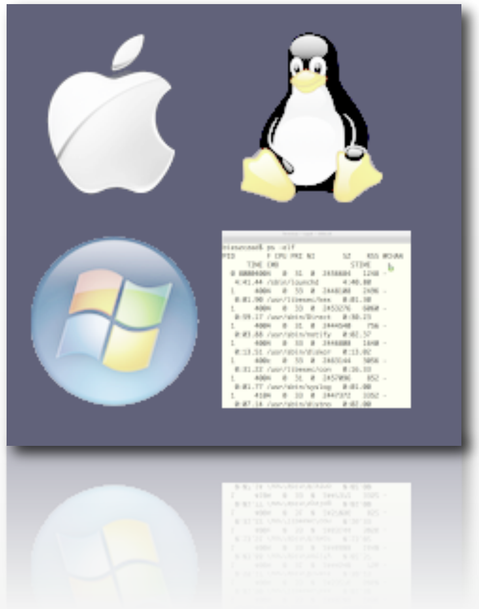


Lecture 12: File System Implementation

COMP362 Operating Systems
Prof. AJ Bieszczad

Outline

- File System Objectives
- File System Structure
- File System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Error Recovery
- Data Consistency
  - Log-Structured FS
  - Journaling FS
  - Copy-On-Write FS

File System Objectives

- File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily

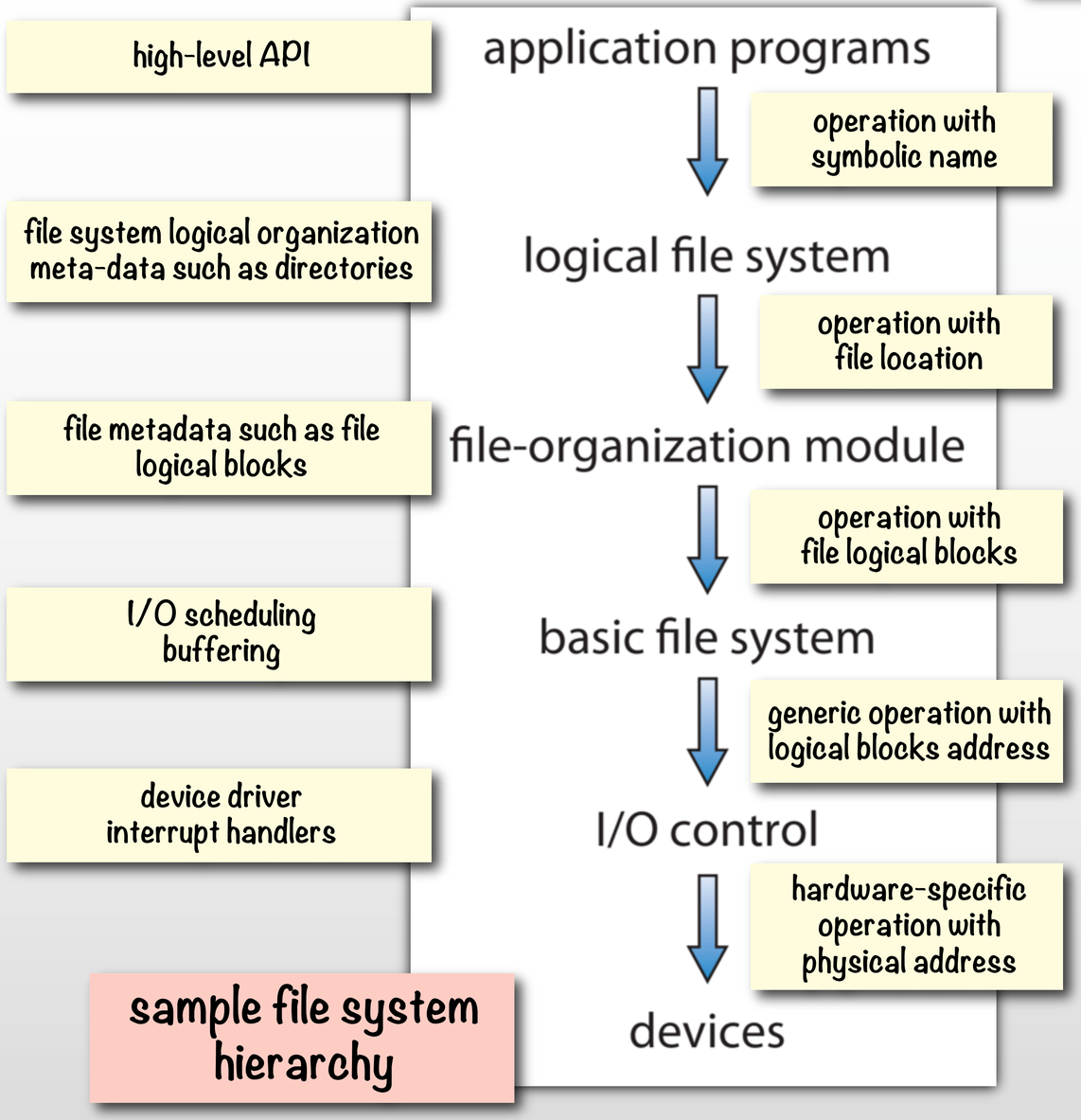
- Two major objectives:

  - Provide user interface to the information stored in secondary storage (such as tapes, disks, SSDs) by defining:
    - a file and its attributes,
    - the operations allowed on a file, and
    - the directory structure for organizing files

  - Create algorithms and data structures to map the logical file system onto the physical secondary-storage devices

File-System Layered Structure

- Secondary storage (tapes, disks, SSD)
- File system organized into layers
  - many details hidden from applications
- Details of device controllers hidden under a uniform interface: device driver
  - drivers are loaded into the OS for specific devices, but the OS-side stays the same



Typical On-Disk File System Structures

- **Boot Control Block**
  - how to boot the OS
    - boot block in Unix file system, partition boot sector in NT File System (NTFS; Windows)
      - for BIOS systems
    - **GUID Partition Table (GPT)**
      - generally requires EFI-based systems (replacement for BIOS) that does not use the boot block, but some OSes can use GPT with BIOS (e.g., Linux) fooling BIOS into believing that the file system is BIOS-compatible

empty for non-bootable volumes

- **Volume Control Block**
  - volume (partition) details: number of blocks, free blocks, block size, file descriptors, etc.
    - content depends on the type of file system as specified in the partition table
  - **superblock** in Unix
  - **Master File Table** (**MFT**) in **NTFS**

- **Directories**
  - collections of file descriptors
    - kept in **inodes** in Unix file system
    - stored in master file table in NTFS

- Per-file **file control blocks** (**FCBs**)
  - details about an individual file or directory
  - **inode** in Unix file system
    - in UNIX, directory is just a special file
  - a row in the master file table in NTFS (like relational database)

File Structure

- File structure
  - logical storage unit
  - collection of related information
- File descriptor
  - File Control Block (a.k.a. FCB; originated in early OSes such as CP/M and DOS)
  - storage structure consisting of information about a file
  - some of the information is copied to computer memory when the file is opened
    - original FCB (CP/M and DOS) kept in user space
    - modern equivalent structures (file handle in Windows or inode in Unix) are kept in kernel space for protection

