CS-446/646

Filesystems

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Filesystems

Filesystem

Main tasks:

- *Persistence* of Data & its structure
- Associate Bytes with name (*Files*)
- Associate names with each other (*Directories*)
- Can implement *Filesystems* on Disk, over *Network*, in *Memory*, in Non-Volatile RAM (NVRAM), on Tape, …
  - Focus on Disk and generalize later

File

Named Bytes on Disk
- Data with some properties
- Contents, size, owner, last read/write time, protection, etc.

How a *File*'s data is managed by the *Filesystem* (more later)
- Basic idea (in Unix):

  A **`struct`** called an *Index Node* : The "*inode*"
  - Describes where on the Disk the *Blocks* for a given *File* are located
    - *inode* also holds some extra information (Metadata)
  - Disk stores an array of *inodes* : The "*inode Table*"
  - *inode #* is the index in the *inode Table* : The "*i-number*"

Filetypes

A *File* can also have a Type

- Understood by the *Filesystem*
  - block, character, device, portal, link, etc.
- Understood by other parts of the OS or runtime libraries
  - executable, dll, source, object, text, etc.

A *File*'s Type can be encoded in its *Filename* or *Contents*

- Windows encodes type in name (**`.com`**, **`.exe`**, **`.bat`**, **`.dll`**, etc.)
- Unix also encodes type in contents (magic numbers, initial characters)
  - e.g. **`#!`** for *Shellscripts*

Basic *File* Operations

Unix

```
creat(name)
rename(oldname, newname)
open(name, how)
read(fd, buf, len)
write(fd, buf, len)
truncate(fd, len)
sync(fd)
lseek(fd, pos)
close(fd)
unlink(name)
```

Windows

```
CreateFile(name, CREATE)
CreateFile(name, OPEN)
ReadFile(handle, …)
WriteFile(handle, …)
FlushFileBuffers(handle, …)
SetFilePointer(handle, …)
CloseHandle(handle, …)
DeleteFile(name)
CopyFile(name)
MoveFile(name)
```

File Access Methods

Filesystem usually provides different access methods (ways for accessing data in a *File*):

- *Sequential Access*
  - Read Bytes one at a time, in order
- *Random Access*
  - *Random* Accessing of a given *Block* Number/Byte-offset
- *Record Access*
  - *Record-Oriented Filesystems* stores *Records* (group of related Data) instead of just Bytes of information
  - *File* is array of fixed-length or variable-length *Records*
  - Read/written either *Sequentially* or *Randomly* by *Record* #
- *Indexed Access*
  - *Filesystem* contains an *index* to a particular field of each *Record* in a *File*
  - Reads specify a value for that field and the system finds the *Record* via the *index*

Directories

- Problem:
  How to reference *Files*

Should users remember where on Disk their *Files* are located (e.g. Disk *Sector Number*)?

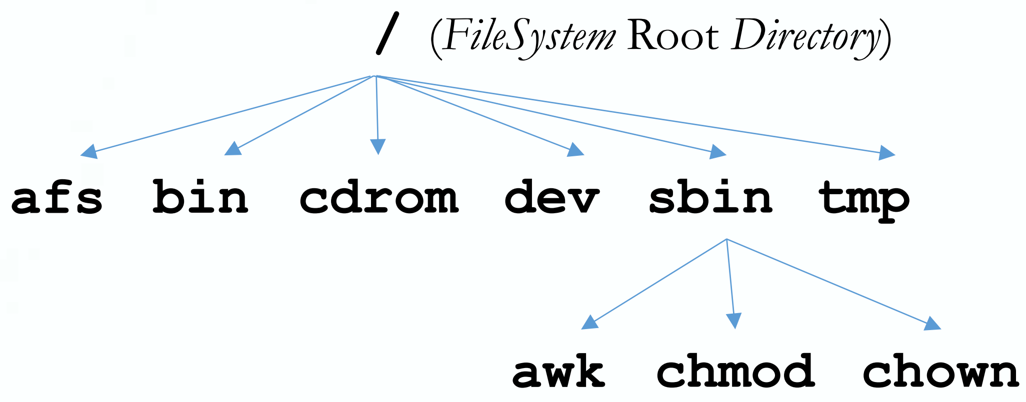
- Human interface means we need human-readable names
- We use *Directories* to map names to *File Blocks*

Directories serve two purposes

- For users, they provide a structured way to organize *Files*
- For *Filesystem*, they provide a convenient naming interface that allows the separation of *Logical Organization* of *Files*, from *Physical Placement* on the Disk

Hierarchical Directories

- Used since CTSS (1960s)
  - Unix picked up and used nicely



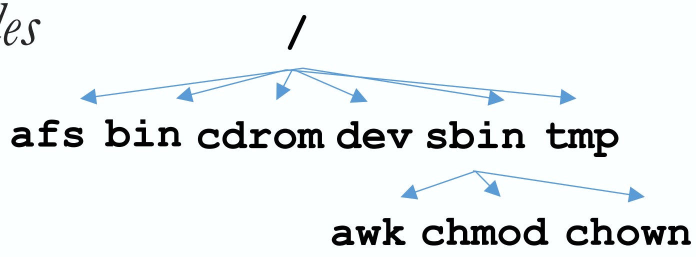
- Large *Namespaces* tend to be *Hierarchical*
  - IP Addresses, Domain Names, scoping in Programming Languages, etc.

