Contents

1. Overview of Malloc
2. What is a Chunk?
3. Arenas and Heaps
4. Thread Local Cache (tcache)
5. Malloc Algorithm
6. Free Algorithm
7. Realloc Algorithm
8. Switching arenas
9. Detecting heap corruption
10. Platform-specific Thresholds and Constants
11. TBD
12. Colophon

One Heap to malloc them all, One Heap to free them, One Heap to coalesce, and in the memory bind them...

Overview of Malloc

The GNU C library's (glibc's) malloc library contains a handful of functions that manage allocated memory in the application's address space. The glibc malloc is derived from ptmalloc (pthreads malloc), which is derived from dlmalloc (Doug Lea malloc). This malloc is a "heap" style malloc, which means that chunks of various sizes exist within a larger region of memory (a "heap") as opposed to, for example, an implementation that uses bitmaps and arrays, or regions of same-sized blocks, etc. In ancient times, there was one heap per application, but glibc's malloc allows for multiple heaps, each of which grows within its address space.

Thus, within this document, we refer to some common terms:

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.

Note that, in this document, we only refer to "memory" as a generic term. While there is some code in glibc's malloc to work with the Linux kernel (or other OS) to hint at what memory should be mapped/backed and what can be returned to the kernel, such discrimination between "real memory" and "virtual memory" is irrelevant to the discussion herein, unless explicitly called out.

What is a Chunk?

Glibc's malloc is chunk-oriented. It divides a large region of memory (a "heap") into chunks of various sizes. Each chunk includes meta-data about how big it is (via a `size` field in the chunk header), and thus where the adjacent chunks are. When a chunk is in use by the application, the only data that's "remembered" is the size of the chunk. When the chunk is free'd, the memory that used to be application data is re-purposed for additional arena-related information, such as pointers within linked lists, such that suitable chunks can quickly be found and re-used when needed. Also, the last word in a free'd chunk contains a copy of the chunk size (with the three LSBs set to zeros, vs the three LSBs of the size at the front of the chunk which are used for flags).

Within the malloc library, a "chunk pointer" or *mchunkptr* does *not* point to the beginning of the chunk, but to the last word in the previous chunk - i.e. the first field in `mchunkptr` is not valid unless you know the previous chunk is free.

Since all chunks are multiples of 8 bytes, the 3 LSBs of the chunk size can be used for flags. These three flags are defined as follows:

A (0x04)
: Allocated Arena - the main arena uses the application's heap. Other arenas use mmap'd heaps. To map a chunk to a heap, you need to know which case applies. If this bit is 0, the chunk comes from the main arena and the main heap. If this bit is 1, the chunk comes from mmap'd memory and the location of the heap can be computed from the chunk's address.

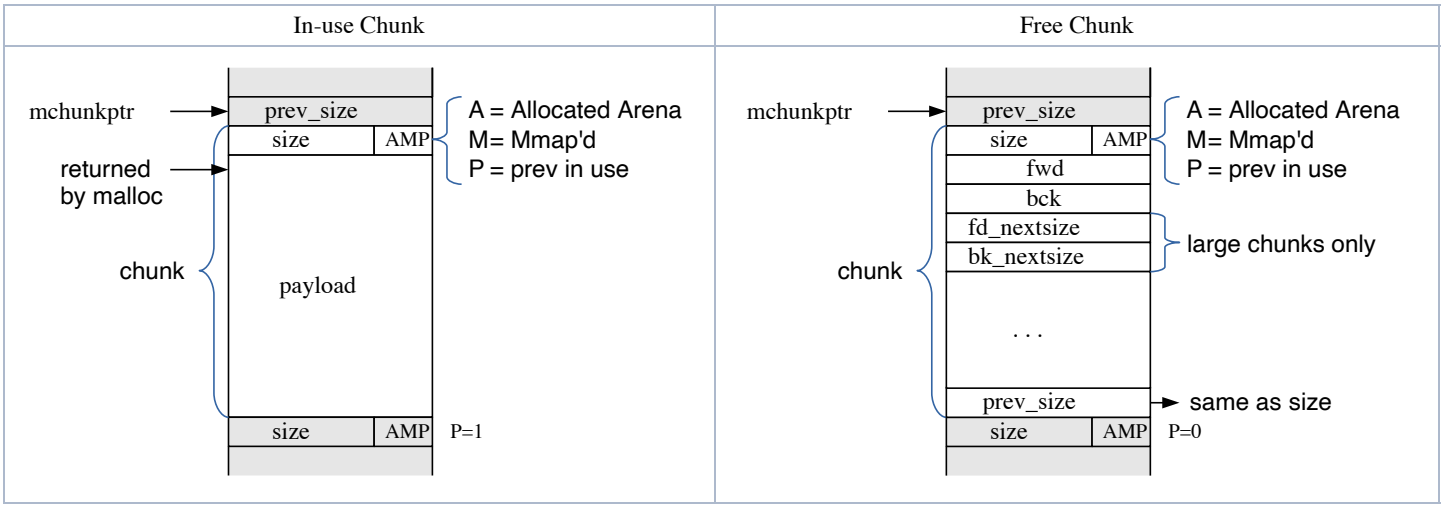
M (0x02)

