ILLINOIS TECH

College of Computing

CS 450 Operating Systems
Introduction to File System

Yue Duan

File System

- **Goals**
  - scale
  - persistence
  - access by multiple processes
- **File System**
  - Interface provides operations involving
    - Files
    - Directories (a special kind of file)

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

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

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)
  - similar to processes being identified by their PID
- Each file is represented by exactly one **inode** in kernel
- Both **inode** as well as file data are stored on disk

Files

- A file can also have a type
  - understood by the file system
    - block, character, device, portal, link, etc.
  - understood by other parts of the OS or runtime libraries
    - executable, dll, souce, object, text, etc.
- A file's type can be encoded in its name or contents
  - Windows encodes type in name
    - com, .exe, .bat, .dll, .jpg, etc.
  - Unix encodes type in contents
    - magic numbers, initial characters (e.g., #! for shell scripts)

Some Basic File Operations

- **Unix**
  - creat(name)
  - open(name, how)
  - read(fd, buf, len)
  - write(fd, buf, len)
  - sync(fd)
  - seek(fd, pos)
  - close(fd)
  - unlink(name)

- **Windows NT**
  - CreateFile(name, CREATE)
  - CreateFile(name, OPEN)
  - ReadFile(handle, ...)
  - WriteFile(handle, ...)
  - FlushFileBuffers(handle, ...)
  - SetFilePointer(handle, ...)
  - CloseHandle(handle, ...)
  - DeleteFile(name)
  - CopyFile(name)
  - MoveFile(name)

Directory

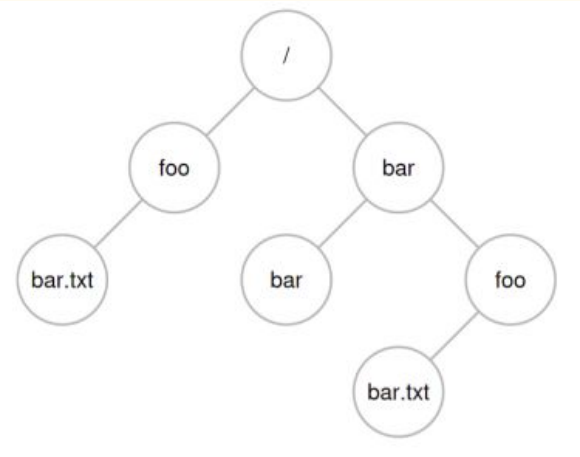
- A special file used to organize other files into a hierarchical structure
  - each directory is a file in its own right, so it has a corresponding inode
- Directories serve two purposes
  - a structured way to organize files for users
  - a naming interface to abstract away physical hardware details
- 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

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

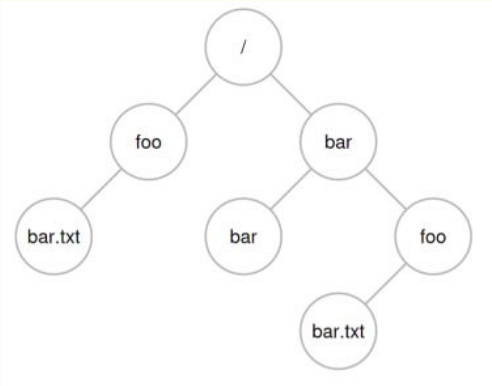
Directory Hierarchy

- 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.



File Path

- File path is the human-readable string used to refer to a node in directory tree.
- Examples:
  - /
  - /foo
  - /bar/foo/bar.txt
- Each valid path corresponds to **exactly one** dentry
  - a dentry points to **exactly one** inode
- Multiple dentries can point to the same inode
  - multiple paths might map to the same file



Path Name Translation

- Assume we want to open /one/two/three
- File system
  - open directory ‘/’
  - search for the entry ‘one’, get location of ‘one’ in dentry
  - open directory ‘one’, search for ‘two’, get location of ‘two’
  - open directory ‘two’, search for ‘three’, get location of ‘three’
  - open file ‘three’
- Systems spend a lot of time walking directory paths
  - OS will cache prefix lookups for performance
    - /a/b, /a/bb, /a/bbb, etc., all share ‘/a’ prefix

Links

- Two types
  - hard links:
    - both path names use same inode number
    - file does not disappear until all hard links removed
    - cannot link directories
  - soft links, a.k.a, symbolic links:
    - similar to the file shortcut feature in Windows
    - contains a separate inode value that points to the original file
    - has only the path of the original file, not the contents.
    - can softlink to dirs

