File System Management

File System Introduction

Lecture 10

Overview

- **File System**
  - Definition
  - Vs Memory Management
  - Motivation
- **File**
  - Metadata
  - Operations
- **Directory**
  - Directory Structure
- **I/O Scheduling**

File System: Motivation

- Physical memory is **volatile**
  - Use external storage to store **persistent** information

- Direct access to the storage media is **not portable**:
  - Dependent on hardware specification and organization

- File System provides:
  - An abstraction on top of the physical media
  - A high level resource management scheme
  - Protection between processes and users
  - Sharing between processes and users

File System: General Criteria

- **Self-Contained**:
  - Information stored on a media is enough to describe the entire organization
  - Should be able to "plug-and-play" on another system
- **Persistent:**
  - Beyond the lifetime of OS and processes
- **Efficient:**
  - Provides good management of free and used space
  - Minimum overhead for bookkeeping information

Memory Management vs File Management

	Memory Management	File System Management
Underlying Storage	RAM	Disk
Access Speed	Constant	Variable disk I/O time
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process **Implicit** when process runs	Non-volatile data **Explicit access**
Organization	**Paging/Segmentation:** determined by HW & OS	Many different FS: ext* (Linux), FAT* (Windows), HFS* (Mac OS)etc.

Key Topics

File System Abstraction

- Discuss the logical entities present in file system
- E.g. Files / Directories

File System Implementation

- Common implementation schemes
- Discuss pros/cons
- Case studies

You mean files and folders are not real?

FILE SYSTEM ABSTRACTIONS

File System Abstraction

Secondary storage - The storage is just a bunch of 0s and 1s
- The metadata of a file (id: name, permissions) are all stored inside the file

- **File System**:
  - Consists of a collection of **files** and **directory structures**
    - **File**: An abstract storage of data
    - **Directory (Folder)**: Organization of files
  - Provides an abstraction of accessing and using the above
- Look at the two abstractions closely next:
  - **File**
  - **Directory** (Folder)

File: Overview

- Basic Definition
- File Metadata
- File Data
  - File structure
  - Access Methods
- File Operations

File: Basic Description

- Represent a logical unit of information created by process
- An ***abstraction***
  - Essentially an **Abstract Data Type:**
  - A set of common operations with various possible implementation

- Contains:
  - **Data:** Information structured in some ways

  - **Metadata:** Additional information associated with the file
    - Also known as **file attributes**

File Metadata

Name:	A human readable reference to the file
Identifier:	A unique id for the file used internally by FS
Type:	Indicate different type of files E.g. executable, text file, object file, directory etc
Size:	Current size of file (in bytes, words or blocks)
Protection:	Access permissions, can be classified as reading, writing and execution rights
Time, date and owner information:	Creation, last modification time, owner id etc
Table of content:	Information for the FS to determine how to access the file

File Name

- Different FS has different **naming rule**
  - To determine valid file name

- Common naming rule:
  - Length of file name
  - Case sensitivity
  - Allowed special symbols
  - File extension
    - Usual form `Name.Extension`
    - On **some** FS, extension is used to indicate **file type**

File Type

- An OS commonly supports a number of **file types**

- Each file type has:
  - An associated set of operations
  - Possibly a specific program for processing

- Common file types:
  - **Regular files:** contains user information
  - **Directories:** system files for FS structure
  - **Special files:** character/block oriented

Two Major Types of Regular Files

- **ASCII files:**
  - Example: text file, programming source codes, etc
  - Can be displayed or printed **as is**

- **Binary files:**
  - Example: executable, Java class file, pdf file, mp3/4, png/jpeg/bmp etc
  - Have a predefined internal structure that can be processed by specific program
    - JVM to execute Java class file
    - PDF reader for pdf file etc

Distinguishing File Type

1. **Use file extension** as indication:
   - Used by Windows OS
   - e.g. **`XXX.docx`** ➔ Words document
   - Change of extension implies a change in file type!

windows is also has case insensitive file names

2. **Use embedded information** in the file:
   - Used by Unix
   - Usually stored at the beginning of the file
   - Commonly known as **magic number**

File Protection

- Controlled access to the information stored in a file
- **Type of access:**
  - **Read:** Retrieve information from file
  - **Write:** Write/Rewrite the file
  - **Execute:** Load file into memory and execute it
  - **Append**: Add new information to the end of file
  - **Delete**: Remove the file from FS
  - **List**: Read metadata of a file

File Protection: How?

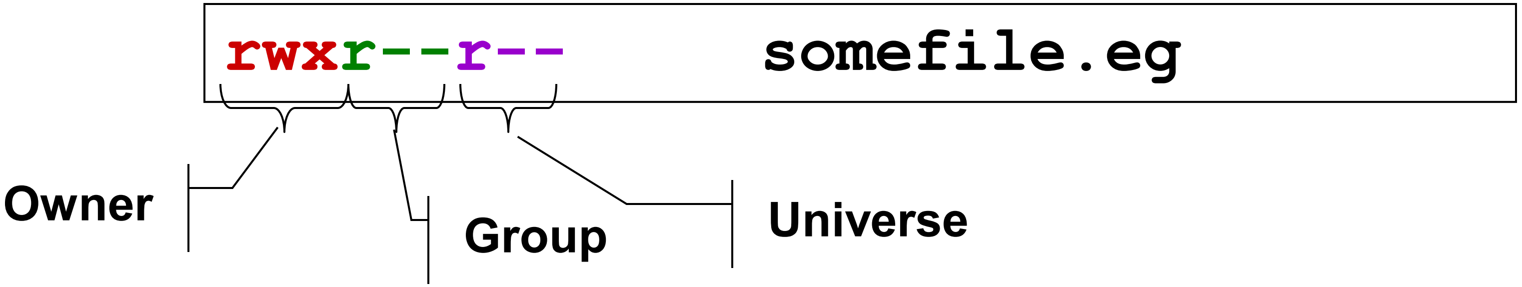
- Most common approach:
  - Restrict access base on the user identity

- Most general scheme:
  - **Access Control List**
    - A list of user identity and the allowed access types
    - **Pros:** Very customizable
    - **Cons:** Additional information associated with file

- A common condensed file protection scheme is discussed next

File Protection: **Permission Bits**

- Classified the users into three classes:
  1. **Owner:** The user who created the file
  2. **Group:** A set of users who need similar access to a file
  3. **Universe:** All other users in the system
- Example (Unix)
  - Define permission of three access types (**R**ead/**W**rite/E**x**ecute) for the 3 classes of users
  - Use "`ls -l`" to see the permission bits for a file



