File System Management (using inodes)

Your Task: Simulate the kernel's management of the file system using inodes.

Disk Facts:

- Total Disk Size = 65,536 bytes
- The disk is divided into 128 Blocks (512 bytes for each block)
- The disk is simulated using file disk.txt. Open this text file to see the contents of the disk

System Calls: Our mini-OS implements 4 custom system calls that provide access to our file system. In our system, each system call is a 4 character command.

- **CREATE (CRET):** Create a file in our file system. The name of the file is mapped in our the directory nodes data block, along with the corresponding starting inode block number of the file.
- **COPY (COPY):** Copies a text file that lives outside our file system and copies it into our file system.
- **DELETE (DELT):** Removes the file from the file system. The name of the file is removed from the directory nodes data block and all blocks used by this file are freed.
- **READ (READ):** Traverses the inode(s) of a file and outputs the contents of the file to standard output.

The CRET and COPY system calls are already implemented and provided in the sample code. Specifically, your task is to implement the READ and DELT system calls. You can use the CRET and COPY implementations as helpful examples.

Assumptions - To simplify the implementation, please make the following assumptions:

- Our file system supports only a single-level directory system. In other words, there are no directories (except the one root directory). In other words, do not worry about implementing multiple directories
- All file names in our file system must be exactly 8 characters.
- Our file system uses a single file (disk.txt) to simulate the contents of disk. To keep it organized and easy to read, assume:
    - Each block has exactly 16 rows, with 32 bytes per row.
    - The first row is a divider (example: ----- Block #4 -----)
    - The last byte in each row is an end-of-line character

SAMPLE CODE: There is a significant amount of sample code provided in this assignment. To help you begin the assignment, we will engage in a thorough code walk-through of the sample code during one (or more) class sessions. If you need more assistance, I highly recommend that you attend the class lab sessions.

SAMPLE CODE AVAILABLE HERE

Sample Output: If you implemented your program correctly, your output for the three included test cases should be similar to the following:

Test File: test.data **Console Output:** console_test.txt **Disk Contents:** disk_test.txt	**Test File:** test2.data **Console Output:** console_test2.txt **Disk Contents:** disk_test2.txt	**Test File:** test3.data **Console Output:** console_test3.txt **Disk Contents:** disk_test3.txt

Submission Instructions:

- Submit the code and Makefile on Blackboard inside a single .tar ball
- Your tar ball should be well organized (type *make clean*) to clear any extra files
- Make sure to test that you are submitting the correct tar ball by extracting it and checking the contents
    - Create a directory: mkdir tmp (so you don't overwrite your existing folder)
    - Copy the tar file into the directory cp filename.tar tmp/
    - Extract the tar file "tar –xvf filename.tar"

Rubric:

Properly Submitted: Code submitted correctly. Contains Makefile and the required programs contained in a single .tar file on Blackboard.	**10 points**
Organization & Error Checking: Code is well organized, contains reasonable error checking and useful comments	**10 points**
Functionality (DELETE System Call) - Full credit for working correctly across all test cases.	**40 points**
Functionality (READ System Call) - Full credit for working correctly across all test cases.	**40 points**
TOTAL	**100 Points**

*** Submission must compile using the MakeFile. Submissions that do not compile will receive a grade of a zero.*