Comp 304: Project 3

Space Allocation Methods

Sitare Arslantürk, 57677

Implementation

The linked and contiguous implementations are the child classes of FileSystem overriding its built-in methods. As a result, each implementation has its own create_file, access, shrink and extend methods. Each of the input files were tested 5 times to get an average time per operation for each of these methods implemented. The results can be found in the experimentation and analysis part. All parts of the project are working.

The Contiguous Implementation

In contiguous implementation, the main difference is that it does not use FAT. It gives all the space to the directory content.

In create_file, I check if there is enough space available to allocate the file into the directory, if not the creation is rejected and the rejection counter is incremented. If there is enough space, then this space is checked for contiguousness. If the space is contiguous, the file is allocated into the directory content. If not, then defragmentation is performed. Defragmentation handles pushing all the files to the left in order to remove the spaces between them. Then the file is allocated to the opened space. The directory table is updated accordingly with the start index and file length.

In access, the method returns the location of the byte given the byte offset. The byte offset is calculated to which block of the file it corresponding in the directory content then that index is returned. This is quite fast for the contiguous implementation compared to the linked.

In extend, the function extends the given file by the number of extension blocks. First, the space corresponding to the extension block number is checked. If such a space does not exist, then the call is rejected and the rejection counter is incremented. If there is enough space to extend the file, then the space is checked for contiguousness. If contiguous, then the extension is allocated starting from the end of the file. If the space is not contiguous, then first defragmentation is performed. All the blocks are pushed from high to low indices which means pushing to left. Then afterwards the files coming after the file with the given file id are moved by the amount of extension to the right in order to open up space at the end of the file with the extension to be performed. With the empty contiguous space available, the file is allocated its extensions at the directory content. Finally, the directory table for the files are updated with respect to their start index and the new file length due to the extension.

In shrink, the file with the given file id is shrunk by the number of shrinking blocks which basically removing it from the directory content. The shrink method goes to the last index of the file and starts removing from the last index. If the shrink amount is bigger than the file length then the shrink call is not performed. Afterwards, the directory table is updated with the new file length.

The Linked Implementation

In linked implementation, the main difference is the usage of FAT. There should be space allotted for the FAT from the directory content's space. In order to do this, we are given the 4 bytes which is the amount of the space it will occupy and with the formula

$$space = \frac{fixed\ block\ list\ size * input\ file's\ block\ size}{4 + input\ file's\ block\ size}$$

We can find the space we can allocate to the directory content. This will cause for a smaller space than contiguous implementation's. In addition due to the formula, as the block size increases, the space left after FAT increases.

In create_file, I check if there is enough space available to allocate the file into the directory, if not the creation is rejected and the rejection counter is incremented. If there is enough space, then the file is allocated to the empty spaces of the directory content. The directory table is updated accordingly with the start index and file length. The FAT is updated in a fashion that the current index shows the next block of the file.

In access, the function traverses FAT in order to find the corresponding byte offset's block. The corresponding block is traversed until the index is found and returned. This is slower than the contiguous implementation, due to the traversal.

In extend, first the space check is performed. If extension space is not available, then the call is rejected and its counter is incremented. If the sufficient space is available then the extension amount is allocated into the directory content to wherever is empty, regardless of contiguousness. The previous -1 in the FAT now links to the first index of the extension. The ending of the file which is the end of the extension now gives -1 to show the file ends here. The linkage between the blocks are defined in the FAT. In FAT, one can find the blocks of the file after looking up the file's starting index from the directory table and then following the FAT until the index gives -1, which is the end of the file. The FAT holds the indexes that point to the next block place of the file. In the end, the FAT, the directory table are updated accordingly with the directory content placements.

In shrink, it has to traverse the FAT in order to get to the last block because shrinking is performed starting from the last block of the file. The contents of the shrunk directory content blocks are set to 0. The FAT is updated so that the amount of shrink is released and does not point to any other index. The last index of the file should now point to -1 in order to show the end of the file. The FAT and directory table are updated accordingly with the new file length and the removed directory content elements.

Experimentation and Analysis

1. Input_8_600_5_5_0

a. Contiguous

Total rejection for create call = 215

Total rejection for extend call = 201

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	3.7	2475.2	125918.9	0
Run2	3.7	2456.1	126374.1	0
Run3	4	2524.9	138281.1	0
Run4	4.3	2565.7	139762.7	0
Run5	3.9	2474.2	125638.4	0
Average	3.92	2499.22	131195.04	0

b. Linked

Total rejection for create call = 333

Total rejection for extend call = 313

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	10.7	34632.6	14065.7	0
Run2	10.8	33836.1	13863.3	0
Run3	10.7	34498.9	14268.9	0
Run4	11	33711.6	13.891.4	0
Run5	10.7	34547.5	14139.6	0
Average	10.78	34245.34	14084.375	0

2. Input_1024_200_5_9_9

a. Contiguous

Total rejection for create call = 18

Total rejection for extend call = 201

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	2.7	52765.5	19828	4.5
Run2	2.9	59676.3	20214.6	4.5
Run3	2.9	59048	20295.7	4.4
Run4	2.8	56556.3	19687.5	4.6
Run5	2.8	62550.4	19680	4.4
Average	2.82	58119.3	19941.16	4.48

b. Linked

Total rejection for create call = 16

Total rejection for extend call = 180

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	25.1	184602.2	159214	2409.5

