C dynamic memory allocation

From Wikipedia, the free encyclopedia

C dynamic memory allocation refers to performing dynamic memory allocation in the C programming language via a group of functions in the C standard library, namely `malloc`, `realloc`, `calloc` and `free`.[1][2][3]

The C++ programming language includes these functions for backwards compatibility; its use in C++ has been largely superseded by operators `new` and `new[]`.[4]

Many different implementations of the actual memory allocation mechanism, used by `malloc`, are available. Their performance varies in both execution time and required memory.

Contents

Rationale

The C programming language manages memory statically, automatically, or dynamically. Static-duration variables are allocated in main memory, usually along with the executable code of the program, and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and automatic-duration variables, the size of the allocation is required to be compile-time constant (before C99, which allows variable-

length automatic arrays[5]). If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

The lifetime of allocated memory is also a concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the *heap*, an area of memory structured for this purpose. In C, the library function `malloc` is used to allocate a block of memory on the heap. The program accesses this block of memory via a pointer that `malloc` returns. When the memory is no longer needed, the pointer is passed to ***`free`*** which deallocates the memory so that it can be used for other purposes.

Some platforms provide library calls which allow run-time dynamic allocation from the C stack rather than the heap (e.g. Unix `alloca()`,[6] Microsoft Windows CRTL's `malloca()`[7]). This memory is automatically freed when the calling function ends. The need for this is lessened by changes in the C99 standard, which added support for variable-length arrays of block scope having sizes determined at runtime.

Overview of functions

The C dynamic memory allocation functions are defined in `stdlib.h` header (`cstdlib` header in C++).[1]

Function	Description
`malloc` (http://en.cppreference.com/w/c/memory/malloc)	allocates the specified number of bytes
`realloc` (http://en.cppreference.com/w/c/memory/realloc)	increases or decrease the size of the specified block of memory. Reallocates it if needed
`calloc` (http://en.cppreference.com/w/c/memory/calloc)	allocates the specified number of bytes and initializes them to zero
`free` (http://en.cppreference.com/w/c/memory/free)	releases the specified block of memory back to the system

Usage example

The standard method of creating an array of 10 int objects:

```
int array[10];
```

However, if one wishes to allocate a similar array dynamically, the following code could

be used:

```
/* Allocate space for an array with ten elements of type int. */
int *ptr = malloc(10 * sizeof(int));
if (NULL == ptr) {
    /* Memory could not be allocated. The program should handle the
     error here as appropriate. */

} else {
    /* Allocation succeeded. Do something with it... */
    /* We are done with the array of ints, and can free the block of
     memory */
    free(ptr);
    /* The pointed-to address must not be used again, unless
     re-assigned by another call to malloc. */
      ptr = NULL;

}
```

`malloc` returns a null pointer to indicate that no memory is available, or that some other error occurred which prevented memory being allocated.

Type safety

`malloc` returns a void pointer (`void *`), which indicates that it is a pointer to a region of unknown data type. The use of casting is only required in C++ due to the strong type system, whereas this is not the case in C. The lack of a specific pointer type returned from `malloc` is type-unsafe behaviour according to some programmers: `malloc` allocates based on byte count but not on type. This is different from the C++ new operator that returns a pointer whose type relies on the operand. (see C Type Safety).

One may "cast" (see type conversion) this pointer to a specific type:

```
int *ptr;
ptr = malloc(10 * sizeof (*ptr));              /* without a cast */
ptr = (int *)malloc(10 * sizeof (*ptr));       /* with a cast */
```

There are advantages and disadvantages to performing such a cast.

Advantages to casting

- C++ does require the cast. Including the cast allows a program (or a header file included in a program) to be both valid C and valid C++.
- The cast allows for older versions of `malloc` that originally returned a `char *`.[8]

Disadvantages to casting

- Under the ANSI C standard, the cast is redundant.

- Adding the cast may mask failure to include the header `stdlib.h`, in which the prototype for `malloc` is found.[8][9] In the absence of a prototype for `malloc`, the standard requires that the C compiler assume `malloc` returns an `int`. If there is no cast, a warning is issued when this integer is assigned to the pointer; however, with the cast, this warning is not produced, hiding a bug. On certain architectures and data models (such as LP64 on 64-bit systems, where `long` and pointers are 64-bit and `int` is 32-bit), this error can actually result in undefined behaviour, as the implicitly declared `malloc` returns a 32-bit value whereas the actually defined function returns a 64-bit value. Depending on calling conventions and memory layout, this may result in stack smashing. This issue is not present in modern compilers, as they uniformly produce warnings that an undeclared function has been used, so a warning will still appear. For example, GCC's default behaviour is to show a warning that reads "incompatible implicit declaration of built-in function" regardless of whether the cast is present or not.
- If the type of the pointer is changed, one must fix all code lines where `malloc` was called and cast (unless it was cast to a `typedef`).

