Space Efficient Parallel Buddy Memory Management
University of Florida CIS TR #92-008

Theodore Johnson
Tim Davis
Dept. of Computer and Information Science, University of Florida
ted@cis.ufl.edu, davis@cis.ufl.edu

April 3, 1992

Abstract

Shared memory multiprocessor systems need efficient dynamic storage allocators, both for system purposes and to support parallel programs. Memory managers are often based on the buddy system, which provides fast allocation and release. Previous parallel buddy memory managers made no attempt to coordinate the allocation, splitting and release of blocks, and as a result needlessly fragment memory. We a present fast, and simple parallel buddy memory manager that is also as space efficient as a serial buddy memory manager. We test our algorithms using memory allocation/deallocation traces collected from a parallel sparse matrix algorithm.

Keywords: Memory management, Concurrent data structure, Buddy system, Parallel algorithm.

1 Introduction

A *memory manager* accepts two kinds of operations: requests to allocate and requests to release blocks of memory, which may be of an arbitrary size. For example, the UNIX system calls `malloc()` and `free()` are requests to a memory manager. A concurrent memory manager handles requests for shared memory in a multiprogrammed uniprocessor, a shared memory multiprocessor, or a distributed shared virtual memory environment. In order to be correct, a concurrent memory manager should correctly allocate and release memory, merge adjacent free blocks, and allocate a block of memory in response to an allocate request if a sufficiently large block is available and isn't claimed by another allocate request. A simple concurrent memory manager might consist of a serial memory manager placed in a critical section. While this algorithm would be sufficient for a multiprogrammed uniprocessor, it can create a serial bottleneck in a parallel processor. Example applications of parallel memory managers are parallel sparse matrix factorization algorithms [2, 4], and buffers for message passing in clustered parallel computers[3].

Most heap memory management algorithms use one of two main methods: free lists and buddy systems. In a *free list* algorithm [12], the free blocks are linked together in a list, ordered by starting address. Several parallel free list algorithms have been proposed [16, 1, 7, 8]. Free list algorithms are much slower than buddy algorithms, which are the subject of this paper. Another concurrent memory manager [11] has been proposed, based on the *fast fits* memory manager [14, 15]. The fast fits data structure is a free list linked into a tree, which allows fast allocate and release operations. Fast fits is slower but more space efficient than buddy algorithms, and are faster but less space efficient than free list algorithms.

Gottlieb and Wilson developed concurrent buddy systems that use fetch-and-add to coordinate processors. Their first algorithm [9, 17] considers a buddy system to be organized as a tree. A count of the number of blocks of each size that are contained in subtree rooted at a node is stored at each node. Concurrent allocators use this information to navigate the tree. Their second algorithm [10, 17] is a concurrent version of the commonly described buddy algorithm. In both of these buddy algorithms, an allocate operation performs a split every time it fails to allocate a block, even if the allocate operation can be satisfied by a block that will be returned by a previously issued split. As a result, memory is needlessly fragmented. In addition, the algorithms can fail to combine neighboring buddies.

In this paper, we present a memory efficient parallel buddy memory manager. We achieve this efficiency by coordinating the allocate, release and split operations so that large block are split into small blocks only when necessary. In addition, our algorithms combine all unallocated buddies. Because the memory manager is based on the buddy system, it will execute quickly. We test our algorithms by simulating them with a trace-driven simulator, and running the simulator with traces collected from several sparse matrix factorization algorithms.

2 Serial Buddy Memory Manager

We use a binary buddy system [12, 13] to create a parallel memory manager. In a buddy system, memory blocks are available only as one of several fixed sizes. Each memory block has a buddy, with which it can combine and form a larger size block. The available sizes are determined by the choice of buddies. A popular buddy system is the binary buddy system, in which a block and its buddy are the same size. When the buddies combine, they form a buddy of the next higher size, so that the size of binary

buddy blocks is $c2^i$, where size of the smallest allocatable block is c (usually a power of 2). Figure 1 shows an example binary buddy system. Another common system is the Fibonacci buddy system, in which block sizes satisfy $F_n = F_{n-1} + F_{n-k-1}$. A Fibonacci buddy system offers more block sizes than a binary buddy system, and so has the potential for less internal fragmentation, but can cause additional external fragmentation since many uselessly small blocks are created. We describe a parallel binary buddy algorithm in this paper; parallel Fibonaci buddy algorithms can be written in a similar manner.