File Protection: **Access Control List**

- In Unix, Access Control List (ACL) can be:
  - Minimal ACL (the same as the permission bits)
  - Extended ACL (added **named users / group**)

```
$ getfacl exampleDir

# file: exampleDir
# owner: ccris
# group: compsc
user::rwx
user:sooyj:rwx
group::r-x
group:cohort21:rwx
mask::rwx
other::---
```

"getfacl" is the command to get ACL information

Permission for Specific User

Permission for Specific Group

Permission "upperbound"

Operations on File Metadata

- **Rename:**
  - Change filename

- **Change attributes:**
  - File access permissions
  - Dates
  - Ownership
  - etc

- **Read attribute:**
  - Get file creation time

File Data: Structure

- **Array of bytes**:
  - The traditional Unix view
  - No interpretation of data: **just raw bytes**
  - Each byte has a unique **offset (distance)** from the file start
- **Fixed length records:**
  - Array of records, can grow/shrink
  - Can jump to any record easily:
    - Offset of the N^{th} record = size of Record * (N-1)

this will be similar to memory management - suffering from internal fragmentation

- **Variable length records**
  - Flexible but harder to locate a record

will suffer from external framgentation

File Data: **Access Methods**

- **Sequential Access:**
  - Data read in order, starting from the beginning
  - Cannot skip but can be rewound

- **Random Access:**
  - Data can be read in any order
  - Can be provided in two ways:
    1. `Read( Offset ):` Every read operation explicitly state the position to be accessed
    2. `Seek( Offset ):` A special operation is provided to move to a new location in file
    - E.g. Unix and Windows uses (2)

can reposition based on a specific offset from 3 ways:
1: from the beginning of the file
2: from the current location of the file
3: from the end of the file

File Data: **Access Methods** (cont)

- **Direct Access:**
  - Used for file contains fixed-length records
  - Allow ***random access to any record directly***
  - Very useful where there is a large amount of records
    - e.g. In database
  - The basic random access method can be view as a special case:
    - Where each record == one byte

can easily move and reposition

File Data: Generic Operations

Create:	New file is created with no data
Open:	Performed before further operations To prepare the necessary information for file operations later
Read:	Read data from file, usually starting from current position
Write:	Write data to file, usually starting from current position
Repositioning:	Also known as seek Move the current position to a new location No actual Read/Write is performed
Truncate:	Removes data between specified position to end of file

File Operations as System Calls

- OS provides file operations as **system calls**:
  - Provide protection, concurrent and efficient access
  - Maintain information

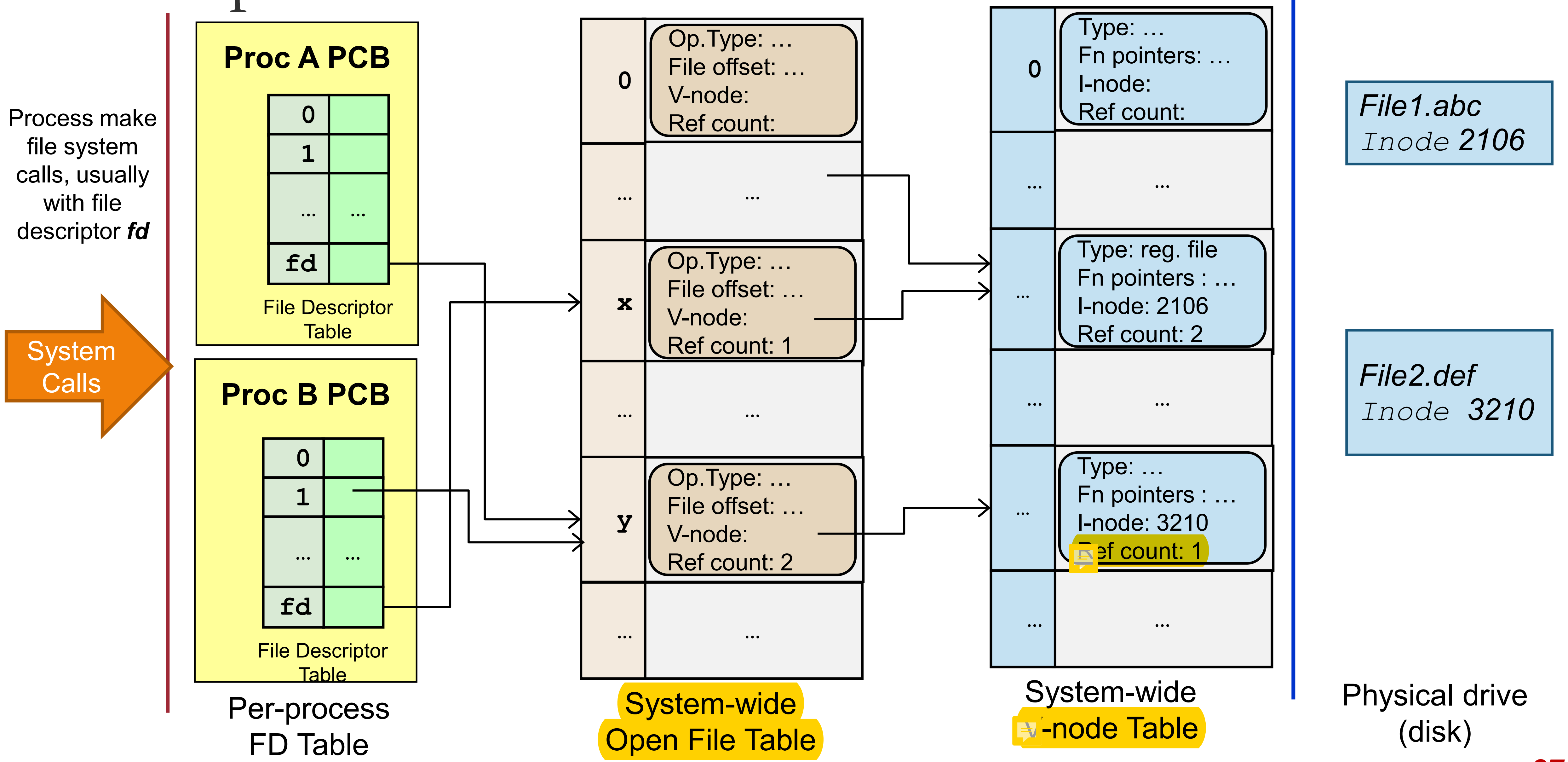
- Information kept for an opened file:
  - **File Pointer:** Current location in file
  - **Disk Location:** Actual file location on disk
  - **Open Count:** How many times has this file opened?
    - Useful to determine when to remove the entry in table

File Information in the OS

- **Consider:**
  - Several processes can open the same file
  - Several different files can be opened at any time
- **Common approach – uses 3 tables:**
  - Per-process open-file table:
    - To keep track of the open files for a process
    - Each entry points to the **system-wide open-file table** entries
  - System-wide open-file table:
    - To keep track of all the open files in the system
    - Each entry points to a **V-node** entry
  - System-wide V-node(virtual node) table
    - To link with the file on physical drive
    - Contains the information about the physical location of the file.

