# Welcome back $^{back}$back$^{back}$ to CS439H!

Do you remember the 21st night of September?

Love was changing the minds of pretenders

While chasing the clouds away

Our hearts were ringing

# Stress

- 439H is **not an easy class**
  - Lots of new material
  - Unfamiliar programming environments
  - Fast, often relentless pace
- Struggling in this course is normal
  - There will be times you won't know the answer or solution
  - This is expected - we want everyone to succeed, but the only way we can help is if you ask for it
- If you find yourself overwhelmed or spending more time on this class than you think you should be, **please reach out** to Dr. Gheith or the TAs
  - We can help out as far as the class goes
  - We can provide other resources if we are not able to help

Mental health resources available at UT

# Reviewing Quiz 2 everybody say AAAAAAA

And open wide

# Question 1: Evil Gheith

Part A:

Advantages

- Queue can be lock free
- Work can be assigned to target core

Disadvantages

- Core/Work Starvation
- Load Balancing Issues

# Question 1: Evil Gheith

Part B: How do we exploit a cooperative scheduling policy?

- Don’t be cooperative! Have go routines that infinite loop, effectively shutting down the core that it runs on

```
//assuming 4 cores
for(int i = 0; i < 3; i++){
     go([]{while(true);}); //shut down 3 of the cores
}

// wait some time (more formally, should use a counter to synchronize)

go([]{Debug::shutdown();}); //this will never run

while(true); //shut down the last core
```

# Question 2: Channels and criticals and futures oh my!

Future from channel

```
Channel c;
get(work):
    c.receive((value) {
        c.send(value);
        work(value);
    });
set(value): c.send(value);
```

This sounds similar to a semaphore...

# Question 2: Channels and criticals and futures oh my!

Future from channel

```
Channel c;

get(work): c->receive(work);

set(value): c->send(value, { set(value) });
```

# Question 2: Channels and criticals and futures oh my!

Critical section from channel

```
Channel c;
run(work):
    c.send(anything, {
        work();
        c.receive((value) {});
    });
```

This really sounds similar to a semaphore...

# Question 2: Channels and criticals and futures oh my!

Critical section from channel

```
Channel c;
Critical() { c.send(anything); }
run(work):
    c.receive((value) {
        work();
        c.send(anything);
    });
```

# Question 3: Invalid Test Cases (Part 1)

```
void kernelMain() {
    Channel<int> c {};
    c.recv([&c](int value) {
        Debug::printf("ping: %d\n", value);
        c->send(value);
    });
    c.send(5, [&c]() {
        c->recv([](int value) {
            Debug::printf("pong: %d\n", value);
            Debug::shutdown();
        });
    });
}
```

- Channel created on the stack, then shared by reference to the receive and send continuation events.
- After kernelMain executes, c goes out of scope so accessing it produces indeterminate results

# Question 3: Invalid Test Cases (Part 2)

```
// Called from kernelMain
void subroutine(Future<bool>* done) {
  Channel* c = new Channel();
  Barrier* b = new Barrier(4);
  for (int i = 0; i < 3; i++) {
      c->receive([] (int value) {
          Debug::printf("got a value
%d\n", value);
          b->sync([] {});
      });
  }
```

```
  b->sync([] {
      // Set the future once all work is done
      done->set(true);
      delete barrier;
  });
  c->send(10, [c] {
      c->send(20, [c] {
          c->send(30, [c] {
              delete c;
          });
      });
  });
}
```

- Variables not getting captured in closures
- b and c may get deleted before the threads get released, which produced indeterminate results

# P4

```
check_feedback([]
    (auto feedback) {
        ASSERT(
            feedback.max() != 'A'
        );
    }
}
```

**How is p4 going?**

A. that’s a thing?
B. Cloned the project.
C. Looked through the starter code.
D. Started planning/writing code
E. Done with at least one part of the project
F. Done with the whole project but still failing a couple test cases
G. Any% p4 Speedrun glitched

# p4 due date

- P4 deadline has been extended a bit
  - Test is now due Tuesday, 9/26
  - Code is now due Thursday, 9/28