MMap'd chunk - this chunk was allocated with a single call to `mmap` and is not part of a heap at all.

P (0x01)

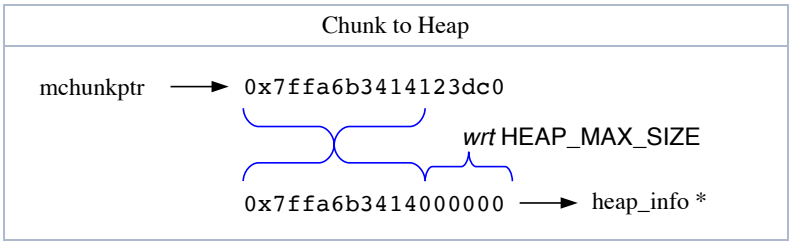
Previous chunk is in use - if set, the previous chunk is still being used by the application, and thus the `prev_size` field is invalid. Note - some chunks, such as those in fastbins (see below) will have this bit set despite being free'd by the application. This bit really means that the previous chunk should not be considered a candidate for coalescing - it's "in use" by either the application or some other optimization layered atop malloc's original code

In order to ensure that a chunk's payload area is large enough to hold the overhead needed by malloc, the minimum size of a chunk is `4*sizeof(void*)` (unless `size_t` is not the same size as `void*`). The minimum size may be larger if the ABI of the platform requires additional alignment. Note that `prev_size` does not increase the minimum chunk size to `5*sizeof(void*)` because when the chunk is small the `bk_nextsize` pointer is unused, and when the chunk is large enough to use it there is more than enough space at the end.



Note that, since chunks are adjacent to each other in memory, if you know the address of the first chunk (lowest address) in a heap, you can iterate through all the chunks in the heap by using the size information, but only by increasing address, although it may be difficult to detect when you've hit the last chunk in the heap.

Allocated heaps are always aligned to a power-of-two address. Thus, when a chunk is in an allocated heap (i.e. the A bit is set), the address of the `heap_info` for that heap can be computed based on the address of the chunk:



Arenas and Heaps

In order to efficiently handle multi-threaded applications, glibc's malloc allows for more than one region of memory to be active at a time. Thus, different threads can access different regions of memory without interfering with each other. These regions of memory are collectively called "arenas". There is one arena, the "main arena", that corresponds to the application's initial heap. There's a static variable in the malloc code that points to this arena, and each arena has a `next` pointer to link additional arenas.

