ILLINOIS TECH

College of Computing

# CS 450 Operating Systems
# Introduction to File System

Yue Duan

# Recap: The File Abstraction

- A file is a named assembly of data.
  - Each file comprises:
    - data – information a user or application stores
    - metadata – information added / managed by OS
  - Need name to access
    - unique id: inode number
    - path
    - file descriptor

# Recap: File Data vs. Metadata

- file data:
  - array of untyped bytes
  - implemented by an array of fixed-size blocks
- metadata:
  - other interesting things OS keeps track of for each file
    - size
    - owner user and group
    - time stamps: creation, last modification, last access
    - security and access permission: who can do what with this file
- **inode** stores metadata and provides pointers to disk blocks containing file data

# Recap: inode

- internal OS data structure representing a file
- inode stands for index node, historical name used in Unix
- Each **inode** is identified by its index-number (inumber)
- Each file is represented by exactly one **inode** in kernel
- Both **inode** as well as file data are stored on disk

# Recap: Directory

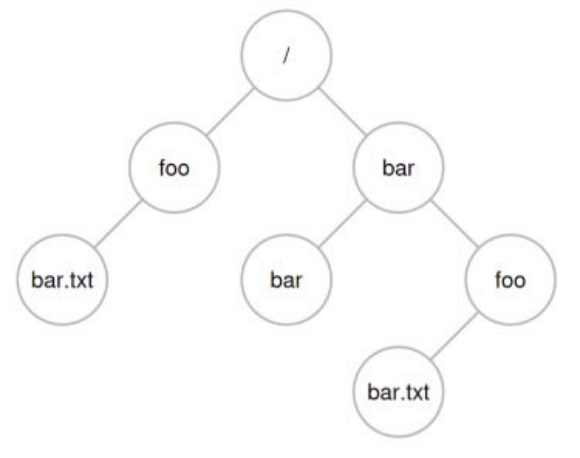
- A special file used to organize other files into a hierarchical structure
  - each directory is a file in its own right, so it has a corresponding inode
- Most file systems support the notion of a current directory
  - **Absolute names** starting from the root of directory tree
    - /bar/foo/bar.txt
  - **Relative names** specified with respect to current directory
    - ./bar.txt

# Recap: Directory Internals

- A directory is a list of entries
  - <name, inumber> pairs
  - name is just the name of the file or directory
  - inumber depends upon how file is represented on disk
  - internal format determined by the FS implementation
- Directory entry:
  - each <name, inumber> pair
  - called a **dentry** in Linux

# Recap: Directory Hierarchy

- Each dentry can point to a normal file or a another directory
- This allows hierarchical (treelike) organization of files in a file system.
- In this tree, all internal nodes are directories and leaves are ordinary files.



# Recap: Links

- Two types
  - hard links:
    - both path names use same inode number
  - soft links, a.k.a, symbolic links:
    - contains a separate inode value that points to the original file

# File Operations: open()

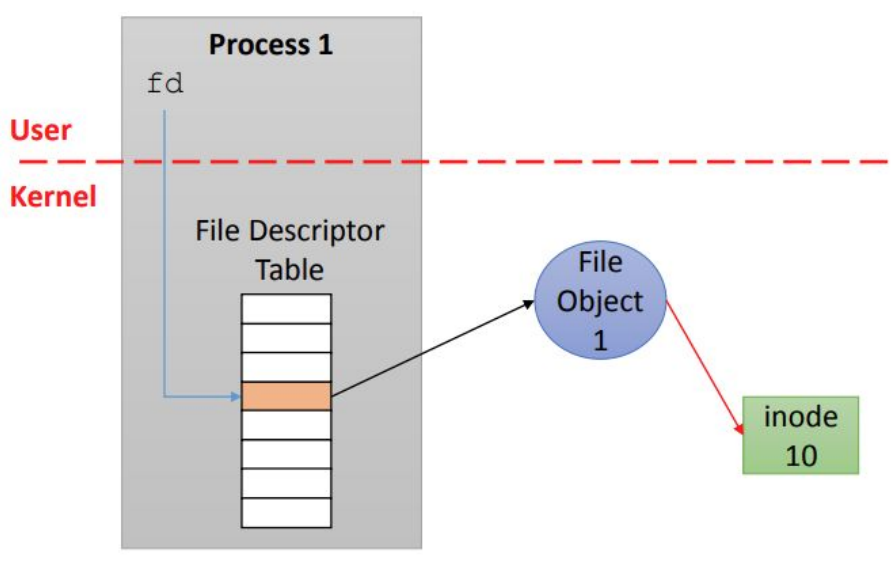
- int open(char *path, int flags, int mode);
- Traverse the directory tree
  - find the **dentry** corresponding to path
- Check a lot of things according to flags
- Examples of flags:
  - O_RDONLY, O_WRONLY, O_RDWR: requested type of access to file
  - O_CREAT: create if not existing
  - And many others; see the man page
- *mode* is used to set the file permissions if a new file is

