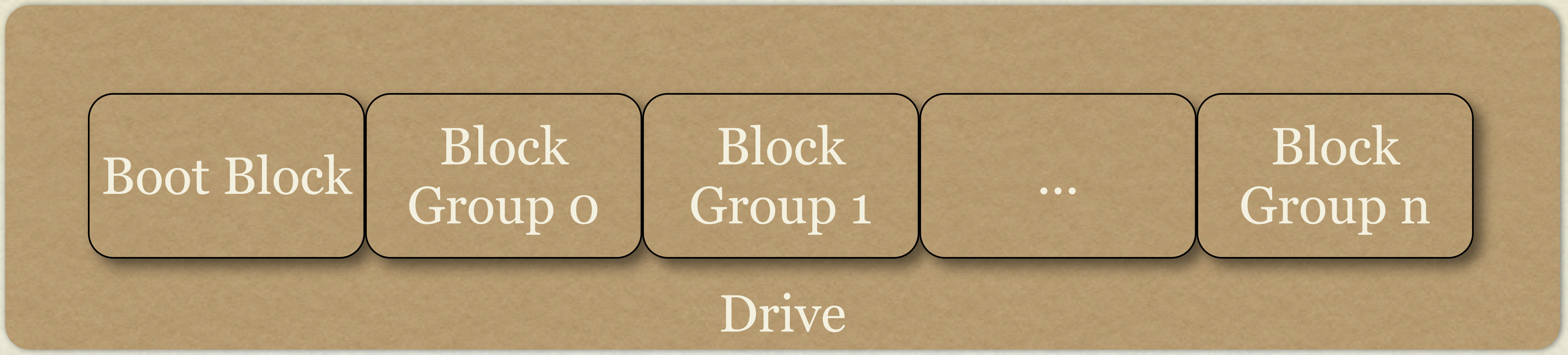
FILESYSTEMS

Mmmm crunchy

PURPOSE

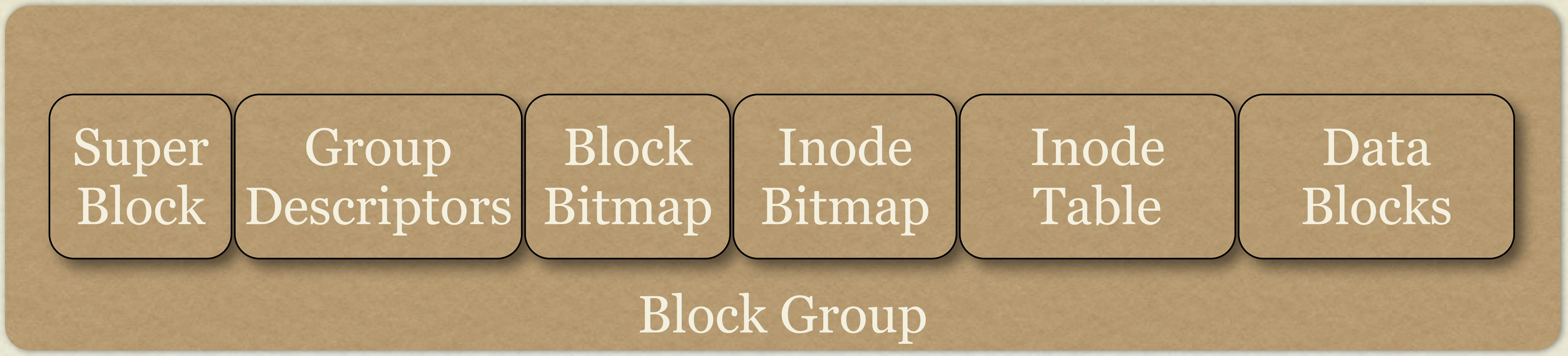
- So all this data...
- How to organize? Whose job?
- Filesystems!

OVERVIEW



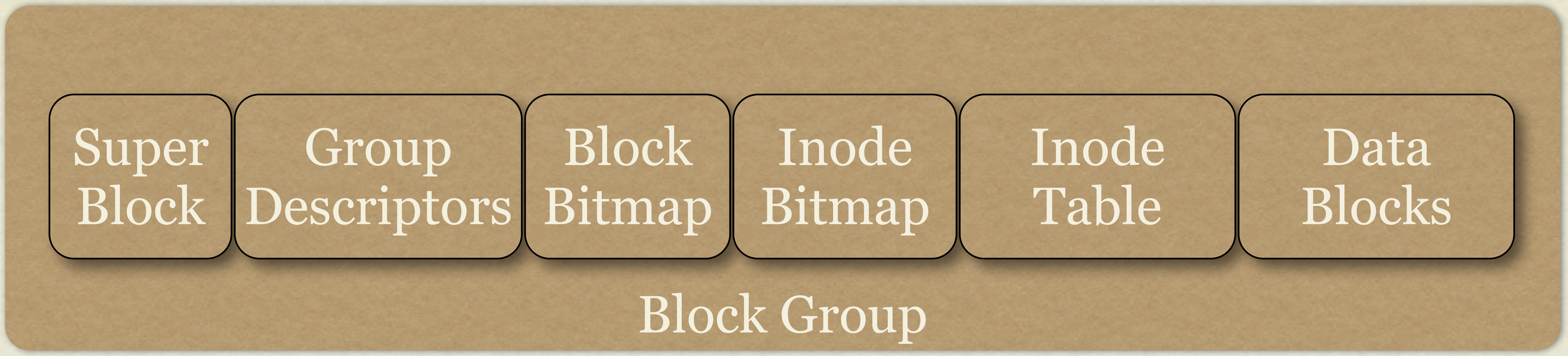
- On the physical drive, information is stored in blocks
- The first block is always the boot block
- The rest of the blocks are pooled and organized into block groups