Directory Internals

A *Directory* is a list of *Directory Entries*

Directory Entry:

- <**name**, **inode#**> tuple: The *inode #* gets us to an *inode*, that contains a Disk location
- *Directories* stored on Disk just like regular *Files*
  - Its *Filetype* is: *Directory*
  - Users can read its Data just like any other *File*
  - Only special *System Calls* can write to its Data
  - Data is list of *Directory Entries* that point to other Disk locations (through *inode*#s)
  - *File* pointed to by an *inode #*, may also be a *Directory* (*inode* is of *Directory Filetype*)
  - *Filesystem* becomes a *Hierarchical Tree*



File contents for **/**

<afs,1021>
<tmp,1020>
<bin,1022>
<cdrom,4123>
<dev,1001>
<sbin,1011>
…

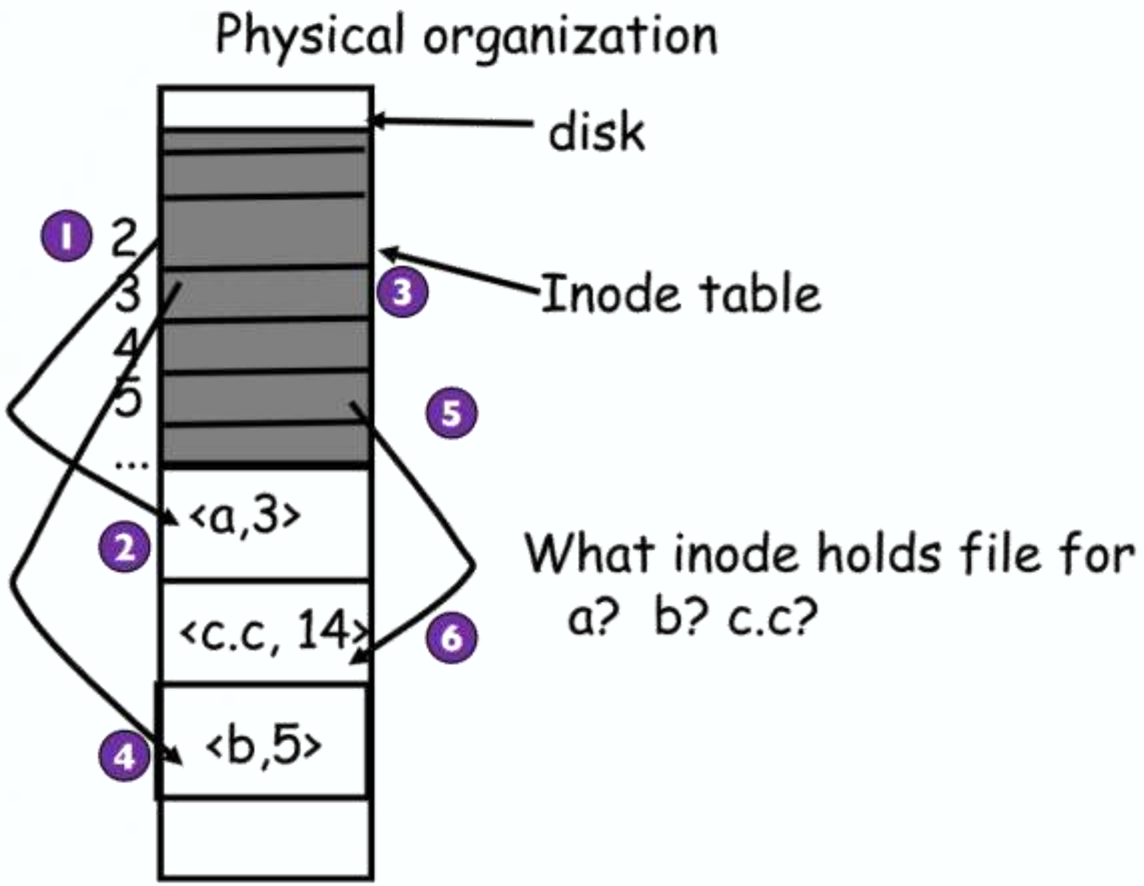
- Simple, and any speedup in *File* operations also speeds up *Directory* operations

***Pathname* Translation** (simple & high-level overview)

- To open *File* **`/one/two/three`** the *Filesystem* will translate as follows:
- *Directory Entries* map *Filenames* to Disk location(s) (via the *inode* # that gets us to an *inode*)
- Open *Directory* **`"/"`**: Root *Directory* is always *inode #2* (more later)
- Search for the *Directory Entry* named **`"one"`**,
  get *inode* # and then *inode* of **`"one"`**, get its Disk location
- Open *Directory* **`one`**, search *Directory Entries* for the one named **`"two"`**,
  get *inode* # and then *inode* of **`"two"`**, get its Disk location
- Open *Directory* **`two`**, search *Directory Entries* for the one named **`"three"`**,
  get *inode* # and then *inode* of **`"three"`**, get its Disk location
- Open *File* **`"three"`**

Unix Example

- `/a/b/c.c`:



Unix *inodes* and *Path* Search (more detailed overview)

- Unix *inodes* are not *Directories*
  - An *inode* describes where on the Disk the Data *Blocks* for a *File* are located
  - A *Directory* is like a *File*, so its *inode* just describes the location of its Data *Blocks* on Disk; but also a *Directory*'s Data *Blocks* contain its list of *Directory Entries*
- Each *Directory Entry* maps a *Filename* to an *inode*

  For a not-so-elaborate *Filesystem* structure, to open File **`"/one"`**:
  - Read *inode Table* (more later) into *Memory* and find for **`"/"`** its *inode* and then its Disk *Block Number*(s)
    That is the location of its Data *Block*(s) on Disk
  - Read the Data *Block* of **`"/"`** into *Memory*, look into its Data for a *Directory Entry* named **`"one"`**
    This *Directory Entry* contains the *inode#* for **`"one"`**
  - Find the *inode* for **`"one"`** in the *inode Table* (should be regular *File*) and then its Disk *Block Number*(s)
    That is the location of its *Data Block*(s) on Disk
  - Read the Data *Block*(s) into *Memory* to access the *File* Data

Naming

- Bootstrapping: Where do you start looking?
  - Root *Directory* always *inode #2* → *Note:*
    - #0 used as a NULL value (indicates an *inode* does not exist); i.e. there is no *inode* #0
    - First *inode* is *inode* #1, and is reserved for recording defective *Blocks* on the Disk
- Special names:
  - Root directory: `/`
  - Current directory: `.`
  - Parent directory: `..`
- Some special aliases are provided by the *Shell*, not the *Filesystem*:
  - User's Home *Directory*: `~`
  - Globbing: `foo.*` (expands to all files starting with `foo.`)
- Using the given naming, only need two operations to navigate the entire *Namespace*:
  - `cd <name>` : Move into (change *Current Working Directory* context to) *Directory Name*
  - `ls` : Enumerate all names in *Current Working Directory* (context)

Basic *Directory* Operations

Unix

Directories implemented in *Files*

- Use *File* operations to create *Directories*
- C-library provides a higher-level abstraction:

```
opendir(name)
readdir(DIR*)
seekdir(DIR*)
closedir(DIR*)
```