Each block in a buddy system, allocated or unallocated, keeps a small amount of information. All blocks must store a flag indicating whether or not the block is free, and also the logarithm of the size of the block (both can fit into a single word). Free blocks also contain two pointers to maintain a doubly linked list. The buddy system maintains a free list for each block size. We number the lists by the logarithm of their size, so that list i contains blocks of size $c2^i$. We will call the size $c2^i$ blocks *level* i blocks, and call the free list that contains them the *level* i list. Figure 2 shows the data structure that holds the buddy system shown in Figure 1.

When a process needs a block of memory of size M, it asks the buddy allocate procedure to give it a block from list i, where i is the smallest integer such that $c2^i \geq M$. The allocate procedure tries to satisfy the request by allocating the block at the head of the level i free list, if possible. If there is a block on the free list, the allocate procedure is finished. If no block is available the allocate procedure must split a level $i+1$ block. A split is accomplished by allocating a block from list $i+1$, giving half of the block to the requesting process, and putting the other half of the block on the level i free list. If there are no blocks on the level $i+1$ free list, the algorithm is applied recursively.

When a process deallocates a memory block, it calls the release procedure with the address of the block. The release procedure reads the size of the block and picks the corresponding free list, i. The release procedure first checks if the released block's buddy is free, and of the correct size (i.e, on list i). The address of a block's buddy can be calculated from the block's size and starting address [12, 13]. The release procedure can determine whether the buddy is free and of the same size by examining first word of the buddy block, which contains the free/allocated bit and the size of the buddy, so the level i free list doesn't need to be searched. If the buddy is free and can be combined with the released block, the release procedure removes the buddy from the size i free list, combines the blocks, and releases the combined block to the level $i+1$ free list. If the the buddy of the combined block is available, the algorithm is

applied recursively. Otherwise, the deallocated block is put on the head of the i^{th} free list.

A buddy system is usually described as managing a block of memory whose total size is the largest possible buddy block (of size $c2^m$). In this case, the free lists in the buddy system are all initially empty, except for the m^{th} list, which contains a single block. A buddy system can be initialized so that a block which is an arbitrary multiple of c is managed. To initialize a buddy system of size cN, put a $c2^i$ size block of memory on the i^{th} free list if and only if the i^{th} bit in $\log_2 N$ is one. An initial binary buddy system that manages 44 blocks is shown in Figure 3. We will use this method to determine the minimum amount of memory needed to successfully complete a trace.

Since the free blocks are maintained in a doubly linked list, allocate and release operations on a particular free list take only a few steps. An allocate or release on the level i free list might require a corresponding operation on the level $i+1$ free list, but a request to allocate or release a block of memory will require at most $O(\log(M))$ such operations where M is the size of the managed memory. Most requests require only a few operations, and finish quickly.

3 The Parallel Buddy Memory Manager

In order to parallelize the buddy memory manager, first observe that two different free lists can be manipulated independently, because the allocate and release operations on the level i free list are concerned only with blocks the corresponding size. An allocate or a release on the level i free list might cause the corresponding operation on the level $i+1$ free list, but the operation on the level $i+1$ free list can be performed in parallel with further operations on the level i free list. So, all free lists in the system can be used in parallel.

Suppose that a process, P, requests a block from the level i free list. If the list is empty, the process will split a level $i+1$ block, which will make 2 level i blocks available. Suppose that another process, Q, requests a block from the level i free list while P's split is being processed. Even though the list is empty, Q doesn't need to split a level $i+1$ block, because P's currently executing split operation will soon provide the needed block. Instead, Q should wait for P's split operation to give it the block which would otherwise be put on the level i free list.

