Journal of Theoretical and Applied Information Technology
31st October 2015. Vol.80. No.3

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

DYNAMIC ALLOCATION OF MEMORY BLOCKS OF THE SAME SIZE

ALEKSANDR BORISOVICH VAVRENYUK
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe highway, 31, Moscow, 115409, Russia
IGOR VLADIMIROVICH KARLINSKIY
Karlsruhe Institute of Technology (KIT)
Kaiserstraße, 12, Karlsruhe, 76131, Germany
ARKADY PAVLOVICH KLARIN
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe highway, 31, Moscow, 115409, Russia
VIKTOR VALENTINOVICH MAKAROV
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe highway, 31, Moscow, 115409, Russia
VIKTOR ALEXANDROVICH SHURYGIN
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Kashirskoe highway, 31, Moscow, 115409, Russia

ABSTRACT

The problem of allocating and freeing operative memory in multiprogramming is relevant for all modern operating systems. The algorithms of almost all memory managers (allocators) used, claiming being universal, either lead to decreasing efficiency of memory usage, or require significant CPU time. This article describes the allocator algorithm proposed by the authors that makes it possible to achieve greater efficiency in using memory when blocks of the same size are allocated. An allocation testing method has been described, and the results of comparing the proposed allocator to a standard allocator of a UNIX system GNU C library have been described.

Keywords: *Allocator, Memory Manager, Bit Matrix, Memory Fragmentation, Operating Systems*

1. INTRODUCTION

Today, multiprogramming mode is implemented virtually in all modern operating systems. This inevitably raises the problem of rational use of PC RAM. At the same time, practice shows that it is essentially impossible to design a universal algorithm for controlling memory. The desire to utilize available memory capacity to the bigger extent inevitably leads to additional CPU time usage during execution of the corresponding memory manager ("allocator") and, vice versa, faster algorithms lead to additional usage of memory for storing own data structures.

Design of "optimal" allocators has been the subject of quite a lot of works of domestic and foreign authors [2], [4], [7], [8], [10]. The result of collective efforts of many authors was standard allocator glibc of the GNU C library [9]. This allocator uses many modern allocation ideas, e.g., the "paired tags algorithm", "twin system", use of "bit matrices", etc.

One of the main problems for any algorithm of the allocator is fighting memory fragmentation [12]. Another important problem is significant cost of CPU time for running allocator's own data structures [15].

There are interesting publications about the problems of memory allocation for real-time systems, which require, above all, reducing CPU time in performing all functions of the allocator [19], [20].

Several authors propose a method for automatic optimization of memory managers [18].

However, due to the possibility to quickly change allocators in modern operating systems, adapting to current needs of the computational process,

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

depending on the characteristics of the tasks solved in the system, it seems appropriate to design an allocator for using it in the system with certain requirements from a mixture of tasks being resolved.

This paper presents the results of studying one of possible algorithms for allocating and deallocating RAM proposed by the authors, in response to requests for fixed size memory areas from applications. In practice, such requests may occur, e.g., when solving large-scale homogeneous computing tasks in the system.

2. DEVELOPMENT OF AN ALTERNATIVE MEMORY MANAGER

If an application always requires blocks of the same size, it makes sense to use an allocator for this very task. In work [5] a method of managing blocks of the same size using bit matrices was shown. The use of bit matrices has an advantage over lists of free blocks, namely, smaller size of service information.

In order to organize a bit matrix, it is necessary to know the approximate number of blocks that it will manage. Since memory is allocated to applications in pages, defining the number of blocks may be made as follows (Fig.1):

