Journal of Theoretical and Applied Information Technology
29th February 2016. Vol.84. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

DYNAMIC MEMORY ALLOCATION USING FREE BLOCKS

[1]ALEKSANDR BORISOVICH VAVRENYUK, [2]IGOR VLADIMIROVICH KARLINSKY, [3]ARKADY PAVLOVICH KLARIN, [4]VIKTOR VALENTINOVICH MAKAROV, [5]VIKTOR ALEKSANDROVICH SURIGIN

[1]National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe Highway, 31, Moscow, 115409, Russia
[2]Karlsruhe Institute of Technology (KIT)
Kaiserstraße, 12, Karlsruhe, 76131 Germany
[3]National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe Highway, 31, Moscow, 115409, Russia
[4]National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe Highway, 31, Moscow, 115409, Russia
[5]National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe Highway, 31, Moscow, 115409, Russia

ABSTRACT

Algorithms for the dynamic allocation of RAM (Random Access Memory) to the operating system when multiprogramming have a significant impact on the efficiency of the operating system as a whole. Memory Manager (allocator) of GNU C Library UNIX standard library, which claims universality, is ineffective in some cases. This article describes the allocator algorithm with a list of clear areas, proposed by the authors, which allows achieving a higher efficiency of the RAM usage. The test methodology is proposed for the developed allocators, and the results of the comparison of the proposed allocator with the allocator of the GNU C Library UNIX standard library are provided.

Keywords: *Allocator, Memory Manager, The Process Of Memory Allocation, Memory Fragmentation, Operating Systems*

1. INTRODUCTION

In any operating system, the issue of the rational use of RAM is one of the most important. But, as practice shows, designing the universal algorithm of the RAM management is fundamentally impossible. The desire to fully use the available storage capacity will inevitably lead to additional costs of CPU time in the performance of the respective memory manager ("allocator") and, on the contrary, faster algorithms require additional memory consumption for storing their own data structures.

Many works by Russian and foreign authors [17], [1], [7], [2], [5] are devoted to the study of memory allocation algorithms in operating systems. The result of the collective efforts made by many authors was the standard allocator glibc of the GNU C Library [2]. This allocator uses many modern ideas of memory allocation and deallocation, such as, for instance, "paired tags algorithm", "doubles system", the use of "bitmaps" [6].

A fight with memory fragmentation is one of the main issues for any allocator [9]. CPU time expenditures for servicing the allocator data structures are another important issue [3].

There are also publications on the memory allocation for real-time systems [10], [11], which require, above all, to reduce the CPU time expenditures for the memory allocation and deallocation.

Some authors [15] suggest an automatic optimization technique for memory managers, but this technique has not received widespread attention.

However, in practice, in the modern operating systems, it is sometimes necessary to quickly change allocators adjusting to the current needs of

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

the computational process, depending on the characteristics of the issues being solved in the system.

This paper presents the research results of one of the possible allocation algorithms proposed by the authors, and the RAM deallocation using list of the clear areas.

2. ALLOCATOR DEVELOPMENT WITH THE FREE BLOCKS

One of the ways to implement a memory manager (allocator) is to use a list of the clear areas. The main problem is the correct organization of the list items. The list item shall keep, at a minimum, a size of the item and a pointer to the next list item. If the list is doubly linked, a pointer is also stored at the previous list item. If the list is implemented separately from the blocks (information about the list item is not stored in the free block), then in each list element the pointer to the free block shall be stored. For sufficiently small size of the allocated memory blocks, the size of the service list item information may be quite large relative to the size of the allocated block. Also, there is the list storage issue in the list organization separately from the blocks. It is necessary to monitor not only the allocation and deallocation of blocks to application program, but also the correct placement of the list items.

Based on these considerations, the list items shall be placed directly in the free blocks, and each item shall contain minimum information for the block size, and the pointer to the next item. But this raises the issue of the detection of related areas, as for the determination of the related areas, it is necessary to cycle through all the list items by comparing the boundary addresses of each area. If the list is sorted by the address ascending, the operation of determining the related areas becomes a little bit easier. If the related block is not found and the address of the current block is longer than the address of the free one when iterating the items from the beginning of the list, the list view can be completed. But any ordered list requires the insertion of a new item in a certain place, which makes is necessary to view the entire list in part from the beginning. In a non-ordered list, the insert of the new item can be performed in one step; the new item can be placed at the beginning or end of the list.

For quicker determination of the related clear areas, "paired tags" described in [3] are used. The size of each block will be stored at its beginning and end, and it will be negative for free blocks, and positive for occupied blocks. Having defined the information stored in the list, the minimum size of the allocated block can be calculated; it is equal to the size necessary to store the pointer to the next block (Figure 1).

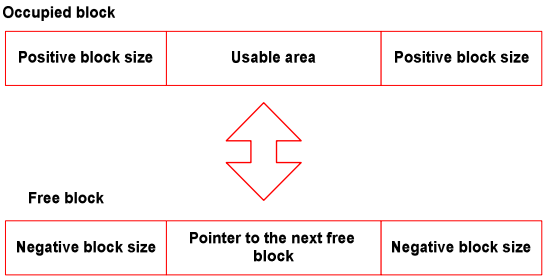

Figure 1. Block State Change

Figure 1 shows that each block has two tags at the edges; in case of the block status change, the sign in the tags also changes. If the block is deallocated, the pointer is still entered to the next free block in the list. Figure 2 shows the overall picture of the location of the free and occupied memory blocks.

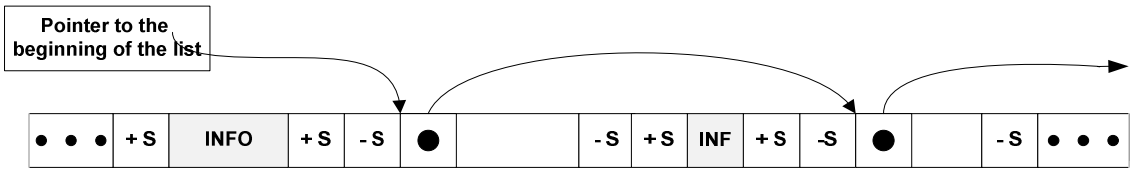

Figure 2. Examples Of The Block Layouts In Memory

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The memory allocation to the application program begins with the free block search in the list of the clear areas. If the list is empty, or there is an appropriate block on it, the allocator asks the additional memory to accommodate the block from the RAM. The appropriate block search in the free blocks list is made by one of the methods: first-fit, best-fit or worst-fit. If the list of the clear areas contains a free block, there may be two options for further action.

