

## The improvement of the *buddy system*

SEBASTIAN SEREWA

Institute of Informatics  
Technical University of Lublin  
ul. Nadbystrzycka 36 b, 20-618 Lublin

---

*Received 20 January 2006, Revised 14 August 2006, Accepted 11 September 2006*

**Abstract:** In the following article the author is going to summarize and show possible enhancements of the buddy system algorithm. This scheme of dynamic memory control is usually embedded in the memory management unit, which is a part of the most widely used modern operating systems. A description of possible cooperation with an operating system, as a way of the overall algorithm improvement was described. An example of the research was presented as one of possible ways of improving the algorithm.

**Keywords:** dynamic memory management algorithms, buddy system, MMU, operating systems

### 1. Introduction

In recent years there is a noticeable rapid growth of interest in the operating systems field. The most plausible reason for this trend seems to be the rising number of operating systems types being accessible for a wide mass of people. The second reason might come from the growing interest in the embedded and real time operating systems.

Although very efficient hardware memory management algorithms have been developing [13], it is still profitable to deal with their software counterparts, as an alternative method of systems performance improvement. All these factors encourage system architects and designers to seek for more efficient and flexible solutions of the software memory management.

### 2. The buddy system

The buddy system is one of the most popular memory managing systems used in the MMUs. Thanks to its simple structure, flexibility, cohesion, ability to cooperate

easily with paging system and further extensions, it plays a major role in a significant number of contemporary operating systems. Moreover, it is still being worked upon [3].

The basic reason of its functionality lies in dividing linear memory into memory areas, so called chunks or simply blocks (in literature sometimes considered as buckets). Each chunk represents a certain size of linear, continuous memory pool, which has a size equal to 2 to power of  $n$ . In most cases the size of blocks varies between  $2^n$  and  $2^m$ , where:  $n \geq 3$  due to the need of reservation of some administrative data in a chunk, and  $m \leq 31$  assuming that  $2^{32}-1$  is the greatest address which might be accessed on the given hardware architecture.

Having split the whole managed linear memory area into the fixed sized chunks; it is easy to issue memory pools requested by operating system. Unfortunately, such an approach has its drawbacks.

### 2.1. Disadvantages of the buddy system

The major and the most harmful feature of this memory system in its plain form is the internal fragmentation problem. Let's examine a memory request of 515 bytes. The system, after rounding the requested size up, will seek for the one with the size of 1024 bytes, as it is the first area which fulfills the program expectations regarding the size. In consequence, such an approach gives a waste of 509 bytes. This brings the first possibility for improvements described in point 2.2.1.

Another feature of the system which is worth investigating is its performance. Splitting and merging adjacent areas is a recurrent operation and thus very unpredictable and inefficient. This issue is investigated in point 2.2.2. The lack of functions (in programmer's interface) for partly freeing assigned memory is a next feature which should be examined [10].

In the following points the author summarizes and proposes a few modifications to the standard buddy control system.

### 2.2 Possible improvements

In overloaded operating systems (especially during the system bootstrap) the activities such as dividing and merging freed memory portions inevitably occupy the time of working algorithm. The separation of actions: resources maintenance, assignment and freeing memory brings the flexibility and performance improvement to the algorithm (this idea extends „lazy buddy system” presented in [10]) and is depicted in point 2.2.2.

The improvement of the algorithm may be split into 3 areas. The first approach bases on development of more efficient resources processing (loops unrolling, flexible data structures, resources management [5][11]). The second deals with development of more efficient data flow. Introducing parallel data processing it is possible to increase

overall system throughput. And the last one, bases on gathered statistics during previous period of time (initial congestions evading presented in point 2.2.3).

Depending on the type of a system and its requirements regarding the worst or average time of memory operation (allocating and freeing), task priorities should be set very carefully. This should be done with deliberation, especially in case of real time operating systems.

### **2.2.1 Avoiding internal fragmentation**

Among others the plain form of buddy system has one disadvantage – it is the internal fragmentation problem. The size of memory returned to the requested programs is usually rounded to the closest size of the smallest available chunk. Let's consider a request of 260 bytes from a set of free blocks. Such an action will cause using a 512 byte chunk as it is the first which is big enough to cover demanded amount of memory.