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

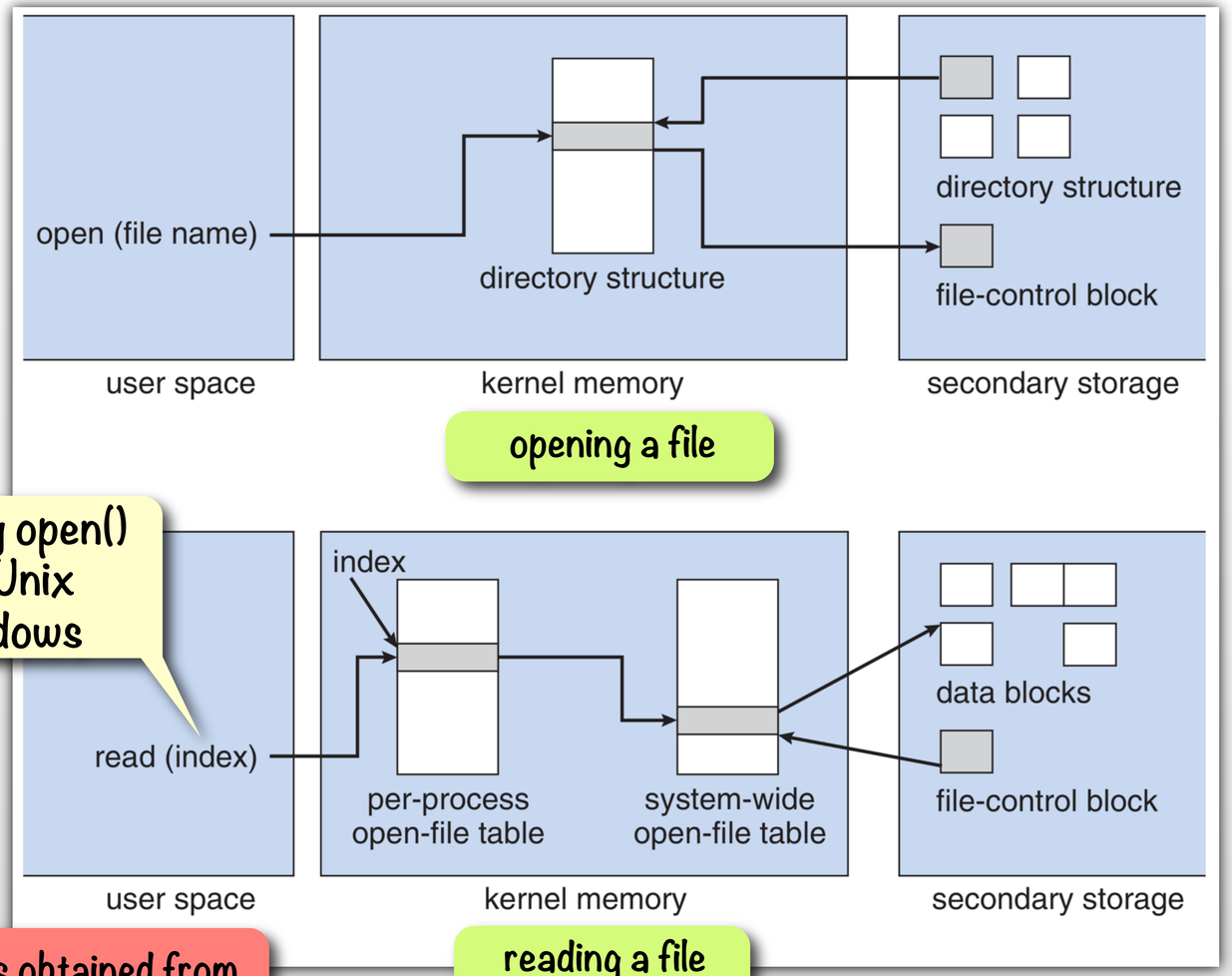
Recall: File Information – POSIX API

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

#define oops(msg, errn) { perror(msg); exit(errn); }

#define fileName "test.txt"

int main(int argc, char *argv[])
{
   struct stat fileInfo;
   if (stat (fileName, &fileInfo) < 0)
      oops("Cannot stat the file.", 1);

   printf("FILE INFO: %s\n\n", fileName);
   printf("%-37s %12s = %10u\n", "Device inode resides on", "st_dev", fileInfo.st_dev);
   printf("%-37s %12s = %10d\n", "Inode's number", "st_ino", fileInfo.st_ino);
   printf("%-37s %12s = %10d\n", "Inode protection mode", "st_mode", fileInfo.st_mode);
   printf("%-37s %12s = %10d\n", "Number or hard links to the file", "st_nlink", fileInfo.st_nlink);
   printf("%-37s %12s = %10d\n", "User-id of owner", "st_uid", fileInfo.st_uid);
   printf("%-37s %12s = %10d\n", "Group-id of owner", "st_gid", fileInfo.st_gid);
   printf("%-37s %12s = %10d\n", "Device type, for special file inode", "st_rdev", fileInfo.st_rdev);
   printf("%-37s %12s = %10d\n", "Time since last access", "st_atimespec", fileInfo.st_atimespec.tv_sec);
   printf("%-37s %12s = %10d\n", "Time since last data modification", "st_mtimespec", fileInfo.st_mtimespec.tv_sec);
   printf("%-37s %12s = %10d\n", "Time since last file status change", "st_ctimespec", fileInfo.st_ctimespec.tv_sec);
   printf("%-37s %12s = %10d\n", "File size, in bytes", "st_size", fileInfo.st_size);
   printf("%-37s %12s = %10d\n", "Blocks allocated for file", "st_blocks", fileInfo.st_blocks);
   printf("%-37s %12s = %10d\n", "Optimal file sys I/O ops blocksize", "st_blksize", fileInfo.st_blksize);
   printf("%-37s %12s = %10o\n", "User defined flags for file", "st_flags", fileInfo.st_flags);
   printf("%-37s %12s = %10d\n", "File generation number", "st_gen", fileInfo.st_gen);
}
```

