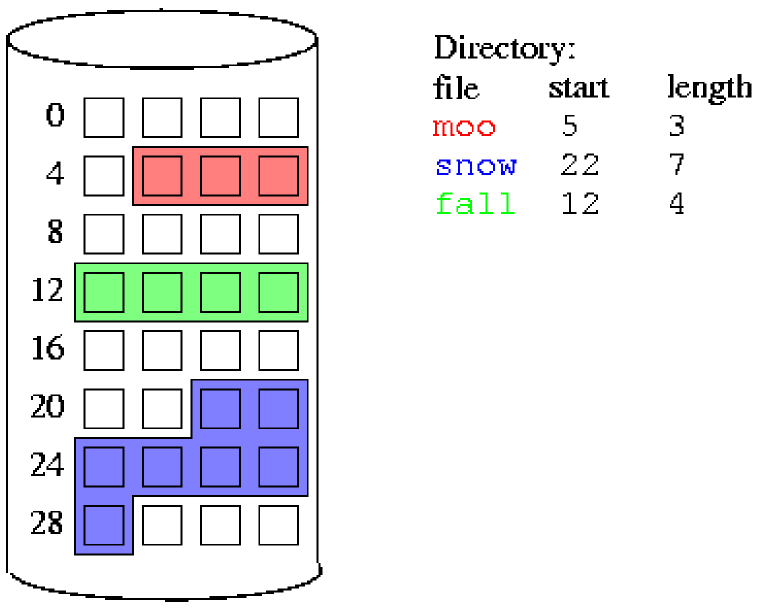
Project 4 | CPE 453
Last Modified: Feb., 2025
You may work in groups of **four** for this assignment

TinyFS and disk emulator

For this assignment, you'll be implementing TinyFS (the tiny file system), mounted on a single Unix file that emulates a block device.

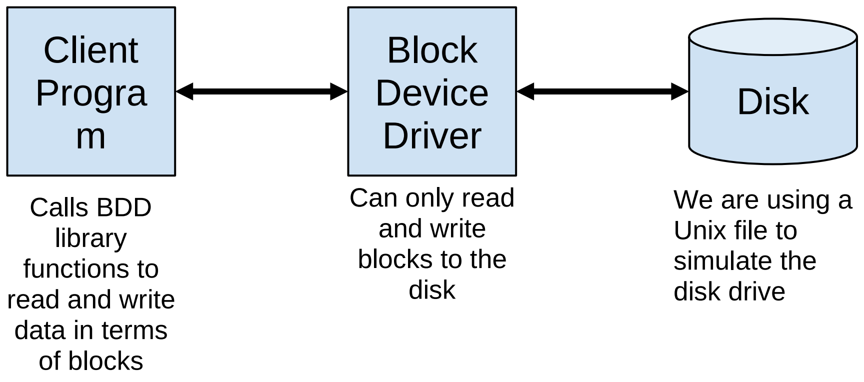


Directory:

file	start	length
moo	5	3
snow	22	7
fall	12	4

Objective

The goal of this assignment is to gain experience with the fundamental operations of a file system. File systems are not only integral to operating systems, but they incorporate many aspects of OS concepts such as message passing, fault tolerance, scheduling, resource management and concurrency.


Client Progra m
Block Device Driver
Disk
Calls BDD library functions to read and write data in terms of blocks
Can only read and write blocks to the disk
We are using a Unix file to simulate the disk drive

Phase I: Disk Emulator

The first part of the assignment is to build a disk emulator, or more generically a “block device emulator”. You will implement an emulator that will accomplish basic block operations, like the kind supported by block devices (e.g. hard disk drives), on a regular Unix file.

libDisk interface functions

The emulator is a library of three functions that operate on a regular UNIX file. The necessary functions are: **`openDisk()`**, **`readBlock()`**, **`writeBlock()`**. There is also a single piece of data that is required: `BLOCKSIZE`, the size of a disk block in bytes. This should be statically defined to 256 bytes using a macro (see below). All IO done to the emulated disk must be block aligned to `BLOCKSIZE`, meaning that the disk assumes the buffers passed in **`readBlock()`** and **`writeBlock()`** are exactly `BLOCKSIZE` bytes large. If they are not, the behavior is undefined.

