CS 370 PA4

File Systems

Due: 2nd December 2022

Lead TA: Taimur Hassan Sarmad

Office Hours: Tuesday/Thursday, 1:30 PM to 2:30 PM

Overview

In this assignment, you are going to be creating **Sasti_FileSystem**, a UNIX-like file system, but simplified. Before starting the assignment, it is recommended to go through this reading to get an idea of fundamental file system concepts which describe how data is stored in files, and how files are stored persistently. Some of these ideas (especially the *inode*) are crucial to understand in order to implement this assignment.

For the sake of simplicity, you are NOT required to implement file names, and directories in this assignment.

Components

There are 3 main components to **Sasti_FileSystem**. These are:

1. The Shell: The first component is a simple shell application that allows the user to perform operations on the file system, such as printing debugging information about the file system, formatting a new file system, mounting a file system, creating and deleting files, and copying data in or out of the file system. This component has been provided to you - you just have to interact with it to use the file system.

2. The Sasti_FileSystem itself: The second component takes the operation specified by the user through the shell and performs them on the Sasti_FileSystem *disk image**. This component has to organize the on-disk data structures and perform all the bookkeeping necessary to allow for persistent storage of data. To store the data, it will need to interact with the disk emulator (which has been provided to you) via methods such as:

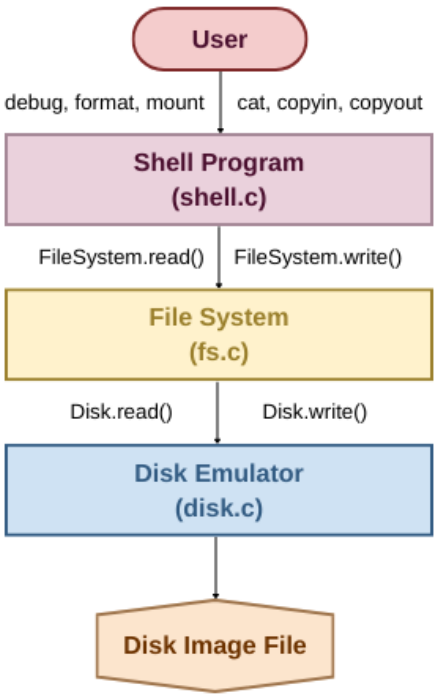
- `disk_read`: This method reads data from the disk image in blocks of size 4096 bytes.
- `disk_write`: This method writes data to the disk image in blocks of size 4096 bytes.

*__NOTE__: The *disk image* is simply a normal file that has been split into blocks of size 4096 bytes. You can think of the disk image as being the hard-drive on which data is stored (in this assignment though, for simplicity, you're not going to be storing data on your actual hard-drive itself, but rather on this 'virtual/emulated' hard-drive).

The skeleton code for this component has been provided to you. You are required to complete this file system interface for this assignment.

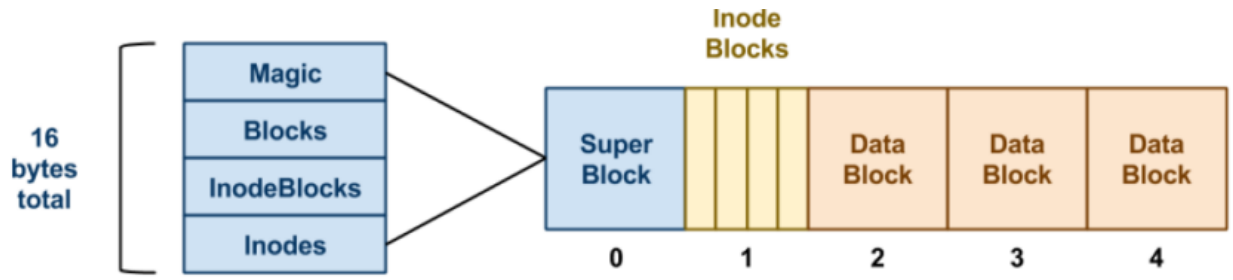
3. The Disk Emulator: Once a normal file has been split into blocks of size 4096 bytes (to make the *disk image*), the third component allows the Sasti_FileSystem to only read/write data in terms of the blocks (using the `disk_read` and `disk_write` methods introduced above). This emulator will persistently store the data to the disk image using the normal `open`, `read` and `write` system calls. This component has been provided to you - you only have to use the APIs provided with it (these APIs will be explained in detail later in this manual).

