CS111ACE Week 9
Directories, Links, and Crash Recovery
Recap

Last class, we were given a piece of hardware and an interface…

read_sector(...);
write_sector(...);
Recap

Last class, we were given a piece of hardware and an interface…

We were able to organize sectors on disk in a few different ways:

```
read_sector(...);
write_sector(...);
```
Recap

Last class, we were given a piece of hardware and an interface…

We were able to organize sectors on disk in a few different ways:

```plaintext
read_sector(...);
write_sector(...);
```

Linked files
Recap

Last class, we were given a piece of hardware and an interface…

We were able to organize sectors on disk in a few different ways:

```cpp
read_sector(...);
write_sector(...);
```
Recap

Last class, we were given a piece of hardware and an interface...

We were able to organize sectors on disk in a few different ways:

```c
read_sector(...);
write_sector(...);
```
Recap

Last class, we were given a piece of hardware and an interface...

At the heart of these systems was a distinction between metadata and payload data.

read_sector(...);
write_sector(...);

For file systems we’ll call this an inode.
Directories

We also need our file system to support directories -> files that store other files (which most people call folders)
Directories

We also need our file system to support directories -> files that store other files (which most people call folders)

We use inodes to represent directories as well (all that differs is the payload)

- We can inspect the i_mode field to tell whether a inode refers to a directory or a normal file.
Directories

We also need our file system to support directories - files that store other files (which most people call folders).

We use inodes to represent directories as well (all that differs is the payload):

- We can inspect the i_mode field to tell whether a inode refers to a directory or a normal file.

Directory payloads store directory entries, a special struct that stores a file’s name and its inumber. Directory entries are 16 bytes, meaning you can have $\frac{512}{16} = 32$ per block.
Directories

Directories are ordered in a tree-like structure, as seen below:

```
/  
|   |
bin Users mnt
|   |
|   |
ls cd pwd
|   |
Shared andrew ANDY_FMRI
|   |
Desktop Applications Downloads
|   |
AndyFolder ExperimentFolder
|   |
myProgram.c sand.py
|   |
```

root directory

scan.pdf

img3015.jpg

minecraft.exe
Directories

Directories are ordered in a tree-like structure, as seen below:

The “root” of the tree is a single directory called the “root” directory that is always described by inode 1. Its name is “/”

myProgram.c
sand.py
minecraft.exe
scan.pdf
img3015.jpg
Directories

Directories are ordered in a tree-like structure, as seen below:

A **path** is the sequence of directories you visit (separated by “/”, starting from root, to get to a specific file (leaf node)).

What is the path for `cd`?
Directories

Directories are ordered in a tree-like structure, as seen below:

A path is the sequence of directories you visit (separated by “/”, starting from root, to get to a specific file (leaf node).

What is the path for cd?

/bin/cd

Ls cd pwd

Shared andrew ANDY_FMRI

Desktop Applications Downloads

AndyFolder ExperimentFolder

myProgram.c sand.py

minecraft.exe scan.pdf img3015.jpg
Directories

How does a file system actually retrieve a file given a path? Let’s use /bin/cd as our example:
If we have time - directories

How does a file system actually retrieve a file given a path? Let’s use `/bin/cd` as our example:

1. Read the sector into memory that contains inode 1 (root directory inode). We **always** know where this block is.
If we have time - directories

How does a file system actually retrieve a file given a path? Let’s use /bin/cd as our example:

1. Read the sector into memory that contains inode 1 (root directory inode). We **always** know where this block is.
2. Use the `i_addr` field to locate the block numbers of the directory payload. Recall that this might require indirect addressing!
   a. Read each sector into memory until you locate the directory entry whose name is “bin”
If we have time - directories

How does a file system actually retrieve a file given a path? Let’s use `/bin/cd` as our example:

1. Read the sector into memory that contains inode 1 (root directory inode). We always know where this block is.
2. Use the `i_addr` field to locate the block numbers of the directory payload. Recall that this might require indirect addressing!
   a. Read each sector into memory until you locate the directory entry whose name is “bin”
3. Read the inumber associated with “bin”. Then, look at its `i_addr` field to find its payload block numbers. Read in payload blocks until you find the directory entry named “cd”. Now you have the inumber for `cd` and can access its contents!

```
"bin" : 2
"User" : 3
"mnt" : 4
"ls" : 5
"cd." : 6
"pwd" : 7
```
Another worked example

Recall the following example from lecture below. Assuming **no** blocks are already in memory, how many sectors do we need to read in if we want to obtain the **very first** payload block of "largefile"? (Include the first payload block)
## Links (briefly)