Suppose that while P's split is being processed and Q is waiting, another process, R, releases a block of memory onto the level i list. If R can detect that an allocate request is waiting for the result of a

split operation, R can give the block directly to the waiting operation, waking up the waiting allocate operation early. So, R can satisfy Q's request immediately. Suppose that another release operation, S, arrives. The process P is still waiting for the result of its split. If S puts its block on the free list and P takes one of the blocks returned by the split, there will be two non-buddy blocks left on the free list. If instead, P uses the block that S releases, the split operation will put both of its blocks on the free list, where they will be promptly combined and put on the level $i+1$ free list, reducing fragmentation.

These considerations show that the parallel buddy system will be more efficient if we keep some information about pending operations in the free lists and use it to make decisions in the current operations. The data structure in our parallel buddy system is an array of free lists. Each free list contains the following fields in addition to those stored by the serial algorithm:

1. `Nrequested:` The number of blocks that have been requested (via to requests to split larger blocks)
2. `Nqueued:` The number of unfulfilled requests for memory blocks of size i.
3. `Queue:` A queue of blocked allocate requests.
4. `Lock:` A structure that allows the free list to be locked.

The three procedures in the parallel buddy algorithm, `allocate`, `release`, and `split` are listed in the appendix. The allocate procedure begins by locking the indicated free list to prevent interference from other processes. If a block is available, the allocate procedure simply returns the block at the head of the list. If the list is empty, the process adds itself to the waiting queue so that it can receive blocks from release and split operations, and increments the number of waiting allocate operations. If there are now more operations waiting for blocks that have been requested by issuing split operations, the allocate operation must issue a split operation. The allocate operation increments the number of requested blocks by two, since one split provides two blocks.

The release operation also begins by locking the free list. If there are waiting allocate operations, the release operation wakes up one of the allocate operations and gives it the block that is being released. Otherwise, the release operation performs the usual operations. The split operation first calls the allocate procedure to obtain a level i block. This block is then broken into two level $i-1$ blocks. The split procedure locks the level $i-1$ free list in order to give the blocks to the level $i-1$ list. Since the split procedure is satisfying the request for two blocks, it decrements the number of requested blocks by two.

If there are at least two waiting allocates, the split procedure gives them the blocks. If there is only one waiting allocate, the split operation wakes it up, and puts the other block on the free list. If there is no waiting allocate (the requests having already been satisfied), the combined block is returned to the level i free list.

The entries in the wait queue need to indicate how to wake up the waiting process, and how to tell the waiting process which block of memory it has been allocated. We assume that each process has a record that can be attached to the wait queue. A process will enter the 'sleep' state after it releases the lock on free list, so that a release or split procedure can wake up the process before it sleeps. Mechanisms for achieving this synchronization must be able to handle this possibility. One option for synchronizing the sleeping allocate operations is by the use of semaphores. Semaphores are likely to require too much overhead for such short waiting periods, so we provide a more efficient method: The entries in the wait queue contain two records: a variable for the waiting processor to spin-wait on, and a space for a pointer to a block of memory. When an allocate operation must wait, it sets the spin variable and puts the entry on the queue. In order to sleep, it spins on the synchronization variable until the variable is reset. When a release or split procedure wakes up the waiting process, it first makes the pointer in the queue entry point to the released block, and then resets the synchronization variable. When the waiting process breaks out of the spin lock, it uses the block of memory pointed to by the queue entry. Note that a process that makes an allocate request might wait in several queues, due to recursive split calls. Therefore, each process needs enough queue entries to put itself on the queue at every free list. This isn't a difficult requirement, though, because each entry is small, and only a logarithmic number of entries are needed. Note also that, since we didn't specify a (FIFO) ordering on the wait queue, a process might be required to wait for a block on level i after it performs a split operation on level $i+1$. This can occur if other waiting allocate requests that arrived later have a higher priority, due to real-time constraints for example.

3.1 Handling Out-of-Memory Conditions

If a process attempts to allocate a block from the highest free list, but the list is empty, then the buddy system is out of memory and can't satisfy some allocate requests. How many requests should be denied isn't immediately clear, since a split operation fetches blocks for up to two allocate requests. Not every allocate request should be denied, though, since unrelated requests for small blocks might be able to

proceed. We need a method for informing all but only those requests affected by the out-of-memory condition.