```
FILE INFO: test.txt

Device inode resides on                    st_dev =  234881032
Inode's number                             st_ino =      31642
Inode protection mode                     st_mode =      16877
Number or hard links to the file         st_nlink =          2
User-id of owner                           st_uid =        503
Group-id of owner                          st_gid =        503
Device type, for special file inode       st_rdev =          0
Time since last access                 st_atimespec = 1239740898
Time since last data modification      st_mtimespec = 1239740907
Time since last file status change     st_ctimespec = 1239740907
File size, in bytes                        st_size =        204
Blocks allocated for file                st_blocks =          0
Optimal file sys I/O ops blocksize       st_blksize =       4096
User defined flags for file               st_flags =          0
File generation number                      st_gen =          0
```

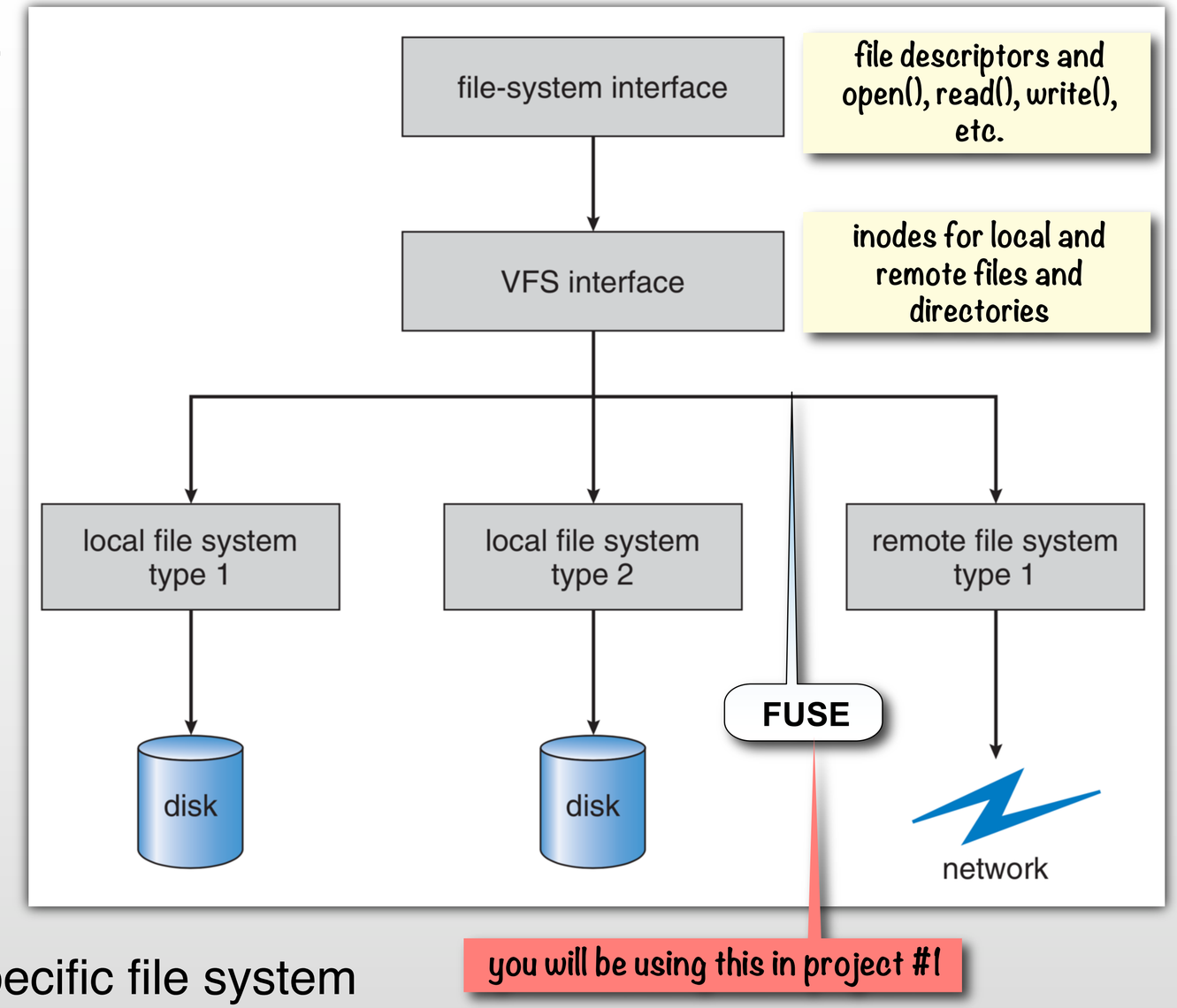
Typical In-Memory File System Structures

- There are a number of supporting file system structures that for the efficiency must be maintained by an operating system after a file system is mounted:
  - directory structure
    - to indicate where file contents are on the disk
  - open-file tables
    - file descriptors of opened files
    - system-wide
    - per process



Virtual File Systems

- **Virtual File System** (**VFS**) provides an optional object-oriented virtualization layer for implementing file systems
  - 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.
  - VFS requests must be interpreted by specific file system modules
- Key virtual objects in VFS:
  - **inode** - object representing an individual file
  - **file** - object representing an open file
  - **superblock** - object representing a filesystem
  - **dentry** - object representing a directory entry
  - Each of the VFS objects has a number of operations defined for it
- The VFS objects are implemented for a specific file system



In-Memory Directory Implementation

- Linear
  - a linear list of file names with pointer to the data blocks
  - sequential access only
  - simple to program
  - time-consuming to search
    - imagine many files
    - could have some higher-level structure like a sorted tree
    - caching in memory alleviates the problem considerably
      - might be too large to keep a complete copy however
  - variable size (adding at the end, removing, etc.)
- Hash Table
  - direct access through a hash function applied to the name
  - decreases directory search time
  - must deal with the collisions
    - efficient solutions do exist though
  - fixed size

Allocation Methods

- An allocation method refers to how disk blocks are allocated for files

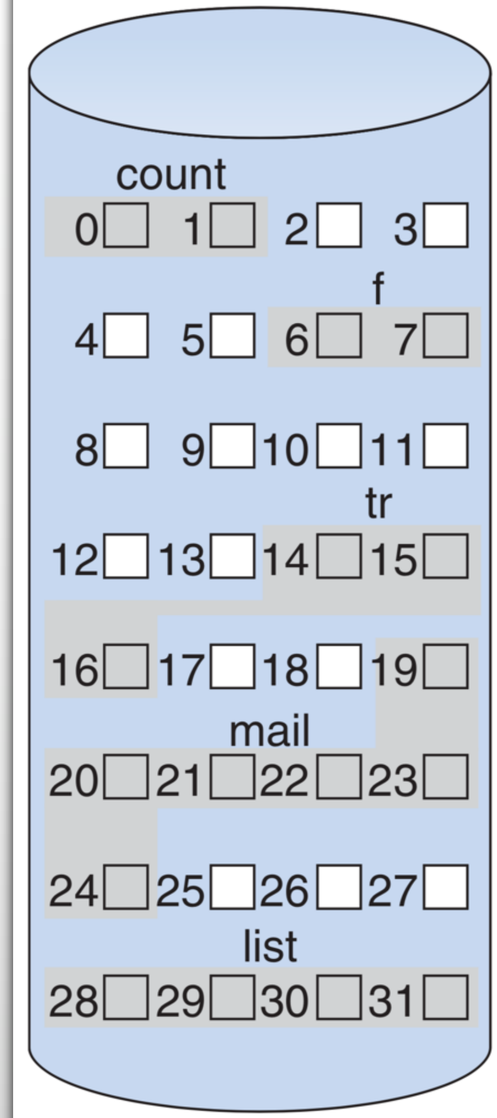
- **Contiguous allocation**
  - Each file occupies a set of contiguous blocks on the disk

- **Linked allocation**
  - Each file is a linked list of disk blocks

- **Indexed**
  - Brings all pointers together into the index block allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple
  - only starting location (block #) and length (number of blocks) are required
- Random access
  - currentBlock + N
- Potential for waste of space
  - dynamic storage-allocation problem
    - external fragmentation!
- Files cannot grow freely
