1. How do you cycle through MINODES
    a.

2. How does balloc work?
    a.

1. How do you open file for R or W
    a. So the mode of the file is 0(read), 1(write), 2(read/write), 3(append)
    b. After you check that the file exists and is a regular file and not a dir and is not already opened by another process you have to find a place to put it in the OFT (open file table) array.
    c. If there is a spot you make a new OFT and assign the values like sharecount, mode, minodeptr.
    d. Then depending on the mode
        i. For read you just set the oftp->offset = 0 because you want to read at the beginning of the file
        ii. For write you truncate(mip) and then set the oftp->offset = 0;
2. How did you handle Indirect/double indirect blocks in read(), write()
    a.
3. How did you do tail filename
    a.
4.

Util.c lines 123 ← This is old mailman algo

```
// 3. get INODE of (dev, ino) into buf[ ]
  blk = (ino - 1) / 8 + iblk;
  disp = (ino - 1) % 8;
  get_block(dev, blk, buf);
```

New Mailman Algo

```
  blk = (ino - 1) / BLKSIZE/sizeof(INODE) + inodes_start;
  offset = (ino - 1) % BLKSIZE/sizeof(INODE);
  get_block(dev, blk, buf);
  ip = (INODE *)buf + offset
```

The inodes_per_block = BLKSIZE/sizeof(INODE) = for other disks it used to be 1024/128 which equals 8. Now it should be 1024/256 = 4

---

balloc()

- Bitmap is used to allocate or deallocate blocks
    - It tells us which blocks are in use and which are free based on their bit corresponding to the block
- Goal: allocate a free block on a given device by searching for an available block in the block bitmap
- 1439 max bits
- We calculate the byte and bit offset
    - Byte offset = block no / 8
    - Bit offset = block no % 8
- We check if the bit corresponding to current block is 0
    - Buf[byte offset] & (1 << bit offset)
    - Byte offset
        - refers to the byte in the block bitmap that contains the bit corresponding to the current block
    - Compares the byte that contains the bit and the bit will be shifted to
- We mark the bit as 1 to indicate the block is now in use
    - We use an OR operation to mark our bit as 1
    - it modifies the value of the byte in buf at the byte offset to allocate the block.
- We write to block back to the disk and return (block number + 1)

open_file()
Goal: Open file for read, write, read/write, or append
Parameters: char* pathname, char* mode
Returns: file descriptor

- Summary:
    - First we check if the file we pass into our function exist by seeing if a MINODE is node when we call path2inode
    - We create the file if it does not exist and assign a MINODE pointer to our new file
    - We check if the file we created is a regular file
    - We check each open file table entry to ensure we find the file matching our parameters
        - Device, inode number, shareCount > 0 (>0 = open file)
    - If we cant find our entry in the open file table that means the file isn't open and we have to make a new file table entry
    - We find a new slot by finding the first index with a shareCount of 0 in the open file table
    - When we make the entry we give it the minode pointer we created to reference the minode we want to open
    - We let the running processes's open file table equal our file table pointer at the index we found
    - We update the access time and/or modification time depending on the mode and mark the node modified before we return it

close_file()
Goal: Close open file in the open file table
Parameters: char* fd
Returns: nothing

- Verify if file descriptor is within range and does not exceed the max number of file descriptors
- We set an open file table pointer equal to the file descriptor of the current running process
    - running->fd[fd]
- We set the running process file descriptor to 0
    - This frees a slot in the open file table
- We reduce the open file table pointer's shareCount by 1 to indicate the file is not being used by any processes
    - We error check this by making sure the open file table pointer is not greater than 0 at this point
- We make a MINODE variable and point it at the open file table pointer's minode pointer
- We mark it as modified and dispose of it using iput
    - If we do not dispose the pointer, the memory allocated for the entry will stay in use even if the file is not open (MEMORY LEAK)

Lseek()
Goal: Move the offset of the file descriptor to the new position
Parameters: char* fd, char* position
Returns: the new position we set for the file descriptor

- Verify if file descriptor is within range and does not exceed the max number of file descriptors
- Verify entry exists by checking if the running process's open file table contains the file descriptor we gave as a parameter
    - It will be NULL if there is no entry
- We get the MINODE pointer associated with the given file descriptor
    - running->fd[fd]->minodeptr
- We set a current position variable to the given file descriptor's offset
    - running->fd[fd]->offset
- Get the file size (this will be used for error checking later)
    - mip->INODE.i_size
- We set a new position variable that is initially equal to the position parameter we sent into the function
- We error check the new position to make sure it is not less than 0 or bigger than the file size
    - Otherwise we won't be editing anything valid (or that exists)
- We set the open filetable entry's offset equal to the new position
- Return new position