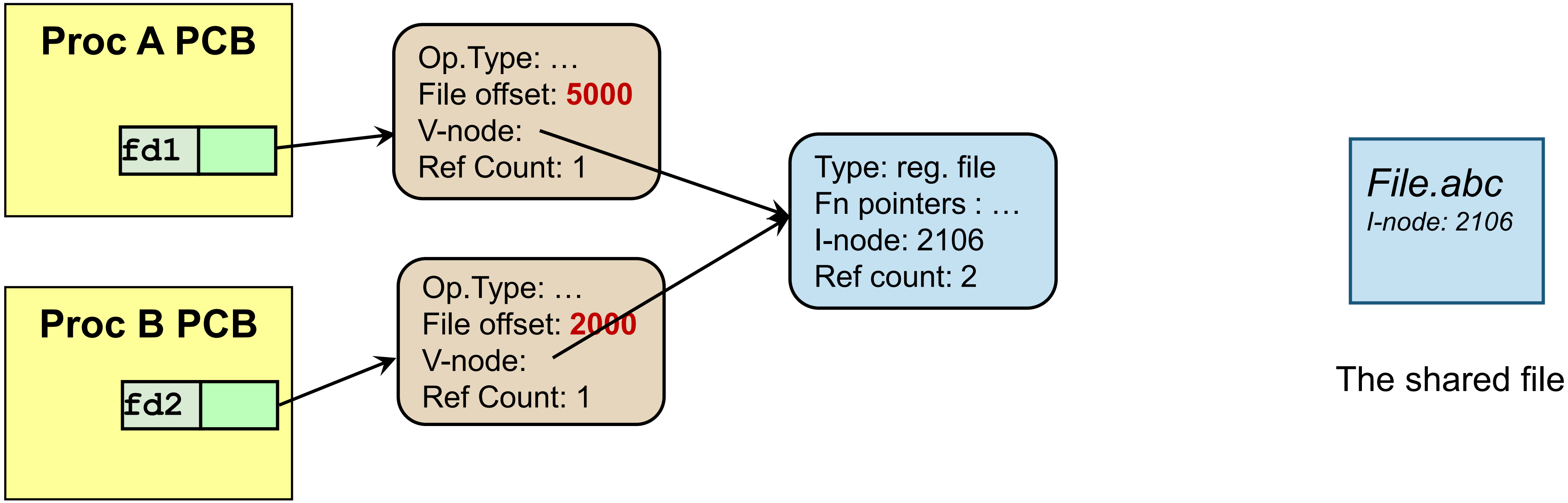
everything here is in kernel space

File Operations: Unix Illustration



Process Sharing File in Unix: Case 1

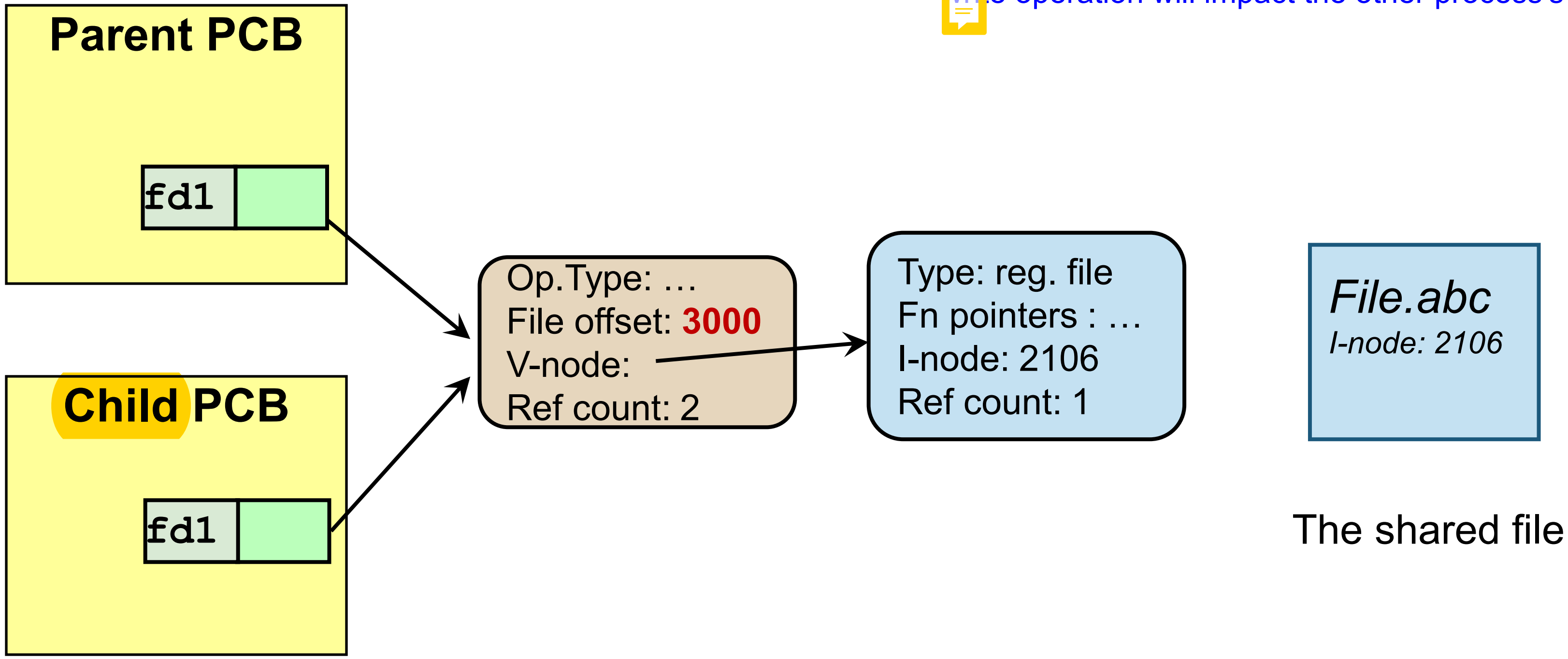
- A file is opened twice from two processes:
  - 2 file descriptors
  - 2 entries in the system-wide open file table
  - I/O can occur at independent offsets
- When:
  - Two process open the same file
  - Same process open the file twice



