CS356 Project 2 Technical Report

5120109159 Zuo Nan

Part 1. Basic Disk-Storage System

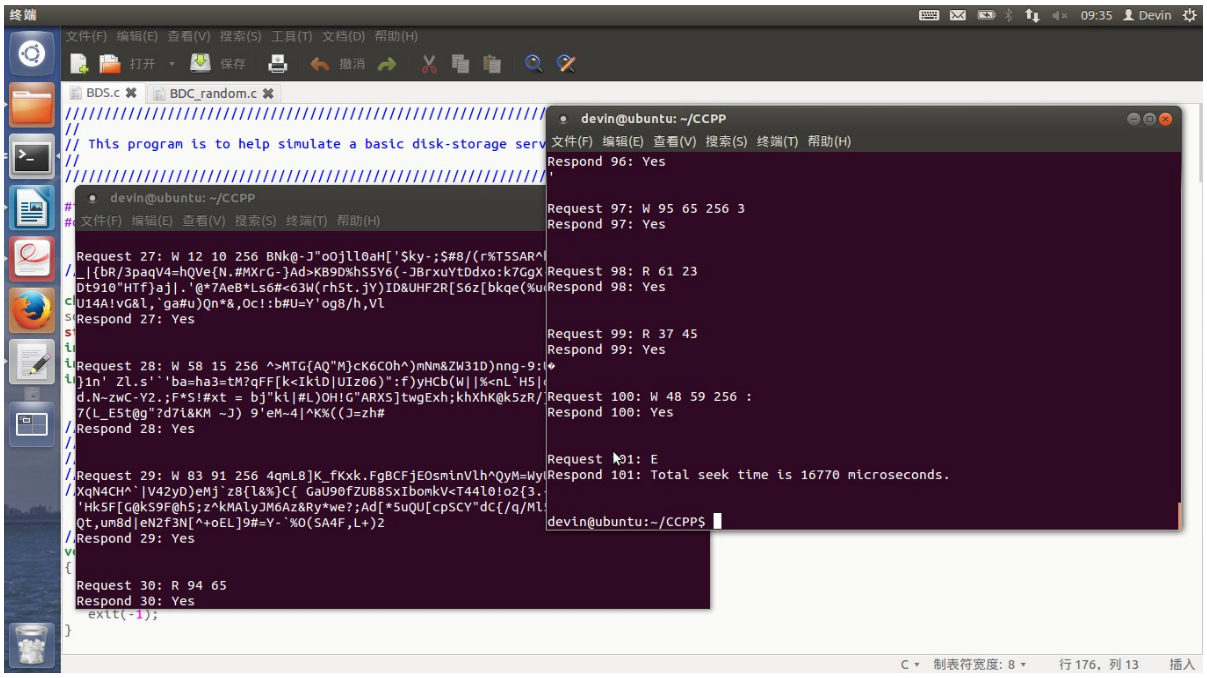
Part 1 is to simulate a basic disk-storage system. In this part, I wrote 3 C language programs: BDS, BDC_command and BDC_random.

Since it is the first step of the whole project, it is extremely essential to find a correct method to establish socket communication between a server and a client. To attain this, I went through the relevant websites and chose an easy way to implement Internet-domain socket communication. In my code files, all *init()* functions are revised based on the on-line codes to help establish sockets. Function *decode()* is used to turn the integer information in a string into int data type (a more powerful *atoi()* function). Function *encode()* works in the contrary direction. These functions are powerful tools in the following programs since information is all translated through socket in string format. Global array *buffer[]* is used to transfer information through socket. In addition, to ensure the socket communication will never encounter a deadlock (say both the server and the client want to write to the socket simultaneously), I made a rule that at any time a client shall ask first and then the server shall answer at once. This rule is in effect in the whole project.

