CIS 345/545 Project 3 Fall 2020

(Due Dec. 1)

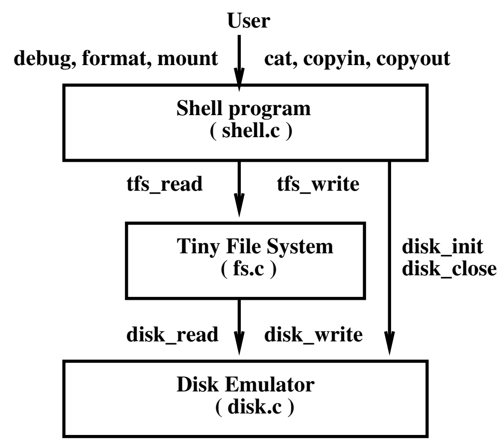
In this project, you will implement a tiny file system on top of a virtual disk. This file system is called TinyFS which is similar to the Unix file system discussed in the class. This virtual disk is actually a single 8MB file that is stored on the real file system in our lab.

To uncompress and extract the files to your working directory, login a Linux workstation in our lab and then type

```
tar xvfz ~cis345s/pub/tinyfs.tar.gz
```

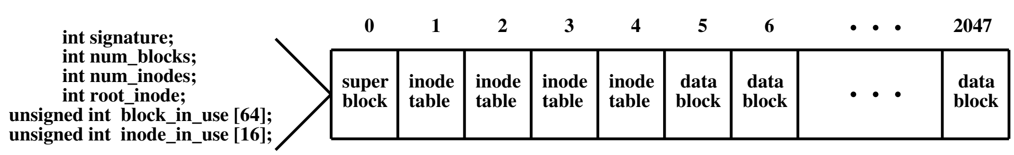
Overview

The TinyFS has three major components: the shell, the file system itself, and the emulated disk, as shown to the right. Your task is to implement the middle component: the file system.



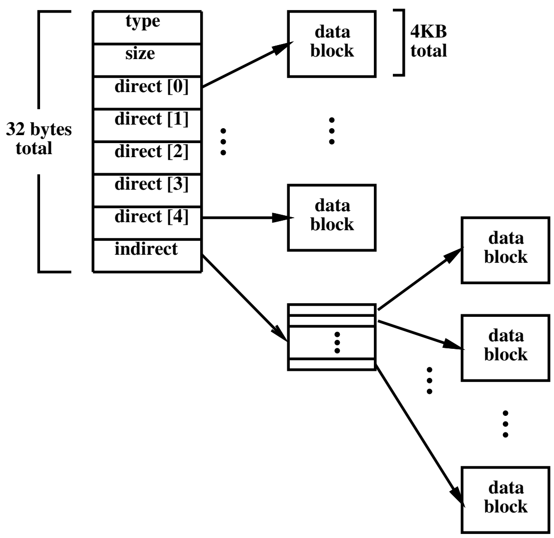
At the top level a user gives typed commands to a shell, instructing it to format or mount a virtual disk, and to copy data in and out of the file system. The shell converts these typed commands into high-level operations on the file system, such as `tfs_format`, `tfs_mount`, `tfs_read`, and `tfs_write`. The file system is responsible for accepting these operations on files and converting them into simple block read and write operations on the emulated disk, called `disk_read`, and `disk_write`. The emulated disk, in turn, stores all of its data in an image file in the file system.

TinyFS has the following layout on disk. It assumes that the disk blocks are the common size of 4KB. The first block of the disk (i.e. disk block zero) is a "superblock" that describes the layout of the rest of the file system. The two blocks following the superblock contain inode data structures. The remaining blocks in the file system are used as data blocks which contain either file data, directory entries, or occasionally indirect pointers. Below is a picture of a TinyFS image:



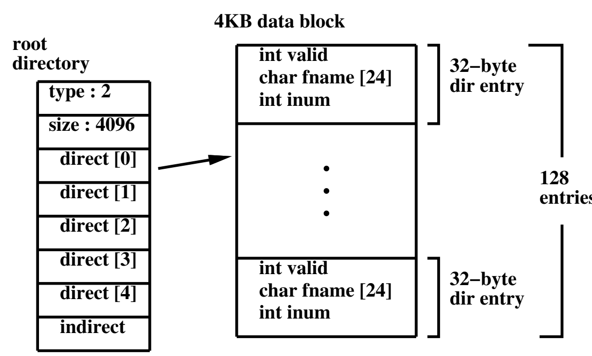
The superblock describes the layout of the rest of the file system. Each of the first four fields of the superblock is a 4-byte (32-bit) integer. The first field `signature` is always the magic number TFS_MAGIC (0x345f2020). The format routine places this number into the very first bytes of the superblock. When the file system is mounted, the OS looks for this signature value. If it is correct, then the disk is assumed to contain a valid file system. If some other number is present, then the mount fails, perhaps because the disk is not formatted or contains some other kind of data.

