SEQUENTIAL FIT ALGORITHMS EVALUATION USING OBJECT-ORIENTED PROGRAMMING

Milica Tufegdžić[1], PhD; Aleksandar Mišković[2], PhD
Academy of professional studies Šumadija, Trstenik, mtufegdzic@asss.edu.rs
Academy of professional studies Šumadija, Kragujevac, amiskovic@asss.edu.rs

Abstract: *Modern applications require sophisticated strategies for memory management which allow efficient allocation of the demanded memory blocks within minimal response time, providing minimum fragmentation. Memory allocation shemas use different allocator algorithms, such as sequential fit, buddy system and segregated fit. The sequential fit algorithm which will be presented and evaluated in the terms of internal fragmentation, uses single linear list of all free blocks and different ways to find a proper unused memory hole of the most appropriate size. Memory chunks' values and different sets of required processes' memory are used as input data in program written in C++ programming language, with the aim to calculate internal fragmentation. The program is written according to basic principles for memory allocation in the sequential fit algorithm. The results for obtained internal fragmentation in the cases of using the first-fit, best-fit and worst-fit allocators are analyzed and compared.*

Keywords: *memory allocators, internal fragmentation, first-fit, best-fit, worst-fit*

1. INTRODUCTION

Efficient management of available system resources, such as Central Processing Unit (CPU), memory and Input/Output devices among competing applications is one of the most crucial operating system tasks. Special attention should be paid to memory utilization and its allocation while executing sophisticated real-world applications, due to the fact that memory is often the most critical resource in terms of speed and capacity [1]. The popularity of object-oriented languages, such as Java and C++, have led to the need for more efficient memory management [2].

Usually, there are two partitions in main memory, one for operating system, so called low memory, and the other named high memory, which holds the user's processes. As there are many processes at the same time in the input queue waiting to be brought into memory, there is a need to allocate available memory and satisfy as many of the requests for memory as possible [3].

There is a variety of memory allocators' design and different models of memory management schemes. Depending on whether a memory partition for the active process is of fixed or variable size during the existence of the process, there are fixed and variable allocation schemas [4]. Determining the number of partitions and their size presents the central problem in fixed allocation schema. Another issue is the problem of internal or external fragmentation. Variable partition schema minimize fragmentation due to dynamic variation of partition size [5].

Memory management is responsible for dynamic allocation of memory parts from a large block of memory (so called heap), depending on the users' requests. There is a possibility for dividing large memory blocks into smaller ones (chunks). At some point in the time some chunks are allocated to a process (live memory), and some of them are being freed and thus become available for future allocations. However, some of the freed chunks may not be available for future allocations and represent garbage. In any case, memory management must keep track about live memory, freed and garbage chunks [2,6].

Different dynamic memory management techniques, strategies, mechanisms, policies and algorithms are used with the aim to allow applications to access memory blocks quickly and without wasting too much space which in turn results in smaller fragmentation [6,7,8,9]. One of the simplest methods implies using one or multiple free lists to keep track of free blocks. Multiple lists result in better performance [8]. Policies as a conclusion methods for dynamiclly allocating blocks of memory precisely decide where to remove or put an allocated block. Mechanisms such as sequential fit, segregated free list (segregated fit), buddy systems, indexed fit and bitmapped fit are examples of policy implementations. They represent a group of different algorithms and are classified according to the way they find a free block of the most appropriate size [9,10,11,12].

Sequential Fit algorithms use searching through the list of available holes in memory. The most common are: first-fit, best-fit and worst-fit. Unused memory holes, i.e. the unused portions of the memory, are found in different ways,