A solution for this inconvenience might lie in shifting a remnant part of a block into another task for processing. The part of the system responsible for dividing would break this into smaller portions and, when necessary, add it to the rest of managed resources. Such a proceeding would lead to time minimizing in which context of called function is occupied. Another benefit comes from avoiding internal fragmentation problem.

### **2.2.2 Actions separation**

The next feature of this memory management scheme is its negative recurrent behavior in case of alternative requests (memory demand and release). When a program releases memory, the original buddy system tries to link the recently freed area with its twin memory with the same size. Then, in recurrence a couple of two adjacent areas are linked and whole linking operation is repeated, if feasible. The meantime memory request may lead to reverse behavior, which in consequence, decreases the overall system performance.

A few implementations of memory management module (e.g. on the Linux) try to avoid this problem. It has been usually done by maintaining an additional list of just released memory chunks, without coalascing. Such an attitude for memory releasing is not always acceptable. Due to the lack of memory pools of desired size the MMU may be blocked for unpredictable amount of time.

In multi-threaded operating systems, separation of the following actions: twins linking, dividing and resources management into 3 independent threads may bring additional benefits. Picture 1 depicts the idea.

Let's deal with the system behavior with released area. Firstly, a chunk of freed memory should be split into 2 to power of n size pieces and saved into a linking queue. Then, if necessary (or simply there is no better activity to do) such a queue would be exported to the main thread which maintains all resources.

Similarly, on the memory request the main thread will return the proper chunk address to the caller as fast as possible and the remained memory is processed (divided into smaller pieces) in a separate thread.

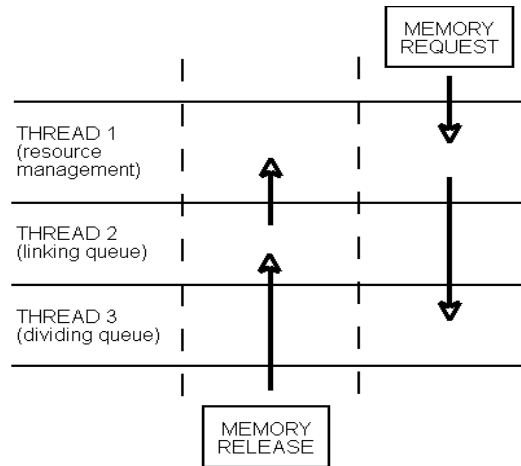


Fig. 1. Data flow.

Synchronization of data flow and resource access might be assured by binary semaphores [12]. The choice of proper IPC model for given architecture, may profit in overall performance increase.

### 2.2.3 Better preliminary classification

At the system initialization stage the whole linear memory area which is managed by buddy system is divided into the portions. The size of an average memory block allocated by operating system usually changes with time. Making the number of preliminary divided chunks dependent on the most frequently utilized memory size, may cause additional benefits. It is particularly helpful in systems with limited launching time. Such an operation might base on previously gathered statistics.

### 2.2.4 Data management and processing improvement

Using an appropriate algorithm for data processing is vital for most programs. In case of presented system at least few points are crucial. The most often used procedures such as counting down the number of zero bits, might be swapped with their rolled out counterparts [5].

Another example of more efficient data processing comes from employing proper structures for data maintenance. Instead of simple queues manipulation, the use of a binary trees should be considered [2]. The balanced binary tree search has  $O(\log(n))$

complexity, while a queue search has  $O(n)$ . Such a change may bring significant boost to data mining. As far as the buddy system is concerned, such an approach to data processing would allow programs not only to more powerful information exploring, but also would provide more deterministic nature.

### 3. Research

To conduct experiments, a simplified memory management module had to be prepared. As a controlling scheme, the buddy system algorithm was chosen.

The testing environment had been written in C, and compiled with gnu c compiler in version 3.3.3. For profiling purposes, gnu gprof 2.15.94 tool had been used. The testing program was launched on the IA32 Pentium II 350 Mhz.