According to the requirements of the project, **BDS** is executed to work as the basic disk-storage server, which regards a certain file as a virtual disk and reads or writes the file just like reading or writing a disk. In my program, a *seek()* function is used to help translate planar physical locations of the virtual disk into linear positions of the file. Function *getMsg(), startReq()* and *sendMsg()* are used to help get respond from or send request to the client and at the same time print the message on the terminal. In *main()* function, the first step is to analyze the arguments in *argv[]* and store the corresponding information, then a while loop is written to serve the requirements of the client. Once an R/W request is received, the server will first *seek* for the corresponding disk track and automatically change an int named *track,* which records the current track number, and meanwhile calculated the additional seek time caused by jumping from one track to another. The loop will break on receipt of an “E” requirement, which is added to the Protocol by myself. Finally, BDS will send the total seek time to the client.

BDC_command is a command-driven client program, you can write requests manually in the terminal when executing this program. It is very simple and only a while loop serves to be its trunk. The loop breaks when function *exitPermit()* detects the seek time sent from the server.

BDC_random is a random data client, who first gets the disk information as well as the number of requests to generate, and then generates random requests in function *clientLoop()* based on the Protocol. In order to match IDS in part 2, except *exitPermit()* function stated above, a function called *nextPermit()* is designed so that IDS can change from one disk scheduling algorithm to another when the number of requests amounts to a certain number.


终端
09:35
Devin
文件(F) 编辑(E) 查看(V) 搜索(S) 工具(T) 文档(D) 帮助(H)
打开
保存
撤消
BDS.c
BDC_random.c
devin@ubuntu: ~/CCPP
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
C
制表符宽度: 8
行 176, 列 13
插入

```
////////////////////////////////////////////////////////
//
// This program is to help simulate a basic disk-storage serv
//
////////////////////////////////////////////////////////
```

```
Request 27: W 12 10 256 BNk@-J"oOjll0aH['$ky-;$#8/(r%T5SAR^
_|{bR/3paqV4=hQVe{N.#MXrG-}Ad>KB9D%hS5Y6(-JBrxuYtDdxo:k7GgX
Dt910"HTf}aj|.'@*7AeB*Ls6#<63W(rh5t.jY)ID&UHF2R[S6z[bkqe(%u
U14A!vG&l,`ga#u)Qn*&,Oc!:b#U=Y'og8/h,Vl
Respond 27: Yes


Request 28: W 58 15 256 ^>MTG{AQ"M}cK6COh^)mNm&ZW31D)nng-9:
}1n' Zl.s'`'ba=ha3=tM?qFF[k<IkiD|UIz06)":f)yHCb(W||%<nL`H5|
d.N~zwC-Y2.;F*S!#xt = bj"ki|#L)OH!G"ARXS]twgExh;khXhK@k5zR/
7(L_E5t@g"?d7i&KM ~J) 9'eM~4|^K%((J=zh#
Respond 28: Yes


Request 29: W 83 91 256 4qmL8]K_fKxk.FgBCFjEOsminVlh^QyM=Wy
XqN4CH^`|V42yD)eMj`z8{l&%}C{ GaU90fZUB8SxIbomkV<T44l0!o2{3.
'Hk5F[G@kS9F@h5;z^kMAlyJM6Az&Ry*we?;Ad[*5uQU[cpSCY"dC{/q/Ml
Qt,um8d|eN2f3N[^+oEL]9#=Y-`%O(SA4F,L+)2
Respond 29: Yes


Request 30: R 94 65
Respond 30: Yes
```

```
  exit(-1);
}
```

```
Respond 96: Yes
'


Request 97: W 95 65 256 3
Respond 97: Yes


Request 98: R 61 23
Respond 98: Yes


Request 99: R 37 45
Respond 99: Yes
�


Request 100: W 48 59 256 :
Respond 100: Yes


Request 101: E
Respond 101: Total seek time is 16770 microseconds.


devin@ubuntu:~/CCPP$ 
```

To make the whole program more graceful on the terminal, I used global variable *num* in both the server end and the client end to represent the serial number of a pair of request and respond, for example, “Request 0”, “Respond 0”, “Request 1”, “Respond 1” and so forth. In order to match IDS program in part 2, the serial number of responds must be sent from a server to its client, rather than generated by the client itself. So in my program, a server always sends a string starting with “*serial number*: ” followed by the real answer “Yes” or “No”.