- If the free block size is equal to the requested size, or if during the block division into two parts the size of the remaining part does not allow to arrange another free block, the free block is completely given to the application program, despite the possibility of internal fragmentation. The tags of the free block are inverted, and the free block is removed from the list.
- If the free block can be divided in half so that the remaining part size is sufficient for the organization of the free block, the one half is back to the application program, and the other half forms a new free block and placed in the list.

A general chart of the memory block allocation to the application program is shown in Figure 3.

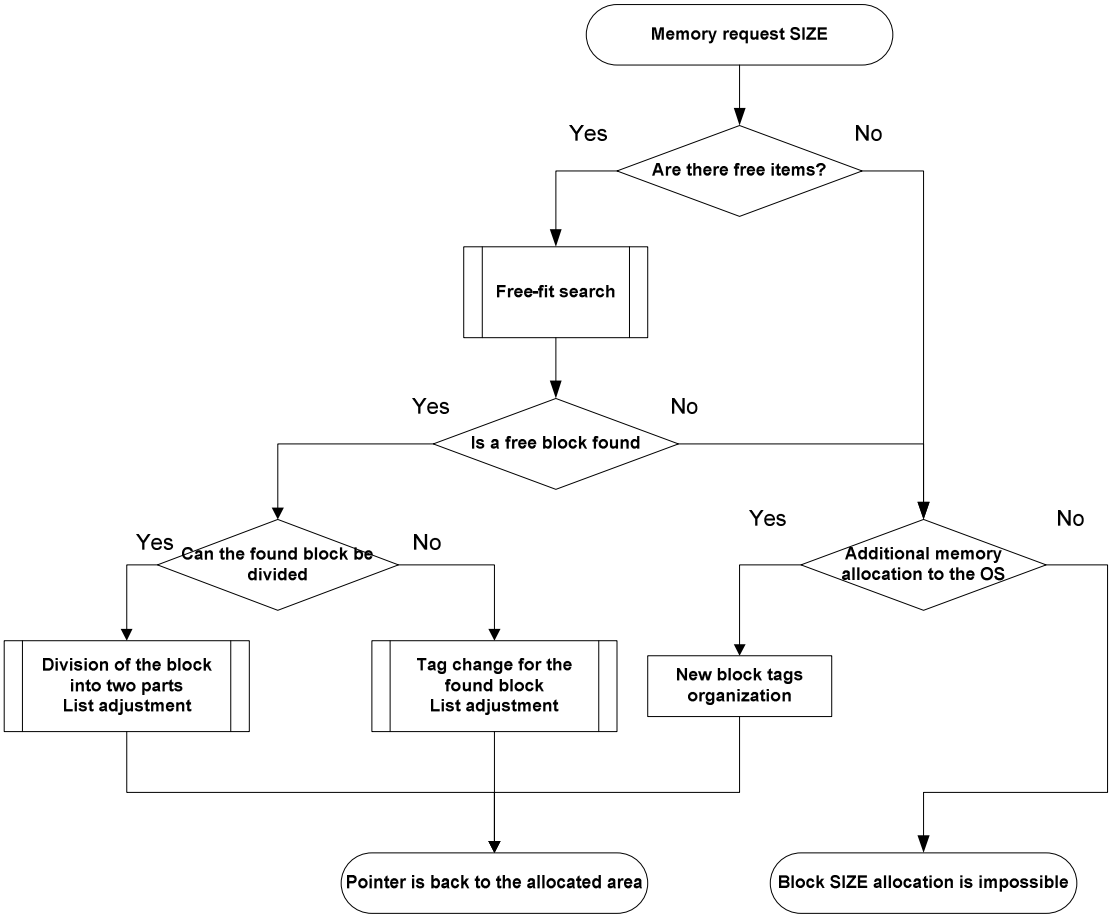

Figure 3. Memory Allocation To The Application Program Algorithm Chart

Journal of Theoretical and Applied Information Technology
29th February 2016. Vol.84. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The deallocation procedure of the occupied block begins with the definition of the related area status using the paired tags. There are four options for the placement of the free and occupied blocks next to the deallocated block (Figure 4).

