Design Idea:

Vanilla File System – MP7:

I have implemented the following:

- Vanilla File System

- Option 1: Design of an extended file system for files up to 64kB long

Vanilla File System:

A very simple file system is implemented with the following assumptions as mentioned in the handout:

- The file system can only manage single-level directory.
- Length of any file is at most one block. (For the compulsory portion)
- File operations are separate from the File System functions

1. **Inode:**
   - id - points to the identifier of the file it holds information about
   - blk_number - block in the file system where the file pointed by the inode is stored
   - inode_is_free - checks if the current inode available or holding any file
   - fle_size - size of file pointed by the inode
   - fs - pointer to the file system

2. **File:**

Variables

- fle_system – pointer to the file system
- fle_identifier – file identifier
- blk_number – block in the file system where the file is stored
- inode_indx – position in inode array assigned to file
- fle_size – size of file
- curr_pos – current position pointed to in file (for read and write operations)
- block_cache – buffer for file, acting as a cache, of size 512 bytes

Functions:

- File(_fs, _id) - Opens the file with file id _id using the file system _fs. Sets the 'curr_pos' to the beginning of the file.
- ~File() - Closes the currently open file and deletes data structures associated it.
- Read(unsigned int _n, char * _buf) – Reads _n characters from the file/block_cache into the buffer _buf, unless EOF is reached and returns actual characters read from file.

- Write(unsigned int _n, char * _buf) – Writes _n characters from the file/block_cache into the buffer _buf, unless EOF is reached and returns actual characters written to file.
- Reset() - Sets the 'curr_pos' to the beginning of the file.
- EoF() – Checks if end of file is reached by 'curr_pos'.

3. FileSystem:

Variables

- free_blk_cnt – count of free blocks in the file system
- inode_cntr – count of free inodes in the file system
- free_blocks –pointer to bitmap of free blocks in the file system. 'F' represents a free block and 'U' represents a used block
- disk – pointer to the SimpleDisk ecosystem
- inodes – pointer to array of inodes in the file system

Functions:

- FileSystem() – Initialized the File System local data structures and inode.
- ~FileSystem() – Writes the local data structures and inode to the disk and unmounts the FileSystem. The first two blocks are reserved for storing 'free_blocks' and 'inodes' respectively.
- Mount(disk) – Mounts the file system from the disk and loads the data structures and inode.
- Format(disk, _size) – Formats the entire disk (of _size) and re-initiates the local data structures and inode.
- LookupFile(_file_id) – If the file with identifier _file_id is found in the file system, returns its associated inode otherwise throws an exception.
- CreateFile(_file_id) – Creates a new file in the system with identifier _file_id by initializing a new inode and assigning a free block.
- DeleteFile(_file_id) – Deletes the file identified by _file_id and its data structures. Also, removes the associated inode and free the block it was stored at.

Test Screenshots / Output:

Option 1: Design of an extended file system for files up to 64kB long

To accommodate longer files, 1 block per file is not enough. For files 64kB long, we need 64kB/512b = 12 blocks. So instead of just storing the 'blk_number', we now maintain a blocks list in the inode and file, that stores all the blocks where the file is stored orderly.

```
unsigned int *blocks
```

Example: To store files of size 1.5kB, we need 3 blocks. Suppose they are stored at block numbers 23, 29, 58. Then the array 'blocks' contain [23,29,58].

To cope with multiple blocks, following changes need to be done:

- **Inode:**
  - Replace 'blk_number' with 'blocks' to store all the blocks where the file is stored orderly.

- **File:**
  - Replace 'blk_number' with 'blocks' to store all the blocks where the file is stored orderly.

- Read – If the end of the current block is reached when reading from the file, go to the next block in the 'blocks' list and continue reading unless last block is reached.
- Write - If the end of the current block is reached when writing to the file, assign a new free block, if file size is under 64kB, and add it to the 'blocks' list. Then, continue writing to the file.
- EOF- return true if the end of the last block in 'blocks' list is reached, otherwise false.

- **FileSystem:**
  - Format – When formatting the disk, all the 'blocks' allocated to the file is freed.
  - CreateFile – Instead of storing the 'blk_number', the newly assigned free block is added to the 'block' list.
  - DeleteFile – Apart from removing the data structures associated to the file, all the 'blocks' allocated to the file is also freed.

Documents changed:

I have changed the following files:

- **file.C**
- **file.H**
- **file_system.C**
- **file_system.H**