Utku Noyan
60717
06.06.2020

Comp304 Report of Project-3

COMP 304 | Project 3
Space Allocation Methods

Introduction

Allocating space efficiently is a main concern when designing a file system or a memory management system. The criteria to assess certain design is space utilization and I/O performance. The objective of this assignment is to understand, implement, and empirically measure the performance of two space allocation methods used in file systems, namely contiguous and linked allocation, against various inputs.

Describing Implementation

I implemented all parts of the project. My program starts with main function and do followings:

Main()

- **Loop through io files:** Selects each io file in each loop
    - **Initialize parameters to evaluate:** Initializes the parameters for evaluation of performance
    - **Loop 5 times for averaging:** Runs the selected file as 5 times to get average performance.

    - **Print results to evaluate design:** After simulation finished for selected file, it prints performance evaluation parameters.

- **run(CA, LA, dir_idx, idx):** Opens selected file, and runs simulation for this file. Contiguous allocation and linked allocation objects are initialized at the same time and do operations at the same line. I keep the variables to measure the performance of operations of each Contiguous and Linked Allocation. After simulation ends, it returns the variables to measure performance.
    - Arguments are
        - CA: Contigous Allocation Class object
        - LA: Linked Allocation Class object
        - dir_idx: index of io file
        - idx: repeat number

Contiguous Allocation():

Contiguous allocation class implements create_file, access, extend, and shrink function with first-fit algorithm. I also implemented compaction and needed some helper function as conversion, shift, fill, fit_space, and is_extendable. The directory consists of integer numbers to keep the which blocs are occupied/empty.

- **initialize():** Given from input file block_size, initialize the Contiguous Allocation object
- **shift(file_id,size)**: It shifts file left to right. if destination space is available.
- **conversion(len): F**inds number of block required from length of bytes
- **is_extendable(idx, size):** It looks for whether a file is extendable or not at a given start index.
- **move(file_id, n_index):** It moves file to new index from start index if destination space is available.
- **fit_space(num_block):** Looks for available space for files having a given number of blocks. Applies first fit strategy.
- **fill(idx, le):** It inserts file in directory with putting positive random integer in directory entry.
- **empty(idx, le):** It makes files that occupied by directory files empty.
- **compaction(s_idx):** In order to do compaction/defragmentation; All files in directory shifted from high to low indexes. This results in empty spaces being gathered at the end of the directory. With the help of compaction, now new files can be added. Because we have limited space, I implemented this function as shifting blocks 1-1. Start index helps to do compaction for

extension because sometimes I move a file which is gonna extend to end. This causes empty space and I solve this by doing again compaction.

- **create file(file id, file length):** This function allocates a space for the file of size file length bytes on the disk, updates the DT and the FAT. The file blocks may be filled with random number greater than zero. A suitable hole needs to be found in a contiguous allocation case. If no whole is found, you need to do compaction of the directory contents. If there is not enough space to store the file, the operation will be rejected with a warning message.

- **access(file id, byte offset):** Returns the location of the byte having the given offset in the directory, where byte offset is the offset of that byte from the beginning of the file.

- **extend(file id, extension):** Extends the given file by the given amount, where extension is the number of blocks not bytes. For simplicity, the extension will always add a block after the last block of the file. If there is no sufficient space to extend the file, the operation will be rejected with a warning message. In contiguous allocation, if there is no contiguous space, you need to do compaction and may reallocate the file blocks. Remember that you have a buffer that can accommodate only a single block.

- **shrink(file id, shrinking):** Shrinks the file by the given number of blocks. The shrinking deallocates the last blocks of the file. Note that deallocation means just that these blocks are no longer referred to by that file and you can use them to store new data, and there is no need to move them or the files adjacent to them at the moment. You can indicate that the block is freed by storing zero in it, knowing that you store random positive values in the filled blocks.

Linked Allocation():

Linked allocation class implements create_file, access, extend, and shrink function. The directory consists of FAT objects to now next index for file's block.

- **initialize():** Given from input file block_size, initialize the Linked Allocation object.

- **conversion(len):** Finds number of block required from length of bytes

- **fit_space(n_block):** Looks for available block for file having given number of block.