# What is a file system?

# What is a file system?

- How we can store and represent data in some storage medium (like a hard drive)
- How do we distinguish between different chunks of data?
  - Files can encapsulate a single meaningful chunk of data, along with some associated metadata
  - Directories can encapsulate groups of files similarly

| | |
|---|---|
| file1.txt | “Lorem ipsum…” |
| image.png | “‰PNG IHDR €8…” |
| directory/ | ??? |

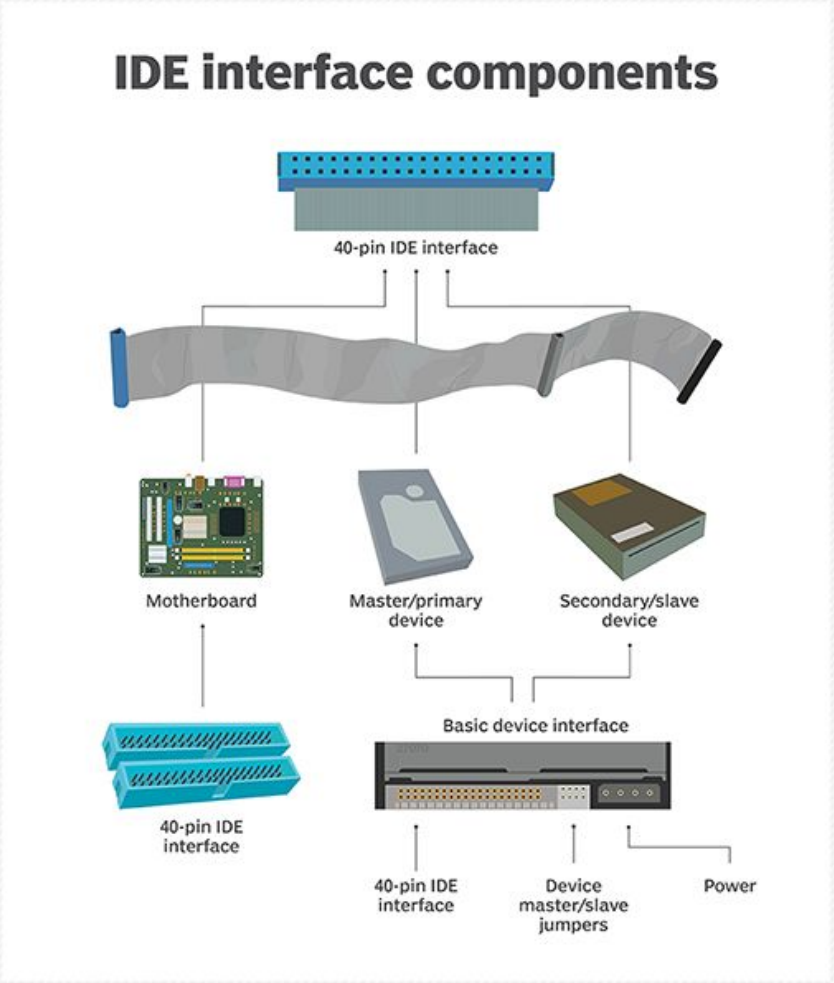
[x x x x x x x x x x x x x x x x x x x x L o r e m i p s u m x x x ... x x x x x x x x x ‰ P N G I H D R € 8 x x x x x x x x x x x x x x x x x x ...]

# Where does the file system go?

- Stored on some external storage medium, usually non-volatile
  - Flash
  - SSDs
  - Hard Disks
  - Etc
  - Sometimes, you can have a file system fully in memory!
- Reading and writing from the medium:
  - Some have a moving magnetic arm that physically moves across the medium to read data - latency depends on where the data physically lives in relation to the arm (e.g. disks)
  - Some are more random-access but can incur wear-and-tear based on which regions are accessed more often (e.g. flash)

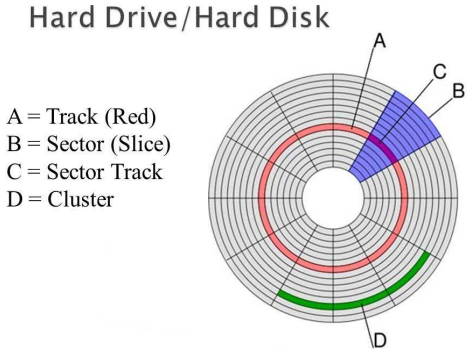
# How do we access the data?