Denying the minimum number of allocate requests is difficult due to the asynchronous nature of the algorithm, since denied split requests might overtake successful split requests. However, we can bound the number of requests that are denied by a simple modification to the algorithm listed in the appendix. If a level i allocate procedure determines that its request is denied, it returns an error code to the calling procedure. If the calling procedure is a user procedure, the user can decide what actions to take (possibly asking for the block of memory again). If the calling procedure is the split procedure, then up to two level $i-1$ requests must be denied. If the level $i-1$ wait queue is empty, then the split procedure stops. Otherwise, up to two entries are removed from the wait queue, and the waiting processes are informed of the denied request by setting the block pointer to a null location. If any of the denied requests are split operations, the process continues recursively.

4 Trace-driven Simulations

In order to test the space efficiency of our algorithms, we wrote a trace-driven simulator. We gathered the traces from a sparse matrix factorization algorithm – an important application that requires significant memory management support. The traces consist of the time of the allocation request, the size of the request, and the length of time that the request block is held (possibly forever). The simulator uses this information to schedule the allocate and release requests. The expected time to perform each separate allocate, release, or split operation is parameterizable and is modeled in the simulator as an exponential distribution.

4.1 Trace Collection

The LU factorization of a sparse matrix forms an important kernel of many problems in scientific computing. A sparse matrix can be informally defined as a matrix with enough zero entries so that significant gains in computing time and storage can be achieved by taking advantage of those entries. One common data structure stores a matrix as a set of compressed sparse vectors, one for each row and/or column of the matrix [6]: a compressed vector $\mathbf{y}(1 \ldots n_{nz})$ holds the n_{nz} nonzero values of a full vector $\mathbf{x}(1 \ldots n)$, and a corresponding integer index vector $\mathbf{i}(1 \ldots n_{nz})$ holds the location of the nonzero entries of x, such that $\mathbf{y}(k) = \mathbf{x}(\mathbf{i}(k))$. A dynamic memory allocation mechanism is required is the calculation sequence is

not known in advance, since the number of nonzeros in a row or column can change as the factorization progresses. Parallel sparse matrix factorization algorithms, such as the D2 algorithm, place an even higher demand on this resource, as multiple processes compete for access to the memory manager [2].

We use the D2 algorithm as a test case for the parallel buddy memory manager. Its key feature is a non-deterministic parallel pivot search that builds a set of *independent pivots*, of size m, say, which when permuted to the diagonal form a diagonal m-by-m submatrix. Most of the numerical operations associated with these pivots can be done in parallel, with a parallel rank-m sparse update to the remaining active submatrix. This step can increase the number of nonzeros in rows and columns of the active submatrix. After the numerical factorization with this pivot set, the pivot rows and columns are fixed in size, and become the next m rows and columns of the U and L factors, respectively (with appropriate scaling). These two steps alternate until it is no longer profitable to take advantage of sparsity, at which point the active submatrix is copied from the sparse data structure into a newly-allocated dense matrix that is just large enough to hold it. The compressed rows and columns are then released.

We generated traces from the D2 algorithm for a set of matrices from the Harwell/Boeing sparse matrix test collection [5]. These matrices come from a wide range of real problems in scientific computing. The two matrices we use are gemat11, an order 4929 matrix with 33108 nonzeros, and gre-1107, an order 1107 matrix with 5664 nonzeros. These matrices were factorized on an Alliant FX/80, a shared memory computer with 8 vector processors. A trace was generated for each matrix by recording the row or column index, number of nonzero entries in the row or column, and a time stamp, whenever a row or column was numerically modified. The final entry in each trace is the single dense submatrix allocated when switching to a dense matrix factorization code.

4.2 Results

Given the traces, we simulate a greater demand on the memory manager by increasing the time to execute a single allocate, release, or split operation. We summarize our results in Table 1. For both matrices, we run a simulation of a serial buddy algorithm to compare its space utilization against that of the parallel algorithms.