Common errors

The improper use of dynamic memory allocation can frequently be a source of bugs.

Most common errors are as follows:

- **Not checking for allocation failures**. Memory allocation is not guaranteed to succeed. If there's no check for successful allocation implemented, this usually leads to a crash of the program or the entire system.
- **Memory leaks**. Failure to deallocate memory using `free` leads to buildup of memory that is non-reusable memory, which is no longer used by the program. This wastes memory resources and can lead to allocation failures when these resources are exhausted.
- **Logical errors**. All allocations must follow the same pattern: allocation using `malloc`, usage to store data, deallocation using `free`. Failures to adhere to this pattern, such as memory usage after a call to `free` or before a call to `malloc`, calling `free` twice ("double free"), etc., usually leads to a crash of the program.

Implementations

The implementation of memory management depends greatly upon operating system and architecture. Some operating systems supply an allocator for malloc, while others supply functions to control certain regions of data. The same dynamic memory allocator is often used to implement both `malloc` and the operator `new` in C++[*citation needed*]. Hence, it is referred to below as the *allocator* rather than `malloc`.

Heap-based

Implementation of the allocator on IA-32 architectures is commonly done using the heap, or data segment. The allocator will usually expand and contract the heap to fulfill

allocation requests.

The heap method suffers from a few inherent flaws, stemming entirely from fragmentation. Like any method of memory allocation, the heap will become fragmented; that is, there will be sections of used and unused memory in the allocated space on the heap. A good allocator will attempt to find an unused area of already allocated memory to use before resorting to expanding the heap. The major problem with this method is that the heap has only two significant attributes: base, or the beginning of the heap in virtual memory space; and length, or its size. The heap requires enough system memory to fill its entire length, and its base can never change. Thus, any large areas of unused memory are wasted. The heap can get "stuck" in this position if a small used segment exists at the end of the heap, which could waste any magnitude of address space, from a few megabytes to a few hundred.

dlmalloc

Doug Lea has developed dlmalloc (ftp://g.oswego.edu/pub/misc/) ("Doug Lea's Malloc") as a general-purpose allocator, starting in 1987. The GNU C library (glibc) uses an allocator based on dlmalloc.[10]

Memory on the heap is allocated as "chunks", an 8-byte aligned data structure which contains a header and usable memory. Allocated memory contains an 8 or 16 byte overhead for the size of the chunk and usage flags. Unallocated chunks also store pointers to other free chunks in the usable space area, making the minimum chunk size 24 bytes.[10]

Unallocated memory is grouped into "bins" of similar sizes, implemented by using a double-linked list of chunks (with pointers stored in the unallocated space inside the chunk).[10]

For requests below 256 bytes (a "smallbin" request), a simple two power best fit allocator is used. If there are no free blocks in that bin, a block from the next highest bin is split in two.

For requests of 256 bytes or above but below the mmap threshold, recent versions of dlmalloc use an in-place *bitwise trie* algorithm. If there is no free space left to satisfy the request, dlmalloc tries to increase the size of the heap, usually via brk system call.

For requests above the mmap threshold (a "largebin" request), the memory is always allocated using the mmap system call. The threshold is usually 256 KB.[11] The mmap method averts problems with huge buffers trapping a small allocation at the end after their expiration, but always allocates an entire page of memory, which on many architectures is 4096 bytes in size.[12]

FreeBSD's and NetBSD's jemalloc

Since FreeBSD 7.0 and NetBSD 5.0, the old `malloc` implementation (phkmalloc) was replaced by jemalloc (http://www.canonware.com/jemalloc/) , written by Jason Evans. The main reason for this was a lack of scalability of phkmalloc in terms of

multithreading. In order to avoid lock contention, jemalloc uses separate "arenas" for each CPU. Experiments measuring number of allocations per second in multithreading application have shown that this makes it scale linearly with the number of threads, while for both phkmalloc and dlmalloc performance was inversely proportional to the number of threads.[13]

OpenBSD's malloc

OpenBSD's implementation of the `malloc` function makes use of `mmap`. For requests greater in size than one page, the entire allocation is retrieved using `mmap`; smaller sizes are assigned from memory pools maintained by `malloc` within a number of "bucket pages," also allocated with `mmap`. On a call to `free`, memory is released and unmapped from the process address space using `munmap`. This system is designed to improve security by taking advantage of the address space layout randomization and gap page features implemented as part of OpenBSD's `mmap` system call, and to detect use-after-free bugs—as a large memory allocation is completely unmapped after it is freed, further use causes a segmentation fault and termination of the program.

