Operating Systems Course Project Report — Project 3

Chen Xilun
5090309272
Shanghai Jiao Tong University
chenxl.cs.sjtu@gmail.com

Abstract

In this project, we mainly implement two disk servers — a basic one and an intelligent one with a request queue and several scheduling algorithms, and a file server which use the basic disk server for storage. We also implement some necessary clients which are capable to communicate with these servers to test the correctness and performance of these servers.

1 Introduction

In this report, we mainly focus on the discussion of performance of variant scheduling algorithms for problem 2 and the implementation idea and details of the file system for problem 3.

You may want to read this document before examining the code of problem 3 since its basic ideas are clarified here.

2 Scheduling Algorithms Comparison

2.1 Experiments

We have done two series of experiments to test the performance of different scheduling algorithms under variant request queue lengths.

In the experiments, we choose $1ms$ to be the *track-to-track seek time* and choose $(cylinder\#, sector\#)$ tuple to be $(10, 100)$ and $(20, 200)$ respectively. We in aggregate generate $N = 200$ requests and test a series of settings of request queue length ranged from 1 to 200.

We test three scheduling algorighms: *First-come, First-serve(FCFS)*,*Shortest Seek Time First(SSTF)* and *Circular-Look(C-LOOK)*.

2.2 Results

The running time (in seconds) are presented in Table 1 and Table 2.

Algorithm / Queue length	FCFS	SSTF	C-LOOK
1	0.815	0.81	0.807
3	0.765	0.55	0.673
5	0.748	0.395	0.527
10	0.731	0.236	0.328
20	0.718	0.147	0.198
40	0.725	0.099	0.126
100	0.738	0.06	0.077
200	0.719	0.052	0.053

Table 1: Running time of different algorithms (Cylinder# = 10,Sector# = 100)

Algorithm / Queue length	FCFS	SSTF	C-LOOK
1	1.489	1.48	1.483
3	1.422	1.028	1.299
5	1.406	0.731	1.017
10	1.412	0.449	0.672
20	1.406	0.254	0.362
40	1.415	0.156	0.209
100	1.405	0.088	0.095
200	1.402	0.054	0.078

Table 2: Running time of different algorithms (Cylinder# = 20,Sector# = 200)

Figure 1 shows the performance comparison of different scheduling algorithms under differ-

ent request queue lengths where $(Cylinder\# = 10, Sector\# = 100)$.

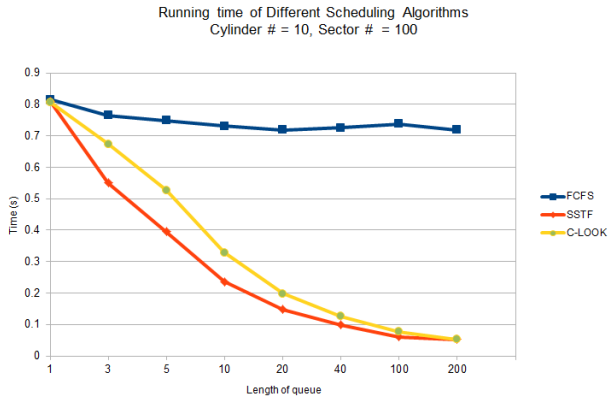

Figure 1: Running time of different algorithms (Cylinder# = 10,Sector# = 100)

Figure 2 shows the running results where $(Cylinder\# = 20, Sector\# = 200)$.

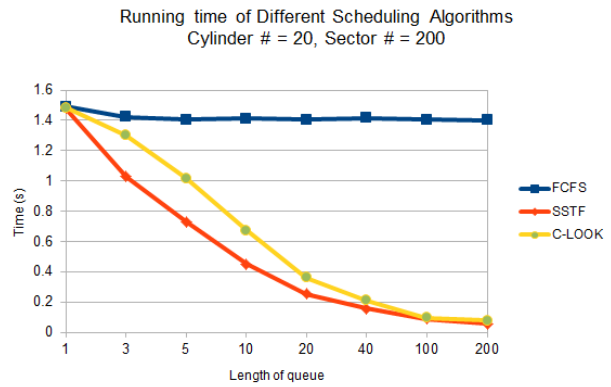

Figure 2: Running time of different algorithms (Cylinder# = 20,Sector# = 200)

Figure 3 and 4 are drawn to scale to show more clearly how the performance are related to the request queue length.

2.3 Analysis

The curves indicate that both SSTF and C-LOOK benefit significantly from the increase of request queue length. And it coordinates with the theory that the performance of FCFS remains relatively constant given increasing request queue lengths.