  - they may need to be rewritten in a newly allocated larger space
    - this may be considered a positive in some respect (e.g., preventing data loss)
  - leads to internal fragmentation as the programmers may tend to request much more than what they need

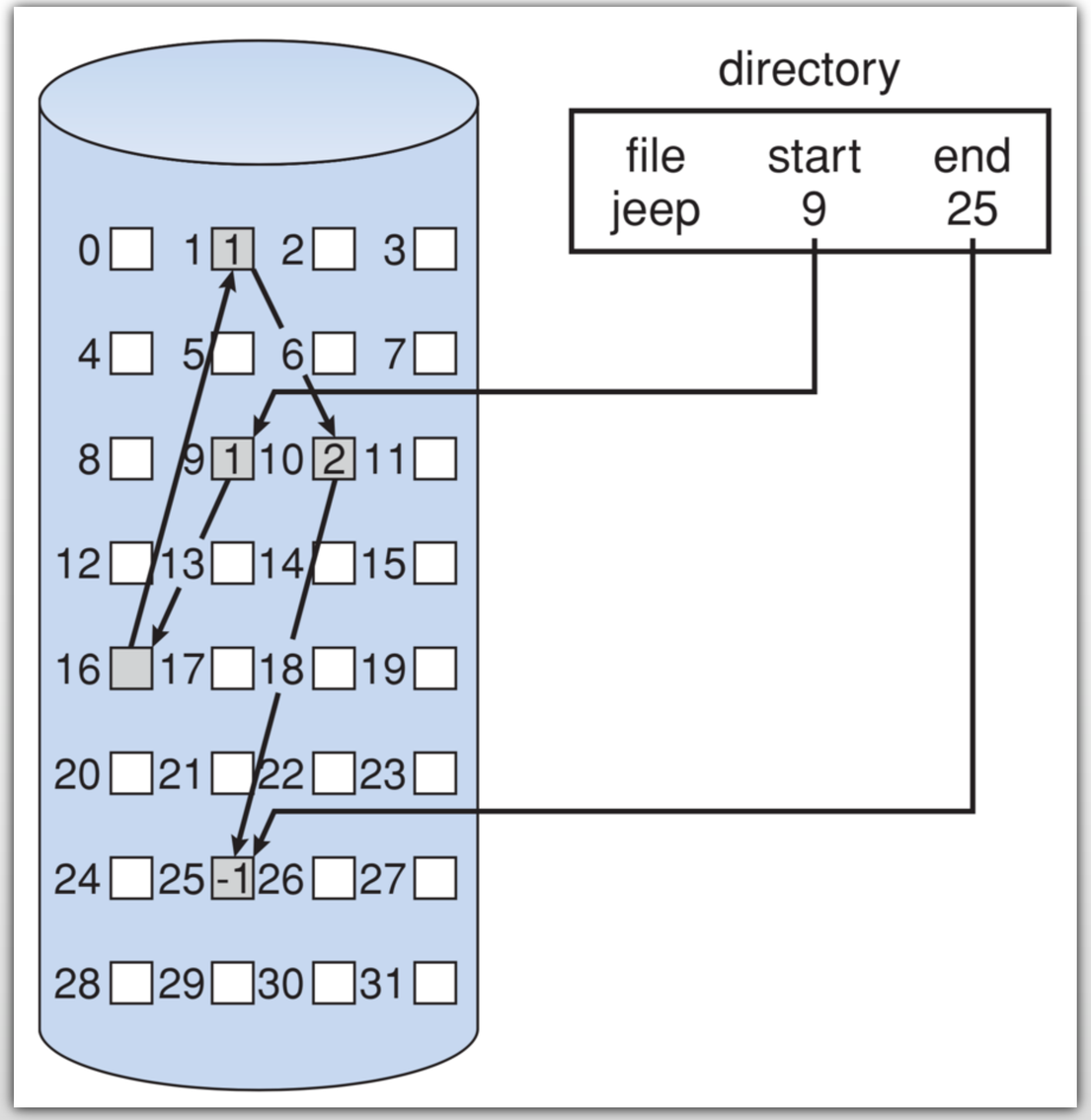


directory

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

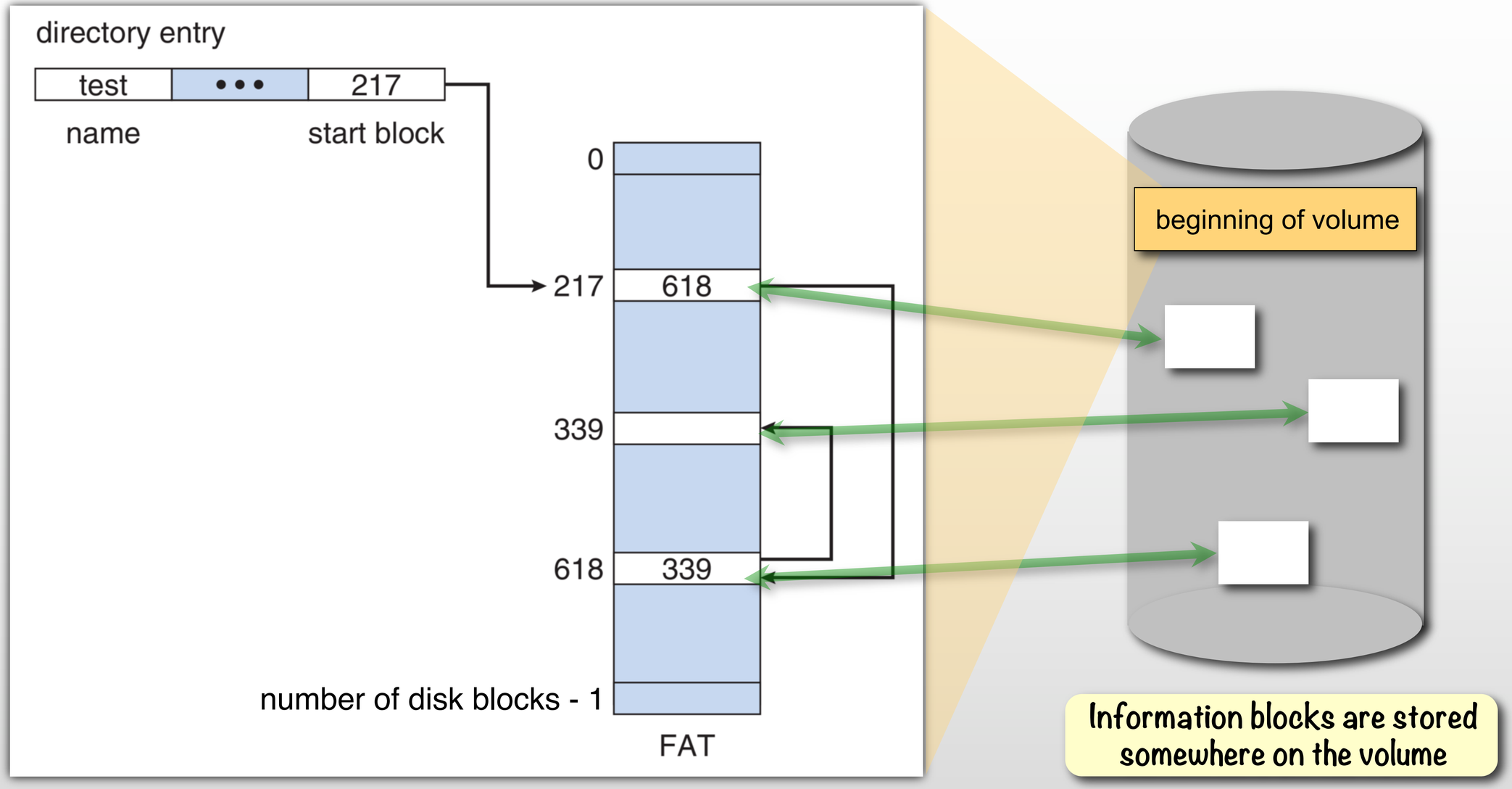
Linked Allocation

- Each file is a linked list of disk blocks
- Blocks may be scattered anywhere on the disk
- Simple
  - need only starting address
- Free-space management system
  - no waste of space
- However
  - no random access
    - must traverse the whole list to get anywhere
  - very difficult to recover from errors
    - e.g., a link can be broken



Example: FAT (File-Allocation Table: DOS, OS/2)

- Linked Allocation

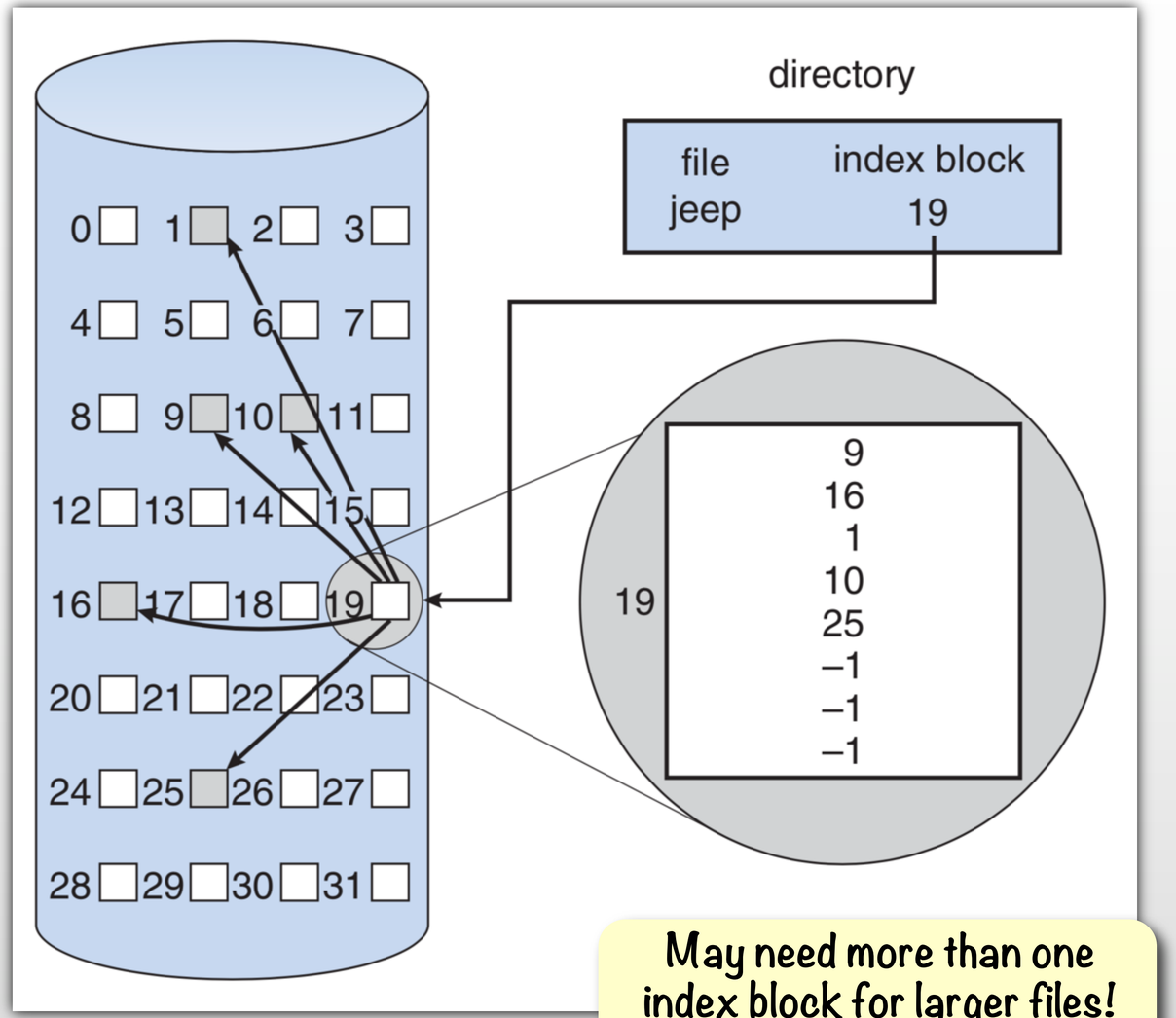


Hybrid Organization: Extent-Based Systems

- A modified contiguous allocation scheme
  - e.g., macOS HFS+, APFS, brtfs, some elements in ext4, zfs as well
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous set of blocks
  - extents are allocated when the original allocation is not sufficient due to the file growth
  - a file consists of one or more extents
    - arranged in a linked list
- Addresses the main problems with the contiguous allocation: fragmentation and efficiency
  - no rewrites to new location necessary
- Variable size extents can be used depending on the size of the file
  - the larger the file, the larger the extent
    - can start with a small extent of just one block, for example; then, grow with larger extents as needed
  - limits internal fragmentation
- File usage statistics of paramount importance!