Soft Links

```
yueduan@HOMEDESK:~/test$ echo "hello" > source
yueduan@HOMEDESK:~/test$ cat source
hello
yueduan@HOMEDESK:~/test$ ln -s source softlink
yueduan@HOMEDESK:~/test$ cat softlink
hello
yueduan@HOMEDESK:~/test$
```

```
yueduan@HOMEDESK:~/test$ ls -lai
total 12
61840 drwxr-xr-x 2 yueduan yueduan 4096 Mar 30 22:48 .
  671 drwxr-xr-x 9 yueduan yueduan 4096 Mar 30 22:48 ..
62076 lrwxrwxrwx 1 yueduan yueduan    6 Mar 30 22:48 softlink -> source
62073 -rw-r--r-- 1 yueduan yueduan    6 Mar 30 22:48 source
```

```
yueduan@HOMEDESK:~/test$ rm source
yueduan@HOMEDESK:~/test$ cat softlink
cat: softlink: No such file or directory
yueduan@HOMEDESK:~/test$
```

```
yueduan@HOMEDESK:~/test$ ls -lai
total 8
61840 drwxr-xr-x 2 yueduan yueduan 4096 Mar 30 22:51 .
  671 drwxr-xr-x 9 yueduan yueduan 4096 Mar 30 22:48 ..
62076 lrwxrwxrwx 1 yueduan yueduan    6 Mar 30 22:48 softlink -> source
```

Hard Links

```
yueduan@HOMEDESK:~/test$ echo "hello" > source
yueduan@HOMEDESK:~/test$ cat source
hello
yueduan@HOMEDESK:~/test$ ln source hardlink
yueduan@HOMEDESK:~/test$ cat hardlink
hello
yueduan@HOMEDESK:~/test$ ls -lai
total 16
61840 drwxr-xr-x 2 yueduan yueduan 4096 Mar 30 22:55 .
  671 drwxr-xr-x 9 yueduan yueduan 4096 Mar 30 22:48 ..
62073 -rw-r--r-- 2 yueduan yueduan    6 Mar 30 22:54 hardlink
62073 -rw-r--r-- 2 yueduan yueduan    6 Mar 30 22:54 source
```

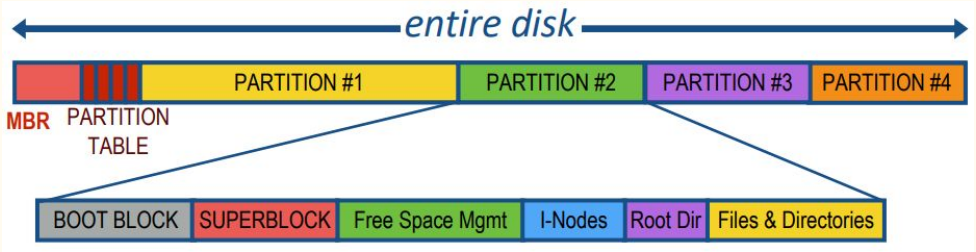
```
yueduan@HOMEDESK:~/test$ rm source
yueduan@HOMEDESK:~/test$ cat hardlink
hello
yueduan@HOMEDESK:~/test$ ls -lai
total 12
61840 drwxr-xr-x 2 yueduan yueduan 4096 Mar 30 22:56 .
  671 drwxr-xr-x 9 yueduan yueduan 4096 Mar 30 22:48 ..
62073 -rw-r--r-- 1 yueduan yueduan    6 Mar 30 22:54 hardlink
```

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

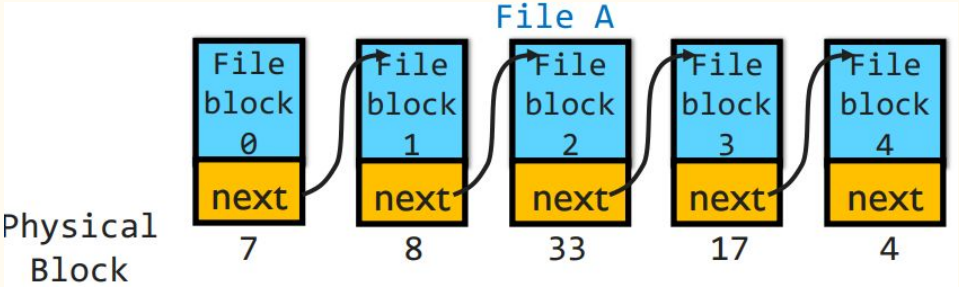
- Files span multiple disk blocks
  - contiguous allocation
    - all bytes together, in order
  - Linked structure
    - each block points to the next, directory points to the first
  - Indexed structure
    - an **index block** contains pointers to many other blocks
    - may need multiple index blocks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

