# A NEW IMPLEMENTATION TECHNIQUE FOR BUDDY SYSTEM

**Nada A.Z. Abdullah, * Manal F. Younis**
Department of Computer Science, College of Science, University of Baghdad. Baghdad-Iraq
*Department of computer Engineering , College of Engineering, University of Baghdad. Baghdad-Iraq

**Abstract**
Buddy system algorithm is dynamic memory control which is usually embedded in the memory management unit, which is a part of the most widely use modern operating systems. Dynamic memory management is an important and essential part of computer systems design. Efficient memory allocation, garbage collection and compaction are becoming increasingly more critical in parallel, distributed and real-time applications using object-oriented languages like C++ and Java. In this paper we present a technique that uses a Binary tree for the list of available memory blocks and show how this method can manage memory more efficiently and facilitate easy implementation of well known garbage collection techniques.

## تقنية جديدة لتنفيذ نظام التبرعم

**ندا عبد الزهرة عبد الله ، *منال فاضل يونس**
قسم علوم الحاسبات، كلية العلوم، جامعة بغداد. بغداد– العراق
*قسم هندسة الحاسبات، كلية الهندسة ، جامعة بغداد. بغداد– العراق

**الخلاصة**
خوارزمية البرعم هي طريقة لتنظيم ذاكرة ديناميكية والتي عادة ما تكون جزء من وحدة ادارة الذاكرة فـــي معظم نظم التشغيل الحديثة. ادارة الذاكرة الداينميكية هي جزء مهم وضروري في تصـــميم انظمـــة الحاســـبة. التخصيص الكفوء للذاكرة و جمع الاجزاء الصغيرة وضغطها اصبح من الامور الضـــرورية والحرجـــة فـــي الانظمة المتوازية والتوزيعية وتطبيقات الانظمة الحقيقة باستخدام لغات البرمجة الكيانية مثل لغة C++ و Java . في هذا البحث تم اقتراح تقنية استخدام (binary tree) للاجزاء الفارغة من الذاكرة وتم اقتراح خوارزميـــات التعامل معها كون (binary tree) اكثر كفاءة في الاستخدام والتنفيذ.

## 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, 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 [1].

Dynamic memory allocation is a classic problem in computer systems. Typically we start with a large block of memory (sometimes called a heap).

When a user process needs memory, the request is granted by carving a piece out of the large block of memory. The user process may free some of the allocated memory explicitly, or the system will reclaim the memory when the process terminates. At any time the large memory block is split into smaller blocks (or chunks), some of which are allocated to a process (live memory), some are freed (available for future allocations), and some are no-longer used by the process but are not available for allocation (garbage). A memory management system must keep track of these three types of memory blocks and attempt to efficiently satisfy as many of the process's requests for memory as possible [2].

The buddy system is one of the most popular memory managing systems use in the memory management systems. The basic reason of its functionality lies in dividing linear memory into memory areas, so called chunks or simply blocks. 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 2n and 2m, where: n >=3 due to the need of reservation of some administrative data in a chunk, and m <=31 assuming that 232-1 is the greatest address which might be accessed on the given hardware architecture [1].

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.

The (binary) buddy system much faster than other heuristics for dynamic memory allocation, such as first-fit and best-fit. Its only disadvantage being that blocks must be powers of two in size, the buddy system is used in many modern operating systems; in particular most versions of UNIX/Linux, for small block sizes. Various implementations have their own particular twists. For example, most versions of Linux try to avoid the potential amortizedO(log n) cost of allocating and deallocating small blocks, by keeping deallocated small blocks on lists. While usually quite effective, this practice can lead to the inability to allocate a large block even when all of memory is free. Because our work is based on the standard buddy system, we review the basic ideas now. At any point in time, the memory consists of a collection of blocks of consecutive memory, each of which is a power of two in size. Each block is marked either occupied or free, depending on whether it is allocated to the user. For each block we also know its size (or the logarithm of its size). The system provides two operations for supporting dynamic memory allocation [3]:

1. Allocate (2k): Finds a free block of size 2k, marks it as occupied, and returns a pointer to it.
2. Deallocate (B): Marks the previously allocated block B as free and may merge it with others to form a larger free block.

## 2. 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. The technique adopted by Linux to solve the external fragmentation problem is based on the well-known buddy system algorithm. All free page frames are grouped into 10 lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 contiguous page frames, respectively. The physical address of the first page frame of a block is a multiple of the group size—for example, the initial address of a 16-page frame block is a multiple of 16 x $2^{12}$ ($2^{12}$ = 4,096, which is the regular page size) [4]. 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 an inefficient [1]. In this paper an improvement is proposed to solve performance problem using more efficient data structure (binary tree).