- **create file(file id, file length):** This function allocates a space for the file of size file length bytes on the disk, updates the DT and the FAT. The file blocks may be filled with random number greater than zero. A suitable hole needs to be found in a contiguous allocation case. If no whole is found, you need to do compaction of the directory contents. If there is not enough space to store the file, the operation will be rejected with a warning message.

- **access(file id, byte offset):** Returns the location of the byte having the given offset in the directory, where byte offset is the offset of that byte from the beginning of the file.

- **extend(file id, extension):** Extends the given file by the given amount, where extension is the number of blocks not bytes. For simplicity, the extension will always add a block after the last block of the file. If there is no sufficient space to extend the file, the operation will be rejected with a warning message. In contiguous allocation, if there is no contiguous space, you need to do compaction and may reallocate the file blocks. Remember that you have a buffer that can accommodate only a single block.

- **shrink(file id, shrinking):** Shrinks the file by the given number of blocks. The shrinking deallocates the last blocks of the file. Note that deallocation means just that these blocks are no longer referred to by that file and you can use them to store new data, and there is no need to move them or the files adjacent to them at the moment. You can indicate that the block is freed by storing zero in it, knowing that you store random positive values in the filled blocks.

File():

File class keeps index for file, length of block and length of byte

- **Initialize():** Initializes variables with given values. I keep null file as -1 index, -1 block length, -1 byte length

Directory Table():

Directory table class keeps table as dictionary

- **Initialize:** Initialize empty dictionary

- **getFile(file_id):** Gets current file if it exists, o.w return FAIL

- **add_file(file_ID, file):** Adds file with corresponding file_ID. If file with file_ID exists, returns FAIL

- **existence(file_ID):** Shows file given ID exist or not

- **update_len_block(file_ID, n_len):** Updates length of block

- **update_len_byte(file_ID, n_len):** Updates length of byte

- **remove_file(file_ID):** Removes file from directory if it exists

File Allocation Table(FAT):

File allocation table class that keeps the next index for the file's block.

- **Initialize:** Initializes the next value of block as -1
- **empty():** Makes next -1
- **update(n):** Updates next index as n

Experimentation and Analysis

		Contigous Allocation	Linked Allocation
input_8_600_5_5_0.txt	Average Time For Creation(nanosecond)	**458882.3333**	**426911.3333**
	Average Time For Access(nanosecond)	**1794.501718**	**1837.113402**
	Average Time For Extend(nanosecond)	**1824.864865**	**1635.315315**
	Average Time For Shrink(nanosecond)	**0**	**0**
	Number Of Files That Their Creation Is Rejected	**72**	**323**
	Number Of Files That Their Extension Is Rejected	**6**	**106**
input_1024_200_5_9_9.txt	Average Time For Creation(nanosecond)	**474008**	**752812**
	Average Time For Access(nanosecond)	**2260**	**2044.705882**
	Average Time For Extend(nanosecond)	**1976.074499**	**1773.782235**
	Average Time For Shrink(nanosecond)	**2023.125794**	**1862.388818**
	Number Of Files That Their Creation Is Rejected	**67**	**70**
	Number Of Files That Their Extension Is Rejected	**145**	**147**

input_1024_200_9_0_0.txt	Average Time For Creation(nanosecond)	**388487**	**681484**
	Average Time For Access(nanosecond)	**1307.704717**	**1260.880979**
	Average Time For Extend(nanosecond)	**0**	**0**
	Average Time For Shrink(nanosecond)	**0**	**0**
	Number Of Files That Their Creation Is Rejected	**80**	**77**
	Number Of Files That Their Extension Is Rejected	**0**	**0**
input_1024_200_9_0_9.txt	Average Time For Creation(nanosecond)	**335322**	**567929**
	Average Time For Access(nanosecond)	**1275.514019**	**1256.323988**
	Average Time For Extend(nanosecond)	**0**	**0**
	Average Time For Shrink(nanosecond)	**1338.295318**	**1278.751501**
	Number Of Files That Their Creation Is Rejected	**71**	**71**
	Number Of Files That Their Extension Is Rejected	**0**	**0**
input_2048_600_5_5_0.txt	Average Time For Creation(nanosecond)	**445591.6667**	**774329.3333**
	Average Time For Access(nanosecond)	**1440.604027**	**1309.731544**
	Average Time For Extend(nanosecond)	**1542.056075**	**1314.392523**
	Average Time For Shrink(nanosecond)	**0**	**0**
	Number Of Files That Their Creation Is Rejected	**72**	**74**
	Number Of Files That Their Extension Is Rejected	**6**	**5**

