Why malloc+memset is slower than calloc?

Asked 14 years, 8 months ago Modified 2 years, 10 months ago Viewed 69k times

301

It's known that `calloc` is different than `malloc` in that it initializes the memory allocated. With `calloc`, the memory is set to zero. With `malloc`, the memory is not cleared.

So in everyday work, I regard `calloc` as `malloc`+`memset`. Incidentally, for fun, I wrote the following code for a benchmark.

The result is confusing.

Code 1:

```
#include<stdio.h>
#include<stdlib.h>
#define BLOCK_SIZE 1024*1024*256
int main()
{
        int i=0;
        char *buf[10];
        while(i<10)
        {
                buf[i] = (char*)calloc(1,BLOCK_SIZE);
                i++;
        }
}
```

Output of Code 1:

```
time ./a.out
**real 0m0.287s**
user 0m0.095s
sys 0m0.192s
```

Code 2:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define BLOCK_SIZE 1024*1024*256
int main()
{
        int i=0;
        char *buf[10];
        while(i<10)
        {
                buf[i] = (char*)malloc(BLOCK_SIZE);
                memset(buf[i],'\0',BLOCK_SIZE);
```

```
                i++;
        }
}
```

Output of Code 2:

```
time ./a.out
**real 0m2.693s**
user 0m0.973s
sys 0m1.721s
```

Replacing `memset` with `bzero(buf[i],BLOCK_SIZE)` in Code 2 produces the same result.

My question is: Why is `malloc`+`memset` so much slower than `calloc`? How can `calloc` do that?

`c` `malloc`

Share
Improve this question
Follow

edited Jul 20, 2013 at 19:03
Philip Conrad
1,489 ● 1 ● 13 ● 24

asked Apr 22, 2010 at 5:40

kingkai
3,729 ● 3 ● 20 ● 14

should've done `buf[i] = (char*)memset(malloc(BLOCK_SIZE),0,BLOCK_SIZE);` - but a smart optimizing compiler should've auto-optimized that anyway – hanshenrik Nov 1, 2023 at 20:50

3 Answers

Sorted by: Highest score (default)

565

The short version: Always use `calloc()` instead of `malloc()+memset()`. In most cases, they will be the same. In some cases, `calloc()` will do less work because it can skip `memset()` entirely. In other cases, `calloc()` can even cheat and not allocate any memory! However, `malloc()+memset()` will always do the full amount of work.

Understanding this requires a short tour of the memory system.

Quick tour of memory

There are four main parts here: your program, the standard library, the kernel, and the page tables. You already know your program, so...

Memory allocators like `malloc()` and `calloc()` are mostly there to take small allocations (anything from 1 byte to 100s of KB) and group them into larger pools of memory. For example, if you allocate 16 bytes, `malloc()` will first try to get 16 bytes out of one of its pools, and then ask for more memory from the kernel when the pool runs dry. However, since the program you're asking about is allocating for a large amount of memory at once, `malloc()` and `calloc()` will just ask for that memory directly from the kernel. The threshold for this behavior depends on your system, but I've seen 1 MiB used as the threshold.

The kernel is responsible for allocating actual RAM to each process and making sure that processes don't interfere with the memory of other processes. This is called *memory protection,* it has been dirt common since the 1990s, and it's the reason why one program can crash without bringing down the whole system. So when a program needs more memory, it can't just take the memory, but instead it asks for the memory from the kernel using a system call like `mmap()` or `sbrk()`. The kernel will give RAM to each process by modifying the page table.

The page table maps memory addresses to actual physical RAM. Your process's addresses, 0x00000000 to 0xFFFFFFFF on a 32-bit system, aren't real memory but instead are addresses in *virtual memory.* The processor divides these addresses into 4 KiB pages, and each page can be assigned to a different piece of physical RAM by modifying the page table. Only the kernel is permitted to modify the page table.

How it doesn't work

Here's how allocating 256 MiB does *not* work:

1. Your process calls `calloc()` and asks for 256 MiB.
2. The standard library calls `mmap()` and asks for 256 MiB.
3. The kernel finds 256 MiB of unused RAM and gives it to your process by modifying the page table.
4. The standard library zeroes the RAM with `memset()` and returns from `calloc()`.
5. Your process eventually exits, and the kernel reclaims the RAM so it can be used by another process.