The remaining fields in the superblock describe the layout of the file system. `int num_blocks` is the total number of blocks on the disk. To make things easier, we use a fixed disk size 8MB in this project. So the total number of blocks is always 2048. We also assume the total number of inodes `num_inodes` is 512. Furthermore, `root_inode` contains the inode number (i.e. 1 in this project) of the root directory. Note that the inode number 0 is reserved as NULL. The arrays `block_in_use` and `inode_in_use` are the bitmaps of the blocks and inodes being used, respectively. The format routine is responsible for setting the four fields and the two bitmaps correctly. The remainder of the superblock is left unused.



The left picture shows an inode entry in TinyFS. Each field of the inode is a 4-byte (32-bit) integer. The `type` field is 1 (i.e. the pre-defined constant REGULAR) if the inode is used for a regular file and is 2 (i.e. the pre-defined constant DIR) if it is used for a directory. Note that all files in TinyFS are stored in a single root directory. Hence, only the root inode has the `type` set to be 2. The `size` field contains the logical size of the data in bytes. For the root inode, its size is set to be one data block size 4096 (in this project). There are 5 `direct` pointers to data blocks, and one pointer to an `indirect` data block. In this context, ”pointer” simply means the number of a block where data may be found. A value of zero may be used to indicate a NULL block pointer. Each inode occupies 32 bytes, so there are 128 inodes in each 4KB inode block. Note that an indirect data block is just a big array of pointers to further data blocks. Each pointer is a 4-byte integer, and each block is 4KB, so there are 1024 pointers per block.

The right figure shows the structure of the root inode and its directory entries. Because the file system is very small, the root directory only needs one data block to store directory entries. Each directory entry is 32-byte and there are 128 entries in a block. There are three fields in a directory entry. The `valid` is 1 if the entry is valid (i.e. being used) and is 0 otherwise. For a valid entry, the `fname` field contains the file name and the `inum` contains the file’s corresponding inode number.



Disk Emulator

We provide you with a disk emulator on which to store your file system. As mentioned before, this virtual disk is actually stored as one big file in the file system, so that you can save data in a ”disk image” and then retrieve it later. In addition, we will provide you with some sample disk images that you can experiment with to test your file system. Just like a real disk, the emulator only allows operations on entire disk blocks of 4KB (DISK_BLOCK_SIZE). You cannot read or write any smaller unit than that. The primary challenge of building a file system is converting the user’s requested operations on arbitrary amounts of data into operations on fixed block sizes.

The interface to the simulated disk is given in `disk.h`:

```
#define DISK_BLOCK_SIZE 4096
int  disk_init( const char *filename, int nblocks );
void disk_read( int blocknum, char *data );
void disk_write( int blocknum, const char *data );
void disk_close();
```

Before performing any sort of operation on the disk, you must call `disk_init` and specify a (real) disk image for storing the disk data, and the number of blocks (i.e. 2048 in this project) in the simulated disk. If this function is called on a disk image that already exists, the contained data will not be changed. When you are done using the disk, call `disk_close` to release the file. These two calls are already made for you in the shell, so you should not have to change them. Furthermore, as the names suggest, the functions `disk_read` and `disk_write` read and write one block of data on the disk. Notice that the first argument is a block number, so a call to `disk_read(0,data)` reads the first 4KB of data on the disk, and `disk_read(1,data)` reads the next 4KB block of data on the disk. Every time that you invoke a read or a write, you must ensure that data points to a full 4KB of memory.

File System

Using the existing simulated disk, you will build a working file system. Take note that we have already constructed the interface to the file system and provided some skeleton code. The interface is given in `fs.h`:

```
void tfs_debug();
int  tfs_format();
int  tfs_mount();
int  tfs_create(const char *);
int  tfs_delete(const char * );
int  tfs_getsize(const char *);
int  tfs_get_inumber(const char *);
int  tfs_read( int inumber, char *data, int length, int offset );
int  tfs_write( int inumber, const char *data, int length, int offset );
```

The various functions must work as follows:

- `tfs_debug` - This function scans a mounted file system and reports on how the inodes and blocks are organized. Once you are able to scan and report upon the file system structures, the rest is easy. Your output from `tfs_debug` should be similar to the following:

```
superblock:
     signature is valid
     29 blocks in use
     4 inodes in use
root inode 1:
     size: 4096 bytes
     direct blocks: 3
foo inode 2:
     size: 13 bytes
     direct blocks: 4
bar inode 150:
     size 81929 bytes
     direct blocks: 10 11 15 17 18
     indirect block: 23
     indirect data blocks: 30 39 42 ...
```

- `tfs_format` - Creates a new file system on the disk, destroying any data already present. Sets four blocks for inodes, clears the inode table, and writes the super block. Returns one on success, zero otherwise. Note that formatting a files system does not cause it to be mounted. Also, an attempt to format an already-mounted disk should do nothing and return failure.