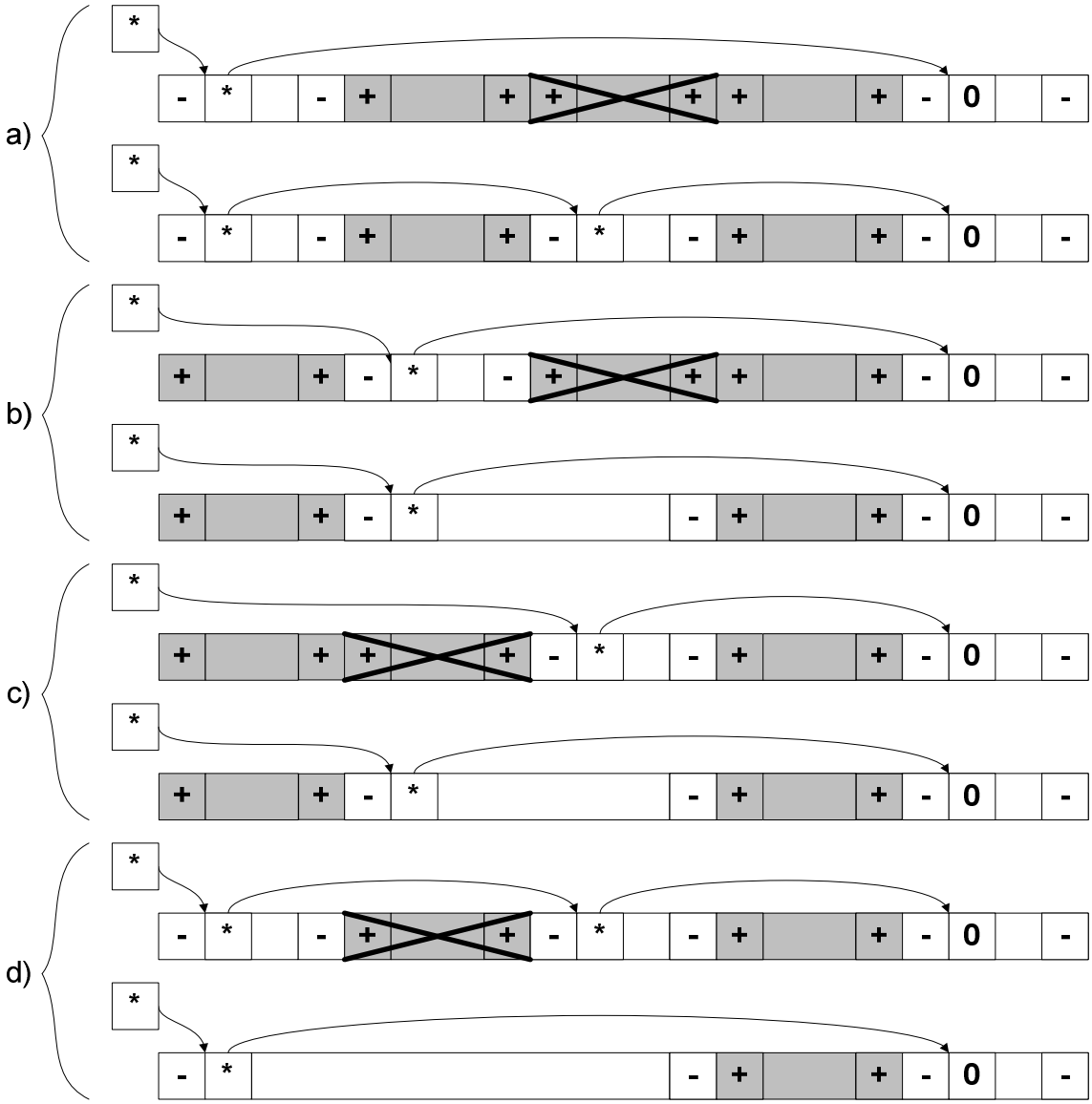

Figure 4. Occupied Block Allocation Available Chaining Options

a) Left and right blocks are missing or occupied. If the deallocated block is the first in the memory allocator, the left block is missing, and therefore there is no paired tag. To eliminate the error when checking the tag on the left, it is necessary to check the equality of the addresses of the allocator memory beginning and deallocated block. But when there are a large number of blocks in the allocator memory, the possibility that the deallocated block is the first is rather small, and the address verification in most cases is a superfluous action. To avoid unnecessary testing operations in the blocks deallocation, the pointer to the list of the clear areas is stored at the beginning of the allocator memory. The pointer may receive a null or positive value, and when checking it as a tag, it will correspond to the occupied block (as the negative value of the tag corresponds to the free blocks). This approach solves the issue of the superfluous address validation and determines the exact position of the pointer on the list of the clear areas. A similar approach is used to determine the missing of the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

right block, but the null tag is stored instead of the pointer at the end of the allocator memory area.

b) After the block deallocation with such "neighbors", the deallocated block is entirely marked as free and added to the list of the clear areas as an independent item.

c) The left block is free, and the right one is missing or occupied. In this situation, there is an association of the deallocated block and the left block. The left tag of the left free block and the right tag of the deallocated block are the paired tags of the new block. After adjusting the paired tags (the final size is equal to the sum of the sizes of each block, plus the size of the area to store two tags), the list adjustment is not required, since the left area has already been included in the list, and the increase in the area size does not affect the position of the list items.

d) The right block is free, and the left one is missing or occupied. Similar to the situation described in the paragraph "a", it is necessary to find the previous list item with respect to the right block and change the pointer of the next item at the beginning of the resulting block only after the block chaining.

e) The left and right blocks are free. In this case, three blocks are combined into one. The left tag of the left block and the right tag of the right block are the paired tags of the new block, and the size value is equal to the sum of the sizes of each block, plus the size of the area for storing four tags. After combining all the blocks in one, the right block should be excluded from the list to maintain the correct state of the list. Since on the list of the free blocks, all blocks are arranged in the address ascending order, the left block pointer to the next item will be equal to the initial address of the right block. The exclusion of the right block is made by replacing the left block pointer to the right block pointer to the next element.

f) The blocks deallocation algorithm is shown in Figure 5.

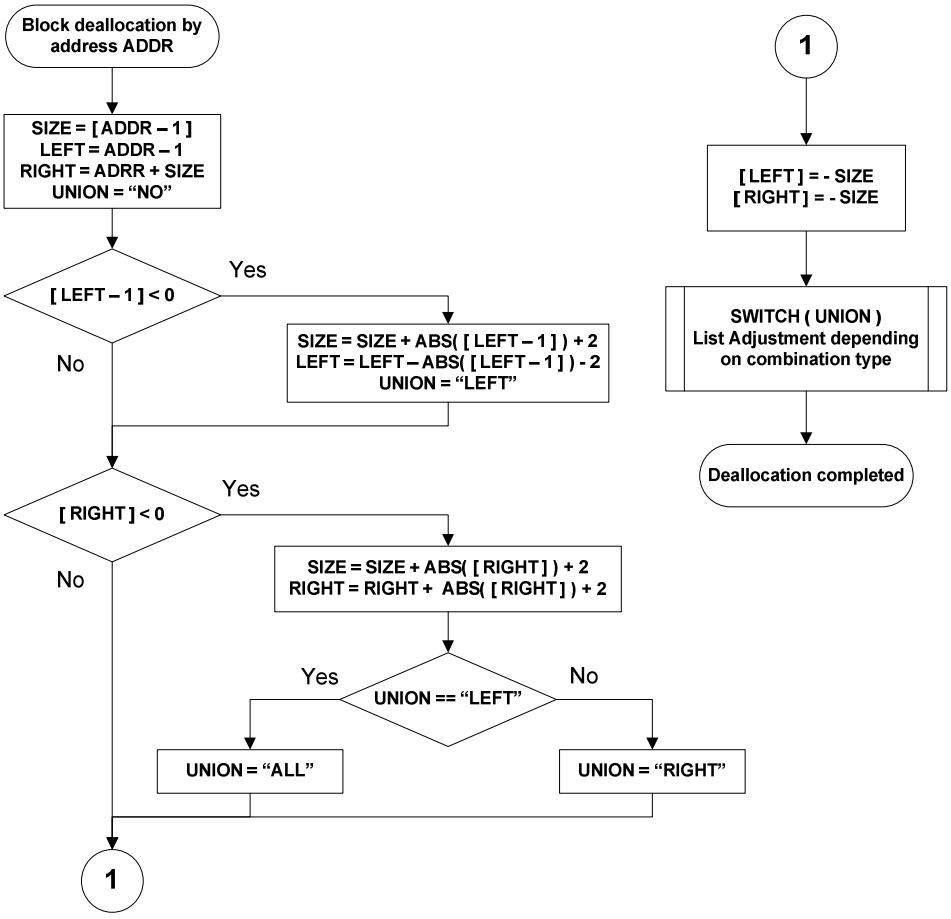

Figure 5. Block Deallocation Algorithm Chart

Journal of Theoretical and Applied Information Technology
29th February 2016. Vol.84. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The developed allocator, unlike the allocator of the standard C library, has a smaller allocated block size and makes it possible to use different methods of searching for a free block; the change of the search method can be performed during the allocator operation.