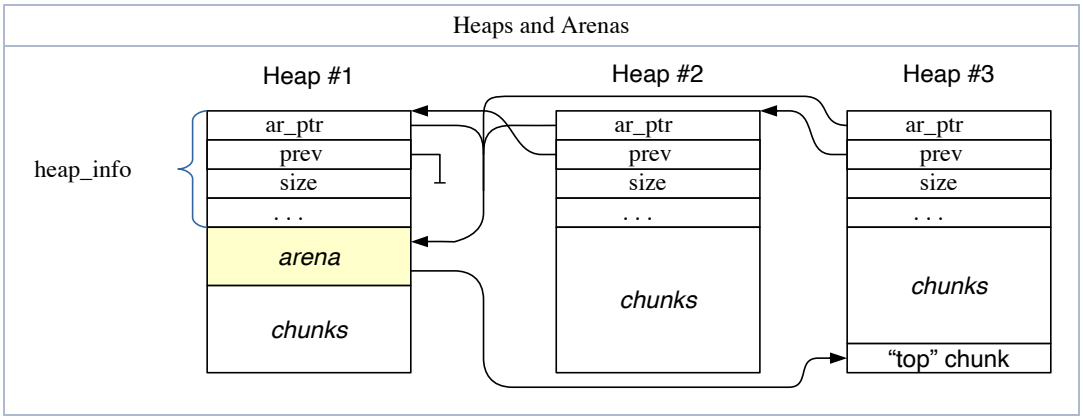
As pressure from thread collisions increases, additional arenas are created via `mmap` to relieve the pressure. The number of arenas is capped at eight times the number of CPUs in the system (unless the user specifies otherwise, see `mallopt`), which means a heavily threaded application will still see some contention, but the trade-off is that there will be less fragmentation.

Each arena structure has a mutex in it which is used to control access to that arena. Note that some operations, such as access to the fastbins, can be done with atomic operations and do not need to lock the arena. All other operations require that the thread take a lock on the arena. Contention for this mutex is the reason why multiple arenas are created - threads assigned to different arenas need not wait for each other.

Threads will automatically switch to unused (unlocked) arenas if contention requires it.

Each arena obtains memory from one or more heaps. The main arena uses the program's initial heap (starting right after .bss et al). Additional arenas allocate memory for their heaps via mmap, adding more heaps to their list of heaps as older heaps are used up. Each arena keeps track of a special "top" chunk, which is typically the biggest available chunk, and also refers to the most recently allocated heap.

Memory for allocated arenas is, conveniently, taken from the initial heap for that arena:



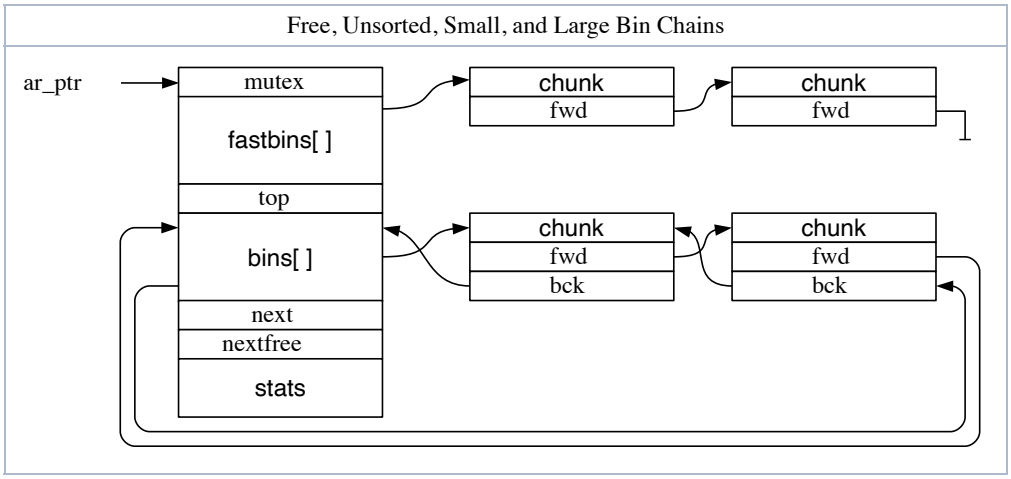
Within each arena, chunks are either in use by the application or they're free (available). In-use chunks are not tracked by the arena. Free chunks are stored in various lists based on size and history, so that the library can quickly find suitable chunks to satisfy allocation requests. The lists, called "bins", are:

Fast
: Small chunks are stored in size-specific bins. Chunks added to a fast bin ("fastbin") are not combined with adjacent chunks - the logic is minimal to keep access fast (hence the name). Chunks in the fastbins may be moved to other bins as needed. Fastbin chunks are stored in an array of singly-linked lists, since they're all the same size and chunks in the middle of the list need never be accessed.