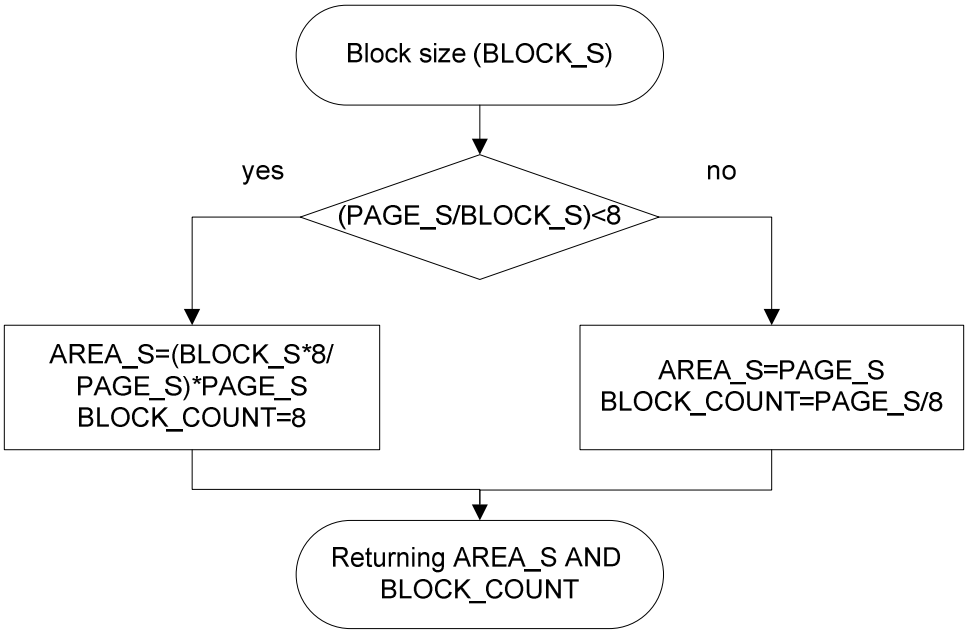

Fig.1 Flowchart of the algorithm for calculating the number of blocks and memory area size

If block size allows placing eight or more blocks on one page, their approximate number is equal to the page size divided by block size, and the operating system will be asked for one page.

If block size does not allow placing eight more blocks on one page, their number is taken as a minimum (eight blocks minus one byte), and the operating system will be asked for a certain number of pages required to placing eight blocks.

Requests for memory allocation from the operating system may be arranged in portions, dimensions of which are calculated on the basis of the size of one block. Each such area will manage a certain number of blocks; it will be a "container" for blocks. If all units are occupied in one such container, and there is the need to place additional blocks, it is necessary to allocate another container. If the container is empty, it can be destroyed, and the used memory may be returned to the operating system. In order to manage containers, it is logical to use a double-linked list, which will make it possible to quickly navigate between them when searching for a free block.

Each container should store the following service information:

- a pointer to the previous container;
- a pointer to the next container;
- the number of occupied blocks;
- the number of bytes in the bit matrix used;

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

• the number of the byte with free blocks in the bit matrix;

It is convenient to store this information at the beginning of each container, and it can be defined as container header. The address space of each container would then have the form shown in Figure 2.

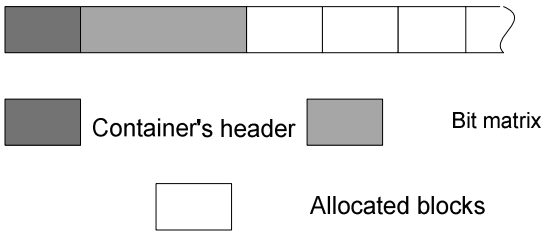

Fig.2 Container's address space

Now it is possible to determine the exact size of the container and the exact number of blocks in it, considering the title of the container, and on the basis of the matrix size. When calculating the number of blocks, the size of the page will be taken into account with consideration of the header and one byte of the bit matrix. After defining container size, there will be an additional test, so that when placing eight objects into the container, place remained for the header and one byte of the bit matrix. If there is no space left for service information, the size of the container is increased by one page. The complete version of the algorithm for calculating container size is shown in Figure 3

.

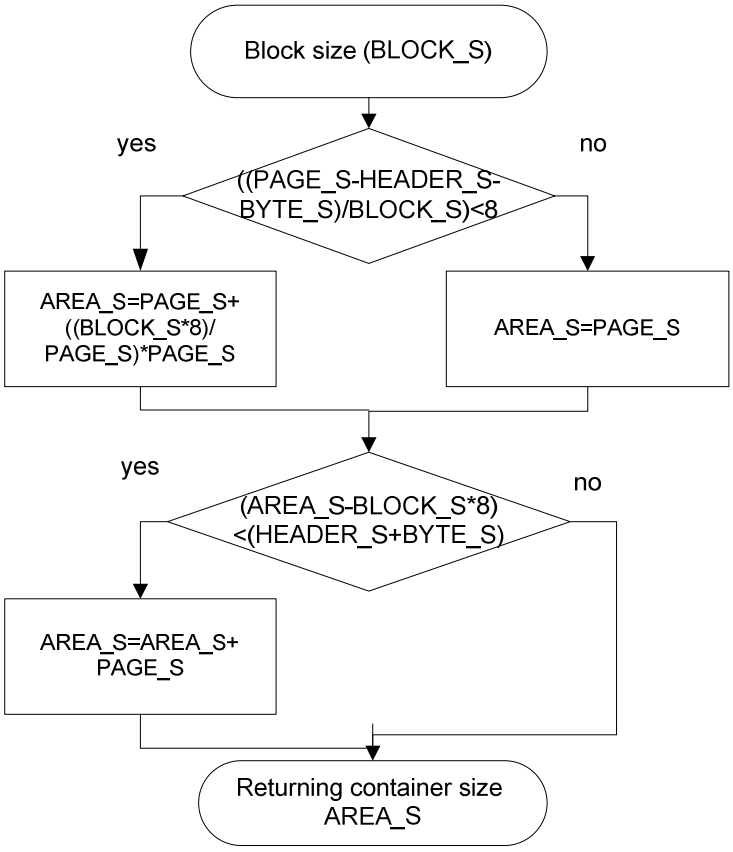

Fig 3. Flowchart of the algorithm for calculating container size

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

All division operations on the flow chart are integers, i.e. after division only the integer part is taken (no rounding), and the fraction is discarded.

