Daniel Lemire's blog

Daniel Lemire is a computer science professor at the Data Science Laboratory of the Université du Québec (TÉLUQ) in Montreal. His research is focused on software performance.

How much memory does a call to ‘malloc’ allocate?

In C, we allocate memory on the heap using the `malloc` function. Other programming languages like C++ or zig (e.g., `std.heap.c_allocator`) may call on `malloc` underneath so it is important to understand how `malloc` works. Furthermore, the same concepts apply broadly to other memory allocators.

In theory, you could allocate just one byte like so:

```
char * buffer = (char*) malloc(1);
```

How much memory does this actually allocate?

On modern systems, the request allocates virtual memory which may or may not be actual (physical) memory. On many systems (e.g., Linux), the physical memory tends to be allocated lazily, as late as possible. Other systems such as Windows are more eager to allocate physical memory. It is also common and easy to provide your own memory allocators, so the behavior varies quite a bit.

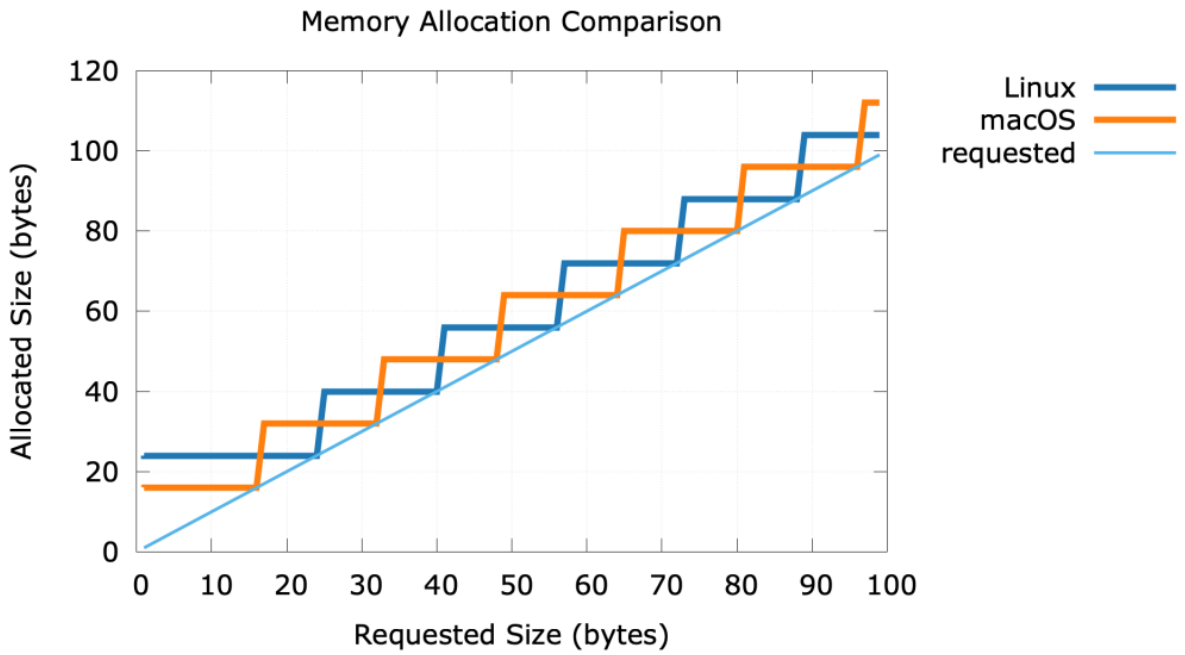
But how much virtual memory does my call to `malloc(1)` typically?

There is likely some fixed overhead per allocation: you can expect 8 bytes of meta-data per allocation although it could be less or more depending on the allocator. You cannot use this overhead in your own program: it is consumed by the system to keep track of the memory allocations. Even if you have a highly efficient allocator with little overhead per allocation, the pointer itself must typically be tracked and, on a 64-bit system, that represents 8 bytes of data.

If you asked for 1 bytes, you are probably getting a large chunk of usable memory: maybe between 16 bytes and 24 bytes. Indeed, most memory allocations are aligned (rounded up) so that the address is divisible by `max_align_t` (typically 16 bytes) and there is a minimum size that you may get. And, indeed, the C language has a function called `realloc` which can be used to extend a memory allocation, often for free because the memory is already available.

You can ask how much memory is available. Under Linux, you can use the `malloc_usable_size` while under FreeBSD and macOS, you can use `malloc_size`. So I can write a small programs that asks how much (virtual) memory was actually granted given a request. For one byte, my macOS laptop gives out 16 bytes while my x64 Linux server seemingly gives 24 bytes. If I plot the memory actually granted versus the memory requested, you see a staircase where, on average, you get 8 extra bytes of memory. That is to be expected if the pointer returned must by at an address divisible by 16 (`max_align_t`). Overall, you should avoid broad generalizations about how Linux or macOS work or compare, and simply keep in mind the broad picture.