# File Operations: open()

- If path is valid and requested access is permitted, open() returns a **file descriptor**
  - an index into the per-process **file descriptor table (FDT)**
  - FDT is a kernel data structure; user program only has a index into it
- Each entry in file descriptor table is a pointer to a **file object**
  - file object represents an instance if an opened file
  - file object then points to the inode

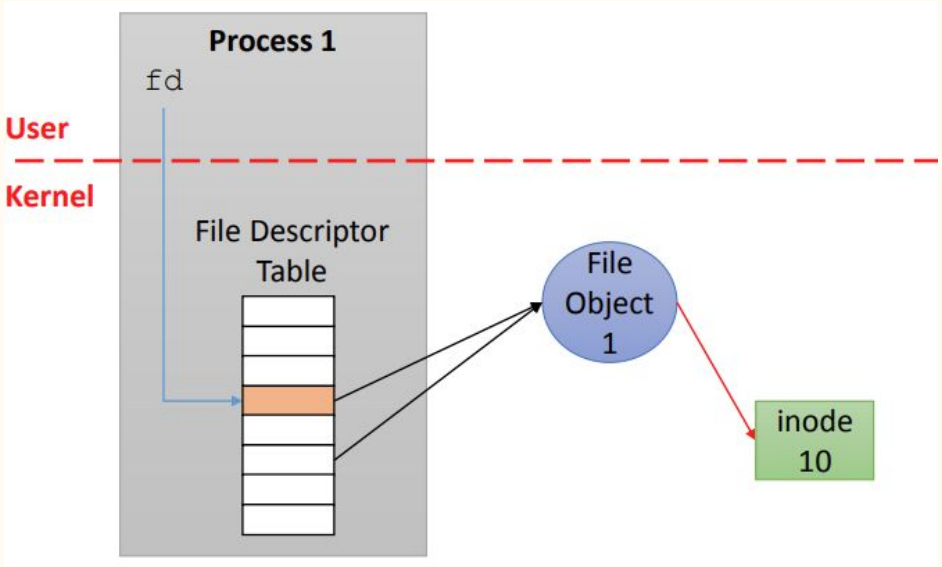
# File Descriptors and File Objects

- *fd* indexes into **FDT**; FDT entry points to **File Object**
- File object points to corresponding inode



