



# **File System Interface**

**ICS332  
Operating Systems**

Henri Casanova ([henric@hawaii.edu](mailto:henric@hawaii.edu))

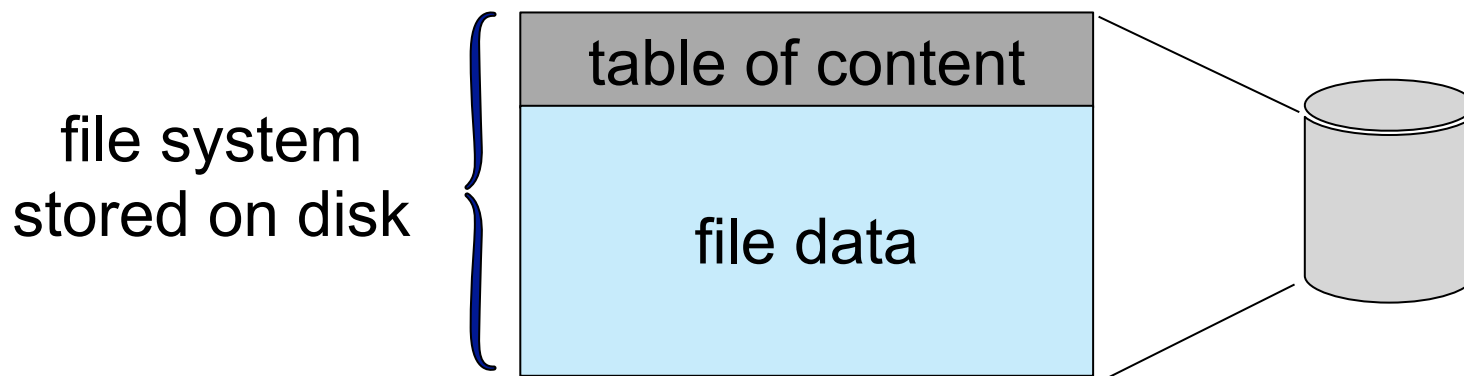


# File System

- The term “File System” is a bit confusing
- It can mean the **component of the OS** that knows how to virtualize storage
  - As in “I am using a very fast file system implementation”
  - We’ll talk about the OS component in the next set of lecture notes

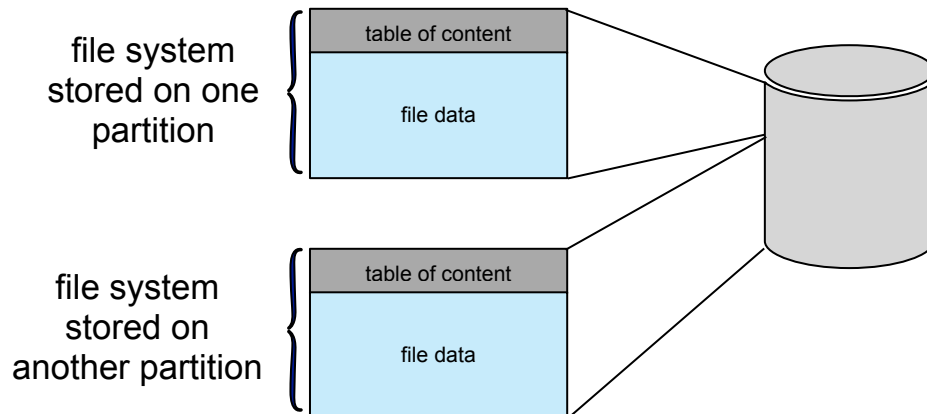
# File System

- The term “File System” is a bit confusing
- It can mean the **component of the OS** that knows how to virtualize storage
  - As in “I am using a very fast file system implementation”
  - We’ll talk about the OS component in the next set of lecture notes
- It can mean the **data stored on a disk partition** or “volume” (i.e., a disk partition that contains a file system)
  - As in “my file system was erased”



# File System

- The term “File System” is a bit confusing
- It can mean the **component of the OS** that knows how to virtualize storage
  - As in “I am using a very fast file system implementation”
  - We’ll talk about the OS component in the next set of lecture notes
- It can mean the **data stored on a disk partition** or “volume” (i.e., a disk partition that contains a file system)
  - As in “my file system was erased”



- There can be multiple file systems stored on the same physical disk, in different partitions

# Files

- The key abstraction to virtualize storage is a **file**
  - An array of bytes where each byte can be read or written
  - Has a low-level name (e.g., a number) and a user-level name (e.g., “foo.txt”)
  - Has associated metadata (owner, last write time, read/write/execute permissions)
  - Has some internal type (regular file, symbolic link, hard link, device)
- What about user-level file types?
  - A common practice is to use “file extensions” (like .txt)
  - In some OSes, it is used to figure out what application can open the files
  - But file extensions are typically just conventions
    - In Linux you can name an executable with a .txt extension if you want
    - In Windows you can, but you need to tell Windows about it (PATHEXT)
  - But in general, the type of a file is based on its content, often on the first few bytes (called the “magic number”)
    - The UNIX `file` command uses this to display useful file “types”
    - Let’s try it on my `/bin/local` directory...

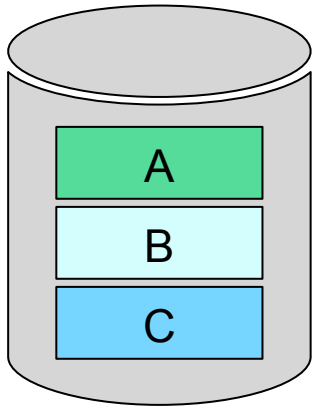
# File Operations

- A file is an abstraction, i.e., an abstract data type, on which the OS defines several operations
  - Creating
  - Writing/Reading
    - A current-file-position pointer is kept per process, updated after each write/read operation
  - Repositioning the current-file-position pointer (a “seek”)
  - Appending at the end of a file
  - Truncating / Deleting
  - Renaming
  - Getting / changing metadata
- OSTEP 39 has **many** examples of using the POSIX file system interface for creating/reading/writing/deleting files
  - I am **not** going to go through all of those here
  - Most of you should be familiar with some of this anyway
  - In these lecture notes we go through some of the ideas/concepts without necessarily showing the corresponding code (see OSTEP for all details)

# Opening Files

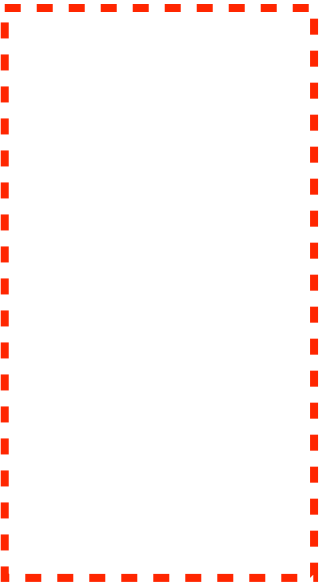
- The OS requires that processes open and close files
  - As you know from writing code in basically any language
- After an open, the OS copies the file system's file entry into an open-file table that is kept in RAM in the kernel
- The OS keeps an **open file tables**
- A process specifies an open file as an index in its own list of open files
  - The famous “filed” (file descriptor) in Linux
- Let's see an example ...

