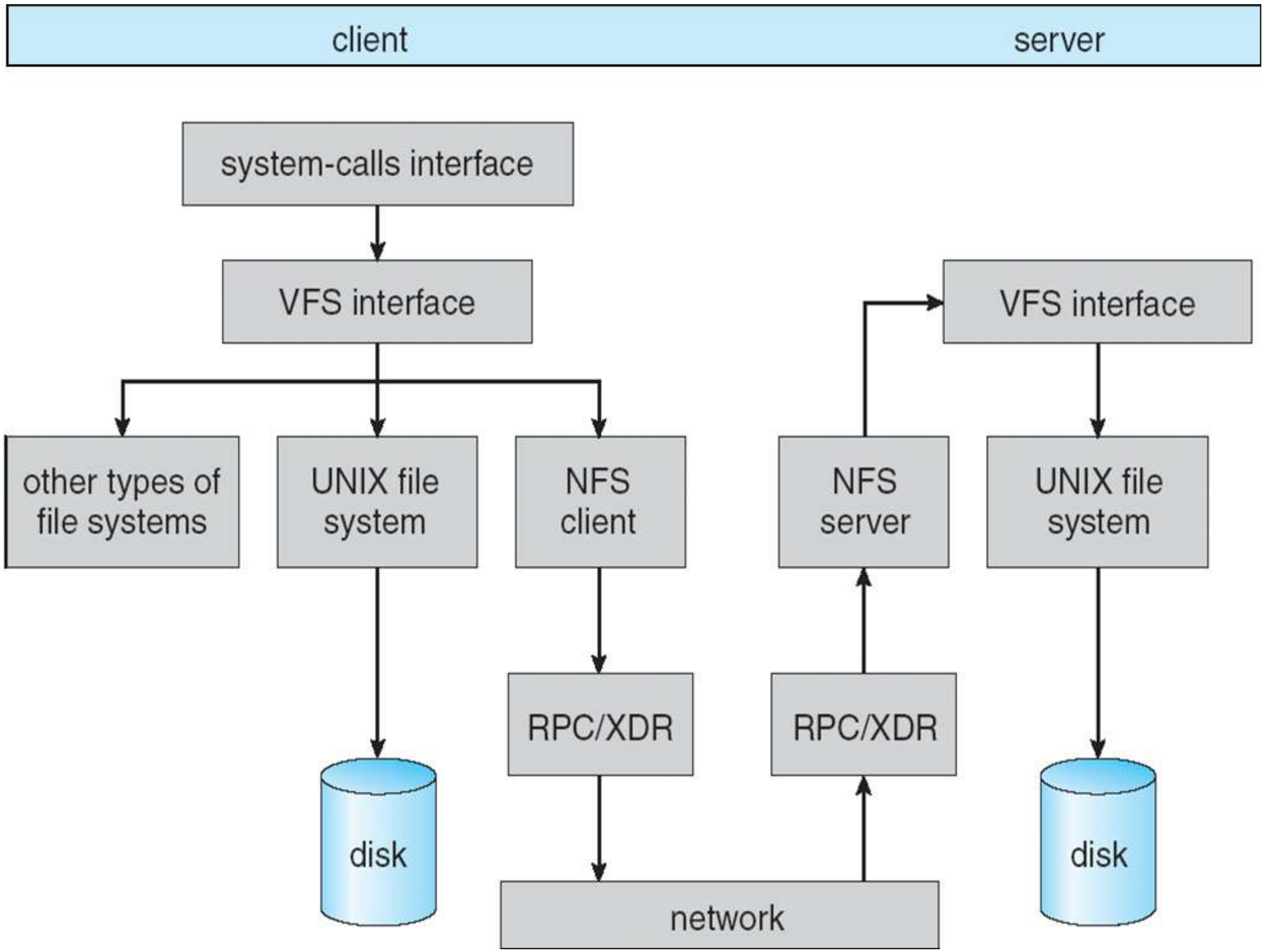
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files

- NFS servers are **stateless**; each request has to provide a full set of arguments
  → (update) NFS V4 was released 2000 and updated in 2010) Comments: influenced by AFS and CIFS, includes performance improvements, mandates strong security, and introduces a stateful protocol