Process Sharing File in Unix: Case 2

- Two file descriptors pointing to the same entry in the system-wide open file table
  - Only one offset ➔ I/O changes the offset for the other process
- When:
  - fork() after file is opened
  - dup () within the same process

write operation will impact the other process's read



The shared file

Just your regular folders

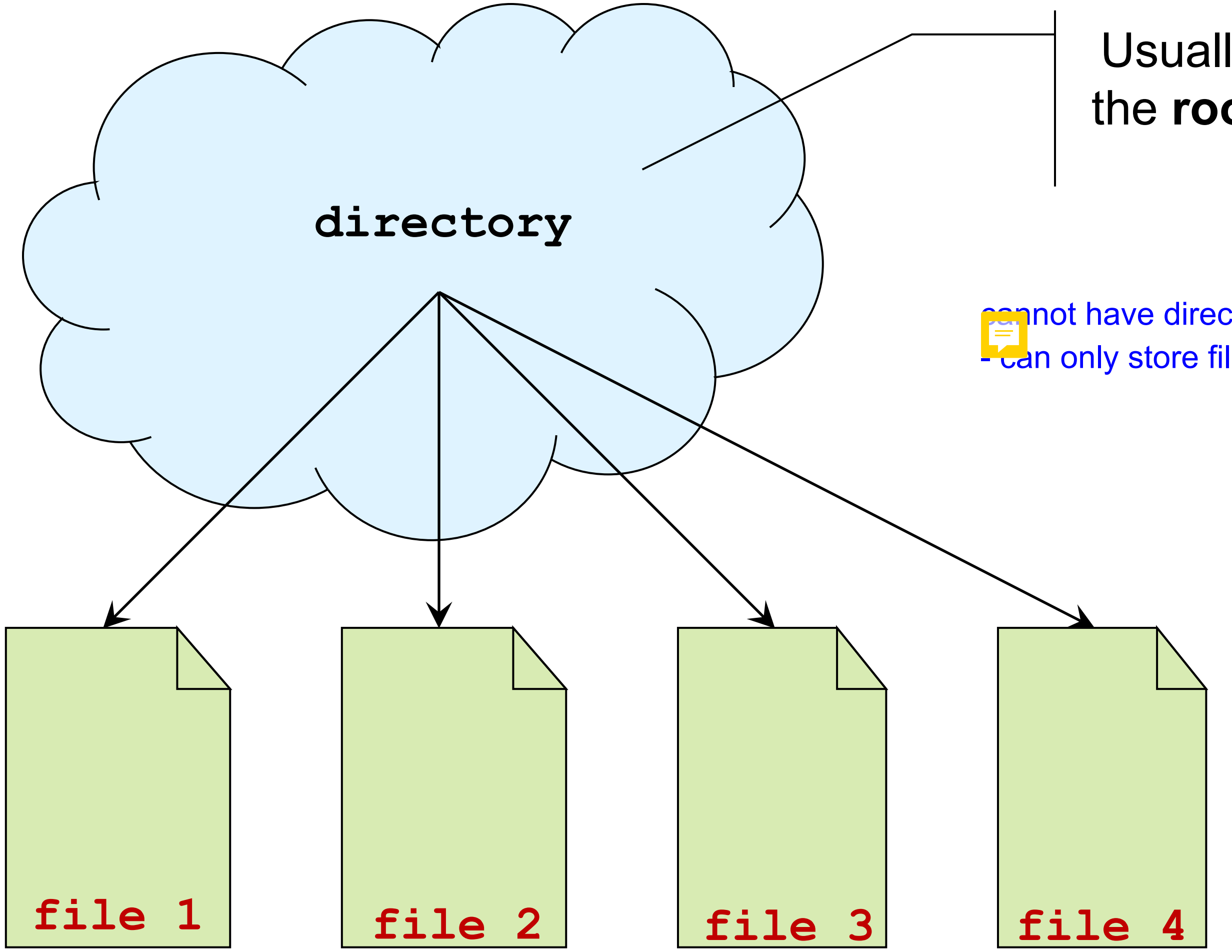
DIRECTORY

Directory: Basics

- **Directory** (**folder**) is used to:
  1. Provide a logical grouping of files
     - The user view of directory
  2. Keep track of files
     - The actual system usage of directory

- Several ways to structure directory:
  - Single-Level
  - Tree-Structure
  - Directed Acyclic Graph (DAG)
  - General Graph