Windows

Explicit *Directory* operations

```
CreateDirectory(name)
RemoveDirectory(name)
```

Very different method for reading *Directory Entries*

```
FindFirstFile(pattern)
FindNextFile()
```

Default Context: *Working Directory*

- Cumbersome to always have to specify full *Pathnames*
  - In Unix, each process has a "*Current Working Directory*" (**`cwd`**)
  - *Filenames* not beginning with **`/`** are assumed to be relative to the **`cwd`**
    - Otherwise translation happens as before

- A Shell also tracks a default list of active contexts
  - A "*Search Path*" for the Programs you are running
    - (**`:`** -separated list of *Pathnames*)
  - Given a *Search Path* **`A:B:C`** , the Shell will check in **`A`**, then **`B`**, then **`C`**
  - Can get around this by using explicit paths, e.g: **`./foo`**

- Example of *Locality*

Hardlinks

More than one *Directory Entries* can refer to a given *File*

- *"Hardlink"* creates a synonym name for a *File* (i.e. creates another *Directory Entry* for it)
- Unix stores count of references (*Hardlinks*) to the *inode* (of that *File*) itself
- If one of the *Hardlinks* is removed (e.g. using **`rm`**), the data are still accessible through any other *Hardlink* that remains
- If all *Hardlinks* are removed, then the space of that *File*'s Data is considered as freed

Existing *File* *Hardlink* to create

```
ln foo bar
```

```
foo      inode #31279
bar      links_count=2
```

Softlinks / Symlinks

Softlinks / *Sym*(bolic)*links* : Synonyms for *Filenames*

- Point to a *File*(/*Dir*)*name*, but object can be deleted from underneath it
  - i.e. does not even have to exist
- Unix implements these like *Directories*:
  - *inode* itself has special *Symlink* bit set, and its Data contain name of *Symlink* target
- When the *Filesystem* encounters a *Symlink inode*, it automatically translates it

Existing *File* *Hardlink* to create

`ln foo bar`

foo
bar

inode #31279
links_count=2

Existing *File* *Symlink* to create

`ln -s bar barz`

barz

inode #35382
links_count=1

File Data
"bar"

File Sharing

- *File* Sharing has been around since Time-sharing
  - Easy to do on a single machine
  - PCs, workstations, and networks get us there (mostly)
- *File* Sharing is important for getting work done
  - Basis for Communication and Synchronization
- 2 key issues when sharing *Files*:
  - Semantics of *Concurrent Access*
    - What happens when one *Process* reads while another writes?
    - What happens when two *Processes* open a *File* for writing?
    - What should be used to coordinate these?
  - *Protection*

Protection

- *Filesystems* implement a *Protection* system
  - Who can access a *File*
  - How they are allowed to access it (read/write/etc.)

- A *Protection* system dictates whether a given *Action* performed by a given *Subject* on a given *Object* should be allowed
  - *Examples:*
    - Your *User* account can read and/or write your own *Files*, but other *Users* cannot
    - You can read **`/etc/motd`** , but you cannot write to it

Protection Representation

Access Control Lists (ACL)

- For each *Object*, maintain a list of *Subjects* and their permitted *Actions*

Capabilities

- For each *Subject*, maintain a list of *Objects* and the permitted *Actions*

Objects

Subjects

	/one	/two	/three
root	rw	rw	rw
alice	rw	-	r
bob	w	r	rw

Capability

ACL

ACLs and *Capabilities*

- *Capabilities* are easier to transfer
  - "A *Capability* is a **token**, **ticket**, or **key** that gives the possessor permission to access an entity or *Object* in a computer system" (Dennis and Van Horn, 1966)
- *ACLs* are easier to manage
  - *Object*-centric, easy to grant, revoke
  - To revoke *Capabilities*, have to keep track of all *Subjects* that have the *Capability*
    - A challenging problem
- *ACLs* have a problem when *Objects* are heavily shared (i.e. many *Subjects (/Users)*)
  - The *ACLs* become very large
  - Mitigation:
    - Use *Subject (/User) Groups* (e.g. in Unix)

Unix *File Protection*

Unix uses both *Protection* approaches in its *Filesystem*

- ACLs:
  https://linux.die.net/man/5/acl
  - Example: Unix *File Permissions*
    - View *ACL:* `getfacl <filename>`

- Capabilities:
  - Example: *File Descriptors*

Capability — *ACL* check, expensive

```
int fd = open("file.txt", O_WRONLY);
if (fd == -1)
  exit(-1);

for (int i = 0; i < 100; i++)
  write(fd, buf + i * 4, 4);
```

Use *Capability* from now on

- How are they used together?
  - e.g. making an **`open()`** *System Call* which gets us a *File Descriptor*

Filesystem Implementation

- A *File Descriptor* is an application of a *Capability*
- When a *File* is `open()`ed, a *File Descriptor* is created and stored in the "**`filp`** *Table*"
  - **Each** *Process* has a **`filp`** *Table*, which is stored in *Kernel-Space*
  - The *User-Space* Application is given access to the index of the *File Descriptor*
    - We often call this index itself the *File Descriptor*, but it is actually just an index to the **`filp`** *Table*
  - The **`filp`** *Table* is actually a *Capability* list (contains a list of *File Descriptors*)
    - Each *File Descriptor* contains a *Permission* part that describes what the *Process* can do to this file; the *File Descriptor* also contains an identifier, which is the address of the file's *inode*

Note: A *File Descriptor* in the *Process'* **`filp`** *Table* is a *Capability* that allows the *Process* to access the *File* in a specific way, **even after a change in the *ACL*** of the *File*

- (i.e. the "token / ticket / key" is not revoked)

Filesystems vs *Virtual Memory*

	Disk	MLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	100 MB/s	550–2500 MB/s	> 1 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	> 1 GB/s
Cost	$0.03/GB	$0.35/GB	$6/GiB
Persistence	Non-volatile	Non-volatile	Volatile

*: Flash write performance degrades over time

Filesystems vs *Virtual Memory*

Disk review:

- Disk reads/writes in terms of *Sectors*, not Bytes
  - Read/write single *Sector* or adjacent groups