- IDE - physical hardware bus that connects to a disk
- Implemented for you, basically don’t worry about it



# How do we locate data in storage?

- Usually not byte addressable - can only access larger chunks of data
- Addressed/Accessed in **sectors**
  - sector: size of byte divisions on the disk itself, independent of what we plan to do with it (usually 512 bytes)
  - Logical sectors vs Physical sectors (we work in logical sectors)

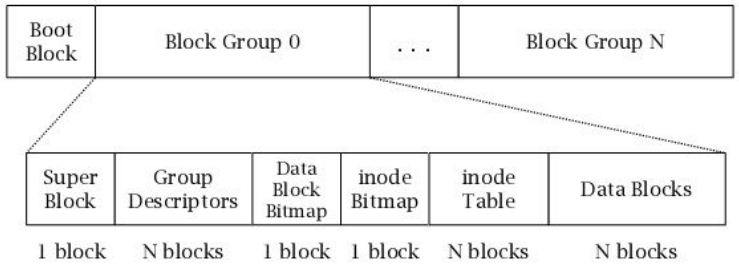


# How do we locate data in storage?

- **Blocks**
  - The byte sizes of blocks within the byte divisions on a *file-system* level;
    not determined by hardware
  - (set by the csid.block_size file in your testcase)
  - If the block size is larger than the sector size, we can group sectors together to store blocks
  - Block size should **never be smaller** than sector size
    - Partial reads/writes are very expensive and inefficient!

# What is Ext2?

- Superblock
  - Contains metadata about the file system as a whole
  - block size, inode size, where the GDT is, etc.
- Block Group Descriptor Table
  - Describes block groups
  - **Block groups** are groups of blocks that inodes, data live in
  - Contains block usage information, inodes, and the actual data blocks
- Our data (files, directories, etc.) are represented by **inodes**



# The Superblock

- 1024 bytes long, located near the beginning of the disk (starts at byte #1024, goes to #2048)
- Contains lots of useful information about the filesystem
  - Block size
  - Block group size
  - Number of blocks
  - Number of inodes
  - Usage statistics (last write, last mount, etc)
  - Other metadata about the filesystem
- Most of these fields are 4 bytes long (uint32_t), a few are 2 bytes long
- (You probably don't want to read each field from disk individually)

# The Block Group Descriptor Table

- Basically a contiguous array of block group descriptors
- What's a block group descriptor?
  - Describes a block group (shocker)
  - Has this stuff:

| Starting Byte | Ending Byte | Size in Bytes | Field Description |
|---|---|---|---|
| 0 | 3 | 4 | Block address of block usage bitmap |
| 4 | 7 | 4 | Block address of inode usage bitmap |
| 8 | 11 | 4 | Starting block address of inode table |
| 12 | 13 | 2 | Number of unallocated blocks in group |
| 14 | 15 | 2 | Number of unallocated inodes in group |
| 16 | 17 | 2 | Number of directories in group |
| 18 | 31 | X | (Unused) |

# Inodes

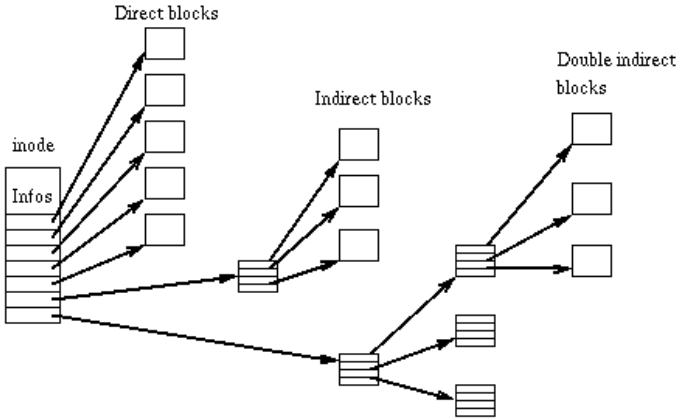
- Describe a clump of data along with metadata for it
  - Creation/Modification/Access date
  - Owners/Permissions
  - Pointers to data
  - and some more
- Indexed **starting at 1**
  - (yes, you will probably need to subtract 1 at points in your code)
- The root directory is inode **2**