Pfd()
Goal: Print all open file directories
Parameters: None
Returns: nothing

- We loop through all entries in the oft array
    - The number of entries is specified by NOFT
- We create an open file table pointer that points to the open file table we declared in main
- We skip all empty entries in the open file table pointer
    - by checking if their shareCount is equal to 0
- This is the part where actually print the file descriptor, the mode, and offset
- We print the mode (READ/WRITE, READ, WRITE, APPEND) based on the mode in the file table pointer
    - oftp->mode
- We print the offset of the open file table pointer
- We create a MINODE pointer that points to the MINODE pointer contained in our open file table pointer
    - MINODE *mip = oftp->minodeptr
- We print the device and inode number of mip

Read()

- Input:
    - int fd, char buf [ ], nbytes
    - file descriptor, buffer to hold read bytes, number of bytes
- Uses these functions
    - read_file() ← main func
    - myread() ← the func that reads from fd
- read_file()
    - This function assumes that the file is opened for read, write, read/write
    - It asks the user which file descriptor tey want to use and how many bytes they want to read. It then passes over that info to myread() to actually read the bytes
- myread()
        - This is the function that actually reads the bytes from fd
    - The offset in the oft points to the current position in the file where the user wants to read bytes from. Offset can be at the start of the file if nothing is done or in the middle somewhere if lseek is used to offset it.
    - This function is in a while loop and will go while the number of bytes (nbytes) and the available bytes are nonzero.
    - You compute the logical block number lbk and startbyte in the block from offset because that will give you the position on where you need to start in relation to the file.

        - You will then read through the direct block (lbk < 12) , indirect blocks (lbk >=12 && lbk < 256 + 12), and double indirect blocks (lbk >= 13 && lbk < 256
    - Explain the Func

truncate()
Goal: Releases an inode's data blocks and set its size to 0, essentially shortening the length of a minode object
Parameter: MINODE* mip
Returns: nothing

- Summary:
    - If deallocates all direct blocks by iterating through INODE i_block array, specifically the first 12 blocks

    - We deallocate the indirect blocks reading the block (13) from the disk
    - We iterate through its array of block pointers
    - As long as the block is non zero, we call bdalloc

    - We deallocate the double indirect blocks by reading the block from disk
    - We iterate through its array of block pointers
    - As long as the block exists, we read that block's block from the disk
    - We iterate through THAT block's array of block pointers and call bdalloc on each non zero block pointer

- First, we deallocate the direct blocks
    - for (int i = 0; i < 12; i++)
    - If the inode's data block exists then we deallocate it with bdalloc
        - bdalloc(mip->dev, mip->INODE.i_block[i]);
    - Set the block to 0
- Then we deallocate the indirect blocks
    - If the 12th block exists, it is read into a buffer using get_block
    - The block contains an array of integers (4 bytes each), which represent blocks being pointed to by the indirect block
    - We reference the indirect block using an integer pointer
        - int *indirect = (int *)buf;
    - We iterate through the indirect array using a for loop
        - The loop can access each integer in the array one at a time
            - We split blksize into integer sized chunks (4 bytes at a time)
        - If the integer is not zero, we have a valid block
        - we deallocate it by calling bdalloc
    - Finally, we call bdalloc on the the 12 block and then set the block to 0
- For the double indirect blocks, we do the same thing as we did for the indirect blocks except we access the 13th index (block 14) and instead of deallocating the block

- We read the double indirect block into a buffer
- We create an integer pointer to the buffer
- We iterate over the indirect block pointers, pointed to by our integer pointer
- For each existing indirect block we read it into another buffer
- We create an integer pointer into the second buffer and then iterate over it to find data blocks
- Any non-zero block is deallocated using bdalloc
- We call bdalloc on the integer pointers we created on our buffers and set their respective blocks to 0 from the minode pointer

dup()
Goal: Duplicates an existing file descriptor and returns a new one that refers to the file descriptor
Parameters: char* fd
Returns: new file descriptor that refers to the file descriptor