We calculate the space utilization by running experiments to determine the smallest amount of memory required to complete a trace, assuming that the smallest block size is four words. We then compute the maximum amount of memory that is ever requested, and report the proportion of required

matrix	operation time	space utilization	number of allocates	allocate response time	direct releases	number of splits	undone splits
gre-1107	serial	70.0%	8984				
	.0001	70.2%	8984	.00019	175	4039	193
	.0004	70.0%		.00091	720	3850	178
	.0008	70.0%		.00283	1804	3879	192
	.0010	70.0%		.00458	2273	3831	175
	.0015	68.1%		.0274	3300	3783	142
	.0020	64.8%		.267	4586	3945	247
gemat11	serial	67.4%	11179				
	.0001	67.4%	11179	.000323	854	10297	12
	.0004	67.4%		.00158	2380	10273	8
	.0008	67.5%		.00522	5647	10264	5
	.0010	67.5%		.00889	6814	10276	8
	.0015	67.4%		.0504	8272	10271	9
	.0020	67.3%		1.64	8512	10297	12

Table 1: Performance Results. Time units are normalized.

to requested memory as the space utilization. The column labeled 'direct release' lists the number of times that a release or a split operation satisfies a waiting allocate operation. The column labeled 'undone splits' lists the number of times a split operation finds no waiting allocate operations, and returns the memory block without breaking it. For comparison, we list the number of requests for memory, and the total number of times a split operation is called. Finally, we list the time to complete an allocate operation, to indicate the demand placed on the memory manager.

The experiments show that the parallel buddy algorithm has a memory efficiency that is usually about equal to that of the serial buddy algorithm. When too great a demand is placed on the memory manager (indicated by a response time much greater than the operation time), release operations can't be processed fast enough and the space utilization drops. As the parallel demand on the memory manager increases, the number of times that waiting allocate operations are satisfied by release or split operations increases. As a result, the number of split operations remains about constant, so that the space utilization also remains about constant. The number of undone splits also remains about constant because while more waiting allocate operations are being satisfied by release operations, more allocate operations are also arriving while the split is being processed.

5 Conclusion

We have presented a fast, simple, and highly parallel memory manager. The parallel buddy algorithm issues requests that larger blocks be split only when necessary, to minimize fragmentation. We test our algorithm by running a trace-driven simulation with memory request traces collected from a sparse matrix LU-factorization algorithm. The simulation results show that the parallel buddy algorithm has the same memory efficiency as the serial buddy algorithm.

The algorithm that we present is notable for being fast and highly parallel, and therefore is well suited for time-critical systems applications. This algorithm will be used to allocate buffers for message passing in a system in which shared memory processor clusters are connected by an ethernet [3].

To continue this work, we plan on parallelizing buddy algorithms that have better memory efficiency, and comparing all available parallel memory managers in a performance study.

6 Acknowledgements

We'd like to thank Fred Roeber for his helpful comments.

References

[1] B. Bigler, S. Allan, and R. Oldehoeft. Parallel dynamic storage allocation. In *International Conference on Parallel Processing*, pages 272–275. IEEE, 1985.

[2] T. A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Anal. Appl.*, 11(3):383–402, 1990.

[3] F.J. Roeber, Raytheon Submarine Signals Division. Personal communication, 1991.

[4] I. S. Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *J. Comp. Appl. Math.*, 27:229–239, 1989.

[5] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[6] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, 1986.

[7] C.S. Ellis and T. Olson. Concurrent dynamic storage allocation. In *Proceedings of the international Conference on Parallel Processing*, pages 502–511, 1987.

[8] R. Ford. Concurrent algorithms for real time memory management. *IEEE Software*, pages 10–23, September 1988.

[9] A. Gottlieb and J. Wilson. Using the buddy system for concurrent memory allocation. Ultracomputer System Software Note 6, Courant Institute, 1981.

[10] A. Gottlieb and J. Wilson. Parallelizing the usual buddy algorithm. Ultracomputer System Software Note 37, Courant Institute, 1982.