Hoard's malloc

The Hoard memory allocator is an allocator whose goal is scalable memory allocation performance. Like OpenBSD's allocator, Hoard uses `mmap` exclusively, but manages memory in chunks of 64 kilobytes called superblocks. Hoard's heap is logically divided into a single global heap and a number of per-processor heaps. In addition, there is a thread-local cache that can hold a limited number of superblocks. By allocating only from superblocks on the local per-thread or per-processor heap, and moving mostly-empty superblocks to the global heap so they can be reused by other processors, Hoard keeps fragmentation low while achieving near linear scalability with the number of threads.[14]

Thread-caching malloc (tcmalloc)

Every thread has local storage for small allocations. For large allocations mmap or sbrk can be used. TCMalloc, a *malloc* developed by Google,[15] has garbage-collection for local storage of dead threads. The TCMalloc is considered to be more than twice as fast as glibc's ptmalloc for multithreaded programs.[16][17]

In-kernel

Operating system kernels need to allocate memory just as application programs do. The implementation of `malloc` within a kernel often differs significantly from the implementations used by C libraries, however. For example, memory buffers might need to conform to special restrictions imposed by DMA, or the memory allocation function might be called from interrupt context.[18] This necessitates a `malloc` implementation tightly integrated with the virtual memory subsystem of the operating system kernel.

Allocation size limits

The largest possible memory block `malloc` can allocate depends on the host system, particularly the size of physical memory and the operating system implementation. Theoretically, the largest number should be the maximum value that can be held in a `size_t` type, which is an implementation-dependent unsigned integer representing the size of an area of memory. The maximum value is $2^{\text{CHAR\_BIT} \times \text{sizeof(size\_t)}} - 1$, or the constant `SIZE_MAX` in the C99 standard.

See also

- Buffer overflow
- Memory debugger
- `mprotect`
- `new` (C++)
- Page size
- Variable-length array

References

1. ^ ***a*** ***b*** *ISO/IEC 9899:1999 specification* (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf) . p. 313, § 7.20.3 "Memory management functions". http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.
2. ^ Godse, Atul P.; Godse, Deepali A. (2008). *Advanced C Programming*. p. 6-28: Technical Publications. pp. 400. ISBN 978-81-8431-496-0.
3. ^ Summit, Steve. "C Programming Notes - Chapter 11: Memory Allocation" (http://c-faq.com/~scs/cclass/notes/sx11.html) . http://c-faq.com/~scs/cclass/notes/sx11.html. Retrieved 30 October 2011.
4. ^ Stroustrup, Bjarne (2008). *Programming: Principles and Practice Using C++*. 1009, §27.4 *Free store*: Addison Wesley. pp. 1236. ISBN 978-0-321-54372-1.
5. ^ "gcc manual" (http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html) . gnu.org. http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html. Retrieved 14 December 2008.
6. ^ "alloca" (http://man.freebsd.org/alloca) . *Man.freebsd.org*. 5 September 2006. http://man.freebsd.org/alloca. Retrieved 18 September 2011.
7. ^ "`malloca()`" (http://msdn.microsoft.com/en-us/library/5471dc8s.aspx) . MSDN Visual C++ Developer Center. http://msdn.microsoft.com/en-us/library/5471dc8s.aspx. Retrieved 12 March 2009.
8. ^ ***a*** ***b*** "Casting malloc" (http://faq.cprogramming.com/cgi-bin/smartfaq.cgi?id=1043284351&answer=1047673478) . Cprogramming.com. http://faq.cprogramming.com/cgi-bin/smartfaq.cgi?id=1043284351&answer=1047673478. Retrieved 9 March 2007.
9. ^ comp.lang.c "FAQ list · Question 7.7b" (http://www.c-faq.com/malloc/mallocnocast.html) . C-FAQ. http://www.c-faq.com/malloc/mallocnocast.html comp.lang.c. Retrieved 9 March 2007.
10. ^ ***a*** ***b*** ***c*** Kaempf, Michel (2001). "Vudo malloc tricks" (http://phrack.org/issues.html?issue=57&id=8&mode=txt) . *Phrack* (57): 8. http://phrack.org/issues.html?issue=57&id=8&mode=txt. Retrieved 29 April 2009.
11. ^ "Malloc Tunable Parameters" (http://www.gnu.org/software/libc/manual/html_node

