Ege Erdogan 64004 - COMP 304 Project 3 Report

Questions

1. When the block size is 1024, considering the total average completed operation time, contiguous allocation is faster in two inputs (`input_1024_200_9_0_9` and `input_1024_200_9_0_0`), and linked allocation is faster in one input (`input_1024_200_5_9_9`). For the inputs which contiguous allocation performed faster with, there are no extend operations. Creation seems to take a long time in linked allocations. For the input linked allocation performed better with, all operations are performed. Contiguous allocation performed worse on all operations except accessing, and especially the create and extend operations took longer than other operations. Faster access is expected in contiguous allocation because once we know the block offset, we can make a single access to the `storage` array to obtain the block, while with linked allocation, we have to iterate through the FAT.
2. (Comparing 2048 and 8-byte block sizes since there is no 32-byte experiment.) With contiguous allocation, there was not much of a difference. Of a total 3000 creation requests, 385 were accepted with with 8-byte blocks and 387 were accepted with 2048-byte blocks. With linked allocation, the difference was higher. 204 requests were accepted with 8-byte blocks, and 388 were accepted with 2048-byte blocks. This is because the FAT requires 250 times more blocks when the block size is 8 bytes compared with 2048 bytes. The extra free space with 2048-byte blocks results in a higher amount of accepted creation requests.
3. In both cases, the head would need to perform the same number of total seeks, say n, until the desired block. With a FAT however, the total distance, and hence the time of the seeks would be shorter since all of the first n-1 seeks would be in a confined region of the disk containing the FAT. The only long seek is performed to reach the desired block. If each block contained a pointer, then the seek distances could be much longer, and take more time.
4. When performing defragmentation, the DT should also be updated at the same time. If the DT can entirely be stored in memory, then the updates will be much faster compared to the case when the DT is partly in memory and partly in secondary storage, because there will be no need to access the secondary storage device (to update DT) until the very end.
5. Due to external fragmentation, it might be the case that there is enough total space to extend a file, but every block is occupied. Then, if we had memory the size of a single block, we could transfer the contents of the blocks with the highest external fragmentation (less content, more empty space) to memory and free multiple blocks. This would reduce the number of rejected extensions since we would be creating new free blocks.

Explanation

All parts of the project work as expected. Below is an explanation of how the four main functions are implemented for both allocation methods.

There is a high number of helper methods to make the main code more readable. I made their names as self-explanatory as possible. Additional information can be found as comments if what they do is unclear.

Contiguous Allocation

Create: create a new file, first the number of blocks needed is calculated. Then, the `haveContiguousSpace` function returns the index of the first block of the first contiguous space (first fit strategy). If there is no space, compaction is performed, and it is checked again whether there is a contiguos block of space or not. Allocation works by assigning each block's index as its value in a sequential manner.

Access: First, given the byte offset, the offset of the block which the target byte is in calculated relative to the starting block of the file. The index is returned.

Extend: Extension is one of the more complicated operations, mainly because of the fact that it may require additional operations to ensure that a file is extended if there is enough space. Here is how the algorithm works:

```
1  n: number of blocks to extend
2  file: file to extend
3  
4  IF there is enough total space:
5    IF there is enough contiguous space after the end of the file:
6      PERFORM EXTENSION
7    ELSE:
8      for each block after the file's end:
9        shift it back file.length times
10       IF there is enough contiguous space after the end of the file:
11         PERFORM EXTENSION and return
12 
13     perform compaction
14     IF file can be extended
15       PERFORM EXTENSION
16     ELSE
17       raise error
18 
```

The shift back operation at lines 8-9 can also be thought of as moving the file forward one block at a time.

If there is a contiguous block of space somewhere after the file's end block, then the file will eventually reach that space and the condition at line 10 will be satisfied. Notice that no file remains split after this operation because it terminates when there is empty space after the file.

If there is enough total space, but not contiguously, then the condition at line 10 will never be satisfied, but the compaction operation at line 13 will make sure that that space follows the file's end. Since the file is the right-most file after the shifting back, and compaction only moves blocks from high to low indices, it will remain in the end. The single block of empty space will be following the file's end.

Shrink: Shrinking is straighforward with contiguous allocation. Given number of blocks at the end of the file are deallocated. An error is raised if the file is shrinked to zero or less length.

Linked Allocation

In linked allocation, there are two data structures.

- `directoryEntries` Integer to integer hash map that stores the directory entries. Keys are ids of the files and values are the start indices.
- `FAT`: The file allocation table is also an integer to integer hash map. Keys and values are block indices. A (key, value) entry signifies that the `value` is the next block after `key`.

Create: It is first checked if there is enough total space available. If so, for each block to be added, the first free index is found by searching from the beginning of the `storage` array, and it is allocated to the file, updating the FAT.

Extend: Extension works in a similar way to creation. It is checked if there is enough space, and if so, free blocks are allocated to the file by searching them from the start.

Access: The offset of the block from the starting index of the file is calculated, and the FAT is followed that offset many times. The index of the reached block is returned.