according to the fitting policy, with the aim to satisfy the memory allocation request [6,7]. In order to reduce fragmentation, some modifications of sequential fit algorithms were proposed, such as lazy-fit algorithm. This algorithm uses pointer increments as primary allocation method and normal fits as backup allocation method [7]. Improving internal memory fragmentation can be done by finding the optimal configuration between set of potential solutions of a segregated free data structure. Optimal configuration is chosen using genetic algorithms [13]. Implementation of reaps as a combination of regions and heaps provides high performance. Reaps represent generalization of general-purpose and region-based allocators [14]. In real-time applications, which requires a different memory management approach, adaptive memory management schema was proposed. This algorithm uses a combination of modified two-level segregated fit methodology and the ability to move memory blocks. It has proved to be a good solution for embedded systems where the available memory is limited [11]. Maintaining free chunks of memory on a binary tree improves the search for the appropriate memory block. This better fit allocation policy has been implemented using nodes in binary tree for keeping track of the size of the largest chunk, which is available in the left and right sub-trees [2]. Separate processor for memory management functions embedded in DRAM is used for transferring the allocation and de-allocation functions' execution from the main CPU. This technique is known as Intelligent Memory Manager [1].
In order to evaluate first-fit, best-fit and worst-fit algorithm, in terms of fragmentation, a program in object-oriented language C++ was written. The program is tested using predefined memory chunks' values and different values for processes as input data. After compiling and executing the code in Visual Studio 2019 Community Edition, Version 16.10.4, unused space is calculated and the results are processed and presented in the form of a diagram. Internal fragmentation is calculated as the ratio between free fragmented space left and total free memory space and expressed in percentages, with the aim to compare obtained results.

2. SEQUENTIAL FIT ALGORITHMS

The main task of memory allocator algorithm is to provide real time support for memory allocation [15]. Allocation method determines the size of fragmentation and well-designed allocators must efficiently deal with memory fragmentation problems, satisfying four design criteria, such as efficiency of representation, speed of allocation, speed of "recycling" and utilization of memory [11,16,17,18]. It is hard to choose an appropriate allocation method, because in most cases it depends on the application [16]. Each method has its own advantages and disadvantages [15].
The simplest implementation maintains a single linked list of free memory chunks (blocks), named free list, in the allocator. This free list is doubly and/or circularly linked. When a request for allocation memory is made, the free list is searched with the aim to find an appropriate suitable block. Fit policy is responsible for the way that appropriate block is found [7,18,19].
For the purpose of this study, sequential fit algorithms were selected because they are the most common and simple to implement. Depending on the way in which free blocks from the list are allocated, we can distinguish first-fit, best-fit, and worst-fit algorithm [1,2,6,7,8,9,10,12,15,18]. Next-fit, as a common optimization of first-fit has proven to cause more fragmentation than best-fit and therefore will be excluded from this study [1,10,19].
First-fit allocator tries to find the first free block that is large enough to to accommodate the incoming process. The free list is searched sequentially, usually from the beginning [1,6,7,10,18,19]. The list of free blocks is maintained in various ways, such as First-In First-Out, Last-In First-Out or address-order mechanisms [7,9,10,12]. In best-fit implementation, searching the free list is done exhaustively, with the aim to allocate the smallest hole that is large enough to satisfy a request [1,6,7,10,18,19]. In worst-fit policy, allocator searches for the largest hole regardless of the hole position [6,16]. In that case, after finding the hole that fits the worst and fulfilling the application request, small fragmentation is avoided, because newly created unallocated block as large as it can be [9,16]. All presented sequential fit algorithms use Knuth's boundary tag technique for coalescing of free blocks [10,19].

3. ALGORITHMS IMPLEMENTATION IN C++

Taking into account different ways to allocate free blocks, a program in C++ object-oriented programming language is written, according to algorithms presented above. C++ was chosen because it is convenient for manipulating raw memory and pointers. Four header files with proper classes' definitions are created, for block, memory, memory manager and process. For example, class Block contains private attributes such as size and bool variable which indicates whether the block is occupied or not. The protected section contains constructors, destructors and functions for setting and resetting private variables' status. In the private section of class Memory, variable size and number of blocks are defined, as well as a dynamic array of block pointers. Friend class MemoryManager enables direct access to the array of blocks. Class Process has private attributes size and variable b that represents memory block index in which the process could potentially be allocated. Special member function constructor, as well as setter and getter block functions are defined in the public section. Class MemoryManager has the following data members: pointer to memory object, array of pointers to processes and number of processes. Public section contains the constructor and destructor function, function AddProcesses for forwarding an array of processes and a function ResetAssignment for releasing blocks from processes. Functions FirstFit, BestFit and WorstFit are also created, while function PrintProcesses print assigned processes. A part of the code in which class Process is defined is presented in Figure 1.

```
Operativni                                    MemoryManager
 1    #pragma once
 2    #include "Memory.h"
 3    #include "Process.h"
 4
 5    class MemoryManager {
 6    private:
 7        Memory* mem;
 8        Process** processes;
 9        int noOfProcesses;
10
11    public:
12        MemoryManager();
13        MemoryManager(int nop);
14        MemoryManager(int nop, int mSize, int nob, int* b);
15        ~MemoryManager();
16
17        void AddProcesses(int* p, int n); // throwable
18
19        void FirstFit();
20        void BestFit();
21        void WorstFit();
```

