Design a File System

Li Chengyao
5140219288

May 28, 2017

1 PROBLEM

1.1 DESCRIPTION

Write a program that implements the file system with the simulated disk. Several methods manipulating the file system are needed to be implemented such as *formatDisk(), shutdown(), create(), open(), inumber(), read(), write(), seek(), close(), delete()*. Some data structures like *Disk, SuperBlock, InodeBlock, IndirectBlock* can be found on www.os-book.com. You can see the detail in the end of chapter 11 of *OPERATING SYSTEM CONCEPTS WITH JAVA (Seventh Edition)*, from page 481 to 487.

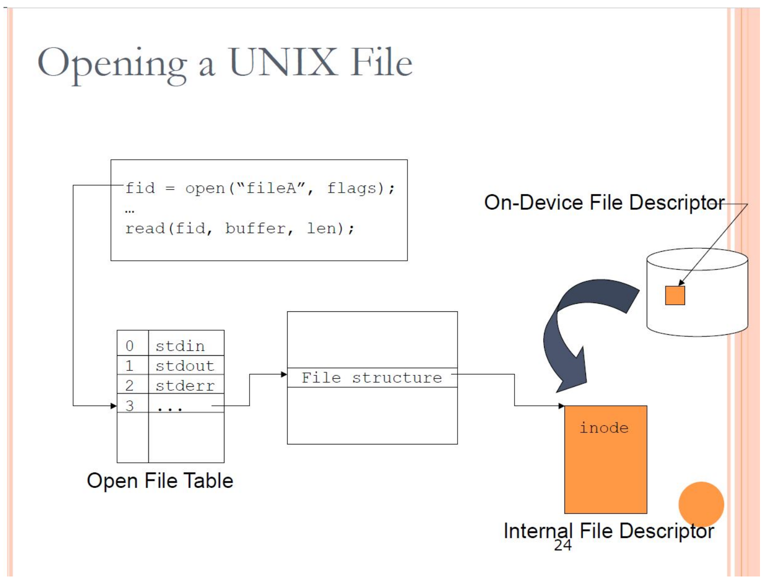
2 ALGORITHM

2.1 PRINCIPLES

In computing, a file system (or filesystem) is a type of data store which can be used to store, retrieve and update a set of files. The term refers to either the abstract data structures used to define files, or the actual software or firmware components that implement the abstract ideas. Some file systems are used on local data storage devices; others provide file access via a network protocol (e.g. NFS, SMB, or 9P clients). Some file systems are "virtual", in that the "files" supplied are computed on request (e.g. procfs) or are merely a mapping into a different file system used as a backing store. The file system manages access to both the content of files and the metadata about those files. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations.

Unix-like operating systems create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy. This means, in those systems, there is one root directory, and every file existing on the system is located under it somewhere. Unix-like systems can use a RAM disk or network shared resource as its root directory.

Unix-like systems assign a device name to each device, but this is not how the files on that device are accessed. Instead, to gain access to files on another device, the operating system



```
fid = open("fileA", flags);
…
read(fid, buffer, len);
```

Figure 2.1: Open A File in Unix-like system

must first be informed where in the directory tree those files should appear. This process is called mounting a file system. For example, to access the files on a CD-ROM, one must tell the operating system "Take the file system from this CD-ROM and make it appear under such-and-such directory". The directory given to the operating system is called the mount point it might, for example, be /media. The /media directory exists on many Unix systems (as specified in the Filesystem Hierarchy Standard) and is intended specifically for use as a mount point for removable media such as CDs, DVDs, USB drives or floppy disks. It may be empty, or it may contain subdirectories for mounting individual devices. Generally, only the administrator (i.e. root user) may authorize the mounting of file systems.

2.2 Operations

2.2.1 formatDisk()

Here I initial the instance *mySuperBlock* of the data structure *SuperBlock* with the number of disk blocks in the file system size, the number of inode blocks *iSize* and also *freeList* which is calculated with *iSize* representing the first block on the free list. Besides, I use the instance *freeBlockList* of data structure *ArrayList* to create the linked list of the free blocks.

2.2.2 shutdown()

After writing all the block data back to the disk, I clear all the states of the *bitmap* on the file table. Then use the method *stop()* in the data structure *Disk* to shut down the simulated disk.

