UNDERSTANDING UNIX

MALLOC IMPLEMENTATION
Part 1

"…memory allocation is widely considered to be either a solved problem or an insoluble one." – [2]

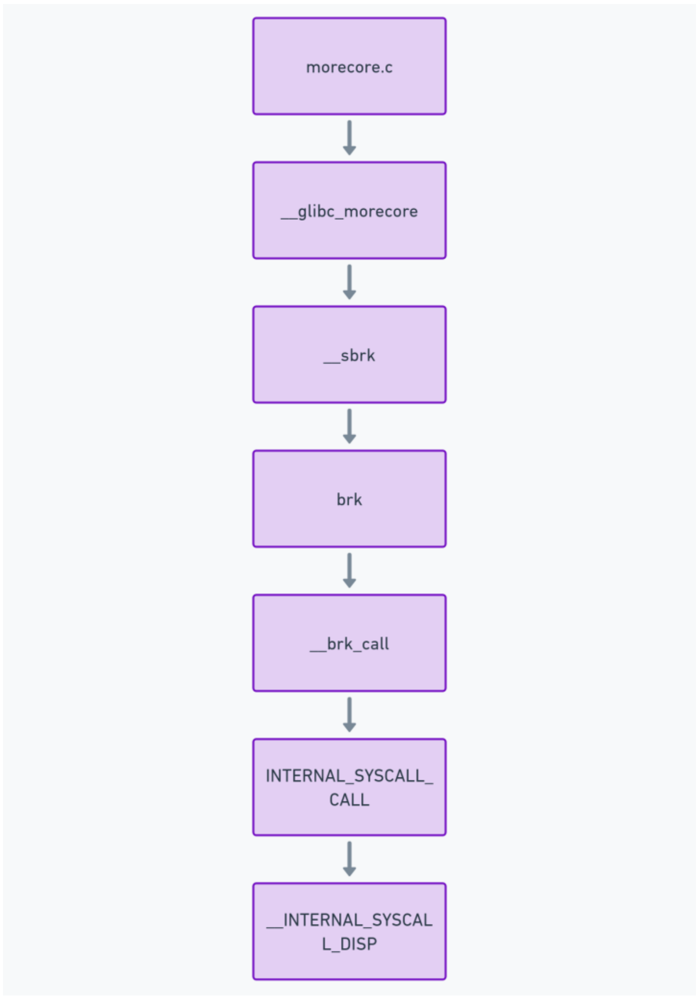
This article is not for someone who is just interest in the term "Embedded Systems" but for those who want to understand what "Embedded Systems" actually is. Instead of just saving this document, if you are curious about the title that I mentioned, then its definitely worth a read. (This entire series)

On my Journey to write a custom **Memory Management System** that can be used in any RTOS, I would like to dive deep into the actual implementation of malloc in Linux. Although the entire memory management system contains many parts, I would first like to understand how the popularly used '**malloc()**' gets the memory.

Let it be a regular user who is beginning to learn ***Dynamic Memory Allocation*** in ***C*** or someone who uses Dynamic Memory allocation on regular basis, they just write **malloc()/calloc()** and magically, the requested memory is given to you by the kernel.

How does it work? Well yes, you give the pointer and the size to be allocated and the system give you the requested amount; but how? Here we go…..

Note: Here we deal just with the understanding of how malloc works. I would be dealing with Real memory and Virtual memory in this article. Comment down If interested in learning about that.



The Dynamic memory allocators we usually use: malloc(), calloc(), realloc(), free(). Here we are only gonna dive into the understanding of malloc().

The current malloc() version of UNIX uses the base implementation of root memory allocator, dlmalloc().

When creating the memory allocator, *Doug Lea* created the dlmalloc() with the following goals:

“

A good memory allocator needs to balance a number of goals:

Maximizing Compatibility

An allocator should be plug-compatible with others; in particular it should obey ANSI/POSIX conventions.

Maximizing Portability

Reliance on as few system-dependent features (such as system calls) as possible, while still providing optional support for other useful features found only on some systems; conformance to all known system constraints on alignment and addressing rules.

Minimizing Space

The allocator should not waste space: It should obtain as little memory from the system as possible, and should maintain memory in ways that minimize *fragmentation* -- ``holes''in contiguous chunks of memory that are not used by the program.

Minimizing Time

The malloc(), free() and realloc routines should be as fast as possible in the average case.

Maximizing Tunability

Optional features and behavior should be controllable by users either statically (via #define and the like) or dynamically (via control commands such as mallopt).

Maximizing Locality

Allocating chunks of memory that are typically used together near each other. This helps minimize page and cache misses during program execution.

Maximizing Error Detection

It does not seem possible for a general-purpose allocator to also serve as general-purpose memory error testing tool such as *Purify*. However, allocators should provide some means for detecting corruption due to overwriting memory, multiple frees, and so on.

Minimizing Anomalies

An allocator configured using default settings should perform well across a wide range of real loads that depend heavily on dynamic allocation -- windowing toolkits, GUI applications, compilers, interpretors, development tools, network (packet)-intensive programs, graphics-intensive packages, web browsers, string-processing applications, and so on.

”[1]

The Dynamic memory allocation happens in Heap. Heap is simply a pool of memory that is available for memory allocation and deallocation whenever the user request whatever memory size.

The ***glibc*** memory is derived from **ptmalloc** (**pthreads** malloc), which is derived from ***dlmalloc***.

Before diving into the code, let us get some familiarity of the terms used to describe the malloc functionality:

“**Arena**

A structure that is shared among one or more threads which contains references to one or more heaps, as well as linked lists of chunks within those heaps which are "free". Threads assigned to each arena will allocate memory from that arena's free lists.

Heap

A contiguous region of memory that is subdivided into chunks to be allocated. Each heap belongs to exactly one arena.

Chunk

A small range of memory that can be allocated (owned by the application), freed (owned by glibc), or combined with adjacent chunks into larger ranges. Note that a chunk is a wrapper around the block of memory that is given to the application. Each chunk exists in one heap and belongs to one arena.

Memory

A portion of the application's address space which is typically backed by RAM or swap.

”[3]

The main properties of the Malloc algorithm implemented in UNIX is:

- For larger requests, i.e >=512 bytes,
    - it uses a pure best-fit allocator.
    - ties normally decided bua FIFO

- For small requests, ie. <= 64 bytes,
    - it uses a caching allocator
    - maintains the pools of quickly recycled chunks

- If the request is in between that,
    - allocates by the combination of the above

- If the request is very large, i.e, >=128KB,
    - It relies on the system’s memory mapping facilities.

Malloc calls sbrk or mmap(). sbrk() calls brk()

- brk() sets the end of the data segment to a specified value
- sbrk(), on the other hand, increments the program's data space by a specified number of bytes

When memory is requested, the kernel first gets the memory then takes responsibility to map that memory in virtual memory.

References:

[1] https://gee.cs.oswego.edu/dl/html/malloc.html

[2] "*Dynamic Storage Allocation: A Survey and Critical Review*" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles from the Department of Computer Sciences, University of Texas at Austin.

[3] https://sourceware.org/glibc/wiki/MallocInternals