## 3. Related work

Several other buddy systems have been proposed, which are briefly survey now. Knuth [5] proposed the use of Fibonacci numbers as block sizes instead of powers of two, resulting in the Fibonacci buddy system. This idea was detailed by Hirschberg [6], and was optimized by Hinds [7] and Cranston and Thomas [8] to locate buddies in time similar to the binary buddy system. Both the binary and Fibonacci buddy systems are special cases of a generalization proposed by Burton [9]. Shen and Peterson [10] proposed the weighted buddy system which allows blocks of sizes $2^k$ and $3{\cdot}2^k$ for all k. All of the above schemes are special cases of the generalization proposed by Peterson and Norman [11] and a further generalization proposed by Russell [12]. Page and Hagins [13] proposed an improvement to the weighted buddy system, called the dual buddy system, which reduces the amount of fragmentation to nearly that of the binary buddy system. Seeger B. and Kriegel H. proposed a new multidimensional access method, called the buddy-tree, to support point as well as spatial data in a dynamic environment. The buddy-tree can be seen as a compromise of the R-tree and the grid file, but it is fundamentally different from each of them. Because grid files loose performance for highly correlated data, the buddy-tree is designed to organize such data very efficiently, partitioning only such parts of the data space which contain data and not partitioning empty data space [14]. Brodal G. S., Erik D. D., Munro J.I.were proposed several modifications to the binary buddy system for managing dynamic allocation of memory blocks whose sizes are powers of two. The standard buddy system allocates and deallocates blocks in $\Theta(\log n)$ time in the worst case (and on an amortized basis), where n is the size of the memory. We present three schemes that improve the running time to O(1) time, where the time bound for deallocation is amortized for the first two schemes. The first scheme uses just one more word of memory than the standard buddy system, but may result in greater fragmentation than necessary. The second and third schemes have essentially the same fragmentation as the standard buddy system, and use $O(2(1+\sqrt{\log n})\log\log n)$ bits of auxiliary storage, which is $\omega(\log^k n)$ but $o(n^{\varepsilon})$ for all $k \geq 1$ and $\varepsilon > 0$. The rest of this paper is organized as follows; section 4 contains Buddy System Data Structure. Section 5 the proposed data structure. Section 6 Complexity analysis. Section 7 simulation. Finally conclusions are listed in section 8.

## 4. Buddy System Data Structure

The buddy system maintains a list of the free blocks of each size (called a free list), so that it is easy to find a block of the desired size, if one is available [3].(Figure 1). illustrates the use of the data structures introduced by the buddy system algorithm. The array zone_mem_map contains nine free page frames grouped in one block of one (a single page frame) at the top and two blocks of four further down. The double arrows denote doubly linked circular lists implemented by the free_list field. Notice that the bitmaps are not drawn to scale [4].

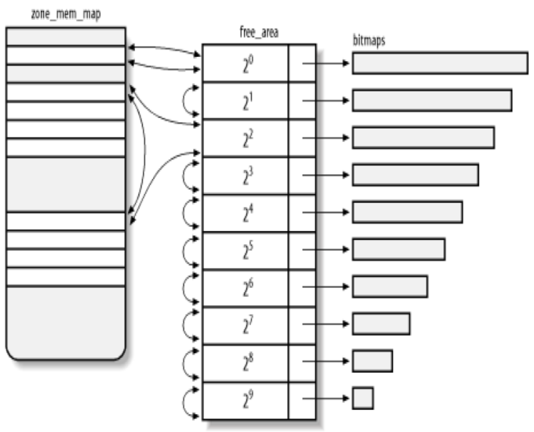

**Figure 1: Data structures used by the buddy system [4]**

## 5. The proposed data structure

As have been seen in section 2 the buddy algorithm has a low performance due to the type of data structure used (queue or linked lists). An improvement of the algorithm to use an efficient data structure. Efficient data structure means faster than linked list in searching or traversing. The binary tree is proposed in this paper to be used in the buddy system. In (figure 2). the new data structure is shown. In our approach every free block will be defined as node_struct of type struct in C language:

```
Typedef Struct node_struct {
Int size;
Int block_no;
node_struct *left;
node_struct *right;
node_struct *LLptr};
```