Figure 1: Part of the code in class MemoryManage

Files Process.cpp, Memory.cpp and MemoryManager.cpp implement functions from the appropriate header files. File main.cpp represents main program in which the implemented algorithms for allocating blocks are called for assigned values of block and process sizes. These values in the form of input data are entered using keyboard.
The part of the program main.cpp, which implements the input of number and capacity of memory blocks, is presented in Figure 2.

```
Operativni                                    (Global Scope)
 7    char answ = 'n';
 8
 9    do {
10        numOfIterations++;
11        std::cout << "\n\n################################################################################";
12        std::cout << "\nTEST " << numOfIterations << ":";
13        std::cout << "\n\nSpecify the number of memory blocks: ";
14        int n = 0;
15        std::cin >> n;
16        int* memBlocks = new int[n];
17        int memSize = 0;
18        int temp;
19        std::cout << "Specify the capacity of memory blocks:\n";
20        for (int i = 0; i < n; i++) {
21            std::cout << "Block " << i + 1 << ": ";
22            std::cin >> temp;
23            if (temp > 0) {
24                memBlocks[i] = temp;
25                memSize += temp;
26            }
27            else {
28                std::cout << "Block size must be a positive integer!" << std::endl;
29                i--;
30            }
```

Figure 2: The part of the file main.cpp

4. DISCUSSION AND RESULTS

For the purpose of this study, five free blocks (holes) are chosen with the following capacity: 100 KB, 500 KB, 200 KB, 300 KB and 600 KB, and four processes with randomly chosen sizes. The program is executed in Visual Studio 2019 Community Edition, Version 16.10.4, for ten sets of values presented in Table 1, as Cases 1 - 10. Blocks are marked as B1, B2, B3, B4 and B5 respectively, while processes are marked as P1, P2, P3 and P4 in accordance to their order in queue.

Table 1: Process' sizes in KB as input data

Process	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10
P1	212	357	115	98	305	550	220	50	528	372
P2	417	210	500	392	567	100	450	150	428	201
P3	112	468	358	254	319	250	558	350	340	568
P4	426	491	200	532	259	400	333	600	348	324

Results for tested data in Case 6 are presented in Figure 3. After entering the number of memory blocks and their capacity, number of processes and their sizes, initial status of the processes is checked. After that, processes' statuses are presented, as well as the blocks to which they were assigned, for first-fit, best-fit and worst-fit algorithms. Unassigned processes are also presented, and free fragmented space is calculated.

