File System Defragmentation: A Modification of Assignment 3

Christopher Hittner
I pledge my honor that I have abided by the Stevens Honor System.

May 2, 2017

1 Introduction

Suppose that we have a filesystem with contiguous allocation block tuple list $B = \{(B_{l,0}, B_{h,0}) \forall i < |B|\}$, where $B_i = (l_i, h_i) \in B$ is a tuple representing allocation of blocks in $[l_i, h_i)$ requiring that $l_i < h_i$. We seek to modify B to create D such that:

$$|D| = 1$$

All of the allocated memory is grouped into a single block

$$\sum_{i=0}^{|B|} (B_{h,i} - B_{l,i}) = D_h - D_l$$

The number of blocks allocated in B is equal to the number of blocks allocated in D

Furthermore, we suppose that we have a set of files allocations F, where F_i is the list of blocks allocated to file i and the values in F_i are ordered by the location in the document. We seek to modify F to create F' such that:

$$\forall f \in F', f \text{ is an ordered list}$$

$$\sum_{F_i'} \sum_{n=0}^{|F_i'|-2} (F_i'[n+1] - F_i'[n]) \text{ is minimized}$$

All of the blocks in the file are as close together as possible

$$\forall i \forall b \in F_i', b \in [D_l, D_h)$$

Every block allocated to a file will be in the new block allocation

$$|F| = |F'|$$

Each file will still exist; none will be created, and none will be removed

$$\forall i, |F_i| = |F_i'|$$

Each file will have the same amount of memory as before

So, the intention is to minimize the external fragmentation so that it is easy to allocate contiguous blocks on the disk. Furthermore, we seek to be able to align the blocks of the individual files such that they can be accessed sequentially.

2 Sort and Drop Approach

The first condition that is handled is the ordering of the blocks allocated to each file. We require that the list of blocks be in increasing order, so that the blocks can be accessed in sequence. This requires that the data be sorted in order of the block address in memory. To sort, we can use an algorithm such as mergesort or quicksort, which has a time complexity of $O(n \log n)$. This operation would be performed on every file on the disk, resulting in a time complexity of $O(|F| b \log b)$, where b is the number of blocks in a file. This operation will merely change the ordering of block usage, and thus will meet all other conditions for the file state.

The second condition that is handled is the grouping of the memory blocks into a single sector. To do this, the defragmenter keeps store of the lowest block that needs to be added to the set of allocated blocks. We call this number p. We check through each block b of each file f in F, and replace the block b with p if $p<b$. Over the entire set of blocks, this will always shrink the number of fragment blocks because there exists a block $b*$ that is the largest allocated block. If there is at least one fragment, then $b*$ will be replaced by the smallest fragment, therefore shrinking the total number of fragments. Furthermore, the block ordering within each file will be preserved, since if the ith block is the first block to be after the first fragment, replacing it with the fragment will preserve the order since the fragment is greater than the previous item, if one exists. Overall, in a system with a list of fragments G and list of allocated blocks K, the time complexity of this part is $O(|G||K|)$.

In total, the worst case for the algorithm is $O(|K| \log |K| + |G||K|)$, since having only one file maximizes the sorting time. We can put a further worst case at $O(|K|^2)$, since if the fragments and allocated blocks occupy equal amounts of the empty disk, the time required to complete depends on the square of the disk size.

In the pseudocode, there are two main loops: one that sorts the blocks, and one that redistributes the blocks. The file block sorting ensures that each file block can be accessed sequentially in the memory structure (ex: Accessing 1, 2, 4 instead of 4, 1, 2). The sector rearrangement ensures that the entire system is arranged into a single block, in order to maximize available space and linearity of files. It also maintains the in-file ordering because it will not replace a block unless the number decreases.

A flaw with this technique is that it does not minimize the distance between consecutive blocks. Rather, it seeks to simply reduce the number of fragments to zero with no regard to the spacing between blocks.

See Algorithm 1 for the pseudocode.

3 Block Building Approach

An alternative solution that does minimize sequential value distance involves using a similar method to bring items closer together. However, it adds a step that moves every file into a single block.

The first step is to move all of the files blocks into a single contiguous block. This will take care of reducing the external fragmentation to zero. This can be done in the same fashion as the previous algorithm, since it does not rely on the file blocks being ordered. This step can be completed in $O(|G||K|)$ time.

The next step is to put all of the files into consecutive blocks. This requires finding the file with the lowest number and bringing all of its other blocks into adjacency with it. This requires going through at most every allocated block (an $O(|K|)$ operation), followed by moving optimal blocks into the frontal file f (upper bounded by $O(|K|^3)$). The end result is that for each file f, its set of blocks will be contiguous.

The final step is to sort all of the files' blocks to be in order on the system. This will guarantee that accessing any file will be sequential. Like before, this will be $O(|F| n \log n)$. The total effect is that the algorithm is $O(|K|^3)$, which offers a significantly lower time advantage over Sort and Drop. This is especially troublesome when dealing with large datasets, since systems with millions of blocks could take thousands of times longer to compute, if not worse.