3. ALLOCATOR IMPLEMENTATION AND DEBUGGING

In different frameworks, the data types can have different sizes; for example, on one framework the pointer may take four bytes, while on the other it will take eight bytes. To avoid possible errors when compiling source code on different frameworks, native data types have been used, such as size_t or ssize_t. Features of the allocator implementation in C are given in [19] in more detail.

The allocator debugging is quite a difficult task, as it is a work with dynamic items. During each start of the program there are various options of the data organization and location in memory. The debugging is offered to be made in two steps.

The first step is to promote consistency of various memory requests (e.g. to determine the correctness of combining the related blocks in the allocator with a list of the clear areas), and view memory dumps for checking the allocator work (Figure 6).

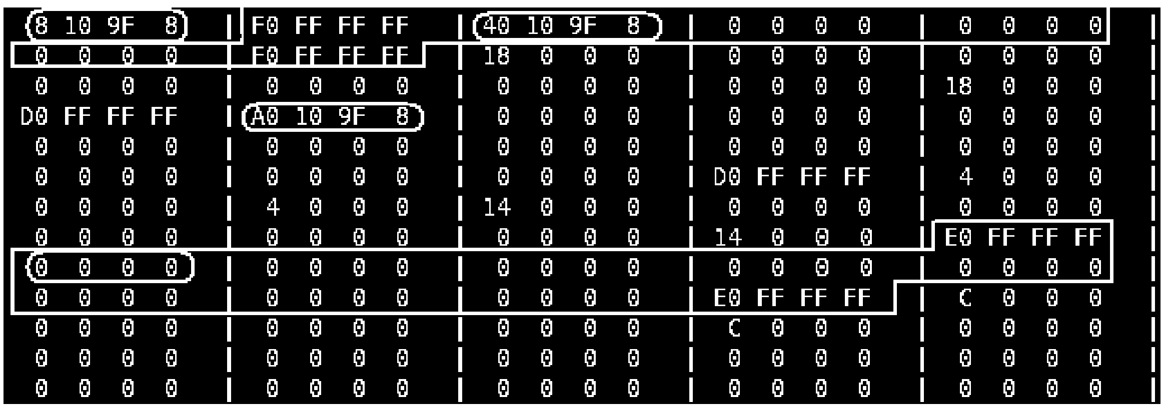

Figure 6. Example Of The Memory Dump For The Allocator Debugging. The Rectangles Mark The Clear Areas. The Ovals Circle The Pointers To The Next Items

The study of the memory dumps at this step is not difficult, because there are usually few requests for the memory allocation, the allocator area size is small, and it is easy to predict the expected results. In case of error situations, it is possible to use the debuggers to step through the program. The main errors are wrong type casting, incorrect address or displacement calculation, as well as an algorithm error.

After debugging the allocator, its testing is required using a special test program that generates random requests for memory allocation and deallocation. Using the test program the errors, which have not been detected at the first step, are found and fixed. In this case, the debugging is quite difficult, since the sequence of the allocator calls is random, and the errors occur each time at different steps of the test program implementation.

Step by step implementation of the program in the debugger before the error may take a long time. Browsing the dumps is also ineffective as the allocator memory area is quite big and tracking all the changes is very difficult in this area.

At this step, it is necessary to understand the possible cause of the error based on the logic of the allocator. The most common errors identified at this step are the lack of prior reset of any variables or areas, as well as errors in rarely running program hosts (in the allocator using the memory returns lists to the operating system when deallocating the last block with chaining with the penultimate block).

4. ALLOCATORS PARALLEL TESTING

The main criteria for comparison are operation speed and memory efficiency (availability of external and internal fragmentation). Here the operation speed is a block allocation and deallocation time (ideally, it is necessary to

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

compare the number of the executed processor instructions). The memory efficiency is a ratio of the requested memory size to the size of the entire memory area used by the allocator, including control footing. There are works that offer different allocators preliminary modeling to identify the most suitable one for a given application [13, 14]. However, most developers come down to possibilities of the standard allocator.

Since the operation of each allocator depends on many factors (a hardware framework, a sequence of requests for the memory allocation and deallocation, sizes of the requested block, etc.), the parallel testing mentioned above shows only the most common elements, and in different tasks and different frameworks the results may differ from those obtained by the authors. In addition, for any allocator, which does not displace the occupied blocks, there are sequences of requests for the memory allocation and deallocation, resulting in the inability to allocate memory for a sufficiently large amount of memory due to the external fragmentation issue. Thus, there may be situations when the allocator is suitable in all parameters of performance and memory efficiency on the basis of common tests, but in a particular task, it is not only ineffective, but even useless.

In this study, the testing of all developed allocators and the standard library allocator was carried out with the help of one test program described in the book [6].

First, it is necessary to check the correctness of the operation of an allocator. To this end, a certain value of one byte is randomly selected, and the entire selected area is filled with this value. Before removing the selected area, the content of this area is checked to meet the previously recorded value. If at least one discrepancy is found, it means that the data in the selected area have been changed during the allocator operation; the allocator algorithm has an error.

The test involves the performance of a predetermined number of iterations. On each such iteration, the following actions are performed.

- If possible, there is an allocation of memory block of a certain size. The lifetime is defined for the selected block (a number of the general iterations).
- If there are blocks, the lifetime of which has expired, their deallocation occurs.
- The lifetime of the remaining blocks decreases.

The size of the selected block is randomly selected from a predetermined range. The block lifetime is randomly selected from a predetermined range too.

The maximum block lifetime, the maximum number of the occupied blocks, the maximum block size, and the number of iterations are set before the test start.

After performing all the iterations of the main loop, a number of blocks is occupied in the program memory. The memory efficiency is calculated as a ratio of the total size of the occupied blocks to the allocator memory size. The working time is defined as a difference of the time tags obtained before the start of the main loop, and after its completion.

Further, all the remaining occupied blocks are deallocated, and the allocator memory area is checked again for return of the deallocated memory to the operating system.

With this testing algorithm after a certain number of iterations, the system comes to "equilibrium" when the number of the allocated blocks to every iteration is approximately equal. Therefore, increasing the number of iterations at the same values of the maximum number of the allocated blocks, the maximum lifetime, and the maximum size can only lead to an increase of the external fragmentation, but not to an increase in the number of the occupied blocks, and the memory used by them. For the simulation of continuous running programs in a real operating system, a sufficiently large number of iterations of the test program are performed.