- Verify if file descriptor is within range and does not exceed the max number of file descriptors (NFD)
- We check whether the file descriptor is open by checking if the running processes open file table contains an entry for the file descriptor
    - running->fd[fd]
- If the file descriptor is open, we increment the share count
- Then we find the first available file descriptor and return it by iterating through the running process's open file table and look for the first null entry
- Then we set the open entry to the file descriptor in our running process's open file table to the file descriptor we gave to our function and then return it
- At this point we have duplicated the file descriptor

dup2()
Goal: Duplicates an existing file descriptor into another existing file descriptor
Parameters: char *fd, char* gd
Returns: nothing

- Verify if both file descriptors are within range and does not exceed the max number of file descriptors
- Check if the group descriptor is open in the running process's open file table
- We want to close the group descriptor file to ensure there are no issues when we replace the group descriptor with the file descriptor
- We create an open file table pointer which points to the file-descriptor index of the open file table array of the running process.
    - OFT *oftp = running->fd[fd]
- Then we set the group descriptor to the open file table pointer at the running process's open file table array at the group descriptor index
    - running->fd[gd] = oftp
- Then we increase the shareCount of the open file table pointer to ensure the open file table entry is open and running

bdalloc()
Goal: deallocate a block on a file system by setting the corresponding bit in the block bitmap to 0, indicating that the block is now free
Parameters: int dev, int blk
Returns: Nothing

- Starts by reading the bitmap block into a buffer using get_block(dev, bmap, buf)
    - To check if the block is valid, it should be greater than 0 and less than the total number of blocks on the device
- We call the clr_bit function on the bit position from the bitmap block so that we can clear the bit in the bitmap that corresponds to the given block number
    - Bit position = (blk - 1)
    - first bit in bitmap represents block 0 so we subtract 1 to get the corresponding bit position in the bitmap
    - We do this because when a block is deallocated, it needs to be marked as free in the file system's data structures so that it can be reused later. By setting it to 0 we indicate the block is free
- We write the buffer block back to the disk
- We get super block by
    - get_block(dev, 1, buf)
    - Sp = (SUPER*) buf
- We get group descriptor block by
    - get_block(dev, 2, buf)
    - Gp = (GD *) buf
- We update the free block count in the super block and the group descriptor by incrementing their s_free_blocks_count and bg_free_blocks_count fields
    - This indicates that a new block is available for allocation
    - sp->s_free_blocks_count++
    - gp->bg_free_blocks_count++
    - These keep track of the number of free data blocks available in the filesystem
- Finally, we write the updates superblock and group descriptor blocks back to the disk using the put_block() function
    - put_block(dev, 1, buf);
    - put_block(dev, 2, buf);

ialloc()
Goal: Allocate an available inode number from the inode bitmap block for the given device
Parameters: int dev

- First reads the inode bitmap block into a buffer using get_block
- We search through the bitmap using a loop, looking for an inode number that is currently marked as available (marked as 0 bit value)
    - We do this by calling test bit and seeing if the bit is unset
- Once an available inode number is found, we set the bit in the bitmap as allocated using set_bit, which will mark the bit as 1

- We write the update bitmap block back to the disk using put_block
- We then decrement the number of free inodes
- We return 0 if there are no available inodes

balloc()
- Bitmap is used to allocate or deallocate blocks
- 1439 max bits
- We calculate the byte and bit offset
    - Byte offset = block no / 8
    - Bit offset = block no % 8
- We check if the bit corresponding to current block is 0
    - Buf[byte offset] & (1 << bit offset)
    - Byte offset
        - refers to the byte in the block bitmap that contains the bit corresponding to the current block
    - Compares the byte that contains the bit and the bit will be shifted to
- We mark the bit as 1 to indicate the block is now in use
    - We use an OR operation to mark our bit as 1
    - it modifies the value of the byte in buf at the byte offset to allocate the block.
- We write to block back to the disk and return (block number + 1)

isEmpty()
Goal: we use in rmdir to make sure that the dir is empty because we can't rm a dir it it has contents in it
Parameters: MINODE *mip
Return: 1 if empty, 0 if not found, -1 if the directory is not valid
- We check if the link count is 2
    - This tells us we have "." and ".."
- We traverse each block and check for their existence
- Store each block into a buffer
- Create current position and and directory pointer variables so we can traverse the block
- We copy the directory entry's name into a buffer
- We compare the name we entered into our buffer with "." and "..".
    - If they exist we return 0 because the directory is not empty
- We set the current position to the directory pointer's record length and set the directory pointer to the current position (as a dir) so that we can keep traversing the direct blocks

iput()
Goal: Releases a minode no longer needed by a process
Return: 1 if completed
- The function checks if the minode pointer is NULL

