Efficient Programming in C

Memory Allocation
Comparison and implementation of basic malloc algorithms

Marcel Ellermann(6315155)

DKRZ - Deutsches Klima Rechenzentrum

Contents

Abstract In this paper we are going to depict some basic algorithms used in many malloc implementations like first-fit and best fit. After introducing those algorithms we are going to compare them regarding efficiency in different use cases and try to derive some general "best-use" rules.

1 Introduction

Memory Management is one of the research fields that yet still was not able to find a solution that is the best. Many different methods have been developed to handle memory in computer systems and most commonly enhance only single aspects like time or space consumption and thereby increase some others. Therefore it is always a trade off which method to use in each case. In this paper we can not analyze all different methods that have been developed or are currently used in standard libraries but try to investigate some basic approaches that are used commonly, though sometimes only partially, in malloc implementations. Hence a short introduction to Memory Management is given in section 2. Some general

approaches are introduced in section 3 which are further analyzed in different use cases in section 4 together with common used malloc implementations from other libraries. In section 5 we try to get to a conclusion and give some advise on when to use which introduced approach.

2 Dynamic Memory Allocation

In this chapter we are going to give a short introduction on Memory Management and how the Unix Kernel is handling physical and virtual memory address-spaces. Moreover we are going to give a short overview of heap memory allocation and depict the problems that occur if dynamic memory allocation is required.

2.1 Memory Management within the Unix Kernel

The Unix kernel describes the physical memory with pages. Pages describe a specific amount of physical memory and their size is architecture dependent. 32-Bit systems usually define a page size of 4 KB and 64-Bit systems usually define a page size of 8 KB. Often more than one page size is supported by the architecture. For a 32-Bit Architecture with 2 GB memory and a page size of 4 KB the MMU (Memory Management Unit) would need 524288 pages to describe the whole memory.[11]

2.2 Virtual Memory and Address spaces

The Unix Kernel differentiates between two different spaces. The kernel space and the user space. The kernel runs in kerne space and can directly communicate with the hardware and has its own secured memory space, whereas the user space is not allowed to directly communicate with the hardware.[7][11] As shown

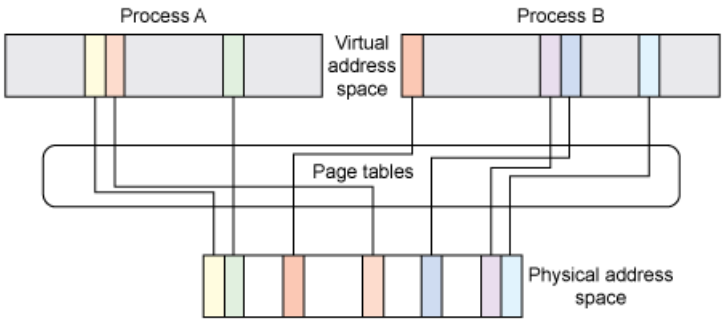

Figure 1. Mapping of virtual address spaces to physical memory [7]

in figure 1 each process receives a virtual address space. This virtual address

space is mapped to physical memory and therefore secure. Security derives from the fact that each process has it's own address space and can't access the other processes' memory, it's isolated. The Memory Management Unit typically handles the task of mapping the virtual address spaces to the physical addresses. It most commonly resides on the CPU. By default the process's virtual memory is not mapped completely, just a small portion is mapped. This is due enormous overhead of managing all page-tables for each process, even when they are only using a small fraction of the total memory available. To request a mapping from virtual memory to physical memory the process hast to make calls the the kernel's interfaces that offer such means. Though virtual memory might be contiguous the same is not guaranteed for the mapped physical memory. The Unix Kernel itself has internal methods to allocate contiguous physical memory but those won't be part of this paper because we will only need the interfaces the Unix Kernel offers the user-space. Those are *sbrk* and *mmap*.

2.3 The sbrk interface

We will briefly introduce the kernel's sbrk interface here since it will be used for our own malloc implementations in section 3. Each process has its virtual memory fragmented into different areas which are used for different purposes. One of those is the data fragment, which is as the name already implicates used for data storage. The sbrk interface is used to increase the size of the data fragment of the current process. It will return the start of the new available memory space on success.[3]

2.4 Segementation Faults

Due to the fact that only portions of the virtual address space are actually mapped to physical memory so called Segmentation Faults may occur. Segmentation Faults are caused by access violations, which means the process does not have the rights to access the requested memory address. This may be due to the properties of the memory address, like read-only memory areas, or it may occur because the process doesn't own the memory. This means the requested address is neither within the processes' data segment nor on its stack. Usually the memory address 0, the null pointer, is protected by the system. Therefore the process has no rights to access that address and a Segmentation Fault is caused when an attempt is made to access it.