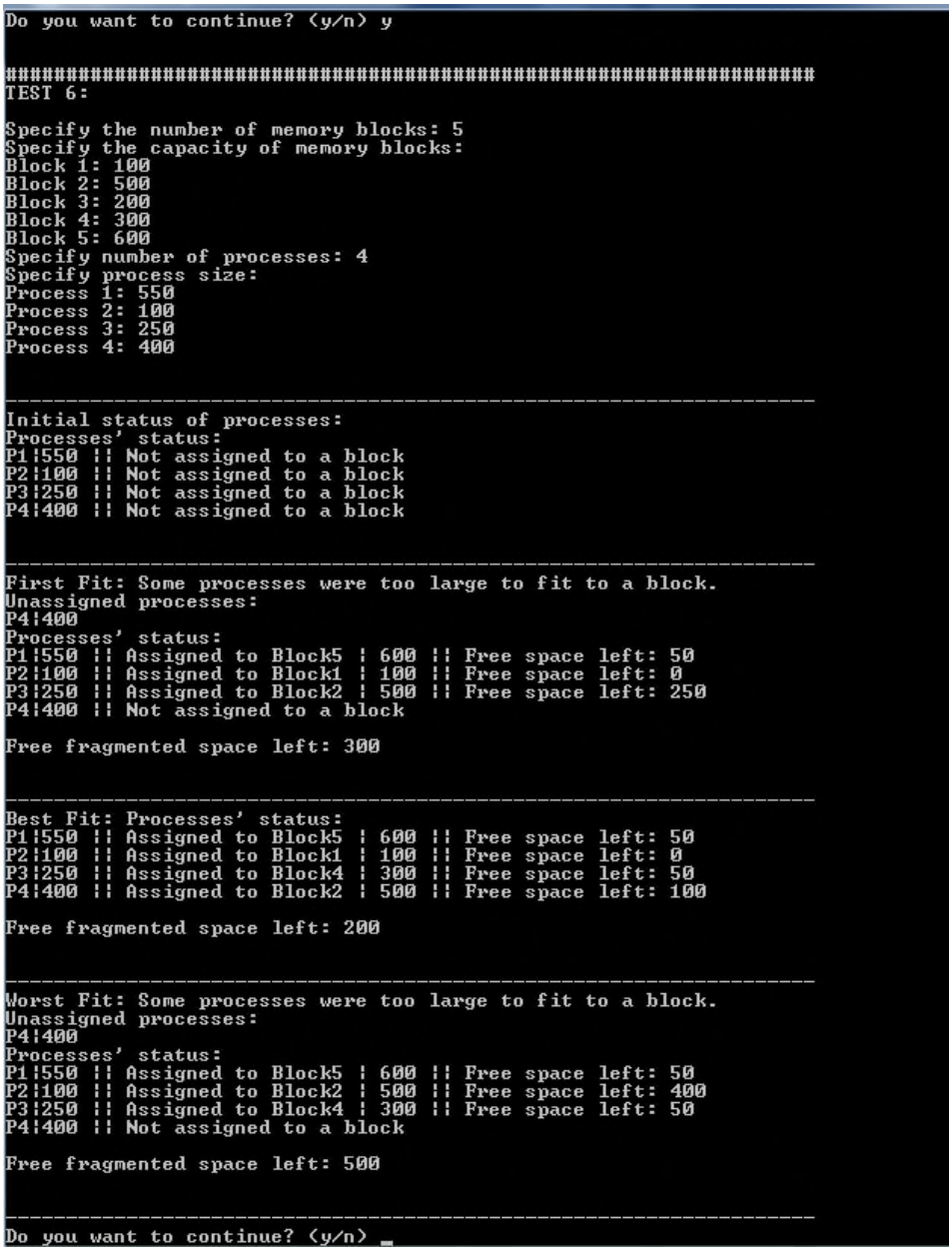

Figure 3: Results of code execution for the input data in Case 6

The results obtained for free fragmented space represent the values of internal fragmentation, while the memory capacity of unassigned blocks represents external fragmentation. For easier analysis and comparison of obtained results, the values for internal and external fragmentation in percentages, marked as IF and EF respectively, are calculated using equations (1) and (2):

$$IF = \frac{internal\ fragmentation}{total\ free\ memory} \cdot 100\ (\%), \quad \textbf{(1)}$$

$$EF = \frac{external\ fragmentation}{total\ free\ memory} \cdot 100\ (\%). \quad \textbf{(2)}$$

Calculated values, together with the absolute values for internal and external fragmentation in KB are presented in Table 2. For clarity and simplicity of the results of code execution in all cases, processes and blocks to which they are assigned are also presented, as well as unassigned processes for all algorithms.