Indexed Allocation

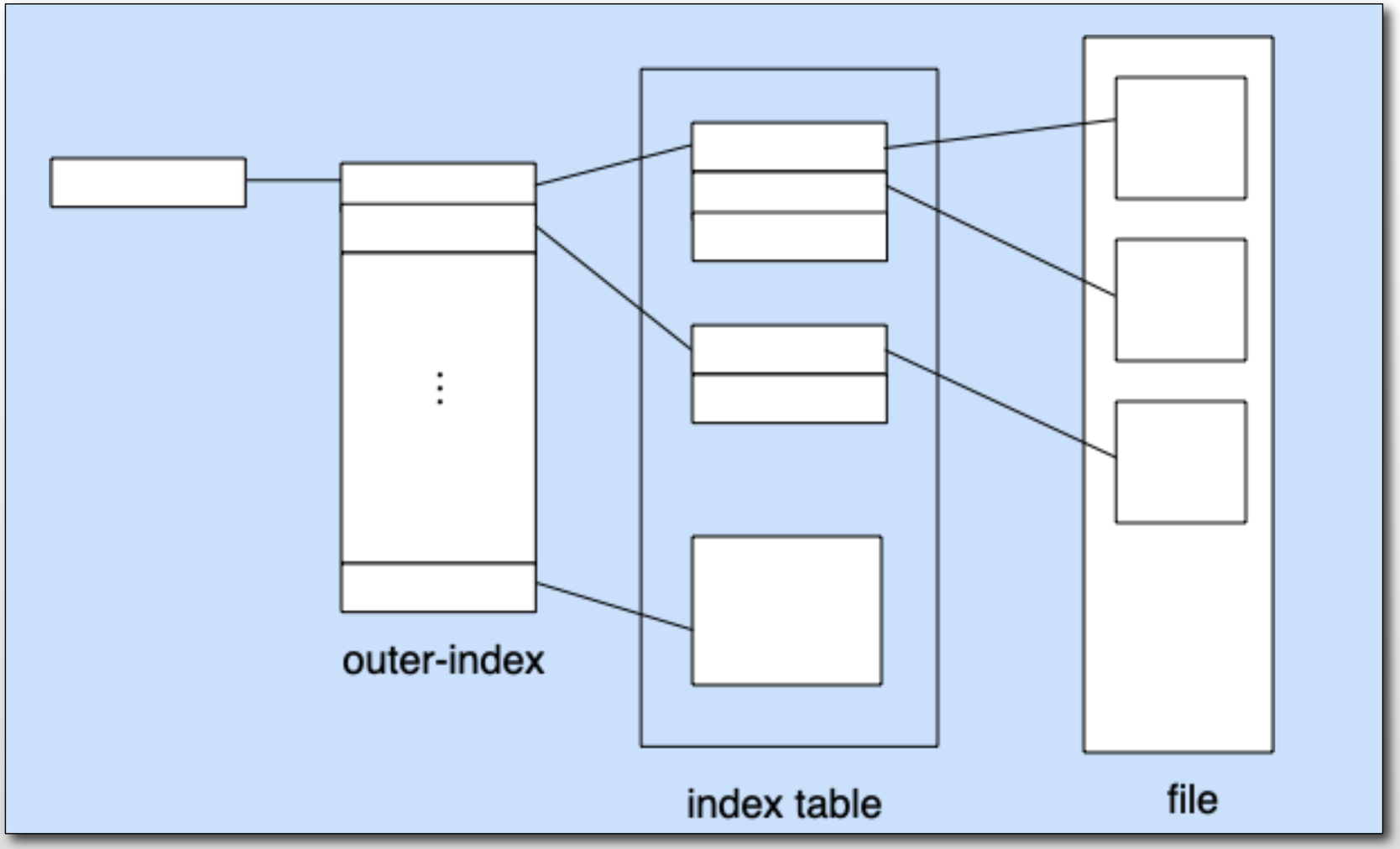
- Brings all pointers together into the index block
  - probably the genesis of inode name
- Need index table
- Random access
- Dynamic access without external fragmentation
- Overhead of index block
  - multi-level indexing helps
    - index table to another index table to another index table… and finally data blocks
  - need to balance fragmentation and access time for many indices



May need more than one index block for larger files!

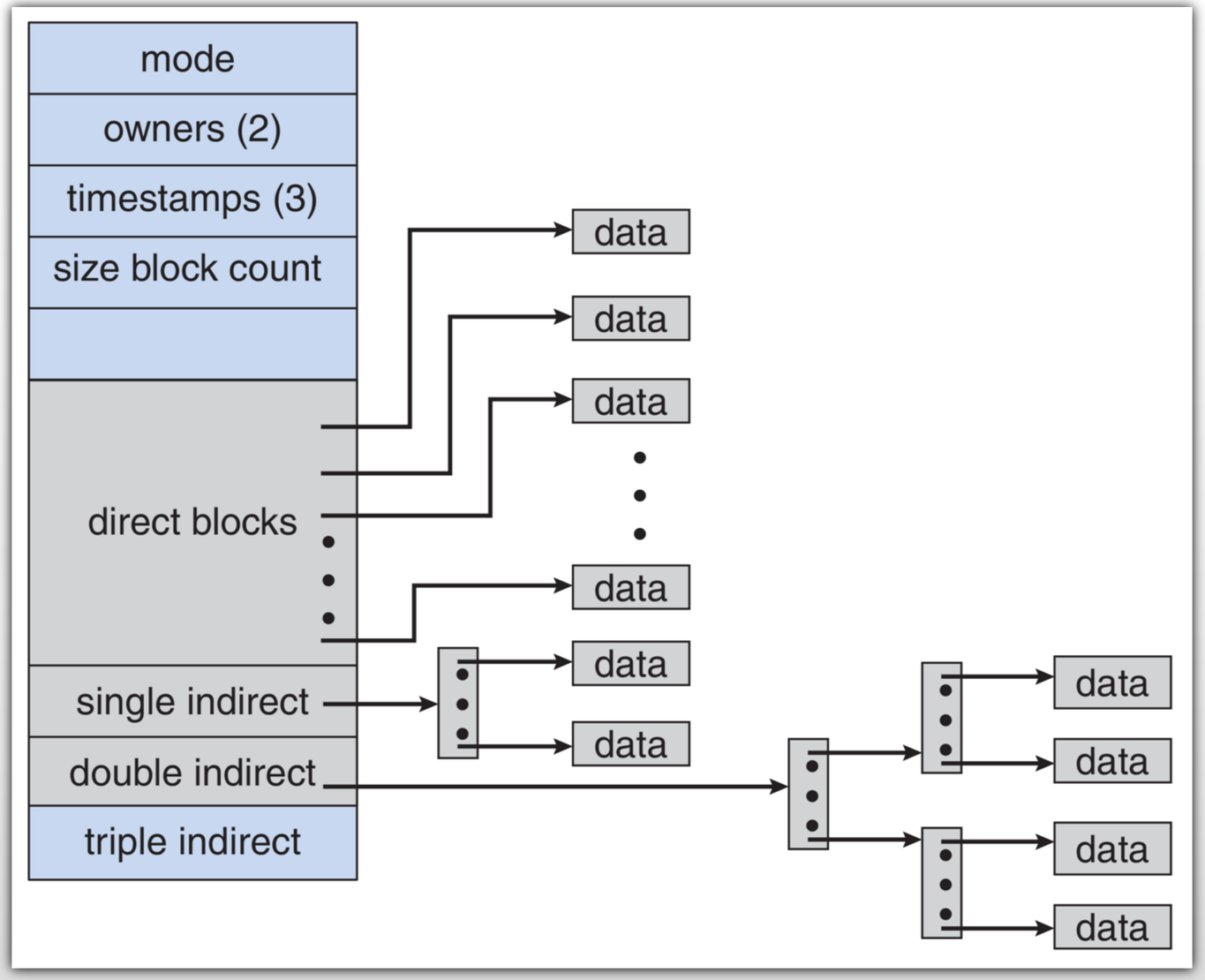
Two-level Indexing

- Example of multi-level indexing for larger files



Combined Scheme Allocation Scheme

- Contiguous allocation for small number of blocks
  - effective for small files
- Multi-level indexing for larger files
- Efficient for files of all sizes
- E.g., ext3



Free-Space Management

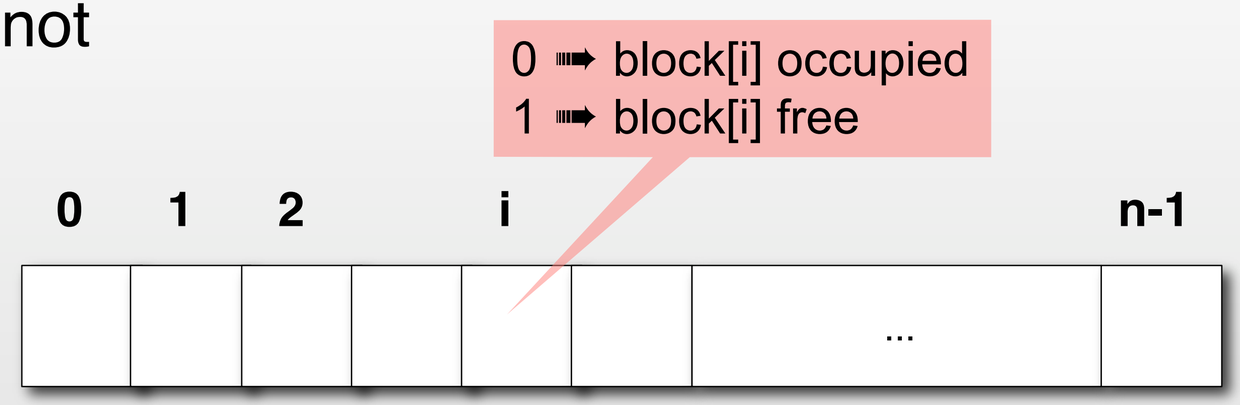
- How do we know which parts of the storage are free?
  - i.e., not allocated to any file, and – therefore – available for new allocations

- Fee space management usually done through either
  - **bit vector**
  - **linked list**
  - **grouping**
  - **counting.**

Bit Vector-Based Free-Space Management

- Fee space management usually done through either
  - **bit vector**, or
  - **linked list**

- Bit vector is a vector of bits that represent all blocks in the file system
- Each bit indicates if the block is free or not
  - e.g., for n blocks



    - efficient computation of the location of the bit for a specific block
    - fast lookup for free blocks
      - e.g., if 0 means taken, and 1 means free, can quickly iterate until non-zero word in the vector, and then shift bits

