

Memory, Part 2: Implementing a Memory Allocator

Vangelis Tsiatsianas edited this page on Feb 23 · 4 revisions

A memory allocator needs to keep track of which bytes are currently allocated and which are available for use. This page introduces the implementation and conceptual details of building an allocator, i.e. the actual code that implements `malloc` and `free`.

This page talks about links of blocks - do I malloc memory for them instead?

Though conceptually we are thinking about creating linked lists and lists of blocks, we don't need to "malloc memory" to create them! Instead we are writing integers and pointers into memory that we already control so that we can later consistently hop from one address to the next. This internal information represents some overhead. So even if we had requested 1024 KB of contiguous memory from the system, we will not be able to provide all of it to the running program.

Thinking in blocks

We can think of our heap memory as a list of blocks where each block is either allocated or unallocated. Rather than storing an explicit list of pointers we store information about the block's size *as part of the block*. Thus there is conceptually a list of free blocks, but it is implicit, i.e. in the form of block size information that we store as part of each block.

We could navigate from one block to the next block just by adding the block's size. For example if you have a pointer `p` that points to the start of a block, then `next_block` will be at `((char *)p) + *(size_t *) p`, if you are storing the size of the blocks in bytes. The cast to `char *` ensures that pointer arithmetic is calculated in bytes. The cast to `size_t *` ensures the memory at `p` is read as a size value and would be necessary if `p` was a `void *` or `char *` type.

The calling program never sees these values; they are internal to the implementation of the memory allocator.

As an example, suppose your allocator is asked to reserve 80 bytes (`malloc(80)`) and requires 8 bytes of internal header data. The allocator would need to find an unallocated space of at least 88 bytes. After updating the heap data it would return a pointer to the block. However, the returned pointer does not point to the start of the block because that's where the internal size data is stored! Instead we would return the start of the block + 8 bytes. In the implementation, remember that pointer arithmetic depends on type. For example, `p += 8` adds `8 * sizeof(p)`, not necessarily 8 bytes!

Implementing malloc

▼ Pages 51

Home

#Example Markdown

#Informal Glossary

#Piazza: When And How to Ask For Help

C Programming, Part 1: Introduction

C Programming, Part 2: Text Input And Output

C Programming, Part 3: Common Gotchas

C Programming, Part 4: Debugging

Deadlock, Part 1: Resource Allocation Graph

Deadlock, Part 2: Deadlock Conditions

File System, Part 1: Introduction

File System, Part 2: Files are inodes (everything else is just data...)


File System, Part 3: Permissions


File System, Part 4: Working with directories


File System, Part 5: Virtual file systems


Show 36 more pages...


Clone this wiki locally


 Clone in Desktop











The simplest implementation uses first fit: Start at the first block, assuming it exists, and iterate until a block that represents unallocated space of sufficient size is found, or we've checked all the blocks.

If no suitable block is found then it's time to call `sbrk()` again to sufficiently extend the size of the heap. A fast implementation might extend it a significant amount so that we will not need to request more heap memory in the near future.

When a free block is found it may be larger than the space we need. If so, we will create two entries in our implicit list. The first entry is the allocated block, the second entry is the remaining space.

There are two simple ways to note if a block is in use or available. The first is to store it as a byte in the header information along with the size and the second to encode it as the lowest bit in the size! Thus block size information would be limited to only even values:

```
// Assumes p is a reasonable pointer type, e.g. 'size_t *'.
isallocated = (*p) & 1;
realsize = (*p) & ~1;  // mask out the lowest bit
```

Alignment and rounding up considerations

Many architectures expect multi-byte primitives to be aligned to some multiple of 2^n. For example, it's common to require 4-byte types to be aligned to 4-byte boundaries (and 8-byte types on 8-byte boundaries). If multi-byte primitives are not stored on a reasonable boundary (for example starting at an odd address) then the performance can be significantly impacted because it may require two memory read requests instead of one. On some architectures the penalty is even greater - the program will crash with a [bus error](#).

As `malloc` does not know how the user will use the allocated memory (array of doubles? array of chars?), the pointer returned to the program needs to be aligned for the worst case, which is architecture dependent.

From glibc documentation, the glibc `malloc` uses the following heuristic: " The block that malloc gives you is guaranteed to be aligned so that it can hold any type of data. On GNU systems, the address is always a multiple of eight on most systems, and a multiple of 16 on 64-bit systems."

For example, if you need to calculate how many 16 byte units are required, don't forget to round up -

```
int s = (requested_bytes + tag_overhead_bytes + 15) / 16
```

The additional constant ensures incomplete units are rounded up. Note, real code is more likely to symbol sizes e.g. `sizeof(x) - 1` , rather than coding numerical constant 15.

Implementing free

When `free` is called we need to re-apply the offset to get back to the 'real' start of the block (remember we didn't give the user a pointer to the actual start of the block?), i.e. to

where we stored the size information.