2.5 The Heap

The concept of a stack should be known in general. Though this concept allows fast allocation of memory and a minimum of administrative work we're rather limited by the fact that we can only free the top of the stack at any given time. Which leads us to the idea of a heap. The heap shall in contrary to the stack have the following key features:

- Free Allocated Memory at any given time
- Allocate Memory at any given time

By being able to free memory at any given time at any position within the heap we are free of any restrictions we have within the heap. The loss of those restrictions though leads to the need of an administration of the memory we're working with because we have to gather and reuse the memory we have freed so far to not waste it. If and only if no free memory is available within the memory currently contained by the heap we request new memory from the operating system and increase the heap's size. This can be done by the sbrk interface introduced in section 2.3. A general structure of a heap is shown in figure 2. Each memory chunk administrated by the heap has small portion of memory

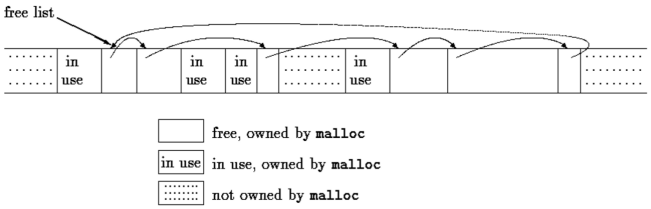

Figure 2. Basic structure of a heap. [8]

that is called the head and is used to describe the chunk's size and the position of the next chunk that contains free memory. By linking to the next chunk a linked-list is formed which is called the free list. The chunks within this list are ordered by ascending address size. The last chunk points to the first one.

Request memory If memory is requested from the heap a chunk of memory has to be found that is at least as large as the amount requested. If no memory chunk is big enough, new memory has to be requested from the operating system.

Free memory If memory is freed it has to be returned into the *free list*. The malloc implementation has to take care of the ascending order of addresses when it inserts the memory chunk into the list.[8]

2.6 External fragmentation

External fragmentation is due to the dynamic allocation and deallocation of memory a problem that arises in heaps. External fragmentation is "*a form of fragmentation that arises when memory is allocated in units of arbitrary size. When a large amount of memory is released, part of it may be used to meet a subsequent request, leaving an unused part that is too small to meet any further requests*"[4]. Methods to reduce external fragmentation will be discussed in section 3.

3 General approaches

Within this section we are going to introduce the basic methods for dynamic memory allocation. This includes methods for chunk selection within the free list of the heap. Alternative heap structures as well as methods to decrease external fragmentation and allocation time under different circumstances.

3.1 Coalescing

One method to decrease external fragmentation drastically is coalescing. Coalescing basically is the merging of two neighbored chunks of free memory. If a memory chunk is freed and returned to the free list, the freeing algorithm checks for the position the freed block is in and coalesces under the following conditions

- The freed chunk starts where a free chunk ends.
- The freed chunk starts where a free chunk starts.

If both condition are true, all three blocks are merged into one big memory chunk.

3.2 First fit

The first fit method for selecting memory from the is rather simple. The free list will be searched from the beginning every time an allocation has to be made. The first block that is big enough to contain the requested size of bytes will be returned and removed from the free list. If the block is larger than the requested amount of bytes it is split to fit the requested size and the left free space is returned to the free list. This approach tends to accumulate small memory chunks at the beginning of the free list, which leads to longer search times for larger memory blocks.[9]

3.3 Next fit

The next fit method is very similar the the fist fit method and could rather be called a tweak of the first fit method. Instead of always start searching from the beginning of the free list the search is continued from the last returned chunk of memory. By doing this the free chunks of memory are most likely distributed uniformly in size over the free list which leads to a reduction of search time in average cases.[9]

3.4 Best fit

Best fit will always return the chunk of memory from the free list which size is closest to the requested size. To do this every chunk of memory within the free list has to be checked which leads to high average search times. Though

the search time increases this attempt also decreases the the amount of external fragmentation. Further more fragmentation can be reduced by specifying a minimal chunk size. The minimal chunk size specifies that a block is not split if the free chunk resulting from that split is smaller than the given minimal chunk size. The minimal chunk size is not only applicable to the best fit method, it can also be used for the next fit and first fit method and is a trade of between external fragmentation and memory efficiency.[9][1]

3.5 Binning

Binning uses a a different heap structure than introduced in 2.5. As you can see in figure 3 we have 128 *bins* each being the head of a free list that contains chunks of memory that have a specific size. Bins 1-64 contain each memory chunks of exact $n * 8$ Bytes, where n is the index of the bin. Bins with an index greater than 64 contain a range of memory chunks. Those bins are sorted by ascending size within the respective free list. Before 1995 instead of sorting the *oldest-first rule* was commonly used. Due to the proof that with sorting *"the minor time investment was worth it to avoid observed bad cases"*[10] this had changed.