"*Read-Modify-Write*"

- How is a single Byte written
  - Read in *Sector* containing the Byte
  - Modify that Byte
  - Write entire *Sector* back to Disk

Sector = Unit of *Atomicity*

- *Sector* Write will be done completely, even if a crash happens in middle
  - Disk saves up enough momentum to guarantee its completion
- Larger *Atomic* units have to be *Synchronized* by OS

Filesystems **vs** ***Virtual Memory***

- Trends:
- Disk Bandwidth and cost-per-bit improving exponentially
  - Similar to CPU speed, Memory size, etc.
- *Seek Time* and *Rotational Delay* improving very slowly
  - Require mechanical motion (e.g. disk arm)
- Disk access is a huge system bottleneck & getting worse
  - Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as a smaller chunk
  - Trade Bandwidth for Latency if you can get lots of related stuff
- Desktop *Memory* size increasing faster than typical workloads
  - More and more of workload fits in *File Cache*
  - Disk traffic changes: Mostly writes and new data
- *Memory* and CPU resources increasing
  - Use *Memory* and CPU to make better decisions
  - Complex prefetching to support more I/O patterns
  - Delay in Data Placement decisions reduces random I/O

Filesystems vs *Virtual Memory*

- Goal: Operations should have as few Disk Accesses as possible & at minimal Space overhead
  - i.e. by grouping related things
- What's hard about grouping *Blocks*?
  - Remember: *Virtual Memory Pages* and *Process Locality*
- Like *Page Tables*, the *Filesystem* Metadata construct mappings
    - e.g. *Page Table*: Provides mapping of *Virtual Page Number* to *Physical Page Number*
  - *File* Metadata:
    Provide mapping of *Byte Offset* to *Disk Block* Address
  - *Directory* Data (i.e. List of *Directory Entries*):
    Provide mapping of Name to *inode # (i-number)*

Filesystems vs *Virtual Memory*

- In both *Filesystems* and *Virtual Memory* management, want location transparency
  - Application shouldn't care about particular Disk *Blocks* or *Physical Memory* locations
- In some ways, *Filesystems* have an easier job than *Virtual Memory*:
  - CPU time to do *Filesystem* mappings not a big deal
  - No *Virtual Address Space* invalidation on *Context Switch*, TLB *Misses*, etc.
  - *Page Tables* deal with sparse *Address Spaces* and random access
  - *Files* often denser (`[0 … filesize – 1]` ~ *Sequentially* Accessed)
- In some ways, a *Filesystem*'s problem is harder:
  - Each layer of translation = potential Disk Access
  - Space a huge premium! (But Disk can be huge)
    - Disk *Cache* Space never enough; amount of Data you can get in a single fetch never enough
  - Range very extreme: Many files < 10 KB, some files GB

Filesystems vs *Virtual Memory*

- Some Working Intuitions:
- *Filesystem* performance dominated by # of Disk Accesses
  - If each Disk Access costs ~10 milliseconds, touching the Disk 100 times = 1 second
  - Can perform a billion ALU operations in same time
- Disk Access cost dominated by Mechanical Motion, not Transfer:
  - 1 *Sector*: $5ms + 4ms + 5\mu s$ ($\approx 512\ B/(100\ MB/s)$) $\approx 9ms$
  - 50 *Sectors*: $5ms + 4ms + .25ms = 9.25ms$
  - Can get 50x the data for only ~3% more overhead
- Important Observations:
  - All *Blocks* in *File* tend to be used together, sequentially
  - All *Files* in a *Directory* tend to be used together
  - All *Names* in a *Directory* tend to be used together

Problem: How to Track a *File*'s Data

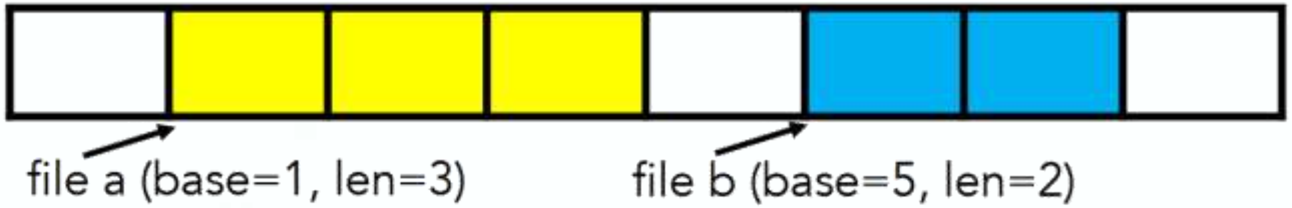
- Disk management:
  - Need to keep track of where *File* contents are on Disk
  - Must be able to use this to map *Byte Offset* to *Disk Block* Address

The "*Index Node*" –or– "*inode*"
- A structure that tracks a *File*'s Disk *Sectors*
- *inodes* must be stored on Disk too (Remember: *Persistence*)

- Things to keep in mind while designing *File* structure:
  - Most *Files* are small
  - Much of the Disk is allocated to large *Files*
  - Many of the I/O operations are made to large *Files*
  - Want good *Sequential* and good *Random* Access
    - Each imposes its own requirements

First Idea: Contiguous Allocation

- "*Extent-based*": Allocate *Files* similarly to *Virtual Memory Segmentation* model
  - When creating a *File*, do "*Allocate-on-Flush*"
  - *inode* contents: *Location*, *Size*



Fragmentation:
What if *File* **c** needs 2 *Sectors*

Example: IBM OS/360

Advantages

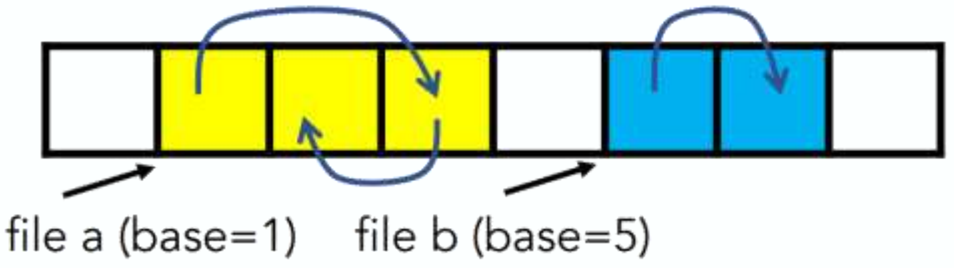
- Simple, fast Access, both *Sequential* and *Random*