After the container is separated, it is necessary to initiate the header: pointers to next and previous containers are set in accordance with the list of containers; the number of the occupied units and the number of bytes in which there are free blocks are set to zero; and the number of bytes in the bit matrix is calculated on the basis of the block size. The exact number of bytes in the bit matrix is calculated in the following sequence:

1. First, the guaranteed free area in the container is calculated (1). The size of the header and the size of the bit matrix required to accommodate the maximum number of blocks in the container is subtracted from the size of the container.

$$AREA\_FREE = AREA\_S - HEADER\_S - AREA\_S / BLOCK\_S / 8 \quad \textbf{(1)}$$

2. Then, on the basis of the guaranteed free area in the container, the exact size of the bit matrix is calculated (2). The number of blocks is taken to be a multiple of 8, so that the bit matrix uses all bits of each byte.

$$BITMATR\_SIZE = AREA\_FREE / BLOCK\_S / 8 \quad \textbf{(2)}$$

This mechanism of calculating size of the bitmap makes it possible to properly place the header, the bit matrix and the blocks themselves in the container. Of course, for such computation, not all memory of the container may be used, but we can guarantee that all blocks are placed in the container, and a correct bit matrix will be created for managing them.

A free block to be allocated to an application is searched for in several stages. At the first stage there is a container with free blocks. The presence of free blocks in a container may be determined in several ways. The container is occupied if the number of occupied blocks is equal to the number of bytes in the bit matrix multiplied by eight (3), since each byte addresses exactly eight blocks, and calculation of the bitmap size ruled out the use of "incomplete" bytes.

$$BUZY\_COUNT = BITMATR\_SIZE * 8 \quad \textbf{(3)}$$

The second method is checking the number of the "free" byte in the header of the container. The numbering of the bytes is zero-based, for filling each byte of the bit matrix, the number of the "free" byte is found anew, and if no free blocks are available, it is equated to the bit matrix size (the next byte after the last one in the bit matrix).

Thus, the availability of free blocks in the container is found by simple comparison of the bit matrix size and the number of free bytes, whose values are stored in the container header. If the values are equal, the container is completely occupied, otherwise there are free blocks. It is better to define the status of the container this way, since there is no need to perform multiplication, as in (3).

If the container is fully occupied, the next container is considered according to the link in the header of the container. If the end of the containers list is reached, the allocator tries to request memory from the operating system for placing another container. If memory allocation was successful, a new container is added to the end of the list, and a free clock is allocated to the application in it. The algorithm of containers management is shown in Figure 4.