# Open File Table



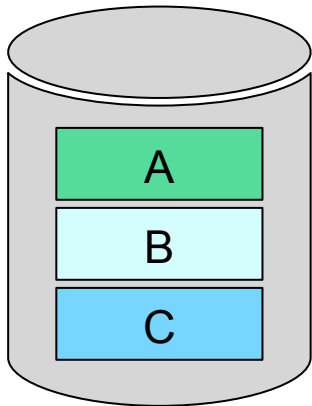
- Consider a system with 3 files on disk

Kernel





# Open File Table

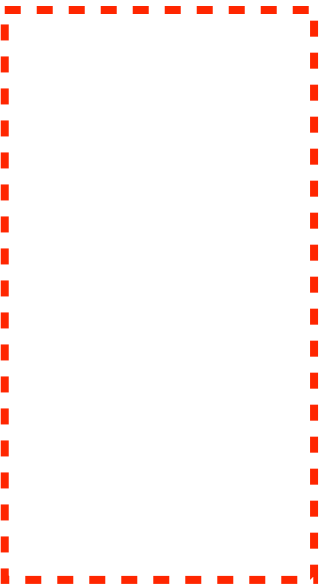


Process #1

Open File A

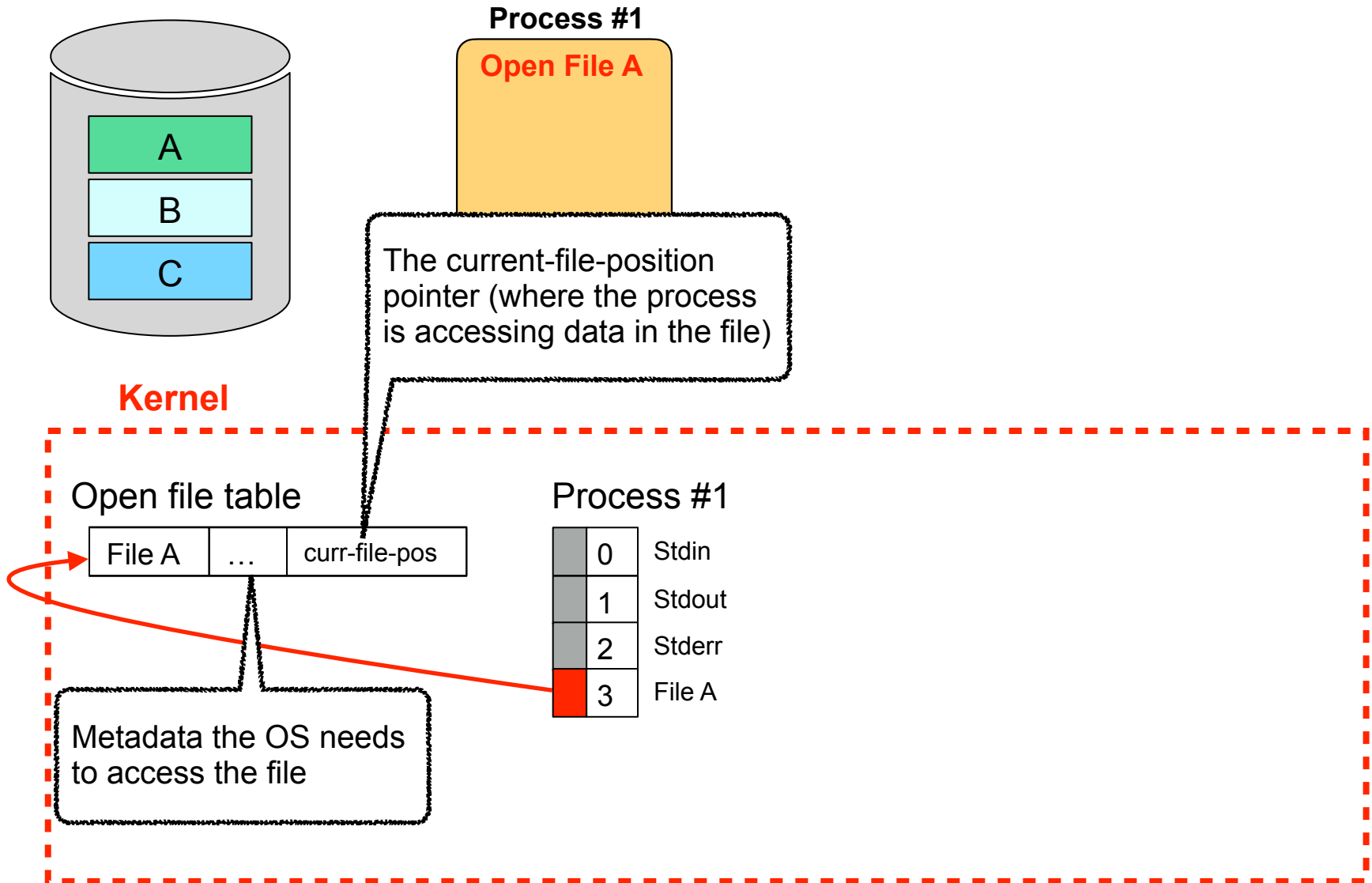


Kernel

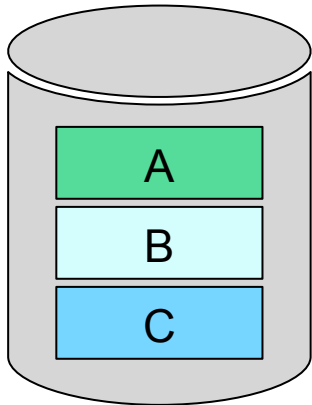


- To open a file a process gives its path to the OS
- The OS asks the file system to use its on-disk “table of content” to locate a data structure that describes the file
  - The block index of the first block, the total number of blocks, other useful metadata
- The OS then creates an entry in the **open file table**
- The OS returns a file description (an integer) to the process, which is an index in its list of opened files
- These stay around until the process closes the file

# Open File Table



# Open File Table



Process #1

Open File A  
Open File C

Kernel

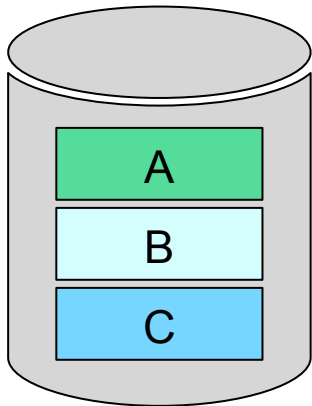
Open file table

File A	...	curr-file-pos
File C	...	curr-file-pos

Process #1

0	Stdin
1	Stdout
2	Stderr
3	File A
4	File C

# Open File Table



Process #1

Open File A  
Open File C  
Open File A

Kernel

Open file table

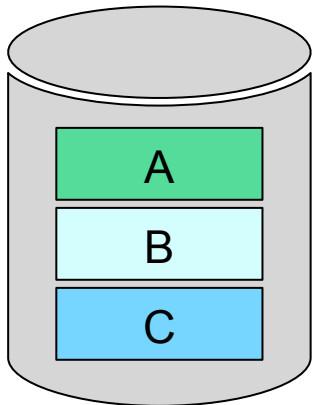
File A	...	curr-file-pos
File C	...	curr-file-pos
File A	...	curr-file-pos

Process #1

0	Stdin
1	Stdout
2	Stderr
3	File A
4	File C
5	File A

A single process can open a file multiple times. It now has more than one current-file-position offsets into the file.