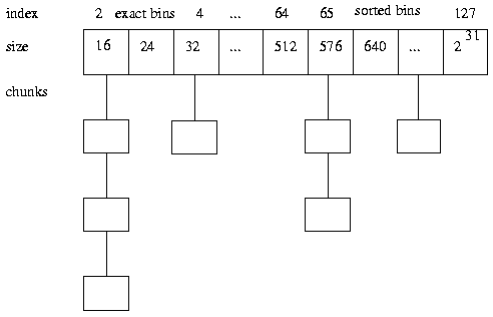

Figure 3. Binning for dynamic memory allocation

3.6 Caching

Caching refers to the fact that small chunks of memory that are freed are not coalesced immediately. This is rather helpful when working with many small allocations and deallocations in a small period of time, i.e. when working with huge trees. Every split of a memory chunk and every merge of a memory chunk has a fixed cost of time attached to it. Therefore caching can save this time by skipping those operations and keeping the freed memory chunk as it is. If the number of cached chunks exceed a given number, they're coalesced again.

4 Use case analysis

In this section we are going to analyze how good our self implemented malloc methods for *First-Fit* and *Next-Fit* are compared to malloc implementations of other libraries commonly used, i.e. the glibcs' malloc [6] implementation or jemalloc[2].

4.1 Glibc Malloc

The default c library used on Ubuntu systems is provided by glibc and henceforth the default malloc implementation used by most of the programmers developing on such systems use the malloc implementation from glibc as well. The malloc implementation from glibc combines many method introduced in section 3. For different memory chunk sizes different approaches are used to optimize speed and memory efficiency.

- For Chunks less or equal to 64 Bytes a caching allocator is used.
- For Chunks greater than 512 Bytes a simple best-fit allocator is used.
- For everything in between *"it does the best it can trying to meet both goals at once"*[5]. Where the goals are time and memory efficiency.

The implementation describes itself as follow: *This is not the fastest, most space-conserving, most portable, or most tunable malloc ever written. However it is among the fastest while also being among the most space-conserving, portable and tunable. Consistent balance across these factors results in a good general-purpose allocator for malloc-intensive programs.*[5].

4.2 Allocation of small memory blocks 8 to 512 Bytes

The following test was conducted on the cluster of the DKRZ. A number of memory chunks were allocated. Each allocated memory chunk was between 8 and 512 Bytes large and was freed after 1 to 3 seconds. The result of this test with the different malloc implementations is displayed in figure 4. As we can see *first-fit* and *next-fit* are pretty similar though *first-fit* slightly out performs *next-fit*. Better than those to approaches is *glibs'* malloc implementation with $365,57$ seconds for 300.000 allocations. It is only beaten by the *jemalloc* implementation with $256,89$ seconds for 300.000 allocations, which is a huge lead.

4.3 Allocation of larger memory blocks 512 to 8096 Bytes

The following test was conducted on the cluster of the DKRZ. A number of memory chunks were allocated. Each allocated memory chunk was between 512 and 8096 Bytes large and was freed after 1 to 3 seconds. The result of this test with the different malloc implementations is displayed in figure 5. The result is pretty similar to the one from section 4. *first-fit* and *next-fit* are even closer together and glib as well as jemalloc have a huger lead towards the other implementations.