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

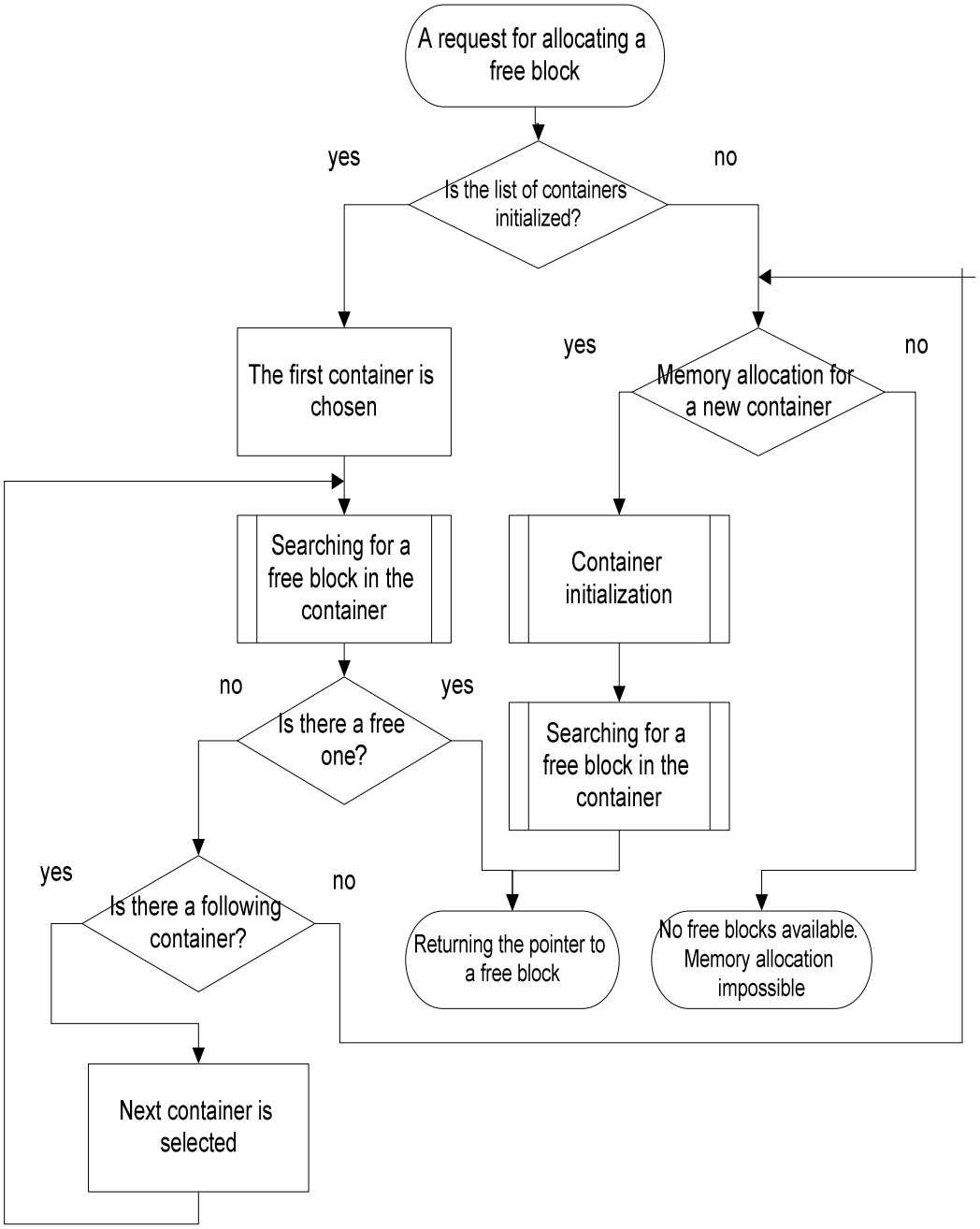

Fig.4 Flowchart of the algorithm of searching a container with a free block

After successful definition of the container with a free block, it is necessary to find the address of the free block and invert the bit value responsible for this block in the bit matrix. The number of the byte with the free block is shown in the header of the container. Let us take value "1" for an occupied block, and "0" for a free block. The byte of the bit matrix contains position of the first free bit, starting with the most significant bit, and the value of this bit is inverted (set to "1"). The block address is easily defined by calculating its shift from the end of the matrix (4).

$$SHIFT = (BLOCK\_S * 8) * NUM\_FREE\_BYTE + POS\_BIT \quad (4)$$

(4)

where:
SHIFT is the shift from to the end of the bit matrix,

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

BLOCK_S is the size of one block;
NUM_FREE_BYTE is the number of the byte in a bit matrix with a free block,
POS_BIT is the position of the bit in the byte, corresponding to the free block.

After the container is separated, it is necessary to initiate the header: pointersto the next and previous containers are set in accordance with the list of containers; the number of the occupied units and the number of bytes in which there are free blocks are set to zero; and the number of bytes in the bit matrix is calculated on the basis of the block size.

The procedure of "removing" an occupied block is similar to the searching procedure, but in this case it is necessary to find container number from block address, the number of the byte in the bit matrix and the bit position. Since all containers are arranged in the same manner, and are placed in the list consecutively (in address ascending order), precise positions are determined using simple formulas.

Starting with the first container, the shift of block address is calculated, relative to the starting address of the container. If the offset is greater than the size of one container, it means that the block searched for is in the following containers, and the system goes to the next container.

Thus, the offset in each container is calculated until the unit's belonging to a specific container is found. After finding the container that contains this block, the offset in blocks is calculated relative to the beginning of the container, i.e. block's sequence number within the container (5).

$$SHIFT\_BLOCK = (BLOCK\_ADDR - START\_ADDR) / BLOCK\_S \quad \textbf{(5)}$$

where:
SHIFT_BLOCK is the block number in the list of blocks,
BLOCK_ADDR is the address of released block,
START_ADDR is the start address of the container,
BLOCK_S is the size of one block.

The number of the byte that is responsible for this block in the bit matrix is obtained simply by dividing the block "shift" by eight, and the bit order in "control" byte will be equal to the remainder of division of block "shift" by eight (6).

$$NUM\_BYTE = SHIFT\_BLOCK / 8; NUM\_BIT = SHIFT\%8 \quad \textbf{(6)}$$

The received bit is inverted (set to 0); the number of occupied blocks in the container is reduced by one. If the number of occupied blocks is equal to zero, and the container is the "last" (located at the end of the list of containers), and not the first, then the memory allocated for this container can be returned to the system. The algorithm of block deallocation is shown in Figure 5.

