COMP304 – Project3 Report

Tolga Sümer – 64534

Implementation:

Directory Table:

* createFile

If there is enough block space for the entry, it adds it.

Else if there is not enough space in the part that we want to add, but there is a hole somewhere else that the block can fit in, it adds it there.

Else defragmentation happens and blocks are added without any holes and the file is added after those blocks.

If there is not enough space, it prints an error message.

* access

It checks the file\_id to see that if the file exists.

If exists, it gets the start position of that block and iterates offset times and returns the address of the block.

* extend

If there is enough space after the last block it extends the block desired times.

If there is not enough space at the end, but there is enough space throughout the memory, it does defragmentation and then adds the blocks after the last block.

* shrink

If the file is in a single block, it shrinks it desired times. But if the file is defragmented along the memory, it goes to its last position and shrinks it one by one desired times.

File Allocation Table:

* createFile

It creates “length” count of blocks and sets their blockID one by one and points each block to the next block. It sets the last block’s next pointer to -1.

* access

It checks if offset is bigger than the file size. If not, it goes to next block one by one offset times and returns the address of the desired block.

* extend

If there is enough space to extend the file, it adds the next block to the end of the last block and sets the last block of the file to the first extended block and adds desired number of blocks. It sets their pointers and sets the last pointer to -1.

* shrink

It finds the last block of the desired file. Then it shrinks the last desired number of blocks.

Directory Content:

* It keeps the values of DT and FAT.

For a certain input file, the block size is indicated after the ’input ’. The input files contain an operation in each line:

• Each line is colon-separated.  
• A line ’c:bytes’ is a create call.  
• A line ’a:file id:offset’ is an access call.  
• A line ’e:file id:extension blocks’ is extension.

• A line ’sh:file id:shrinking blocks’ is shrinking.

Missing Implementation:

* I just did defragmentation not compaction for contiguous allocation.

Experimentation and Analysis:

For input: input\_8\_600\_5\_5\_0.txt

The number of files that their creation is rejected: DT=214, FAT=214

The number of files that their extension is rejected: DT=0, FAT= 197

The average running time of an operation:

Create:

DT = (227 + 257 + 216 + 254 + 239)/5 = 238.6 ns

FAT = (428 + 410 + 411 + 402 + 407)/5 = 411.6 ns

Access:

DT = (0 + 1 + 0 + 0 + 2)/5 = 0.6 ns

FAT = (13 + 9 + 19 + 12 + 10)/5 = 12.6 ns

Extend:

DT = (22 + 38 + 34 + 25 + 33)/5 = 30.4 ns

FAT = (276 + 275 + 295 + 244 + 232)/5 = 264.4 ns

Shrink:

DT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

FAT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

For input: input\_1024\_200\_5\_9\_9.txt

The number of files that their creation is rejected: DT = 23, FAT=9

The number of files that their extension is rejected: DT = 461, FAT=103

The average running time of an operation:

Create:

DT = (19 + 14 + 13 + 19 + 24)/5 = 17.8 ns

FAT = (800 + 822 + 842 + 803 + 820)/5 = 817.4 ns

Access:

DT = (2 + 2 + 2 + 2 + 1)/5 = 1.8 ns

FAT = (10 + 14 + 13 + 17 + 9)/5 = 12.6 ns

Extend:

DT = (78 + 89 + 84 + 69 + 83)/5 = 80.6 ns

FAT = (2956 + 3181 + 3228 + 3217 + 3157)/5 = 3147.8 ns

Shrink:

DT = (39 + 38 + 53 + 42 + 49)/5 = 44.2 ns

FAT = (530 + 500 + 521 + 462 + 541)/5 = 510.8 ns

For input: input\_1024\_200\_9\_0\_0.txt

The number of files that their creation is rejected: DT = 80, FAT = 80

The number of files that their extension is rejected: DT = 0. FAT = 0

The average running time of an operation:

Create:

DT = (25 + 35 + 30 + 24 + 21)/5 = 27 ns

FAT = (494 + 533 + 513 + 609 + 538)/5 = 537.4 ns

Access:

DT = (25 + 23 + 19 + 17 + 25)/5 = 21.8 ns

FAT = (871 + 956 + 914 + 1208 + 905)/5 = 970.8 ns

Extend:

DT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

FAT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

Shrink:

DT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

FAT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

For input: input\_1024\_200\_9\_0\_9.txt

The number of files that their creation is rejected: DT = 45, FAT = 0

The number of files that their extension is rejected: DT = 0, FAT = 0

The average running time of an operation:

Create:

DT = (72 + 66 + 70 + 75 + 73)/5 = 71.2 ns

FAT = (298 + 312 + 302 + 335 + 323)/5 = 314 ns

Access:

DT = (22 + 11 + 6 + 12 + 12)/5 = 12.6 ns

FAT = (9 + 12 + 13 + 12 + 26)/5 = 14.4 ns

Extend:

DT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

FAT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

Shrink:

DT = (29 + 17 + 22 + 29 + 23)/5 = 24 ns

FAT = (46 + 67 + 68 + 105 + 96)/5 = 76.4 ns

For input: input\_2048\_600\_5\_5\_0.txt

The number of files that their creation is rejected: DT = 213, FAT = 213

The number of files that their extension is rejected: DT = 0 FAT = 182

The average running time of an operation:

Create:

DT = (193 + 183 + 198 + 169 + 178)/5 = 184.2 ns

FAT = (383 + 394 + 423 + 415 + 387)/5 = 400.4 ns

Access:

DT = (1 + 0 + 0 + 2 + 1)/5 = 0.8 ns

FAT = (15 + 12 + 16 + 11 + 33)/5 = 17.4 ns

Extend:

DT = (28 + 27 + 36 + 29 + 34)/5 = 30.8 ns

FAT = (224 + 213 + 221 + 222 + 222)/5 = 220.4 ns

Shrink:

DT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

FAT = (0 + 0 + 0 + 0 + 0)/5 = 0 ns

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?

In all of the cases contiguous allocation has a shorter average operation time. Because of accessing is faster in contiguous allocation, finding the block and shrinking and extending it is faster. Creation and extension are the dominating operations. Because of my implementation, no operations are better in linked allocation.

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?

Creation rejection when block size is 2048 = 814

Creation rejection when block size is 8 = 1414

So, dealing with small block size decreases FAT memory utilization.

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.

FAT provides a better space utilization because with the advantage of linked allocation even a single free block between the used blocks can be linked and allocated to a file. This approach doesn't suffer from external fragmentation.

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?

If there is more memory, there would be less suffering from external fragmentation, thus defragmentation would be faster.

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?

We need at least extra memory available of a size equal to the size of the DT.