File System Interface

ICS332

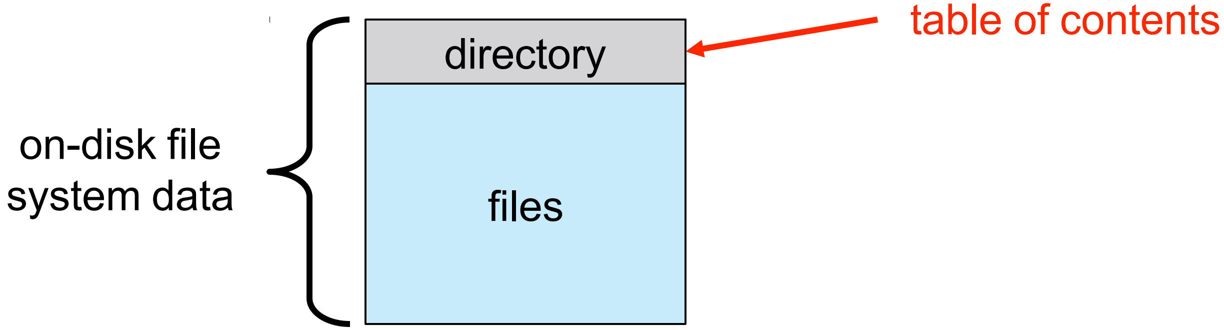
Operating Systems

Files and Directories

- Features
  - A file system implements the file abstraction for secondary storage
  - It also implements the directory abstraction to organize files logically
- Usage
  - It is used for users to organize their data
  - It is used to permit data sharing among processes and users
  - It provides mechanisms for protection

File System

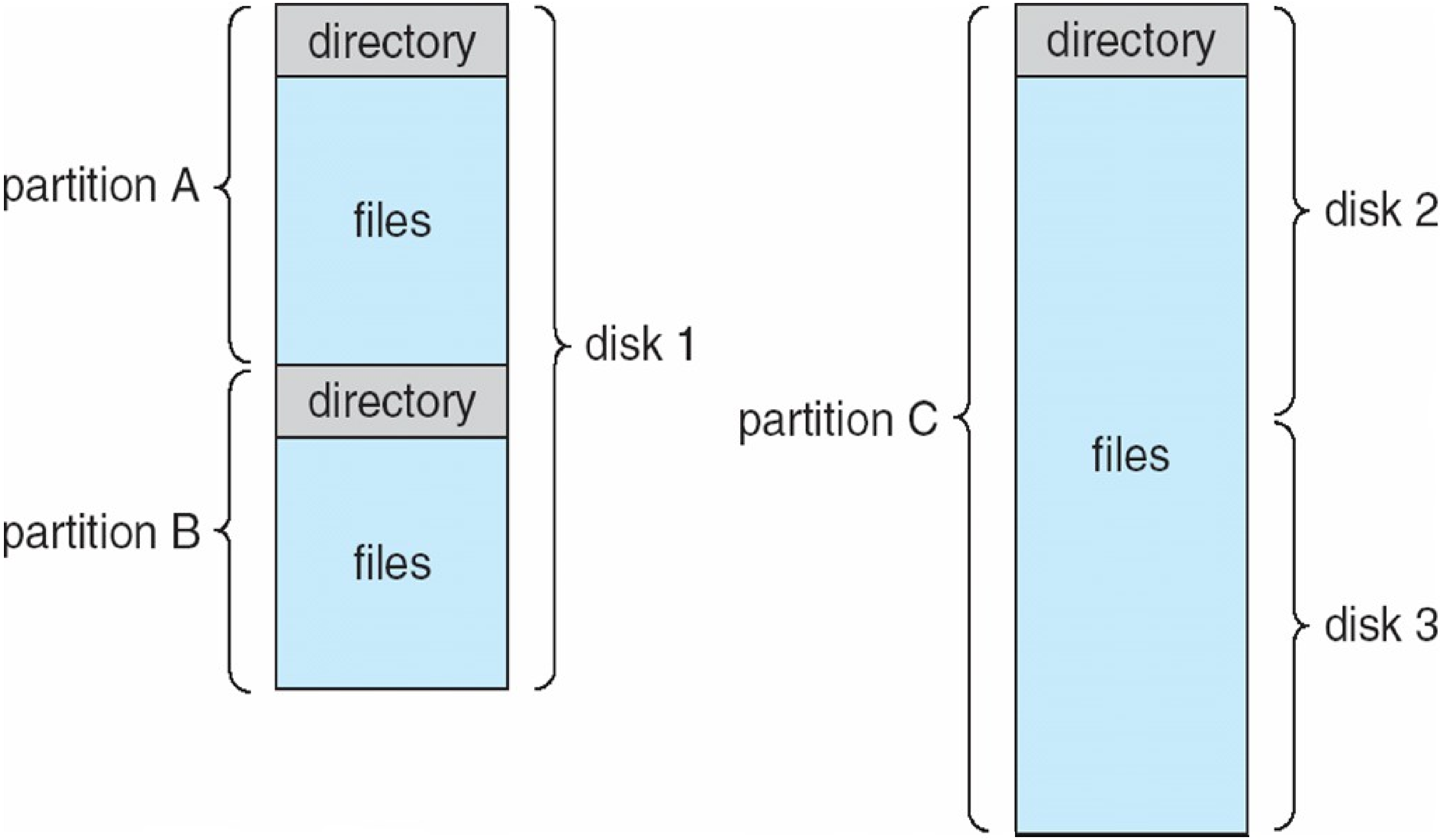
- The term “File System” is a bit confusing
  - The component of the OS that knows how to do “file stuff”
  - A set of algorithms and techniques
  - The content on disk that describes a set of files



- Remember that a disk can be partitioned arbitrarily into logically independent partitions
- Each partition can contain a file system
  - In this case the partition is often called a volume (e.g., C:, A:)
- One can have multiple disks, each with arbitrary partitions, each with a different file system on it

File Systems

- Example with 2 disks, and 3 file systems



File and File Type

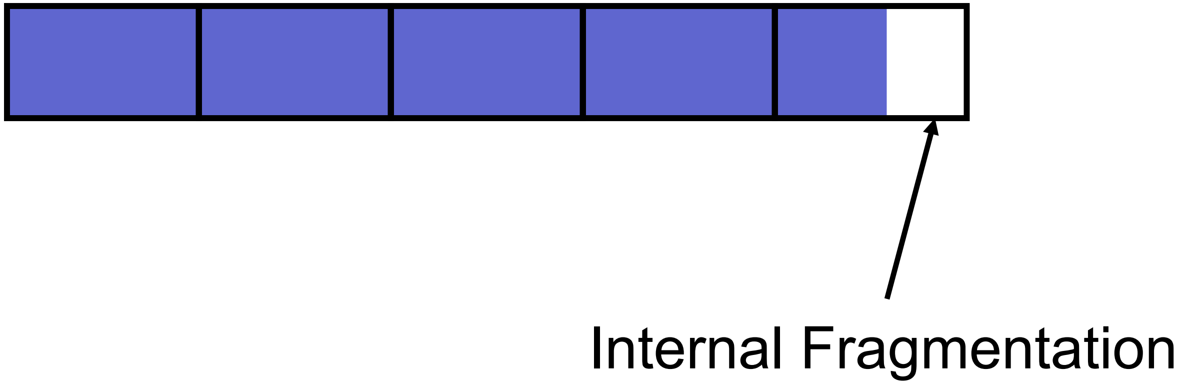
- A file is data + properties (or attributes)
  - Content, size, owner, last read/write time, protection, etc.
- A file can also have a type
  - Understood by the File System
    - e.g., regular file, logical link, device
  - Or understood by the OS
    - Executable, shared library, object file, text, binary, etc.
- In Windows a file type is encoded in its name
  - .com, .exe, .bat, ...
  - Some known to the OS, some just known to applications
- In Mac OS X, each file is associated with a type and the name of the program that created it
  - Done by the create() system call for all files
  - Allows for double clicks to remember which program to use
- In Linux a file type is encoded only in its content
  - “Magic” numbers, first bytes (#!...)
  - Some files have no type and filenames are arbitrary