The main loop of the program which emulates the operating system behaviour is presented on listing 1.

```
[...]
uIterator= 0x0;
memset( tab, 0x0, sizeof( tab ) );
while ( uIterator < 1000*MAXV ) /* MAXV == 1000 */
{
    uIndex= rand( ) % MAXV;
    if ( NULL == tab[ uIndex ] )
    {
        uRand= rand( ) % 65535;
        /* The buddy system memory allocation */
        tab[ uIndex ]= MMM_BSAllocMemory( io_pSliceHeap, uRand );
        if ( NULL == tab[ uIndex ] )
        {
            uErrors++;
        }
    }
    else
    {
        /* The buddy system memory deallocation */
        if ( 0x0 != MMM_BSDeAllocMemory( io_pSliceHeap, tab[ uIndex ] ) )
        {
            printf( "Error!!!\n" );
            break;
        }
        tab[ uIndex ]= NULL;
    }
    uIterator ++;
}
[...]
```

Listing 1 Simple emulation program.

The program had been prepared for testing in two versions. The difference is shown on the picture 2.

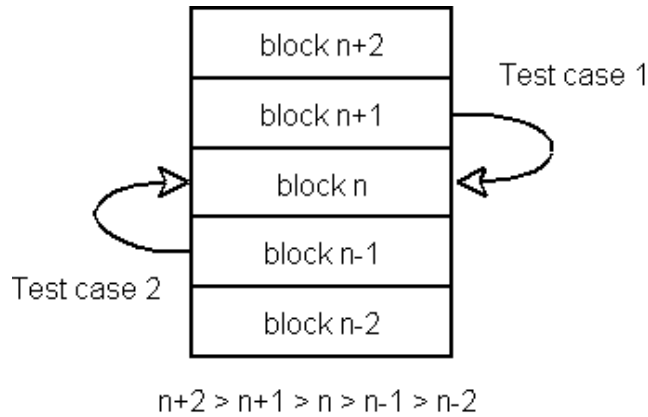


Fig. 2. Two test cases.

The first case used modified allocation routines (`MMM_BSAAllocMemory`), which in case of lack of free blocks, tried to divide bigger chunk into two smaller pieces. The second test case instead of splitting, tried to merge two smaller chunks of memory into a bigger one.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	self calls	self s/call	total s/call	name
79.98	472.67	472.67	11503124	0.00	0.00	<b>MMM_AddChunk2BitBucket</b>
19.05	585.26	112.59	426	0.26	0.80	<b>MMM_BSMergeMemory</b>
0.36	587.38	2.12	11497604	0.00	0.00	MMM_RemoveTheChunkFromBitBucket
0.30	589.18	1.80	5999011	0.00	0.00	MMM_CountDownZerosRight
0.12	589.88	0.70	999092	0.00	0.00	MMM_ArrangeMemIntoBitBuckets
0.07	590.30	0.42	1	0.42	590.90	MMM_TestBS
0.04	590.52	0.21	499759	0.00	0.00	MMM_BSDeAllocMemory
0.03	590.68	0.17	500241	0.00	0.00	MMM_BSAAllocMemory
0.02	590.82	0.14	500241	0.00	0.00	MMM_CountDownZerosLeft
0.01	590.88	0.07	503862	0.00	0.00	MMM_RemoveAnyChunkFromBitBucket

[...]

Listing 2 Flat profile of test case 1

In presented solution two direction queues were used as a data storing algorithm. Each queue represented a set of blocks of particular size. An item in a queue represents one chunk. Both test cases had the same working conditions. The program part presented on listing 1 was iterated through 106 loops. The same experiment was con-

ducted in test case 2. On listing 2, „flat profile” of gprof output is presented – test case 1. On listing 3, „flat profile” of profiler output is presented – test case 2.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
72.87	265.29	265.29	11493826	0.00	0.00	<b>MMM_AddChunk2BitBucket</b>
25.76	359.09	93.80	7465	0.01	0.03	<b>MMM_BSMergeMemory</b>
0.47	360.81	1.72	11490198	0.00	0.00	MMM_RemoveTheChunkFromBitBucket
0.46	362.48	1.67	5998205	0.00	0.00	MMM_CountDownZerosRight
0.20	363.21	0.73	999166	0.00	0.00	MMM_ArrangeMemIntoBitBuckets
0.10	363.56	0.35	1	0.35	364.01	MMM_TestBS
0.05	363.73	0.17	500258	0.00	0.00	MMM_BSAllocMemory
0.03	363.85	0.12	499742	0.00	0.00	MMM_BSDeAllocMemory
0.03	363.95	0.10	500258	0.00	0.00	MMM_CountDownZerosLeft
0.01	363.98	0.03	500692	0.00	0.00	MMM_RemoveAnyChunkFromBitBucket
[...]						