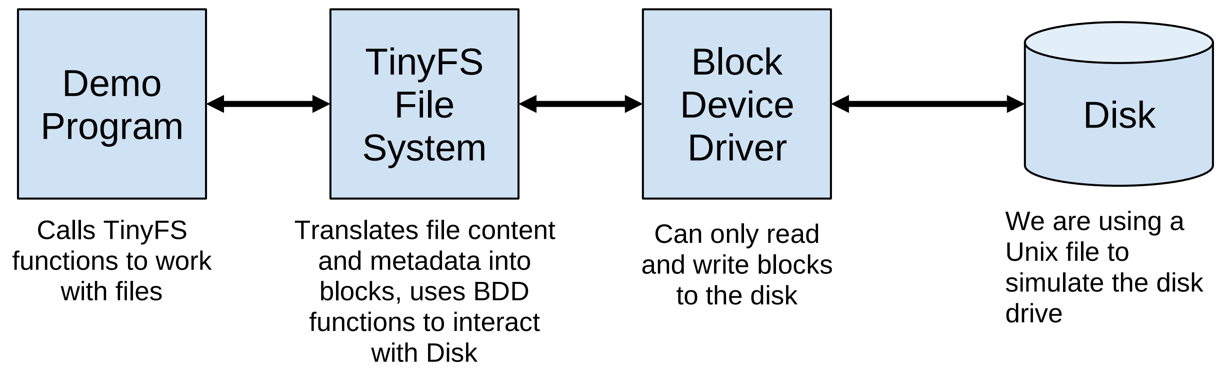
```
int openDisk(char *filename, int nBytes);
/* This functions opens a regular UNIX file and designates the first nBytes of it as space for the emulated disk. If nBytes is not exactly a multiple of BLOCKSIZE then the disk size will be the closest multiple of BLOCKSIZE that is lower than nByte (but greater than 0) If nBytes is less than BLOCKSIZE failure should be returned. If nBytes > BLOCKSIZE and there is already a file by the given filename, that file’s content may be overwritten. If nBytes is 0, an existing disk is opened, and the content must not be overwritten in this function. There is no requirement to maintain integrity of any file content beyond nBytes. The return value is negative on failure or a disk number on success. */

int closeDisk(int disk); /* self explanatory */

int readBlock(int disk, int bNum, void *block);
/* readBlock() reads an entire block of BLOCKSIZE bytes from the open disk (identified by ‘disk’) and copies the result into a local buffer (must be at least of BLOCKSIZE bytes). The bNum is a logical block number, which must be translated into a byte offset within the disk. The translation from logical to physical block is straightforward: bNum=0 is the very first byte of the file. bNum=1 is BLOCKSIZE bytes into the disk, bNum=n is n*BLOCKSIZE bytes into the disk. On success, it returns 0. -1 or smaller is returned if disk is not available (hasn’t been opened) or any other failures. You must define your own error code system. */

int writeBlock(int disk, int bNum, void *block);
/* writeBlock() takes disk number ‘disk’ and logical block number ‘bNum’ and writes the content of the buffer ‘block’ to that location. ‘block’ must be integral with BLOCKSIZE. Just as in readBlock(), writeBlock()
```

```
must translate the logical block bNum to the correct byte position in
the file. On success, it returns 0. -1 or smaller is returned if disk
is not available (i.e. hasn't been opened) or any other failures. You
must define your own error code system. */
```


Demo Program
TinyFS File System
Block Device Driver
Disk
Calls TinyFS functions to work with files
Translates file content and metadata into blocks, uses BDD functions to interact with Disk
Can only read and write blocks to the disk
We are using a Unix file to simulate the disk drive

Phase II: TinyFS file system implementation

TinyFS is a very simple file system. It is purposefully under-specified, giving you the freedom to implement it using many of the algorithms and primitives you've learned throughout this course. TinyFS does not support a hierarchical namespace, i.e. there are no directories beyond the root directory, and all the files are in a flat namespace.