# Open File Table



Process #1

Open File A  
Open File C  
Open File A

Process #2

Open File A

## Kernel

Open file table

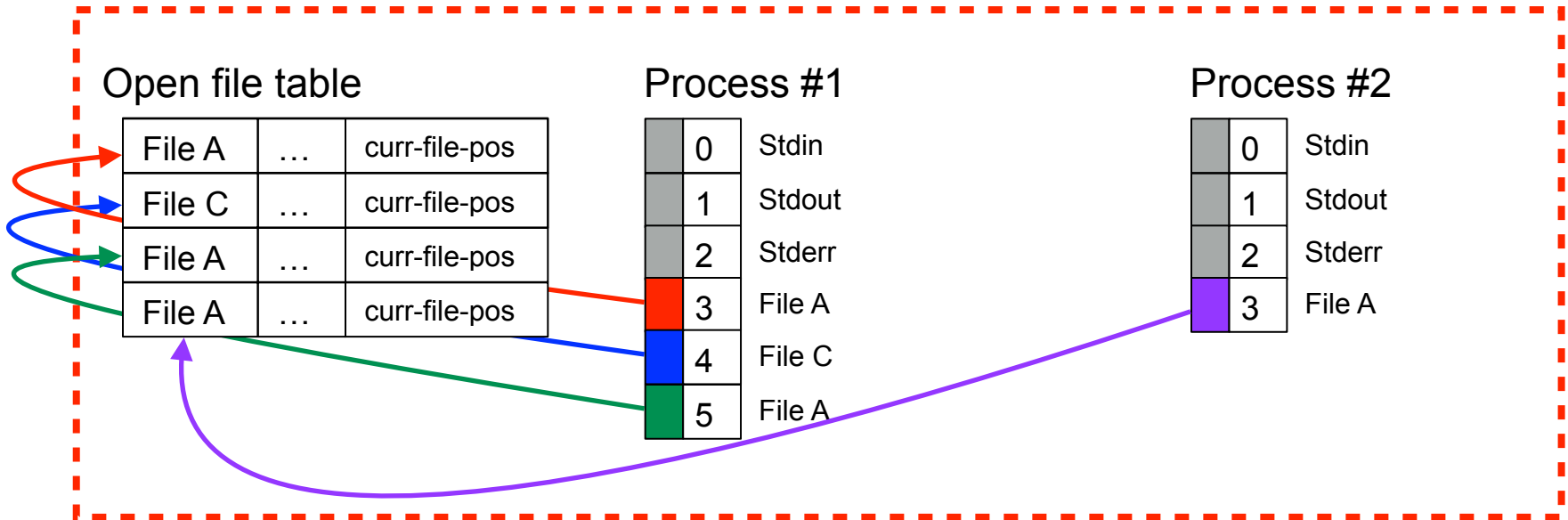
File A	...	curr-file-pos
File C	...	curr-file-pos
File A	...	curr-file-pos
File A	...	curr-file-pos

Process #1

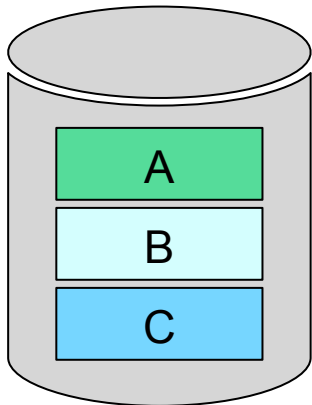
0	Stdin
1	Stdout
2	Stderr
3	File A
4	File C
5	File A

Process #2

0	Stdin
1	Stdout
2	Stderr
3	File A



# Open File Table



Process #1

Open File A  
Open File C  
Open File A  
**Close(3)**

Process #2

Open File A

**Kernel**

Open file table

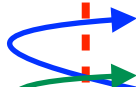
File C	...	curr-file-pos
File A	...	curr-file-pos
File A	...	curr-file-pos

Process #1

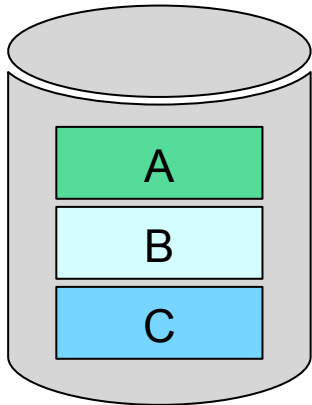
0	Stdin
1	Stdout
2	Stderr
3	
4	File C
5	File A

Process #2

0	Stdin
1	Stdout
2	Stderr
3	File A



# Open File Table



## Process #1

Open File A  
Open File C  
Open File A  
Close(3)  
**Close(1)**

## Process #2

Open File A

## Kernel

### Open file table

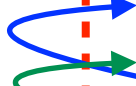
File C	...	curr-file-pos
File A	...	curr-file-pos
File A	...	curr-file-pos

### Process #1

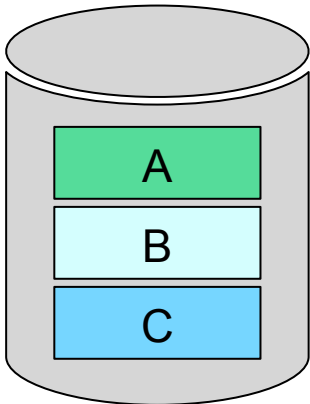
0	Stdin
1	Stdout
2	Stderr
3	
4	File C
5	File A

### Process #2

0	Stdin
1	Stdout
2	Stderr
3	File A



# Open File Table



## Process #1

Open File A  
Open File C  
Open File A  
Close(3)  
Close(1)  
**Open File B**

## Process #2

Open File A

## Kernel

### Open file table

File B	...	curr-file-pos
File C	...	curr-file-pos
File A	...	curr-file-pos
File A	...	curr-file-pos

### Process #1

0	Stdin
1	Stdout
2	Stderr
3	
4	File C
5	File A

Stdout is redirected to File B!

### Process #2

0	Stdin
1	Stdout
2	Stderr
3	File A



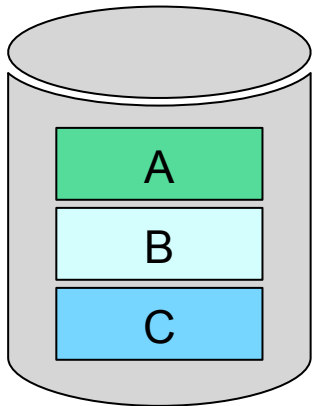
# File Locking

- Bad things may happen when multiple processes reference the same file and update its content
  - Just like when threads update the same memory location
- All file systems provides “file locks”
  - Can be acquired for a full file or for a portion of a file
- The OS may require mandatory locking for some files
  - e.g., for writing for a log file that many system calls write to
- You may have encountered that
  - Let's try to `sudo apt update` in two terminals at the “same” time
- Applications have to implement their own locking
  - From a Shell script on Linux, one can use `flock`
  - Java implements `lock()` / `release()` methods as part of the `java.nio.channels` package

# Shared Open File Table Entries

- Each time a process opens a file, an entry is created in the Kernel's open file table
- But there are ways in which a process can “duplicate” an existing file descriptor (see the Processes module)
  - By being a child of a process with that file descriptor
  - But using the `dup ( )` syscall
- In this case, multiple file descriptors can point to the same open file table entry
- OSTEP Figure 39.2 shows code in which a parent and child shared the same open file table entry
  - The child updates the current-file-position pointer, and the parent “sees” that update!