**Figure 2: Structure of tree node**

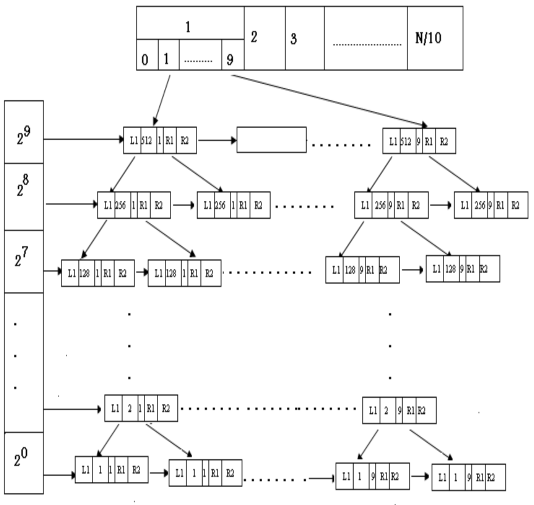

**Figure 3: Data structure of the binary buddy system**

All nodes of the same size are linked in a linklist of its size and the array of size [11] store 10 pointers every pointer points to a linklist, the definition of this array is as follow:

```
Arr_size * node_struct [10];
```

**Figure 4: Definition of array**

On the other side every node linked with a binary tree, at beginning every 512 size free block is the root of binary tree which can be divided into two nodes of size 256. Therefore every node must has left and right pointers to represents the binary tree, and has side pointer to link the node with a link list of its size. Another array has been defined to store all the roots pointers. The array root-arr of size (N) where N represents number of blocks of size 512 in the memory.

```
Tree1 *aa[10];
Tree1 *root_arr[sizeof(aa[10]);
```

**Figure 5. Definition of the root node**

Any allocation will be start from the size_arr, while every deallocation will be start from root_arr.

## 5.1. Allocation

The proposed allocation is the same allocation as classic buddy system (first-fit), with additional work due to the new data structure used (binary tree). The allocation algorithm is shown below.

```
Allocation algorithm (Allocate a free node to
    a process and release it from the
    free_blocks list)
•   If a request to free block of size R
    continuous page frame.
      The algorithm check first whether a free
        block of size R exists, by checking the
          link list of this size which arr_size
                              [log2(R)] points to
If found Then
        1) Remove the head of the link list
           of size R.
        2) Mainpulate the binary tree.
Else
Looks for the next large block
If found
1- Allocate block of size R to the request
2- Dividing the remaining into blocks of
   size 2 to power X where X € [1,2,…..,8]
3- Insert the divided blocked at the head of
   the link list of its size.
4- Link every free block to its appropriate
   place in its binary tree.
```

## 5.2. Deallocation

The reverse operation to allocation, releasing blocks of page frames, gives rise to the name of this algorithm. In classic buddy system the kernel attempts to merge together pairs of free buddy blocks of size b into a single block of size 2b. Two blocks are considered buddy if:

- Both blocks have the same size, say b.
- They are located in contiguous physical addresses.
- The physical address of the first page frame of the block is a multiple of 2 X b X $2^{12}$.

The algorithm is iterative, if it succeeds in merging released blocks; it doubles b and tries again so as to create even bigger blocks. The deallocation in buddy system needs searching the links list to create larger free chunk.

In proposed deallocation algorithm according to the address of free chunk, the search begins from the root of the binary tree of the chunk to find the appropriate place or merge this chunk with its neighbor.

```
Deallocation algorithm (insert new free memory chunk of size N and the address D)

1- Find the chunk's tree (D/512 * page size)
Gives the block number
Root_arr [block number/10][block number %10] points to the root of chunk's tree.
2- If not found
2.1.    This chunk will be the root
2.2.    Set left and right pointers to null
2.3.    According to its size insert this free
     chunk as a head of link – list of its size.
3. If found
3.1.    Compare the size of root with new chunk
    3.1.1. If equals then
      Combined root with the new free block to
      create new block of size (2N)
     3.1.2. If (New free chunk>root) Then
       New chunk will be the root and link the
        root to left pointer of the new root.
      3.1.3. If (New chunk < root)
           Then found the appropriate place
3.1.3.1.  Set the root as the current node
3.1.3.2.  Check if the current node has left or
          right pointers
      Then Check the size of the child of the
      current node
a-   If equals then combined the new free block with the child block to create 2N free block.
b-   If (child > new free block) Then
    The new free block will be the parent of the left or right child
c-   If (left or right child <new free block) Then
              Set the child as the current node
              Goto step 3.1.3.2
3.2. According to the size of the new free chunk created insert this free chunk as a head of the link-list of its size.
```

## 6. Complexity analysis

Deallocation algorithm proposed which search the binary tree to find a appropriate place for the free chunk. Deallocation algorithm is very similar to binary tree traversal and the execution time complexity depends on the number of levels (L) in the tree, hence, the complexity of deallocation is O(L).