This program makes it possible to get only an approximate idea of the testing allocator, but does not guarantee the same behavior of the allocator in specific tasks and specific frameworks.

The general scheme of the testing algorithm is shown in Figure 7.

Journal of Theoretical and Applied Information Technology
29th February 2016. Vol.84. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

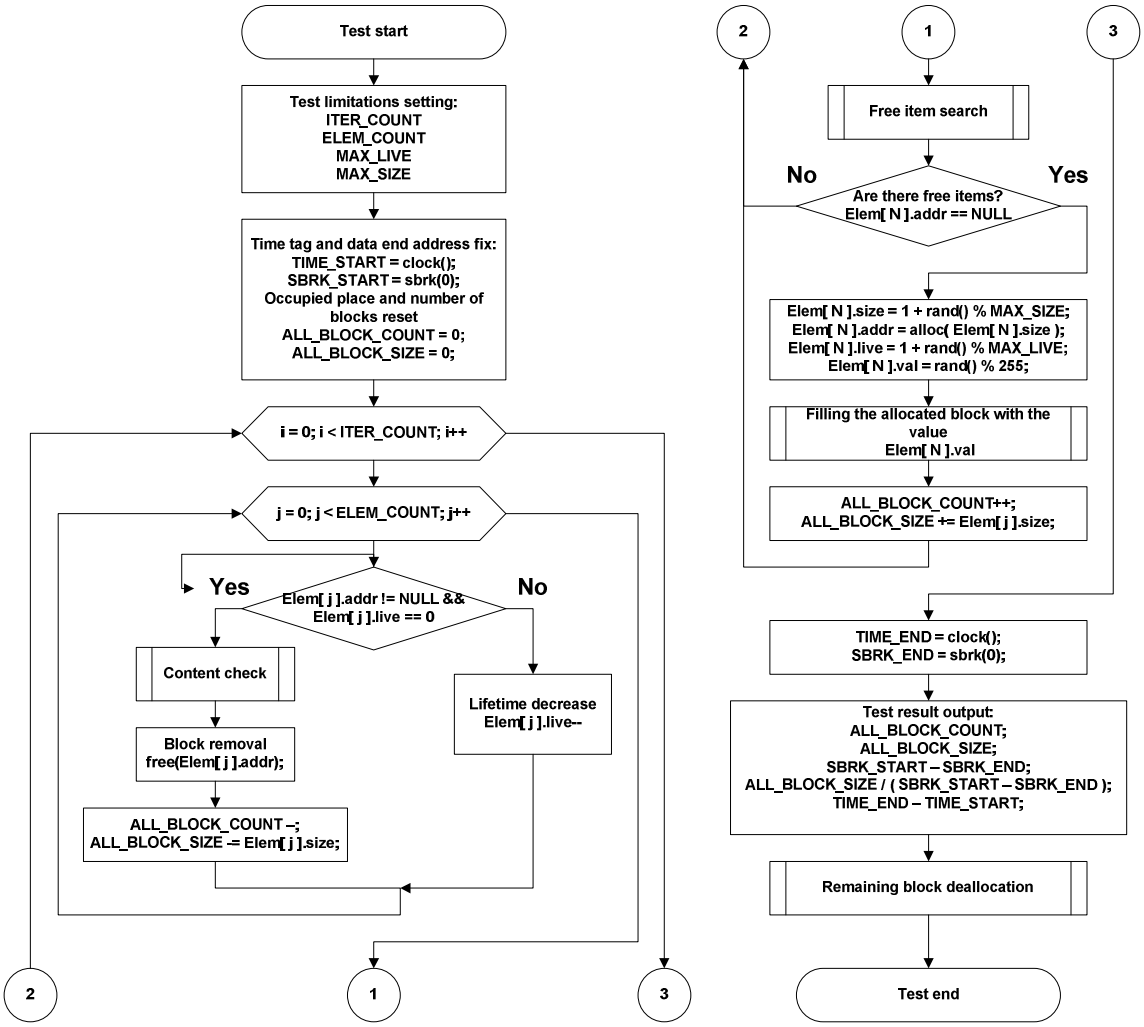

Figure 7. General Testing Algorithm Scheme

4.1. Random Size (x86) Block Allocation Test

The main results of this test are the following:

- Test time is necessary to compare the allocators' performance.
- A number of the occupied blocks at the test end.
- A summarized size of the occupied blocks.
- The allocator memory area size.
- The memory efficiency is necessary to compare the effectiveness of the allocators.
- The allocator memory area size after the deallocation of all occupied blocks reflects the allocator capacity to return unused memory to the operating system.

Each allocator has passed several tests with the x86 architecture in the library of glibc 2.13 edition, and the average results have been derived.

To have a general idea about each allocator, a test with the following input data was carried out:

- Number of iterations: 50,000
- The maximum number of allocated blocks: 5,000
- The maximum lifetime of an allocated block: 5,000
- The maximum size of an allocated block: 256 bytes

The test results of the standard library allocator (malloc) are presented in Table 1.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 1. Malloc Test Results

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	7,260	2,484	316,046	385,024	82.08	385,024
2	8,220	2,499	321,815	376,832	85.40	376,832
3	8,330	2,508	321,546	385,024	83.51	385,024
4	8,150	2,485	321,927	376,832	85.43	376,832
5	8,290	2,502	321,835	376,832	85.41	376,832
6	7,450	2,538	324,550	376,832	86.13	376,832
7	8,080	2,466	312,753	376,832	83.00	376,832
8	8,280	2,497	318,959	376,832	84.64	376,832
9	7,270	2,506	319,688	376,832	84.84	372,832
10	7,310	2,533	326,046	385,024	84.68	385,024
Total:	**7,864**	**2,502**	**320,517**	**379,290**	**84.51**	**378,890**

The allocator test results with the lists of clear areas (lsalloc) with the free block search by best-fit method are presented in Table 2.