- If not then we decrement the shareCount of the minode because it will no longer be in use
- We check the shareCount again to ensure it is not in use by another process
- As long as the shareCount is 0 we can check if the minode has been modified
- We calculate the disk block number and offset of the INODE within the block
- We use get_block to read the disk block containing the INODE into the buffer
- We get an INODE pointer by casting the buffer to an inode pointer and add the offset to it
- We call put_block to write the updated buffer back to the disk where the MINODE resides
- We remove the MINODE from the cache list by traversing the list and looking for a node that matches the MINODE we want
- If we

path2inode()
Goal: Return a pointer of the inode for the given pathname
Parameters: int dev, int blk
Returns: a pointer of the inode for the given pathname

- Check if the pathname is an absolute or relative path
- If the first character of the pathname is “/” we have an absolute path
    - If relative,, we start at the current working directory
- Tokenize the pathname
    - Use “/” as delimiter
- For each token in the pathname, we check if it is “.”, which means we are in the current directory and do not need to do anything
    - If the token is “..”, we need to move up a level in the directory tree
        - We find the the parent directory using findino to obtain the parent inode number
        - Release the current directory MINODE using iput
        - Retrieve the parent directory minode using iget and the inode number we got
- If we are not dealing with “.” or “..”, we use the search function to find search the current directory for the file/directory and save the inode number
    - If we could not find the file or directory we release the current directory’s minode and return NULL to indicate the file or directory does not exist
- If we found the file, we release the current directory’s MINODE and retrieve the MINODE of the of the found file using iget with the inode number we got earlier

make_dir()
Goal: make dir

- To allocate a block
    - balloc(dev)
- To allocate an inode

    - ialloc(dev)
- We load the inode into memory with iget and the inode number
- We call get block to a buffer, using the inode we got from allocating a block
- We add the “.” and “..” entries
- Write the block to the disk
- We use enter_child to enter child directory entry into the parent directory’s inode

iget()
Parameters: int dev, int ino

- We check to see if the if we can find a minode matching the dev and ino we gave the function
    - We return the mip if we found it and increment the sharecount, cachecount, and hits
- We set our minode as the head of the freeList because we’re going to return a free inode from the list
- We remove the minode from the freeList
    - By setting freeList = freeList->next
- Update shareCount = 1, set dev, ino, cachecount = 1
- Mark minode as modified
- Using the inode number we passed into our function, we calculate the block and offset
- We get the block and store it into a buffer and copy the contents into an inode
    - INODE *ip = (INODE*) buf + offset
- We copy the INODE into our MINODE’s INODE structure
- We add the the new minode to the cachelist and return the minode
    - mip->next = cacheList
    - cacheList = mip

Findino()
Parameters: MINODE* mip, int* myino

- We get the first block of the minode and store it into a buffer
- We use directory pointers and current position pointers to traverse the buffer
- We check the name of the directory entry
- If we find the “..” directory then we return the inode of the parent directory
    - dp->inode
- Move to the next directory entry if we did not find it

enter_child()

- Summary:
    - Used to insert a new child directory entry into a parent directory’s inode
    - We first search for a suitable directory entry in the parent’s data blocks to insert the new entry
    - If there is no suitable entry, we allocate a new data block and the entry is added to it

        - We update any necessary parameters (inode size and i_block array) and write the inode to the disk
- Need len is the space required to store a new directory entry in a directory block
    - 8 = size of inode and rec_len fields
    - We add 3 as padding to ensure we can align the name field on a 4 byte boundary (number of bytes needed to store entry on a disk
- We traverse the 12 direct blocks
- We ensure the block exists every time
- We store the block into a buffer
- We traverse the block using the current pointer until we get to the end of the block (we are seeing if we can find a suitable entry and block for our new entry)
    - (while cp < buf + blocksize)
    - We calculate the ideal length for the directory and the remaining space of the directory entry
        - Remain = dp->rec_len - ideal_len
    - We check each directory entry until we find one with enough remaining space for our new directory entry
        - As long as remain is bigger than the ideal len
    - We create a new entry and write it the block to the disk
- If we reach this point, no space in existing data blocks so we have to use balloc to create a new data block and add our new entry as its first entry
- The new block is added to the parent's i_block array and then write the block to the disk
    - Inode size is incremented by block size