Unsorted
: When chunks are free'd they're initially stored in a single bin. They're sorted later, in malloc, in order to give them one chance to be quickly re-used. This also means that the sorting logic only needs to exist at one point - everyone else just puts free'd chunks into this bin, and they'll get sorted later. The "unsorted" bin is simply the first of the regular bins.

Small
: The normal bins are divided into "small" bins, where each chunk is the same size, and "large" bins, where chunks are a range of sizes. When a chunk is added to these bins, they're first combined with adjacent chunks to "coalesce" them into larger chunks. Thus, these chunks are never adjacent to other such chunks (although they may be adjacent to fast or unsorted chunks, and of course in-use chunks). Small and large chunks are doubly-linked so that chunks may be removed from the middle (such as when they're combined with newly free'd chunks).

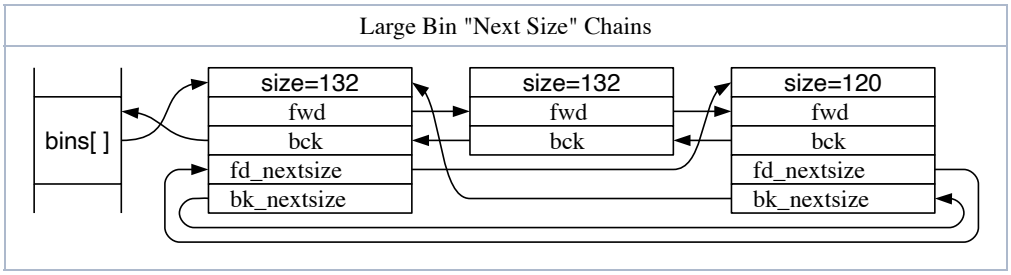
Large
: A chunk is "large" if its bin may contain more than one size. For small bins, you can pick the first chunk and just use it. For large bins, you have to find the "best" chunk, and possibly split it into two chunks (one the size you need, and one for the remainder).



Note: in the diagram above (and below), all the pointers are to a "chunk" (`mchunkptr`). Since the bins are not chunks (they're arrays of `fwd`/`bck` pointers), a hack is used to provide an `mchunkptr` to a chunk-like object which overlaps the bins "just right" so that the

`mchunkptr`'s `fwd` and `bck` fields can be used to access the appropriate bin.

Because of the need to find the "best" fit for large chunks, large chunks have an additional doubly-linked list linking the first of each size in the list, and chunks are sorted by size, largest through smallest. This allows malloc to quickly scan for the first chunk that's big enough. Note: if multiple chunks of a given size are present, the *second* one is typically the chosen one, so that the next-size linked list need not be adjusted. Chunks inserted into the list are added after a chunk of the same size, for the same reason. Note that the *_nextsize pointers may be NULL, and may point to its own chunk forming a loop of one, in addition to larger loops.

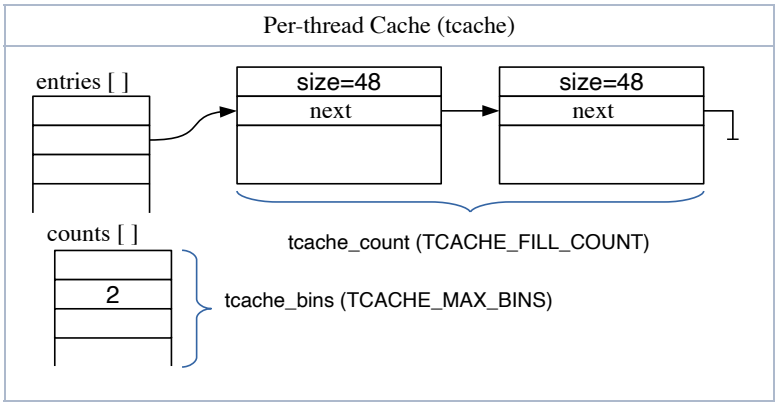


Thread Local Cache (tcache)