BLOCK GROUPS



- Each block groups contains a copy of the super block and descriptions of all the block groups
- The superblock holds information on the entire filesystem
- Block and inode bitmaps provide fast lookup information on free and allocated blocks and inodes

BLOCK GROUPS



- The inode table holds all of the inodes (more on inodes in a minute!)
- The data blocks contain the actual *data* that is contained in the files on the filesystem

WOW, WHAT?

- Don’t worry - what’s important to understand is the inode and it’s relationship with data blocks.
- Superblocks, block groups, bitmaps and tables are important to know about, but their details are beyond this course

INODES

- Inodes, or Information Nodes, hold all of the meta information for a file (or directory! those are just special kinds of files!)
- Details about ownership, size, permissions, times, ACLs and more are stored in the inode.
- But most importantly, the inode points to data blocks which store the *contents* of the file.

WHAT ABOUT THE FILE NAME?

- Good question! You would think it would be stored in the inode, but it's not! That's where directories come in...
- A directory is a special type of file whose contents (in the data blocks!) is a list of name/inode pairs.
- There are many reasons to do it this way, including performance, simplicity and hard link capability

LET’S DIAGRAM THIS OUT

It’s easier to handle questions on the whiteboard ;)

ANY OTHER QUESTIONS?

Bueller? Bueller?

FILE TYPES

- So far, the presentation has covered regular files and directories. There are other file types:
  - Soft (symbolic) links
  - Named pipes and sockets
  - Device files (block and character)

PERMISSIONS

- Linux supports 3 main types of access on a file:
  - read: View the contents
  - write: Modify the contents and metadata
  - execute: “Run” the contents
- Actually, it’s slightly more complex because it’s different for files and directories…

PERMISSIONS

	Files	Directories
Read	View the contents	List contents
Write	Change the contents/ metadata	Create/delete entries, change metadata
Execute	"Run" the contents	Operate with directory as CWD

AWESOME... SO?

- Combining these permissions allows for the most common access levels:
  - Read only
  - Read/Write
  - Execute
  - etc
- Now to add a little more granularity, users and groups...

OWNERSHIP

- All files are associated with one user and one group. This creates the foundation for the main meat of the security infrastructure in the Linux (and Unix) operating system.
- When a process attempts an operation on a file, the user and group of the process (because every process is associated with one user and one group! surprise!) are compared with the user and group of the file, which determines what level of permissions is granted or denied on the file...

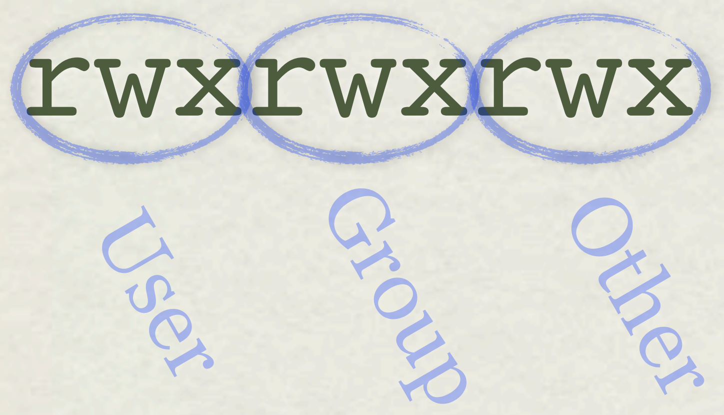
PUTTING IT ALL TOGETHER...

- Every file has 3 levels of permissions:
  - User
  - Group
  - Other
- When a process seeks access, the process user is compared to the file user - if they match, the process gets the User permissions. Next Group. If no match, Other level access

THE TRIPLE OF TRIPLES

- All of the permission information is neatly summarized with 9 characters:

  - `rwxrwxrwx`



- The presence of the letter indicates the permission is granted, a hyphen in it’s place indicates the permission is denied. Read only: `r--r--r--`

SPECIAL PERMISSIONS

- There are a few special permissions available:
  - Set User ID: Used on executables. When the file is “run”, it runs as the user that owns the file.
  - Set Group ID: Same as SetUID, but for the group.
  - Sticky Bit: Interesting story about the name and history, but nowadays, used on group/other writable directories to protect contents of directory by limiting write ability to only be allowed if accessing user matches user on file.