Disadvantages (think of corresponding *Virtual Memory Segmentation* scheme)

- *Files* may not dynamically grow after creation
- *External Fragmentation*

Better Idea: *Linked Files*

A single *File*'s *Clusters* are "*Fragmented*" throughout the Disk



- Basically a Singly-Linked List on Disk
  - Keep a Singly-Linked List of all *Free Blocks*
  - *inode* contents: a pointer to *File*'s first *Block*
  - In each *Block*, keep a pointer to the next one
- *Examples* (sort-of): Alto, TOPS-10, DOS FAT

Advantages
- Easy dynamic growth & *Sequential* Access, no *External Fragmentation*

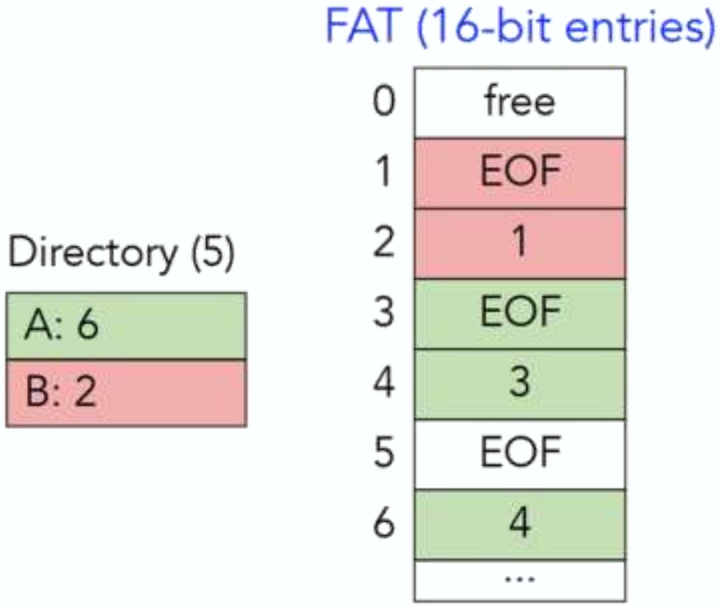
Disadvantages
- Linked Lists on Disk a bad idea because of Access Times
- *Random* Access very slow (e.g. have to traverse whole *File* to Access last *Block*)
- Pointers necessary in every *Block*, skewing Data alignment

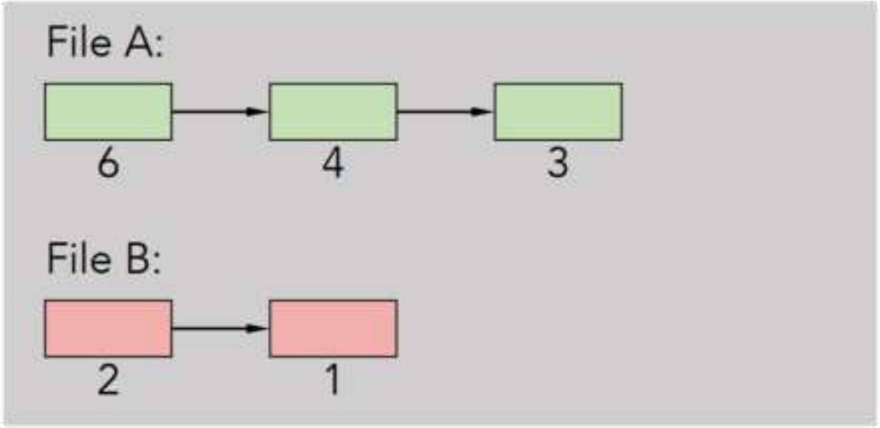
Filesystems

File Allocation Table **(FAT)** – *Example* : **DOS *Filesystem***

Linked Files with key optimization:

- Links are placed in fixed-size "*File Allocation Table*" (FAT), rather than individually inside each *Block*
- Still does Pointer chasing, but can cache entire FAT in *Memory* (cheaper than Disk Access)





FAT Discussion

Entry size = 16 bits (initial FAT16 in MS-DOS 3.0)

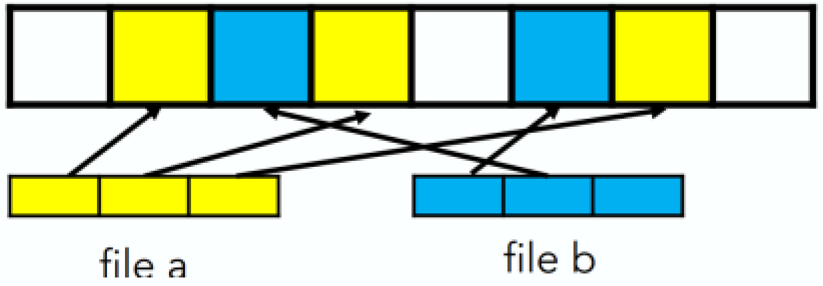
- Maximum size of the FAT: $2^4 = 65{,}536$ *Entries*
- Given a 512 Byte *Block*, maximum size of *Filesystem*: $2^4 * 512$ B $= 32$ MiB
- One solution: Go to bigger *Blocks* (Pros? Cons?)