# Direct and indirect pointers

- Inodes do not store their data content directly inline (except symlinks)
- Direct block indices direct give the block index of their relevant data
  - Only a single lookup necessary
- Indirect block indices point to blocks that themselves contain indices of blocks
  - Multiple levels of lookup necessary depending on the level of indirection

# Direct and indirect pointers

- How are the block indices of data laid out?
- 12 direct indices for the first set of data (12 blocks of data)
- 1 index of a block of singly indirect block indices ((block size/4) blocks of data)
- 1 index of a block of doubly indirect block indices ((block size/4)$^2$ blocks of data)
- 1 index of a block of triply indirect block indices ((block size/4)$^3$ blocks of data)



# Files

- Stored as inodes as well
  - Metadata tag differentiates this
- The data of the file is just the content itself
- there’s nothing else special about a file

# Directories

- Stored as inodes
  - Metadata tag tells you this
- The data of the directory is an array of directory entries
  - Each entry contains an inode number as well as a name
    - This is what the file name is!
  - Inode number of 0 can be used to indicate an empty entry (doesn't point to anything)
  - Entries do **not** span across blocks
    - Entries may be padded in size or have empty entries to meet this condition

# Hard links

- We never guaranteed that inodes are uniquely stored in one directory across the whole file system!
- **Hard links** take advantage of this
  - Two (or more) separate directory entries that happen to have the same inode number
    - They use the same inode + content
  - Can have very different file names and be in different directories

# Symbolic links

- Refers to another file in the file system
- The data of the symbolic link is a path that points to another file
  - If a symlink path is short enough, then we don't use block indices - we directly store the path inline in the inode

# How does this all relate to p4?

- `block_io.h`
  - Our abstraction for reading from storage
  - Options to read a block or a specific range of storage
    - All reads ultimately redirect to reading one block at a time
  - Implemented for you
- `ext2.h/ext2.cc`
  - Ext2: The class that handles our general file system operations
  - Node: The class that handles inode-specific operations
  - You must implement
- `ide.h/ide.cc`
  - An instantiation of block_io that implements reading from the physical disk sectors
  - Simulates the real IDE from hardware - implemented for you

# How does this all relate to p4?

- Test folders!
  - Since we are testing a file system, your tests now include a folder that represents our file system
  - You can add symlinks, directories, files
    - Hard links don’t play well with git unfortunately
  - Test case will consist of a normal `.cc`/`.ok` file as well as a `.dir` folder
  - The makefile will format this folder into an ext2 image that QEMU uses as our storage device

# C considerations

- Reading interface takes an integer representing where to read and a char buffer - the char buffer needs to be created beforehand and is filled by the function. Can be casted to whatever data type you want (what's in the file?)
  - It may be worth casting to a custom struct when you're dealing with data you know the structure of, otherwise you just have an array of bytes
  - How to make sure space isn't added in the wrong place? C structs need to be aligned

```
struct Thing {
    uint32_t a;
    uint16_t b;
    uint32_t c;
    uint16_t d;
}
```

[a a a a b b _ _ c c c c d d _ _]

  - This causes issues if you expect the data to not have holes in it

# C considerations

- Solution: packed structs
  - Add a note to the compiler telling it not to add alignment

```
struct Thing {
    uint32_t a;
    uint16_t b;
    uint32_t c;
    uint16_t d;
}__attribute__((packed));
```

[a a a a b b c c c c d d]

https://wiki.osdev.org/Ext2

THIS PAGE IS YOUR BEST FRIEND!!!

## Quick Summaries

### How To Read An Inode

1. Read the Superblock to find the size of each block, the number of blocks per group, number Inodes per group, and the starting block of the first group (Block Group Descriptor Table).
2. Determine which block group the inode belongs to.
3. Read the Block Group Descriptor corresponding to the Block Group which contains the inode to be looked up.
4. From the Block Group Descriptor, extract the location of the block group's inode table.
5. Determine the index of the inode in the inode table.
6. Index the inode table (taking into account non-standard inode size).

Directory entry information and file contents are located within the data blocks that the inode points to.

### How To Read the Root Directory

The root directory's inode is defined to always be 2. Read/parse the contents of inode 2.

# Other Resources

https://www.nongnu.org/ext2-doc/ext2.html

https://wiki.osdev.org/Ext2

https://en.wikipedia.org/wiki/Ext2

# Questions?