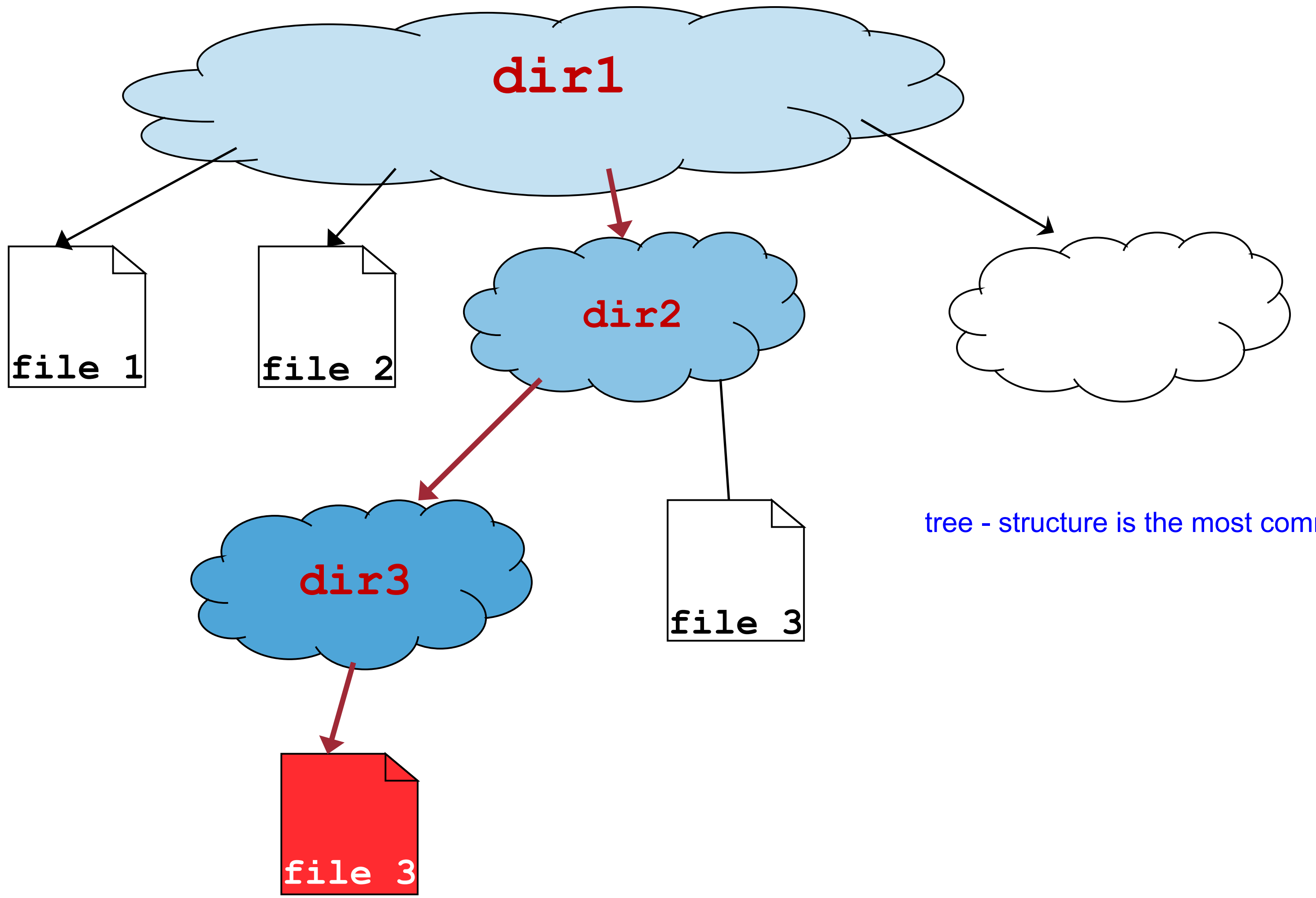
Directory Structure: **Single-Level**



Usually known as the **root directory**

cannot have directory within the root directory
- can only store files

Directory Structure: **Tree-Structured**

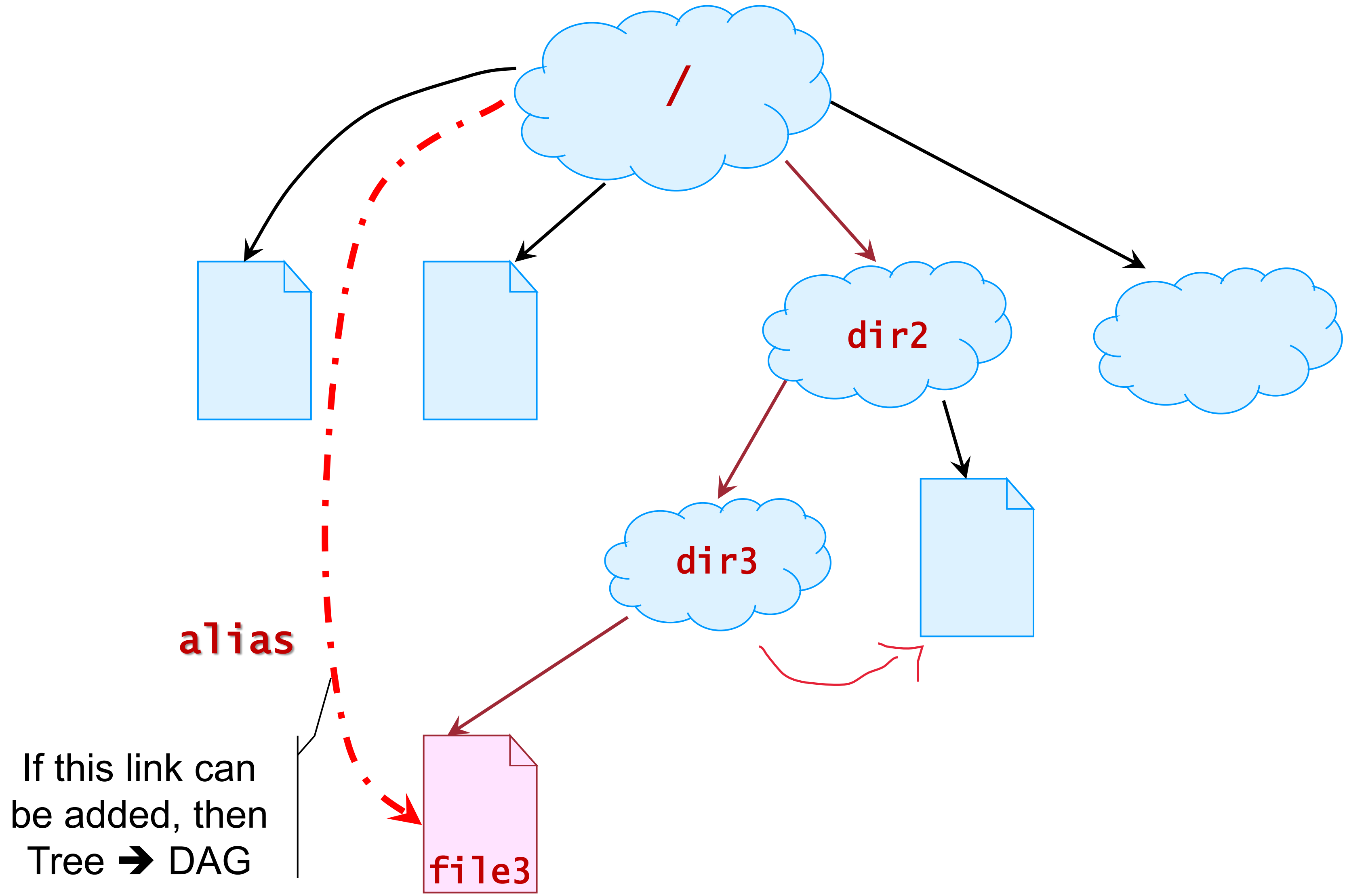


tree - structure is the most common

Directory Structure: Tree-Structured

- **General Idea:**
  - Directories can be recursively embedded in other directories
  - Naturally forms a tree structure
- Two ways to refers to a file:
  - **Absolute Pathname:**
    - Directory names followed from root of tree + final file
    - i.e. the Path from root directory to the file
  - **Relative Pathname:**
    - Directory names followed from the **current working directory (CWD)**
    - CWD can be set explicitly or implicitly changed by moving into a new directory under shell prompt

Directory Structure: **DAG**



Directory Structure: DAG

- If a **file** ***can be shared:***
  - Only one copy of actual content
  - "Appears" in multiple directories
    - With different path names
- Then tree structure ➔ DAG

- Two implementations in Unix:
  - **Hard Link**
    - Not allowed for directories
  - **Symbolic Link**
    - This has an "interesting" effect….