How it actually works

The above process would work, but it just doesn't happen this way. There are three major differences.

- When your process gets new memory from the kernel, that memory was probably used by some other process previously. This is a security risk. What if that

memory has passwords, encryption keys, or secret salsa recipes? To keep sensitive data from leaking, the kernel always scrubs memory before giving it to a process. We might as well scrub the memory by zeroing it, and if new memory is zeroed we might as well make it a guarantee, so `mmap()` guarantees that the new memory it returns is always zeroed.

- There are a lot of programs out there that allocate memory but don't use the memory right away. Sometimes memory is allocated but never used. The kernel knows this and is lazy. When you allocate new memory, the kernel doesn't touch the page table at all and doesn't give any RAM to your process. Instead, it finds some address space in your process, makes a note of what is supposed to go there, and makes a promise that it will put RAM there if your program ever actually uses it. When your program tries to read or write from those addresses, the processor triggers a *page fault* and the kernel steps in to assign RAM to those addresses and resumes your program. If you never use the memory, the page fault never happens and your program never actually gets the RAM.
- Some processes allocate memory and then read from it without modifying it. This means that a lot of pages in memory across different processes may be filled with pristine zeroes returned from `mmap()`. Since these pages are all the same, the kernel makes all these virtual addresses point to a single shared 4 KiB page of memory filled with zeroes. If you try to write to that memory, the processor triggers another page fault and the kernel steps in to give you a fresh page of zeroes that isn't shared with any other programs.

The final process looks more like this:

1. Your process calls `calloc()` and asks for 256 MiB.
2. The standard library calls `mmap()` and asks for 256 MiB.
3. The kernel finds 256 MiB of unused *address space,* makes a note about what that address space is now used for, and returns.
4. The standard library knows that the result of `mmap()` is always filled with zeroes (or *will be* once it actually gets some RAM), so it doesn't touch the memory, so there is no page fault, and the RAM is never given to your process.
5. Your process eventually exits, and the kernel doesn't need to reclaim the RAM because it was never allocated in the first place.

If you use `memset()` to zero the page, `memset()` will trigger the page fault, cause the RAM to get allocated, and then zero it even though it is already filled with zeroes. This is an enormous amount of extra work, and explains why `calloc()` is faster than `malloc()` and `memset()`. If you end up using the memory anyway, `calloc()` is still faster than `malloc()` and `memset()` but the difference is not quite so ridiculous.

This doesn't always work

Not all systems have paged virtual memory, so not all systems can use these optimizations. This applies to very old processors like the 80286 as well as embedded processors which are just too small for a sophisticated memory management unit.

This also won't always work with smaller allocations. With smaller allocations, `calloc()` gets memory from a shared pool instead of going directly to the kernel. In general, the shared pool might have junk data stored in it from old memory that was used and freed with `free()`, so `calloc()` could take that memory and call `memset()` to clear it out. Common implementations will track which parts of the shared pool are pristine and still filled with zeroes, but not all implementations do this.

Dispelling some wrong answers

Depending on the operating system, the kernel may or may not zero memory in its free time, in case you need to get some zeroed memory later. Linux does not zero memory ahead of time, and Dragonfly BSD recently also removed this feature from their kernel. Some other kernels do zero memory ahead of time, however. Zeroing pages during idle isn't enough to explain the large performance differences anyway.

The `calloc()` function is not using some special memory-aligned version of `memset()`, and that wouldn't make it much faster anyway. Most `memset()` implementations for modern processors look kind of like this:

```
function memset(dest, c, len)
    // one byte at a time, until the dest is aligned...
    while (len > 0 && ((unsigned int)dest & 15))
        *dest++ = c
        len -= 1
    // now write big chunks at a time (processor-specific)...
    // block size might not be 16, it's just pseudocode
    while (len >= 16)
        // some optimized vector code goes here
        // glibc uses SSE2 when available
        dest += 16
        len -= 16
    // the end is not aligned, so one byte at a time
    while (len > 0)
        *dest++ = c
        len -= 1
```

So you can see, `memset()` is very fast and you're not really going to get anything better for large blocks of memory.

The fact that `memset()` is zeroing memory that is already zeroed does mean that the memory gets zeroed twice, but that only explains a 2x performance difference. The

performance difference here is much larger (I measured more than three orders of magnitude on my system between `malloc()+memset()` and `calloc()`).

Party trick

Instead of looping 10 times, write a program that allocates memory until `malloc()` or `calloc()` returns NULL.

What happens if you add `memset()`?

Share

Improve this answer

Follow

edited Feb 10, 2022 at 15:12

Jonathan Leffler

752k ● 145 ● 946 ● 1.3k

answered Apr 22, 2010 at 5:54

Dietrich Epp

213k ● 38 ● 359 ● 422

7 @Dietrich: the virtual memory explaination of Dietrich about OS allocating the same zero filled page many times for calloc is easy to check. Just add some loop that write junk data in every allocated memory page (writing one byte every 500 bytes should be enough). The overall result should then become much closer as system would be forced to really allocate differents pages in both cases. – kriss Apr 22, 2010 at 6:43

1 @kriss: indeed, although one byte every 4096 is sufficient on the vast majority of systems – Dietrich Epp Apr 22, 2010 at 6:46

1 @mirabilos: Actually, implementations tend to be even more sophisticated. Memory allocated by `mmap()` is allocated in large chunks, so the `malloc()` / `calloc()` implementation may keep track of what blocks are still pristine and full of zeroes. So `calloc()` can avoid touching memory even if it doesn't get the memory from `mmap()`, i.e., it was already part of the heap but hasn't been used yet. – Dietrich Epp Mar 31, 2014 at 20:49

1 @mirabilos: I've also seen implementations with a "high water mark", where addresses beyond a certain point are zeroed. I'm not sure what you mean by "error-prone"—if you are worried about applications writing to unallocated memory, then there is very little you can do to prevent insidious errors, short of instrumenting the program with mudflap. – Dietrich Epp Mar 31, 2014 at 21:24

11 Whilst not speed related, `calloc` is also less bug prone. That is, where `large_int * large_int` would result in an overflow, `calloc(large_int, large_int)` returns `NULL`, but `malloc(large_int * large_int)` is undefined behaviour, as you don't know the actual size of the memory block being returned. – Dunes Mar 23, 2018 at 9:41

17

Because on many systems, in spare processing time, the OS goes around setting free memory to zero on its own and marking it safe for `calloc()`, so when you call `calloc()`, it may already have free, zeroed memory to give you.

Share Improve this answer Follow

answered Apr 22, 2010 at 5:48

Chris Lutz

75.3k ● 16 ● 131 ● 184

2 Are you sure? Which systems do this? I thought that most OSs just shut down the processor when they were idle, and zeroed memory on demand for the processes that allocated as soon as they write to that memory (but not when they allocate it). – Dietrich Epp Apr 22, 2010 at 6:00

@Dietrich - Not sure. I heard it once and it seemed like a reasonable (and reasonably simple) way to make `calloc()` more efficient. – Chris Lutz Apr 22, 2010 at 6:06

@Pierreten - I can't find any good info on `calloc()`-specific optimizations and I don't feel like interpreting libc source code for the OP. Can you look up anything to show that this optimization doesn't exist / doesn't work? – Chris Lutz Apr 22, 2010 at 6:13

17 @Dietrich: FreeBSD is supposed to zero-fill pages in idle time: See its vm.idlezero_enable setting. – Zan Lynx Mar 7, 2011 at 21:47

1 @DietrichEpp sorry to necro, but for example Windows does this. – Andreas Grapentin Nov 11, 2014 at 19:37

3

On some platforms in some modes malloc initialises the memory to some typically non-zero value before returning it, so the second version could well initialize the memory twice

Share Improve this answer Follow

answered Apr 22, 2010 at 5:51

Stewart
4,028 ● 18 ● 20

`malloc()` doesn't initialize the RAM. It gives you a pointer to the newly-allocated RAM with whatever garbage data was in it before. `calloc()` allocates and initializes it to 0. – Felix An Nov 19, 2022 at 6:25

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.