A naive implementation would simply mark the block as unused. If we are storing the block allocation status in the lowest size bit, then we just need to clear the bit:

```
*p = (*p) & ~1; // Clear lowest bit
```

However, we have a bit more work to do: If the current block and the next block (if it exists) are both free we need to coalesce these blocks into a single block. Similarly, we also need to check the previous block, too. If that exists and represents an unallocated memory, then we need to coalesce the blocks into a single large block.

To be able to coalesce a free block with a previous free block we will also need to find the previous block, so we store the block's size at the end of the block, too. These are called "boundary tags" (ref Knuth73). As the blocks are contiguous, the end of one blocks sits right next to the start of the next block. So the current block (apart from the first one) can look a few bytes further back to lookup the size of the previous block. With this information you can now jump backwards!

<http://www.eecs.harvard.edu/~mdw/course/cs61/mediawiki/images/5/53/Lectures-malloc1.pdf> Slide 29 (Hey world - an open source licensed diagram of the above would be nice) -Not sure if the above link is open source or licensed. Confirmation pending.

Performance

With the above description it's possible to build a memory allocator. It's main advantage is simplicity - at least simple compared to other allocators! Allocating memory is a worst-case linear time operation (search linked lists for a sufficiently large free block) and de-allocation is constant time (no more than 3 blocks will need to be coalesced into a single block). Using this allocator it is possible to experiment with different placement strategies. For example, you could start searching from where you last free'd a block, or where you last allocated from. If you do store pointers to blocks, you need to be very careful that they always remain valid (e.g. when coalescing blocks or other malloc or free calls that change the heap structure)

Explicit Free Lists Allocators

Better performance can be achieved by implementing an explicit doubly-linked list of free nodes. In that case, we can immediately traverse to the next free block and the previous free block. This can halve the search time, because the linked list only includes unallocated blocks.

A second advantage is that we now have some control over the ordering of the linked list. For example, when a block is free'd, we could choose to insert it into the beginning of the linked list rather than always between its neighbors. This is discussed below.

Where do we store the pointers of our linked list? A simple trick is to realize that the block itself is not being used and store the next and previous pointers as part of the block (though now you have to ensure that the free blocks are always sufficiently large to hold two pointers).

We still need to implement Boundary Tags (i.e. an implicit list using sizes), so that we can correctly free blocks and coalesce them with their two neighbors. Consequently, explicit free lists require more code and complexity.

With explicit linked lists a fast and simple 'Find-First' algorithm is used to find the first sufficiently large link. However, since the link order can be modified, this corresponds to different placement strategies. For example if the links are maintained from largest to smallest, then this produces a 'Worst-Fit' placement strategy.

Explicit linked list insertion policy

The newly free'd block can be inserted easily into two possible positions: at the beginning or in address order (by using the boundary tags to first find the neighbors).

Inserting at the beginning creates a LIFO (last-in, first-out) policy: The most recently free'd spaces will be reused. Studies suggest fragmentation is worse than using address order.

Inserting in address order ("Address ordered policy") inserts free'd blocks so that the blocks are visited in increasing address order. This policy required more time to free a block because the boundary tags (size data) must be used to find the next and previous unallocated blocks. However, there is less fragmentation.

Case study: Buddy Allocator (an example of a segregated list)

A segregated allocator is one that divides the heap into different areas that are handled by different sub-allocators dependent on the size of the allocation request. Sizes are grouped into classes (e.g. powers of two) and each size is handled by a different sub-allocator and each size maintains its own free list.

A well known allocator of this type is the buddy allocator. We'll discuss the binary buddy allocator which splits allocation into blocks of size 2^n ($n = 1, 2, 3, \dots$) times some base unit number of bytes, but others also exist (e.g. Fibonacci split - can you see why it's named?). The basic concept is simple: If there are no free blocks of size 2^n , go to the next level and steal that block and split it into two. If two neighboring blocks of the same size become unallocated, they can be coalesced back together into a single large block of twice the size.

Buddy allocators are fast because the neighboring blocks to coalesce with can be calculated from the free'd block's address, rather than traversing the size tags. Ultimate performance often requires a small amount of assembler code to use a specialized CPU instruction to find the lowest non-zero bit.

The main disadvantage of the Buddy allocator is that they suffer from internal fragmentation, because allocations are rounded up to the nearest block size. For example, a 68-byte allocation will require a 128-byte block.

Further Reading and References

- See [Foundations of Software Technology and Theoretical Computer Science 1999](#)

- [proceedings](#) (Google books,page 85)
- ThanksForTheMemory UIUC lecture Slides ([pptx](#)) ([pdf](#)) and
- [Wikipedia's buddy memory allocation page](#)

Other allocators

There are many other allocation schemes. For example [SLUB](#) (wikipedia) - one of three allocators used internally by the Linux Kernel.

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