Table 2: Internal and external fragmentation, assigned and unassigned blocks and processes

Case no.	Algorithm	Assigned process → block	Unassigned process/es	Internal fragmentation		Unassigned block/s	External fragmentation	
				KB	%		KB	%
1.	First-fit	P1→B2, P2→ B5, P3→B3	P4	559	32.88	B1, B4	400	23.53
	Best-fit	P1→B4, P2→B2, P3→B3, P4→B5	-	433	25.47	B1	100	5.88
	Worst-fit	P1→B5, P2→B2, P3→B4	P4	659	38.76	B1, B3	300	17.65
2.	First-fit	P1→B2, P2→B4, P3→B5	P4	365	21.47	B1, B3	300	17.65
	Best-fit	P1→B2, P2→B4, P3→B5	P4	365	21.47	B1, B3	300	17.65
	Worst-fit	P1→B5, P2→B2	P3, P4	533	31.35	B1, B3, B4	600	35.29
3.	First-fit	P1→B2, P2→B5, P4→B3	P3	485	28.53	B1, B4	400	23.53
	Best-fit	P1→B3, P2→B2, P3→B5, P4→B4	-	427	25.12	B1	100	5.88
	Worst-fit	P1→B5, P2→B2, P4→B4	P3	585	34.41	B1, B3	300	17.65
4.	First-fit	P1→B1, P2→B2, P3→B4, P4→B5	-	224	13.18	B3	200	11.76
	Best-fit	P1→B1, P2→B2, P3→B4, P4→B5	-	224	13.18	B3	200	11.76
	Worst-fit	P1→B5, P2→B2, P3→B4	P4	656	38.59	B1, B3	300	17.65
5.	First-fit	P1→B2, P2→B5, P4→B4	P3	269	15.82	B1, B3	300	17.65
	Best-fit	P1→B2, P2→B5, P4→B4	P3	269	15.82	B1, B3	300	17.65
	Worst-fit	P1→B5, P3→B2, P4→B4	P2	517	30.41	B1, B3	300	17.65
6.	First-fit	P1→B5, P2→B1, P3→B2	P4	300	17.65	B3, B4	500	29.41
	Best-fit	P1→B5, P2→B1, P3→B4, P4→B2	-	200	11.76	B3	200	11.76
	Worst-fit	P1→B5, P2→B2, P3→B4	P4	500	29.41	B1, B3	300	17.65
7.	First-fit	P1→B2, P2→B5	P3, P4	430	25.29	B1, B3, B4	600	35.29
	Best-fit	P1→B4, P2→B2, P3→B5	P4	172	10.12	B1, B3	300	17.65
	Worst-fit	P1→B5, P2→B2	P3, P4	430	25.29	B1, B3, B4	600	35.29
8.	First-fit	P1→B1, P2→B2, P3→P5	P4	650	38.24	B3, B4	500	29.41
	Best-fit	P1→B1, P2→B3, P3→B2, P4→B5	-	250	14.71	B4	300	17.65
	Worst-fit	P1→B5, P2→B2	P3, P4	900	52.94	B1, B3, B4	600	35.29
9.	First-fit	P1→B5, P2→B2	P3, P4	144	8.47	B1, B3, B4	600	35.29
	Best-fit	P1→B5, P2→B2	P3, P4	144	8.47	B1, B3, B4	600	35.29
	Worst-fit	P1→B5, P2→B2	P3, P4	144	8.47	B1, B3, B4	600	35.29
10.	First-fit	P1→B2, P2→B4, P3→B5	P4	259	15.24	B1, B3	300	17.65
	Best-fit	P1→B2, P2→B4, P3→B5	P4	259	15.24	B1, B3	300	17.65
	Worst-fit	P1→B5, P2→B2	P3, P4	527	31.00	B1, B3, B4	600	35.29

With the aim to compare and conduct evaluation of presented algorithms, the results of the study are shown in Figures 4 and 5, in the form of graphics.