DAG: Unix Hard Link

- **Consider:**
  - Directory **A** is the owner of file **F**
  - Directory **B** wants to share **F**
- **Hard Link:**

modifying the entries to create a link to the inode for the actual file

  - **A** and **B** has **separate pointers** point to the actual file F in disk
  - **Pros:**
    - Low overhead, only pointers are added in directory
  - **Cons:**
    - Deletion problems:
      - e.g. If B deletes F? If A deletes F?
      - Ref. count is needed
  - Unix Command: "`ln`"

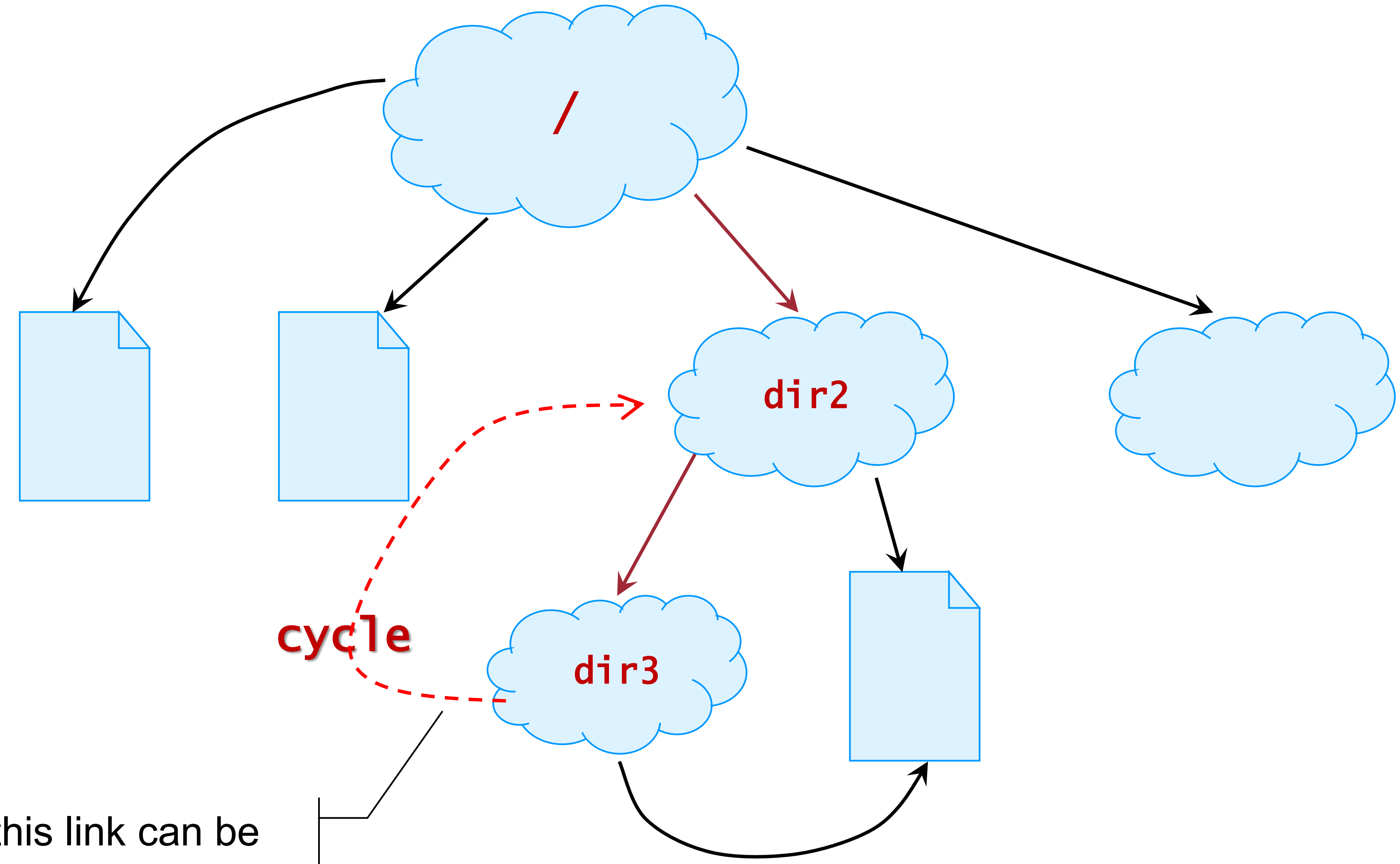
DAG: Unix Symbolic Link

- **Symbolic Link:**

symbolic link will create a new file with the path to the actual file that we want

  - The symbolic link is a ***special link file***, **G**
    - **G** contains the path name of **F**
  - When **G** is accessed:
    - Find out where is **F**, then access **F**
  - **Pros:**
    - Simple deletion:
      - If the symbolic link is deleted: **G** deleted, not **F**
      - If the linked file is deleted: **F** is gone, **G** remains (but not working)
  - **Cons:**
    - Larger overhead:
      - Special link file take up actual disk space
  - Unix Command: "`ln -s`"