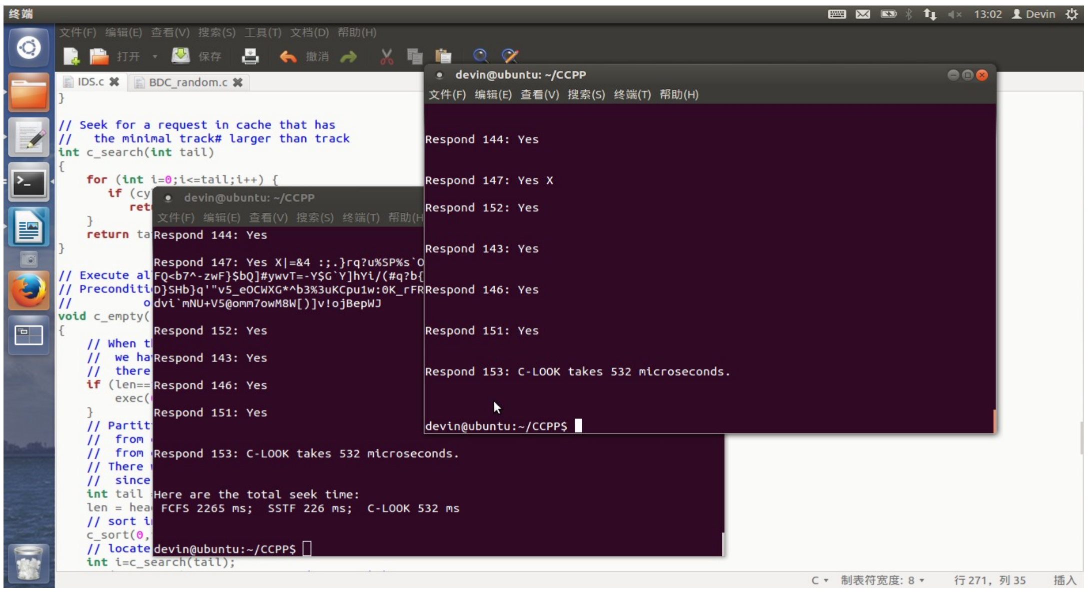
Part 2. Intelligent Disk-Storage System

Part 2 is to implement a much smarter disk-storage server, which can store the requests first and execute them according to some disk scheduling algorithm at a proper time. According to the requirement of TA, IDS should share the same random data client with BDS. To fulfill this requirement, I went out of my way to revise BDC_random and fortunately it turned out to be fine. But one thing I should emphasize is that **you'd better not use my BDC_command in this part** because the input/output format is rather complicated and only random data client is competent for this work.

As what has been mentioned in part 1, the serial number of respond is sent from IDS so that the client may know to which request the current respond is answering. Since in this part requests are stored in the cache of server and are delayed to be executed, the sending order of responds are likely not the same as their corresponding request order, so I store the request order in an array named *reno[]*.

Another thing worth mentioning is that since I have made a rule that request and respond must turn up in turn, my IDS will send nonsensical respond “#” to the client when its cache is not full and there's no request to exit. On the other hand, BDC_random will send nonsensical request “\n” to the server when all requests have been sent and the server is still sending responds. Both the server and the client can easily recognize the nonsensical information sent by its peer.

In this part, the server will execute the same number of requests using three scheduling algorithms respectively, though the requests are actually disparate. For example, if you input N in the terminal of BDC_random when it inquires the number of R/W requests, in this part it will generate 3*N requests rather than N requests and send them to IDS. IDS will execute first N requests according to FCFS, next N requests according to SSTF, and last N requests according to C-LOOK. Whenever a scheduling is terminated, a total seek time will be sent to the client. The three seek time will be shown together on the terminal of the server at the end of the program.