File Structure

- Question: should the OS know about the structure of a file?
  - The more different structures the OS knows about the more "help" it can provide applications that use particular file types
  - But then, the more complicated the OS code is
    - And it may be too restrictive: e.g., assume all binary files are executable!
- Modern OSes support very few files structures:
  - Files are sequences of bytes that the OS doesn't know about but that have meaning to the applications
  - Certain files are executables and must have a specific format that the OS knows about
    - Executable formats have evolved throughout the years, partly to accommodate dynamic loading
  - The OS may expect a certain directory structure defining an application
    - e.g., Mac OS X "application bundles"

Internal File Structure

- We've seen that the disk provides the OS with a block abstraction (e.g., 512 bytes)
  - All disk I/O is performed in number of blocks
- Each file is stored in a number of blocks



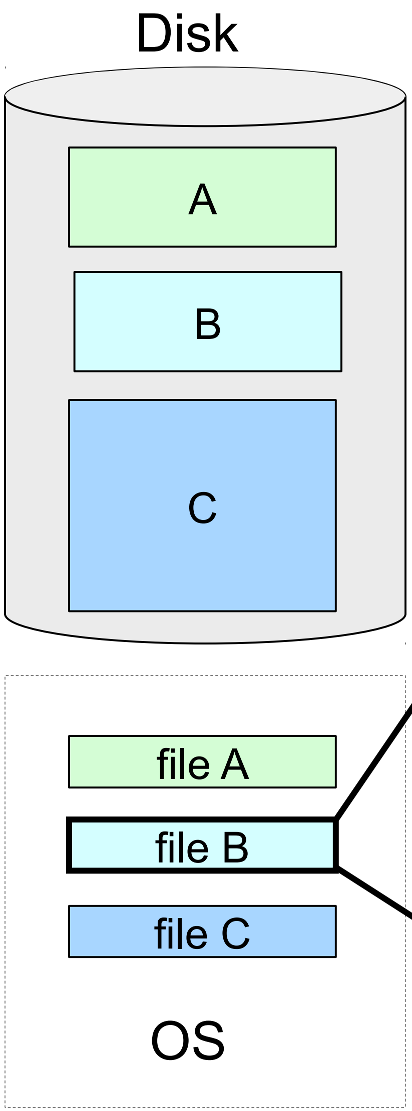
File Operations

- A file is an abstraction, i.e., an abstract data type
- As such the OS defines several file operations
- Basic 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
    - This is called a “seek”
  - Appending at the end of a file
  - Truncating
    - Down to zero size
  - Deleting
  - Renaming

Open Files

- The OS requires that processes open and close files
  - Otherwise, the OS would spend its time searching directories for file names for each file operation
- After an open, the OS copies the file system's file entry (i.e., attributes) into an open-file table that is kept in RAM in the kernel
- The OS keeps two kinds of open-file tables
  - One table per process
  - One global table for all processes
- A process specifies which file the operation is on by giving an index in its local table
  - The famous "filed" (file descriptor) in Linux
- The OS keeps track of a "reference count" for each open file in the global table
  - Incremented each time a process opens the file
  - Decremented each time a process closes the file
- Let's see an example