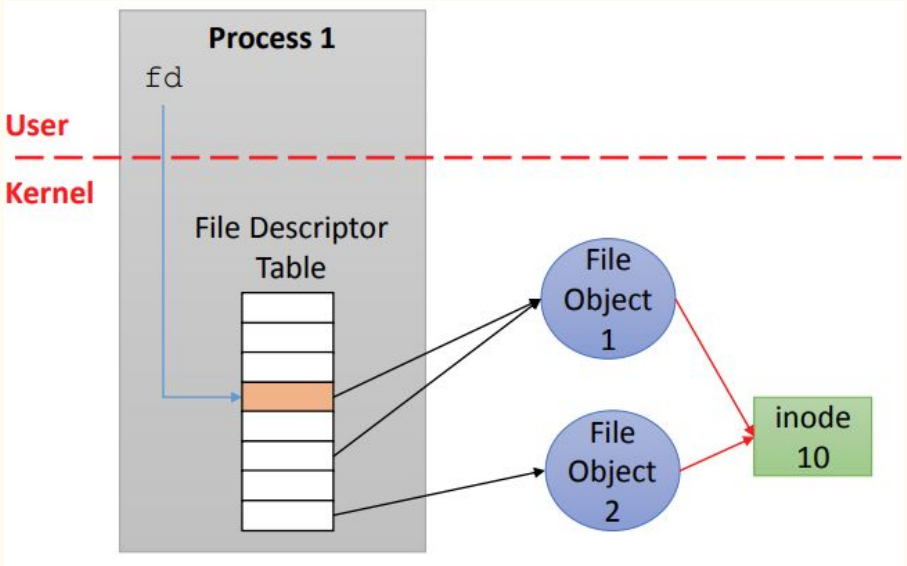
# File Descriptors and File Objects

- Multiple entries in same FDT may point to same file object
  - e.g., after a dup() syscall



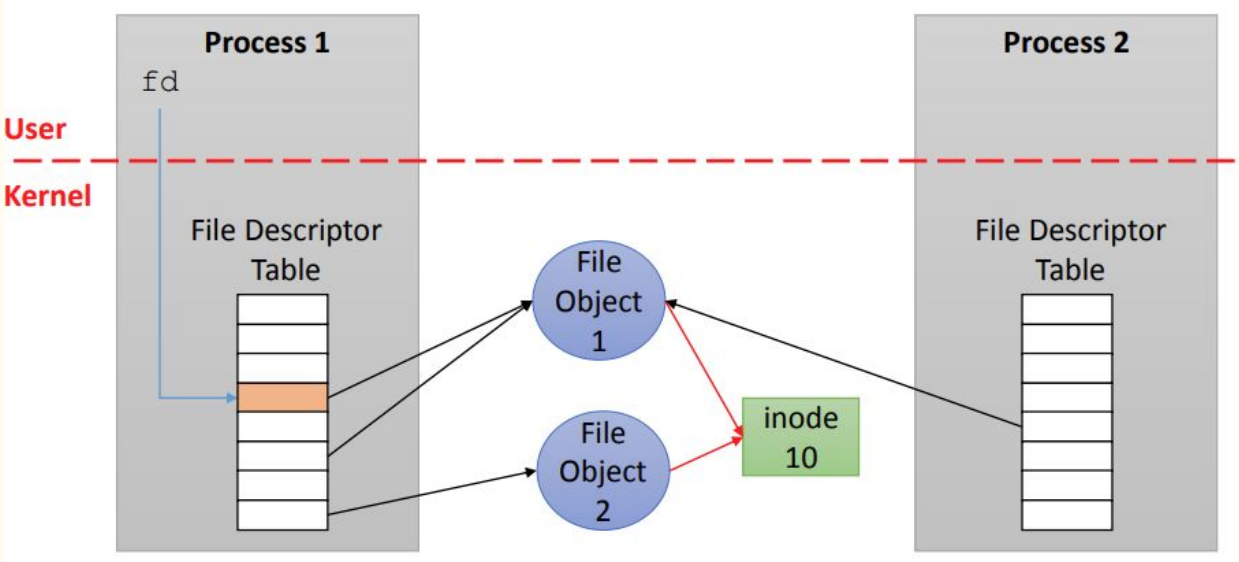
# File Descriptors and File Objects

- Multiple file objects might point to the same inode
  - e.g., if the file has been opened multiple times
  - either by the same process or a different one

