Project2 Technical Report

ZhangZuoming 5120309626

1. Implementation of project2

Since BDS, BDC_command, BDC_random, FC is very similar from person to person, so I will include in IDS and FS implementation

(1) IDS

When cache is set to n, each time it reads a command from BDC_random, it will automatically add 1 to another variable N which is initially set to 0. When a command comes, it will save the command in an array and save cylinder number in another array in their arrive order. Then we divide N by n and divide it to (N+(n-1))/n parts, and organize each n part in the following order:
Suppose start position of the whole N schedule is 0, which is initial prev = 0.
FCFS: not change;
SSTF: for(i = 0; i<n; ++i){

```
next = min(abs(array[i] – prev)); // here i is in a loop from 0 to n-1
prev = next;
```

}

And reorder them in the array and finally count the SSTF delay when they are organized.

CLOOK:

In each part which contains n cylinder numbers, my program will set prev as initial number, and choose from those number which are not less than prev and organize them in rising order. If all numbers not less than prev are used up, we will append those which are less than prev in rising order to the queue. For example, prev = 10, and the numbers are 12 7 18 25 13 9 1. The order should be like : (10) 12 13 18 25 1 7 9,Pay attention that prev should not be in the queue.

(2) FS

I will mostly show it through pictures:

The whole diskfile will look like the following:

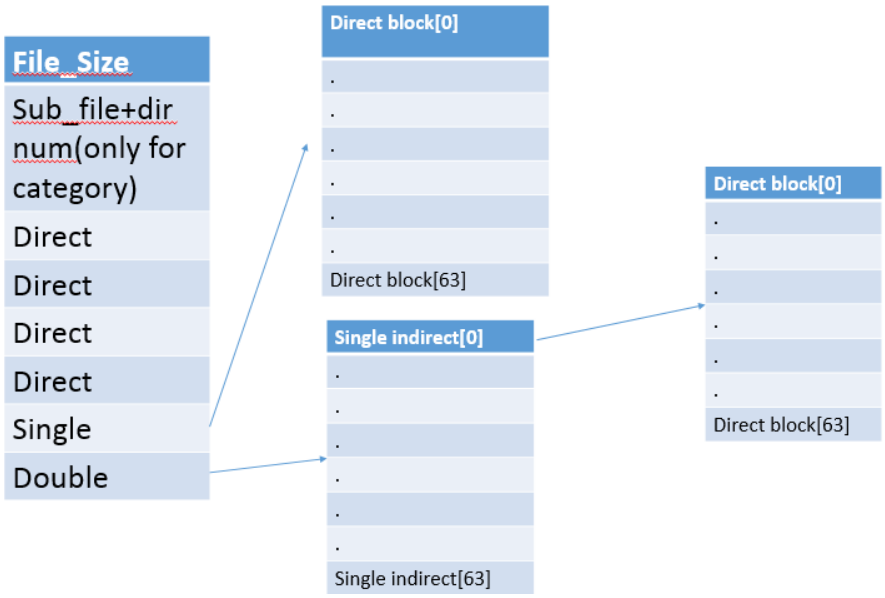
	A	B	C	D	E	F	G
1	Inode_Start						
2							
3			Data_Block_Start				
4							
5							
6							
7							
8			Inode_Bitmap_Start		Data_Bitmap_Start		SuperBlock

Please pay attention that, SuperBlock, which contains the general information such as inode numbers, data block numbers and so on is saved to the last block of the file. The file system start positions are not fixed and they vary according to cylinder and sector numbers. Pay attention that although I write Inode_bitmap_start and Data_bitmap_start in separate

blocks, they may have some part in the same block in the file system and data_bitmap is closely appended to inode_bitmap. Also, it is actually a byte map instead of bit map because I use '0' and '1' in char instead of in binary bit. (If we can use C++ library, I will use bitset, which is very convenient!) Inode size is 64 bytes and data block is 256 bytes, so I can store 4 inodes in a single data block, and (Data_bitmap_start – Inode_bitmap_start) = 4 * (data_block_start – inode_start).
Inode is in the following format:

File_Size
Sub_file+dir num(only for category)
Direct
Direct
Direct
Direct
Single
Double

Each grid is for 8 bytes and numbers are represented in 8 bytes char format. Single indirect and double indirect are exactly what the book says, and each is 256 bytes, 8 bytes per node. As following shows:



Inode can be used both for file and category. File Inode's pointers contains blocks of file content, while category Inode's pointers contains blocks of categories. Each sub file or category of a category node is 32 bytes, 1 byte is for differentiate file(1) and category(0). 8 bytes are for sub file or category inode address. And the internal 32 -1 -8 -1 = 22 bytes are for name.(The extra * is taken into consideration). So the first block of a directory may look like the following:

0	.*	00000000000000000000	11111111
0	..*	00000000000000000000	22222222
0	name*	000000000000000000	33333333
1	filename*	00000000000000	44444444
.....			
.....			
.....			
.....			

The third slot is a directory node pointer which points to 33333333, the fourth is a file node pointer which points to 44444444. Pay attention that I save "." and ".." as directory content so that cd . and cd .. can be operated as cd to other ordinary sub directories. The ".." address for root node is itself. Some codes prove that my project use Inode is as following:

```
void Strench_File_to_New_Size (int Socket_fd, int Inode_Num, int New_Size)
```

```
// LESS THAN CONDITION
if (New_Block < Origin_Block) {
    if (Origin_Block <= 4) { // Direct
        for (i = Origin_Block; i > New_Block; --i) {
            Get_Npart_of_Inode (Socket_fd, Inode_Num, i + 1, Update_Num);
            Update_Bitmap (Socket_fd, atoi(Update_Num), 0, 0);
        }
    }
    else if (Origin_Block == 5) {
        Get_Npart_of_Inode (Socket_fd, Inode_Num, 6, Update_Num);
        Get_Npart_of_Single_Dir (Socket_fd, atoi(Update_Num), 0, Direct_Pointer);
        Update_Bitmap (Socket_fd, atoi(Update_Num), 0, 0);
        Update_Bitmap (Socket_fd, atoi(Direct_Pointer), 0, 0);
        for (i = Origin_Block - 1; i > New_Block; --i) {
            Get_Npart_of_Inode (Socket_fd, Inode_Num, i + 1, Update_Num);
            Update_Bitmap (Socket_fd, atoi(Update_Num), 0, 0);
        }
    }
    else if(Origin_Block <= 36) { // Single_Indirect
        Get_Npart_of_Inode (Socket_fd, Inode_Num, 6, Update_Num);

        if (New_Block >= 5) {
            for (i = Origin_Block; i > New_Block; --i) {
                Get_Npart_of_Single_Dir (Socket_fd, atoi(Update_Num), i - 5, Direct_Pointer);

                Update_Bitmap (Socket_fd, atoi(Direct_Pointer), 0, 0);
            }
        }
        else {
            for (i = Origin_Block; i > 5; --i) {

                Get_Npart_of_Single_Dir (Socket_fd, atoi(Update_Num), i - 5, Direct_Pointer);
```

```
void Get_Super_Block (int Socket_fd) { // Get value from super block
    char buffer[1024];
    char *Super_Info[10];
    Memcpy_from_C_S (Socket_fd, CYLINDER - 1, SECTOR - 1, buffer);
    Parse_Slash_Line (buffer, Super_Info);
    Inode_Num = atoi(Super_Info[0]);
    Inode_Empty_Num = atoi(Super_Info[1]);
    Inode_Block_Num = atoi(Super_Info[2]);
    Bitmap_Block_Num = atoi (Super_Info[3]);
    Bitmap_Start = atoi (Super_Info[4]);
    Data_Num = atoi(Super_Info[5]);
    Data_Empty_Num = atoi (Super_Info[6]);
}
```

The first shows that this project has operations on direct, single indirect pointers. The second shows that the superblock contains the condition of inode, data block and so on.