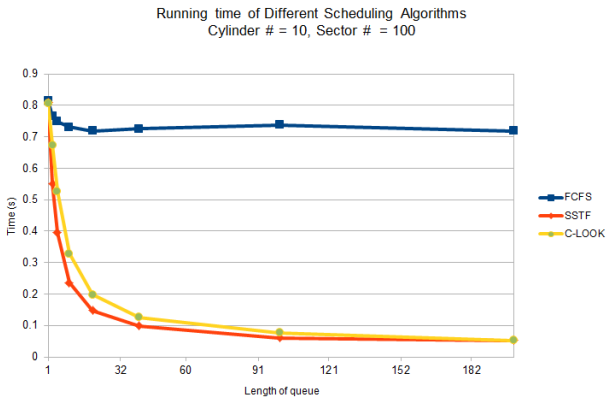

Figure 3: Drawn to scale running time for (Cylinder# = 10,Sector# = 100)

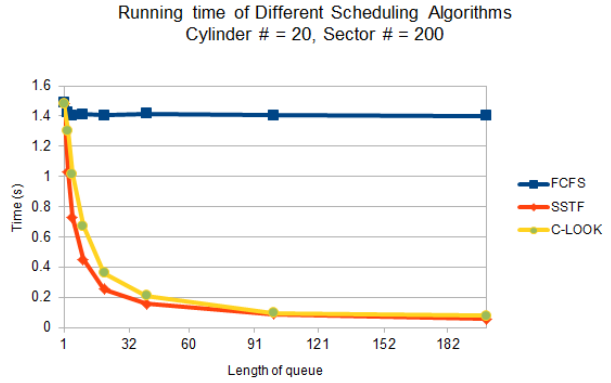

Figure 4: Drawn to scale running time for (Cylinder# = 20,Sector# = 200)

From Figure 3 and 4 we notice that both SSTF and C-LOOK obtain less performance benefit as the queue become longer. There are no significant difference in performance when the request queue length goes beyond 90. On the other hand, a request queue of 20 in length gives a great improvement over the absence of a queue.

When comparing the performance of SSTF and C-LOOK, we infer from Figure 1 and 2 that SSTF outperforms C-LOOK more noticably when the queue length is relatively small. With a queue of 10 in length, SSTF surpasses C-LOOK in performance by 39%(10 cylinders) and 50%(20 cylinders). As the queue become longer, the performance gap between two algorithms are narrowed. With a 100-long queue, SSTF only take a 8%(20 cylinders) ad-

vantage over C-LOOK.

This phenomenon can be readily explained by considering how C-LOOK works. With C-LOOK, the scheduler scans the disk tracks and serves requests in the meanwhile. Every time it reaches the end, it goes back to the startpoint without serving any requests. This introduces overhead of an idle transpassing. Suppose the requests are uniformly distributed along all the tracks. So when the queue is full, the likely scenario is the arm goes from current cylinder to the end of the disk and give an idle transpass and serves the remaining requests. This generates an idle transpass. Consequently, less idle transpasses are introduced when longer queue are used, thus rendering the performance characteristics discussed earlier.

3 File System

This section expounds the implementation ideas and details of the file system in problem 3.

The discussion of our file system mainly consists of two parts: on-disk information and in-memory information. The implementation considerations of on-disk information are of compactness and convenience of manipulation, while the considerations of in-memory information are of execution efficiency and facilitation of coding.

3.1 Disk Block Taxonomy

In this project, we are required to store all the data including the file data as well as the necessary information needed by the file system in the simulated disk server.

In our file system, the blocks of the simulated disk are categorized into 5 types, which are:

Type blocks These blocks stores the type information of all the disk blocks.

FAT blocks These blocks stores the *File Allocation Table*.

Directory blocks These blocks stores the directory infomation, one for a file.

Normal data These blocks stores the ordinary file data.

Free space These blocks are free blocks and available for use.

3.2 Disk Organization

We now present how these blocks are organized in the simulated disk server.

3.2.1 Type Blocks

Type blocks are stored in the beginning of the disk. We use one byte(a char) for each block to store their type information.

We use 't' for type blocks, 'f' for FAT blocks, 'd' for directory blocks, 'n' for normal file data and 'v' for available blocks.

So we allocate the first $typeBlockNum$ blocks to store these type information, where

$$typeBlockNum = \lceil \frac{cNum * sNum}{blockSize} \rceil \quad (1)$$

In addition, we mark redundent bytes in the last type block [1] by 'i' indicating they are invalid to simplify the programming.

3.2.2 FAT Blocks

The FAT blocks resides right after the type blocks. They are similar to the type blocks in terms of organization.

We store an integer for each block to indicate the next data block of it. If it is the last data block of a file, it is set to be -1.

Notice that only normal data blocks have these FAT information, all other blocks have a -1 entry.