Consider the following directory payloads on the same disk:

<table>
<thead>
<tr>
<th>/home/trip</th>
<th>/home/john</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ “Desktop”, 23 }</td>
<td>{ “unixV6”, 75 }</td>
</tr>
<tr>
<td>{ “Minecraft.exe”, 150 }</td>
<td>{ “cdrom”, 363 }</td>
</tr>
<tr>
<td>{ “Downloads”, 25 }</td>
<td>{ “etc”, 4 }</td>
</tr>
<tr>
<td>{ “Pictures”, 24 }</td>
<td>{ “bin”, 5 }</td>
</tr>
<tr>
<td>{ “CS111.c”, 1304 }</td>
<td></td>
</tr>
</tbody>
</table>
Links (briefly)

Consider the following directory payloads on the same disk:
What would happen if /home/john added a file called “CS111_Link.c” that had the
same inumber as “CS111.c” in /home/trip ??

```
/home/trip
{ “Desktop”, 23 }
{ “Minecraft.exe”, 150 }
{ “Downloads”, 25 }
{ “Pictures”, 24 }
{ “CS111.c”, 1304 }
```

```
/home/john
{ “unixV6”, 75 }
{ “cdrom”, 363 }
{ “etc”, 4 }
{ “bin”, 5 }
```
Links (briefly)

Consider the following directory payloads on the same disk:
What would happen if `/home/john` added a file called “CS111_Link.c” that had the same inumber as “CS111.c” in `/home/trip` ??

```
/home/trip
{ “Desktop”, 23 }
{ “Minecraft.exe”, 150 }
{ “Downloads”, 25 }
{ “Pictures”, 24 }
{ “CS111.c”, 1304 }

/home/john
{ “unixV6”, 75 }
{ “cdrom”, 363 }
{ “etc”, 4 }
{ “bin”, 5 }
{ “CS111_Link.c”, 1304 }
```
Links (briefly)

Consider the following directory payloads on the same disk:
This is what we call a “hard link.” The idea here is that two separate “files” are actually referring to the same inumber, meaning that they refer to the same inode.

```
/home/trip
{ "Desktop", 23 }
{ "Minecraft.exe", 150 }
{ "Downloads", 25 }
{ "Pictures", 24 }
{ "CS111.c", 1304 }
```

```
/home/john
{ "unixV6", 75 }
{ "cdrom", 363 }
{ "etc", 4 }
{ "bin", 5 }
{ "CS111_Link.c", 1304 }
```
Links (briefly)

Consider the following directory payloads on the same disk:
This is what we call a **hard link**. The idea here is that two separate “files” are actually referring to the same inumber, meaning that they refer to the same inode.

/home/trip

{ “Desktop”, 23 }
{ “Minecraft.exe”, 150 }
{ “Downloads”, 25 }
{ “Pictures”, 24 }
{ “CS111.c”, 1304 }

/home/john

{ “unixV6”, 75 }
{ “cdrom”, 363 }
{ “etc”, 4 }
{ “bin”, 5 }
{ “CS111_Link.c”, 1304 }

If you delete a file that has a **hard link**, the file will only be cleared from disk once ALL instances of the inumber are deleted.

Directories, like “Desktop” cannot have hard links (duplicated inumbers). You can create loops in the tree structure of the directory system (this violates the tree-like property of directories)
Links (briefly)

Alternatively, you could create a file whose payload contents are just the path to another file. This soft or symbolic link (file) can be changed at any time.
Links (briefly)

Alternatively, you could create a file whose payload contents are just the path to another file. This **soft** or **symbolic** link (file) can be changed at any time.

```
ln -s repos/Win23_CS111A_S1/ my_link
```

Here, I make a **soft link** from an existing file, repos/Win23_CS111A_S1/, to the name **my_link**, which doesn’t exist. After this command, the file **my_link** is created and put in the current directory. Clicking on it takes me into repos/Win23_CS111A_S1.
Links (briefly)

Alternatively, you could create a file whose payload contents are just the path to another file. This **soft** or **symbolic (symlink)** link (file) can be changed at any time.

```
ln -s repos/Win23_CS111A_S1/ my_link
```

Here, I make a **soft link** from an existing file, repos/Win23_CS111A_S1/, to the name **my_link**, which doesn’t exist. After this command, the file **my_link** is created and put in the current directory. Clicking on it takes me into repos/Win23_CS111A_S1.

You can try calling **cat** on the link or opening it in a text editor, but the actual link encoding isn’t visible to the user.