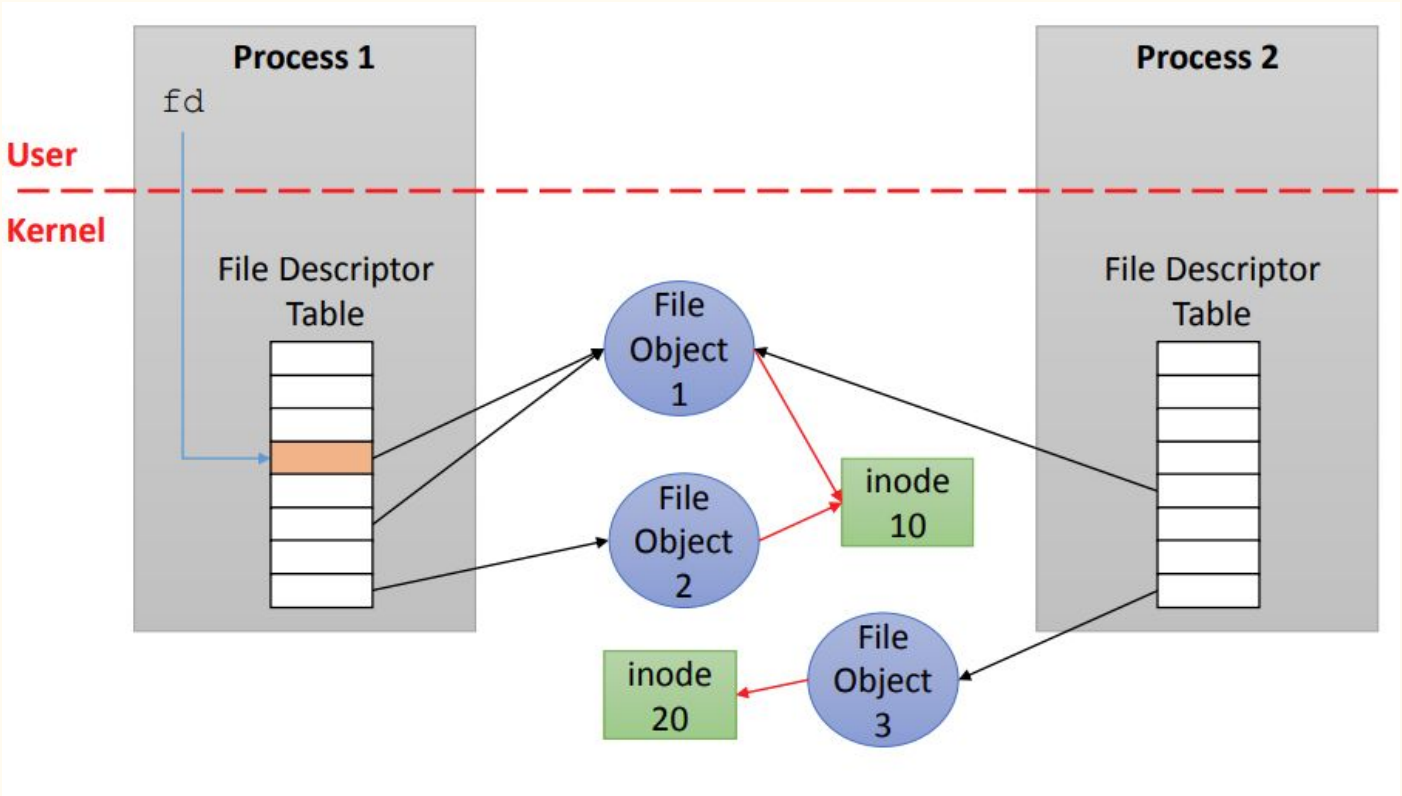


# File Descriptors and File Objects

- The same file object might be pointed to by FDTs of different processes
  - e.g., due to fork()
  - FDT gets copied at fork time.



# File Descriptors and File Objects



# Why File Objects?

- Why don’t FDT entries directly point to inodes?
- Because each time you open a file, you might use different flags
  - e.g., different permission requests
- Kernel tracks the “current offset” of each open file
  - multiple open instances of the same file may be accessing the file at different offsets

# Core FS Objects

- **inode**:
  - represents a file
  - keeps metadata as well as pointers to data blocks
- **dentry**(directory entry):
  - name-to-inode mapping
- **file object**:
  - represents an opened file
  - keeps pointer to inode (or dentry), access permissions, and file offset