The diagram below shows how these 3 components work with each other to 'mimic' the actual UNIX file system.



Sasti_FileSystem Design and Layout

A disk image of Sasti_FileSystem has the following layout. The file system assumes that all the disk blocks have a common size of 4KB (4096 Bytes).



An example of a Sasti_FileSystem disk image that begins with a superblock

- The first block of the disk is the **superblock**, which describes the layout of the rest of the file system. The superblock consists of four fields itself:
    1. ***Magic (4 bytes)***: The first field is always the MAGIC_NUMBER or 0xf0f03410. The `format()` routine places this number into the very first bytes of the superblock as a filesystem "signature". When the filesystem is mounted, the OS looks for this magic number. If it is correct, then the disk is assumed to contain a valid filesystem. If some other number is present, then the mount fails, perhaps because the disk is not formatted or contains some other kind of data.
    2. ***Blocks (4 bytes)***: The second field is the total number of blocks, which should be the same as the number of blocks on the disk.
    3. ***InodeBlocks (4 bytes)***: The third field is the number of blocks set aside for storing inodes. The format routine is responsible for choosing this value, which should always be 10% of the Blocks, rounding up, when necessary.

       In this example, there are 4 disk blocks available → 10% of 4 is 0.4 → Rounding up to the nearest integer, this becomes 1 → Hence, there is 1 inode block in this example.

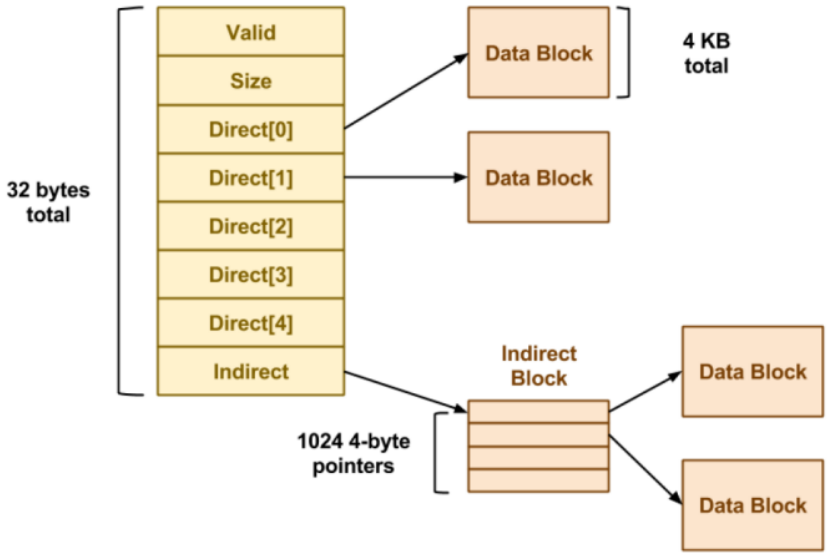
    4. ***Inodes (4 bytes)***: The fourth field is the total number of inodes in the inode blocks.

  Note that the superblock data structure is quite small: only 16 bytes. The remaining (4096 - 16 =) 4080 bytes of disk block zero are left unused.

- Then come the **inode** blocks themselves.

> Before diving into the inodes and inode blocks, it is useful to recall what inodes are exactly. Data is stored on your disk in the form of fixed-size blocks. If you save a file that exceeds a standard block-size, your computer will find the next available segment on which to store the rest of your file. Over time, that can get super confusing. This is where inodes come in. While they don't contain any of the file's actual data, it stores the file's metadata, including all the storage blocks on which the file's data can be found.

A certain number of blocks following the superblock contain just inode data structures. Typically, this number is 10% of the total number of disk blocks (as explained above). Each inode in Sasti_FileSystem has the following structure.



The structure of an inode in Sasti_FileSystem

The fields are explained below.

1. ***Valid (4 bytes)***: This field is 1 if the inode has been created (and is thus valid), and 0 otherwise.
2. ***Size (4 bytes)***: This field stores the logical size of the inode data, in bytes.

3. ***Direct[x] (4 bytes)***: There are a total of 5 direct pointers to data blocks (a data block is just 4KB of raw data). A data block is simply an array of characters which stores the content of the file.

