# COMP 304: Project 3
# Space Allocation Methods

Due: Sun, 31 May 2020, 11.59 pm

**Notes:** The project is must be done **individually**. You may discuss the problems with other students and post questions to the OS discussion forum but the submitted work must be your own work. This assignment is worth 20% of your total grade.

**Corresponding TA: Fareed Qararyah (email: fqararyah18@ku.edu.tr)**

## Description

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.
The following are the assumptions and limitations of the file system:

- The file system consists of a single directory.
- The directory has a fixed size of 32768 blocks.
- The block size is fixed for a single experiment, but might change from experiment to experiment as discussed later.
- The directory has a **Directory Table (DT)**. An entry in DT consists of: file identifier, the index of the starting block of the file and the file size.
- Linked allocation should be implemented using **File-Allocation Table** (FAT).
- There is no extra space. However, we assume that there is sufficient space to store the DT, and a fixed 32768 blocks to store our data. In the linked allocation, part of this fixed space will be used to keep the FAT.
- Other than DT and FAT, you are not supposed to define any additional data structures. We assume you have a fixed memory, e.g. a set of registers, sufficient to store fixed number of variables. In addition, you have a buffer of size equal to one block size, meaning that if you want to alter the location of a file, you can do it block by block.
- A file can occupy one or more blocks. A single block cannot be shared.
- In contiguous allocation, we have three alternatives for hole allocation. You are supposed to implement first fit strategy.

## Functionalities

Your design is expected to provide a capability of serving four main requests:

- `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 need to be found in contiguous allocation case. If no whole is found, you need to do compaction/defragmentation 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 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 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 more referred 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 block is freed by storing zero in it, knowing that you store random positive values in the filled blocks.

These requests are simplified abstractions of the manipulations in a file system. When a file is created or extended you may fill the newly allocated space with a random number greater than zero. When a block is deallocated you may fill it with zero. Since we are not requesting a write operation on a certain offset, you do not need to allocate **block_size** for each block, you can just allocate a single integer random value. The **block_size** will be needed just to do some calculation like the size of a file to be created, or an offset to be returned.
Note, the defragmentation is expected to be done in one direction. This direction is from high to low indices. This is to have consistent results. Under the given assumption and in certain scenarios, implementing defragmentation in a different way might affect the extension rejection ratio and we need to avoid this.

## Experimentation and Analysis

A main goal of this project is to be able to decide when to use which allocation strategy. You are provided with text files each of them contains a sequence of operations representing system with certain allocation characteristics. You need to evaluate your design against these inputs. For each input you are expected to report:

- The number of files that their creation is rejected.
- The number of files that their extension is rejected.
- The average running time of an operation. For this you should run with each input 5 times and take the average.

For a certain input file, the block size is indicated after the 'input_'. For example a file named input_1024_150_5_5_0.txt has a block size of 1024. 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.

After reporting the experiment results, you need to answer the following questions:

- 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.)
- 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.)
- 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.)
- 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.)
- 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.)

## Deliverables

You are required to submit the followings packed in a zip file (named your-username(s).zip) to blackboard :

- You can implement your program using any of programming languages of C, C++, Java or Python. Upload your source. Upload your .c, .cpp, .java or .py files. Please comment your implementation, it helps getting higher marks if the results are incorrect.
- The relative performance of your code matters in addition to the correctness of the results. If you implement one of the strategies poorly, it might produce slower average operation time in all cases.
- A report briefly describing your implementation, particularly which parts of your code work, which parts do not work. Report should address the questions in the experiments and analysis.
- Grading: Contiguous Allocation (37 pts= 7.5 pts for each functionality + 7 pts performance), Linked Allocation (37 pts= 7.5 pts for each functionality + 7 pts performance), Report and Answering Analysis Questions (26 pts)

GOOD LUCK.