- Space overhead of FAT itself is trivial
  - 2 Bytes (16-bit Pointer size) / 512 Byte *Block* = ~ 0.4%
- *Reliability*: How to protect against errors?
  - Create duplicate copies of FAT on the Disk
  - State duplication a very common theme in *Reliability*
- Bootstrapping: Where is Root *Directory*?
  - Fixed *Location* on Disk: The special "*Root Directory Region*"
  - FAT has: 1) *Reserved Sectors*, 2) *FAT Region* (contains FAT #1 & FAT #2 (optional *Reliability* copy)
    3) *Root Directory Region*, 4) *Data Region*

FAT	FAT (opt)	Root dir	…

Another Idea: *Indexed Files*

- Have an *Array* that holds all of a File's *Block* Pointers
  - Similarly to a *Page Table*, so will have similar issues
  - Max *File Size* determined by *Array*'s size
    - **Static or Dynamic?**
  - Allocate *Array* to hold *File*'s *Block* Pointers on *File* creation
  - Allocate actual *Blocks* on demand using *Freelist* of the Disk's *Blocks*



Advantages

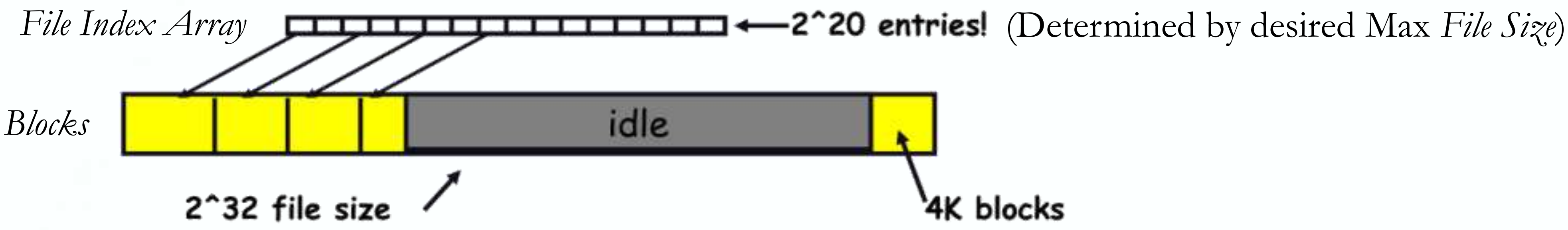
- Both *Sequential* and *Random* Access easy

Disadvantages

- *Mapping Table* (for all *File Arrays*) requires large chunk of **Contiguous Space**
- Same problem we were trying to solve initially

Indexed Files **Discussion**

- Issues similar as with *Page Tables*
  - Facilitate large Max *File Size* → Suffer lots of unused *Entries* (per *File Index Array*)
  - *Mapping Table* (all *File Index Arrays*) requires large chunk of **Contiguous Space**



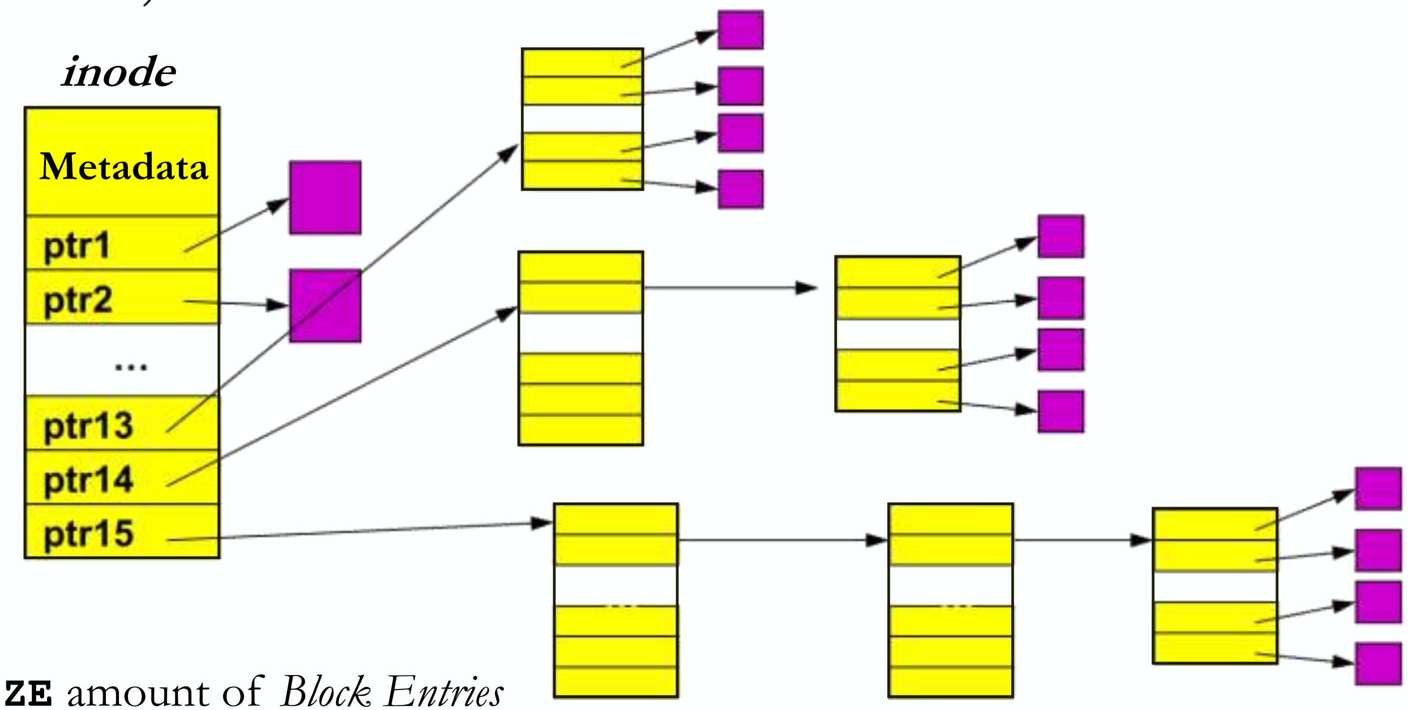
- Solve similarly: Add Indirection Layers & use *Multi-Level Index Arrays*

Multi-Level Indexed Files*: Unix *inodes

Unix *inode*:

- *inode* = 15 *Block* Pointers (+ *Metadata*)

- First 12 are *Direct Blocks*
  - Giving faster Access performance for initial *Blocks*
- Then 1 *Single*-indirect *Block*,
  1 *Double*-indirect *Block*,
  1 *Triple*-indirect *Block*