We use an int for each entry and allocate $fatBlockNum$ blocks after the type blocks, where

$$fatBlockNum = \lceil \frac{cNum * sNum * sizeof(int)}{blockSize} \rceil \quad (2)$$

3.2.3 Directory Blocks

We use a single block to store the information for a file. We do not store plain text since it is awkward to manipulate. Instead, we store an object of a predefined class directly into the disk.

The class is shown in Listing 1.

```
class File
{
public:
  //Some constructors here
```

[1] We have in total $cNum * sNum$ blocks but the type block can hold as many as $blockSize * typeBlockNum$ entries. There may be some redundency.

```
  char name[244];
  int size;
  int startBlock;
  int dirBlock;
};
```

Listing 1: File Class

This class is tailored to be exactly equal in size to the block size, so that it can be put into one block.

We use the codes in Listing 2 to store it to and read it from the disk.

```
  //Read the dir block #i
  readDisk(res,*i / sNum,*i % sNum);
  File f;
  memcpy(&f,res,sizeof f);

  //Write object f into directory block #
      dir
  File f;
  writeDisk((char*)&f,sizeof(f),dir/sNum,
      dir%sNum);
```

Listing 2: Reading and writing directory blocks

3.3 In-memory Information

These on-disk informations are enough to keep track of the whole file system. For example, if we want to find out the free block list, we just read the type blocks and go through the type information. We then find the list of free blocks.

This method, however, involves a lot of I/O operations and are extremely slow if we adopt such methods to all the operations supported by the file system. Also, this method is hard to program and results in poor code.

For these two considerations, we introduce several additional in-memory informations organized differently to those on-disk ones.

Listing 3 shows a list of such in-memory infomations, which are to be discussed now.

```
  set<int> freeBlocks; //Free blocks list
  set<int> dirBlocks; //Directory blocks
      list
  char* typeTable; //Block type info
  int* fatTable; //FAT Table

  //These two are already discussed
      earlier
  int typeBlock; //# of type blocks
  int fatBlock; //# of FAT blocks
  map<string,File> files; //File lists
```

Listing 3: In-memory informations

typeTable* and *fatTable These two are copies of on-disk type blocks and FAT blocks. The rationale of maintaining such infomation is to reduce the number of I/O operations as well as the complexity of coding.

Without these infomation, every time we need to access these two data, suppose we need to access the FAT of a specific block, we need to first calculate which FAT block it resides in, then read the whole FAT block out and find data for that given block. There may be a number of such actions even in a single file operation. For example, to read a large file, we may access the FAT for many times, which introduces many I/O operations without such in-memory copy of FAT.

These two in-memory copy serves as a *write-back cache* of the type blocks and FAT blocks. When we need to access the type blocks or FAT blocks, we simply read the in-memory array. When modification is done to these blocks, we merely write them to the in-memory array. When a file operation is done, we commit such modifications by synchronizing between the disk blocks and the in-memory arraies. Listing 4 shows the code for synchronization between in-memory and on-disk informations.

```
//Write the in-memory typeTable to disk
void writeTypeTable()
{
  for (int i=0;i<typeBlock;i++)
    writeDisk(typeTable+i*blockSize,
        blockSize,i/sNum,i%sNum);
}

//Write the in-memory FAT to disk
void writeFATTable()
{
  for (int i=typeBlock;i<typeBlock+
      fatBlock;i++)
    writeDisk((char*)fatTable+(i-typeBlock
        )*blockSize,blockSize,i/sNum,i%
        sNum);
}
```

Listing 4: Synchronization between in-memory and on-disk information

freeBlocks* and *dirBlocks We maintain the list of all the free blocks and directory blocks for efficiency. Consider what operations should be supported by these two lists?

For free blocks list, we may want to insert, delete and find the minimum member of it. For the directory blocks list, we may want also want fast insertion, deletion as well as search for one element efficiently. According to such requirements, we use *Black-Red Tree*(Here we use STL *set* container) to maintain these two lists, which can perform insertion, deletion and search in $O(\log n)$ time.

These two additional information can significantly reduce the number of I/O operations and the overall time complexity of the file server. For example, without any in-memory information, finding a free block may take as many as $Cylinder\# * Sector\#$ operations. With a free blocks list implemented by *linked-list*, inserting a free block may take as many as $Cylinder\# * Sector\#$ operations.

files This data structure keeps track of all the files on the disk. It is the collection of all the directory blocks. However, it is organized differently by an STL *map* container. So it maps an entry of directory information to its file name.

This greatly facilitates those operations which need to locate the file by its name efficiently. For instance, to read from a file, we receive the file name as a descriptor. With this data structure, we can locate the directory information of this file within $O(\log n)$ time without any disk operations.

Besides the efficiency improvement, with all these in-memory informations, coding becomes fairly straightforward since many cumbersome disk operations are avoided or collected to be handled together. We would be faced of scattered disk operations all the where without such informations.

Further implementation details can be revealed by viewing the codes.