



Interaction

- [Help](#)
- [About Wikipedia](#)
- [Community portal](#)
- [Recent changes](#)
- [Contact page](#)


- Tools
- What links here
- Related changes
- Upload file
- Special pages
- Permanent link
- Page information
- Wikidata item
- Cite this page

Print/export

- Create a book
- Download as PDF
- Printable version

Languages

- Deutsch
- فارسی
- 한국어
- Italiano
- Polski
- Português

 Edit links

From Wikipedia, the free encyclopedia

Contents [\[hide\]](#)

- 1 How it works
- 2 In practice
- 3 Implementation and efficiency
- 4 See also
- 5 References

There are various forms of the buddy system, in which each block is subdivided into two smaller blocks, are the simplest and most common variety. Every memory block in this system has an *order*, where the order is an integer ranging from 0 to a specified upper limit. The size of a block of order n is proportional to 2^n , so that the blocks are exactly twice the size of blocks that are one order lower. Power-of-two block sizes make address computation simple, because all buddies are aligned on memory address boundaries that are powers of two. When a larger block is split, it is divided into two smaller blocks, and each smaller block becomes a unique buddy to the other. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

Starting off, the size of the smallest possible block is determined, i.e. the smallest memory block that can be allocated. If no lower limit existed at all (e.g., bit-sized allocations were possible), there would be a lot of memory and computational overhead for the system to keep track of which parts of the memory are allocated and unallocated. However, a rather low limit may be desirable, so that the average memory waste per allocation (concerning allocations that are, in size, not multiples of the smallest block) is minimized. Typically the lower limit would be small enough to minimize the average wasted space per allocation, but large enough to avoid excessive overhead. The smallest block size is then taken as the size of an order-0 block, so that all higher orders are expressed as power-of-two multiples of this size.

The **programmer** then has to decide on, or to write code to obtain, the highest possible order that can fit in the remaining available memory space. Since the total available memory in a given computer system may not be a power-of-two multiple of the minimum block size, the largest block size may not span the entire memory of the system. For instance, if the system had 2000K of physical memory and the order-0 block size was 4K, the upper limit on the order would be 8, since an order-8 block (256 order-0 blocks, 1024K) is the biggest block that will fit in memory. Consequently it is impossible to allocate the entire physical memory in a single chunk; the remaining 976K of memory would have to be allocated in smaller blocks.

The following is an example of what happens when a program makes requests for memory. Let's say in this system, the smallest possible block is 64 kilobytes in size, and the upper limit for the order is 4, which results in a largest possible allocatable block, 2^4 times 64K = 1024K in size. The following shows a possible state of the system after various memory requests.

Step	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							

2.3	2 ¹		2 ¹	2 ²		2 ³
2.4	2 ⁰	2 ⁰	2 ¹	2 ²		2 ³
2.5	A: 2 ⁰	2 ⁰	2 ¹	2 ²		2 ³
3	A: 2 ⁰	2 ⁰	B: 2 ¹	2 ²		2 ³
4	A: 2 ⁰	C: 2 ⁰	B: 2 ¹	2 ²		2 ³
5.1	A: 2 ⁰	C: 2 ⁰	B: 2 ¹	2 ¹	2 ¹	2 ³
5.2	A: 2 ⁰	C: 2 ⁰	B: 2 ¹	D: 2 ¹	2 ¹	2 ³
6	A: 2 ⁰	C: 2 ⁰	2 ¹	D: 2 ¹	2 ¹	2 ³
7.1	A: 2 ⁰	C: 2 ⁰	2 ¹	2 ¹	2 ¹	2 ³
7.2	A: 2 ⁰	C: 2 ⁰	2 ¹	2 ²		2 ³
8	2 ⁰	C: 2 ⁰	2 ¹	2 ²		2 ³
9.1	2 ⁰	2 ⁰	2 ¹	2 ²		2 ³
9.2	2 ¹		2 ¹	2 ²		2 ³
9.3	2 ²			2 ²		2 ³
9.4	2 ³					2 ³
9.5	2 ⁴					