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 工具(T) 文档(D) 帮助(H)
打开
保存
撤消
IDS.c
BDC_random.c
devin@ubuntu: ~/CCPP
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
13:02
Devin
制表符宽度: 8
行 271，列 35
插入

```
// Seek for a request in cache that has
//   the minimal track# larger than track
int c_search(int tail)
{
    for (int i=0;i<=tail;i++) {
```

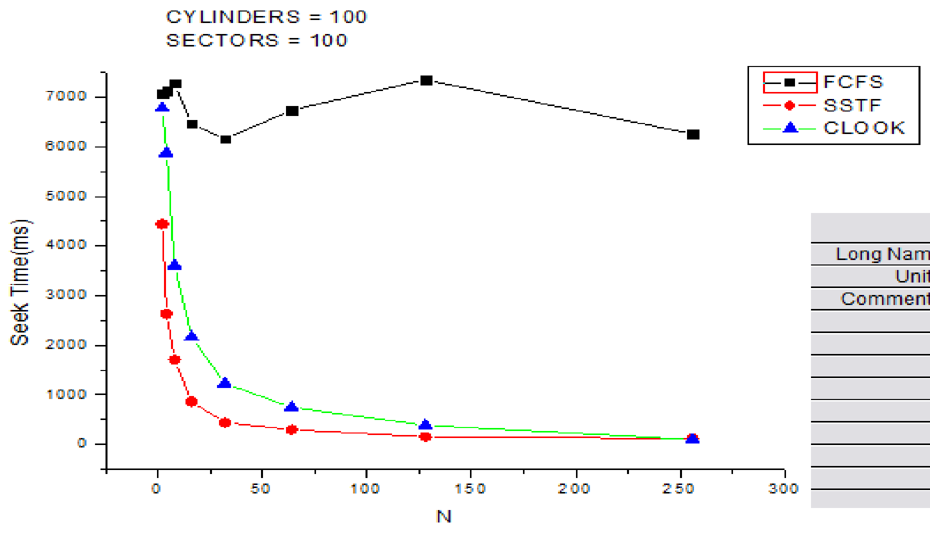
```
Respond 144: Yes

Respond 147: Yes X|=&4 :;.}rq?u%SP%s`O
FQ<b7^-zwF}$bQ]#ywvT=-Y$G`Y]hYi/(#q?b{
D}SHb}q'"v5_eOCWXG*^b3%3uKCpu1w:0K_rFR
dvi`mNU+V5@omm7owM8W[)]v!ojBepWJ

Respond 152: Yes

Respond 143: Yes

Respond 146: Yes

Respond 151: Yes


Respond 153: C-LOOK takes 532 microseconds.


Here are the total seek time:
 FCFS 2265 ms;  SSTF 226 ms;  C-LOOK 532 ms


devin@ubuntu:~/CCPP$ 
```

```
Respond 144: Yes


Respond 147: Yes X

Respond 152: Yes


Respond 143: Yes


Respond 146: Yes


Respond 151: Yes


Respond 153: C-LOOK takes 532 microseconds.


devin@ubuntu:~/CCPP$ 
```

To demonstrate the superiority of such an intelligent disk-storage server with cache, I collected 16 groups of data by running my program. We use N to represent the size of cache of IDS, and show the total seek time of different scheduling algorithms as follows:

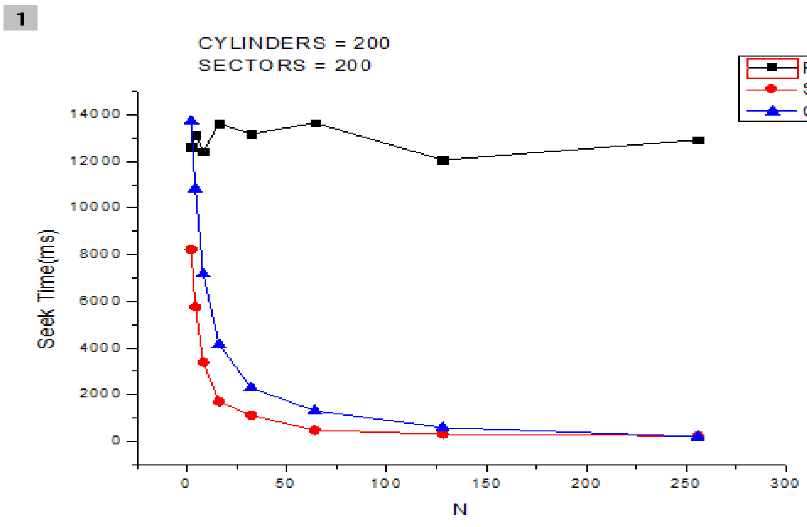
1. When CYLINDERS = 100, SECTORS = 100:

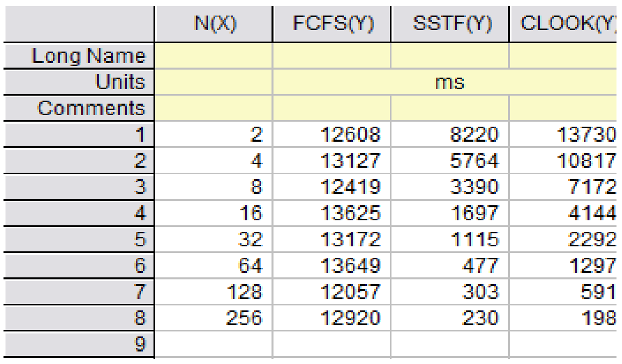
1



	N(X)	FCFS(Y)	SSTF(Y)	CLOOK(Y)
Long Name				
Units			ms	
Comments				
1	2	7067	4442	6772
2	4	7127	2625	5876
3	8	7278	1705	3604
4	16	6468	862	2163
5	32	6156	439	1225
6	64	6741	296	742
7	128	7350	152	386
8	256	6258	120	99
9				

2. When CYLINDERS = 200, SECTORS = 200:





	N(X)	FCFS(Y)	SSTF(Y)	CLOOK(Y
Long Name				
Units		ms		
Comments				
1	2	12608	8220	13730
2	4	13127	5764	10817
3	8	12419	3390	7172
4	16	13625	1697	4144
5	32	13172	1115	2292
6	64	13649	477	1297
7	128	12057	303	591
8	256	12920	230	198
9				

From the data and plots above, we can see when CYLINDERS and SECTORS are determined, the total seek time of FCFS will have little variations due to the increase of cache size, while the seek time of SSTF and C-LOOK will go down apparently. That is because FCFS always executes the requests in chronological order and hence cache does not affect its total seek time, while SSTF and C-LOOK will execute the requests in cache in a new order. By further observation, we can conclude that with the same condition, FCFS is always the scheduling that takes the longest seek time and SSTF always takes the shortest, since SSTF always choose the optimal request to execute in each step. Another thing is that as cache size grows, the decrease of the seek time of SSTF and C-LOOK will slow down, since when cache size grows to the number of requests or even more than it, nearly all requests will be stored into the cache first and then be executed. In this case, the total seek time will only depend on the scheduling algorithm you use rather than the cache size.

Now let's turn to look at the concrete realization of my **IDS** program.

According to the lab requirements, I used a two-dimensional array *cache[][]* to simulate a cache in the server, which can store up to 512 requests. However, when the program is executed, the cache size will be determined by a command line parameter, which is stored as an int called NUM. Unlike BDS in the first part, IDS will serve "*I*" request only at the start of the program – that is a function called *firstRes()*. The other requests, including "R", "W" and "E", will be first stored into the cache and then be executed at a proper time. The execution of these requests is written in a function called *exec()*, which will choose the cache item and send answer (as well as respond serial number) to the client.

My FCFS scheduler regards *cache[]* as a double-ended queue, by using *head* to record the position of the earliest request and *len* to record the position for a coming request. Such a data structure not only is easy to realize, but can avoid a large amount of waste of space as well. Each time a request is received, IDS will put it into *cache[len]*; once *cache[]* is full (NUM items), IDS will try execute the earliest request. After an "E" request is received, IDS will execute all requests from *head* to *len-1*.

Unlike FCFS scheduler, SSTF scheduler deems *cache[]* as something like a stack, or rather a "priority stack". The priority of each cache item is stored in an array called *cyli[]*, which records the track number of each

R/W request. Whenever *cache[]* is full, IDS will use a O(n) algorithm to determine the request with the nearest track number to *track* in the cache, swap it with the last cache item and then execute it. That is what function *sstf_deq()* does. Like FCFS scheduling, when "E" request is received, IDS will call *sstf_deq()* repeatedly until no request is left.