```
blockNumber = numberOfBitsPerWord * numberOfZeroValueWords + offsetOfFirstOneBit
```

Bit Vector Challenges

- Easy to get contiguous files, but…
- Requires extra space (in memory and on disk)
- Problem: Increasing size of disks
- Example:

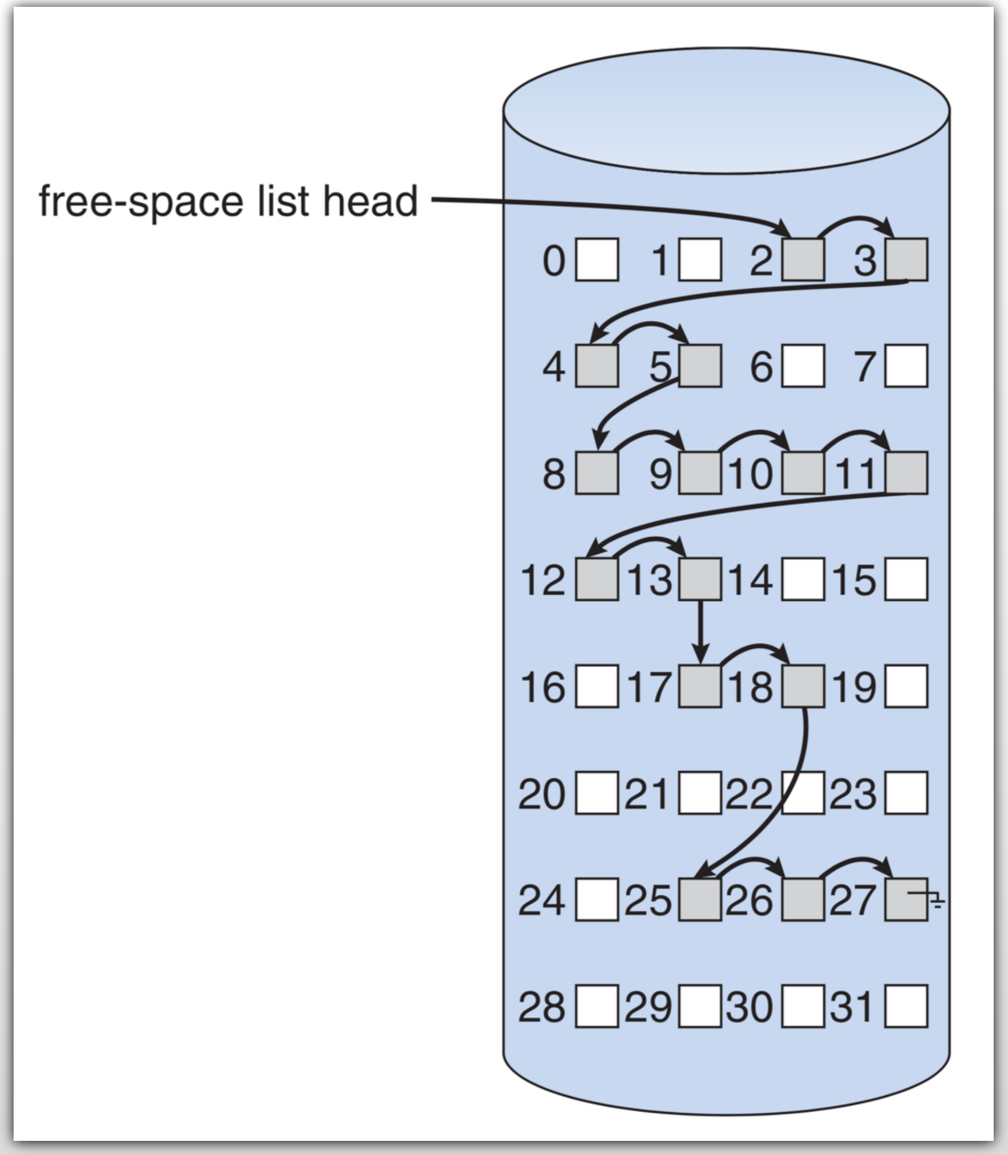
Say we have block size = 2^{12} bytes (4096 bytes or 4kB), and n is the size of the bitvector. Then for

- disk size = 2^{30} bytes (1 GB): n = $2^{30}/2^{12} = 2^{18}$ bits (or 256 KB)
- disk size = 2^{44} bytes (16 TB): n = $2^{44}/2^{12} = 2^{22}$ bits (or 2 GB)

- Possible solution (e.g., ext4, zfs):
  - divide blocks into groups and have bit vectors for groups
  - try to keep files inside a single group

Linked List-Based Free Space Management

- Linked list
  - traversals are expensive
    - access time!
    - fortunately, traversing not needed frequently
      - usually the first block in the list will do
      - it's a list of free blocks!
- No space wasted
- Hard to get contiguous space
  - contiguous space speeds up reading and writing
    - kernels may need such allocation
      - e.g., for swapping memory pages



Grouping and Counting

- **Grouping**
  - linked list of groups of free blocks
  - the last block in a group holds the address of the next group
    - comparable to the extent-based file allocation; but with free blocks

- **Counting**
  - keeps addresses of groups of blocks
    - maintains a table of indices to free groups of blocks
    - index to the first block in a group along with the number of free contiguous blocks in the group

Data Consistency in File Systems

- Need to protect free list:
  - pointer to the linked free list
  - content of the bit map
    - must be kept on disk
    - copy in memory and disk should not differ
      - cannot allow for a block to have a situation where bit[i] = 1 in memory and bit[i] = 0 on disk

- Consistency checking
  - compares information in meta-data with data blocks on disk, and tries to fix inconsistencies
  - usually must be run offline (but not in zfs)

UNIX

