In this lecture we will look at discussing HW3 and HW4.

**HW 3 overview**
- It required simulating a file system.
- We did not allocate blocks on the disk.
- Methods implemented:
  - Directories
  - Files
  - `mkdir`
  - `rm`
  - `read`
  - `append`
  - `delete`

- The basic structure for HW 3 was:

```
Home --> Joe --> file.txt --> Len --> Blocks
```

- One of the challenges was implementing the *read* operation.

```
Read
```

It required that the *read* handle data that should be skipped and also handle the empty space at the end of the block (and *read* only the part specified).

The idea was to try and express the operation as a single loop:

```
read (offset, len)
block = 1st block
while offset ≥ 512
  if block = next block
    offset = 512
  y
```
to copy = min (file len - offset, len requested)
while to copy > 0
    this copy = min (512, to copy)
    read from block this. copy bytes starting @ offset
    block = next block
to copy -= 512
offset = 0

HW4 Overview
- In this assignment we will use the File system in USER space (FUSE) toolkit.

- The basic structure for the assignment will be

  struct fuse_ops myops = {
  "getattr",
  "mknod",
  "mkdir",
  "unlink",
  "rename",
  "open": check to see if a file is there. If a valid file handle is found, should not latter say "NO SUCH FILE",
  "read",
  "truncate": chop the file to a specified length,
  "write",
  "release": close,}
The operations are then called using
main()
{ fuse-main(....., x myops)
}

Parts Involved
Mounting

- Superblock
  - Occupied one block
  - It tells us how long the block size is and how long the FAT is.
  - Using the superblock we know where the root directory starts.

In Assignment/HW 3 we used/developed an in-memory file system and used any data structure.
In HW 4 we need to develop a generic routine that will let us read from a file.

The basic Directory Structure is shown as follows:

Each directory entry has all the necessary file information and the only things stored outside the directory are the data blocks. This implementation is expensive in terms of memory because each name entry is 43B long.
For HW3, the file structure resembled

```
+----+(Len)
   |    |
   |----- REFERENCES
   |      |
   |      |
   |      |
```

For HW4, the above structure is one of the entries of the Directory structure explained above.

It will be beneficial to learn the `MAN` command.

The `getattr` operation can be modeled as `stat`: which reads a directory entry for a particular file.

The `open` operation can be developed as

```c
int open (char *path, file_info *fp) {
    where path is of the form `/dir1/file1`
    if lookup path = 'NOT FOUND'
        ERROR
    if permissions fail or is_Directory
        ERROR
    otherwise
        return OK
}
```

- The operation must return the right error codes.
- The parsing of paths is similar to HW3.
- The user sees an uninterrupted file system space.
- If it follows the UNIX convention: If the operation is successful, it returns a 0 or positive number. A failure is signalled by a negative number which are error codes.

![Diagram](image)
The other operations can be described to have the following functionality.

**mknod**: Creates a file. In this operation we do not allocate blocks but make directory entries.

**mkdir**: Allocates a single block for the directory when you create it.

**unlink**: Frees up the diskblocks allocated for a file.

Using the structure, **Free Space Management** is relatively easier.

---

![Diagram](image)

The FAT contains a **freebit** and the **next pointer** to a block. As per the structure, represented above each entry in the structure corresponds to blocks. The **FAT contains entries for each block in system**. While allocating blocks for a file, we set the free bit and go to the next available free block. For each block allocated which is not the end of the file the EOF field is not set. When the last block for a file is reached the EOF field is set.

---

**Corresponding entry in the structure for Block**

**Next pointer**

**Free** -- indicates block is free or not

**EOF** -- indicates end of file

---

This structure does not give constant time access even if the offset is known.
For the assignment use a Data Structure in UNIX
  * getattr struct static *st

  \[
  \text{model} \quad \text{st-mode} \\
  \text{st-nlink} = 1 \\
  \text{st-uid} \\
  \{ \text{st-gid} \} \text{ fields in directory structure} \\
  \text{st-size} \\
  \text{st-blksize} \\
  \text{st-blocks}
  \]

  UNIX permissions
  \[
  0777 \\
  rwx rwx rwx
  \]

  \[
  \text{st-atime} \quad \text{store a single modification} \\
  \text{st-mtime} \quad \text{time and return it for all the} \\
  \text{three}
  \]

  To find the information about file pass the structure to
  lstat (int Fd, struct stat * )

  To find the status of the file system

  \[
  \text{f-bsize} : \text{blocksize (1024)} \\
  \text{f-blocks} : \text{total # f blocks - use superblock & FAT} \\
  \text{f-bfree} : \text{free blocks} \\
  \text{f-files} : \text{total files} \\
  \text{f-ffree} : \text{free files} \\
  \text{f-namelen} : 43 \text{ (in our case)}
  \]

  The Basic flow can be represented as

  \[
  \begin{align*}
  \text{0} & \quad \text{Traverse Directory Table} \\
  \downarrow & \quad \text{Allocate/Free Blocks} \\
  \text{Startup} & \quad \text{Read/write blocks} \\
  \text{Open Disk Image} & \quad \text{Give it to System I/O}
  \end{align*}
  \]
Recommendations for HW4

- Although you want to encapsulate the details, you do not want to read and cache in memory. The implementation should have Block I/Os for every block in memory.

- You would want to start with a minimal implementation and build on your structure.

- Add optimizations in the following manner
  - Determine how the Block I/Os occur for the different operations and improve on them
  - We could have compaction for the data blocks.

End of Part I - Niraj S. Kudalkar