Run2	24.7	183253.9	158600.5	2375.8
Run3	25	175701.2	159763.8	2468.1
Run4	23.7	176741.8	154901.1	2329.6
Run5	25.7	182931.6	164649.2	2425.8
Average	24.84	180646.14	159425.72	2401.76

3. Input_1024_200_9_0_0

a. Contiguous

Total rejection for create call = 80

Total rejection for extend call = 0

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	2.5	3143.9	0	0
Run2	2.5	3191.8	0	0
Run3	2.5	3125.5	0	0
Run4	2.5	3380.8	0	0
Run5	2.4	3106.7	0	0
Average	2.48	3189.74	0	0

b. Linked

Total rejection for create call = 79

Total rejection for extend call = 0

Column1	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	23.6	195674.3	0	0
Run2	22.7	196379.6	0	0
Run3	22.4	199872.7	0	0
Run4	22.1	198054.8	0	0
Run5	23.3	202702.7	0	0
Average	22.82	198536.82	0	0

4. Input_1024_200_9_0_9

a. Contiguous

Total rejection for create call = 0

Total rejection for extend call = 0

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink

Run1	2.4	166835.9	0	10.9
Run2	2.6	166627.3	0	11.9
Run3	2.4	168271.2	0	11.5
Run4	2.7	165362.4	0	11.8
Run5	2.4	164357.9	0	10.8
Average	2.5	166290.94	0	11.38

b. Linked

Total rejection for create call = 0

Total rejection for extend call = 0

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	6.8	86555.5	0	882.6
Run2	6.5	88379	0	849.6
Run3	6.5	85695.8	0	862.5
Run4	6.8	84531.8	0	853.5
Run5	6.7	88639	0	860.3
Average	6.66	86760.22	0	861.7

5. Input_2048_600_5_5_0

a. Contiguous

Total rejection for create call = 213

Total rejection for extend call = 181

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	3.5	2579.8	128437.9	0
Run2	3.6	2555.6	127541.5	0
Run3	3.6	2578.4	126763.5	0
Run4	3.6	2553.7	124836.9	0
Run5	3.6	2655.2	125976.8	0
Average	3.58	2584.54	126711.32	0

b. Linked

Total rejection for create call = 212

Total rejection for extend call = 178

microseconds	Time per Access	Time per Create	Time per Extend	Time per Shrink
Run1	11	76654.7	31662.2	0
Run2	10.7	76052.6	31518	0
Run3	10.8	75307	31072.1	0
Run4	11.5	76096.9	31393.5	0
Run5	11.1	78158.4	32301.8	0
Average	11.02	76453.92	31589.52	0

Questions

1. In input_1024_200_9_0_0, there are no shrink or extend operations. Thus, contiguous performs the best in this output. The extend operation's defragmentation causes it to perform slower compared to the linked implementation. Also, another factor is that access calls are slower in linked implementation due to the traversal of the FAT which allows the contiguous implementation to take the upper hand because it does not need to traverse, it directly accesses. The contiguous operation performs the worse in input_1024_200_5_9_9 due to the usage of all four operations. In input_1024_200_9_0_9, the contiguous again performs worse, this is I believe is the result of the defragmentation implemented within create because there are no extend operations in this input. We can conclude that linked is better with no access call due to traversal and has the upper hand in create and extend due to defragmentation.

2. The rejection ratios of the block size of 8 is higher than the rejection rations of block size of 2048. This is due to the space usage that was left on the behalf of directory content. As the number of block sizes got smaller, FAT consumes less memory, leaving more space for the directory content. This means more create and extend operations and less rejections. This is again due to the formula generated which is

$$space\ left\ after\ FAT = \frac{fixed\ block\ list\ size * input\ file's\ block\ size}{4 + \ input\ file's\ block\ size}$$

   The fixed block list size is given as 32768. The input files's block size is given to us with the input file. When the input file size is 8, the space left after FAT utilization is 21.845 of the 32768 that was given to us. With input size, 2048, this becomes 32704 of the 32768. This shows that the FAT utilization is much smaller when the block size is bigger.

3. The case in which pointer to the next block is stored as a part of the concerned block is costly in terms of space utilization. This is because each pointer is supposed to be stored inside block. This consumes block space and the file needs more blocks to be allocated due to this additional space held by the pointer. However, with the implementation we performed, the links are stored in the FAT which takes space from the directory content. Although both might seem like space

consuming, our implementation was more efficient in the way that it held all the pointing indexes together, all in one place which is the FAT. So we only need to spare some space for the FAT rather than allocating pointer space in every block.

4. Currently, the defragmentation was implemented with no additional data structures. This additional space means we could have stored all the empty places within this space and could have performed the defragmentation in a much faster way using this additional space. We could have used an additional directory table to hold only the empty indexes and allocate directly into that space which could have taken direct access O(1) and linear time complexity to allocate.

5. Since we perform defragmentation, we only need the amount of extension space in order to perform the extension. So, if we had the amount of extension minimally, we can perform the extend call. The extend calls give us the number of blocks, so the memory we need would be the number of blocks to extend times the input file's block size in bytes. We can also check the previous shrink operations to decide if we need any additional memory. If the shrink operations is to be performed and is not given an amount bigger or equal to the file length, then it would empty up space equal to the extension then it is guaranteed that the extend function can be called and perform the extension. All of these potentially, will lower the rejection rates.