Open File Tables

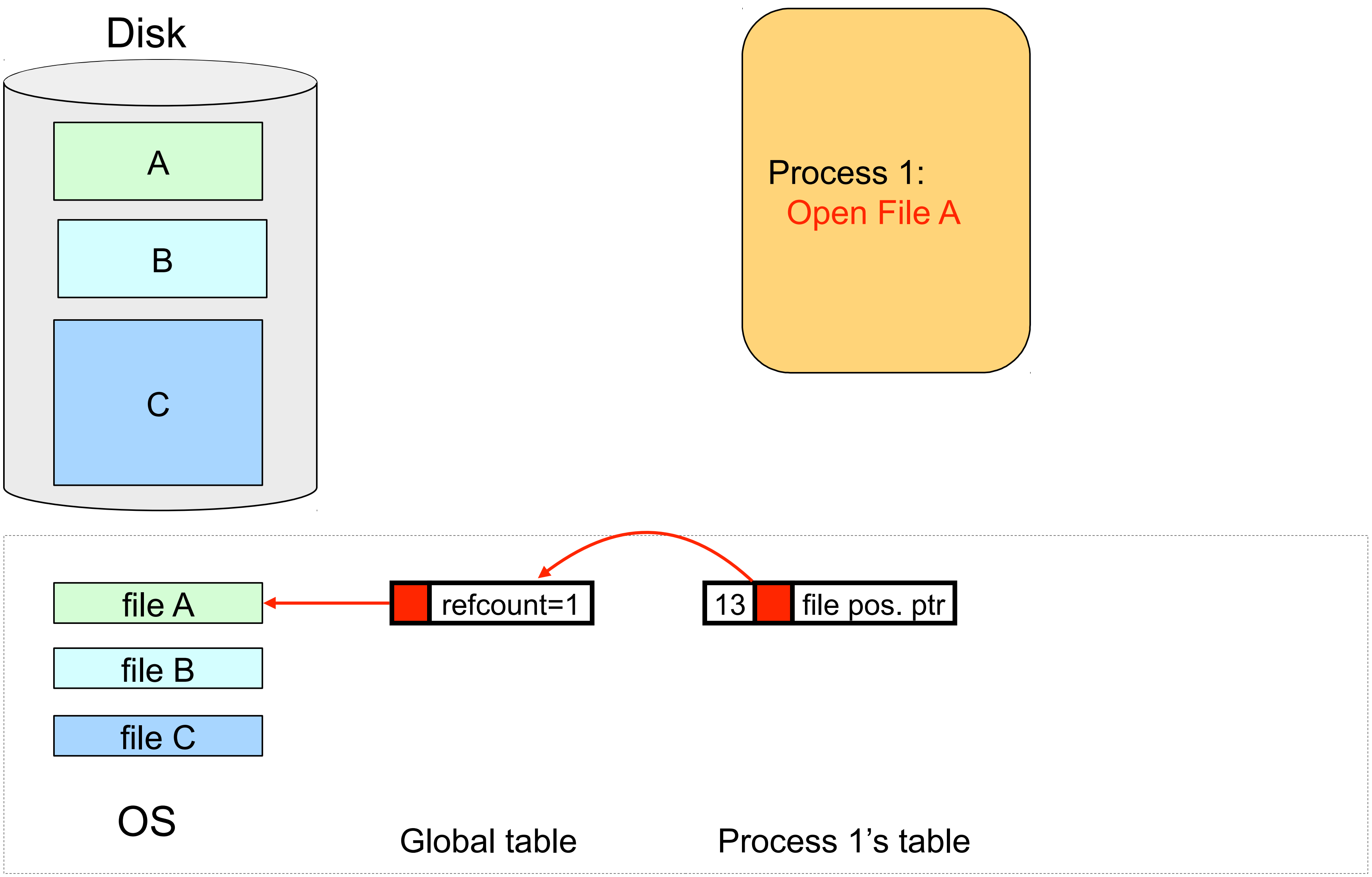


Data structure that contains

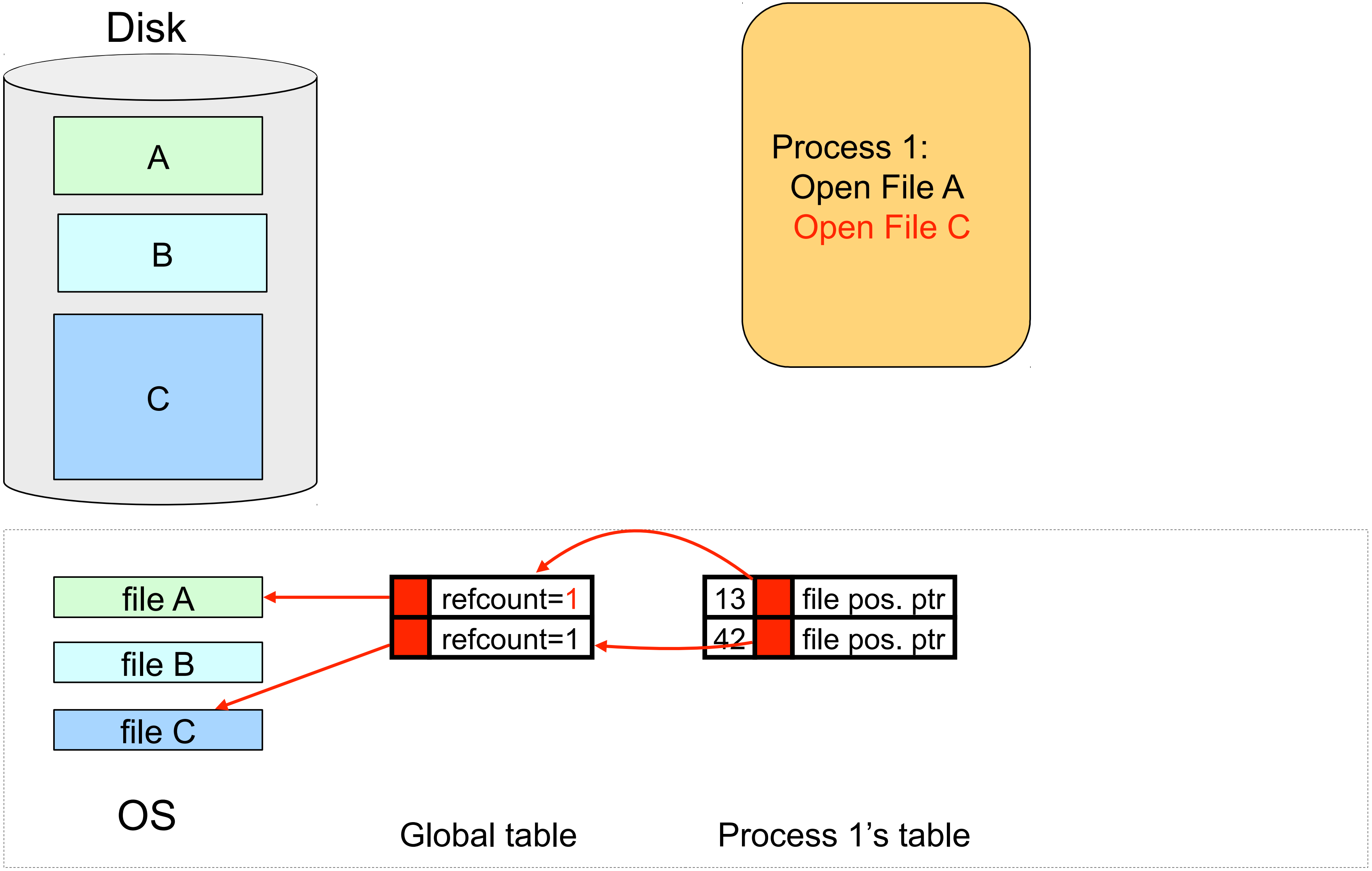
- Location of file on disk
  - Block number of the first block
  - Number of blocks
  - Size in bytes
- Attributes
  - permissions
  - owner
  - time of creation
  - date of last modification
  - ...

- This is kept on disk, but for now we take a logical view of it
  - Let’s just say it’s “in the OS”, or “in the File System”