LS()
Parameters: char pathname

- I thought i_block[0] contained pointers to the data blocks where its directory entries are stored
- We check if the path exists after calling path2inode and setting it to a MINODE
- We pass the minode as our parent directory to the print dir function
- Print dir grabs the first block from the parent inode and assigns it to a buf
- We type cast the buffer as a directory so we can traverse it
- We copy the directory's name into a string
- We check if the directory contains a valid inode (non zero)
-

Cat()

Goal: Prints off the contents of a file by looping through the file buf that is given my myread(). I use a while to continuously read the bytes for the amount of BLKSIZE into mybuf. Inside the loop we *cp which is the current value of the byte in mybuf and check to see if the file ends and return if that is the case or we check to see if cp is a newline character and if it isn't all of that we print off the character to the screen.

Head()

Goal: to print the top 10 lines of a file. We do this by doing a similar feature to cat where we have a

Tail()

Write()

Copy()

Move()

Other

Moving from disk with 128 inodes sizes to 256 inode sizes

256 bytes allocated per inode one each block
Previously 128

```
int ifactor = BLKSIZE / (inode_size * 2); // Number of inodes per block
blk = ((ino - 1) % inodes_per_block) / ifactor + inodes_start;
disp = ((ino - 1) % inodes_per_block) * inode_size * 2 + ((ino - 1) / inodes_per_block) * 128;
```

When a process opens a file, an unused entry in the oft array is located and associated with the file. The index of the oft entry is then stored in an unused entry of the running->fd (running process open file table) array.

IMPORTANT INDIRECT NOTES:

- You cannot access the indirect blocks directly. You have to call get_block() for each level of indirection
- indirect block is just a disk block that contains 256 integers, each corresponding to a block number.
- Indirect blocks do not contain inodes
- Their purpose is to store information about the physical location of data on a disk

Util.c functions

iget()

Goal: Return the inode that we got from the disk and is now in memory

Process:

1. We do a for loop that assigns mip to cacheList and goes mip = mip->next while mip != 0
    a. We check if mip->dev == dev AND mip->ino == ino
    b. Then we mark down that its in use for this process and cachecount++
    c. We return mip because then that means that its already in the cacheList.
2. Mip == freelist and we check if we have minodes to use
3. We remove the minode from the freeList and set the new values to it.
4. We use mailman to calc blk and offset and get the block and typecast it as an INODE struct
5. We add the values of ip to mip->INODE
6. We then add the new minode to the front of the cacheList with mip->next = cacheList, then we adjust the head of the cacheList to the front or where mip is.
7. We then return mip to use within the function that called it

Process SIMPLIFIED

We for loop through to check to see if the mip we are looking for exists in the cacheList from a past iget call. This step should be gone over most times if we do iget() then iput(). We then make mip == freelist so we can get a new minode then we use mailman to set blk and offset to get_block info of ip. Once we have the inode info for ip we have to typecast it and then we add mip to the front of cacheList and then move the head of cacheList to the new front. We then return mip

iput()

Goal: Releases the minode that is in cachlist

Process:

1. We are given mip and do three checks to see if it is NULL, currently in use for this process and if its modified
2. We then use mailman to get the blk and offset then we get_block and then assign it to an INODE ip then we put that value set of ip into the mip->INODE place.
3. We then write to mip->dev the blk info of size buf.
4. We then remove mip from cacheList using a prev and current MINODE pointer.

Path2indoe()

- It returns an INODE pointer to the file's inode or 0 if the file is inaccessible.

Balloc

Used in: enter_child, my_creat, mkdir

Minode[0-...-63] = from init, it can store the process's
shareCount = For other process's to see if this minode is currently in use
cacheCount = Incremented every time its used, Its touched in hits
cacheList = In memory storage of get_blocks from iget, iput()
freeList
Local mip
Modified