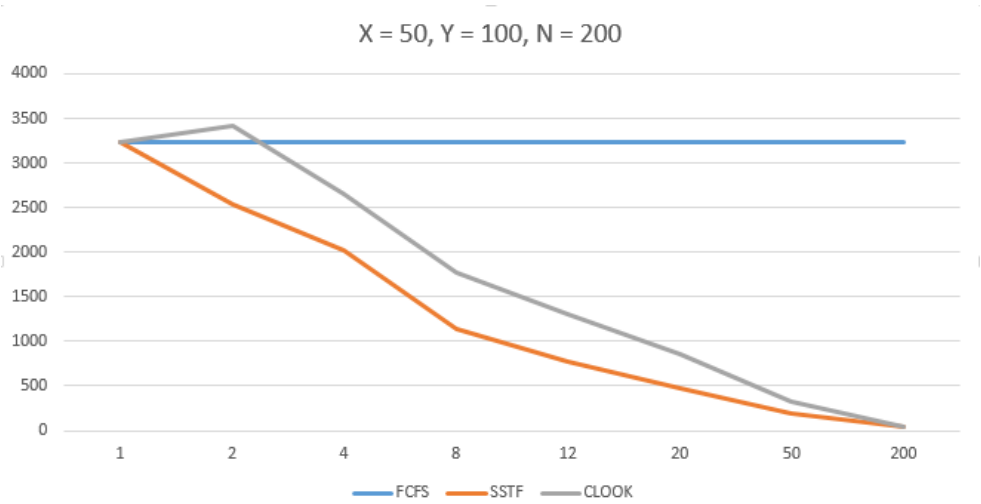
2. IDS analysis

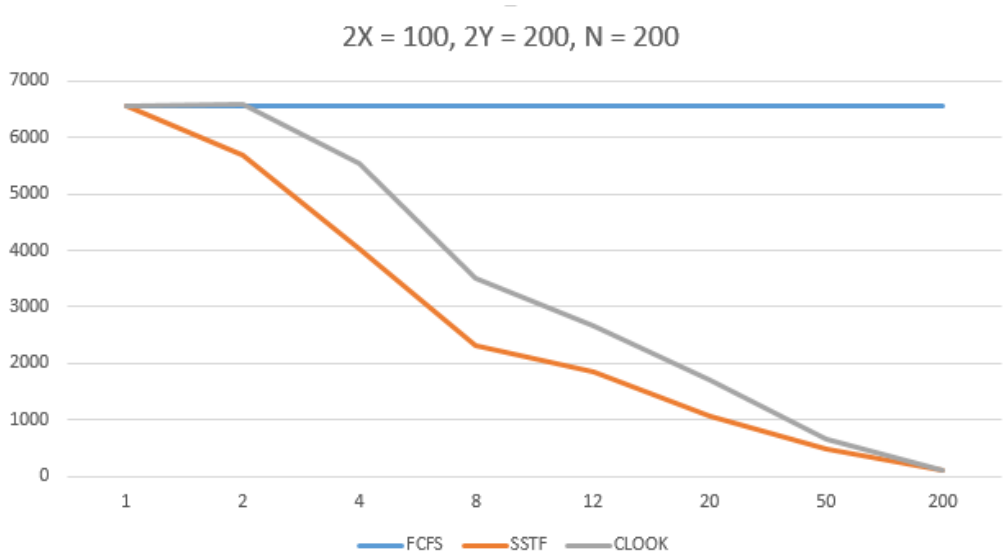
(1) Data:

X = 50, Y = 100, N = 200, TrackToTrackDelay = 1

n	X = 50 , Y = 100, N = 200			2X = 100, 2Y = 200, N = 200		
	FCFS	SSTF	CLOOK	FCFS	SSTF	CLOOK
1	3241	3241	3241	6563	6563	6563
2	3241	2532	3411	6563	5698	6581
4	3241	2020	2646	6563	4039	5534
8	3241	1136	1774	6563	2328	3507
12	3241	782	1313	6563	1861	2674
20	3241	474	861	6563	1050	1710
50	3241	190	324	6563	486	670
200	3241	49	49	6563	99	99

(2) Performance diagram:





(3) Diagram analysis:

From the diagrams we can find that when n = 1, all the three algorithm run as FCFS. When n is very small but not equals to 1, CLOOK may even cost more time than FCFS, this is because when no number is larger than prev, it will move to the smallest number in the cache waiting for operation instead of the nearer one, thus wasting a lot of time. But when n is larger, it is becomes more efficient than FCFS.

SSTF is a very fast schedule algorithm, although not the optimal one, but it is very close to the optimal one and is much faster than other schedule algorithms. But when n becomes larger, SSTF does not have so much advantage than CLOOK as it has when n is smaller, so we can guess that when n becomes larger, CLOOK is a very good schedule method since SSTF is difficult to implement in real disk operation.