The used algorithms and principles are quite simple to understand and implement in the form of software code in any programming language. In writing this allocator in the C programming language, in order to search for a free block and deallocate the occupied blocks, recursive function to cycle to the next container have been implemented in the list of containers.

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

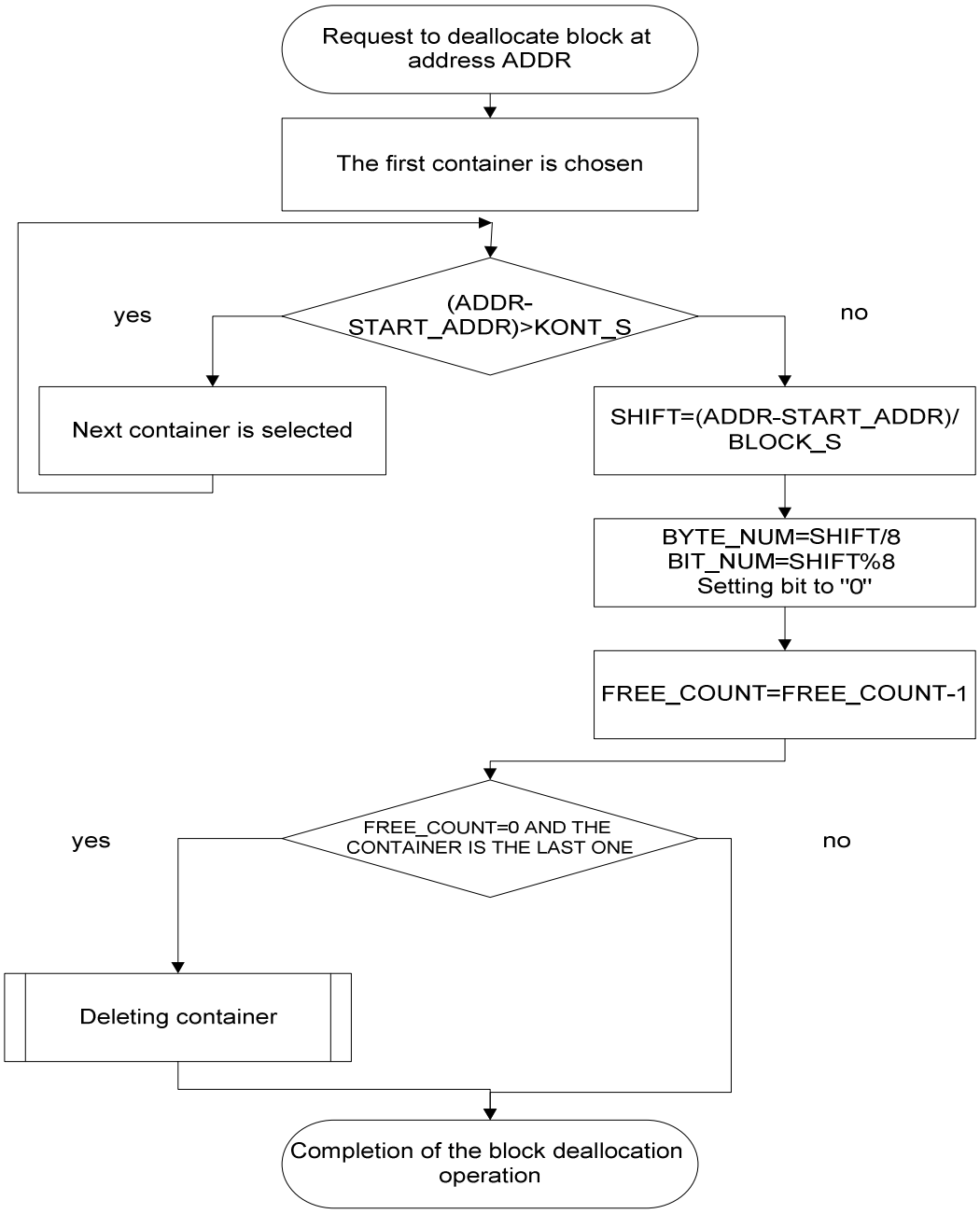

Fig.5 Flowchart of the block deallocation algorithm

Some limitations should be noted in using the allocator described, due to the fact that the size of each container is a multiple of memory page size, and for certain block sizes internal fragmentation (unused area in the container) can be quite large. However, for certain block size, container memory is used nearly completely, and the efficiency of memory usage is generally higher than in case of organizing the allocator with the use of lists of free blocks.

3. IMPLEMENTATION FEATURES IN THE C PROGRAMMING LANGUAGE

On various platforms data types may have different size, for example, a pointer may take four bytes on one platform and eight on another. To avoid possible errors in compilation of the source code on different platforms platform-specific data types were used, such as size_t or ssize_t. On

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

different platforms such data types are overridden by using the #define compiler directive in header files, for example in stddef.h.