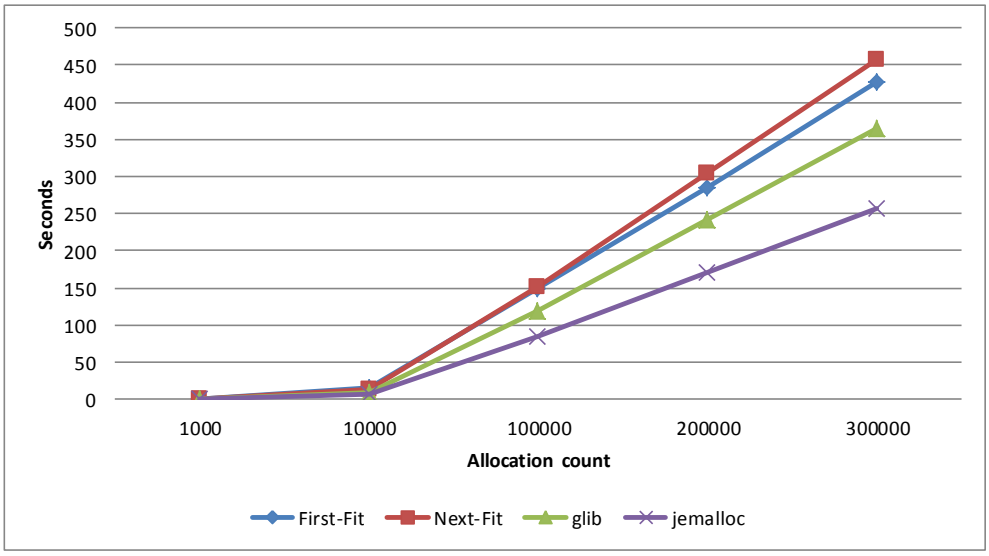

Figure 4. Allocation duration 1-3 seconds. Chunk size 8 - 512 Byte

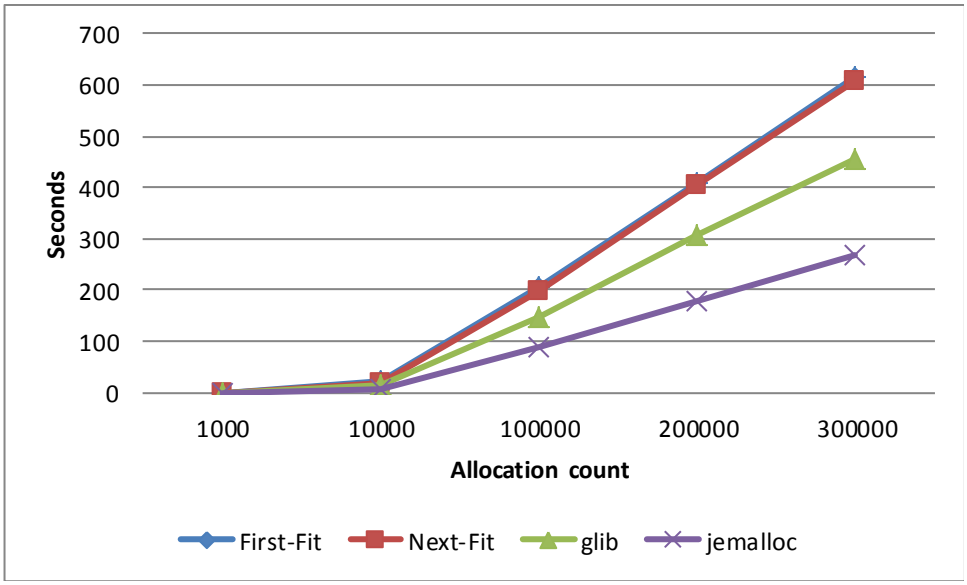

Figure 5. Allocation duration 1-3 seconds. Chunk size 512 - 8069 Byte

5 Conclusion

Reviewing the results of the testes conducted in section 4 there are some things to point out. First of all they have to be treated carefully since they're controversially to theory. In theory next-fit should outperform first-fit, though the opposite is shown here. Despite that the results show that one is better off using the default implementation or other third party libraries since own implementation require a lot of optimization to outperform any of the introduced already present malloc implementations. If one needs fast memory allocators for time dependent applications it is worth looking into some malloc implementations if the program is very memory intensive. For example jemalloc is about 200 seconds faster for large memory blocks and 300.000 allocations in the given test setting. The results may differ in other cases were less deallocations or a broad range of memory sizes are required.

References

1. Carter Bays. A comparison of next-fit, first-fit and best-fit. *Communications of the ACM*, 20(3):191–192, March 1977.
2. Canonware. Jemalloc. `http://www.canonware.com/jemalloc/`. [Accessed on 18.03.2014].
3. Panagiotis Christias. Linux programmer's manual. `http://unixhelp.ed.ac.uk/CGI/man-cgi?sbrk+2`. [Accessed on 09.01.2014].
4. JOHN DAINTITH. External fragmentation. `http://www.encyclopedia.com/doc/1O11-externalfragmentation.html`. [Accessed on 09.01.2014].
5. GNU. Gllib c malloc header file. `http://code.woboq.org/userspace/glibc/malloc/malloc.c.html`. [Accessed on 09.01.2014].
6. GNU. Linux programmer's manual - malloc. `http://man7.org/linux/man-pages/man3/malloc.3.html`. [Accessed on 09.01.2014].
7. M. Tim Jones. User space memory access from the linux kernel. `http://www.ibm.com/developerworks/linux/library/l-kernel-memory-access/index.html`, 2011. [Accessed on 09.01.2014].
8. Brian W. Kernighan and Dennis M. Ritchie. *The C programming Language*. Prentice-Hall, 2 edition, 1988.
9. Donald E. Knuth. *The art of computer programming - Fundamental algorithms*, volume 1. Addison Wesley, 2006.
10. Doug Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`. [Accessed on 09.01.2014].
11. Robert Love. *Linux Kernel Development*. Novell Press, 2 edition, 2005.