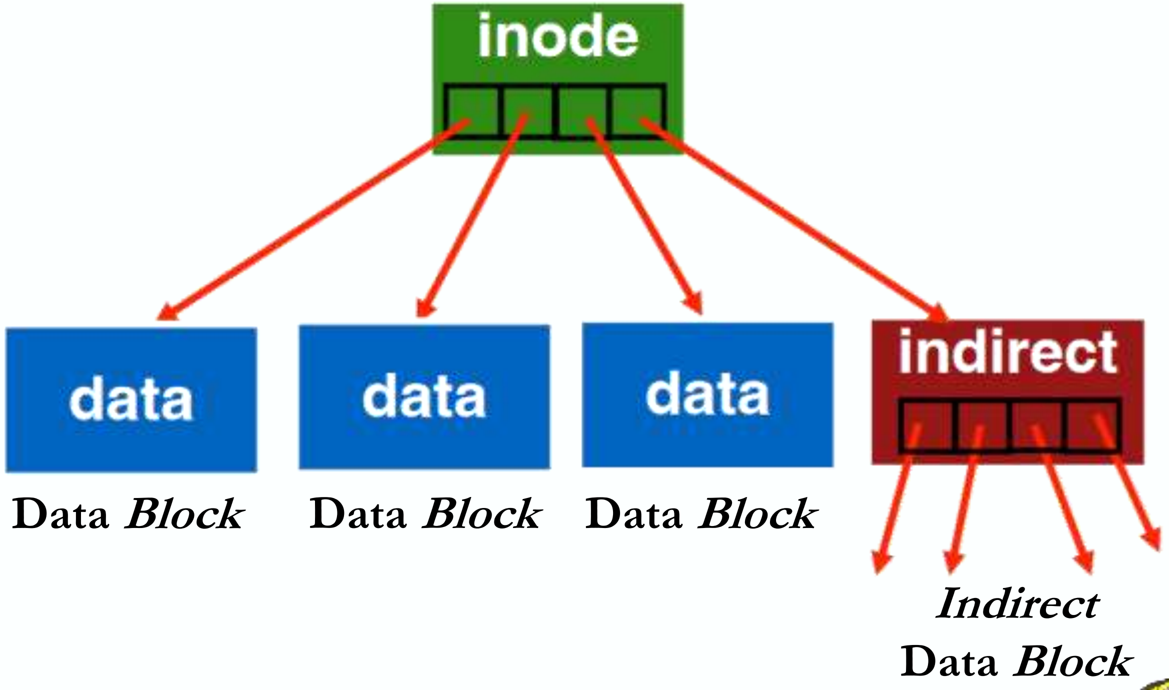
Note:
***Indirect Blocks*:** Contain **`BLKSIZE/PTRSIZE`** amount of *Block Entries*

More about *inodes*

Note: Information about **`file_type`** (*File*/*Directory*/*Symlink*/*Character Device*/etc.) stored in corresponding *Directory Entry* that points to this *inode*, e.g.: **`<inode, file_type, name_len, name>`**

`struct inode`

`uid` (*Owner*)
`gid` (*Group*)
`rwx` (*Permissions*)
`size` (of *File* in Bytes)
`blocks` (# *Blocks* of this *File* – whether used or not)
`atime` (*Access* Time)
`mtime` (*Modification* Time)
`ctime` (*Creation* Time)
`dtime` (*Deletion* Time)
`links_count` (*Hardlinks* – # of *Paths*)
`block[15]` (12+1+1+1 Data *Blocks*)



More about *inodes*

The "*inode Table*" : Array that stores *inodes* (one *inode* for every *File*)

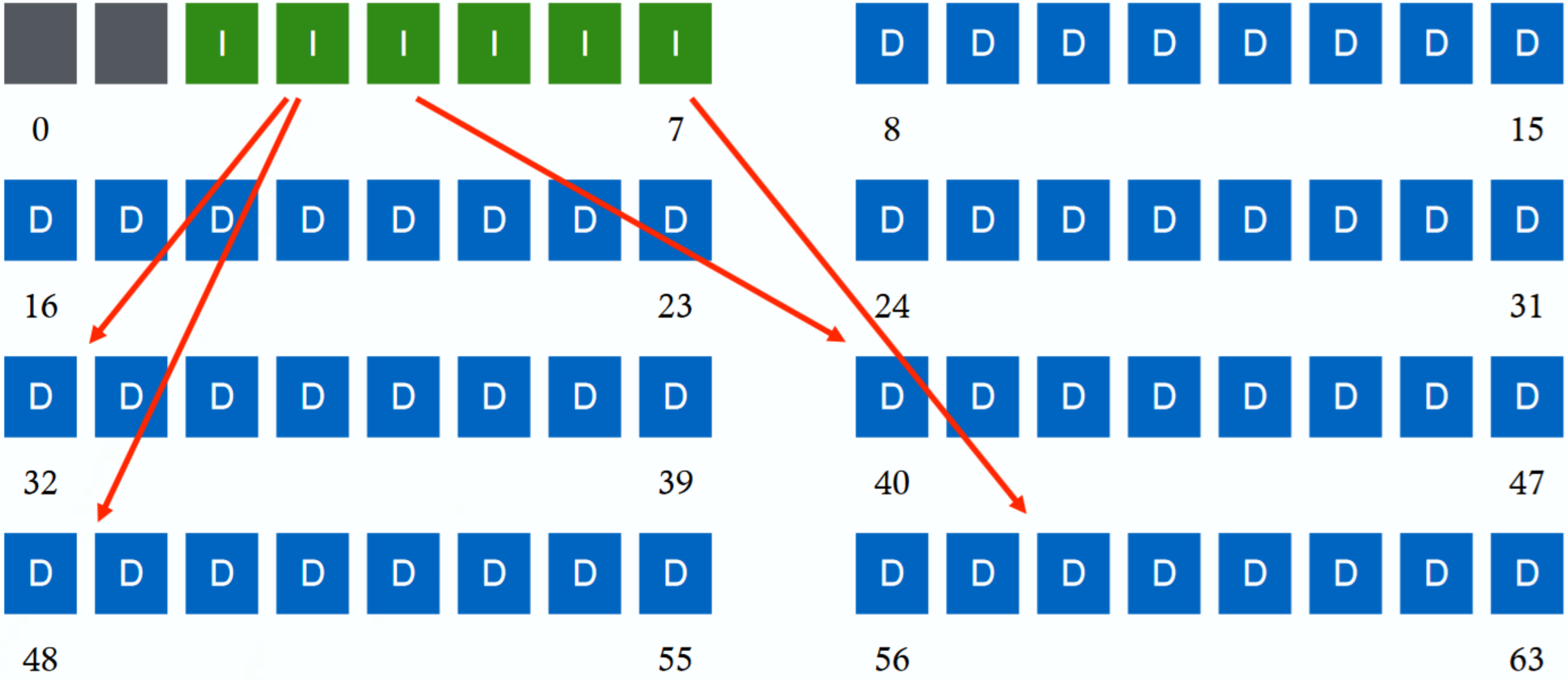
- (Similar functionality to the *Mapping Table* for all *Files* on the *Filesystem*)