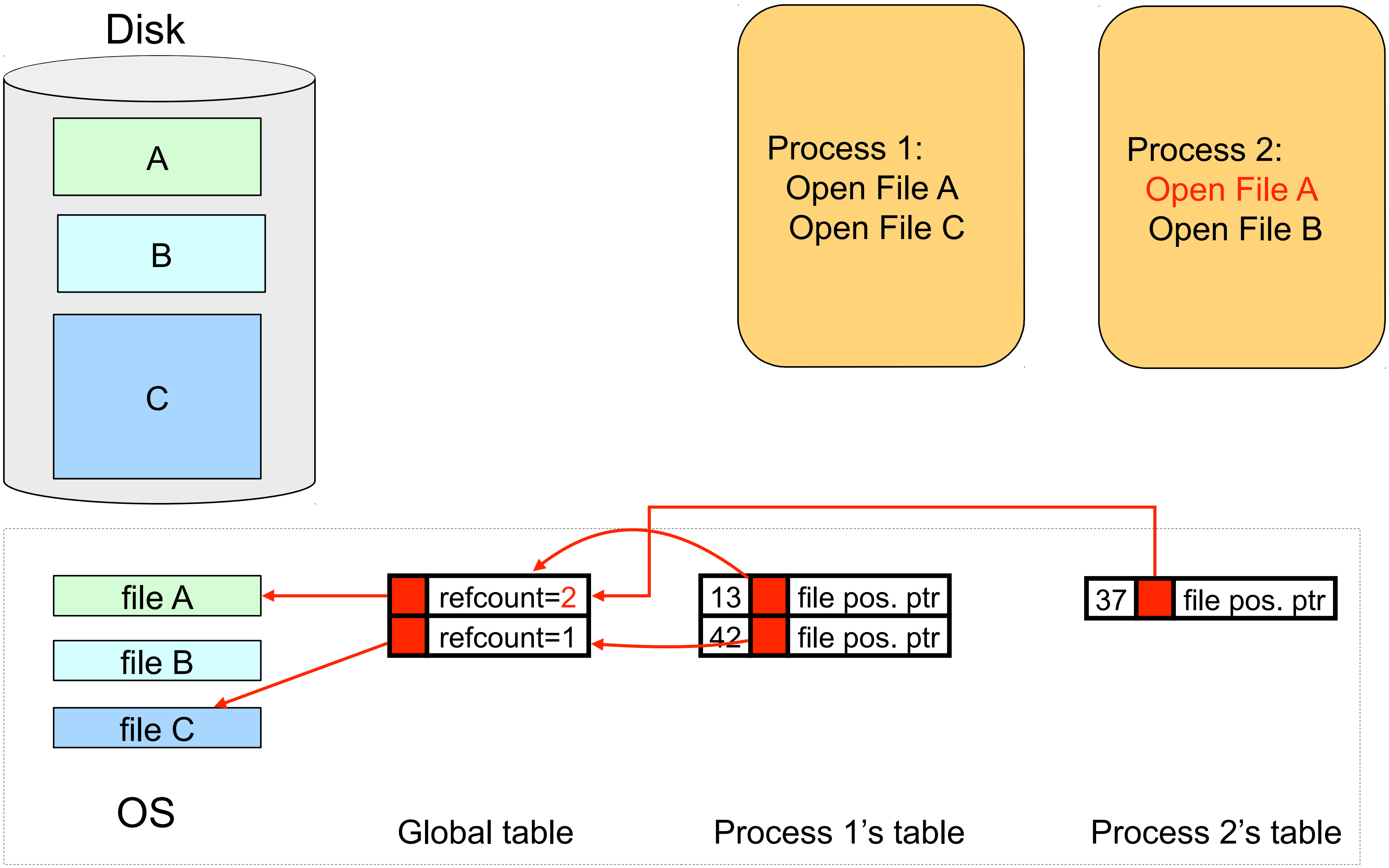
Open File Tables



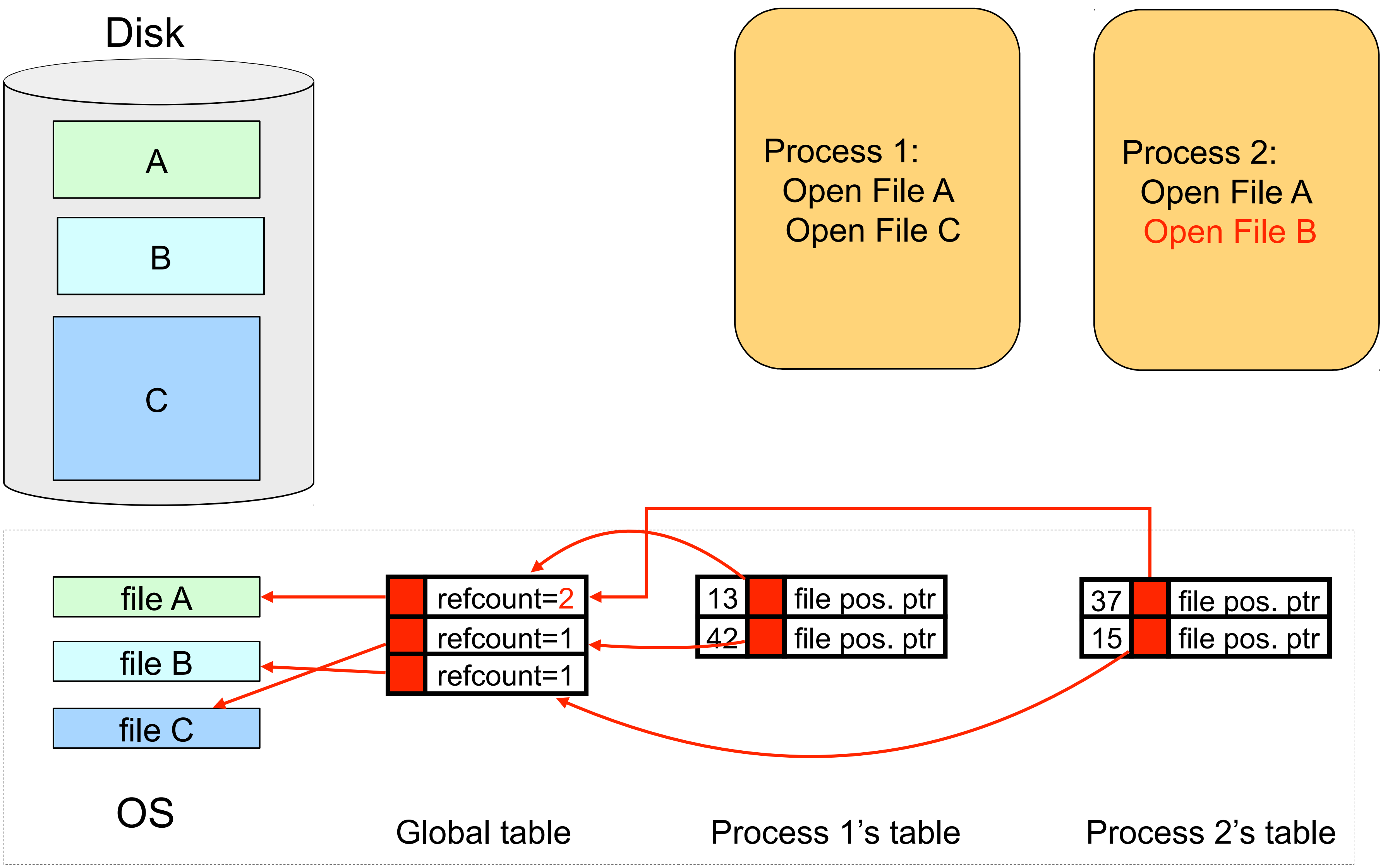
Open File Tables



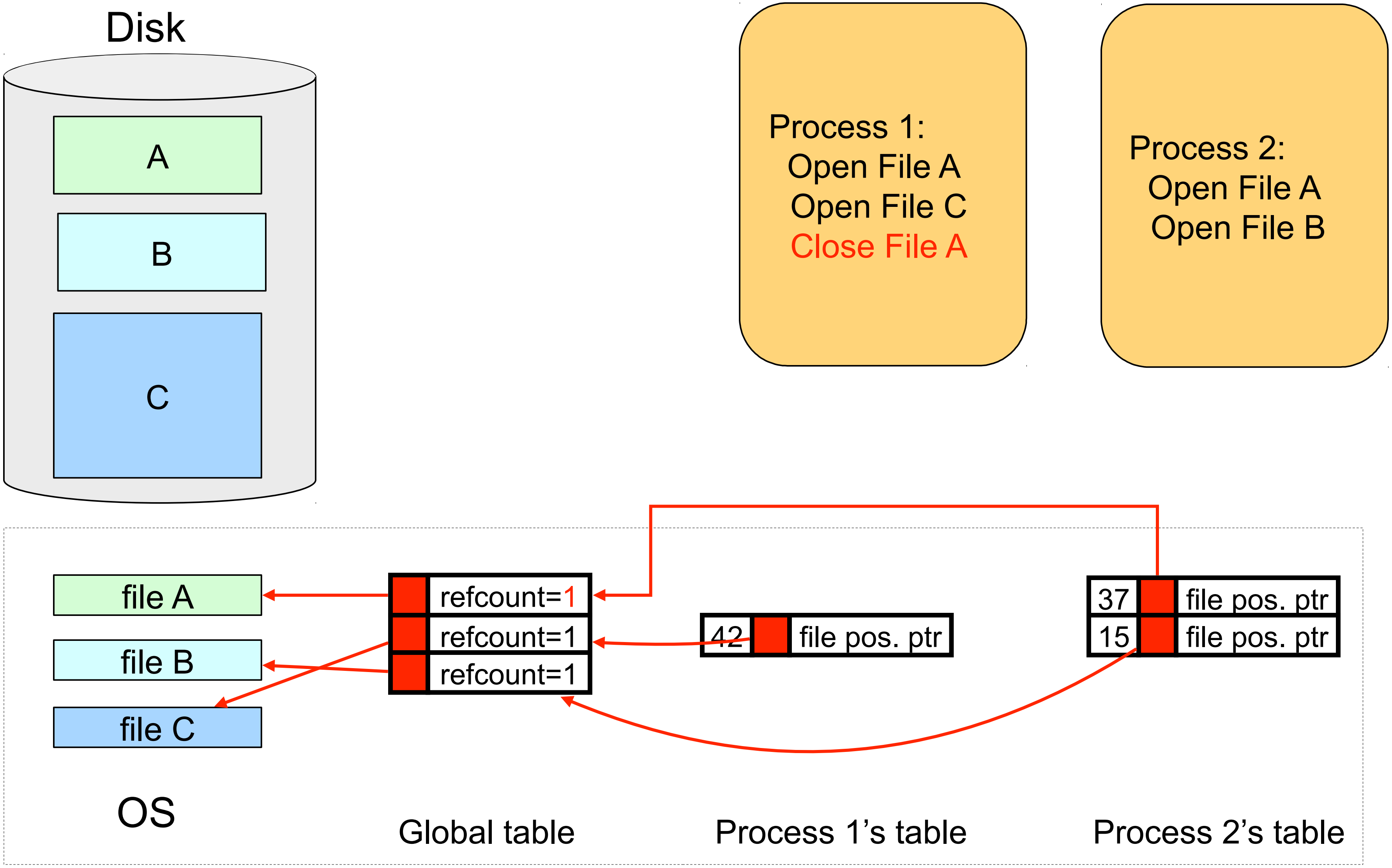
Open File Tables



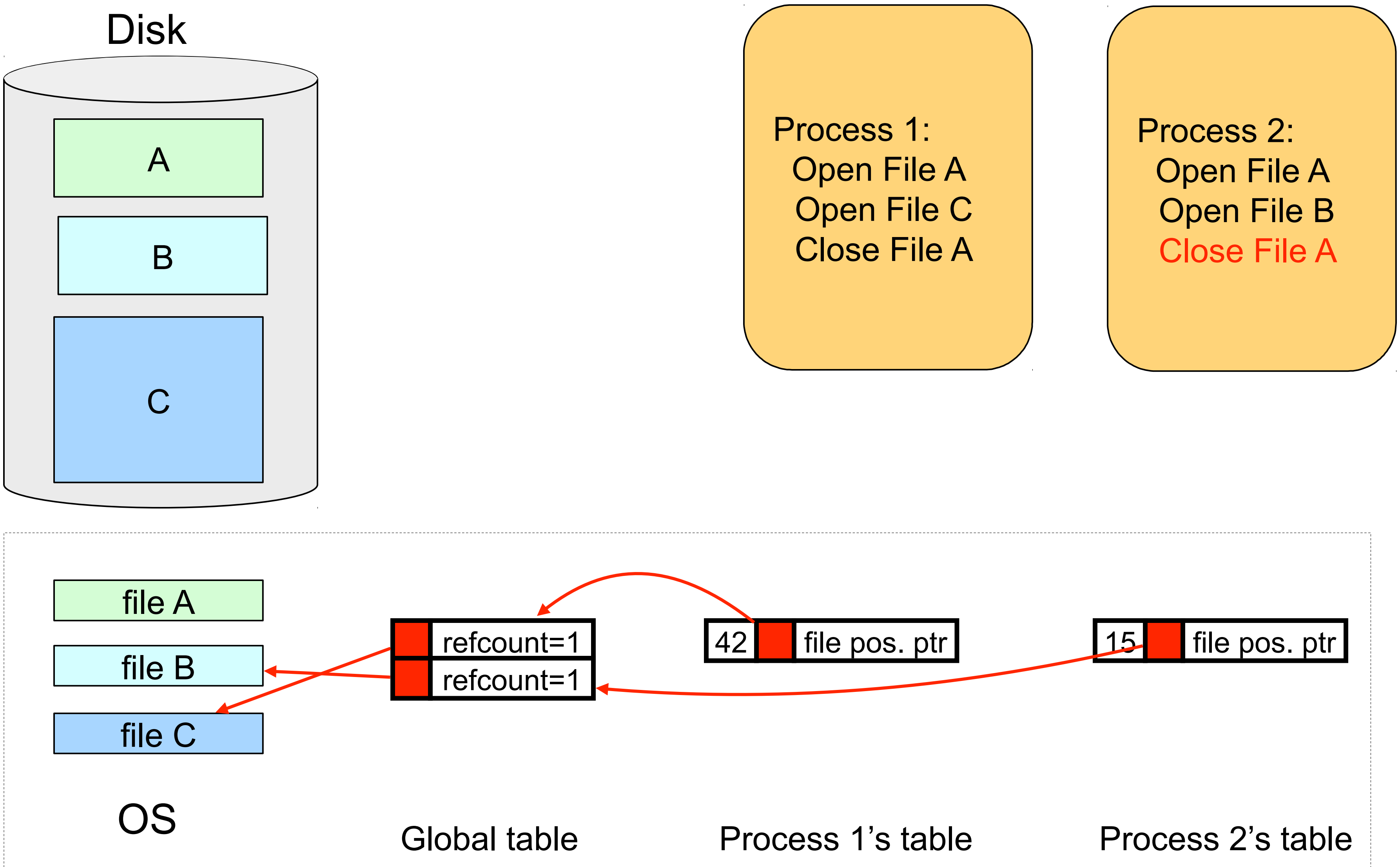
Open File Tables



Open File Tables



Open File Tables



File Locking

- Bad things may happen when multiple processes reference the same file
  - Just like when threads reference the same memory
- A file lock 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
- Typically applications have to implement their own locking
- And of courses there can be deadlocks and all the messiness of thread synchronization

- Let’s look at the Java example in Fig. 10.1 in the book

File Locking in Java

```
import java.io.*;
import java.nio.channels.*;
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String arsg[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            FileChannel ch = raf.getChannel();
            // this locks the first half of the file - exclusive  (one writer)
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /** Now modify the data . . . **/
            // release the lock
            exclusiveLock.release();
```

File Locking in Java (cont.)

```
            // this locks the second half of the file - shared (multiple readers)
            sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);
            /** Now read the data . . . **/