In allocation algorithm the search for free chunk is the same as the traditional buddy system (this search is very fast) since the free chunks are arranged in 10 linked list sorted by size and the allocation used first algorithm to determine the needed free chunk. After find the first-fit free chunk the tree must be modified. This modification requires changes to at least two pointers, and delete this chunk from tree. Allocation requests are satisfied using traditional binary search algorithms with a complexity of O(1). In our case, the search is speeded up by the Max_Left and Max_Rigth that guide the search to the appropriate sub-tree. When a node is deleted we consider the following cases.

a. If the deleted node has a right child, the right child will replace the deleted node. The left child of the deleted node (if one exists) becomes the left child of the new node. Otherwise, if the replacing node (the right child of the deleted node) has a left sub-tree, we traverse the left sub-tree and the leftmost child of the replacing node becomes the parent of the left child of the deleted node.
b. If the deleted node has no right child then the left child replaces the deleted node.

Since the case "a" above may involove traversing down the tree (from the right child of the deleted node), the worst case complexity is O(1).

## 7. Simulation

In order to evaluate the benefits of our approach to memory management, we developed simulators that accept requests for memory allocation and deallocation. In all implementations, we included deallocation objects whenever possible.

## 8. Conclusion

The buddy system is one of the most popular memory managing systems used in the memory management system. 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.

In this paper we described the use of Binary Trees for maintaining the available chunks of memory. The Binary Tree is based on the starting address of memory chunks. In addition this information is used during allocation to find a suitable chunk of memory. The Binary Tree implementation permits immediate merging of newly freed memory with other free chunks of memory. Binary Tree naturally improves the search for appropriate size blocks of memory over Linear Linked lists. Buddy system arrange the free chunk as queues and the complexity of queues and the complexity of queue search has O(n).

**References**

1. Serewa S. **2006**. The improvement of the buddy system. *Theoretical and Applied Informatics* ISSN 1896-5334, 18(2), pp. 133-134.
2. Rezaei M.and Kavi, K.M. **2000**. A New Implementation Technique for Memory Management. Proceedings of the SoutheastCon, Nashville, TN, pp.1-2
3. Brodal, G. S., Erik D. D.and Munro, J.I.**1999**. Fast Allocation and Deallocation with an Improved Buddy System. Proceedings of the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science p.2
4. Bovet, D. P. and Cesati M. **2002**.Understanding the Linux Kernel, 2nd Edition, O'Reilly: ISBN : 0-596-00213-0, pp. 257-259.
5. Donald E. K. **1968**. Dynamic storage allocation. *In The Art of Computer Programming*, Addison-Wesley1(2.5): 435–455.
6. Hirschberg,D.S**.** **1973**. A class of dynamic memory allocation algorithms. *Communications of the ACM*, 16(10):615–618, Oct.
7. Hinds,J.A. **1975**. Algorithm for locating adjacent storage blocks in the buddy system. *Communications of the ACM*. 18(4):221–222, Apr.
8. Cranston B. and Thomas, R. **1975**. A simplified recombination scheme for the Fibonacci buddy system. *Communications of the ACM*, 18(6):331–332, June.
9. Burton,W. **1976**. A buddy system variation for disk storage allocation. *Communications of the ACM*, 19(7):416–417, July.
10. Kenneth, K. S. and James, L. P. **1975**. A weighted buddy method for dynamic storage allocation. *Communications of the ACM*, 17(10):558–562, Oct.
11. James, L. P. and Theodore, A. N. **1977**. Buddy systems. *Communications of the ACM*, 20(6):421–431, June.
12. Russell,D.L. **1977**. Internal fragmentation in a class of buddy systems. *SIAM Journal on Computing*, 6(4):607–621, Dec.
13. Ivor, P. and Hagins, J. **1986**. Improving the performance of buddy systems. *IEEE Transactions on Computers*, C-35(5):441–447, May.
14. Seeger, B. and Kriegel, H. **1990**. The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems. Praktische Informatik, University of BFtEMEN, D-2800 BREMEN 33, WEST GERMANY, Proceedings of the 16th VLDB Conference Brisbane, Australia.