Shrink: Shrink is the most complicated operation for linked allocation. It works by first calculating the the new end block of the file. Afterwards, all the blocks of the file following that block are deallocated. Another method could be to start from file's end and work backwards, but that would be harder to implement since FAT links are unidirectional.

Results

The results from the experiments are as follows:

Contiguous Allocation

```
input_1024_200_5_9_9.txt
Experiment results from file: input_1024_200_5_9_9.txt
  Total completed operation counts:
    Create: 384
    Extend: 2587
    Access: 850
    Shrink: 3635
  Average operation times (ms):
    Create: 3.9375
    Extend: 4.564746810977967
    Access: 0.001176470588235294
    Shrink: 5.502063273727648E-4
    TOTAL:  1.7870171673819744
  Creations rejected: 616
  Extensions rejected: 4388

input_1024_200_9_0_9.txt
```

```
Experiment results from file: input_1024_200_9_0_9.txt
  Total completed operation counts:
    Create: 821
    Extend: 0
    Access: 8025
    Shrink: 3429
  Average operation times (ms):
    Create: 0.7661388550548112
    Extend: NaN
    Access: 1.2461059190031152E-4
    Shrink: 0.0
    TOTAL:  0.05132382892057027
  Creations rejected: 179
  Extensions rejected: 0

input_1024_200_9_0_0.txt
Experiment results from file: input_1024_200_9_0_0.txt
  Total completed operation counts:
    Create: 122
    Extend: 0
    Access: 1837350
    Shrink: 0
  Average operation times (ms):
    Create: 0.00819672131147541
    Extend: NaN
    Access: 4.680654203064196E-5
    Shrink: NaN
    TOTAL:  4.73476602636666E-5
  Creations rejected: 878
  Extensions rejected: 0

input_8_600_5_5_0.txt
Experiment results from file: input_8_600_5_5_0.txt
  Total completed operation counts:
    Create: 385
    Extend: 354
    Access: 2755
    Shrink: 0
  Average operation times (ms):
    Create: 0.007792207792207792
    Extend: 7.3841807909604515
    Access: 3.629764065335753E-4
    Shrink: NaN
    TOTAL:  0.7492844876931883
  Creations rejected: 2615
  Extensions rejected: 2421

input_2048_600_5_5_0.txt
Experiment results from file: input_2048_600_5_5_0.txt
```

```
  Total completed operation counts:
    Create: 387
    Extend: 354
    Access: 2750
    Shrink: 0
  Average operation times (ms):
    Create: 0.007751937984496124
    Extend: 6.161016949152542
    Access: 3.6363636363636E-4
    Shrink: NaN
    TOTAL:  0.6258951589802348
  Creations rejected: 2613
  Extensions rejected: 2321
```

Linked Allocation

```
input_1024_200_5_9_9.txt
Experiment results from file: input_1024_200_5_9_9.txt
  Total completed operation counts:
    Create: 385
    Extend: 2630
    Access: 406
    Shrink: 3606
  Average operation times (ms):
    Create: 0.43636363636363634
    Extend: 0.28669201520912546
    Access: 0.0024630541871921183
    Shrink: 0.004714364947310039
    TOTAL:  0.13376974526825103
  Creations rejected: 615
  Extensions rejected: 4345

input_1024_200_9_0_9.txt
Experiment results from file: input_1024_200_9_0_9.txt
  Total completed operation counts:
    Create: 825
    Extend: 0
    Access: 7510
    Shrink: 3457
  Average operation times (ms):
    Create: 1.0218181818181817
    Extend: NaN
    Access: 2.663115845539281E-4
    Shrink: 0.0014463407578825572
    TOTAL:  0.07208276797829037
  Creations rejected: 175
  Extensions rejected: 0

input_1024_200_9_0_0.txt
```

```
Experiment results from file: input_1024_200_9_0_0.txt
  Total completed operation counts:
    Create: 121
    Extend: 0
    Access: 1849020
    Shrink: 0
  Average operation times (ms):
    Create: 1.1570247933884297
    Extend: NaN
    Access: 0.0014348141177488616
    Shrink: NaN
    TOTAL:  0.0015104310596109221
  Creations rejected: 879
  Extensions rejected: 0

input_8_600_5_5_0.txt
Experiment results from file: input_8_600_5_5_0.txt
  Total completed operation counts:
    Create: 204
    Extend: 193
    Access: 1935
    Shrink: 0
  Average operation times (ms):
    Create: 0.4117647058823529
    Extend: 0.16580310880829016
    Access: 0.0
    Shrink: NaN
    TOTAL:  0.04974271012006861
  Creations rejected: 2796
  Extensions rejected: 2582

input_2048_600_5_5_0.txt
Experiment results from file: input_2048_600_5_5_0.txt
  Total completed operation counts:
    Create: 388
    Extend: 357
    Access: 2720
    Shrink: 0
  Average operation times (ms):
    Create: 0.45618556701030927
    Extend: 0.17647058823529413
    Access: 0.0014705882352941176
    Shrink: NaN
    TOTAL:  0.07041847041847042
  Creations rejected: 2612
  Extensions rejected: 2318
```