Deleting a **symlink** doesn’t affect the real file, but if you delete the real file, the **symlink** will “break,” and clicking on it will cause an error.
Think-Pair-Share on Links

1. I create 1 **hard** link to “Minecraft.exe”. I then create 1 **soft** link to “Minecraft.exe.” How many new **inumbers** do I need to make these links?

2. Can you make a **soft** link to a **hard** link?
   a. Can you make a **soft** link to a **soft** link?
   b. Can you make a **hard** link to a **soft** link?

3. If I make a **soft** link and then *change its location* (i.e. put it in another directory), what will happen? Will it still work?
Quick Primer on Working Directory
Quick Primer on Working Directory
Lightning Review of Crash Recovery
Lightning Review

We have 3 different kinds of crash recovery systems…

- fsck
- Ordered Writes
- Write-Ahead Logging (“Journaling”)
Lightning Review

- fsck is a system that checks to see if we had a clean shutdown on startup.
  - If a special flag on the disk is not a ‘1’ (indicating a clean shutdown), we know that we crashed!
Lightning Review

- **fsck** is a system that checks to see if we had a clean shutdown on startup.
  - If a special flag on the disk is not a ‘1’ (indicating a clean shutdown), we know that we crashed!
- **fsck** will do a scan of your data/metadata on disk (inodes, free list, directories) and try and repair anything that looks sus.

**Sus things:**
- A block listed in an inode’s i_addr AND listed in the free list? 
- A block listed in two or more inode i_addr’s?
- An inode that doesn’t exist in any directories (but it’s reference count > 0)
Lightning Review

- **fsck** is a system that checks to see if we had a clean shutdown on startup.
  - If a special flag on the disk is not a ‘1’ (indicating a clean shutdown), we know that we crashed!
- **fsck** will do a scan of your data/metadata on disk (inodes, free list, directories) and try and repair anything that looks sus.
- **fsck cons**: it takes a *long* time to run, we can still lose information (i.e. lost and found), it can’t repair our filesystem if we lose important things, and it could move around block with sensitive information.

**Sus things:**
- A block listed in an inode’s *i_addr* AND listed in the free list?
- A block listed in two or more inode *i_addr’s*?
- An inode that doesn’t exist in any directories (but it’s reference count > 0)
Lightning Review

- Ordered Writes isn’t meant to *replace* fsck, it’s more of an additional technique.
Lightning Review

- Ordered Writes isn’t meant to replace `fsck`, it’s more of an additional technique.
- If we require writes to always happen in a previously defined order, it makes certain scenarios impossible...
Ordered Writes isn’t meant to replace fsck, it’s more of an additional technique.

If we require writes to always happen in a previously defined order, it makes certain scenarios impossible…

Impossible Scenarios…

- If we always remove from the free list before allocating the new inode, we can never have a block be free and in an inode.
- To generalize this:
  - Always initialize the thing itself before making a pointer to the thing
  - Nullify old pointers before reusing a block
  - Never remove the last reference to a file unless you’re deleting it.
Lightning Review

- Ordered Writes isn’t meant to replace `fsck`, it’s more of an additional technique.
- If we require writes to always happen in a previously defined order, it makes certain scenarios impossible…
- Ordered writes rely on a write-through block cache to keep disk up-to-date. This `really` reduces the efficacy of the block cache.
- You can also leak blocks, as we showed in case #1

Impossible Scenarios…
- If we always remove from the free list before allocating the new inode, we can never have a block be `free` and in an inode.
- To generalize this:
  - Always initialize the thing itself before making a pointer to the thing
  - Nullify old pointers before reusing a block
  - Never remove the last reference to a file unless you’re deleting it.

Performance vs Consistency
Lightning Review

- Write-Ahead Logging is an append-only log on disk. We simply write to the log before doing any metadata operations. The change must be on the log AND the log must be flushed to disk before the change is considered “complete”.
  - Key idea: so long as the logged operations are correct, we can replay this log when we crash to restore our system!

It is possible to log data as well as metadata, but doing so significantly increases the size of your log. We typically just log metadata.
Lightning Review

- Can I add this operation to the log?
  - add block 99421 to inode 862 as block index 93
Lightning Review

- Can I add this operation to the log?
  - patch 4 bytes at offset 324 in block 6159972 with the value 99421
Lightning Review

- Can I add this operation to the log?
  - append 4 bytes to block 712389

KACHOW

idempotent: An action can be performed many times without affecting the state of the system.