Table 2. Lsalloc Test Results (Best-Fit Method)

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	9,040	2,550	327,727	381,292	85.95	8
2	9,220	2,525	334,644	391,359	85.51	8
3	9,230	2,497	328,889	383,122	85.84	8
4	9,390	2,479	317,949	376,636	84.42	8
5	9,410	2,508	315,058	379,662	82.98	8
6	9,180	2,513	317,707	376,540	84.38	8
7	9,380	2,451	313,902	369,685	84.91	8
8	9,350	2,547	333,657	388,548	85.87	8
9	9,270	2,505	328,771	389,211	84.47	8
10	9,100	2,496	316,608	373,516	84.76	8
Total:	**9,257**	**2,507**	**323,491**	**380,957**	**84.91**	**8**

The allocator test results with lists of the clear areas (lsalloc) with the free block search by first-fit method are presented in Table 3.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 3. Lsalloc Test Results (First-Fit Method)

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	9,000	2,511	323,396	385,132	83.97	8
2	8,240	2,512	323,510	381,282	84.85	8
3	9,260	2,498	328,197	385,539	85.13	8
4	9,160	2,476	317,757	377,199	84.24	8
5	9,310	2,502	317,421	376,487	84.31	8
6	9,250	2,508	321,822	379,369	84.83	8
7	9,610	2,499	325,263	385,166	84.45	8
8	9,170	2,513	325,777	386,530	84.28	8
9	9,260	2,463	313,504	389,704	80.45	8
10	9,170	2,496	320,795	379,688	84.49	8
Total:	**9,143**	**2,498**	**321,744**	**382,610**	**84.10**	**8**

The allocator test results with lists of the clear areas (lsalloc) with the free block search by worst-fit method are presented in Table 4.

Table 4. Lsalloc Test Results (Worst-Fit Method)

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	9,240	2,572	334,662	390,721	85.65	8
2	9,190	2,487	319,781	380,975	83.94	8
3	7,690	2,436	317,932	392,598	80.98	8
4	7,810	25,39	322,042	390,277	82.52	8
5	8,360	2,509	318,102	376,541	84.48	8
6	7,920	2,536	332,655	386,316	86.11	8
7	7,850	2,502	325,626	393,256	82.80	8
8	8,210	2,471	317,236	384,794	82.44	8
9	7,850	2,510	328,182	385,110	85.22	8
10	7,840	2,559	330,391	385,740	85.65	8
Total:	**8,196**	**2,512**	**324,661**	**386,633**	**83.98**	**8**

Final results of the tests are presented in charts in Figures 8, 9 and 10.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

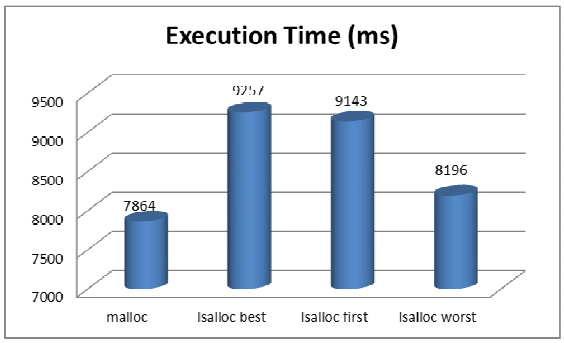

Figure 8. Test Execution Time Chart

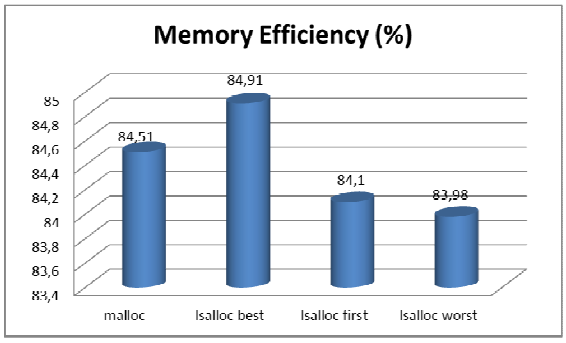

Figure 9. Allocators Memory Efficiency Chart

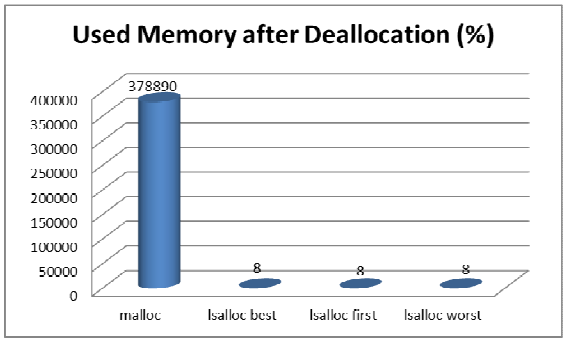

Figure 10. Allocators Memory Return To Operation System Chart

On the basis of the test results, the following conclusions can be made:

- While increasing the memory efficiency, the allocator operation speed is significantly reduced.
- The pointer "size of the returned memory to the operating system after all blocks deallocation" is very important. For example, if the program makes extensive use of dynamic memory allocation in the initial stage of implementation, and then deallocates all the allocated blocks and continues to implement without the use of the dynamic memory, the "holding" of the allocated memory is impractical, and in addition, such applications may have a negative impact on the operating system as a whole. The test results show that the standard library allocator does not always return the entire memory to the operating system.

The results of these tests cannot be considered as entirely objective, as they were carried out with a small number of iterations, and the execution time also depended on the operating system load during the test. In order to obtain more accurate results, it was necessary to test with a large number of iterations.

For further testing, the allocators are selected using the free blocks lists and the standard library allocator; the number of iterations has been increased to 5,000,000.

The test results of the standard library allocator (malloc) are presented in Table 11.

Table 11. Malloc Test Results

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	752,120	2,502	317,836	393,216	80.83	393,216
2	799,010	2,519	327,373	393,216	83.26	393,216
3	841,330	2,535	324,419	393,216	82.50	393,216
4	839,210	2,535	323,231	393,216	82.20	393,216
5	840,680	2471	313,815	393,216	79.81	393,216
Total:	**814,470**	**2,512**	**321,335**	**393,216**	**81.72**	**393,216**

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The allocator test results with the lists of clear areas (lsalloc) with the free block search by best-fit method are presented in Table 12.

Table 12. Lsalloc Test Results (Best-Fit Method)

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	854,870	2,488	317,207	384,405	82.52	8
2	812,810	2,479	320,419	385,656	83.08	8
3	930,460	2,524	318,268	378,336	84.12	8
4	938,480	2,522	320,720	385,017	83.30	8
5	925,700	2,453	311,862	382,169	81.60	8
Total:	**892,464**	**2,493**	**317,695**	**383,117**	**82.93**	**8**

The allocator test results with the lists of clear areas (lsalloc) with the free block search by first-fit method are presented in Table 13.

Table 13. Lsalloc Test Results (First-Fit Method)

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	822,200	2,478	316,324	380,748	83.08	8
2	810,460	2,512	329,606	385,256	85.56	8
3	933,530	2,535	320,683	381,121	84.14	8
4	929,800	2,495	319,975	384,734	83.17	8
5	932,310	2,486	326,239	386,161	84.48	8
Total:	**885,660**	**2,501**	**322,565**	**383,604**	**84.09**	**8**

Final results of the tests when the number of iterations is 5,000,000 are presented in charts in Figures 11 and 12.