   For the sake of simplicity, you can think of a 'pointer' being the number of a block where the data may be found. A value of 0 may be used to indicate a null block. Each inode occupies 32 bytes, so there are 128 inodes in each 4KB inode block.

4. ***Indirect (4 bytes)***: The Indirect field is basically just an integer that represents the block number (in the disk/disk image) from where a *Data Block* starts. This Data Block, instead of storing the actual data of the file itself, essentially stores a big array of pointers, with each pointer pointing to further data blocks (and these data blocks are the ones which actually contain the file data). Each pointer is a 4-byte int, and each block is 4KB, so there are 1024 pointers per block.

   Why is this even needed? Well, recall that the blocks of data are uniform (fixed size). It is possible that you have a file so big that its contents can't just be stored in the data blocks that are pointed to by the five *Direct* pointers. In this case, the *Indirect* field in the file's inode will point to a data block which will be used to store an array of pointers (again, 'pointers' in this assignment are just block numbers in the disk image) instead an array of characters (which will be used to store the data). Each 'pointer' in this array of pointers will point to an 'actual', good old data block. This provides a good way to store large files.

For this assignment, you CAN assume that a file will never exceed the size of its inode.

- Then come the Data Blocks. As explained above, these can be used for 2 purposes:
  a. To store the contents of the actual file itself in an array of characters.
  b. To store an array of integers that act as pointers to further data blocks,

in case the file is too large cannot entirely be stored through the direct blocks.

- One thing missing in Sasti_FileSystem is the free block bitmap. A real file system would keep a free block bitmap on disk, recording one bit for each block that was available or in use. This bitmap would be consulted and updated every time the file system needed to add or remove a data block from an inode. Since Sasti_FileSystem does not store this on-disk, you are required to keep a free block bitmap in memory. That is, there must be an array of integers, one for each block of the disk, noting whether the block is in use or available.

  When it is necessary to allocate a new block for a file, the system must scan through the array to locate an available block and allocate it. Similarly, when a block is freed, it must be marked in the bitmap. Suppose that the user makes some changes to the Sasti_FileSystem, and then reboots the system (i.e. restarts the shell). Without a free block bitmap, Sasti_FileSystem cannot tell which blocks are in use and which are free. Fortunately, this information can be recovered by scanning the disk. Each time that a Sasti_FileSystem filesystem is mounted, the system must build a new free block bitmap from scratch by scanning through all of the inodes and recording which blocks are in use. This is much like performing an `fsck()` (file system check) every time the system boots.

The Disk Emulator

The Disk Emulator (that stores the file system) has been provided to you. This "disk" is actually stored as one big file in the filesystem, so that you can save data in a disk image and then retrieve it later. In addition, you are provided with some sample disk images that you can experiment with to test your filesystem. Just like a real disk, the emulator only allows operations on entire disk blocks of 4KB. You cannot read or write any smaller unit than that. The primary challenge of building a filesystem 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.

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 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 program, so you should not have to change them.

Once the disk is initialized, you may call `disk_size()` to discover the number of blocks on the disk. As the names suggest, `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.

Implementing the File System

The Sasti_FileSystem looks much like the Unix file system. Each "file" is identified by an integer called an *inumber*. The *inumber* is simply an index into the array of inode structures that starts in block 1. When a file is created, Sasti_FileSysrem chooses the first available *inumber* and returns it to the user. All further references to that file are made using the *inumber*.

Using the disk emulator to build a working file system, you have been provided the skeleton code for the interface of Sasti_FileSystem in ***fs.h*** and ***fs.c***.

You are required to implement the functions as follows:

1. **`fs_format() [5 MARKS]`:** Creates a new filesystem on the disk, destroying any data already present. Sets aside ten percent of the blocks for inodes, clears the inode table, and writes the superblock. Returns 1 on success, 0 otherwise. Furthermore, place the given magic number (FS_MAGIC) in the *magic* field of the superblock. Note that formatting a filesystem does not cause it to be mounted. Also, an attempt to format an already-mounted disk should do nothing and return failure.
2. **`fs_mount() [5 MARKS]`**: Examine the disk for a filesystem. If one is present, read the superblock, build a free block bitmap, and prepare the filesystem for use. Also confirm if the *magic* field of the superblock matches the given magic number (FS_MAGIC). Return one on success, zero otherwise. Note that a successful mount is a prerequisite for the remaining calls.
3. **`fs_debug() [10 MARKS]`**: Scan a mounted filesystem and report on how the inodes and blocks are organized. If you can write this function, you have won half the battle! Once you are able to scan and report upon the file system structures, the rest is easy. Your output from `fs_debug()` should be similar to the following:

```
superblock:
    magic number is valid
    1010 blocks on disk
    101 blocks for inodes
    12928 inodes total
inode 1:
    size: 45 bytes
```

```
    direct blocks: 103 194
inode 2:
    size: 81929 bytes
    direct blocks: 105 109
    indirect block: 210
    indirect data blocks: 211 212 213 214 ...
```

