COMP 304 - Operating Systems Hakan Şahin (64355)
Project 3: Space Allocation Methods

In this project, we were asked to simulate two methods of file allocation, Contiguous and FAT (Linked). We were asked to implement them using only Directory Table (will be referred as DT) and File Allocation Table (will be referred as FAT). We were asked to implement these allocation methods for pre-given inputs that were given with the project. My implementation is also commented in detail, so for further information .java files can be read. My implementation works without any errors.

In the project implementation, I used a total of 9 java classes which are the following in alphabetical order: Block.java, ContiguousAllocation.java (first main class), Directory.java, DirectoryTableContiguous.java, DirectoryTableFAT.java, EntryContiguous.java, EntryFAT.java, FAT.java and FATAllocation.java (second main class).

In order to run my code for a specific input file, all you have to do is locate the attached snippet below in both ContiguousAllocation.java and FATAllocation.java, which are the main methods and change those lines accordingly to which file you want to test:

```
private static final int BLOCK_SIZE = 1024;
private static final String FILE_NAME = "input_1024_200_5_9_9.txt";
```

The example output prints out where a rejection happens (the line), the total amount of rejections and the running time. An example output is given bellow:

(...)

```
ACCESS: Cannot locate the file! a:189:27356
ACCESS: Byte is off limits! a:22:137505
ACCESS: Byte is off limits! a:48:87709
CREATE: Not enough space! c:330480
Create Rejects: 18
Extend Rejects: 201
Shrink Rejects: 104
Access Rejects: 17
Total time: 3010 ms.
```

Rejection Results:

The rejection amounts are same for both Contiguous and FAT Allocation

input_8_600_5_5_0.txt:
Create Rejects: 215
Extend Rejects: 201
Shrink Rejects: 0
Access Rejects: 39

input_1024_200_5_9_9.txt:
Create Rejects: 18
Extend Rejects: 201
Shrink Rejects: 104
Access Rejects: 17

input_1024_200_9_0_0.txt:
Create Rejects: 80
Extend Rejects: 0
Shrink Rejects: 0
Access Rejects: 42115

input_1024_200_9_0_9.txt:
Create Rejects: 0
Extend Rejects: 0
Shrink Rejects: 0
Access Rejects: 0

input_2048_600_5_5_0.txt:
Create Rejects: 213
Extend Rejects: 181
Shrink Rejects: 0
Access Rejects: 52

Average Running Times:

input_8_600_5_5_0.txt

Contiguous Allocation	**FAT Allocation**
runtime 1: 163 milliseconds	runtime 1: 294 milliseconds
runtime 2: 153 milliseconds	runtime 2: 303 milliseconds
runtime 3: 160 milliseconds	runtime 3: 307 milliseconds
runtime 4: 155 milliseconds	runtime 4: 301 milliseconds
runtime 5: 157 milliseconds	runtime 5: 301 milliseconds
average: 157.6 milliseconds	average: 301.2 milliseconds

input_1024_200_5_9_9.txt

Contiguous Allocation	**FAT Allocation**
runtime 1: 2784 milliseconds	runtime 1: 1042 milliseconds
runtime 2: 2705 milliseconds	runtime 2: 1032 milliseconds
runtime 3: 2955 milliseconds	runtime 3: 1104 milliseconds
runtime 4: 2652 milliseconds	runtime 4: 1069 milliseconds
runtime 5: 2725 milliseconds	runtime 5: 1114 milliseconds
average: 2764.2 milliseconds	average: 1072.2 milliseconds

input_1024_200_9_0_0.txt

Contiguous Allocation	**FAT Allocation**
runtime 1: 1018 milliseconds	runtime 1: 1474 milliseconds
runtime 2: 955 milliseconds	runtime 2: 1499 milliseconds
runtime 3: 990 milliseconds	runtime 3: 1447 milliseconds
runtime 4: 982 milliseconds	runtime 4: 1516 milliseconds
runtime 5: 995 milliseconds	runtime 5: 1481 milliseconds
average: 988 milliseconds	average: 1483.4 milliseconds