/Malloc-Tunable-Parameters.html) . GNU. http://www.gnu.org/software/libc/manual /html_node/Malloc-Tunable-Parameters.html. Retrieved 2 May 2009.
12. ^ Sanderson, Bruce (12 December 2004). "RAM, Virtual Memory, Pagefile and all that stuff" (http://support.microsoft.com/kb/555223) . Microsoft Help and Support. http://support.microsoft.com/kb/555223.
13. ^ Evans, Jason (16 April 2006). "A Scalable Concurrent malloc(3) Implementation for FreeBSD" (http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf) . http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf. Retrieved 18 March 2012.
14. ^ Berger, Emery D.; McKinley, Kathryn S.; Blumofe, Robert D.; Wilson, Paul R. (2000). "Hoard: A Scalable Memory Allocator for Multithreaded Applications" (http://www.cs.umass.edu/~emery/pubs/berger-asplos2000.pdf) . http://www.cs.umass.edu /~emery/pubs/berger-asplos2000.pdf. Retrieved 18 March 2012.
15. ^ TCMalloc homepage (//code.google.com/p/gperftools/)
16. ^ Ghemawat, Sanjay; Menage, Paul; *TCMalloc : Thread-Caching Malloc* (http://goog-perftools.sourceforge.net/doc/tcmalloc.html)
17. ^ Callaghan, Mark (18 January 2009). "High Availability MySQL: Double sysbench throughput with TCMalloc" (http://mysqlha.blogspot.com/2009/01/double-sysbench-throughput-with_18.html) . *Mysqlha.blogspot.com*. http://mysqlha.blogspot.com/2009/01 /double-sysbench-throughput-with_18.html. Retrieved 18 September 2011.
18. ^ "kmalloc()/kfree() include/linux/slab.h" (http://people.netfilter.org/~rusty/unreliable-guides/kernel-hacking/routines-kmalloc.html) . *People.netfilter.org*. http://people.netfilter.org /~rusty/unreliable-guides/kernel-hacking/routines-kmalloc.html. Retrieved 18 September 2011.

External links

- Definition of malloc in IEEE Std 1003.1 standard (http://www.opengroup.org /onlinepubs/9699919799/functions/malloc.html)
- Lea, Doug; *The design of the basis of the glibc allocator* (http://gee.cs.oswego.edu /dl/html/malloc.html)
- Gloger, Wolfram; *The ptmalloc homepage* (http://www.malloc.de/en/index.html)
- Berger, Emery; *The Hoard homepage* (http://www.hoard.org)
- Douglas, Niall; *The nedmalloc homepage* (http://www.nedprod.com/programs /portable/nedmalloc/)
- Evans, Jason; *The jemalloc homepage* (http://www.canonware.com/jemalloc/)
- *Simple Memory Allocation Algorithms* (http://www.osdcom.info/content /view/31/39/) on OSDEV Community
- Berger, Emery; *Hoard: A Scalable Memory Allocator for Multithreaded Applications* (http://www.cs.umass.edu/~emery/pubs/berger-asplos2000.pdf)
- Michael, Maged M.; *Scalable Lock-Free Dynamic Memory Allocation* (http://www.research.ibm.com/people/m/michael/pldi-2004.pdf)
- Bartlett, Jonathan; *Inside memory management* - The choices, tradeoffs, and implementations of dynamic allocation (http://www-106.ibm.com/developerworks /linux/library/l-memory/)
- Memory Reduction (GNOME) (http://live.gnome.org/MemoryReduction) wiki page with lots of information about fixing malloc
- C99 standard draft, including TC1/TC2/TC3 (http://www.open-std.org/jtc1/sc22 /wg14/www/docs/n1256.pdf)

- Some useful references about C (http://paste.tclers.tk/1596)
- ISO/IEC 9899 – Programming languages – C (http://www.open-std.org/jtc1/sc22/wg14/www/standards)

Retrieved from "http://en.wikipedia.org/w/index.php?title=C_dynamic_memory_allocation&oldid=508343988"
Categories: Memory management | Memory management software | C standard library

---