Listing 3 Flat profile of test case 2

Having analyzed all profiler outputs, it can be easily noticed that in two cases, two the most time consuming functions were `MMM_AddChunk2BitBucket` and `MMM_BSMergeMemory`. The first is responsible for adding particular chunk of memory to the given queue and the second tries to coalesce as much memory as possible from given queue.

The test case 2 overall working time was about 39% shorter in comparison to the test case 1. This is due to the fact of evenly distribution of free chunks in queues in the test case 2.

#### 4. Conclusions

There are many ways of improvement of the buddy system algorithm. The presented approach bases on more efficient structures manipulation and cooperation with the operating system.

The advancement of data processing with use of balanced binary trees was shown, as an alternative to simple queue management. The idea of dispersion of chunks modifications and its possible effects on the running system have been presented. The most harmful features of the buddy system and a few possible solutions for them were also shown. The results gathered during the tests may be used for further improvement and as a starting point for comparison.

The presented paper might be a base for further research and enhancement of the software version of buddy system memory management scheme.

## References

- [1] Bach J. Maurice: *Budowa systemu operacyjnego UNIX*, Wydawnictwo Naukowo Techniczne Warszawa, 1995 (in Polish).
- [2] Cormen H. Thomas, Leiserson E. Charles, Rivest L. Ronald, Stein Clifford: *Wprowadzenie do algorytmow*, Wydawnictwo Naukowo Techniczne, Warszawa, 2003 (in Polish).
- [3] Defoe DC, Cholleti SR, Cytron RK: *Upper bound for defragmenting buddy heaps*, Assoc Computing Machinery, New York, p. 222-229, 2005.
- [4] Goczyński Ryszard, Tuszyński Michał: *Mikroprocesory 80286, 80386 i i486*, Komputerowa Oficyna Wydawnicza HELP, Warszawa, 1991 (in Polish).
- [5] Heatfield Richard, Kirby Lawrence: *C Unleashed*, Sams Publishing Indianapolis, 2000.
- [6] Herlihy M, Luchangco V, Martin P, Moir M: *Nonblocking memory management support for dynamic-sized data structures*, Assoc Computing Machinery, New York, p. 146-196, 2005.
- [7] Knuth E. Donald: *Art of computer programming*, Addison-Wesley Publishing Company, 1981.
- [8] Lo CTD, Srisa-An W, Chang JM: *Performance analyses on the generalized buddy system*, Iee-Inst Elec Eng, Hertford, p. 167-175, 2001.
- [9] Lo CTD, Srisa-An W, Chang JM: *The design and analysis of a quantitative simulator for dynamic memory management*, Elsevier Science Inc, New York, p. 443-453, 2004.
- [10] Vahalia Uresh: *Jadro systemu UNIX*, Wydawnictwo Naukowo Techniczne 2001 (in Polish).
- [11] Rene Alexander, Graham Bensley: *Optymalizacja oprogramowania*, Wydawnictwo RM, 2001 (in Polish).
- [12] Silberschatz Abraham, Galvin Peter B.: *Podstawy systemów operacyjnych*, Wydawnictwo Naukowo Techniczne, 2000 (in Polish).
- [13] Witawas Srisa-an, Chia-Tien, Dan Lo. J. Morris Chang: *Object resizing and reclamation through the use of hardware bit-maps*, Department of Computer Science, Illinois Institute of Technology, 2001.

### Udoskonalony system bliźniaków

#### Streszczenie

W niniejszym artykule autor przedstawił oraz podsumował możliwe usprawnienia dynamicznego systemu zarządzania pamięcią, jakim jest **system bliźniaków**. System ten jest najczęściej wbudowany w moduł zarządzania pamięcią, który jest częścią nowoczesnych systemów operacyjnych. Zaprezentowano opis możliwej współpracy z systemem operacyjnym jako sposób poprawy ogólnej wydajności algorytmu. Przykładowe badanie oraz sposób przeprowadzenia został przedstawiony jako jedna z możliwych dróg nad dalszymi usprawnieniami tego algorytmu.