[11] T. Johnson. A concurrent fast-fits memory manager. Technical Report Electronic TR91-009, available at anonymous ftp site cis.ufl.edu:/cis/tech-reports/tr91/tr91-009.ps.Z, University of Florida, Dept. of CIS, 1991.

[12] D. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 1968.

[13] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[14] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In *Proc. of the Ninth ACM Symposium of OPerating System Principles*, pages 30–32, 1983.

[15] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1983.

[16] H. Stone. Parallel memory allocation using the fetch-and-add instruction. Technical Report RC 9674, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1982.

[17] J. Wilson. *Operating System Data Structures for Shared-memory MIMD Machines with Fetch-and-add.* PhD thesis, NYU, 1988.

A Parallel Buddy Algorithms

```
allocate(i){
   dosplit=false
   lock(freelist[i])
```

```
   if (the free list is empty){
      (* no blocks, so wait until some are available *)
      add self to freelist[i].Queue
      freelist[i].Nqueued++
      if(freelist[i].Nqueued>
         freelist[i].Nrequested){
         (* don't ask for more than you need *)
         dosplit = true
         freelist[i].Nrequested+=2
      }
      unlock(freelist[i])
      if(dosplit)
         split(i+1)
      sleep
   }
   else {
      get the block from the head of the free list
      unlock(freelist[i])
   }
}

release(i,,M){
    lock(freelist[i])
    if(freelist[i].Nqueued[i]>0) {
          (* give to blocked request if possible *)
       remove p from freelist[i].Queue
       freelist[i].Nqueued[i]- -
       unlock(freelist[i])
       wake up P, give it M
    }
    else {
       if(M's buddy is free) { (* coalesce *)
          remove buddy from the free list
          unlock(freelist[i])
          combine M and buddy
          release(i+1,min(M,buddy)
       }
       else { (* don't combine *)
          add M to the free list
          unlock(freelist[i])
       }
    }
}

split(i){
   M=allocate(i)
   split M into M and B
   lock(freelist[i-1])
   freelist[i-1].Nrequested-=2
```

```
      (* satisfying the request for 2 *)
   if(freelist[i-1].Nqueued>0) {
      remove P from freelist[i-1].Queue
      freelist[i-1].Nqueued- -
      remember to wake up P, give it M
         (* do it outside of locks *)
      if(freelist[i-1].Nqueued>0) {
         remove Q from freelist[i-1].Queue
         freelist[i-1].Nqueued- -
         remember to wake up Q, give it B
      }
      else {
         Navailable[i-1]++
         add B to the free list
      }
      unlock(freelist[i-1])
      wake up remembered processes
   }
   else {
      unlock(freelist[i-1])
      release(i,combine(M,B))
   }
}
```