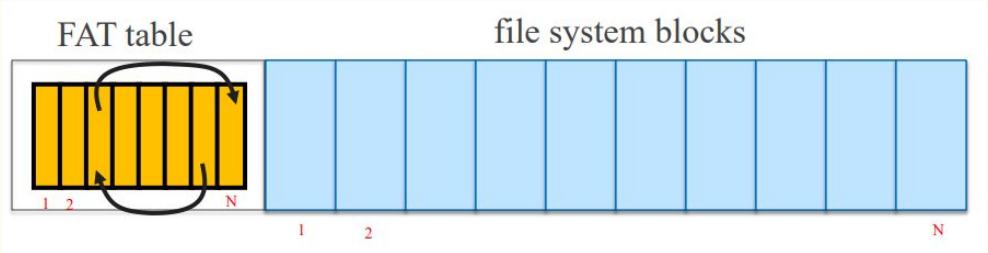
Linked-List File Storage

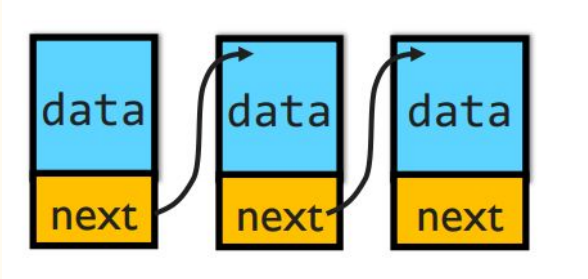
- Each file is stored as linked list of blocks
  - first word of each block points to next block
  - rest of disk block is file data
  - + space utilization: no space lost to external fragmentation
  - + simple: only need to store 1st block of each file
  - – performance: random access is slow
  - –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

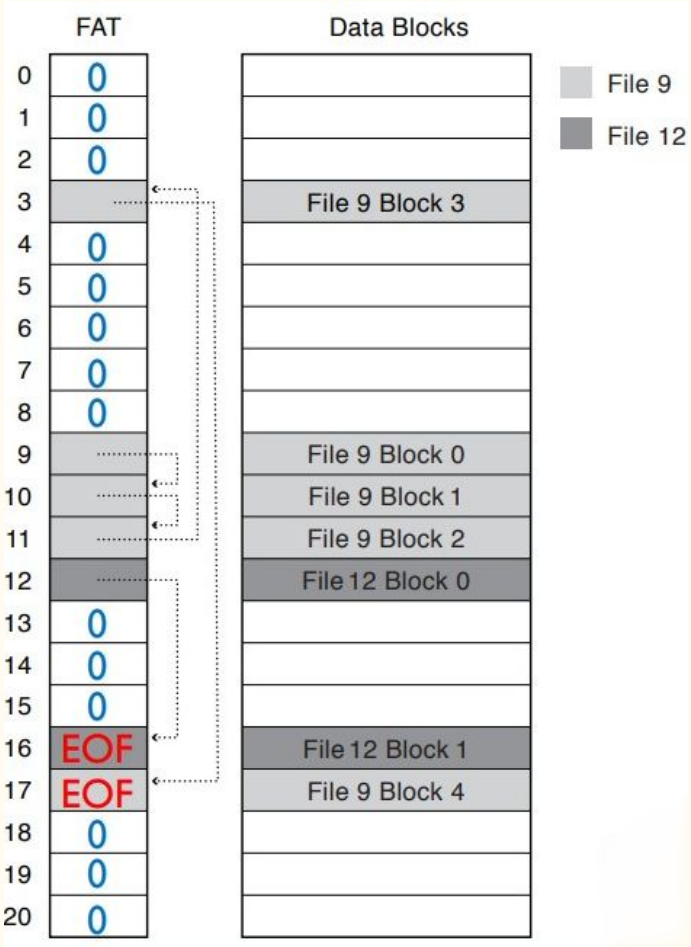




FAT File System

- 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!