While this malloc is aware of multiple threads, that's pretty much the extent of its awareness - it knows there are multiple threads. There is no code in this malloc to optimize it for NUMA architectures, coordinate thread locality, sort threads by core, etc. It is assumed that the kernel will handle those issues sufficiently well.

Each thread has a thread-local variable that remembers which arena it last used. If that arena is in use when a thread needs to use it the thread will block to wait for the arena to become free. If the thread has never used an arena before then it may try to reuse an unused one, create a new one, or pick the next one on the global list.

Each thread has a per-thread cache (called the *tcache*) containing a small collection of chunks which can be accessed without needing to lock an arena. These chunks are stored as an array of singly-linked lists, like fastbins, but with links pointing to the payload (user area) not the chunk header. Each bin contains one size chunk, so the array is indexed (indirectly) by chunk size. Unlike fastbins, the tcache is limited in how many chunks are allowed in each bin (`tcache_count`). If the tcache bin is empty for a given requested size, the next larger sized chunk is not used (could cause internal fragmentation), instead the fallback is to use the normal malloc routines i.e. locking the thread's arena and working from there.



Malloc Algorithm

In a nutshell, malloc works like this:

- If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller. No attempt is made to use an available chunk from a larger-sized bin.
- If the request is large enough, `mmap()` is used to request memory directly from the operating system. Note that the threshold for mmap'ing is dynamic, unless overridden by M_MMAP_THRESHOLD (see mallopt() documentation), and there may be a limit to how many such mappings there can be at one time.
- If the appropriate fastbin has a chunk in it, use that. If additional chunks are available, also pre-fill the tcache.
- If the appropriate smallbin has a chunk in it, use that, possibly pre-filling the tcache here also.

- If the request is "large", take a moment to take everything in the fastbins and move them to the unsorted bin, coalescing them as you go.
- Start taking chunks off the unsorted list, and moving them to small/large bins, coalescing as you go (note that this is the only place in the code that puts chunks into the small/large bins). If a chunk of the right size is seen, use that.
- If the request is "large", search the appropriate large bin, and successively larger bins, until a large-enough chunk is found.
- If we still have chunks in the fastbins (this may happen for "small" requests), consolidate those and repeat the previous two steps.
- Split off part of the "top" chunk, possibly enlarging "top" beforehand.

For an over-aligned malloc, such as `valloc`, `pvalloc`, or `memalign`, an overly-large chunk is located (using the malloc algorithm above) and divided into two or more chunks in such a way that most of the chunk is now suitably aligned (and returned to the caller), and the excess before and after that portion is returned to the unsorted list to be re-used later.

Free Algorithm

Note that, in general, "freeing" memory does not actually return it to the operating system for other applications to use. The `free()` call marks a chunk of memory as "free to be reused" by the application, but from the operating system's point of view, the memory still "belongs" to the application. However, if the top chunk in a heap - the portion adjacent to unmapped memory - becomes large enough, some of that memory may be unmapped and returned to the operating system.

In a nutshell, free works like this:

- If there is room in the tcache, store the chunk there and return.
- If the chunk is small enough, place it in the appropriate fastbin.
- If the chunk was mmap'd, munmap it.
- See if this chunk is adjacent to another free chunk and coalesce if it is.
- Place the chunk in the unsorted list, unless it's now the "top" chunk.
- If the chunk is large enough, coalesce any fastbins and see if the top chunk is large enough to give some memory back to the system. Note that this step might be deferred, for performance reasons, and happen during a malloc or other call.