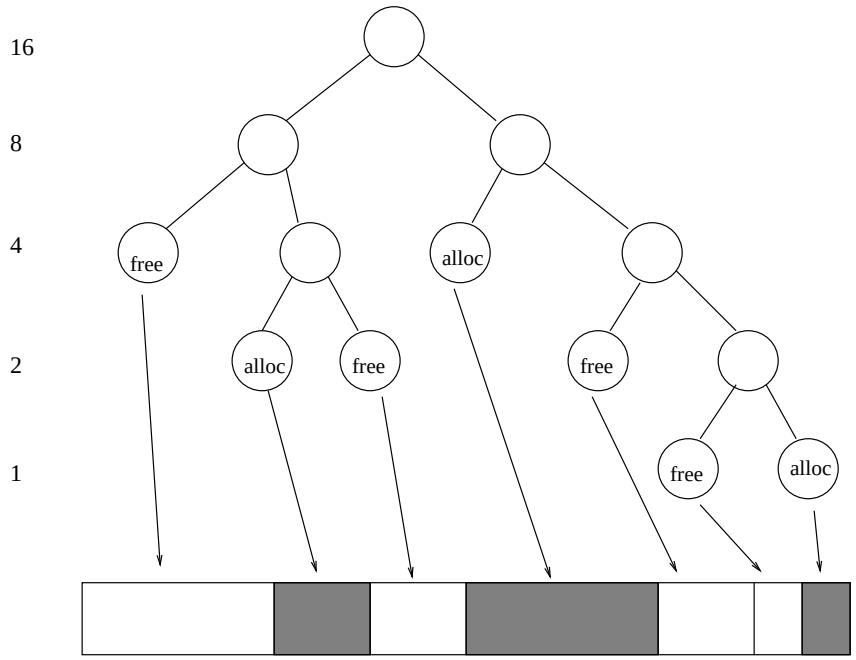

Figure 1: Tree representation of a binary buddy system

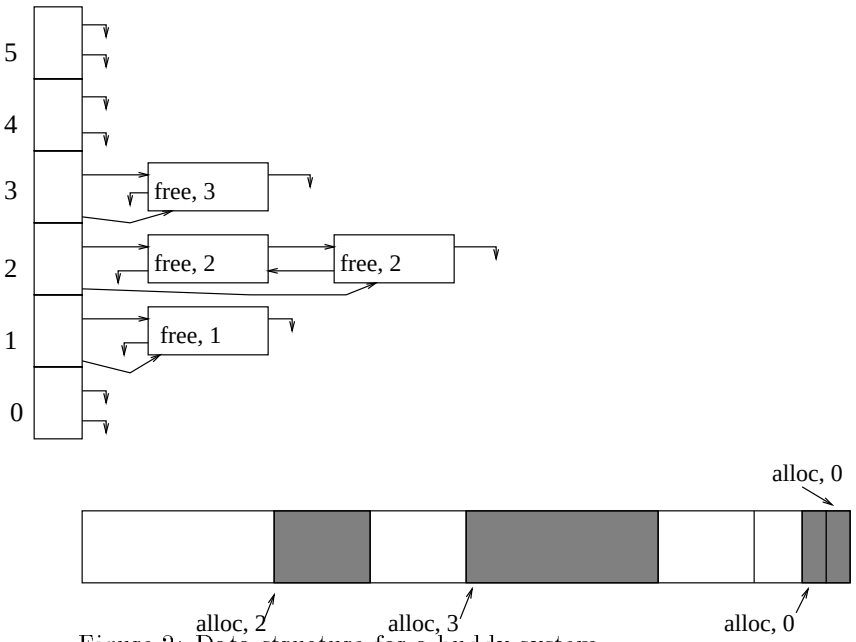

Figure 2: Data structure for a buddy system

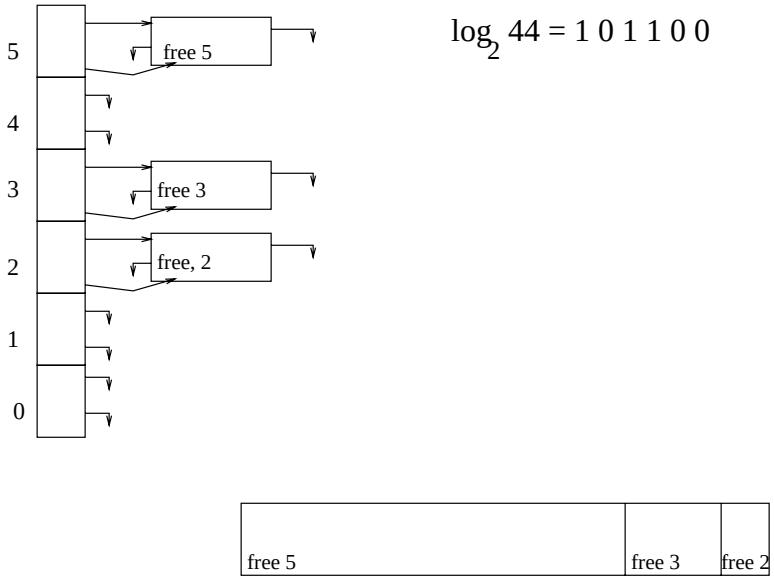

Figure 3: Initial configuration for a binary buddy system that stores 44 blocks