Block Types

The disk blocks of TinyFS may be any of these types:

Block name	Block code	Description (not exhaustive)	number possible	size (bytes)
superblock	1	must contain the magic number, pointer to root inode, and the free block-list implementation	1	256
inode	2	must contain the name of the file, the file size and a data block indexing implementation	many	256
file extent	3	contains file content, possibly also block# of the inode block	many	256
free	4	is ready for future writes	many	256

1. superblock

The superblock stores metadata about the file system and is always stored at logical block 0. The block contains three different pieces of information. 1) It specifies the “magic number,” used for detecting when the disk is not of the correct format. For TinyFS, that number is 0x44, and it is to be found exactly on the second byte of every block. 2) It contains the block number of the root inode (for directory-based file systems). 3) It contains a pointer to the list of free blocks, or some other way to manage free blocks. How you implement the free block list is up to you, but it might be done, for example, by having a pointer to the first free block in a chain of free blocks, or by implementing a bit vector and storing it directly within the superblock.

2. inode

An inode block keeps tracks of metadata for each file within TinyFS. In real file systems, this is typically ownership (user, group), file type, creation time, access time, *etc*. For TinyFS core only the file’s name and size is required. You must support names up to 8 alphanumeric characters (not including a NULL terminator), and no longer. For example: “file1234”, “file1” or “f”.

For inode blocks, you must design where and how to store the file metadata. This includes how you index the data blocks (file extents) that correspond to the inode. Here again, there are many possible implementations, including a linked list of blocks, direct indexing, multi-level indexing, or even content-based addressing (see additional features below).

3. file extent

A file extent block is a fixed sized block that contains file data and (optionally) a pointer to the next data block. If the file extent is the last (or only) block, the remaining bytes and the pointer should be set to 0x00.

4. free block

Free blocks are empty and available for writing. But just as any other block, they have to have the required bytes 0 and 1 (see below). Once again, you have many options for managing your free block list. For example, you may choose to use the link field (found at byte 2) to form a chain of free blocks.

Block storage format

The following bytes are defined for all blocks: Bytes 0 and 1 must be formatted as specified. A suggestion for bytes 2 has been made for you to use it to store the address of another block which acts like a link in a chain, but is optional. If you do not use it as a link, you must leave it blank. Block 3 must be left blank as well.

Byte	Content	Byte	Content
0	[block type = 1|2|3|4]	1	0x44
2	[address of another block?]	3	[empty]
4	[data starts]	5	[second byte of data]
6	[third byte of data]	7	...

TinyFS interface functions:

Nine API functions are needed to implement the TinyFS interface.

```
int tfs_mkfs(char *filename, int nBytes);
/* Makes a blank TinyFS file system of size nBytes on the unix file
specified by 'filename'. This function should use the emulated disk
library to open the specified unix file, and upon success, format the
file to be a mountable disk. This includes initializing all data to 0x00,
setting magic numbers, initializing and writing the superblock and
inodes, etc. Must return a specified success/error code. */

int tfs_mount(char *diskname);
int tfs_unmount(void);
/* tfs_mount(char *diskname) "mounts" a TinyFS file system located within
'diskname'. tfs_unmount(void) "unmounts" the currently mounted file
system. As part of the mount operation, tfs_mount should verify the file
system is the correct type. In tinyFS, only one file system may be
mounted at a time. Use tfs_unmount to cleanly unmount the currently
mounted file system. Must return a specified success/error code. */

fileDescriptor tfs_openFile(char *name);
/* Creates or Opens a file for reading and writing on the currently
mounted file system. Creates a dynamic resource table entry for the file,
and returns a file descriptor (integer) that can be used to reference
this entry while the filesystem is mounted. */

int tfs_closeFile(fileDescriptor FD);
/* Closes the file, de-allocates all system resources, and removes table
entry */

int tfs_writeFile(fileDescriptor FD,char *buffer, int size);
/* Writes buffer 'buffer' of size 'size', which represents an entire
file's content, to the file system. Previous content (if any) will be
```

```
completely lost. Sets the file pointer to 0 (the start of file) when
done. Returns success/error codes. */

int tfs_deleteFile(fileDescriptor FD);
/* deletes a file and marks its blocks as free on disk. */

int tfs_readByte(fileDescriptor FD, char *buffer);
/* reads one byte from the file and copies it to buffer, using the
current file pointer location and incrementing it by one upon success.
If the file pointer is already past the end of the file then
tfs_readByte() should return an error and not increment the file pointer.
*/

int tfs_seek(fileDescriptor FD, int offset);
/* change the file pointer location to offset (absolute). Returns
success/error codes.*/
```