- *inode Table*: Fixed-size (can support max # of *inodes*) when *Disk* is initialized; can't be changed
- Lives in known *Location*, originally at outer edge of Disk
  - This resulted in long seeks between *inodes* and *File* Data, improved by having many small *inode Tables* spread across the Disk

- The index of an *inode* in the *inode Table* called the "*i-number*"
  - Internally, the OS refers to *Files* by *i-number*
  - When *File* is opened, its *inode* is brought in to *Memory*
  - Written back to Disk when modified and *File* closed, or timeout elapses

More about *inodes*

Example: inode Table located at start of Disk



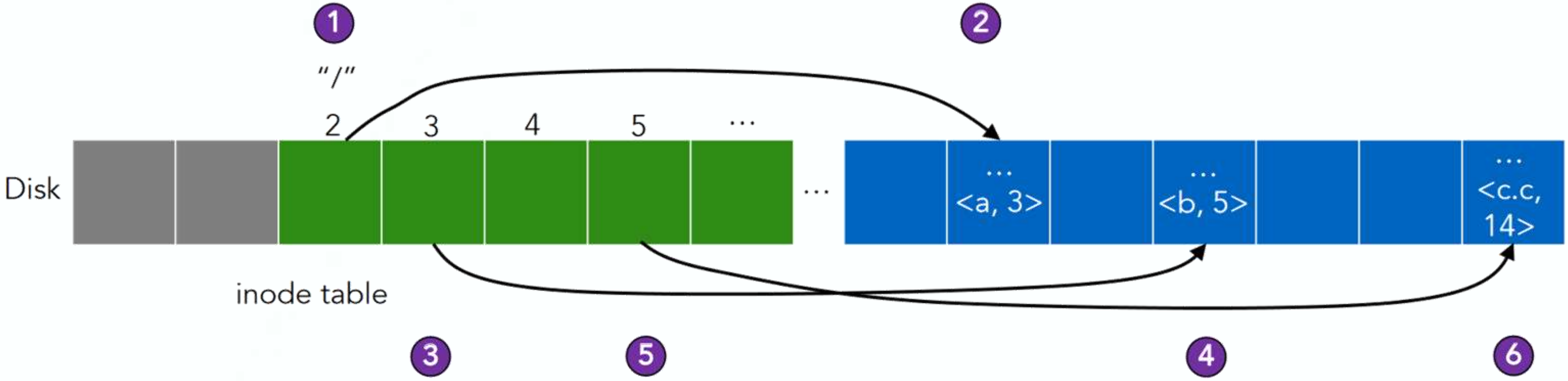
Remember: **Unix *inodes* and *Path* Search**

- Unix *inodes* are not *Directories*
  - An *inode* describes where on the Disk the Data *Blocks* for a *File* are located
  - A *Directory* is like a *File*, so its *inode* just describes the location of its Data *Blocks*
    - Remember: Its Data is a List of *Directory Entries*

- *Directory Entries* map *Filenames* to *inodes*
  - To open **`/one`**, read *inode* #2 (**`"/"`**) from *inode Table* on Disk into *Memory*
  - Read Data *(Directory Entries)* of **`"/"`** from Disk into *Memory*, find *Entry* for **`"one"`**
  - This *Entry* gives the *inode* # for **`"one"`**
  - From the *inode Table* again, find the *inode* for **`"one"`**
  - The *inode* says where the Data *Blocks* of **`"one"`** are on Disk
  - Read its first *Block* into *Memory* to access the initial Data stored by the *File*

Note: 4 Disk Accesses required

Unix Example: `/a/b/c.c`



- *inode #2* holds File for **`/`**
- *inode #3* holds File for **`a`**
- *inode #5* holds File for **`b`**
- *inode #14* holds File for **`c.c`**

File Buffer Cache

- Disk operations are slow, but Applications exhibit *Locality* for reading and writing *Files*
- Idea: Cache *File Blocks* into *Memory* to capture *Locality*
  - Called the "*File Buffer Cache*"
  - *File Buffer Cache* is system-wide, used and shared by all *Processes*
  - Reading from the *File Buffer Cache* helps Disk-based operations to perform more like *Memory*-based ones
  - Even a small *File Buffer Cache* can be very effective
- Issues
  - The *File Buffer Cache* competes with *Virtual Memory* for Space (tradeoff here)
  - Like with *Virtual Memory*, there are actual limitations (due to *Physical Memory*)
  - Need *Replacement Algorithms* again (LRU usually used)

Caching Writes

- OSes typically do "*Write-Back Caching*"
  - Maintain a queue of *Uncommitted Blocks*
  - Periodically flush the queue to Disk (30 second threshold)
  - If *Blocks* changed many times in 30 secs, only need one I/O
  - If *Blocks* deleted before 30 secs (e.g. `/tmp`), no I/Os needed
- Unreliable, but practical
  - On a crash, all writes within last 30 secs are lost
  - Modern OSes do this by default; too slow otherwise
- *System Calls* exist to enable Applications to force-write data to Disk
  - e.g. `fsync()`: Flush *OS Buffers* to Physical Media (Device)

Read-Ahead

- Many *Filesystems* implement "*Read-Ahead*"
  - *Filesystem* "predicts" that the *Process* will request a next *Block*
  - *Filesystem* requests it from the Disk ahead of time
  - This can happen while the *Process* is executing on the previous *Block*
    - Overlap I/O with execution
  - When the *Process* requests next *Block*, it will already be in the *Read-Ahead Cache*
  - Complements the *Disk (Hardware) Cache*
    - *Remember:* Disk is also is doing *Read-Ahead*

- For *Sequentially*-Accessed *Files* can be a big win
  - Unless *Blocks* for the *File* are scattered across the Disk
  - *Filesystems* try to prevent that though (through proper *File* allocation)

CS-446/646

Time for Questions !