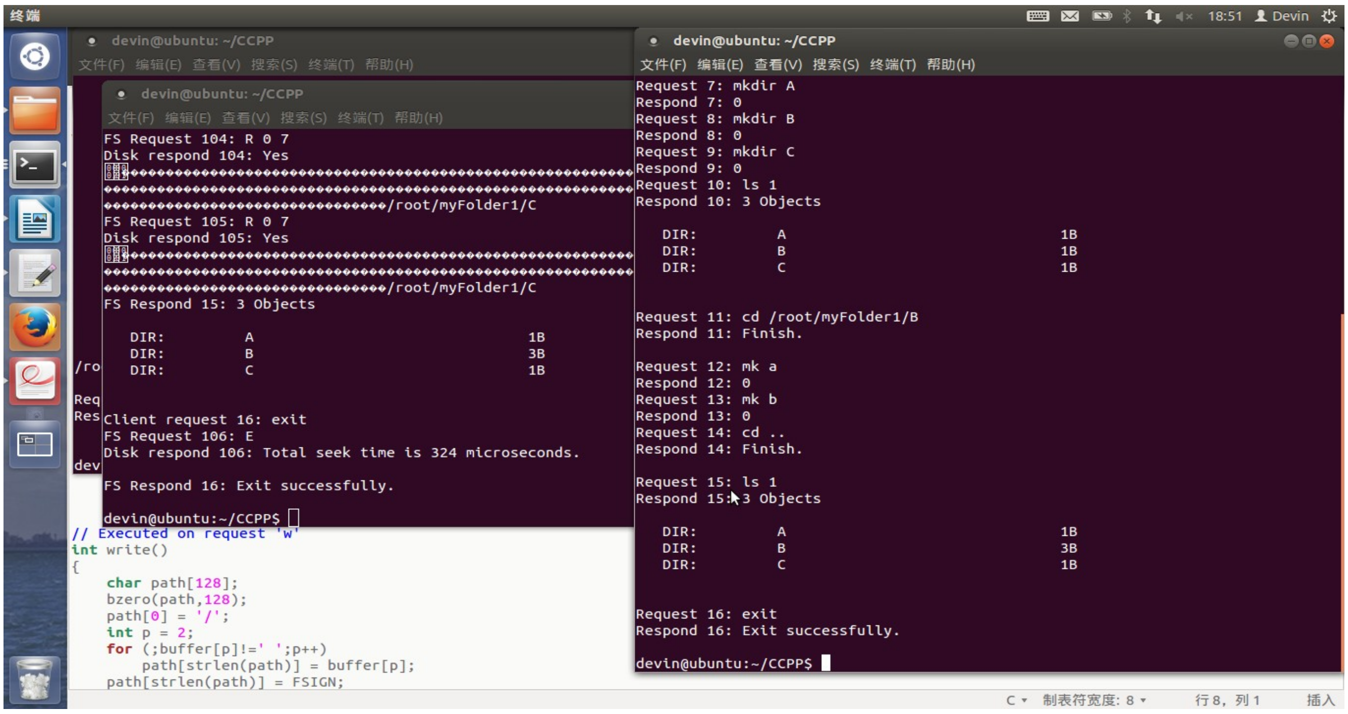
The most challenging part for me is C-LOOK scheduler. This time, once the cache is full or the client is willing to exit, IDS will execute all remaining requests rather than only one request. I designed a function called *c_empty()* to do such work. Each time *c_empty()* is called, it will only execute all the requests stored in *cache[]* before the function call, and once it executes an item and send a respond, it is possible that a new request will come, so it's conceivable that it must be a recursive function whose basis condition is that there's only "E" request is left. If there's only "E" request is left, IDS will execute it and end the program smoothly; otherwise it will call *c_empty()* again to try finish all the jobs. When *c_empty()* function works, it will first use quick-sort algorithm to sort all the current cache items according to the track numbers in *cyli[]*. Request "E" has a sentinel track number -1 – once it is detected, it will not be executed unless it is the only item in the cache. C-LOOK scheduler always moves the disk arm from the current track to larger track number first and then goes back to 0 if there are requests left with smaller track numbers. So in contrast with SSTF, it wastes a lot of seek time in moving the disk arm from one end to the other, but it is more feasible in interactive systems because no request will be starved in any dynamic scenario.