4. **`fs_create() [15 MARKS]`**: Create a new inode of zero length. On success, return the inumber. On failure, return -1.
5. **`fs_delete() [15 MARKS]`**: Delete the inode indicated by the *inumber*. Release all data and indirect blocks assigned to this inode and return them to the free block map. On success, return one. On failure, return 0.
6. **`fs_read() [25 MARKS]`**: Read data from a valid inode. Copy *`length`* bytes from the inode into the *`data`* pointer, starting at *`offset`* in the inode. Return the total number of bytes read. The number of bytes actually read could be smaller than the number of bytes requested, perhaps if the end of the inode is reached. If the given *inumber* is invalid, or any other error is encountered, return 0.
7. **`fs_write() [25 MARKS]`**: Write data to a valid inode. Copy *`length`* bytes from the pointer *`data`* into the inode starting at *`offset`* bytes. Allocate any necessary direct and indirect blocks in the process. Return the number of bytes actually written. The number of bytes actually written could be smaller than the number of bytes requested, perhaps if the disk becomes full. If the given *inumber* is invalid, or any other error is encountered, return 0. Note: the number of bytes actually written could be smaller than the number of bytes requested, perhaps if the disk becomes full.

It's quite likely that the filesystem module will need a number of global variables in order to keep track of the currently mounted filesystem. For example, you will certainly need a global variable to keep track of the current free block bitmap, and perhaps other items as well. You may have heard from other places that global variables are "bad". In the context of operating systems, they are common and quite normal.

Sasti_FileSystem/`fs.c`

Your job is to implement Sasti_FileSystem as described above by filling in the implementation of fs.c. We have already created some sample data structures to get you started. To begin with, we have defined a number of common constants that you will use. Most of these should be self explanatory:

```
#define MAGIC_NUMBER 0xf0f03410
#define INODES_PER_BLOCK 128
#define POINTERS_PER_INODE 5
#define POINTERS_PER_BLOCK 1024
```

Note that `POINTERS_PER_INODE` is the number of direct pointers in each inode structure, while `POINTERS_PER_BLOCK` is the number of pointers to be found in an indirect block.

The superblock and inode structures are easily translated from the pictures above:

```
struct fs_superblock
{
      // Superblock structure
      int32 magic; // File system magic number
      int32 nblocks; // Number of blocks in file system
      int32 ninodeblocks; // Number of blocks reserved for inodes
      int32 ninodes; // Number of inodes in file system
};
```

```
struct fs_inode
{
      int isvalid; // Whether or not inode is valid
      int size; // Size of file
      int direct[POINTERS_PER_INODE]; // Direct pointers
      int indirect; // Indirect pointer
};
```

Note carefully that many inodes can fit in one disk block. A 4KB chunk of memory containing 128 inodes would look like this:

```
fs_inode inode[INODES_PER_BLOCK];
```

Each indirect block is just a big array of 1024 integers, each pointing to another disk block. So, a 4KB chunk of memory corresponding to an indirect block would look like this:

```
int pointers[POINTERS_PER_BLOCK];
```

Finally, each data block is just raw binary data used to store the partial contents of a file. A data block can be defined as an array of 4096 bytes:

```
char data[BLOCK_SIZE];
```

Because a raw 4 KB disk block can be used to represent four different kinds of data: a superblock, a block of 128 inodes, an indirect pointer block, or a plain data block, we can declare a union of each of our four 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 fs_block
{
      struct fs_superblock super; // Superblock
      struct fs_inode inode[INODES_PER_BLOCK]; // Inode block
      int pointers[POINTERS_PER_BLOCK]; // Pointer block
      char data[BLOCK_SIZE]; // Data block
};
```

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

```
fs_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 magic number of the super block, we might do this:

```
x = block.super.magic;
```

On the other hand, suppose that we wanted to load disk block 59, assume that it is an indirect block, and then examine the 4th pointer. Again, we would use disk_read) to load the raw data:

```
disk_read(59, block.data);
```

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

```
x = block.pointers[4];
```

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 4KB raw chunk of data (`block.data`). But, once loaded, the filesystem layer knows that this data has some structure. The filesystem layer can view the same data from another perspective by choosing another field in the union.