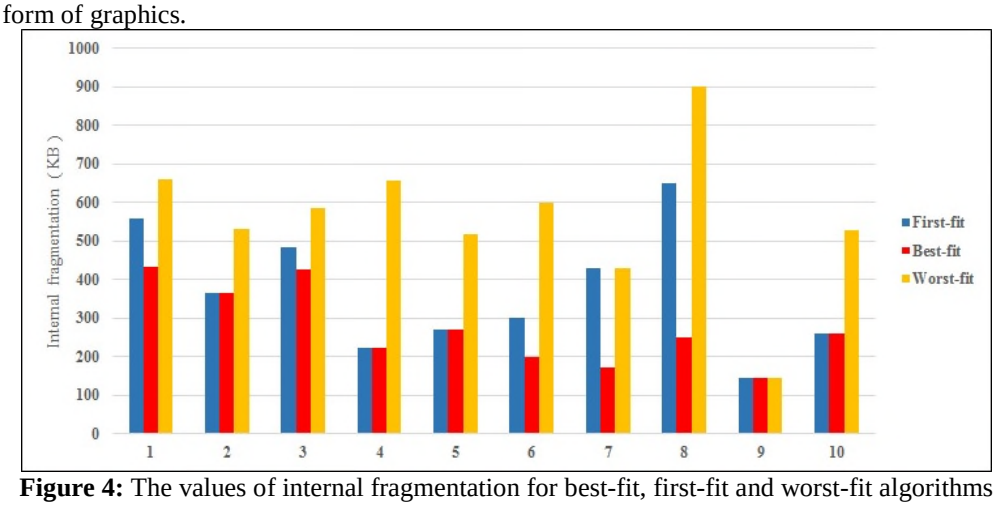

Figure 4: The values of internal fragmentation for best-fit, first-fit and worst-fit algorithms

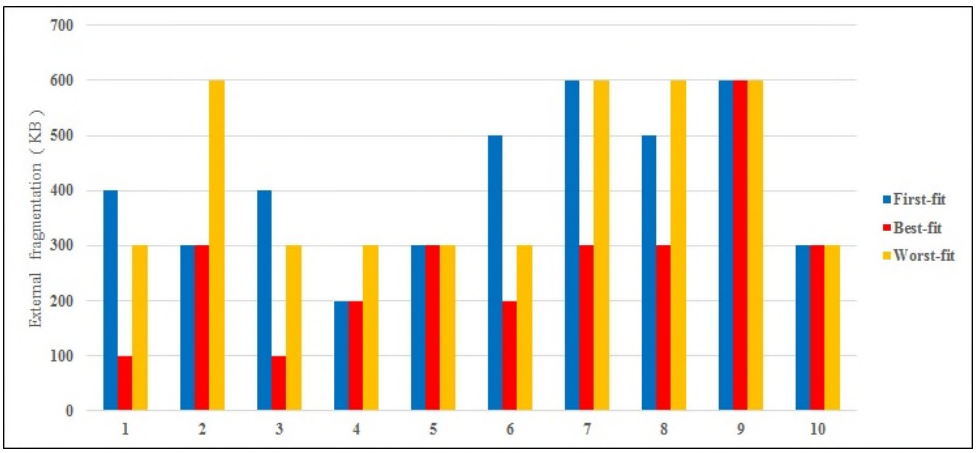

Figure 5: The values of external fragmentation for best-fit, first-fit, and worst-fit algorithms

In 50% of the presented cases, best-fit algorithm allows all incoming processes to be assigned to appropriate blocks. In such cases internal and external fragmentation have the smallest values, or are at least equal to the value in the case of first-fit algorithm. In terms of internal and external fragmentation, worst-fit algorithm is the least efficient, compared to first-fit and best-fit. Generally, best-fit algorithm has proven to be the best in terms using available memory space, with the average value of internal fragmentation 16.14%, while the average value of external fragmentation for first-fit and worst-fit are 21.68% and 31.65%, respectively. The average value for external fragmentation is the lowest in the case of best-fit algorithm (15.88%), compared to the values for first-fit (24.12%) and worst-fit algorithm (24.71%). In some cases, such as case 9, the values for internal and external fragmentation are the same for all tree algorithms. This fact can be related to the process' sizes and their order of arrival, taking into account that the memory partitions are the same size as in the other cases.

5. CONCLUSION

Efficient memory management is a very important issue, especially in the cases of real-world applications, where memory blocks are constantly being allocated and deallocated many times during their execution. Another additional issue is that the time spent for multiple allocation and deallocation actions affects system performances. It can also leads to out-of-memory conditions, due to internal and external fragmentation.

In this study, different issues of first-fit, best-fit and worst-fit algorithms, as examples of sequential fit algorithms are compared in terms of assigned processes and blocks, as well as internal and external fragmentation. The results show that the best-fit algorithm provides the most efficient use of memory in the sense of assigned process and values of internal and external fragmentation, compared to first-fit and worst-fit algorithms. Due to the fact that there is always a part of unused free memory blocks, future studies will include combinations of sequential fit algorithms and segregated fit. Some other methods for efficient memory management such as compaction, paging and segmentation should be considered.