Part 3. File System Based on I-node

In this part, a file system is implemented as an interface between a client program and a disk server, and thus a user may regard the data in the disk as abstract data types rather than blocks or bytes and manipulate the data by sending system calls listed in the Protocol. To test the system and help to debug, a command-line driven client called **FC** is written based on BDC_command. The following paragraphs will mainly talk about **FS** program, which implements the file system.

First of all, it's necessary to illustrate my design of disk blocks. In my design, FS program will attach each block in the disk to a logical position number from 0 to CYLINDERS*SECTORS-1, which exactly conforms to the organization of the real disk file. Each byte of the disk will store either a character or an integer ranging from -128 to 127. There will be five types of blocks in the disk: one for bit vectors, two for directory i-nodes and two for file contents. **Bit vector** blocks are stored in all block[2048*k] (k=0,1,2,3...); such a block will records whether itself and its following 2047 blocks have been allocated or not, and each bit represents a block, 0 for allocated, 1 for free. To test whether block[pos] is occupied, you can test the (pos%8)th bit of the (pos %2048/8)th byte in block[pos/2048*2048]. As for **directories**, I organize them in a **root tree** structure, whose leaves can be files or directories, but inner nodes must be directories. The root of the whole tree has the path "/root". Any directory will have an i-node block as marking its logical position: block[0] stores its parent directory logical position; block[1] records the number of content blocks (not including its subdirectories and the files in it); block[2] - block[188] store the positions of its first 187 contents (subdirectories or files); block[189] and block[190] store positions of single indirect blocks; block[190] stores the position of a double indirect block; the remaining 64 bytes store the absolute path of the directory. Indirect blocks have another i-node format (the 3rd type of block), it will store 256 positions of contents or other indirect blocks. Hence a directory will have up to 187+256*2+256*256 = 66235 blocks of contents, which is large enough as far as I know. The root directory has logical position 1 all the time and its parent byte is set as -1. The last two types of blocks contain **file** contents. In my FS, each file is organized as a **doubly-linked list**. The first block of a file represents its logical position: block[0] is the position of its parent directory; block[1] records the number of blocks the file occupies; block[2] – block[190] store its the first 189 bytes of the file; block[191] points to the next content block of the file (as a linked list, -1 for nil); the remaining 64 bytes store the absolute path of the file ended with '#' as a file sign. As for the other node in the linked list, the block format is disparate: block[1] points to its previous node; block[1] – block[254] stores 254 bytes of file contents; block[255] points to its next node. That's all about five kinds of blocks in the disk.

There are many important details in my block design. For example, you can test the second byte of a directory node to judge whether it's an i-node or an indirect block since only in indirect blocks this byte is negative. By the same token, you can also decide whether a file node is the first one or not by testing its last byte, since only the first node will have 0 in this byte. In addition, we have three ways to represent the position of an item of the directory contents: the first one is apparent, just using its logical position; the second one is using its parent directory position and its relative position in that directory (e.g. a file may be the Nth content of a directory, so its relative position is N-1); the third one is using the position of the i-node that points to it (either direct or indirect) and its pointer location in that *prev* block. Changing from one representation to another is widely used in file/ directory searching, making or removing.