# Open File Table



Parent

Open File A  
Open File C  
Fork Child

Child



Kernel

Open file table

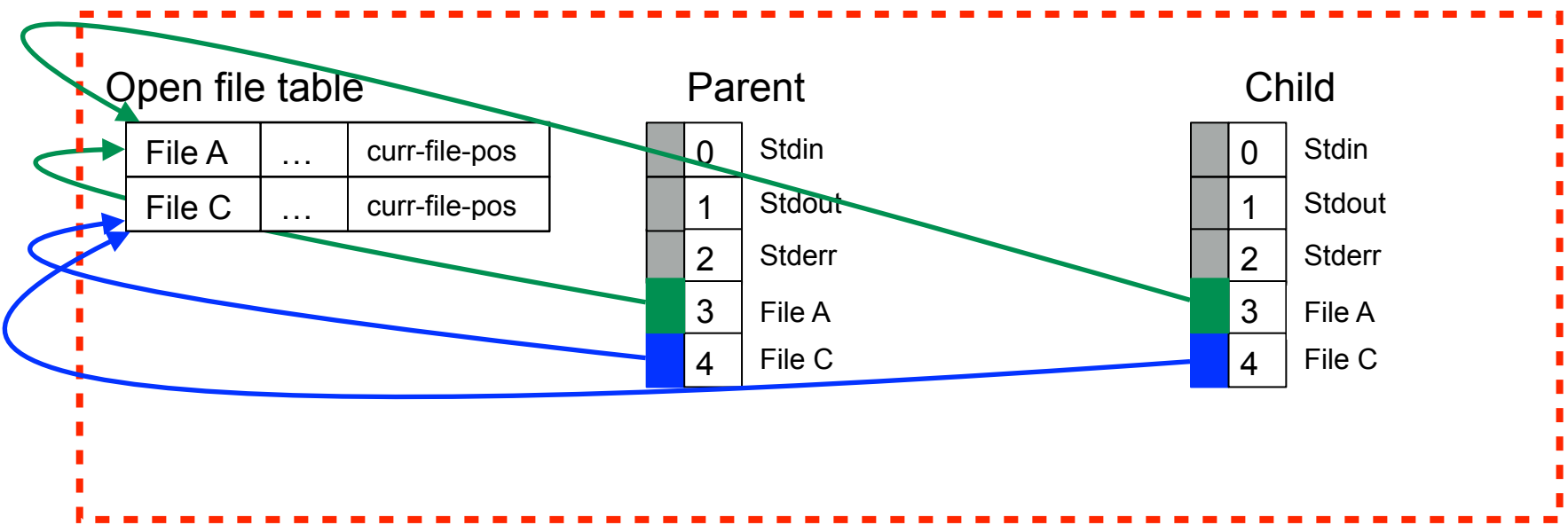
File A	...	curr-file-pos
File C	...	curr-file-pos

Parent

0	Stdin
1	Stdout
2	Stderr
3	File A
4	File C

Child

0	Stdin
1	Stdout
2	Stderr
3	File A
4	File C



# Directories

- The other main abstraction provided by the file system is the **directory (or folder)**:
  - A special kind of file: its data is not user data but just as list of files/directories it “contains”
- This makes it possible to create a hierarchy
- In Figure 39.1, the **absolute path** to the rightmost **bar.txt** file is **/bar/foo/bar.txt**
  - In UNIX the separator is ‘/’ and the top-level directory is called ‘/’
  - Paths look different in Windows (c:  
**\bar\foo\**)
- “.” and “..” are used to mean “this directory” or “the parent directory”
- Often one talks of the **path relative to a working directory**
  - In Figure 31.9 **../bar/foo/bar.txt** is a relative path to the working directory **/foo**

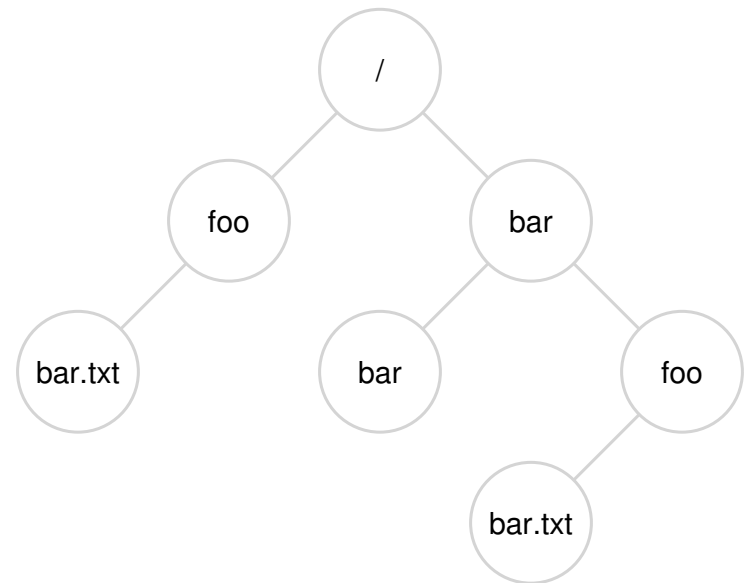


Figure 39.1: An Example Directory Tree

# Hard Links

- Remember that we said that a file has an internal name and a user-level name
- The internal “name” is really a pointer to some data structure that describes everything about the file
  - That data structure is stored on disk as part of the file system (the famous inode in UNIX systems, see next set of lecture notes)
- But there can be multiple user-level names for the same file!
- A user-level name for a file is called a “link”
- There is a `link()` syscall to associate a new name to a file
  - It takes as input an already existing name and the new name
- The UNIX `ln` command uses this system call to create new links
  - `ln <existing name> <new name>`
- Let's try it, and look the internal file name using `ls -li...`

# Link Count

- The links we created in the previous slides are called **hard links**
  - There was no difference between the two files
  - You can remove one without any problem
  - If you remove both, the file disappears
- There is a reference count in the internal file data structure (called the **link count**), and when it reaches zero the file and its content are deleted
  - We can see the link count using the **stat** command
  - Let's try that...

# Hard Link

- Hard links can be used for all kinds of purposes, for instance:
- On Linux, whenever a process opens a file, a hard link to it is created
- Say that process with PID 2233 uses `open()` on `/home/henric/somefile`
- `open()` returns a file descriptor, i.e., an integer, say 55
- At that point, a hard link to `/home/henric/somefile` is created in `/proc/2233/fd/55`
- If, while the process is running, `/bin/rm /home/henric/somefile` is executed, then the file survives because its reference count is non-zero!!
  - Essentially, you can't remove the data for a file while a process is using it, which is probably a good thing
- This allows you to retrieve a file that you've erased by mistake as long as some process has it opened
  - You might want to create hard links to your important files anyway
- Let's try this on a Linux box...