SUPER BLOCK struct
u32 s inodes count; /* Inodes count */
u32 s blocks count; /* Blocks count */
u32 s r blocks count; /* Reserved blocks count */
u32 s free blocks count; /* Free blocks count */
u32 s free inodes count; /* Free inodes count */
u32 s first data block; /* First Data Block */
u32 s log block size; /* Block size */
u32 s log cluster size; /* Allocation cluster size */
u32 s blocks per group; /* # Blocks per group */
u32 s clusters per group; /* # Fragments per group */
u32 s inodes per group; /* # Inodes per group */
u32 s mtime; /* Mount time */
u32 s wtime; /* Write time */
u16 s mnt count; /* Mount count */
s16 s max mnt count; /* Maximal mount count */
u16 s magic; /* Magic signature */ // more non essential fields u16 s inode size; /* size of inode structure *

Magic number determines tell you what type of file system it is, like ext2 or ext4

INODE STRUCT
u16 i mode; // 16 bits = |tttt|ugs|rwx|rwx|rwx|
u16 i uid; // owner uid
u32 i size; // file size in bytes
u32 i atime; // time fields in seconds
u32 i ctime; // since 00:00:00,1 1 1970
u32 i mtime; u32 i dtime;
u16 i gid; // group ID
u16 i links count; // hard link count
u32 i blocks; // number of 512 byte sectors
u32 i flags; // IGNORE
u32 i reserved1; // IGNORE

u32 i block[15]; // See details below
u32 i pad[7]; // for inode size = 128 bytes

Show

Contains

```
sudo mount $1 /mnt

sudo ls -l /mnt

sudo umount /mnt
```

Does

Line1 - mounts the disk
Line2 - ls's the dirs and files that exist inside the disk
Line3 - Unmounts the disk

Head_tail

Contains

```
sudo mount disk2 /mnt

echo Linux head /mnt/small
echo --------------------------
sudo head /mnt/small
echo --------------------------

echo Linux tail /mnt/small
echo --------------------------
sudo tail /mnt/small
echo --------------------------

sudo umount /mnt
```

Does

- `sudo mount disk2 /mnt`: This command mounts a disk named "disk2" at the mount point "/mnt". The "sudo" command is used to execute this command with superuser privileges.
- `echo Linux head /mnt/small`: This command prints the string "Linux head /mnt/small" to the terminal.
- `echo --------------------------`: This command prints a line of dashes to the terminal.

- `sudo head /mnt/small`: This command reads the first few lines of the file named "small" on the mounted disk at the mount point "/mnt". The "sudo" command is used to execute this command with superuser privileges.
- `echo --------------------------`: This command prints a line of dashes to the terminal.
- `echo Linux tail /mnt/small`: This command prints the string "Linux tail /mnt/small" to the terminal.
- `echo --------------------------`: This command prints a line of dashes to the terminal.
- `sudo tail /mnt/small`: This command reads the last few lines of the file named "small" on the mounted disk at the mount point "/mnt". The "sudo" command is used to execute this command with superuser privileges.
- `echo --------------------------`: This command prints a line of dashes to the terminal.
- `sudo umount /mnt`: This command unmounts the disk that was mounted at the mount point "/mnt". The "sudo" command is used to execute this command with superuser privileges

Diff2

Contains

```
sudo mount disk2 /mnt

echo diff large with newlarge
sudo diff /mnt/large  /mnt/newlarge

echo diff huge  with newhuge
sudo diff /mnt/huge   /mnt/newhuge

sudo umount /mnt
```

Does

1. `sudo mount disk2 /mnt`: This command mounts the file system from the device `disk2` to the mount point `/mnt`. This makes the files on `disk2` accessible through the `/mnt` directory.
2. `echo diff large with newlarge`: This command prints the string "diff large with newlarge" to the terminal. This is just a message to indicate that the following command will compare the files named `large` and `newlarge`.

3. `sudo diff /mnt/large /mnt/newlarge`: This command compares the files named `large` and `newlarge` that are located in the `/mnt` directory. The `diff` command compares the contents of two files line by line and reports the differences between them.
4. `echo diff huge with newhuge`: This command prints the string "diff huge with newhuge" to the terminal. This is just a message to indicate that the following command will compare the files named `huge` and `newhuge`.
5. `sudo diff /mnt/huge /mnt/newhuge`: This command compares the files named `huge` and `newhuge` that are located in the `/mnt` directory. The `diff` command compares the contents of two files line by line and reports the differences between them.
6. `sudo umount /mnt`: This command unmounts the file system that was mounted at `/mnt`. This ensures that any changes made to the files are properly saved before the device is disconnected or removed.

Sharecount = To see if other process's are using the inode
Cachecount = Thisis how many many times we access the inode while its in memory
Cachelist = the in memory list where all the minodes are