Directory Structure: **General Graph**



If this link can be added, then Tree
➔ General Graph

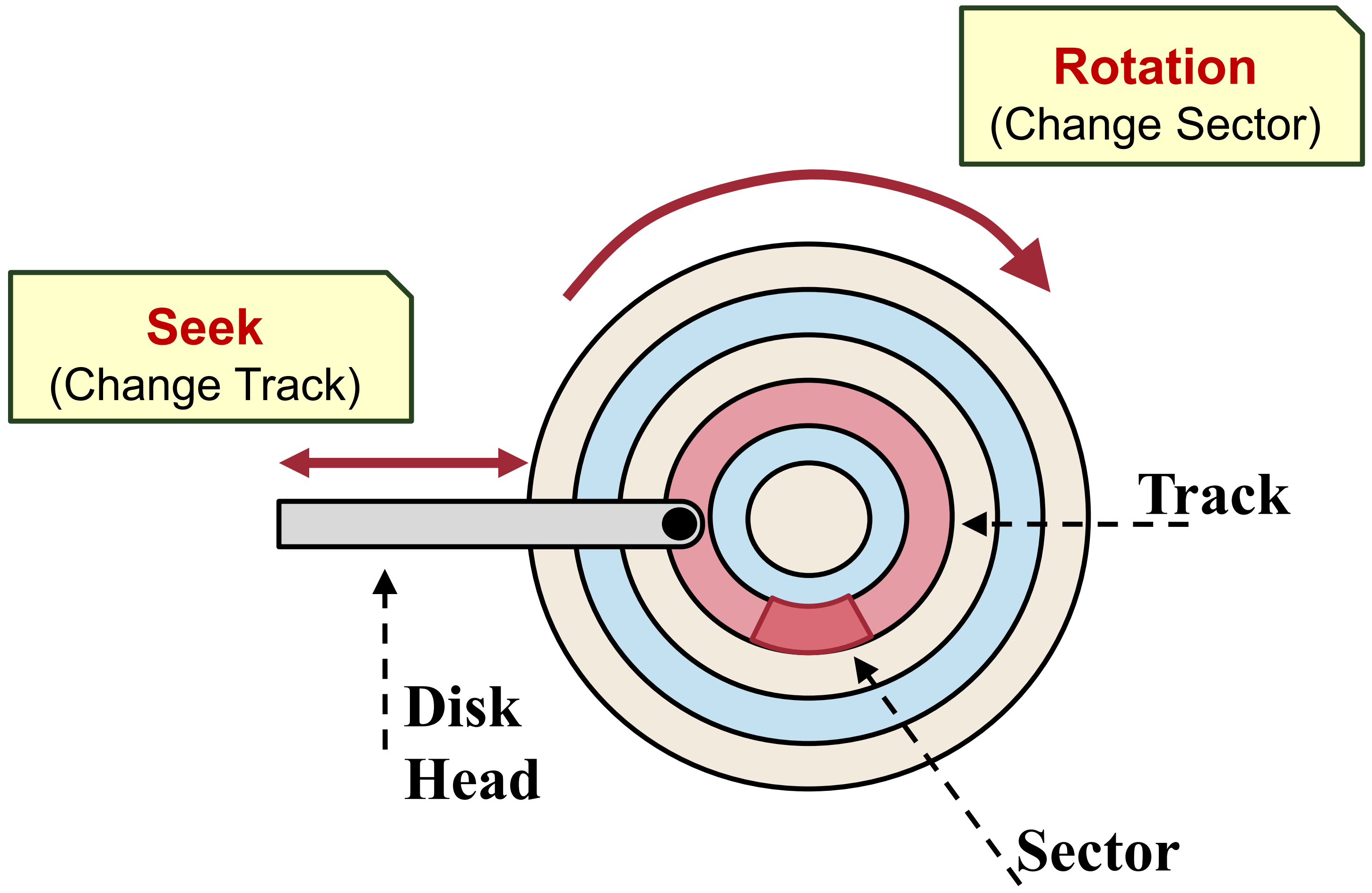
Directory Structure: General Graph

- General Graph Directory Structure is ***not desirable:***
  - **Hard to traverse**
    - Need to prevent infinite looping
  - **Hard to determine when to remove a file/directory**
- **In Unix:**
  - Symbolic link is allowed to link to directory
    - General Graph **can be created**

I'm afraid you have to wait…..

I/O SCHEDULING

Magnetic Disk in One Glance



Disk Scheduling: The Problem

- Due to the significant seek and rotational latency, OS should schedule the disk I/O requests
- I/O (disk) scheduling:
  - Intention of reducing **overall waiting time**
  - As rotational latency is hard to mitigate, we focus on reducing the **seeking time**
  - Balance the need for high throughput while trying to fairly share I/O requests amongst processes