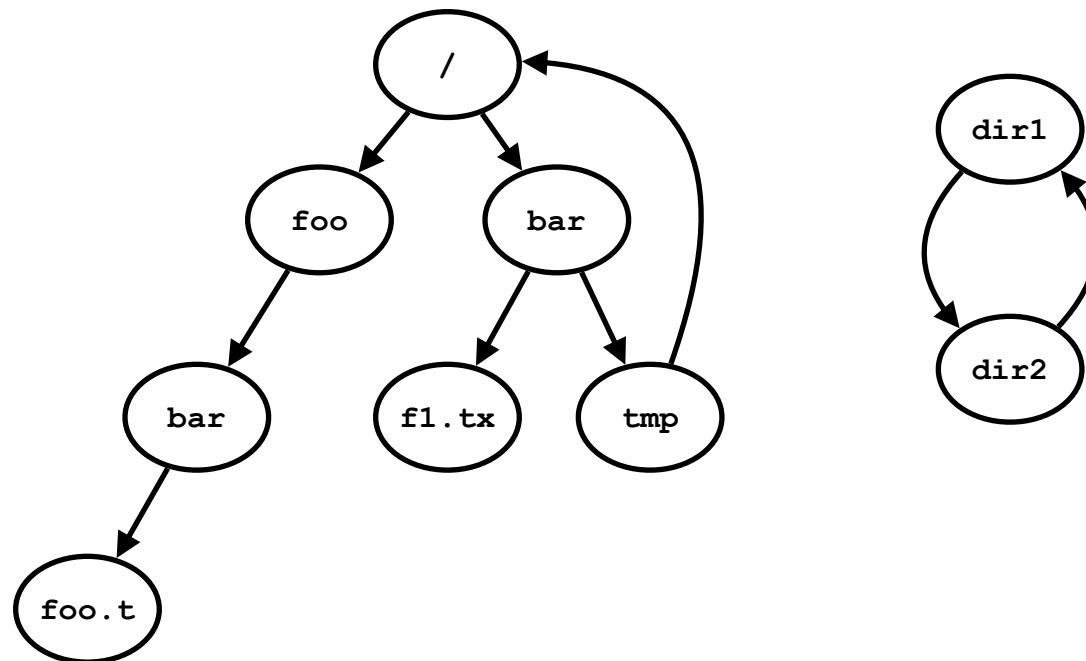
# Soft (or Symbolic) Links

- **Soft links** are just “shortcuts” that points to a path
- If you remove a soft link, nothing happens to the file
  - i.e., soft links don't contribute to the link count
- Hard links have limitations, as they only make sense within the same file system, but soft links can point anywhere, to any partition, because they're just paths
- A soft links is really a special kind of file
  - Just like Desktop shortcuts in Windows for instance
  - If the file pointed to has a longer name, the soft link size will be bigger!
- Soft links are created using `ln -s`
  - Let's try it (and use `ls -l`) ...
- A soft link can be **dangling** if its target is removed!
  - That cannot happen with hard link
  - Let's test that...



# Hard-linking Directories?

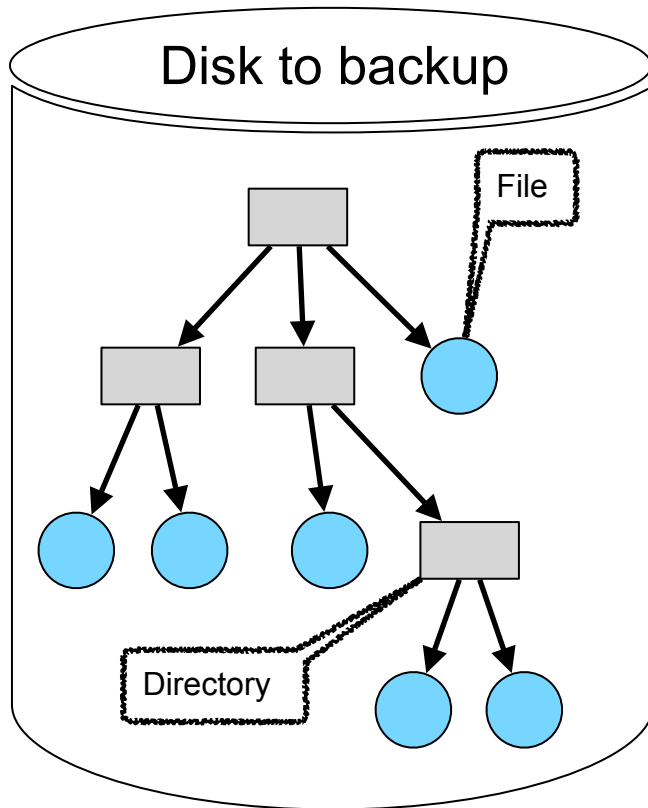
- In Linux, it is not possible to create a hard link to a directory
- This is to avoid cycles in the directory hierarchy!
  - Makes algorithms for traversing the hierarchy more complicated
  - Requires garbage collection of disconnected subgraphs



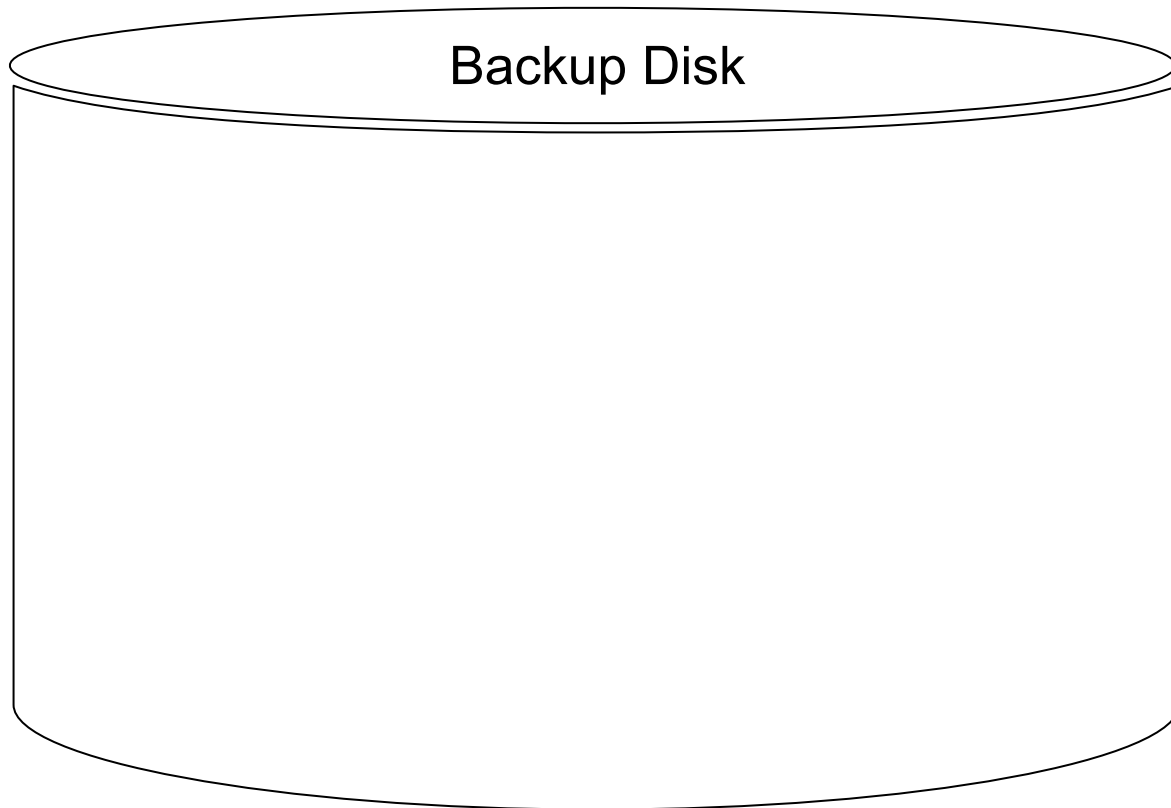
# MacOS Time Machine

- MacOS allows hard-linking of directories
  - Which makes implementing the file system more complicated
- Time Machine is the backup mechanism for MacOS
- It uses **hard links**
  - Every time a backup is made, a new backup directory is created that contains a snapshot of the current state of the file system
  - Files that have not been modified are hard links to previously backed up version
    - A new backup should be mostly hard links instead of file copies (major space saving)
  - When an old backup directory is wiped out, then whatever files have a reference count of zero are removed (no longer part of more recent data)
- Let's see an example....