REFERENCES

[1] Rezaei M., Kavi MK. Intelligent memory manager: Reducing cache pollution due to memory management functions. Journal of Systems Architecture, Volume 52, Issue 1; 2006: 41–55. doi:10.1016/j.sysarc.2005.02.004

[2] Rezaei M., Kavi MK. A New Implementation Technique for Memory Management. Nashville, USA: Proceedings of the IEEE SoutheastCon 2000 Preparing for The New Millennium; 2000: 332-339. doi: 10.1109/SECON.2000.845587.

[3] Silberschatz A., Galvin PB., Gagne G. Operating system concepts. 9th edition, Wiley; 2012.

[4] Faraz A. A review of memory allocation and management in computer systems. Computer Science & Engineering: An International Journal (CSEIJ), 6(4), 2016: 1-19. doi:10.5121/cseij.2016.6401

[5] Operating Systems - Memory Management Fixed Partitioning, Variable Partitioning. [accessed 23. July 2021.]; available at https://examradar.com/memory-management-2/

[6] Kabari LG., Gogo TS. Efficiency of Memory Allocation Algorithms Using Mathematical Model. International Journal of Emerging Engineering Research and Technology, Volume 3, Issue 9; 2015: 55-67.

[7] Chung YC., Moon S. Memory Allocation with Lazy Fits. ISMM '00: Proceedings of the 2nd international symposium on Memory management, October 2000: 65–70. doi:0.1145/362422.362457

[8] Diwase D., et al. Survey Report on Memory Allocation Strategies for Real Time Operating System in Context with Embedded Devices. International Journal of Engineering Research and Applications (IJERA), Volume 2, Issue 3, 2012: 1151-1156.

[9] Hasmukhbhai SV. Memory Management in Real-Time Operating System, PhD thesis, Vadodara-390002 (India), Department of computer science & engineering, Faculty of technology & engineering, The Maharaja Sayajirao University of Varoda, 2018.
[10] Johnstone MS., Wilson PR. The Memory Fragmentation Problem: Solved?. ACM SIGPLAN Notices, Volume 34, Issue 3, 1999: 26–36. doi:10.1145/301589.286864
[11] Deligiannis I., Kornaros G. Adaptive Memory Management Scheme for MMU-Less Embedded Systems. 11th IEEE Symposium on Industrial Embedded Systems (SIES), 2016: 1-8. doi: 10.1109/SIES.2016.7509439.
[12] Özer C. A dynamic memory manager for FPGA applications, Master thesis, Ankara, Turkey, The graduate school of natural and applied sciences of middle east technical university, 2014.
[13] Rosso CD. Reducing Internal Fragmentation in Segregated Free Lists using Genetic Algorithms. WISER '06: Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research, May, 2006: 57–60. doi:10.1145/1137661.1137674
[14] Berger ED., Zorn BG., McKinley KS. Reconsidering Custom Memory Allocation. OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2002: 1–12. doi:10.1145/582419.582421
[15] Awais MA. Memory Management: Challenges and Techniques for Traditional Memory Allocation Algorithms in Relation with Today's Real Time Needs. International journal of multidisciplinary sciences and engineering, 7(3), 2016: 13-19.
[16] Lindblad J. Handling memory fragmentation. [accessed 04. July 2021.]; available at https://www.edn.com/handling-memory-fragmentation
[17] Dynamic memory management. [accessed 25. July 2021.]; available at http://www.bradrodriguez.com/papers/ms/pat4th-c.html
[18] Heikkilä V. A Study on Dynamic Memory Allocation Mechanisms for Small Block Sizes in Real-Time Embedded Systems, Master thesis, Finland, University of Oulu, Faculty of Science, Department of Information Processing Science, Information Processing Science, 2013.
[19] Wilson PR., et al. Dynamic Storage Allocation: A Survey and Critical Review. In: Baler H.G. (eds) Memory Management. IWMM 1995. Lecture Notes in Computer Science, vol 986. Springer, Berlin, Heidelberg; 1995. doi:10.1007/3-540-60368-9_19