In calculating addresses shifts, to support correctness of compilation and work on various platforms, the use of constant values is excluded. For example, if we know that address A stores a value of type int (4 bytes), the address after this value will be equal to A + 4. But if on another platform type int takes 8 bytes, addressing to A + 4 would be erroneous. Moreover, the program may be compiled without errors, and even may work properly for some time, but this "overwriting" of a part of the variable can lead to unexpected errors, which are difficult to identify and to eliminate. In order to eliminate these errors when calculating the correct offset, the sizeof operator is used, which calculates the correct size of data type. Replacing calculation of address A + 4 with A + sizeof(int) ensures correct memory access when the program is compiled on different platforms.

The main purpose of the allocator is work with memory, and in the C programming language, pointers are used for these purposes. A pointer holds the memory address value, and the pointer may be used to gain access to the data stored at this address. Since in allocator development data is mainly unstructured (it is not always possible to use a pointer to data structure for obtaining specific values), the programmer has to remember what data is stored in memory, and where exactly. Mainly, pointers of types void * and void ** (pointer to pointer) are used, and for obtaining data, explicit type indication is used. For example, if a pointer to A (type void *) points to data of type int, the access can be obtained with the help of operations of dereference (operation *) and type casting: * (int *) A. And if the pointer A points to a data structure (struct data), the access to the fields of the structure is obtained by using operations of indirect selection (operator ->) and type conversion: ((struct data *) A)->member.

4. COMPARATIVE TESTING OF ALLOCATORS

Comparison of various allocators is quite a challenging task. There are even works where preliminary modeling of various allocators is proposed in order to identify the most suitable ones for certain applications [13]. In practice, however, programmers prefer to use standard allocators, or write an allocator their own, if the standard one is not suitable by some criteria.

The main criteria are speed and efficiency of memory usage, which decreases due to the presence of external and internal fragmentation. Operation speed refers to the time of block allocation and deallocation (ideally, the number of operations executed by the processor should be compared). Memory efficiency is understood as the relationship between requested memory size and the size of a memory used by the allocator for completing the request, including data and service information.

Work of each allocator depends on many factors, such as the platform, allocator scope of application, the sequence in which memory the allocation and deallocation requests come, the size of requested blocks, etc. It is necessary to take into account the fact that many allocators permit making additional settings that affect their execution. Therefore, comparative testing of allocators shows general issues, and in different tasks and in different platforms, the test results may vary significantly.

It is worth remembering that for any allocator that does not move occupied blocks, there is such a sequence of requests for memory allocation and deallocation that leads to the impossibility of allocating enough free memory (the problem of external fragmentation). Due to this fact, situations may develop, where the allocator is suitable in all aspects of the performance and efficiency of memory usage, on the basis of general tests, but in specific task it proves to be ineffective.

In this work, all allocators were tested with the use of the same testing program described in book [6] with various input data and on various platforms. A standard library allocator will be used with default settings, except for the M_TOP_PAD option, which will be set to 4,096 bytes. This is necessary for correct assessment of memory usage efficiency in case of small total size of all blocks.

The essence of the test is to perform a specified number of iterations, with the following operations within each iteration.

If possible, a memory block of certain size is allocated.

Life time, i.e., the number of total iterations is set for the allocated block.

If there are blocks with expired lifetime, they are deallocated.

The life time of all the remaining blocks is reduced.

The size of the allocated block is either constant and predetermined, or is selected randomly from a

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

predefined range. Block lifetime is also defined randomly from a predefined range.

In order to check correctness of the allocator running from the point of view of data safety, a certain value with the same byte size is chosen in allocated blocks, and the allocated area is populated with this value. Before deleting the allocated area, the contents of the area will be checked for mismatches with the recorded value. If at least one mismatch is detected, it means that the data in the allocated area was overwritten during allocator execution, and the algorithm of the allocator contains an error.

The maximum number of blocks, maximum block size, maximum lifetime of the block and the number of iterations are to be set before the test. Also, the address of the end of the data segment (sbrk(0)) before using the allocator, and the timestamp retrieved by the clock() function, are memorized prior to the main cycle.

After all iterations of the main cycle, a certain number of blocks remain in the program memory. Based on these blocks, efficiency of memory usage is calculated, namely, the total size of occupied area to the memory of the allocator rate is calculated. The running time is defined as the difference between the timestamps obtained before the main cycle, and after its completion.

Next, all remaining occupied blocks are deallocated, and allocator memory is checked again for returning the deallocated memory to the operating system.

For each occupied block, the following structure is used:

struct Elem{

char *addr; a pointer to the beginning of the block allocated.

int size; size of the block allocated.

int live; the lifetime of the block allocated.

charval; the value written to the block allocated.