```
$ man fsck
$ sudo fsck -y
```

```
C:> chkdsk
```

DOS/Windows

Ensuring File System Consistency

- Mechanism to ensure file system consistency include
  - **log-structured file systems**
  - **journaling**
    - based on log-structured file systems
  - **copy-on-write file systems**

Log-Structured File Systems

- Log-structured file systems write information to media into new blocks
  - no rewriting
  - along with the meta data
- The file structure is a circular list buffer
  - new or modified information is added always at the head
  - when the new version is written, then the old data is marked as obsolete
- A “garbage collector” is run to make room at the tail
  - two cases:
    - information obsolete: just move on the tail and make the obsolete blocks available
    - information valid: rewrite to the head, then move the tail
  - disk full if no blocks with obsolete information
- If the file system crashes, then the system is guaranteed to remain consistent
  - because old data is still there
    - the old data is made obsolete only if the new data successfully written
- Logging may be done for parts of a file system; e.g., slabs in zfs

Journaling File Systems

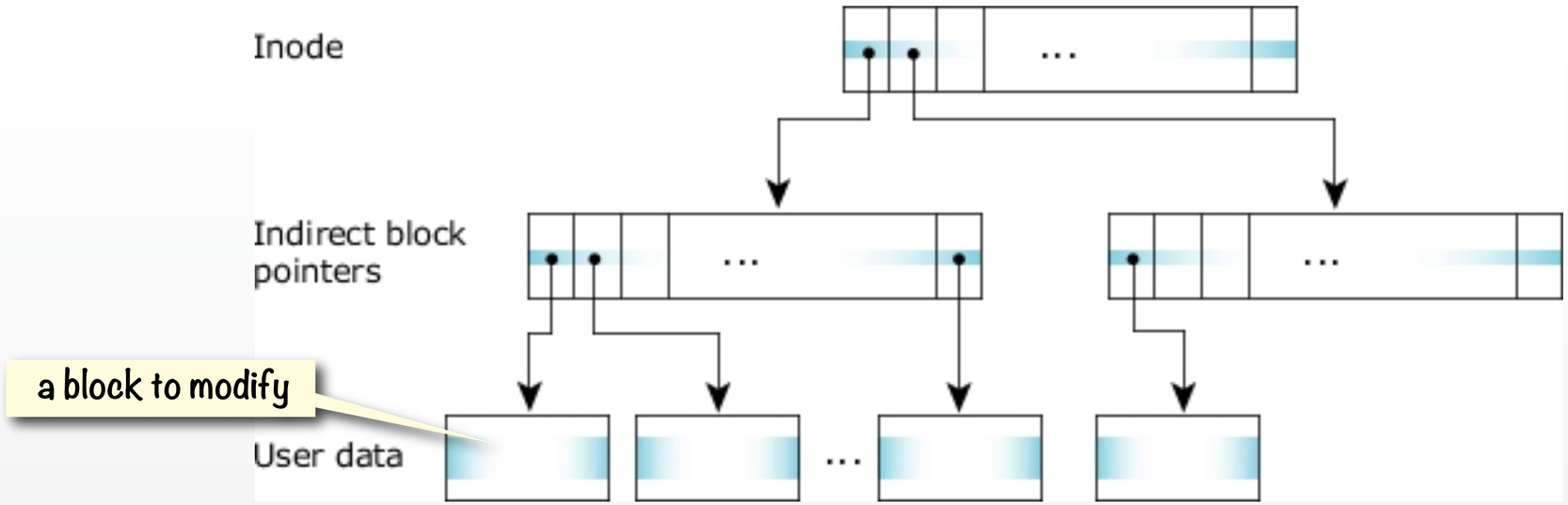
- Provides data consistency protection (e.g. due to a power down)
- Journaling file systems record each update to the file system as a transaction
  - circular writes ideas similar to log-structured file systems, but only transactions are logged, and not data
- All transactions are written to a log (a “to do” list)
  - log is a part of the medium committed for that purpose (e.g., a special file)
  - a transaction is considered committed once it is written to the log (i.e., made an item in the “to do” list)
    - “committed” means “ready to be processed”
      - as far as the process requesting the operation is concerned, the operation is done
  - however, the file system is not updated until the transaction is processed
    - i.e., the actual data referred to from the transactions in the log (i.e., in the TODO list) have not been modified
- The transactions in the log (i.e., in the TODO list) are processed asynchronously and the corresponding data are written to the file system
- Only after the file system has been in fact modified (the data have been written), the transaction is removed from the log (i.e., the item is removed from the “to do” list)

Recovery in Journaling File Systems

- If the system (or just the file system) crashes, all transactions in the log (i.e., all items in in the “to do” list) will still be processed
  - the transactions are processed on the system restart
  - if transactions are committed (i.e., meta-data along with the actual data), then the system is guaranteed to remain consistent
  - all transactions partially logged (not committed) are not executed

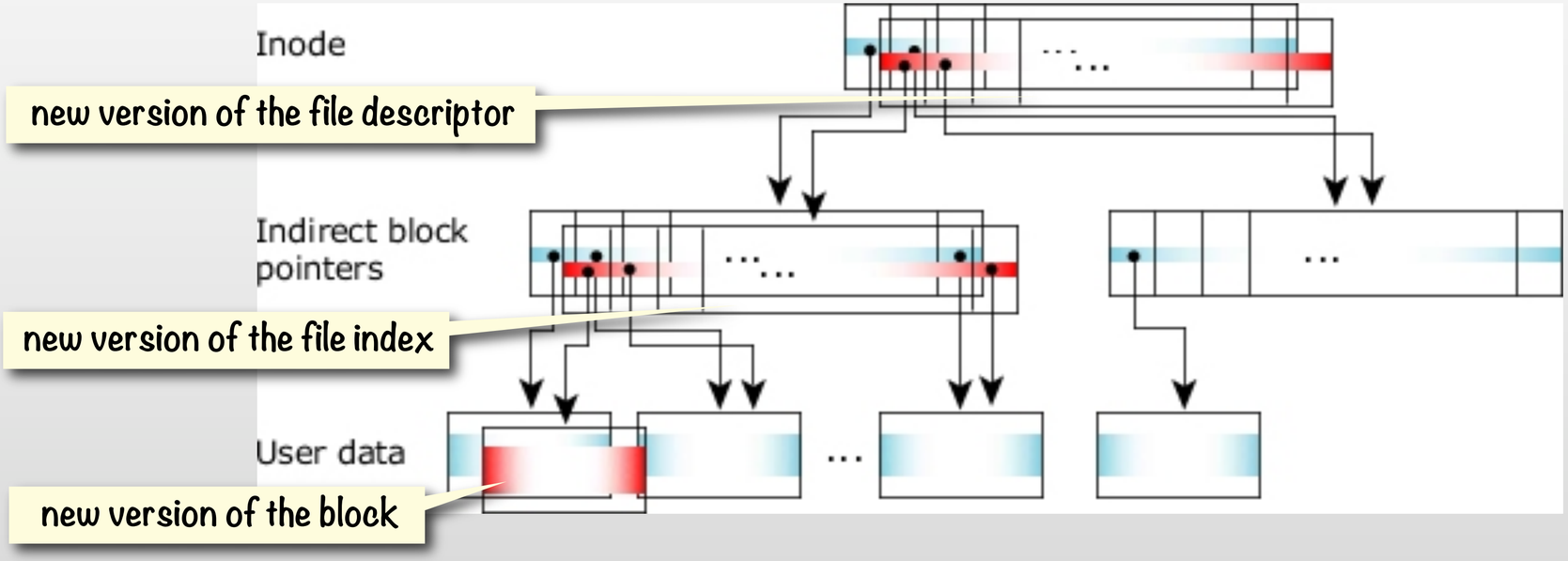
- Many modern file systems have such capabilities
  - e.g., ext4, NTFS

Copy-On-Write File System



before changes

you will implement this technique in project #1



after changes

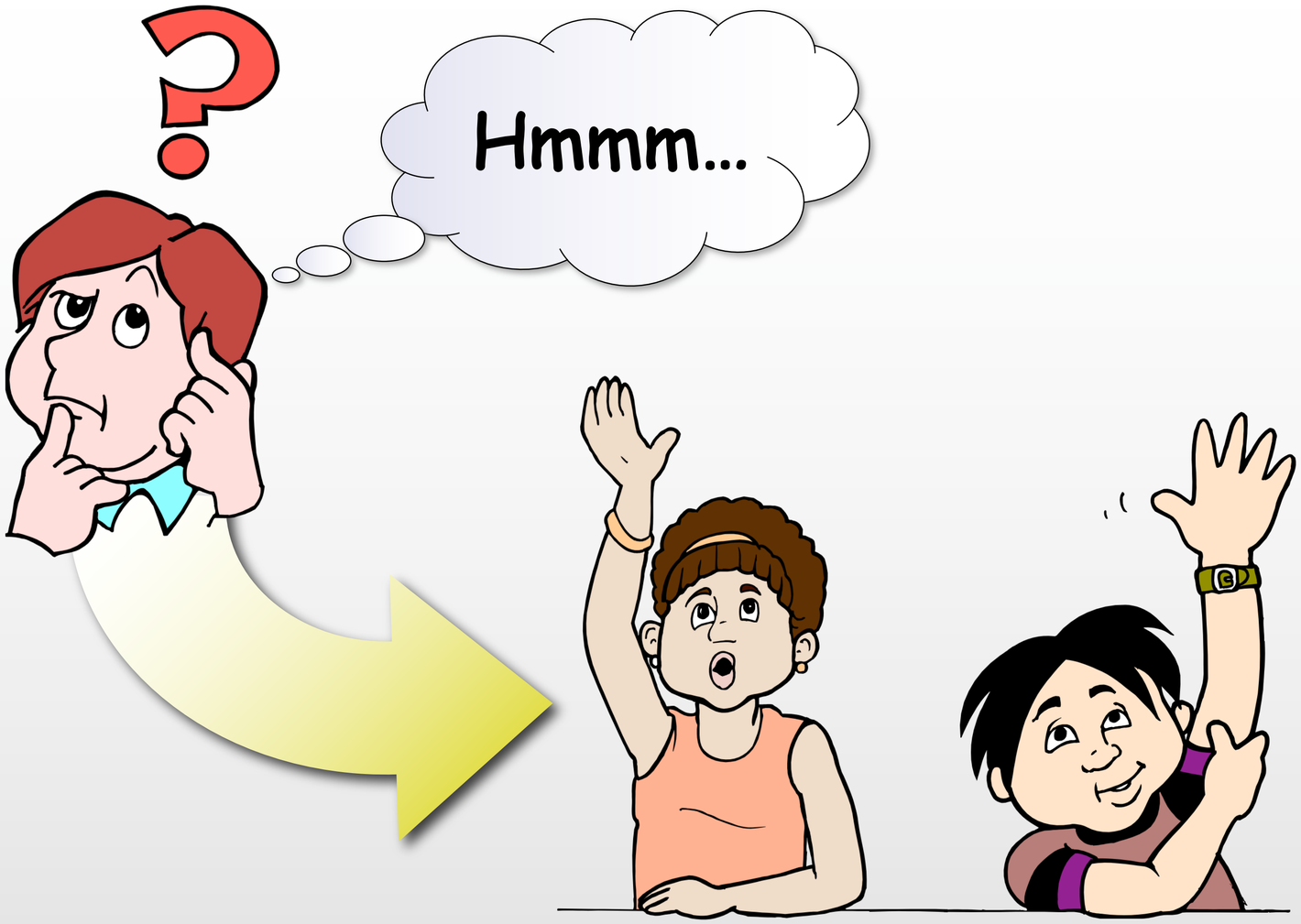
old

new

Explore Further...

- There are a lot of file systems in existence
  - file system entry
  - list of file systems
  - comparison of file systems

- See the Wikipedia links for most important:
  - NTFS (Windows)
  - ReFS (Windows)
  - HFS+ (Mac OS X)
  - APFS (Mac OS X)
  - etx3/ext4 (Linux)
  - zfs (Solaris, zfs on FUSE)
  - brtfs (Oracle)



COMP362 Operating Systems

Prof. AJ Bieszczad