We need to make our log operations idempotent.
Over time the log can get really long. We use checkpoints in our log to identify the last consistent point in our file system!
Lightning Review

- Over time the log can get really long. We use **checkpoints** in our log to identify the **last consistent point** in our file system!
- Before a checkpoint can be created, **all pending log changes must be flushed to disk**.
Lightning Review

- Logging Pros:
  - Writing the log is append-only (sequential), which is the optimal disk access pattern
  - As long as you replay the log, you won't have any file system inconsistencies
  - We can delay writing the log to disk to get better performance (why?)
Lightning Review

- Logging Cons:
  - Although delaying the log write-back improves overall performance, it means that we have a larger window where we can lose data due to a power failure.
  - There is a tradeoff between performance, durability, and consistency.
Practice Problem!

- Consider the following *high level* file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y
Practice Problem!

- Consider the following high level file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y

Think-Pair-Share with the following questions in mind…

1. What steps does the file system have to do for each of these things? Write the steps out for each operation (accessing bitmap? creating inode? creating dirent? writing to blocks?)
2. Do the steps you thought about have a particular ordering? What kinds of things could happen if we don’t respect that ordering (for example, fsck with no ordered writes)?
Practice Problem!

- Consider the following *high level* file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y

Steps the FS takes (in logical / ordered writes / logging order):
- Find free blocks for the contents of X, Y, and Z
- Mark the bitmap entries for these blocks as “used”
- Make 3 inodes that point to X, Y, and Z’s blocks, respectively
- Add directory entries for X, Y, and Z to make sure they ‘exist’ in our file system somewhere
- FINALLY write to them

-------
- Mark bitmap entry for file Y as ‘free’
- Remove dirent for Y / references
- Remove inode entry for Y

-------
- Repeat the first group of steps for file A

If we use `fsck` with no write-ordering, a lot of bad things can happen…
- We can have 2 live inodes pointing to the same block(s)
Practice Problem!

- Consider the following *high level* file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y

Steps the FS takes (in logical / ordered writes / logging order):
- Find free blocks for the contents of X, Y, and Z
- Mark the bitmap entries for these blocks as “used”
- Make 3 inodes that point to X, Y, and Z's blocks, respectively
- Add directory entries for X, Y, and Z to make sure they ‘exist’ in our file system somewhere
- FINALLY write to them

-------
- Mark bitmap entry for file Y as ‘free’
- Remove dirent for Y / references
- Remove inode entry for Y (we might not zero out the block… why?)

-------
- Repeat the first group of steps for file A

If we use fsck with no write-ordering, a lot of bad things can happen…
- We can have 2 live inodes pointing to the same block(s)
Practice Problem!

- Consider the following *high level* file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y

Steps the FS takes (in logical / ordered writes / logging order):
- Find free blocks for the contents of X, Y, and Z
- Mark the bitmap entries for these blocks as “used”
- Make 3 inodes that point to X, Y, and Z’s blocks, respectively
- Add directory entries for X, Y, and Z to make sure they ‘exist’ in our file system somewhere
- FINALLY write to them

---
- Mark bitmap entry for file Y as ‘free’
- Remove dirent for Y / references
- Remove inode entry for Y (we might not zero out the block… why?)

---
- Repeat the first group of steps for file A

If we use `fsck` with no write-ordering, a lot of bad things can happen…
- We can have 2 live inodes pointing to the same block(s)
- We can have “leaked” blocks that are ‘used’ in the bitmap but don’t exist anywhere!
Practice Problem!

- Consider the following *high level* file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y

Steps the FS takes (in logical / ordered writes / logging order):
- Find free blocks for the contents of X, Y, and Z
- Mark the bitmap entries for these blocks as “used”
- Make 3 inodes that point to X, Y, and Z’s blocks, respectively
- Add directory entries for X, Y, and Z to make sure they ‘exist’ in our file system somewhere
- FINALLY write to them

---
- Mark bitmap entry for file Y as ‘free’
- Remove dirent for Y / references
- Remove inode entry for Y (we might not zero out the block… why?)

---
- Repeat the first group of steps for file A

If we use `fsck` with no write-ordering, a lot of bad things can happen…
- We can have 2 live inodes pointing to the same block(s)
- We can have “leaked” blocks that are ‘used’ in the bitmap but don’t exist anywhere!
Practice Problem!