What do we conclude? You probably should avoid allocating on the heap tiny blocks of memory (i.e., smaller than 16 bytes). Furthermore, you may not want to optimize the allocation size down to a few bytes since it gets rounded up in any case. Finally, you should make use of `realloc` if you can as you can often extend a memory region, at least by a few bytes, for free.



Daniel Lemire, "How much memory does a

call to ‘malloc’ allocate?," in *Daniel Lemire's blog*, June 27, 2024, https://lemire.me/blog/2024/06/27/how-much-memory-does-a-call-to-malloc-allocates/.

PUBLISHED BY

Daniel Lemire
A computer science professor at the University of Quebec (TELUQ).
View all posts by Daniel Lemire →

7 thoughts on “How much memory does a call to ‘malloc’ allocate?”

JB

June 28, 2024 at 9:29 am

Note that it is possible to have zero overhead in malloc. This would be done by trading space for time: in the arena you’d just compare the malloc’ed pointer to the pointer values in the bins (so instead of immediate access to the metadata by looking at the overhead space, you get the same information by using the address to figure where the relevant metadata is). This turns O(1) access and O(alloc) space into 0-space but O(arena size)… you’d do this only if space is critical but runtime irrelevant. I have yet to see a situation where this is warranted.

Also the overhead space (usually stored in front of the pointer returned to user) also needs to be aligned and sized to the minimum accessible memory location (so pointer-sized), so often the minimum will be 16 bytes. All in all, very small allocations are best avoided.

Verisimilitude

June 29, 2024 at 1:53 am

I’ve noticed a minor grammatical error:

> So I can write a small programs that inquires how much (virtual) memory was actually granted given a request.

I was just earlier today finishing up an Ada program that allocated everything on the stack. Once I told the system to give every program an unlimited stack size, I no longer got Storage_Error thrown in my face. One reason Ada is pleasant would be how it allows the programmer to avoid dynamic memory allocation in most cases. Of course, Storage_Error makes this safe, whereas I’m to understand the C language alloca routine just silently blows up in this case.

Laurent

June 29, 2024 at 10:17 am

This made me want to play with jemalloc on my Ubuntu 20.04 machine 🙂

Default C library:
size usable size
1 24
25 40
41 56
57 72
73 88
89 104
105 120
121 136

With jemalloc:
size usable size
1 8
9 16
17 32
33 48
49 64
65 80
81 96
97 112
113 128

Clive Robinson

June 30, 2024 at 4:29 pm

Back in the 8bit days when memory really was a scarce resource sometimes you had only 120bytes of RAM various tricks had to be used.

In slightly better resourced systems people used stacks and arrays of memory types.

So if you only had 2byte and 4byte data sizes you could use a single byte as a pointer. If the sign bit was clear it pointed to 4byte and if set 2byte so a shift, mask and offset

add gave a 16bit pointer.

For various reasons such very resource limited microcontrollers are still used in “new products” even today.

Maks Verver

July 3, 2024 at 6:39 pm

This technique is not just used in resource-limited microcontrollers! It also lives on in the Java VM, which uses a similar trick, called “Compressed OOPs”.

Since the Java heap is a contiguous slice of memory, and Java objects are always aligned to 8-byte boundaries, a pointer to a heap object can be encoded as (base + 8*oop), where the oop is a 32-bit offset. This works for heap sizes of up to 32 GiB, which is plenty for most applications (the default is typically 4 GiB).

This cuts the size of pointers in half on 64-bit systems, which significantly reduces the total memory used, though loading/storing pointers becomes more expensive.

There is an additional optimization, called “zero-based compressed OOPs”, where the JVM tries to allocate the heap at virtual address 0, which removes the need for a base pointer, which frees up a register and makes conversions slightly faster. (This is not supported on all operating systems; in particular security-focused distributions typically require that the lowest addresses remain unmapped, to defend against null-pointer dereferences.)

Robert Eisele

July 25, 2024 at 6:01 pm

While reading your article, I was waiting for a hint what would be the >optimal< way to implement a malloc (w.r.t. memory alignment and the necessity to realloc). Also does it make sense to scale the allocated region linearly? I've seen implementations that scaled exponentially, with the expense of unnecessary copying data to other regions when a realloc doesn't fit anymore. What do you think is better in your test? Linux? macOS? Depends on the architecture?

Daniel Lemire

July 25, 2024 at 6:57 pm

There are different memory allocators and a developer can write their own.

I strongly suspect that there is no such thing as an optimal malloc for all use cases.