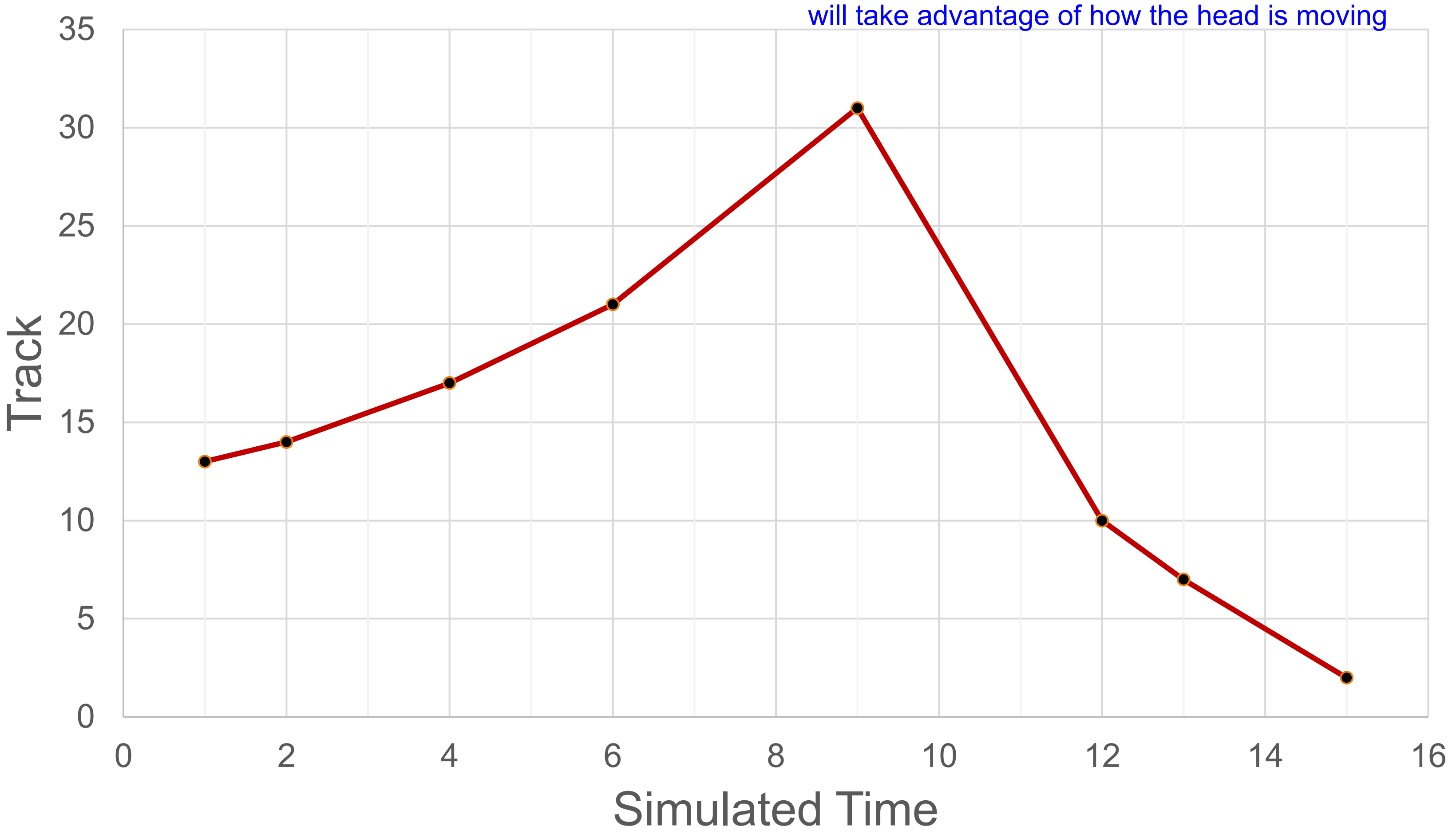
Disk Scheduling: Algorithms

- Consider the following disk I/O requests indicated by only the **track number (magnetic disks)**:
  - **13, 14, 2, 10, 17, 31, 21, 7**
- A few obvious candidates:
  - **FCFS**
  - **SSF** (**S**hortest **S**eek **F**irst)
    - "SJF" modified for the disk context
  - The **SCAN** family (aka **Elevator**):
    - Bi-Direction [Innermost ←→ Outermost] (SCAN)
    - 1-Direction [Outermost→ Innermost]   (C-SCAN)

    - Very intuitive: Imagine the tracks are floors in a building, and the disk head is the elevator servicing the floors (Figure out the algorithm before lecture ☺)

SCAN: Disk Head Movement

- disk I/O requests indicated by only the **track number** :
  [13, 14, 2, 10, 17, 31, 21, 7]



I/O Scheduling: Newer Algorithms

- **Deadline** - 3 queues for I/O requests:
  - Sorted
    can improve by increasing the priority for reading so that will read first since it is faster than write
  - Read FIFO - read requests stored chronologically
  - Write FIFO - write requests stored chronologically
- **noop (No-operation)** - no sorting
- **cfq (Completely Fair Queueing)** - time slice and per-process sorted queues
- **bfq (Budget Fair Queuing) (Multiqueue) -** fair sharing based on the number of sectors requested

process which requires alot more sectors will have a lower priority

Summary

- Covered basics of file system from a user point of view
- Understand the basic requirements of a FS
- Understand the components of a FS:
  - File and Directory
- Discussed OS responsibility in I/O scheduling

For your reference only

UNIX FILE OPERATIONS

File Operations Example: Unix System Calls

- Header Files:
  - `#include <sys/types.h>`
  - `#include <sys/stat.h>`
  - `#include <fcntl.h>`

- File related Unix System Calls
  - `open()`, `read()`, `write()`, `lseek()`, `close()`
- General Information:
  - Opened file has an identifier
    - **File Descriptor:** Integer
    - Used for other operations
  - File is access on a byte-by-byte basis
    - No interpretation of data

Opening Files: `open( )`

- Function Call:

```
int open( char *path, int flags )
```

- Return:
  - `-1`: Failed to open file
  - `>=0`: **file descriptor**, a unique index for opened file
- Parameters:
  - `path`: File path
  - `flags`: Many options can be set using bit-wise-OR
    - Read, Write or Read+Write mode
    - Truncation, Append mode
    - Create file if no exists
    - … Many many more ☺

Opening Files: **`open()`** (cont)

- Example:

```
int fd;    //file descriptor

//Open an existing file for read only
fd = open( "data.txt", O_RDONLY );

//Create the file if not found, open for read + write
fd = open("data.txt", O_RDWR | O_CREAT );
```

- By convention:
  - Default file descriptors:
    - **STDIN** (0), **STDOUT** (1), **STDERR** (2)

Read Operation: `read()`

- Function Call:

  `int read(int fd, void *buf, int n)`

- Purpose:
  - reads up to **n** bytes from current offset into buffer **buf**
- Return:
  - number of bytes read, can be `0...n`
  - `<n` : end of file is reached
- Parameters:
  - **fd**: file descriptor (must be opened for read)
  - **buf**: An array large enough to store **n** bytes
- `read()` is ***sequential read***:
  - starts at current offset and increments offset by bytes read

Write Operation: `write()`

- Function Call:

  `int write(int fd, void *buf, int n)`
- Purpose:
  - `writes` up to **n** bytes from current offset from buffer **buf**
- Return:
  - **-1:** Error
  - **>= 0:** Number of bytes written
- Parameters:
  - **`fd`**: file descriptor (must be opened for write)
  - **`buf`**: An array of at least **n** bytes with values to be written
- `Possible errors:`
  - exceeds file size limit, quota, disk space, etc.
- write() is ***sequential write:***
  - starts at current offset and increments offset by bytes written
  - can increase file size beyond EOF ➔ append new data

Repositioning: `lseek()`

- Function Call:

  ```
  off_t lseek(int fd, off_t offset, int whence)
  ```
- Purpose:
  - Move current position in file by **offset**
- Return:
  - **-1:** Error
  - **>= 0:** Current offset in file
- Parameters:
  - **`fd`**: file descriptor (must be opened)
  - **offset**: positive = move forward, negative = move backward
  - **whence:** Point of reference for interpreting the **offset**
    - **SEEK_SET:** absolute **offset** (count from the file start)
    - **SEEK_CUR:** relative **offset** from current position (+/-)
    - **SEEK_END:** relative **offset** from end of file (+/-)
- Can seek anywhere in file, even beyond end of existing data

Closing Files: `close()`

- Function Call:

```
int close( int fd )
```

- Return:
  - **-1:** Error
  - **0:** Successful
- Parameters:
  - **`fd:`** file descriptor (must be opened)
- With close():
  - **`fd`** no longer used anymore
  - Kernel can remove associated data structures
  - The identifier **`fd`** can be reused later
- By default:
  - Process termination automatically closes all open files