}

With such an algorithm of testing, the system is, after a certain number of iterations, in "equilibrium state", the number of allocated blocks for each iteration is approximately the same. Therefore, increasing the number of iterations with the same values of the maximum number of allocated blocks, of the maximum lifetime and the maximum size leads only to an increasing external fragmentation, but not to increasing the number of occupied blocks and the memory used by them. A sufficiently large number of iterations imitates operation of continuously running programs for a long time, where the problem of fragmentation after dynamic memory allocation is most important.

It should be noted that this test program makes it possible to obtain only an approximate idea about the allocator being tested, but cannot guarantee the same behavior of the allocator in case of specific tasks on specific hardware platforms.

5. A TEST WITH ALLOCATION OF CONSTANT SIZE BLOCKS (X86)

After tests with allocation of random size blocks, allocators were tested for allocating blocks of constant size. The need for such test can be explained by the following factors:

When allocating blocks of the same size, methods of searching for a free block (the methods of the first suitable, the best suitable, etc.) become equally valuable, since any free block will always be the best match. Viewing the entire list in order to find the best suitable one would be an unnecessary operation in this case.

There is no need to store the size of each block, as they all are of the same size. This factor greatly influences the efficiency of memory usage in case of many free and occupied blocks.

The need of garbage collection is partially eliminated, since when a new block is added, the free parts are scanned from the beginning of the area of allocator allocation, and the allocated blocks will be grouped in the beginning of this area, which is done by the garbage collection operation itself.

Disadvantages of allocators in allocating fixed size blocks are most evident in case of relatively small blocks, since the size of service data may be too large, compared to the size of the blocks.

Testing was performed on the x86 architecture, standard version of the glibc 2.13 library. Input data for tests:

number of iterations: 50,000

the maximum number of allocated blocks: 5,000

the maximum lifetime of an allocated block: 5,000

the constant size of an allocated block: 32 bytes

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

The results of testing a standard library allocator (malloc) and an allocator of blocks with the same size (osalloc) are shown in tables 1 and 2 respectively.

Table 1. The results of testing the malloc allocator

Test No.	Run time (msec)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency, (%)	Area size after deallocation (bytes)
1	8,080	2,476	79,232	106,496	74.40	106,496
2	8,160	2,512	80,384	106,496	75.48	106,496
3	7,350	2,462	79,424	106,496	74.58	106,496
4	8,190	2,506	80,192	106,496	75.30	106,496
5	8,080	2,483	79,456	106,496	74.61	106,496
Total:	**7,972**	**2,488**	**79,738**	**106,496**	**74.87**	**106,496**

Table 2. Results of testing the osalloc allocator (32 bytes block size)

Test No.	Run time (msec)	Number of blocks	Block size (bytes)	Area size (bytes)	Efficiency. (%)	Area size after deallocation (bytes)
1	7,380	2,473	79,136	86,016	92.00	81,920
2	7,220	2,488	79,616	86,016	92.56	81,920
3	7,170	2,517	80,544	90,112	89.38	86,016
4	7,150	2,516	80,512	90,112	89.35	86,016
5	7,200	2,489	79,648	90,112	88.39	86,016
Total:	**7,224**	**2,497**	**79,891**	**88,474**	**90.34**	**84,378**

6. CONCLUSIONS

The test results show that standard library allocators feature rather low efficiency of memory usage. These values are explained by the fact that for every allocated memory block, the size of service data is quite large.

The allocator for blocks with the same size showed better results, due to the use of bit matrices for managing free and occupied blocks. It also showed better runtime results. However, the disadvantage of this allocator is incomplete memory return to the operating system after all occupied blocks are deallocated.

The developed allocator was tested using the testing program [6]. Test results showed that

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

the developed allocator can be competitive with the memory manager of the standard C language library, and advantages of the allocator are manifested in various tasks. Test results also showed that the performance and the efficiency of memory usage by allocators depend not only on their algorithms, but on many other factors as well, such as the architecture of the computing system, size of the blocks allocated, duration of memory allocation and deallocation operations. Therefore, it is wrong to categorically state that one allocator is better than the other. In choosing an allocator for a specific task, it is necessary to perform a number of tests and to identify the most suitable one.

The described allocator is not ideal for all tasks of dynamic memory allocation, and it may be improved for completing certain tasks due to minor changes in the algorithm, or optimizing the code for a specific architecture. Besides, with minor changes in the source code, the proposed allocator can be used not only in UNIX operating systems, but in other platforms as well: both software and hardware-based ones.

The tests have proven practicability of using special allocators intended for allocating blocks of constant size. In the future, it is advisable to study the dependence between the memory usage efficiency and the execution time for different hardware platforms, with appropriate testing of the proposed allocator.