2.2.3 create()

With the allocated file descriptor number and the initial *Inode*, I create the new fileąŕs file descriptor in the file table.

2.2.4 open()

According to the given *inumbers*, get the file descriptor to find its location.

2.2.5 INUMBER()

Get the *inumbers* with its file descriptor number.

2.2.6 READ()

First find the fileąŕs *Inode* according to its file descriptor in the file table. Using the pointers in the *Inode*, we can find the actually location of the file on the data blocks. That is to say, the content of the pointer is the number of the data block. The first 10 pointers are pointed to the blocks directly, which is called directly block, while the remaining 3 pointers are pointed to the blocks indirectly with multi-level structure. So if the corresponding pointer is not 0, that means this pointer has pointed to a data disk. Then I use the method *read()* in the data structure *Disk* to read it out to the buffer.

2.2.7 WRITE()

Similar to *read()*, I use the *Inode* to find where I can write the data and let the pointer have the value of *freeList*, then update the free list *freeBlockList*. Attention that the pointer has a same rule described in *read()*.

2.2.8 SEEK()

According to the offset and whence, update the *seekPointer* of the specific file descriptor.

2.2.9 CLOSE()

Write the its *Inode* back to disk and then clear its state in file table.

2.2.10 DELETE()

Clear its state in file table and also release the block it takes. The released blocks will be added to the free list.

3 RESULTS

3.1 ENVIRONMENT

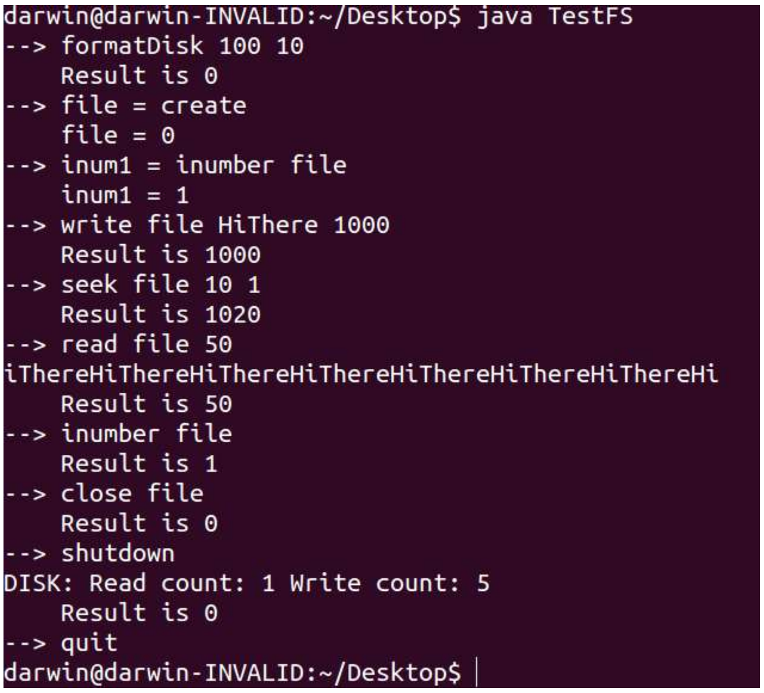
- Ubuntu 14.04 LTS
- Java Development Kit 1.8.0_131
- Eclipse

3.2 SCREENSHOTS OF THE RESULT

We use Linux Terminal to compile and execute the program. The result is shown in Fig. 3.1.

3.3 THOUGHTS

File system is quite complicated to comprehend while more complicated to be realized as well.



```
darwin@darwin-INVALID:~/Desktop$ java TestFS
--> formatDisk 100 10
    Result is 0
--> file = create
    file = 0
--> inum1 = inumber file
    inum1 = 1
--> write file HiThere 1000
    Result is 1000
--> seek file 10 1
    Result is 1020
--> read file 50
iThereHiThereHiThereHiThereHiThereHiThereHiThereHi
    Result is 50
--> inumber file
    Result is 1
--> close file
    Result is 0
--> shutdown
DISK: Read count: 1 Write count: 5
    Result is 0
--> quit
darwin@darwin-INVALID:~/Desktop$
```

Figure 3.1: Screenshot of a File System