In your `tinyFS.h` file, you must also include the following definitions:

```
/* The default size of the disk and file system block */
#define BLOCKSIZE 256
/* Your program should use a 10240 Byte disk size giving you 40 blocks
total. This is a default size. You must be able to support different
possible values */
#define DEFAULT_DISK_SIZE 10240
/* use this name for a default emulated disk file name */
#define DEFAULT_DISK_NAME "tinyFSDisk"
/* use as a special type to keep track of files */
typedef int fileDescriptor;
```

Error Codes

You must specify a set of unified error codes returned by your TinyFS interfaces. ***All error codes must be negative integers*** (-1 or lower), but it is up to you to assign specific meaning to each. Error codes must be informational only, and not used as status in subsequent conditionals. Create a file called `tinyFS_errno.h` and implement the codes as a set of statically defined macros. Take a look at `man 3 errno` on the UNIX* machines for examples of the types of errors you might catch and report.

Assignment & Additional Features

- Implement the core interface functions above (70%).
- Add additional areas of functionality from the list (a-h) below (30%). Note that some features count as two. You are free to implement the features in your own way, so be creative, but understand they must be demoed in your final program. Feel free to do a little research, and base your design decisions on existing solutions.
    a. Fragmentation info and defragmentation (10%)
        - implement `tfs_displayFragments()` /* this function allows the user to see a map of all blocks with the non-free blocks clearly designated. You can return this as a linked list or a bit map which you can use to display the map with */
        - implement `tfs_defrag()` /* moves blocks such that all free blocks are contiguous at the end of the disk. This should be verifiable with the tfs_displayFraments() function */
    b. Directory listing and file renaming (10%)
        - `tfs_rename(fileDescriptor FD, char* newName)` /* renames a file. New name should be passed in. File has to be open. */
        - `tfs_readdir()` /* lists all the files and directories on the disk, print the list to stdout -- Note: if you don't have hierarchical directories, this just reads the root directory aka "all files" */
    c. Hierarchical directories (20%)
        - Support hierarchical directories by creating a root directory link in the superblock and re-designing the inode block, so that it can indicate a directory (same name requirements as a file). Use absolute paths for all files and directories.
            - discuss your design and implementation
        - `tfs_createDir(char *dirName)` /* creates a directory, name could contain a "/"-delimited path) */
        - `tfs_removeDir(char *dirName)` /* deletes empty directory */
        - `tfs_removeAll(char *dirName)` /* recursively remove dirName and any file and directories under it. Special "/" token may be used to indicate root dir. */
        - `tfs_openFile()`: modify so that it supports directories as part of the file name. Return an error if any directory in the path does not exist. It must maintain backwards compatibility.

    d. Read-only and writeByte support (10%)
        - implement the ability to designate a file as "read only". By default all files are "read write" (RW).
        - `tfs_makeRO(char *name)` /* makes the file read only. If a file is RO, all tfs_write() and tfs_deleteFile() functions that try to use it fail. */
        - `tfs_makeRW(char *name)` /* makes the file read-write */
        - `tfs_writeByte(fileDescriptor FD, int offset, unsigned`

`int data)`, a function that can write one byte to an exact position inside the file.