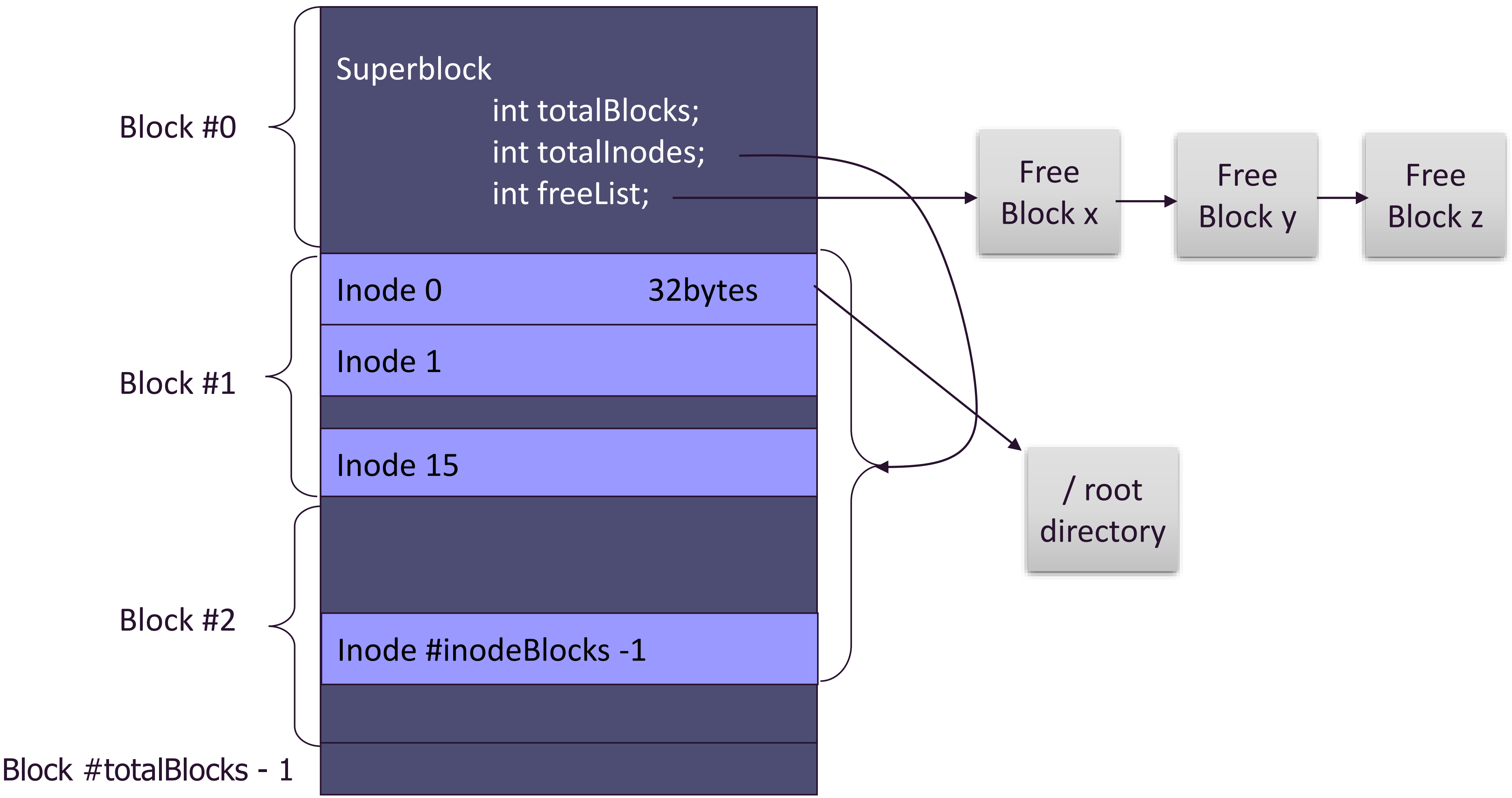
- Other comments :OpenBSD's Theo de Raadt wrote: "NFSv4 is not on our roadmap. It is a ridiculous bloated protocol which they keep adding crap to." -- Caveat Emptor

Schematic View of NFS Architecture



ThreadOS File System Overview

CSS430 ThreadOS File System Superblock and Inodes



CSS430 ThreadOS File System
"/" Root Directory

Entry[] (iNumber)	fname size	fileName
0	1	/
1	4	init
2	4	fsck
3	4	clri
4	4	motd
5	5	mount
6	5	mknod
7	6	passwd
8	6	umount
9	9	checklist
10	6	fsdblb
	6	config
inodeBlock-1	5	getty

- Directory()
  - Initialize "/" directory
- bytes2directory(byte data[])
  - Initialize directory with byte[] which have been retrieved from disk
- directory2bytes()
  - Converts directory information into byte[]
- ialloc(String filename)
  - Allocate an iNumber
- ifree(short iNumber)
  - Deallocate the iNumber
- namei(String filename)
  - Return this file's iNumber

CSS430 ThreadOS File System