# Core FS Objects

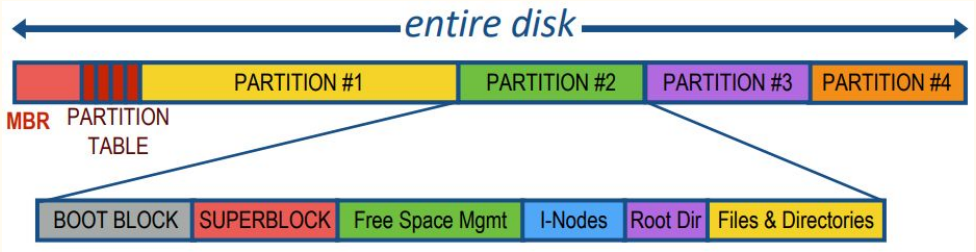
- **Superblock**:
  - global metadata of a file system
  - e.g., a magic number to indicate FS type
  - e.g., allocation bitmaps to find free inodes and data blocks
  - many file systems put this as first block of partition
- Superblocks, inodes and dentries are stored on-disk
  - and cached in memory when accessed
- File object is only in memory

# File System General Layout

- File systems define block size (e.g., 4KB)
  - Disk space is allocated in granularity of **blocks**
- A **Master Block** determines location of root directory
  - at fixed disk location
- A **free map** determines which blocks are free, allocated
  - usually a bitmap, one bit per block on the disk
  - also stored on disk, cached in memory for performance
- Remaining blocks store files (and dirs), and swap

# Disk Layout

- File System is stored on disks
  - sector 0 of disk called Master Boot Record (MBR)
    - boot loader
    - partition table (partitions' start & end addrs)
  - remainder of disk divided into partitions
    - each partition starts with a boot block
    - boot block loaded by MBR and executed on boot
    - remainder of partition stores file system



# File Storage Layout Strategies

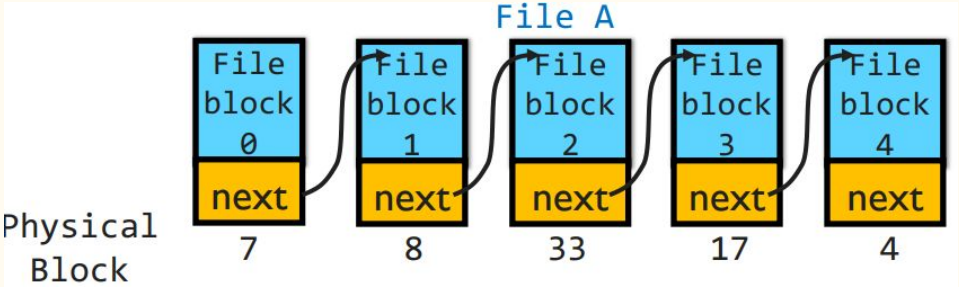
- Files span multiple disk blocks
  - contiguous allocation
    - all bytes together, in order
  - Linked structure
    - each block points to the next, directory points to the first
  - Indexed structure
    - an **index block** contains pointers to many other blocks
    - may need multiple index blocks (linked together)

# Contiguous Allocation

- All bytes of file are stored together, in order
  - + simple: state required per file: start block & size
  - + efficient: entire file can be read with one seek
  - – fragmentation: external fragmentation is bigger problem
  - – usability: user needs to know size of file at time of creation