Figure 11. Test Execution Time Chart

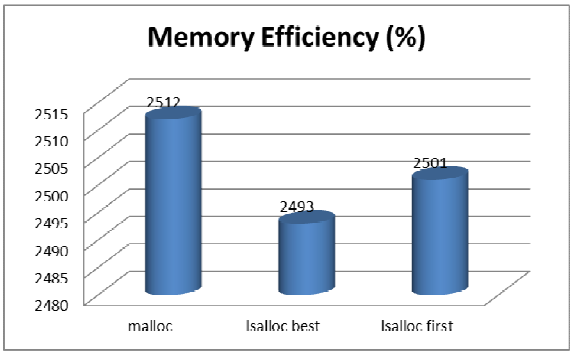

Figure 12. Allocators Memory Efficiency Chart

The tests show that the increase in the number of iterations has quite a strong effect on the final results. This is not surprising, as the main deficiencies of the allocators appear in applications

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

that are implemented without interruption for a long time, for hours, days, months. One of the ways of getting rid of the fragmentation during runtime is to restart the application; after restarting, the memory manager area is re-initialized, and in the initial stages the fragmentation is minimized. But there are applications where a restart for some reason cannot be performed or is performed infrequently.

In longer tests performing, a reduction in the difference of test execution time can be observed. For example, in the first tests (when the number of iterations equals to 50,000), the allocator test execution time with the clear areas list by the best-fit method was 15% more than time necessary to perform the allocator standard library test, and in the last tests, it is more up to 9%.

Also, there is a significant difference in memory efficiency. If in early tests the difference is only a few tenths of a percent, in that case the difference is almost three percent more. The fact is interesting that in the case of longer execution time using the first-fit method under the free block searching in the allocator with the clear areas list gives significantly better results in performance and memory efficiency compared to the best-fit method.

4.2. Random Size (x64) Block Allocation Test

Using the x64 architecture, the testing was carried out similar to the testing presented in the previous paragraph. The following allocators were tested: malloc, lsalloc (first-fit), lsalloc (best-fit). The standard library glibc 2.13 edition was used.

The input data for the test:

- Number of iterations: 5,000,000
- The maximum number of allocated blocks: 5,000
- The maximum lifetime of an allocated block: 5,000
- The maximum size of an allocated block: 256 bytes

The test results of the standard library allocator (malloc) are presented in Table 14.

Table 14. Malloc Test Results

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	574,810	2,512	319,260	434,176	73.53	434,176
2	580,320	2,500	325,368	425,984	76.38	425,984
3	733,770	2,528	325,017	425,984	76.30	425,984
4	575,010	2,510	327,593	425,984	76.90	425,984
5	614,470	2,493	325,721	425,984	76.46	425,984
Total:	**615,676**	**2,509**	**324,592**	**427,622**	**75.92**	**427,622**

The allocator test results with the lists of clear areas (lsalloc) with the free block search by best-fit method are presented in Table 15.

Table 15. Lsalloc Test Results (Best-Fit Method)

Test No.	Execution Time (ms)	Number of Blocks	Block Size (byte)	Area Size (byte)	Efficiency (%)	Area Size After Deallocation (byte)
1	594,520	2,543	327,719	415,052	78.96	16
2	594,300	2,453	315,876	409,323	77.17	16
3	588,780	2,467	321,395	410,127	78.36	16
4	593,110	2,452	314,674	412,397	76.30	16
5	593,370	2,480	317,793	399,583	79.53	16
Total:	**592,816**	**2,479**	**319,491**	**409,296**	**78.07**	**16**

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The allocator test results with the lists of clear areas (lsalloc) with the free block search by first-fit method are presented in Table 16.

Table 16. Lsalloc Test Results (First-Fit Method)

Test No.	Execution time (ms)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency (%)	Area size after deallocation (bytes)
1	589,070	2,500	322,498	409,535	78.75	16
2	593,790	2,525	327,116	407,251	80.32	16
3	594,480	2,450	317,471	405,964	78.20	16
4	593,010	2,496	321,755	408,961	78.68	16
5	592,040	2,452	317,858	404,222	78.63	16
Total:	**592,478**	**2,485**	**321,340**	**407,187**	**78.92**	**16**

Final results of the tests are presented in charts in Figures 13, 14 and 15.

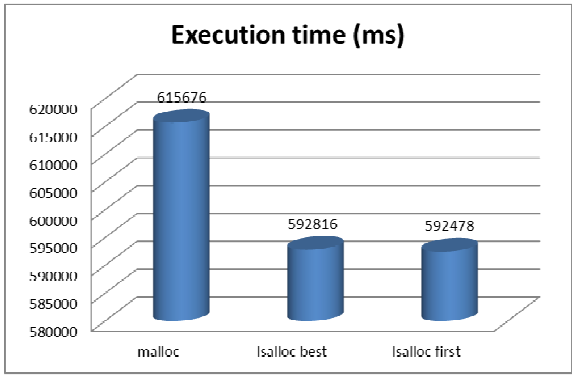

Figure 13. Test Execution Time Chart

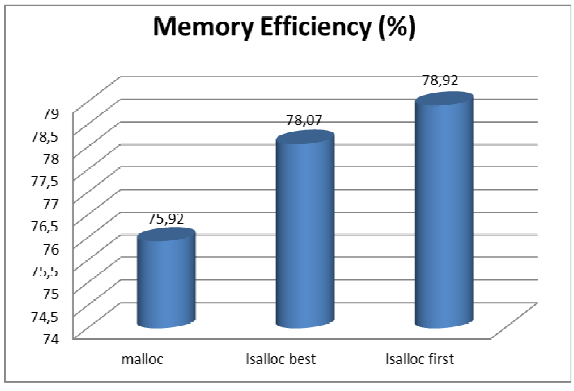

Figure 14. Allocators Memory Efficiency Chart

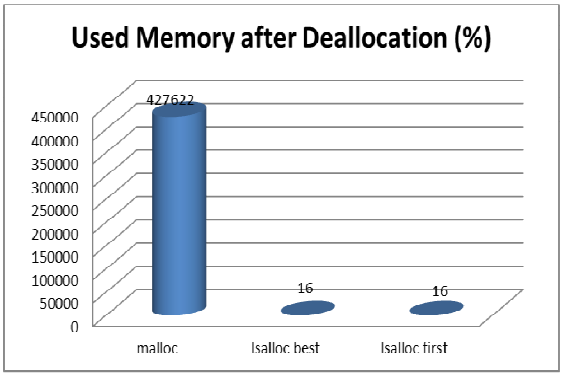

Figure 15. Allocators Memory Return To Operation System Chart

5. CONCLUSION

The test results of the x64 architecture are quite different from the test results in the x86 architecture. The main cause is the size of the pointers in each architecture. There shall be 4 bytes to store the addresses in a computer with x86, and 8 bytes to store the addresses in a computer with x64. At first glance, the difference is not so great, but if it is considered that the dynamic memory allocation algorithms in addition to the data store a large number of the pointers for the lists organization, as well as the intermediate pointers, the difference in the total memory amount for storing the pointers becomes noticeable.