- `tfs_mount` - This function examines the disk for a file system. If one is present, read the superblock, and prepare the file system for use. Return one on success, zero otherwise. Note that a successful mount is a pre-requisite for the remaining calls.

- `tfs_create` - This function creates a new regular file of zero length. The filename is passed through the parameter. It finds a free inode through the inode map and then stores a new directory entry under the root. On success, return the (positive) inode number. On failure, return zero. (Note that this implies zero cannot be a valid inode number.)

- `tfs_delete` - It deletes the file specified by the parameter. Release all data and indirect blocks assigned to this inode and return them to the free block map. On success, return (positive) inode number. On failure, return 0.

- `tfs_getsize` - It returns the logical size of the given file, in bytes. Note that zero is a valid logical size for a file. On failure, return -1.

- `tfs_get_inumber` - This function searches the root directory entries and returns the inode number of the given file. On failure, return 0.

- `tfs_read` - This function reads data from the specified inode number `inumber`. That is, the user who calls this function needs to get the inode number via `tfs_get_inumber` first. This function copies `"length"` bytes from the inode into the `"data"` pointer, starting at `"offset"` in the inode. The total number of bytes read will be returned. The number of bytes actually read could be smaller than the number of bytes requested, perhaps if the end of the file is reached. If the given `inumber` is invalid, or any other error is encountered, return 0.

- `tfs_write` - It writes data to a valid inode. Same as `tfs_read`, the user who calls this function needs to get the inode number via `tfs_get_inumber` first. This function copies `"length"` bytes from the pointer `"data"` into the inode starting at `"offset"` bytes. It will allocate any necessary direct and indirect blocks in the process. Afterwards, it returns the number of bytes actually written. If the given inumber is invalid, or any other error is encountered, return 0.

It's quite likely that the file system module will need a number of global variables in order to keep track of the currently mounted file system. For example, you will certainly need a global variable to keep track of the current free block bitmap, and perhaps other items as well. In the context of operating systems, they are common and quite normal. Furthermore, the file system has to maintain state persistently. You need to write all modifications, such as the inode block, superblock, etc., back to disk.

Implementation Notes

Your job is to implement TinyFS as described above by filling in the implementation of the file `fs.c`. You do not need to change any other code modules. We have already created some data structures to get you started. These can be found in `fs.c`.

Note that a raw 4 KB disk block can be used to represent five different kinds of data: a superblock, a block of 128 inodes, a directory of 128 entries, an indirect pointer block, or a plain data block. This presents a bit of a software engineering problem: how do we transform the raw data returned by `disk_read` into each of the five kinds of data blocks?

C provides a nice bit of syntax for exactly this problem. We can declare a union of each of our five different data types. A union looks like a struct, but forces all of its elements to share the same memory space. You can think of a union as several different types, all overlaid on top of each other.

```
union tfs_block {
   struct tfs_superblock super;
   struct tfs_inode inode[INODES_PER_BLOCK];
   struct tfs_dir_entry dentry[NUM_DENTRIES_PER_BLOCK];
   int pointers[POINTERS_PER_BLOCK];
   char data[DISK_BLOCK_SIZE];
};
```

Note that the size of a `tfs_block` union will be exactly 4KB: the size of the largest members of the union. To declare a variable of type:

```
union tfs_block block:
```

Now, we may use `disk_read` to load in the raw data from block zero. We give `disk_read` the variable `block.data`, which looks like an array of characters:

```
disk_read(0,block.data);
```

But, we may interpret that data as if it were a struct superblock by accessing the super part of the union. For example, to extract the signature of the super block, we might do this:

```
x = block.super.signature;
```

To check the block `k` is being used or not, we can check its bitmap:

```
if(block.super.block_in_use[k/32] & (1 << (k%32)) )
```

To set the corresponding bitmap of block `k`, we can use

```
block.super.block_in_use[k/32] |= (1 << (k%32));
```

On the other hand, suppose that we wanted to load disk block 49, assume that it is an indirect block, and then examine the 3rd pointer. Again, we would use `disk_read` to load the raw data:

```
disk_read(49,block.data);
```

But then use the pointer part of the union like so:

```
x = block.pointers[3];
```

The union offers a convenient way of viewing the same data from multiple perspectives. When we load data from the disk, it is just a 4 KB raw chunk of data (block.data). But, once loaded, the file system layer knows that this data has some structure. The file system layer can view the same data from another perspective by choosing another field in the union.

Turning it in

Each group (at most two students) has to submit your program and report electronically. The report file `report.pdf` which includes the description of your code, experiences in debugging/testing, etc. should be put under the `TinyFS` directory. The cover page should contain your picture(s), name(s) and the login id you used to turnin the project.

Use the following command on grail (under the `TinyFS` directory:

`make submit`

Start on time and good luck. If you have any questions, send e-mail to `sang@cis.csuohio.edu`.