Simple File System

Overview

This is an updated and simplified version of the Unix File System called **SimpleFS** as shown in the right. In this application, we have three components:

1. **Shell**: The first component is a shell program that allows the user to perform operations on the **SimpleFS** such as printing debugging information about the file system, formatting a new file system, mounting a file system, creating files, and copying data in or out of files. Note that the shell does not directly access the underlying disk, but instead calls functions in the **Shell Program** (`fsShellCreate()`, `fsShellCopyin()`, `fsShellCopyout()`, `fsShellDump()`, `fsShellFormat()`, `fsShellMount()`, `fsShellQuit()`, `fsShellHelp()`, `fsShellCreate()`, `fsShellDelete()`, `fsShellList()`).
2. **File System**: The second component takes the operations specified by the user through the **shell** and performs them on the **SimpleFS** data structures. The file system keeps track of which blocks are free on the disk, which blocks are used by which files, and which data is contained in each of the files. To store the data, it will need to interact with the **disk emulator** via methods such as `Disk_Read(disk, int, char*)`, `Disk_Write(disk, int, char*)`, `Disk_Open(char*)`, which allow the file system read and write to the disk image in 4KB block blocks.
3. **Disk Emulator**: The third component emulates a disk by dividing a normal file (called a **disk image**) into 4KB byte blocks and only allows the **File System** to read and write it blocks of blocks. This emulator will persistently store the data to the disk image using the `Disk_Open(char*)` and `Disk_Close()` methods.

The shell and disk emulator components are provided for you. You only have to complete the **file system** portion of the application for this lab.

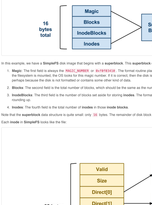
Simple File System Design

To implement the **file system** components, you will first need to understand the **SimpleFS** block layout. As noted previously, this lab assumes that disk blocks are of size **4KB**. The first block of the disk is the **superblock** that describes the layout of the rest of the **SimpleFS**. The superblock contains the following information about the file system:

1. **Magic**: The first 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.
2. **Blocks**: The second field is the total number of **inodes** in those **inode blocks**.

Note that the **superblock** structure is quite small: only 16 **bytes**. The remainder of disk block zero is left unused.

Each **inode** in **SimpleFS** looks like the file:



In this example, we have a **SimpleFS** disk image that begins with a **superblock**. The **superblock** consists of four fields:

1. **Magic**: The first field is a magic number that identifies the file system as being a **SimpleFS**. This number is unique to our file system and should be in every first byte of the **superblock** as a word of filesystem "signature". When formatting a new file system, the `fsMakeSuper()` function will first check that the **superblock** does not already contain this magic number. If it does, it will abort the formatting. Otherwise, it will write this magic number to the **superblock** to mark it as formatted. 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**: The second field is the total number of blocks on the disk. Formatting will use this value to set the number of blocks on the disk.
3. **Inodes**: 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.

Note that the **superblock** structure is quite small: only **12** bytes. The remainder of disk block zero is left unused.

Each **inode** in **SimpleFS** looks like the file:

Each field of the **inode** is a 4 byte (32 bit) integer. The `Valid` field is 1 if the **inode** is valid (i.e. has been created) and is **0** otherwise. The `Size` field contains the logical size of the **inode** data in bytes. 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 **0** means that the block pointer is not in use. There also is one double indirect block pointer, which points to a block full of block pointers, which in turn point to data blocks.

Note that an **indirect** data block is just a big array of pointers to further data blocks. Each pointer is a 32 bits int, and each block is **4KB**, so there are **1024** pointers per block. The data blocks pointed to by an **indirect** or **double indirect** block are just treated as a long string of bytes.

Suppose that the user creates a file on the **SimpleFS** file system. As discussed in class, a new/blank file would take a free block (perhaps one block), recording one for each block that was available in or use. This bitmap would be consulted and updated every time the filesystem needed to allocate a new block or free a data block from an inode.

Because **SimpleFS** stores files 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. When a block is freed, it must be likewise recorded in the bitmap.

Suppose that the user creates a new directory in the root directory of the **SimpleFS**. However, and then retrieves the system. He restarts the shell. Without a free block bitmap, **SimpleFS** cannot tell which blocks are in use and which are free. Fortunately, this information can be recovered by scanning the disk. Each time the **SimpleFS** filesystem is mounted, the system must build a new free block bitmap from scratch by examining all of the inodes and recording which blocks are in use. (This is much like performing an `fsck` every time the system boots.)

SimpleFS blocks much like the Unix file system. Each `Inf` is identified by an integer called an **inumber**. The **inumber** is simply an index into the array of inode structures that starts in block one. Given an **inumber**, it is easy to calculate the block on disk that should contain that inode. Once the block is read into memory, it is easy to index into that block using the **inumber**. Using **SimpleFS**, a directory is simply a file containing a list of mappings of filenames to inumbers. Many operations, however, that will not be part of this assignment.

More details about this lab and your deliverables are described below.

File Systems

Unix may seem that the systems are a solved problem with remarkable examples such as Ext4, XFS, and NTFS, the growth in big data and the emergence of SSDs as the primary storage medium has once again made file systems a hot topic. Today, we have second-generation file systems in the form of ZFS, Btrfs, and AppleFS, which build upon the foundation set by previous file systems to provide new features such as snapshots, compression, and encryption. Here's a list of interesting file systems and how they work.