input_1024_200_9_0_9.txt

Contiguous Allocation	**FAT Allocation**
runtime 1: 82 milliseconds	runtime 1: 260 milliseconds
runtime 2: 79 milliseconds	runtime 2: 257 milliseconds
runtime 3: 89 milliseconds	runtime 3: 232 milliseconds
runtime 4: 83 milliseconds	runtime 4: 270 milliseconds
runtime 5: 100 milliseconds	runtime 5: 286 milliseconds
average: 86.6 milliseconds	average: 261 milliseconds

input_2048_600_5_5_0.txt

Contiguous Allocation	**FAT Allocation**
runtime 1: 173 milliseconds	runtime 1: 283 milliseconds
runtime 2: 154 milliseconds	runtime 2: 295 milliseconds
runtime 3: 151 milliseconds	runtime 3: 337 milliseconds
runtime 4: 185 milliseconds	runtime 4: 300 milliseconds
runtime 5: 173 milliseconds	runtime 5: 285 milliseconds
average: 167.2 milliseconds	average: 300 milliseconds

Questions

1) With test instances having a block size of 1024, in which cases (inputs) contiguous allocation has a shorter average operation time? Why? What are the dominating operations in these cases? In which linked is better, why?

input_1024_200_5_9_9: In this file, FAT Allocation has lower average time than Contiguous Allocation. This should be the case, since there is a lot of extend and shrink operations. Therefore there should be a lot of cases where we need to defragment the directory in order to extend where as in FAT Allocation, there is no need to do defragmentation. Therefore in this case FAT is better since there are a lot of extend and shrink operations which causes defragmentation.

input_1024_200_9_0_0: In this file, Contiguous Allocation has lower average time than FAT Allocation. This should be the case, since the file consists of only creates and accesses. access in Contiguous is cost efficient, where all the information is already given in the DT where as in FAT Allocation, you have to iterate in the FAT in order to tell which block you are trying to access to. Therefore in this case Contiguous is better since access is easier.

input_1024_200_9_0_9: In this file, Contiguous Allocation has lower average time than FAT Allocation. This should be the case, since the file consists of only creates, accesses and shrinks. As explained above, accesses are better in Contiguous since no iteration is needed and for shrinks, there is no need to iterate as well. However both these operations needs iteration in the FAT Data Structure in the FAT Allocation. Therefore in this case Contiguous is better since it also handles shrinks very well.

2) Comparing the difference between the creation rejection ratios with block size 2048 and 8, what can you conclude? How did dealing with smaller block sizes affect the FAT memory utilization?

For the file that has block size of 2048, we have 213 create rejects out of 600, which yields to 35.50% in ratio. For the file that has a block size of 8, we have 215 rejects out of 600, which yields to about 35.83% in ratio.

I haven't realized that FAT memory allocation uses memory from the directory instead of using extra memory until the last day of the project, therefore my results of rejects are same in both Contiguous and FAT.

However, if it was done correctly, I would assume that 8 block size would have more rejects than the 2048, since every fat entry is 4 bytes, where 2 entries would fill a block in 8 block size, where 256 entries would fill a block in 2048 block size.

3) FAT is a popular way to implement linked allocation strategy. This is because it permits faster access compared to the case where the pointer to the next block is stored as a part of the concerned block. Explain why this provides better space utilization.

If we use the other linked allocation mentioned above, we don't lose 4 bytes from Directory everytime we insert a new FAT entry, because the pointer to the next block is stored in the block itself instead of a seperate table that uses the shared memory with Directory. Therefore mentioned linked allocation provides better space utilization.

4) If you have extra memory available of a size equal to the size of the DT, how can this improve the performance of your defragmentation?

Since we defragment the directory in the contiguous allocation, and we shift it only one by one in order to not lose data. If I had extra memory available, I would be able to store some of the content in the directory to the extra memory and improve or maybe completely remove shifting.

5) How much, at minimum, extra memory do you need to guarantee reduction in the number of rejected extensions in the case of contiguous allocations?

My Contiguous Allocation method only rejects extension when there is not enough space available in blocks in the directory. Therefore any extra memory that is given in the directory (instead of 32768, higher values than 32768) would reduce the number of rejected extensions.