- **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?(10 pt.)**

  - Contiguous Allocation has a shorter average operation time for create_file and access operation since both work O(1) time if there is no need for compaction. If we are creating and removing many files, this causes a need for compaction and operation time starts to take longer than expected.
  - If we compare files input_1024_200_5_9_9.txt,input_1024_200_9_0_0.txt, input_1024_200_9_0_9.txt, the total time passed for files as: input_1024_200_5_9_9.txt > input_1024_200_9_0_0.txt > input_1024_200_9_0_9.txt
  - The extend time passed for file input_1024_200_5_9_9.txt is very large because there are many compaction operations needed. Compaction requires many times because it moves block 1-1 due to insufficient memory.
  - The shrink time passed for file input_1024_200_9_0_9.txt is very large. The access time passed for file input_1024_200_9_0_0.txt is very large because there are many calls for access.
  - In linked allocation, each file is a linked list of disk blocks and keeps the next index with FAT. The disk blocks can be scattered anywhere on the disk. Linked Allocation has a shorter average operation time for extend and shrink in general. This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory. This method does not suffer from compaction. This makes it relatively better in terms of memory utilization.
  - If we compare files input_1024_200_5_9_9.txt,input_1024_200_9_0_0.txt, input_1024_200_9_0_9.txt, the total time passed for files as: input_1024_200_9_0_0.txt > input_1024_200_5_9_9.txt > input_1024_200_9_0_9.txt
  - The extend time passed for file input_1024_200_9_0_0.txt is larger than input_1024_200_5_9_9.txt.

- **Comparing the difference between the creation rejection ratios with block size 2048 and 32, what can you conclude? How did dealing with smaller block sizes affect the FAT memory utilization? (5 pt.)**

  - The rejection ratio for file creation for contiguous allocation **72, 67, 80, 71, 72** for files
  - Input_8_600_5_5_0.txt, input_1024_200_5_9_9.txt, input_1024_200_9_0_0.txt, input_1024_200_9_0_9.txt, input_2048_600_5_5_0.txt. The rejection ratio for file creation for contiguous allocation **6,6** for files Input_8_600_5_5_0.txt,input_2048_600_5_5_0.txt.
  - The reason why rejection ratio for file creation for contiguous allocation is similar, is the only change is the remaining empty byte number in blocks, so this does not cause much change.

- The rejection ratio for file creation for linked allocation **323, 70, 77, 71, 74** for files Input_8_600_5_5_0.txt, input_1024_200_5_9_9.txt, input_1024_200_9_0_0.txt, input_1024_200_9_0_9.txt, input_2048_600_5_5_0.txt. The rejection ratio for file creation for contiguous allocation **106,5** for files Input_8_600_5_5_0.txt,input_2048_600_5_5_0.txt. The reason why the rejection ratio for file extension for contiguous allocation is very different, is since the block size is very crucial, we allocated memory for FAT in memory, so the number of blocks that can be used for file allocation is reduced.

- **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. (5 pt.)**

  - The first makes it hard for the file to be split into blocks and can command an increase in allocated block size because of the addition of link entries, which can result in additional blocks with nearly free space. Furthermore, when joints are in a block, you are required to move backward and forward till you enter the specified block, while when FAT, you move in FAT till you find the position.

- **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?(3 pt.)**

  - We are compacting the blocks by moving them successively, looking for the following block, and recurring the identical runs, and we may move blocks as an entire rather than blocks by piece, which seems to diminish the running time of the consolidated process if we had additional data structure.

- **How much, at minimum, extra memory do you need to guarantee reduction in the number of rejected extensions in the case of contiguous allocations? (3 pt.)**

  - In this implementation due to limited memory the extra memory I need to guarantee reduction as the size of file plus minimum size for extension I got in rejection numbers in simulation. In general the number is the minimum size for extension I got in rejection numbers in simulation.