On the average, for all algorithms, the memory efficiency fell by seven per cent.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The comparison of the performance shall be carried out within the architecture, where the tests have been conducted. The allocator with the clear areas lists is somewhat more efficient in the use of memory than the standard library allocator, although it loses in performance a little. Using the best-fit method is justified for a small number of the requested blocks, or for a small number of memory allocation and deallocation loops. In other cases, the best-fit method has a better performance when searching for the free blocks. Though, the best-fit method did not show outstanding results, but its testing may also be of interest to the developers. The return of almost all memory to the operating system is an important advantage of the allocator with the clear areas lists during the deallocation of all the occupied blocks.

In this paper, different algorithms for dynamic memory allocation have been considered, and based on them the allocator with the free blocks list has been developed and implemented in the programming language C.

All the developed allocator options have been tested using the test program. The test results have showed that the developed allocators are able to compete with the memory manager of the C standard library, and the benefits of each allocator are manifested in different tasks. Also, the test results have revealed that the memory performance and efficiency of the allocators depend not only on their algorithms, but also on many other factors, such as the architecture of a computer system, the size of the allocated blocks, the duration of the memory allocation and deallocation. Therefore, it is not clear that one allocator is better than another. When choosing the allocator for a specific task, it is necessary to hold a series of tests and to identify the most appropriate one.

The developed allocators are not ideal for all memory dynamic allocation tasks, and can be improved for specific tasks due to insignificant changes in algorithms or optimizing the program code for a specific architecture. Moreover, the developed allocators can be applied not only in the family of UNIX operating systems, but also in other frameworks with minor changes of the source code.

The developed allocators provide a choice to the application programmers, and their source codes may be useful for the system programmers in dealing with dynamic memory allocation issues.

REFERENCES:

[1] Glass, G. and K. Eybls, 2004. UNIX for Programmers and Users (3rd ed., revised and enlarged). Saint Petersburg: BHV-Petersburg.

[2] GNU C Library Source Code, n.d. Date Views 03.05.2011 http://ftp.gnu.org/gnu/glibc/glibc-2.13.tar.gz.

[3] Hasan, Y. and M. Chang, 2005. A Study of Best-Fit Memory Allocators. Computer Languages, Systems & Structures, 31(1): 35-48. Date Views 10.02.2016 dx.doi.org/10.1016/j.cl.2004.06.001.

[4] Irtegov, D., 2008. Introduction to Operating Systems (2nd ed.). Saint Petersburg: BHV-Petersburg.

[5] Jones, M.T., 2009. Anatomy of Linux Dynamic Libraries. Date Views 03.05.2011 www.ibm.com/developerworks/ru/library/l-dynamic-libraries/index.html.

[6] Knuth, D.E., 2005. Art of Computer Programming: Vol. 1. Basic Algorithms (3rd ed.: Trans. from English). Moscow: Williams Publishing House.

[7] Lea, D., 1996. A Memory Allocator. Date Views 10.04.2011 http://gee.cs.oswego.edu/dl/html/malloc.html.

[8] Love, R., 2008. Linux. System Programming. Saint Petersburg: Peter.

[9] Mamagkakis, S., C. Baloukas, D. Atienza, F. Catthoor, D. Soudris and A. Thanailakis, 2006. Reducing Memory Fragmentation in Network Applications with Dynamic Memory Allocators Optimized for Performance. Computer Communications, 29(13-14): 2612-2620. Date Views 10.02.2016 dx.doi.org/10.1016/j.comcom.2006.01.031.

[10] Masmano, M., I. Ripoll, J. Real, A. Crespo and A.J. Wellings, 2008. Implementation of a Constant-Time Dynamic Storage Allocator. Software: Practice and Experience, 38(10): 995-1026.

[11] Masmano, M., I. Ripoll, P. Balbastre and A. Crespo, 2008. A Constant-Time Dynamic Storage Allocator for Real-Time Systems. Real-Time Systems, 40(2): 149-179.

[12] Rezaei, M. and K.M. Kavi, 2006. Intelligent Memory Manager: Reducing Cache Pollution due to Memory Management Functions. Journal of Systems Architecture, 52(1): 41-55. Date Views 10.02.2016 dx.doi.org/10.1016/j.sysarc.2005.02.004.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[13] Risco-Martín, J.L., J.M. Colmenar, D. Atienza and J.I. Hidalgo, 2011. Simulation of High-Performance Memory Allocators. Microprocessors and Microsystems, 35(8): 755-765.

[14] Risco-Martín, J.L., J.M. Colmenar, D. Atienza and J.I. Hidalgo, 2011. Simulation of High-Performance Memory Allocators. Microprocessors and Microsystems, 35(8): 755-765.

[15] Risco-Martín, J.L., J.M. Colmenar, J.I. Hidalgo, J. Lanchares and J. Díaz, 2014. A Methodology to Automatically Optimize Dynamic Memory Managers Applying Grammatical Evolution. Journal of Systems and Software, 91: 109-123. Date Views 10.02.2016 dx.doi.org/10.1016/j.jss.2013.12.044.

[16] Robachevsky, A.M., S.A. Nemnyugin and O.L. Stesik, 2008. UNIX Operating System (2nd ed., revised and enlarged). Saint Petersburg: BHV-Petersburg.

[17] Tanenbaum, E., 2010. Modern Operating Systems (3rd ed.). Saint Petersburg: Peter.

[18] The GNU C Library. Virtual Memory Allocation and Paging, n.d. Date Views 22.04.2011 http://www.gnu.org/software/libc/manual/html_node/Memory.html.

[19] Vavrenyuk, A.B., I.V. Karlinskiy, A.P. Klarin, V.V. Makarov and V.A. Shurygin, 2015. Dynamic Allocation of Memory Blocks of the Same Size. Journal of Theoretical and Applied Information Technology, 80(3): 419-430.