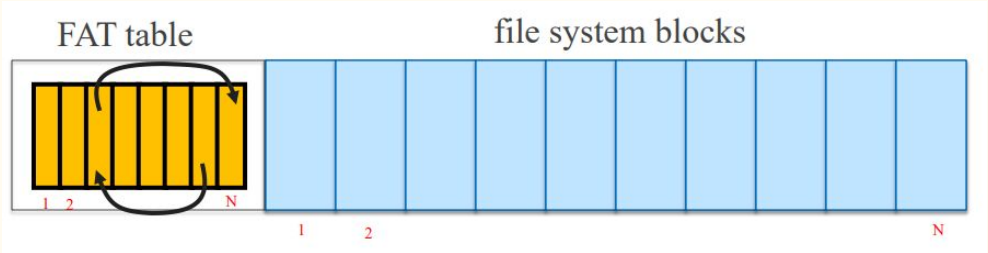
# Linked-List File Storage

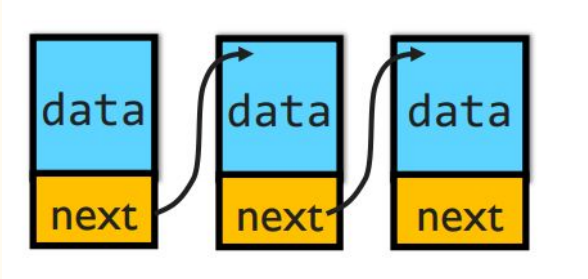
- Each file is stored as linked list of blocks
  - first word of each block points to next block
  - rest of disk block is file data
  - + space utilization: no space lost to external fragmentation
  - + simple: only need to store 1st block of each file
  - – performance: random access is slow
  - – space utilization: overhead of pointers



# FAT File System

- File Allocation Table (FAT)
  - Used in MS-DOS, precursor of Windows
  - Still used (e.g., CD-ROMs, thumb drives, camera cards)
  - FAT-32, supports $2^{28}$ blocks and files of $2^{32}$-1 bytes
  - The FAT Table
    - a linear map of all blocks on disk
    - each file is a linked list of blocks
      - with metadata located in the first block of the file

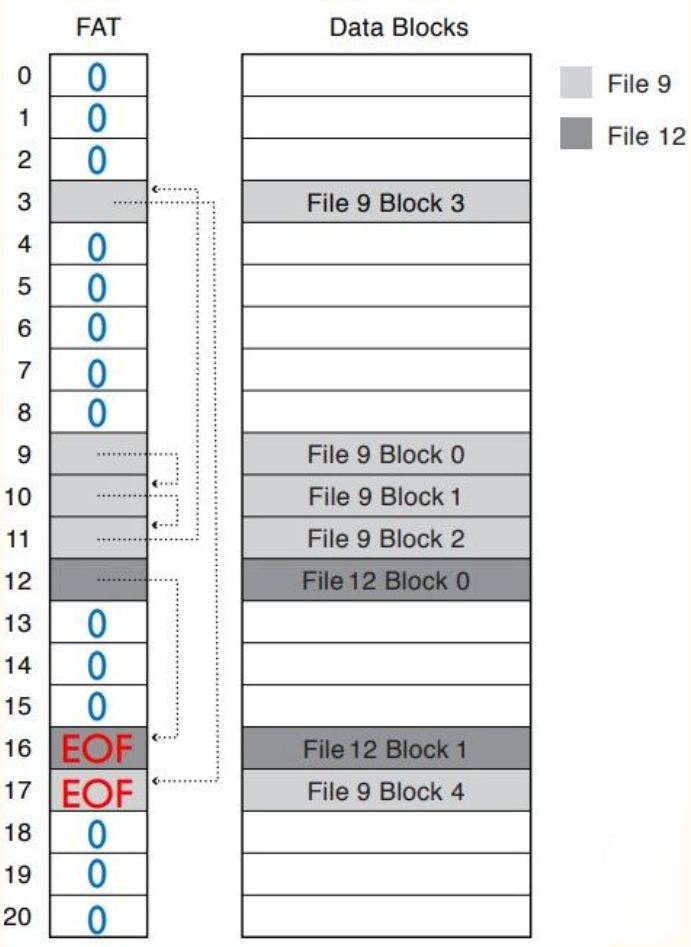


# FAT File System

- 1 entry per block
- EOF for last block
- 0 indicates free block
- directory entry maps name to FAT index

| Directory | |
|---|---|
| bart.txt | 9 |
| maggie.txt | 12 |
| | |



# How is FAT?

- + simple: state required per file: start block only
- + widely supported
- + no external fragmentation (**what about internal?**)
- + block used only for data
- + can grow file easily
- – poor random access
- – limited metadata
- – limited access control
- – limitations on volume and file size

# THANK YOU!