7. AFTERWORD

The results of the proposed study may be useful for resolving the following common practical problems:

- selecting the most appropriate allocator for solving a mixture of problems known in advance, within a certain operating system;
- assess performance of the allocator using the proposed method with regard to the parameters of hardware and software used in the operating system;
- defining the optimal block size and container size when memory is allocated for a specific task;
- assessing efficiency of memory usage in multi-threaded mode when large amounts of data are processed in SMP systems.

The main characteristics of the allocators (query execution time and efficiency of memory usage) are largely determined by the size of the allocated block; therefore, in order to choose a specific allocator, one should study the dependencies of these characteristics on the size of the block. In addition, since all listed results were obtained in course of developing allocators for the x86 architecture, it is advisable to perform similar studies for the 64-bit architecture that is widely used at the moment.

REFERENCES:

[1] Robachevsky, A.M., Nemnugin, S.A., &Stesik, O.L. (2008). *UNIX operating system* (2nd ed., revised and amended, pp. 656). Saint Petersburg: BHV - Petersburg.

[2] Tannenbaum, E. (2010). *Modern operating systems* (3rd ed, pp. 1120). Saint Petersburg: Piter.

[3] Love, R. (2008). *Linux. System programming* (pp. 416). Saint Petersburg: Piter.

[4] Glass, G., &Ables, K. (2004). *UNIX for Programmers and Users* (3rd ed., revised and amended, pp. 848). Saint Petersburg: BHV – Petersburg.

[5] Irtegov, D. (2008). *Introduction to Operating Systems* (2nd ed., pp. 1040). Saint Petersburg: BHV - Petersburg.

[6] Knuth, D.E. (2005). *The art of programming, volume 1. Main algorithms* (3rd ed.: Translation from English, pp. 720). Moscow: "Williams" Publishing House.

[7] Lee, D. (2011, April 10). A Memory Allocator. Doug Lee's Home Page. Retrieved from http://gee.cs.oswego.edu/dl/html/malloc.html.

[8] The GNU C Library. Virtual Memory Allocation and Paging. The GNU Operating System (2011, April 22). Retrieved from http://www.gnu.org/software/libc/manual/html_node/Memory.html

[9] GNU C Library Source Code. The GNU Operating System (2011, May 3). Retrieved from http://ftp.gnu.org/gnu/glibc/glibc-2.13.tar.gz.

[10] Tim Jones, M. (2011, May 3). Anatomy of Linux dynamic libraries. IBM developer Works

[11] Retrieved from http://www.ibm.com/developerworks/ru/library/l-dynamic-libraries/index.html.

[12] Mamagkakis, S., Baloukas, Ch., Atienza, D., Catthoor, F., Soudris, D., &Thanailakis, A. (2006, August). Reducing memory fragmentation in network applications with dynamic memory allocators optimized for

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

performance.*Computer Communications*, *Volume 29, 13–14*, 2612-2620.http://dx.doi.org/10.1016/j.comcom.2006.01.031

[13] Risco-Martín, J.L., Manuel Colmenar, J., Atienza, D., & Ignacio Hidalgo, J. (2011, November).

[14] Simulation of high-performance memory allocators. *Microprocessors and Microsystems*, *Volume 35, 8*, 755-765.

[15] Hasan, Y., & Chang, M. (2005, April). A study of best-fit memory allocators. *Computer Languages, Systems & Structures*, *Volume 31, 1*, 35-48.http://dx.doi.org/10.1016/j.cl.2004.06.001

[16] Risco-Martín, J.L., Manuel Colmenar, J., Ignacio Hidalgo, J., Lanchares, J., &Díaz, J. (2014). A methodology to automatically optimize dynamic memory managers applying grammatical evolution. *Journal of Systems and Software*, *91*, 109-123.http://dx.doi.org/10.1016/j.jss.2013.12.044

[17] Rezaei, M., &Kavi, K.M. (2006). Intelligent memory manager: Reducing cache pollution due to memory management functions. *Journal of Systems Architecture*, *Volume 52, 1*, 41-5.http://dx.doi.org/10.1016/j.sysarc.2005.02.004

[18] Risco-Martín, J.L., Manuel Colmenar, J., Atienza, D., & Ignacio Hidalgo, J. (2011, November). Simulation of high-performance memory allocators. *Microprocessors and Microsystems*, *Volume 35, 8*, 755-765.

[19] Masmano, M., Ripoll, I., Real, J., Crespo1, A., &Wellings, A.J. (August 2008). Implementation of a constant-time dynamic storage allocator. *Software: Practice and Experience, Volume 38, 10,* 995–1026. DOI: 10.1002/spe.858

[20] Masmano, M., Ripoll, I., Balbastre, P., & Crespo, A. (2007). *A constant-time dynamic storage allocator for real-time systems.* ISSN: 09226443 DOI: 10.1007/s11241-008-9052-7