            // release the lock
            sharedLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        } finally {
            if (exclusiveLock != null)
                exclusiveLock.release();
            if (sharedLock != null)
                sharedLock.release();
        }
    }
}
```

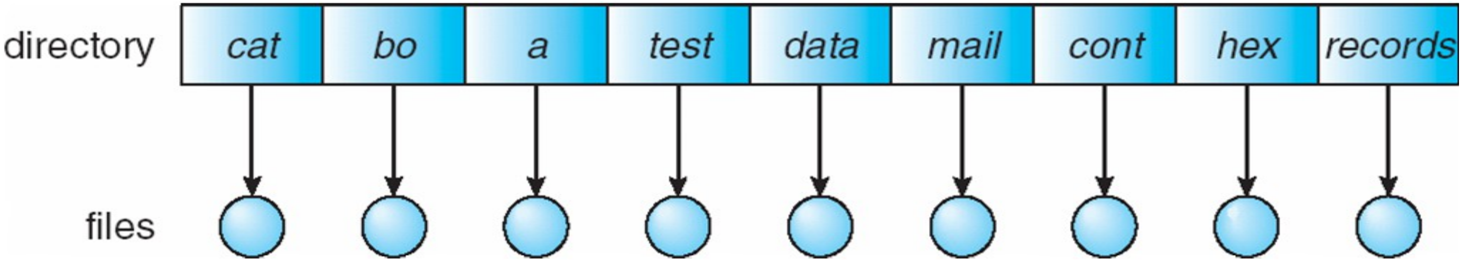
Access Methods

- Sequential Access
  - One byte at a time, in order, until the end
  - Read next, write next, reset to the beginning
- Direct Access
  - Ability to position anywhere in the file
  - Position to block #n, Read next, write next
  - Block number is relative to the beginning of the file
    - Just like a logical page number is relative to the first page in a process' address space
- Indexed Access
  - A file contains an index of "file record" locations
  - One can then look for the object in the index, and then "jump" directly to the beginning of the record
- Linux and Windows: Direct access
  - It's up to you to implement your own application-specific index
  - But internally the FS does some indexing of blocks, as we'll see
- Older OSes provided other, more involved methods, including indexing
  - e.g., you could tell the OS more information about the logical structure of your file

Directories

- We're used to file systems that support
  - multiple directory levels
  - the notion of a *current* directory