终端
devin@ubuntu: ~/CCPP
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
18:51 Devin
C ▾ 制表符宽度: 8 ▾ 行 8，列 1 插入

```
FS Request 104: R 0 7
Disk respond 104: Yes
����������������������������������������������������������������
�����������������������������������������������������������������
���������������������������������/root/myFolder1/C
FS Request 105: R 0 7
Disk respond 105: Yes
����������������������������������������������������������������
�����������������������������������������������������������������
���������������������������������/root/myFolder1/C
FS Respond 15: 3 Objects

   DIR:         A                               1B
   DIR:         B                               3B
   DIR:         C                               1B


Client request 16: exit
FS Request 106: E
Disk respond 106: Total seek time is 324 microseconds.

FS Respond 16: Exit successfully.

devin@ubuntu:~/CCPP$
```

```
// Executed on request 'w'
int write()
{
    char path[128];
    bzero(path,128);
    path[0] = '/';
    int p = 2;
    for (;buffer[p]!=' ';p++)
        path[strlen(path)] = buffer[p];
    path[strlen(path)] = FSIGN;
```

```
Request 7: mkdir A
Respond 7: 0
Request 8: mkdir B
Respond 8: 0
Request 9: mkdir C
Respond 9: 0
Request 10: ls 1
Respond 10: 3 Objects

   DIR:         A                               1B
   DIR:         B                               1B
   DIR:         C                               1B


Request 11: cd /root/myFolder1/B
Respond 11: Finish.

Request 12: mk a
Respond 12: 0
Request 13: mk b
Respond 13: 0
Request 14: cd ..
Respond 14: Finish.

Request 15: ls 1
Respond 15: 3 Objects

   DIR:         A                               1B
   DIR:         B                               3B
   DIR:         C                               1B


Request 16: exit
Respond 16: Exit successfully.

devin@ubuntu:~/CCPP$
```

Another thing I need to explain is the socket communication of FS. On the one hand, FS need to communicate with the client by getting requests or sending responds; on the other hand, FS will communicate with BDS through another socket. Generally speaking, I use an array *buffer[]* to cope with both two sets of socket communication, just like the maneuver used in part 1 and part 2, and many functions tools for communication are based on this array. Besides, I use an int array *vect[]* as a cache of bit vector blocks in the disk to speed up the searching for a vacant block. Once a block is allocated or released, function *wrVect()* will immediately revise both the cache and the corresponding contents in the disk file. Another cache is used for the current directory i-node, which is named *curr[]*. An integer currPos is used to record the position of the current directory, and *curr[]* is exactly the content of block[*currPos*]. Once you start FS program or make a format request, you will find yourself in the root directory, that means *currPos*=1. For this cache, I also made a rule that modifications to *curr[]* shall be followed by modifications to block[*currPos*], which is called "write-through". This is similar to what *wrVect()* does, but here there's no single function to guarantee it. I must user communication functions *getCurr()* and *sendCurr()* at proper time to guarantee the synchronization of the disk file and the cache contents of FS.

Considering FS program is rather complicated, I may not as well explain more details about it here. I used a top-down design method, and for each request in the protocol, there may be many functions to support its execution. For more details of implementation, please refer to my codes and comments in file FS.c.

My Harvest in This Project

Since I came to study computer science and technology, this project is the most challenging one I have ever seen. It requires not only a good knowledge of basic file system concepts, but also demands mature programming skills. One thing I wish to complain about is that using C language is quite different from using C++, because you cannot design a class whenever you want and what you have are just functions. This is fine for a short program since there is no need to establish many abstract data types, but it's really disturbing to deal with a program with one thousand lines or more. However, this project really taught me many important lessons. For example, my past experiences of programming have driven me into a bad habit of making an extensive use of global variables to solve problems (especially in OJ and USACO, etc). But global variables do not guarantee encapsulation, and thus in programs like BDS and IDS, I have to take pains to follow the tracks of array *buffer[]* in case it will get me into trouble. Another thing waste me a lot of time is the use of string functions, especially in part 3, where I wanna use strings to transfer bytes and the string functions often terminate its work whenever it meets a zero. So in program FS, I have to give each non-ASCII-code a shifting (DELT=13) to avoid the existence of zero in the bytes I wanna transfer; what's more, I have to waste 1/8 of the whole disk file to avoid 11110010 being transferred through socket, and I must confess it's a fatal drawback to my file system. It's a good lesson for me that strings are never equivalent to an array of 8-bit binary codes, and I must take care when I use string functions in future.