SPECIAL PERMISSIONS

- `ls` uses a simple format to display the special permissions:
  - SetUID: `rw`**`s`**`rwxrwx`
  - SetGID: `rwxrw`**`s`**`rwx`
  - Sticky: `rwxrwxrw`**`t`**
- Note that a lowercase letter is used if the underlying execute bit is set, otherwise it will be an uppercase letter
  - SetUID without execute set for user: `rwSrwxrwx`

CHANGING OWNERSHIP

- Two commands are available for changing the ownership of a file:
  - `chown`: Change Owner - changes the user owner of a file
    - `chown bob memo.txt`
  - `chgrp`: Change Group - changes group owner of file
    - `chgrp mgmt memo.txt`

CHOWN IT UP

- `chown` can actually change the group owner as well, so you don't need to bother messing with `chgrp`
    - `chown :mgmt memo.txt`
- You can do both at once, in fact!
    - `chown bob:mgmt memo.txt`

CHANGING PERMISSIONS

- Changing permissions is slightly more involved. The command is `chmod` (change mode)
- There are two basic ways to represent the permissions:
  - human friendly
  - octal

HUMAN FRIENDLY CHMOD

- When using human friendly permission specification, you just need to specify what *level* permission you want to change, *how* you want to change it, and *what* the permissions are..
- A table will clear up the mud...

HUMAN FRIENDLY CHMOD

	Who?	How?	What?
Symbols	u, g, o	+, -, =	r, w, x, s, t
Explanation	user, group, other	add, subtract, set	read, write, execute, set id, sticky

SO...

- Examples:
    - `chmod u+x file`
    - `chmod go-r file`
    - `chmod u=rw,go= file`
- Yes, you can combine “equations” to make different changes by separating them with commas, as in the last example

OCTAL?

- Octal refer to a *base* for a *numbering system*. Namely, base 8. Humans think and count in base 10, decimal. Computers work in base 2 (binary) and sometimes base 16 (hexadecimal). Octal is just another one, useful for permissions
- Short of a long, grueling discussion of numbering systems, you're going to have to just do some memorization here...

OCTAL!

Octal	Binary	Permissions
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

OCTAL

- Each octal digit fully represents all three primary permissions, so to specify all the basic permission levels for a file, all you need are 3 octal digits (user, group, other)!
  - `chmod 777 file`
  - `chmod 755 file`
  - `chmod 644 file`
  - `chmod 000 file`

EXERCISES

- Add write permissions for everyone to ‘file1’. Change the owner to ‘user’ and the group to ‘user’. (It won’t change, but if you did it right you won’t get an error message)
- Explain the following permissions: rw-r-----
- What’s special about inode #2?
- What is an inode?

LINKS

- Linux filesystems support two types of links, hard and soft
- Soft links are the easiest to understand, and have cousins in most operating systems, which makes them familiar
- After discussing soft links, we will tackle hard links

SOFT LINKS

- A soft (or symbolic) link is like a shortcut in windows: it’s a file that simply “points” to another file.
- In Linux, the pathname “pointed to” (source) is stored in the data blocks of the soft link (target)
- A soft link is an actual file, consuming an inode and using data blocks to store whatever pathname it’s pointing to

SOFT LINKS

- To create a soft link, use the `ln` command with the -s option:
  - `ln -s memo.txt link-to-memo.txt`
- In this example, `memo.txt` is the <u>source</u> and `link-to-memo.txt` is the <u>target</u>
- This command **creates a new file**, `link-to-memo.txt`, of type <u>link</u>, which points to `memo.txt`

SOFT LINK TRIVIA

- Since soft links merely store a pathname (absolute or relative), they can link to anything, anywhere. Local filesystem, other filesystems, network filesystems, removable media filesystems. They can even point to invalid pathnames! The kernel cares not!
- Removing a soft link does not remove the file pointed to, only the link file.
- Soft links do not have permissions themselves (no need!)

HARD LINKS

- With the foundation formed from the first dozen slides of this lecture, understanding hard links should not be difficult. Just a new concept to wrangle.
- A hard link is simply one of the name/inode pairs in a directory. Though when we think about *link*, we think of another access point to the file.
- Technically, all files are hard linked - via the directories.
- By default, there is only one of these links...

HARD LINK TRIVIA

- When a new hard link is created, it simply adds another reference (filename) in a directory to that inode (file)
- Removing a hard link does not remove the file unless it was the only hard link to that inode
- Hard links, due to their nature with inodes and directories, only operate within a filesystem - you can not create a hard link from one filesystem to another
- How do permissions work?

EXERCISES

- In your home directory, create a soft link to 'file1'. Verify the link by cat-ing the contents out. Compare the inode numbers.
- In 'test', create a hard link to 'file1'. Verify the link by cat-ing the contents out and also compare inode numbers.
- Why would you use a hard link instead of a soft link?
- Which type of link can point across filesystems?

EDITING FILES

- Time for a Nerd Holy War
- Editor of choice, anyone? (TUI only - if anyone throws down with a GUI editor, you've failed the class already!)
- In my opinion, `vi` (or `vim`) wins   =)
- `emacs` is great, powerful and fast, but it's just not *common* enough. Plus, the control-x madness is, well, madness! ;)

VI DEMONSTRATION

Emacs users, bite your tongues!

```
slideshow.end();
```