This allocation could have occurred in the following manner

1. The initial situation.
2. Program A requests memory 34K, order 0.
 1. No order 0 blocks are available, so an order 4 block is split, creating two order 3 blocks.
 2. Still no order 0 blocks available, so the first order 3 block is split, creating two order 2 blocks.
 3. Still no order 0 blocks available, so the first order 2 block is split, creating two order 1 blocks.
 4. Still no order 0 blocks available, so the first order 1 block is split, creating two order 0 blocks.
 5. Now an order 0 block is available, so it is allocated to A.
3. Program B requests memory 66K, order 1. An order 1 block is available, so it is allocated to B.
4. Program C requests memory 35K, order 0. An order 0 block is available, so it is allocated to C.
5. Program D requests memory 67K, order 1.
 1. No order 1 blocks are available, so an order 2 block is split, creating two order 1 blocks.
 2. Now an order 1 block is available, so it is allocated to D.
6. Program B releases its memory, freeing one order 1 block.
7. Program D releases its memory.
 1. One order 1 block is freed.
 2. Since the buddy block of the newly freed block is also free, the two are merged into one order 2 block.
8. Program A releases its memory, freeing one order 0 block.
9. Program C releases its memory.
 1. One order 0 block is freed.
 2. Since the buddy block of the newly freed block is also free, the two are merged into one order 1 block.
 3. Since the buddy block of the newly formed order 1 block is also free, the two are merged into one order 2 block.
 4. Since the buddy block of the newly formed order 2 block is also free, the two are merged into one order 3 block.
 5. Since the buddy block of the newly formed order 3 block is also free, the two are merged into one order 4 block.

As you can see, what happens when a memory request is made is as follows:

- If memory is to be allocated
 1. Look for a memory slot of a suitable size (the minimal 2^k block that is larger or equal to that of the requested memory)
 1. If it is found, it is allocated to the program
 2. If not, it tries to make a suitable memory slot. The system does so by trying the following:
 1. Split a free memory slot larger than the requested memory size into half

2. If the lower limit is reached, then allocate that amount of memory
 3. Go back to step 1 (look for a memory slot of a suitable size)
 4. Repeat this process until a suitable memory slot is found
- If memory is to be freed
 1. Free the block of memory
 2. Look at the neighboring block - is it free too?
 3. If it is, combine the two, and go back to step 2 and repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbour block is encountered

Implementation and efficiency [[edit](#)]

In comparison to other simpler techniques such as [dynamic allocation](#), the buddy memory system has little [external fragmentation](#), and allows for [compaction](#) of memory with little overhead. The buddy method of freeing memory is fast, with the maximal number of compactions required equal to $\log_2(\text{highest order})$. Typically the buddy memory allocation system is implemented with the use of a [binary tree](#) to represent used or unused split memory blocks. The "buddy" of each block can be found with an [exclusive OR](#) of the block's address and the block's size.

However, there still exists the problem of [internal fragmentation](#) — memory wasted because the memory requested is a little larger than a small block, but a lot smaller than a large block. Because of the way the buddy memory allocation technique works, a program that requests 66K of memory would be allocated 128K, which results in a waste of 62K of memory. This problem can be solved by [slab allocation](#), which may be layered on top of the more coarse buddy allocator to provide more fine-grained allocation.

One version of the buddy allocation algorithm was described in detail by Donald Knuth in volume 1 of *The Art of Computer Programming*.^[2] The [Linux kernel](#) also uses the buddy system, with further modifications to minimise external fragmentation, along with various other allocators to manage the memory within blocks.^[3]

`jemalloc`^[4] is a modern memory allocator that employs, among others, the buddy technique.

See also [[edit](#)]

- [Memory pool](#)
- [Stack-based memory allocation](#)

References [[edit](#)]

1. ↑ Kenneth C. Knowlton. A Fast storage allocator. *Communications of the ACM* 8(10):623-625, Oct 1965. *also* Kenneth C Knowlton. A programmer's description of L6. *Communications of the ACM*, 9(8):616-625, Aug. 1966 [see also : Google books [1] ↗ page 85]
2. ↑ Knuth, Donald (1997). *Fundamental Algorithms. The Art of Computer Programming 1* (Second ed.). Reading, Massachusetts: Addison-Wesley. pp. 435–455. ISBN 0-201-89683-4.
3. ↑ Mauerer, Wolfgang (October 2008). *Professional Linux Kernel Architecture*. Wrox Press. ISBN 978-0-470-34343-2.
4. ↑ Evans, Jason (16 April 2006), *A Scalable Concurrent `malloc(3)` Implementation for FreeBSD* (PDF), pp. 4–5

Categories: [Memory management algorithms](#)

This page was last modified on 2 September 2015, at 20:40.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