Realloc Algorithm

Note that realloc of NULL and realloc to zero size are handled separately and as per the relevant specs.

In a nutshell, realloc works like this:

For MMAP'd chunks...

Allocations that are serviced via individual `mmap` calls (i.e. large ones) are realloc'd by `mremap()` if available, which may or may not result in the new memory being at a different address than the old memory, depending on what the kernel does.

If the system does not support `munmap()` and the new size is smaller than the old size, nothing happens and the old address is returned, else a malloc-copy-free happens.

For arena chunks...

- If the size of the allocation is being reduced by enough to be "worth it", the chunk is split into two chunks. The first half (which has the old address) is returned, and the second half is returned to the arena as a free chunk. Slight reductions are treated as "the same size".
- If the allocation is growing, the next (adjacent) chunk is checked. If it is free, or the "top" block (representing the expandable part of the heap), and large enough, then that chunk and the current are merged, producing a large-enough block which can be possibly split (as above). In this case, the old pointer is returned.
- If the allocation is growing and there's no way to use the existing/following chunk, then a malloc-copy-free sequence is used.

Switching arenas

The arena to which a thread is attached is generally viewed as an invariant that does not change over the lifetime of the process. This invariant, while useful for explaining general concepts, is not true. The scenario for changing arenas looks like this:

- Thread fails to allocate memory from the attached arena.
  - Assumes we tried coalescing, searched free list, processed unsorted list etc.
  - Assumes we tried to expand the heap but either the `sbrk` failed or creating the new mapping failed.
- If previously using a non-main arena with `mmap`'d heaps the thread is switched via `arena_get_retry` to the main arena with an `sbrk`-based heap, or switched to non-main arena (from free list or a new one) if previously using the main arena. As an optimization this is not done if the process is single threaded, and we fail at this point returning `ENOMEM` (it is assumed that if `sbrk` did not work, and we tried `mmap` to extend the main arena, that a non-main arena will not work either).

Note that a thread may change frequently between arenas in low-memory conditions, switching from main-arena which `sbrk`-based to a non-main arena which is `mmap`-based, all in an attempt to find a heap with enough space to satisfy the allocation.

Detecting heap corruption

The malloc subsystem undertakes reasonable attempts to detect heap corruption throughout the code. Some checks detect errors consistently (e.g., passing a pointer which is not sufficiently aligned to `free`). However, most checks are heuristic and can be fooled by completely fake chunks that look real (e.g., checks forward/backward linking of chunks). If that happens, heap corruption may go unnoticed for some time, or might not be reported at all.

The common forms of corruption are handled with calls to `malloc_printerr`; these checks are always included in the code. Further checks use `assert` and are therefore disabled by building glibc with `-DNDEBUG`. In current glibc, both kinds of checks terminate the process via a call to `__libc_messsage`, which eventually calls `abort`. Very old versions of glibc supported continuing in the present of heap corruption, but support for that has been removed.

Platform-specific Thresholds and Constants

These values are provided for entertainment purposes only, and are not guarateed to be correct, but they should have been at one point, at least in general.

Parameter	32 bit	i386	64 bit
MALLOC_ALIGNMENT	8	16	16
MIN_CHUNK_SIZE	16	16	32
MAX_FAST_SIZE	80	80	160
MAX_TCACHE_SIZE	516	1,020	1,032
MIN_LARGE_SIZE	512	1,008	1,024
DEFAULT_MMAP_THRESHOLD	131,072	131,072	131,072
DEFAULT_MMAP_THRESHOLD_MAX	524,288	524,288	33,554,432
HEAP_MIN_SIZE	32,768	32,768	32,768
HEAP_MAX_SIZE	1,048,576	1,048,576	67,108,864

TBD

- Tunables
- Remainder of API calls

Colophon

The images in this document were created with LibreOffice Draw. For each *.svg file, there is a *.odg file which is the source.

None: MallocInternals (last edited 2022-08-09 17:51:50 by DJ Delorie)