The Shell (`shell.c`)

The shell we have provided for you is an interactive loop that

- Repeatedly prints the prompt "Sasti_FileSystem> "
- Parses the input
- Executes the command specified on that line of input
- Waits for the command to finish

This is repeated until the user types exit or quit.

In this project, the shell has the following built-in commands:

1. `help` - lists all the built-in Sasti_FileSystem commands that are supported by your shell.
2. `format` - format a new Sasti_FileSystem
3. `mount` - mounting a Sasti_FileSystem
4. `debug` - print debugging information about the Sasti_FileSystem
5. `create` - create an inode
6. `remove` - remove an inode
7. `cat` - output information from an inode
8. `copyin` - copying data in from the Sasti_FileSystem
9. `copyout` - copy data out to the Sasti_FileSystem
10. `help` - list the built-in commands provided in the shell
11. `exit` - terminate the shell program

We have provided for you a simple shell that will be used to interface with your filesystem and the emulated disk. When grading your work, we will use the shell to test your code, so be sure to test extensively.

To use the shell, build the source code using the *Makefile* provided and run with ./sasti_filesystem with the name of a disk image, and the number of blocks in that image. For example, to use the *image.5* example given below, run:

```
% make
% ./sasti_filesystem image.5 5
```

Or, to start with a fresh new disk image, just give a new filename and number of blocks:

```
% ./sasti_filesystem mydisk 25
```

Once the shell starts, you can use the help command to list the available commands:

```
sasti_filesystem> help
Commands are:
      format
      mount
      debug
      create
      delete  <inode>
      cat     <inode>
      copyin  <file> <inode>
      copyout <inode> <file>
      help
      quit
      exit
```

Most of the commands correspond closely to the filesystem interface. For example, `format`, `mount`, `debug`, `create` and `delete` call the corresponding functions in the filesystem. Make sure that you call these functions in a sensible order. A filesystem must be formatted once before it can be used. Likewise, it must be mounted before being read or written.

The complex commands are `cat`, `copyin`, and `copyout` reads an entire file out of the filesystem and displays it on the console, just like the Unix command of the same name. `copyin` and `copyout` copy a file from the local Unix filesystem into your emulated filesystem. For example, to copy the file *myfile.txt* into inode 10 in your filesystem, do the following:

```
sasti_filesystem> copyin myfile.txt 10
```

Note that these three commands work by making a large number of calls to `fs_read` and `fs_write` for each file to be copied.

Advice

1. Implement the functions roughly in order presented in *fs.h*. We have deliberately presented the functions of the filesystem interface in order of difficulty. Implement `fs_debug`, `fs_format`, and `fs_mount` first. Make sure that you are able to access the sample disk images provided. Then, perform creation and deletion of inodes without worrying about data blocks. Implement reading and test again with disk images. If everything else is working, then attempt `fs_write`.
2. Divide and conquer. Work hard to factor out common actions into simple functions. This will dramatically simplify your code. For example, you will often need to find and store individual inode structures by number. This involves a fiddly little computation to transform an *inumber* into a block number, and so forth. So, make two little functions to do just that:

   ```
   void inode_load( int inumber, struct fs_inode *inode )

   void inode_save( int inumber, struct fs_inode *inode )
   ```

   Now, everywhere that you need to load or save an inode structure, call these functions. You may also wish to have functions that help you manage and search the free block map:

   ```
   void initialize_free_blocks()

   int fs_allocate_block()
   ```

   Anytime that you find yourself writing very similar code over and over again, factor it out into a smaller function.

Submission Details

- You are to submit the following files:
  - `fs.h`
  - `fs.c`
  - `disk.h`
  - `disk.c`
  - `shell.c`
  - `Makefile`
- Place all of the files in a **single** ZIP (no, RAR won't work) file titled:
  - <roll_number>_PA4.zip
    - Example: 23100056_PA4.zip