Note: This assignment is based heavily on Project 4: File Systems by Doug Thain.

Deliverables

Timeline

Repository

To start this lab, you must fork the Simple file system lab repository on GitLab.

```
https://gitlab.com/CIS401/Simple-file-system
```

Once the repository has been forked, you need to:

1. Make the repository **private**.
2. Configure **access** to the repository.
   - Make sure you are the instructional staff.

Source Code

Folder Structure:

```
simple-file-system
├── Makefile              # This is the project Makefile
├── bin                   # This contains the application executables and scripts
├── disk-emulator         # This contains the SimpleFS library header files
│   ├── lib               # This contains the Disk Emulator header file
│   └── src               # This contains the File System header file
├── doc                   # This contains the doxygen documentation
│   ├── disk-e            # This contains the Disk Emulator implementation code
│   └── fs                # This contains the File System implementation code
├── lib                   # This contains the SimpleFS library
└── tests                 # This contains the test scripts
```

You must maintain this folder structure for your project and place files in their appropriate place.

Of the provided files, you are only expected to modify the `disk-emulator/src/fs.h` and `disk-emulator/src/fs.c` files as described below.

To build the project, you can compile your **make clean and then make**:

```
$ make clean
rm -f src/libfs/*.o src/libdisk/*.o lib/libfs.a src/shell/fsShell
rmdir -p lib
gcc -Wall -fPIC -c -o src/libfs/fs.o src/libfs/fs.c
gcc -Wall -Include -fPIC -o src/libfs/fs.o src/libfs/fs.c
ar -rcs lib/libfs.a src/libfs/fs.o src/libdisk/disk.o
gcc -L lib -o bin/fsShell src/shell/fsShell.o -lfs
```

K.I.S.S.

Remember, the exact organization of the lab code is up to you, keep in mind that you will be graded in part on coding style, cleanliness, and organization. This means your code should be consistently indented, well commented, easily readable, well organized, and appropriate naming among other things.

- Break long functions into smaller functions.
- Make sure each function does one thing and does it well.
- Abstract, but don't over do it.

Disk Emulator

As noted above, we provide you with a **disk emulator** on which to store your filesystem. This "disk" is actually stored as one big file in the file system, so that you can save data in a **disk image** and re-load it later. The disk emulator stores a fixed number of 4KB blocks, and provides you with an API that you can implement with to load your filesystem. Just like a real disk, the emulator only allows operations on whole blocks. Unlike a real disk, the emulator does not have any sector or track structure, and is much, much simpler and faster than real disk. The primary challenge of building a filesystem is connecting the user's requested operations on arbitrary amounts of data into operations on fixed block sizes.

The interface to the emulator can be a given in `disk-emulator/lib/disk.h`:

```
const int, / BLOCK_SIZE = 4096;

typedef struct Disk {
    int fildes;/descriptor;
    size_t Blocks;
    int Reads;
    int Writes;
    int mounted;
    int *mount;
} Disk;
extend /mounts(Disk*disk Disk * self,int blockno, Disk *disk);
int /dismount(Disk*disk Disk * self);
int /isMounted(Disk*disk Disk * self, int blockno, int *mount);
int /sanity(Disk*disk Disk * self);
int /mounted(Disk*disk Disk * self);
int /Reads(Disk*disk Disk * self);
int /Writes(Disk*disk Disk * self);
void /reset(Disk*disk Disk * self);
void /readBlock(Disk*disk Disk * self,int blockno, char *data);
int /writeBlock(Disk*disk Disk * self,int blockno, char *data);
}
```

Unfortunately any sort of operation on the disk, you must call `Disk_open()` method and specify a (next disk image for storing the data disk, and the number of blocks in the emulated 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, the destructor will automatically release the file. Opening the disk image and closing it when done is the only time you need to interact with the operating system.

Once the disk is open, you can use the `Disk_Read()` to discover the number of blocks on the disk. As the name suggest, `Disk_readBlock()` and `Disk_writeBlock()` read and write one block of data on the disk. Note that the first argument is a block number, so a call to `Disk_readBlock(disk,0,buf)` reads the first **4KB** of data on the disk, and `Disk_readBlock(disk,1,buf)` 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.

Additionally, you can register and unregister a disk as mounted by calling the `Disk_mount()` and `Disk_unmount()` methods respectively. The `Disk_mounted()` method returns whether or not the disk has been registered as mounted.

Note that the disk has a few programming conveniences that a real disk would. A disk is a rather finicky – if you send it invalid commands, it will likely crash the system or behave in other undefined ways. For example, if you try to read from "inode -7" you send it an invalid command, it will halt the program with an error message. For example, if you attempt to read or write a block from a disk that has not been opened, it will crash.

File System

Using the existing **disk emulator** described above, you will build a working **file system**. Take note that we have already constructed the interface to the filesystem and provided some skeleton code. The interface is given in `disk-emulator/lib/fs.h`:

```
typedef struct Filesystem {
    int *Disk;
    int *mounted;
    int *Disk;

    int *mounted;
    int *Disk;
    int *mounted;
    int *Disk;
    int *mounted;
    int *Disk;
```