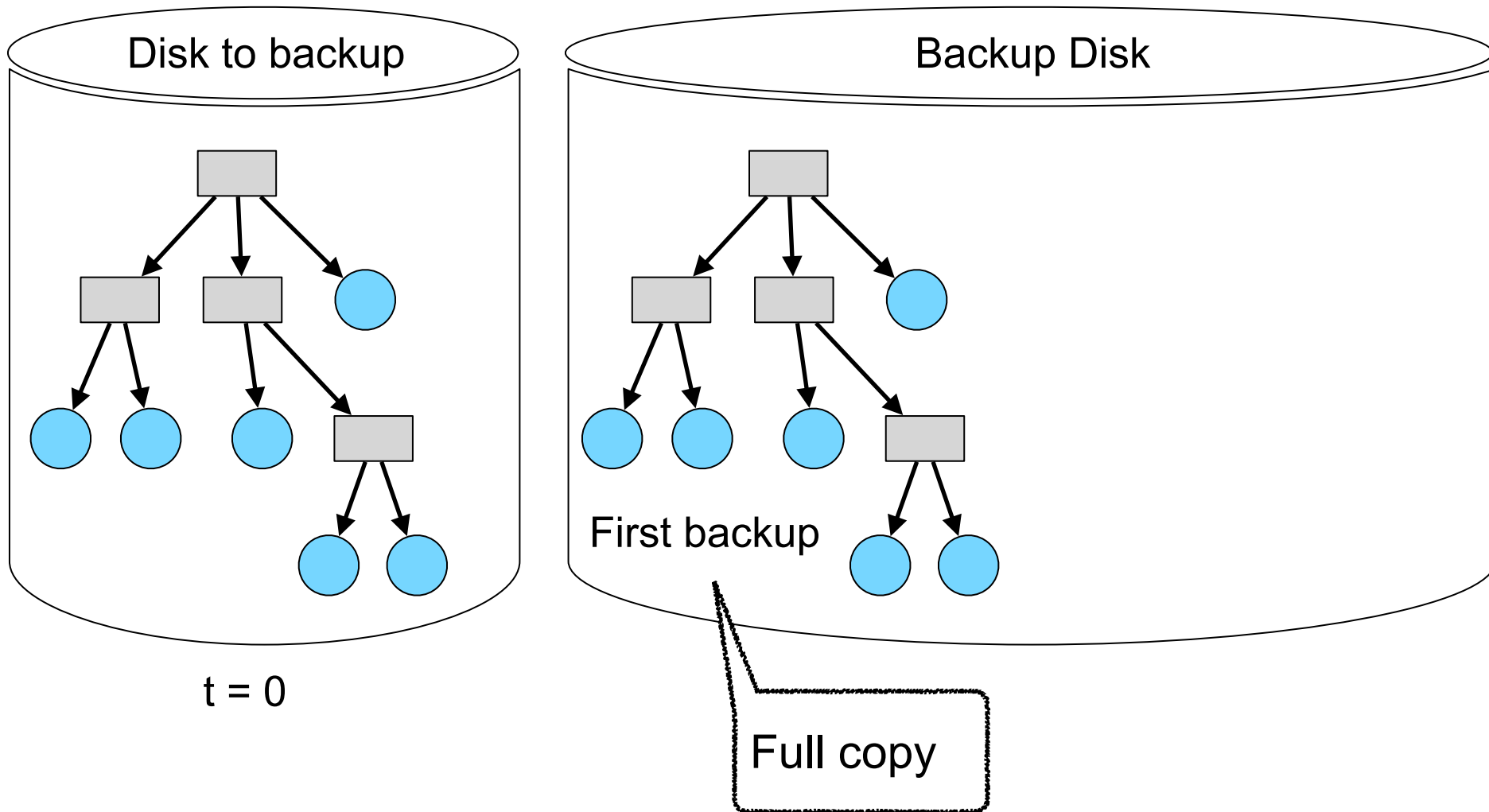
# MacOS Time Machine



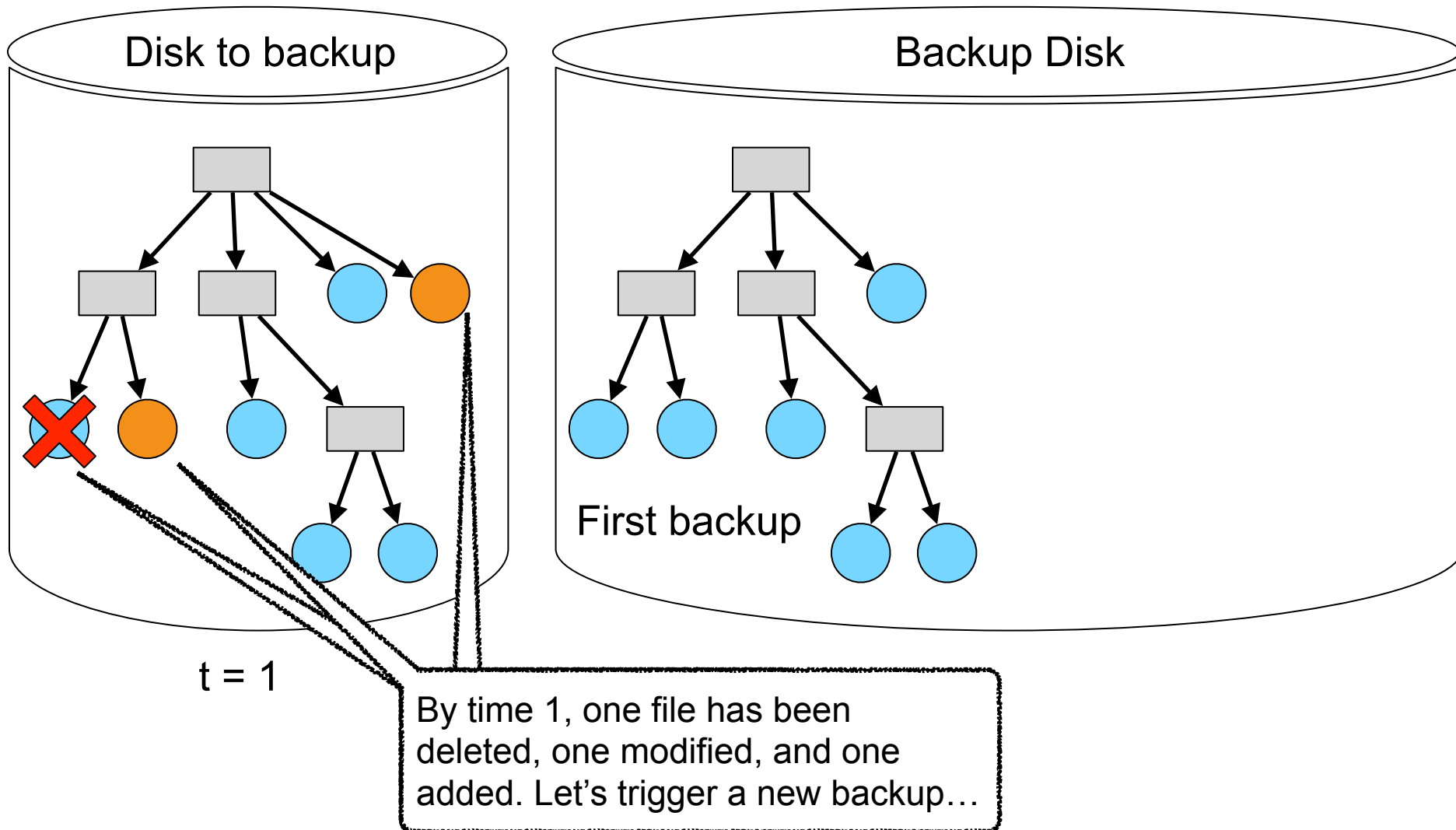
$t = 0$



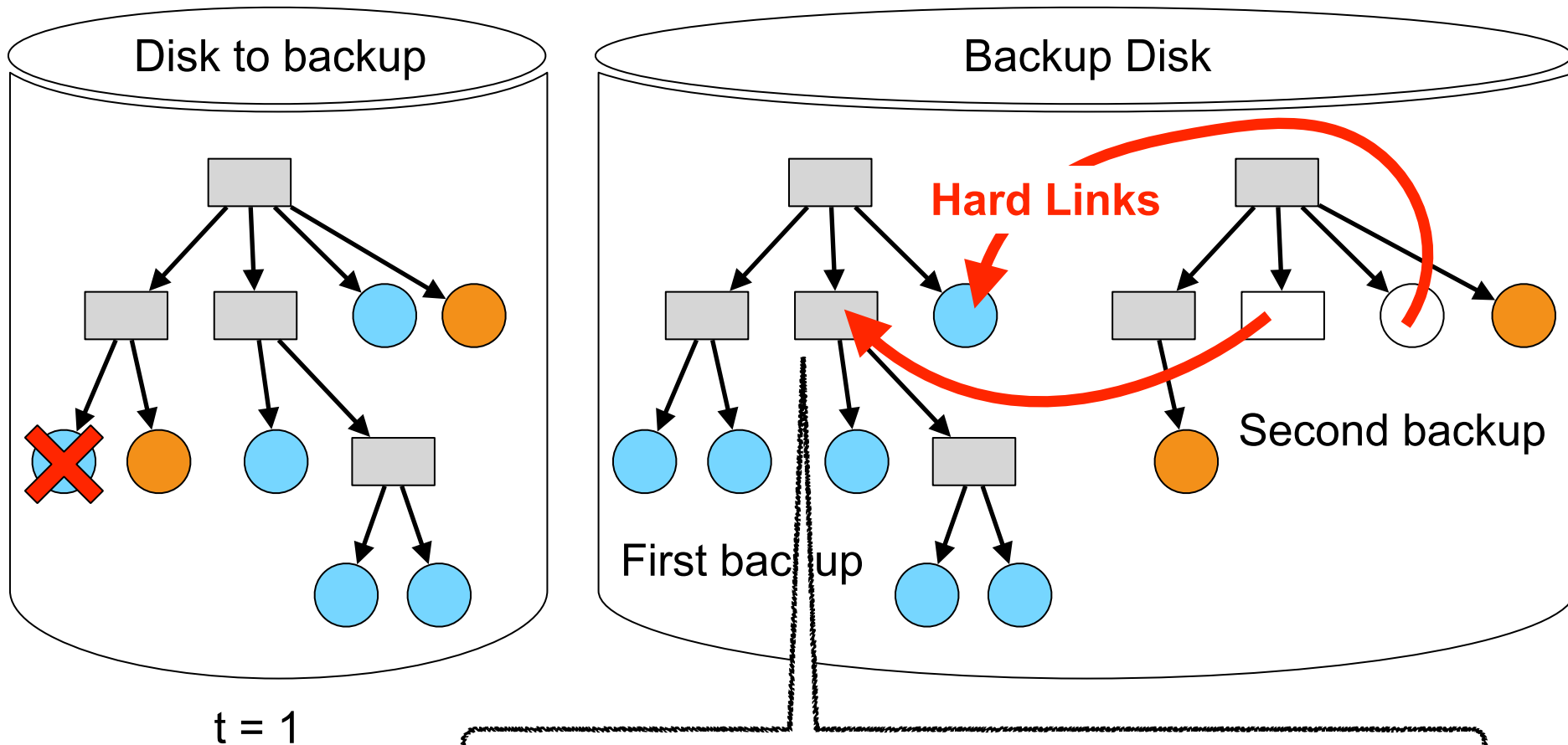
# MacOS Time Machine



# MacOS Time Machine

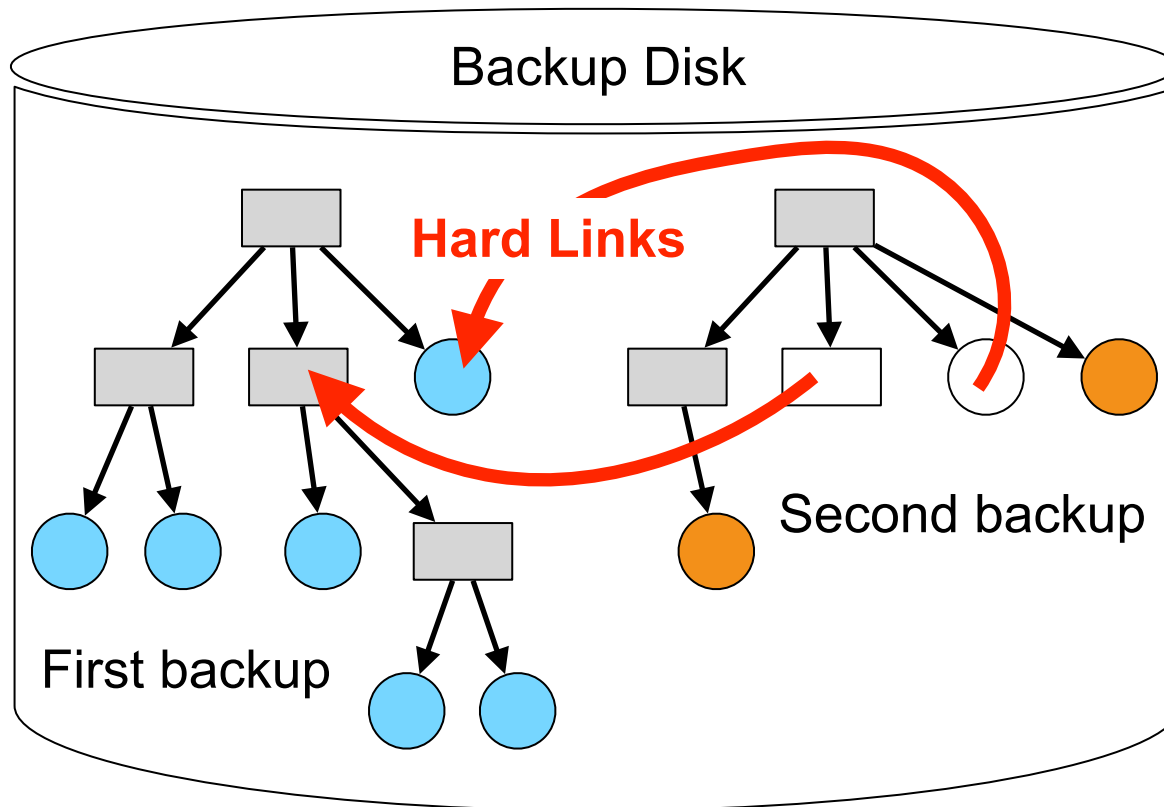


# MacOS Time Machine



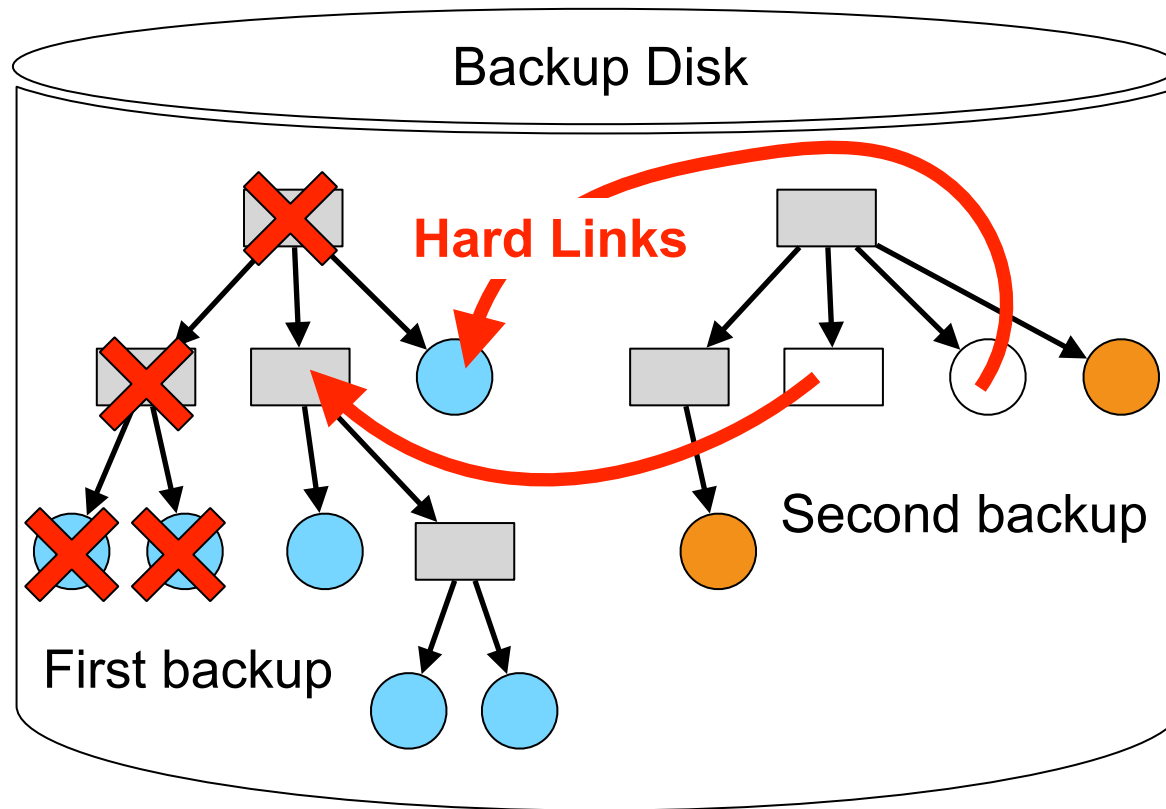
When the user does a new backup, content that has not changed is **hard-linked** to the previous backup!

# MacOS Time Machine



- The user can now safely delete the first backup by deleting the top-level directory...

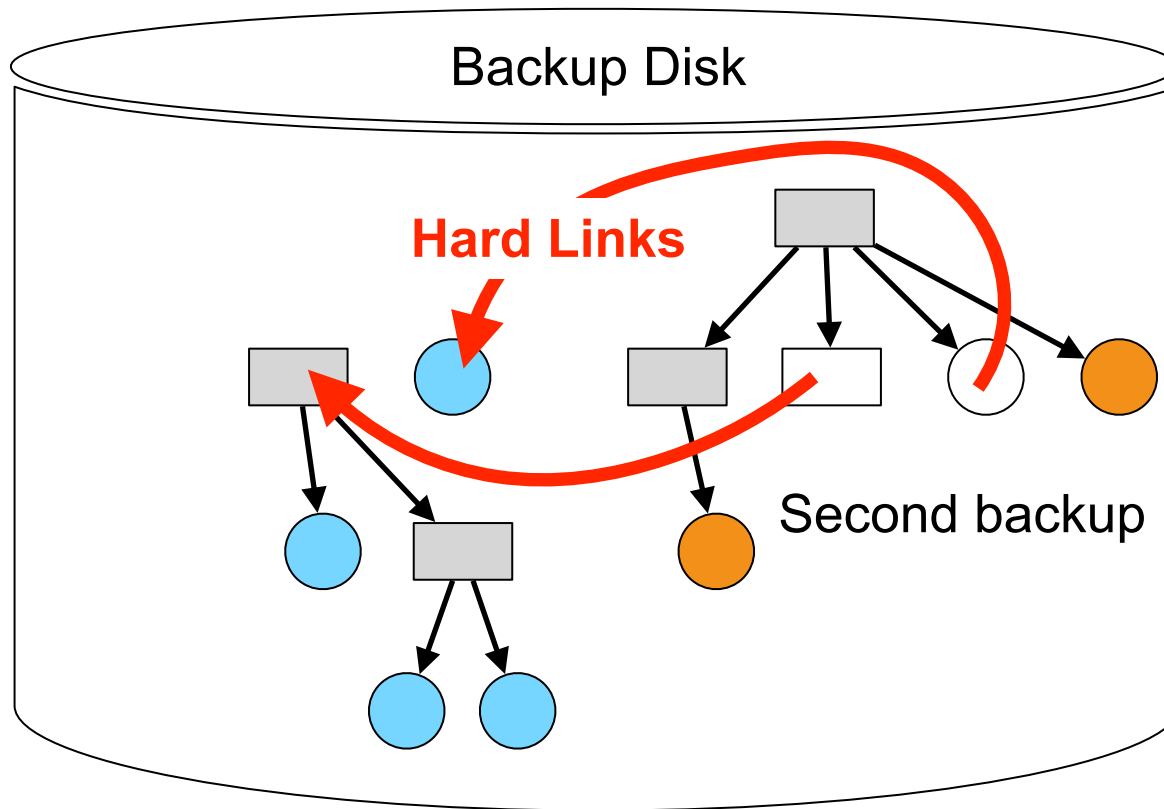
# MacOS Time Machine



- The user can now safely delete the first backup by deleting the top-level directory...



# MacOS Time Machine



- The user can now safely delete the first backup by deleting the top-level directory...



# MacOS Time Machine

## ■ Advantages:

- Extremely simple to implement
- The backup can be navigated in all the normal ways, without any specific software!
- Provided backups are frequent, they are done by creating mostly hard links (which is MUCH faster than copying data)

## ■ Drawback:

- If you change 1 byte in a 10GB file, then you copy the whole file again