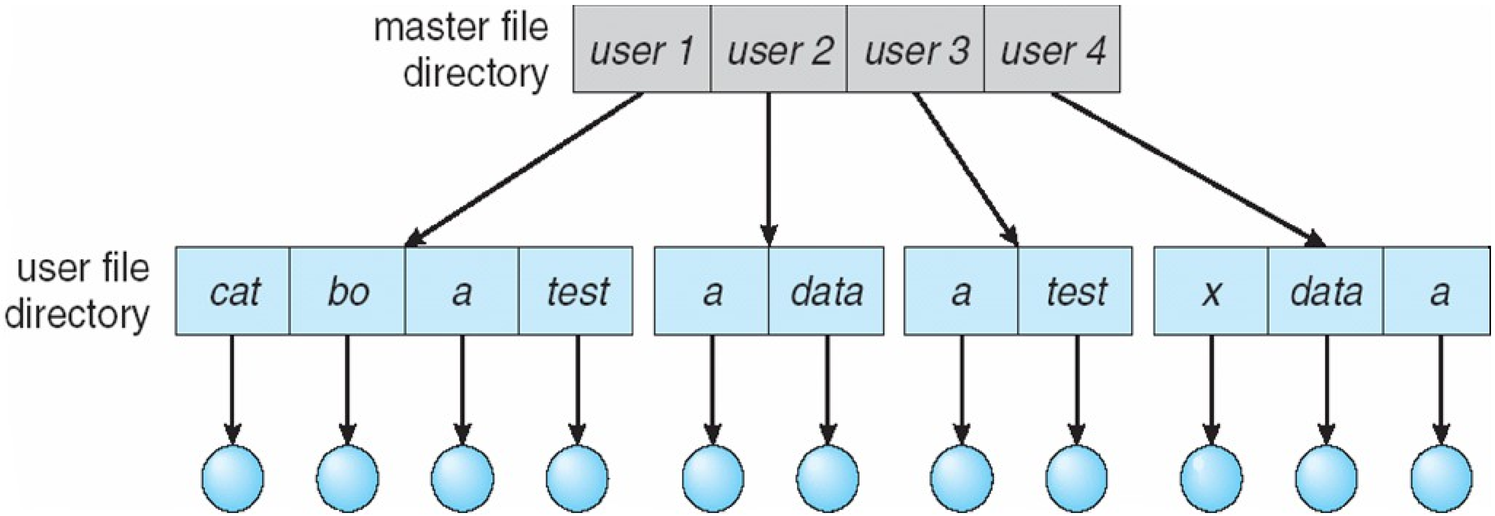
- Single-Level directory



  - Naming conflicts
    - Have to pick longer, and longer unique names
  - Slow searching