- `tfs_writeByte(fileDescriptor FD, unsigned int data)` is also acceptable. (uses current file pointer instead of offset).

e. Timestamps (10%)
   - implement creation, modification and access timestamps for each file to be stored in the inode block
   - `tfs_readFileInfo(fileDescriptor FD)` /* returns the file's creation time or all info (up to you if you want to make multiple functions) */
   - return format is up to you

f. Implement content-based address (20%)
   - Instead of addressing data blocks by their offsets, address them by their content
   - In this way, identical blocks will be shared between files, reducing the total number of data block necessary

g. Implement full-disk encryption (20%)
   - All data and metadata should be encrypted using a semantically secure block cipher in a sensible mode of operation (e.g. CTR, CBC, or better XTS). This page on disk encryption theory may be helpful.
   - You may (and should) use an pre-existing cryptography library, such as OpenSSL for your core cryptographic operations.
   - The key and data in memory may be unencrypted, but at rest data (data on disk) should always be encrypted.
   - The key used to encrypt/decrypt data should be derived from a password given when you format the file system; you must use a secure key derivation function (*e.g.* PBKDFv2, scrypt or bcrypt).
   - Modify `tfs_mkfs()` to accept a password, and format and encrypt an initial TinyFS image.
   - `tfs_mount()` should also be modified to take in the user's password, "unlocking" the file system; all other TinyFS interfaces should remain unaltered.
   - Discuss your design decisions and threat model (*i.e.* what attacks your system is strong and weak against.)

h. Implement file system consistency checks (10%)
   - Upon mount, verify the entire file system is in a consistent state, and fail to mount and report an error if it is not.
   - An inconsistent file system might include:
     - blocks on both the free list and allocated to an inode
     - data blocks that are not on the free list of allocated to an inode
     - blocks that have been corrupted due to latent disk failures
     - Other scenarios you should think of and test.
   - See this set of lecture notes for more information on file system consistency
   - Discuss the types of inconsistency your file system detects.

- Write a demo program that includes your TinyFS interface to demonstrate the basic functionality of the required functions and your chosen additional functionality. You can display informative messages to the screen for the user to see how you demonstrate these.
- Implementation: **This program must be implemented in C.**
- EXTRA CREDIT: Extra credit points, which will count towards your total programming assignment score, may be awarded for completing up to two additional features. Each single feature will contribute up to 10 points, with those features that "count as 2," up to 20 points. No more than 20 points, in total, will be awarded for extra credit.

File Resources

You can get these on unix machines:
`cp ~/foaad/www/class/453/[FILE] .`
https://users.csc.calpoly.edu/~foaad/class/453/tfs_makefile (basic Makefile)
https://users.csc.calpoly.edu/~foaad/class/453/diskTest.c (basic disk test, only a very basic test)
https://users.csc.calpoly.edu/~foaad/class/453/tfsTest.c (very very basic TinyFS test)

Deliverables

As usual, submit a **tar.gz** archive via PolyLearn with the following:

- all source files: .c, .h (at least three separate source files)
    - emulator file (libDisk)
    - tinyFS interface file (libTinyFS). This file will access libDisk for disk emulator functionality
    - tinyFSDemo driver file that contains a main(), and includes libTinyFS headers (but not libDisk)
- A Makefile that compiles all the libraries and makes the following executable:
    - tinyFSDemo
- a README with:
    - First line should start with the word "FEATURES: ", followed by a series of capital letters back to back in alphabetical, each corresponding to one of the "additional features" above. If you did not do any additional features, then just leave that line as "FEATURES: ", if you did for example features (a) and (d), then the first line of your README should be "FEATURES: AD". If you did (e) and (f), then "FEATURES: EF"
    - Names of all partners
    - An explanation of how well your TinyFS implementation works, including tradeoffs you made and why
    - An explanation of which additional functionality areas you have chosen and how you have shown that it works.
    - Any limitations or bugs your file system has.
- Fill out the survey on Canvas
- Submit a short demo video showing all the features of your project