## ■ We can implement this on Linux, but we don't have hard-linking of directories!

- So for each backup we have to re-create the whole directory structure and create hard links to all individual files!
- Such implementations do exist, but they have much higher overhead than Time Machine
  - Unless you install a file system that allows hard-linking of directories

# File System Mounting

- There are typically multiple file systems on the same machine
- Each file system is **mounted** at a special location, the **mount point**
  - Typically seen as an initially empty directory
- When given a mount point, a volume, a file system type, the OS
  - Asks the device driver to read the volume on the device
  - Checks that the volume does contain a valid file system
  - Makes note of the new file system at the specified mount point, where the content will be seen as if it was just the content of a directory
- Mac OS X: all volumes are mounted in the **/Volumes/** directory
  - Including temporary volumes on USB keys, CDs, etc.
- UNIX: volumes can be mounted anywhere
  - The **mount** command lists all mounted volumes
- Windows: volumes were identified with a letter (e.g., A:, C:), but current versions, like UNIX, allow mounting anywhere

# Protection

- File systems provide controlled access
- General approach: Access Control Lists (ACLs)
  - For each file/directory, keep a list of all users and of all allowed accesses for each user
  - Protection violations are raised upon invalid access
- Problem: ACLs can be very large
- Solution: consider only a few groups of users and only a few possible actions
- UNIX:
  - User, Group, Others not in Group, All (ugoa)
  - Read, Write, Execute (rwx)
  - Represented by a few bits
  - **chmod** command:
    - e.g., `chmod g+w foo` (add write permission to Group users)
    - e.g., `chmod o-r foo` (remove read permission to Other users)



# Conclusion

- You have all used file systems before, so these lecture notes attempts to focus on things that some of you may not have been completely familiar with
  - Open file table
  - Hard links
- See OSTEP 39 for more details
- In the next set of lecture notes we'll look at how a file system is implemented!