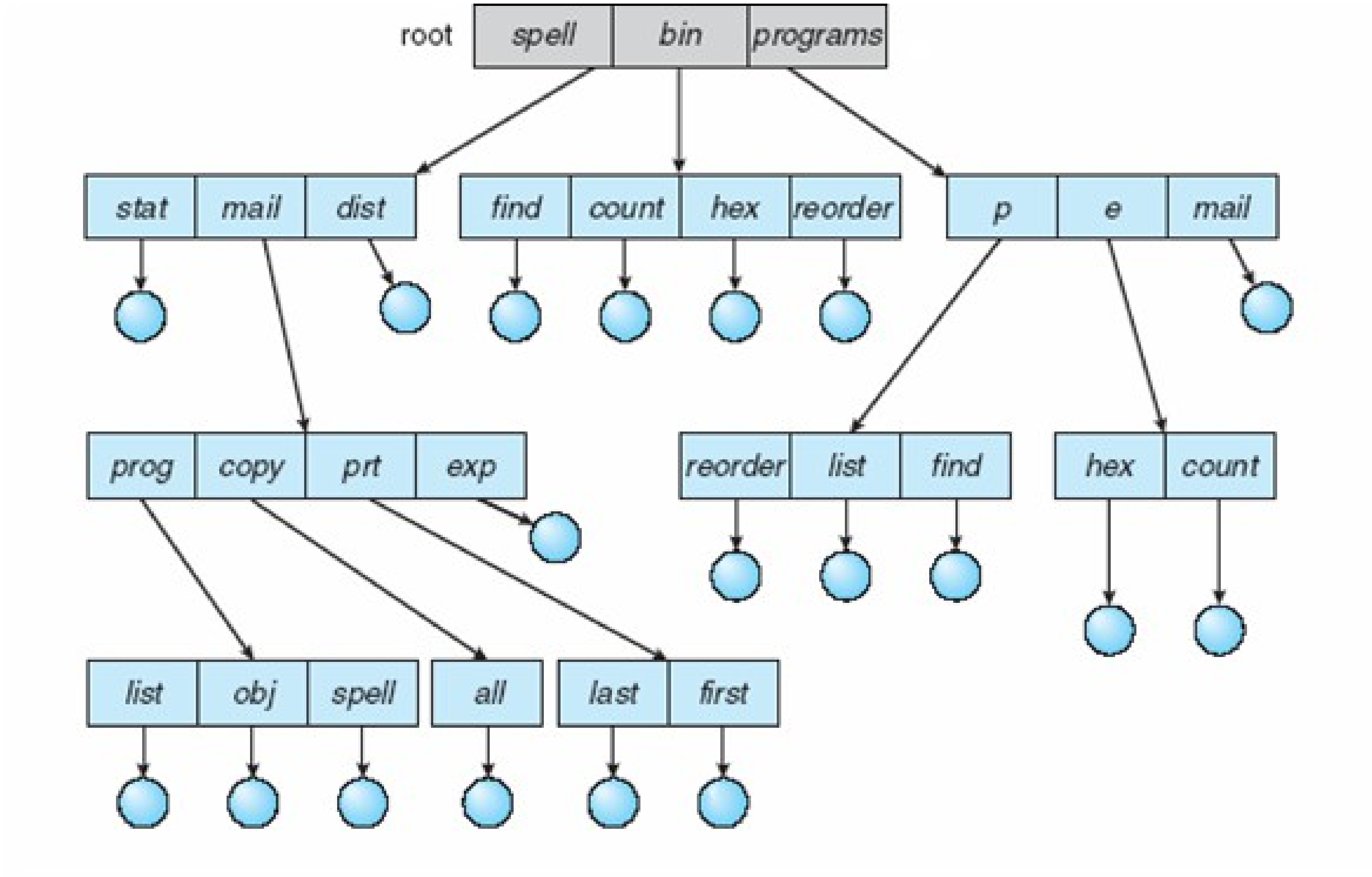
Directories

- Two-Level Directory



  - Faster searching
  - Still naming conflict for each user

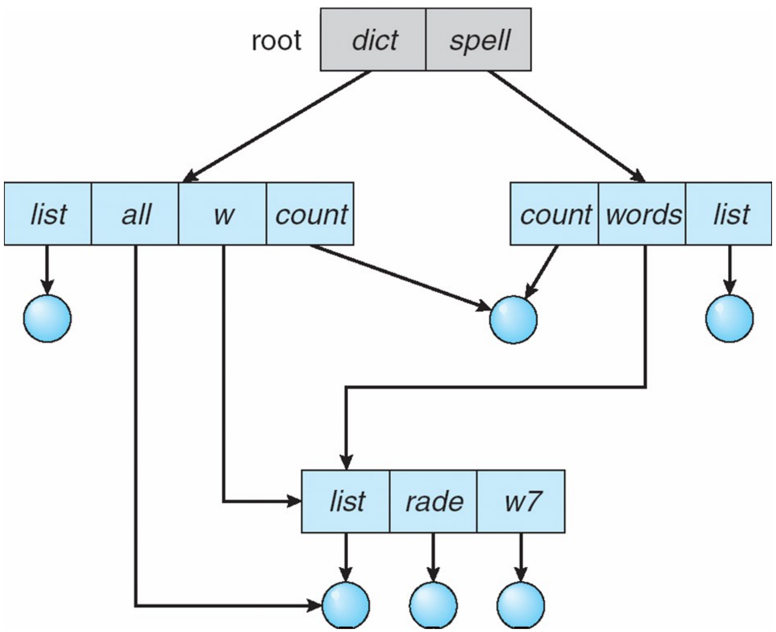
Tree-Structured Directories



Tree-Structured Directories

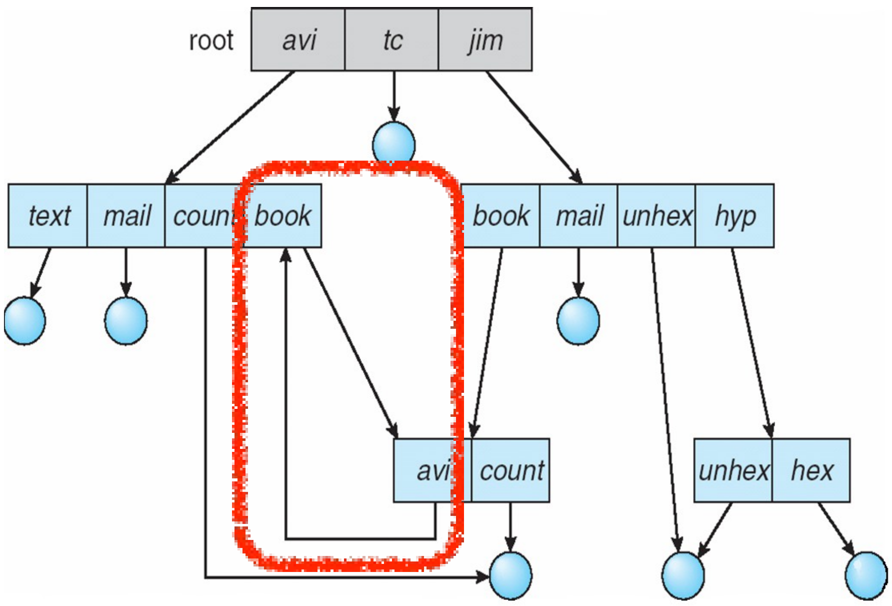
- More general than 1- and 2-level schemes
- Each directory can contain files or directories
  - Differentiated internally by a bit set to 0 for files and 1 for directories
- Each process has a current directory
  - Relative paths
  - Absolute paths
- Path name translation, e.g., for “/one/two/three”
  - Open “/” (the file system knows where that is on disk)
  - Search it for “one” and get its location
  - Open “one”, search it for “two” and get its location
  - Open “two”, search it for “three” and get its location
  - Open “three”
- The OS spends a lot of time walking directory paths
  - Another reason why one separates “open” from “read/write”
  - The OS attempts to cache “common” path prefixes
- But what we have in modern systems is actually more complicated...

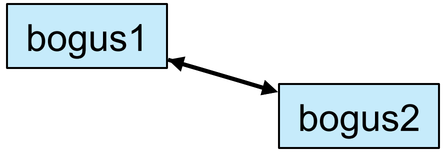
Acyclic Graph Directories



- Files/directories can be shared by directories
- A hard link is created in a directory, to point to or reference another file or directory
  - Identified in the file system as a special file
- The file system keeps track of reference count for each file, and deletes the file when the last reference is removed
- A symbolic link does **not** count toward the reference count
  - You can think of it as an alias for the file (if you remove the alias, nothing happens)
  - If the target file is removed then the alias simply becomes invalid
- This is the UNIX view of links, as implemented by the "In" command
  - No hard-linking of directories
- Acyclic is good for quick/simple traversals
- Simple way to prohibit cycles: no hard linking of directories!
  - Used in Linux

General Graph Directory

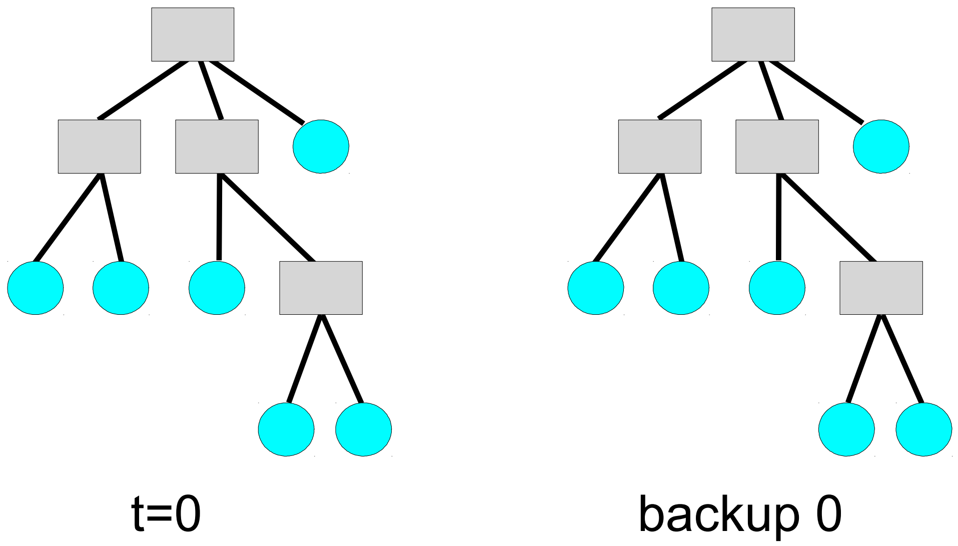




- In this scheme users can do whatever they want
- Directory traversals algorithms must be smarter to avoid infinite loops
- Garbage collection could be useful because ref counts may never reach zero
  - But way too expensive in practice

Mac OSX Time Machine

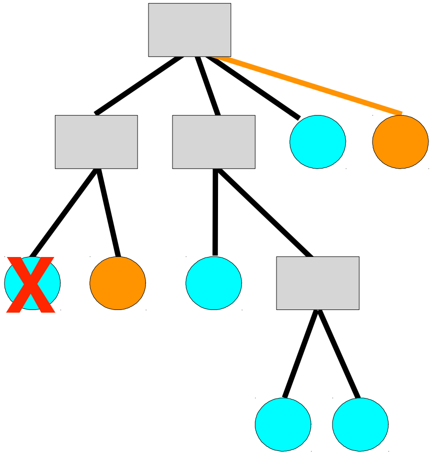
- Time Machine is the backup mechanism introduced with Leopard
- It uses hard links
  - Every time a new backup is made, a new backup directory is created that contains a snapshot of the current state of the file system
  - Files that haven't been modified are hard links to previously backed up version
    - A new backup should be mostly hard links instead of file copies(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)



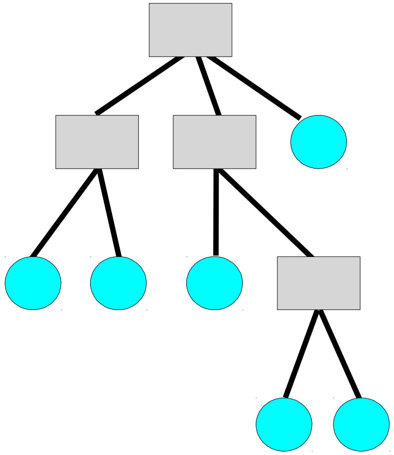
At time t=0, the first backup is initialized, meaning that it's a full copy of the directory structure and files

Mac OSX Time Machine

- Time Machine is the backup mechanism introduced with Leopard
- It uses hard links
  - Every time a new backup is made, a new backup directory is created that contains a snapshot of the current state of the file system
  - Files that haven't been modified are hard links to previously backed up version
    - A new backup should be mostly hard links instead of file copies(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)



t=1

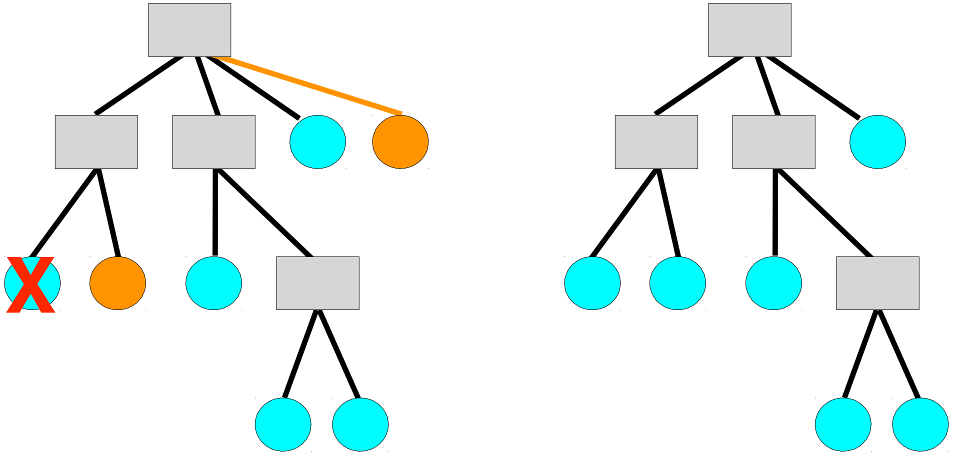


backup 0

By time t=1, a file has been modified, another one is added, an another one is deleted