- Consider the following *high level* file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y

Steps the FS takes (in logical / ordered writes / logging order):
- Find free blocks for the contents of X, Y, and Z
- Mark the bitmap entries for these blocks as “used”
- Make 3 inodes that point to X, Y, and Z’s blocks, respectively
- Add directory entries for X, Y, and Z to make sure they ‘exist’ in our file system somewhere
- FINALLY write to them

-------
- Mark bitmap entry for file Y as ‘free’
- Remove dirent for Y / references
- Remove inode entry for Y (we might not zero out the block… why?)

-------
- Repeat the first group of steps for file A

If we use *fsck* with no write-ordering, a lot of bad things can happen…
- We can have 2 live inodes pointing to the same block(s)
- We can have “leaked” blocks that are ‘used’ in the bitmap but don’t exist anywhere!
- We can have an inode pointing to stale data (or garbage)
Practice Problem!

- Consider the following high level file system steps:
  1. Create 3 new files, X, Y, and Z
  2. Write data to files X, Y, and Z
  3. Delete file Y
  4. Create file A, using the free’d blocks from file Y

Steps the FS takes (in logical / ordered writes / logging order):
- Find free blocks for the contents of X, Y, and Z
- Mark the bitmap entries for these blocks as “used”
- Make 3 inodes that point to X, Y, and Z’s blocks, respectively
- Add directory entries for X, Y, and Z to make sure they ‘exist’ in our file system somewhere
- FINALLY write to them

-------
- Mark bitmap entry for file Y as ‘free’
- Remove dirent for Y / references
- Remove inode entry for Y (we might not zero out the block… why?)

-------
- Repeat the first group of steps for file A

If we use fsck with no write-ordering, a lot of bad things can happen…
- We can have 2 live inodes pointing to the same block(s)
- We can have “leaked” blocks that are ‘used’ in the bitmap but don’t exist anywhere!
- We can have an inode pointing to stale data (or garbage)
Practice Problem!

- Consider the following *crash recovery* log:
  - **Begin log entry 1**
    - Find free blocks \(x_1, x_2, x_3\) to store file 1
    - Update the free map to mark \(x_1, x_2, x_3\) blocks as used
    - Allocate a new inode \(i_1\) pointing to \(x_1, x_2, x_3\)
    - Add a directory entry to our new file's parent directory that refers to \(i_1\)
    - Start writing into \(x_1, x_2, x_3\)
  - **Commit log entry 1**
  - **Begin log entry 2**
    - Find free blocks \(y_1, y_2\) to store file 2
    - Update the free map to mark \(y_1, y_2\) as used
    - Allocate a new inode \(i_2\) pointing to \(y_1, y_2\)

- Suppose the system crashes now. Assume that Begin/Commit pairs mark logging *transactions*. Describe the state of the filesystem after the system recovers and replays the log. (ex. Are the files gone? What about the contents of the files?)
Consider the following crash recovery log:

- Begin log entry 1
  - Find free blocks $x_1, x_2, x_3$ to store file 1
  - Update the free map to mark $x_1, x_2, x_3$ blocks as used
  - Allocate a new inode $i_1$ pointing to $x_1, x_2, x_3$
  - Add a directory entry to our new file's parent directory that refers to $i_1$
  - Start writing into $x_1, x_2, x_3$
  - Commit log entry 1

- Begin log entry 2
  - Find free blocks $y_1, y_2$ to store file 2
  - Update the free map to mark $y_1, y_2$ as used
  - Allocate a new inode $i_2$ pointing to $y_1, y_2$

Suppose the system crashes now. Assume that Begin/Commit pairs mark logging transactions. Describe the state of the filesystem after the system recovers and replays the log. (ex. Are the files gone? What about the contents of the files?)
Consider the following crash recovery log:

- Begin log entry 1
  - Find free blocks \( x_1, x_2, x_3 \) to store file 1
  - Update the free map to mark \( x_1, x_2, x_3 \) blocks as used
  - Allocate a new inode \( i_1 \) pointing to \( x_1, x_2, x_3 \)
  - Add a directory entry to our new file's parent directory that refers to \( i_1 \)
  - Start writing into \( x_1, x_2, x_3 \)
  - Commit log entry 1

- Begin log entry 2
  - Find free blocks \( y_1, y_2 \) to store file 2
  - Update the free map to mark \( y_1, y_2 \) as used
  - Allocate a new inode \( i_2 \) pointing to \( y_1, y_2 \)

Suppose the system crashes now. Assume that Begin/Commit pairs mark logging transactions. Describe the state of the filesystem after the system recovers and replays the log. (ex. Are the files gone? What about the contents of the files?)

Just because entry 1 committed, doesn't mean that the actual steps were performed in memory!