The advantage of Block Building over Sort and Drop is that it moves data blocks into continuous sectors. This allows for the disk to read the files in a far more efficient manner, especially when doing sequential reads. Since it uses parts of Sort and Drop to complete the task, it is expected that Block Building take at least as long. However, since it must compare every file with every other file to swap blocks, it loses the time advantage that Sort and Drop has.

See Algorithm 2 for the pseudocode.

4 Analysis

To test the program, I modified the file loading code for the assignment to randomly generate fragments. This was done by adding and removing from each file on the disk one by one. I then ran 'defrag' using each of the two algorithms.

With Sort and Drop, the file system would complete in roughly 28 seconds. In five trials, I recorded 30.91, 27.72, 27.40, 26.55, and 27.34 seconds as the completion time.

Block Building took significantly longer. In one trial, I started the program around midnight, and found it running at 1 AM, but not 9 AM. A more formal test revealed that it took roughly four hours and thirty one minutes to complete.

These findings correlate with the relative scale of the time complexities of the algorithm, in that Block Building requires a degree of time more than Sort and Drop, which in this case resulted in the program requiring nearly six hundred times more time to complete.

5 Conclusion

Overall, each algorithm was able to reduce the number of external fragments to zero. However, it proved to be much faster to ignore making each file's blocks consecutive. Whereas moving blocks would only require a matter of seconds, reorganizing each file's block allocations requires a huge amount of time to complete. The sacrifice is that the files would be sorted, but would not be sequential.

To replicate my findings, one can use the simulated filesystem to defragment. The command can be invoked with "defrag". If no arguments are given, the program will automatically use Sort and Drop to defragment. Given "1", it will use Sort and Drop, and given "2", it will use Block Building. Quotations are not part of the command.

Algorithm 1: Sort And Drop

Input: Allocation tuples B, File set F
Output: New allocation D
Result: Each f has been modified, if necessary, to fit D
`/* Sort each list of files */`
foreach $f \in F$ **do**
  $B_f \leftarrow$ block list of f
  $sort(B_f)$
end
`/* Get the set of allocated blocks */`
$D \leftarrow \emptyset$
foreach $(i, j) \in B$ **do**
  $S \leftarrow \{i, i+1, \ldots, j-1\}$
  $D \leftarrow D + S$
end
`/* Iteratively move block allocations downward */`
$lo \leftarrow$ Smallest block number such that $lo \notin D$
while $D_{|D|-1} \neq |D| - 1$ **do**
  foreach $b \in D$ **do**
    if $b > lo$ **then**
      `/* Replace b with lo */`
      $B_f \leftarrow$ block list containing b
      $D \leftarrow D + \{b\} - \{lo\}$
      $i \leftarrow$ index of b in B_f
      $B_f[i] = lo$
      $lo \leftarrow$ Smallest block number such that $lo \notin D$
    end
  end
end
return D

Algorithm 2: Block Building

Input: Allocation tuples B, File set F
Output: New allocation D
Result: Each f has been modified, if necessary, to fit D

```
/* Get the set of allocated blocks                                    */
```
$D \leftarrow \emptyset$
foreach $(i, j) \in B$ **do**
  $S \leftarrow \{i, i+1, \ldots, j-1\}$
  $D \leftarrow D + S$
end
```
/* Iteratively move block allocations downward                        */
```
$lo \leftarrow$ Smallest block number such that $lo \notin D$
while $D_{|D|-1} \neq |D| - 1$ **do**
  foreach $b \in D$ **do**
    if $b > lo$ **then**
      `/* Replace b with lo                                           */`
      $B_f \leftarrow$ block list containing b
      $D \leftarrow D + \{b\} - \{lo\}$
      $i \leftarrow$ index of b in B_f
      $B_f[i] = lo$
      $lo \leftarrow$ Smallest block number such that $lo \notin D$
    end
  end
end
```
/* Make each file continuous, ignoring sort                           */
```
$S \leftarrow clone(F)$
$lo \leftarrow 0$
while $|S|>0$ **do**
  $f \leftarrow NULL$
  foreach $g \in S$ **do**
    if $lo \in B_g$ **then**
      $f \leftarrow g$
      break
    end
  end
  $i \leftarrow lo$
  foreach $g \in S$ **do**
    if $i \geq |B_f| + lo$ **then**
      break
    end
    $a \leftarrow B_f[i]$
    foreach $b \in B_g$ **do**
      if $b \geq lo + |B_f|$ **then**
        $B_g \leftarrow B_g + \{a\} - \{b\}$
        $B_f \leftarrow B_f - \{a\} + \{b\}$
        $i \leftarrow i + 1$
        break
      end
    end
  end
  $lo \leftarrow lo + |B_f|$
end
```
/* Reorder the blocks allocated to each file                          */
```
foreach $f \in F$ **do**
  $B_f \leftarrow$ block list of f
  $sort(B_f)$
end
return D