Mac OSX Time Machine

- Time Machine is the backup mechanism introduced with Leopard
- It uses hard links
  - Every time a new backup is made, a new backup directory is created that contains a snapshot of the current state of the file system
  - Files that haven't been modified are hard links to previously backed up version
    - A new backup should be mostly hard links instead of file copies(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)



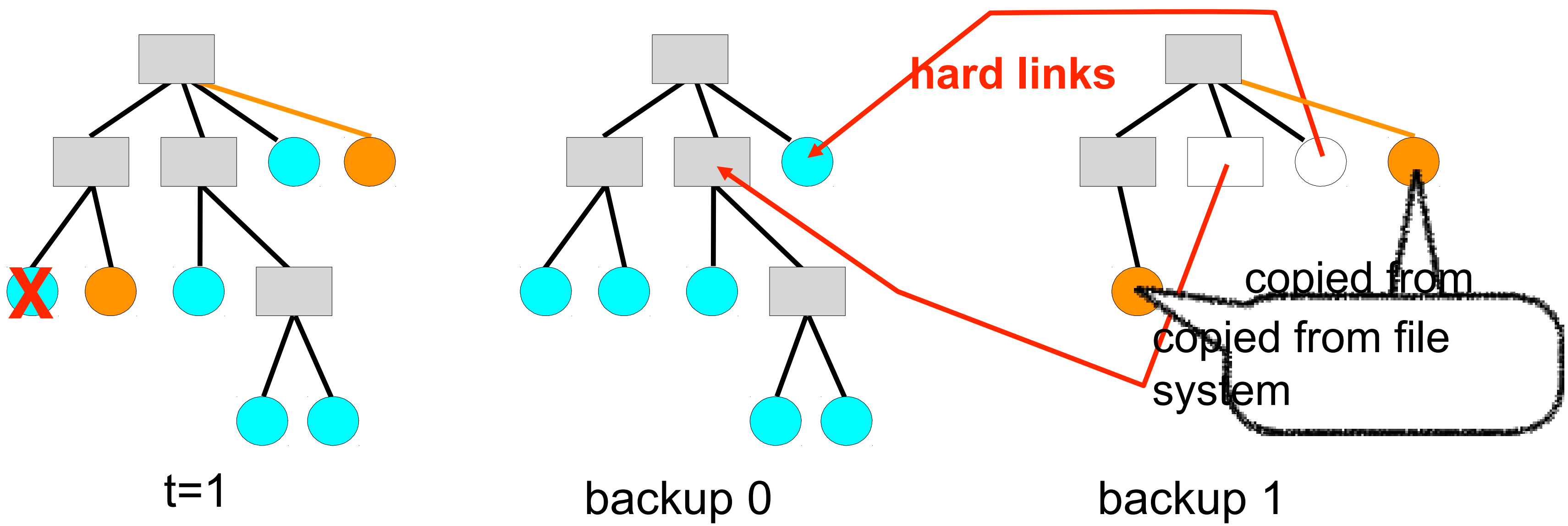
t=1

backup 0

At time t=1, a new backup is triggered by the user

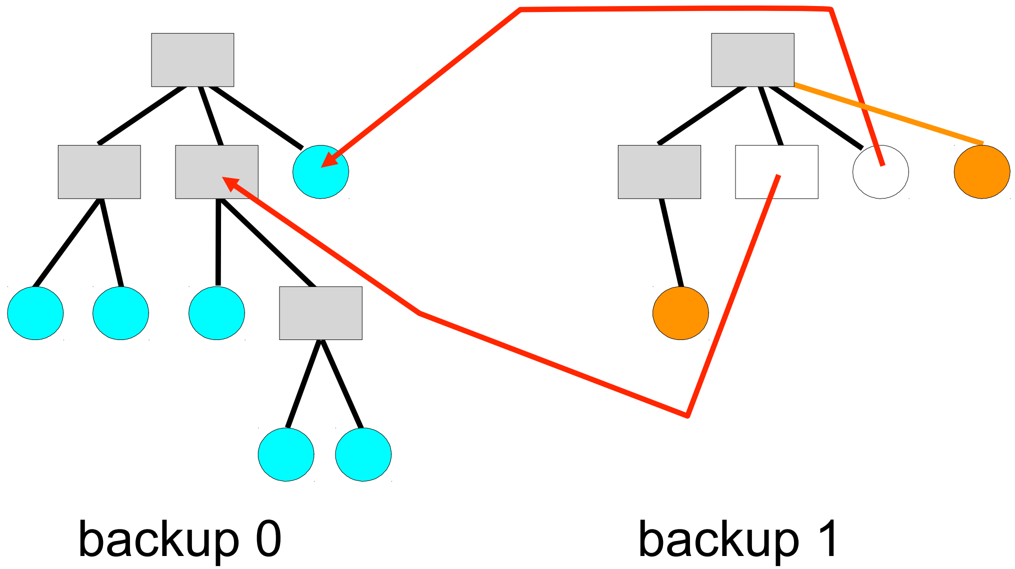
Mac OSX Time Machine

- Time Machine is the backup mechanism introduced with Leopard
- It uses hard links
  - Every time a new backup is made, a new backup directory is created that contains a snapshot of the current state of the file system
  - Files that haven't been modified are hard links to previously backed up version
    - A new backup should be mostly hard links instead of file copies(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)



Mac OSX Time Machine

- Time Machine is the backup mechanism introduced with Leopard
- It uses hard links
  - Every time a new backup is made, a new backup directory is created that contains a snapshot of the current state of the file system
  - Files that haven't been modified are hard links to previously backed up version
    - A new backup should be mostly hard links instead of file copies(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)



The user can now remove backup 0

Mac OSX Time Machine

- Advantages
  - Extremely simple to implement
  - The back up can be navigated in all the normal ways, without Time Machine
  - 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 10GB
  - But how often does this happen??
- For efficiency, Mac OSX allows hard linking of directories
  - Cycles in the directory hierarchy must be detected, i.e., more complicated file system code
  - Complexity deemed worthwhile by Mac OSX developers

Time Machine on Linux?

- Give how simple and elegant Time Machine is, one may want to implement it on Linux
  - Could be an interesting course project
- But because Linux doesn't allow hard-linking of directories, one would have to recreate the **whole** directory structure for each backup
  - While would take space and, more importantly perhaps, a lot of time
- Such implementations exist, but if you use a standard Linux file system that does not allow cycles in the directory structures, it won't be efficient

Hard Links on Linux

- It turns out that, on Linux, whenever a file is opened by a process, a hard link to the file is created
- Say that process with PID 2233 calls the open() system call to open a file “/home/casanova/somefile”
- open() returns a “file descriptor”, i.e., an integer, say 55
- At that point, a hard link to “/home/casanova/somefile” is created in “/proc/2233/fd/55”
- If, while the process is running, “/bin/rm /home/casanova/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...

File System Mounting

- There can be multiple file systems
- Each file system is "mounted" at a special location, the mount point
  - Typically seen as an empty directory
- When given a mount point, a volume, a file system type, the OS
  - asks the device driver to read the device's directory
  - checks that the volume does contain a valid file system
  - makes note of the new file system at the specified mount point
    - The OS keeps a list of mount points
- 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
- Windows: volumes were identified with a letter (e.g., A:, C:), but current versions, like UNIX, allow mounting anywhere
- On Linux the "mount" command lists all mounted volumes

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

- In the next set of lecture notes we'll look at how a file system is implemented...