Memory Allocators 101 - Write a simple memory allocator

Code related to this article: github.com/arjun024/memalloc

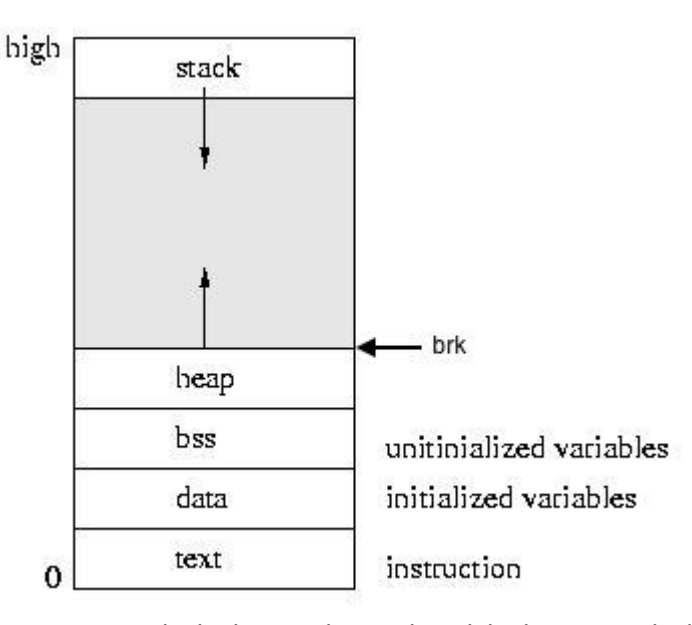
This article is about writing a simple memory allocator in C.
We will implement malloc(), calloc(), realloc() and free().

This is a beginner level article, so I will not spell out every detail.
This memory allocator will not be fast and efficient, we will not adjust allocated memory to align to a page boundary, but we will build a memory allocator that works. That's it.

If you want to take a look at the code in full, take a look at my github repo memalloc.

Before we get into building the memory allocator, you need to be familiar with the memory layout of a program. A process runs within its own virtual address space that's distinct from the virtual address spaces of other processes. This virtual address space typically comprises of 5 sections:

Text section: The part that contains the binary instructions to be executed by the processor.
Data section: Contains non-zero initialized static data.
BSS (Block Started by Symbol) : Contains zero-initialized static data. Static data uninitialized in program is initialized 0 and goes here.
Heap: Contains the dynamically allocated data.
Stack: Contains your automatic variables, function arguments, copy of base pointer etc.



As you can see in the image, the stack and the heap grow in the opposite directions.
Sometimes the data, bss and heap sections are collectively referred to as the "*data segment*",
the end of which is demarcated by a pointer named program break or **brk**.
That is, *brk* points to the end of the heap.

Now if we want to allocate more memory in the heap, we need to request the system to increment brk. Similarly, to release memory we need to request the system to decrement *brk*.

Assuming we run Linux (or a Unix-like system), we can make use of `sbrk()` system call that lets us manipulate the program break.

Calling `sbrk(0)` gives the current address of program break.
Calling `sbrk(x)` with a positive value increments brk by x bytes, as a result allocating memory.
Calling `sbrk(-x)` with a negative value decrements brk by x bytes, as a result releasing memory.
On failure, sbrk() returns `(void*) -1`.

To be honest, sbrk() is not our best buddy in 2015. There are better alternatives like *mmap()* available today. sbrk() is not really thread safe. It can can only grow or shrink in LIFO order.
If you do a `man 2 sbrk` on your macbook, it will tell you:

```
The brk and sbrk functions are historical curiosities left over from earlier days before the advent of virt
ual memory management.
```

However, the glibc implementation of malloc still uses sbrk() for allocating memory that's not too big in size.[1]
So, we will go ahead with sbrk() for our simple memory allocator.

malloc()

The *malloc(size)* function allocates *size* bytes of memory and returns a pointer to the allocated memory.
Our simple malloc will look like:

```c
void *malloc(size_t size)
{
	void *block;
	block = sbrk(size);
	if (block == (void*) -1)
		return NULL;
	return block;
}
```

In the above code, we call sbrk() with the given size.
On success, *size* bytes are allocated on the heap.
That was easy. Wasn't it?

The tricky part is freeing this memory.
The *free(ptr)* function frees the memory block pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc().
But to free a block of memory, the first order of business is to know the size of the memory block to be freed. In the current scheme of things, this is not possible as the size information is not stored anywhere. So, we will have to find a way to store the size of an allocated block somewhere.

Moreover, we need to understand that the heap memory the operating system has provided is contiguous. So we can only release memory which is at the end of the heap. We can't release a block of memory in the middle to the OS. Imagine your heap to be something like a long loaf of bread that you can stretch and shrink at one end, but you have to keep it in one piece.
To address this issue of not being able to release memory that's not at the end of the heap, we will make a distinction between freeing memory and releasing memory.
From now on, freeing a block of memory does not necessarily mean we release memory back to OS. It just means that we keep the block marked as free. This block marked as free may be reused on a later malloc() call. Since memory not at the end of the heap can't be released, this is the only way ahead for us.

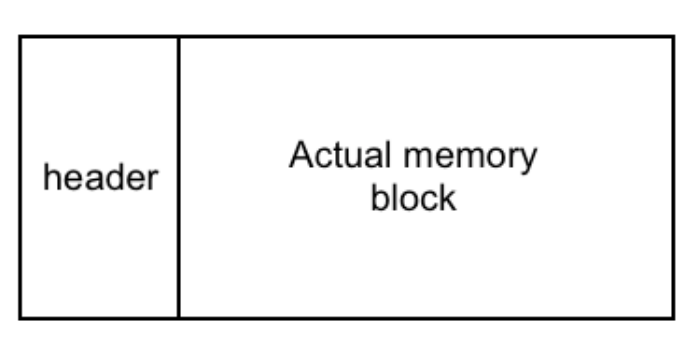
So now, we have two things to store for every block of allocated memory:
1. size
2. Whether a block is free or not-free?

To store this information, we will add a header to every newly allocated memory block.
The header will look something like this:

```c
struct header_t {
	size_t size;
	unsigned is_free;
};
```

The idea is simple. When a program requests for *size* bytes of memory, we calculate `total_size = header_size + size`, and call `sbrk(total_size)`. We use this memory space returned by sbrk() to fit in both the header and the actual memory block. The header is internally managed, and is kept completely hidden from the calling program.

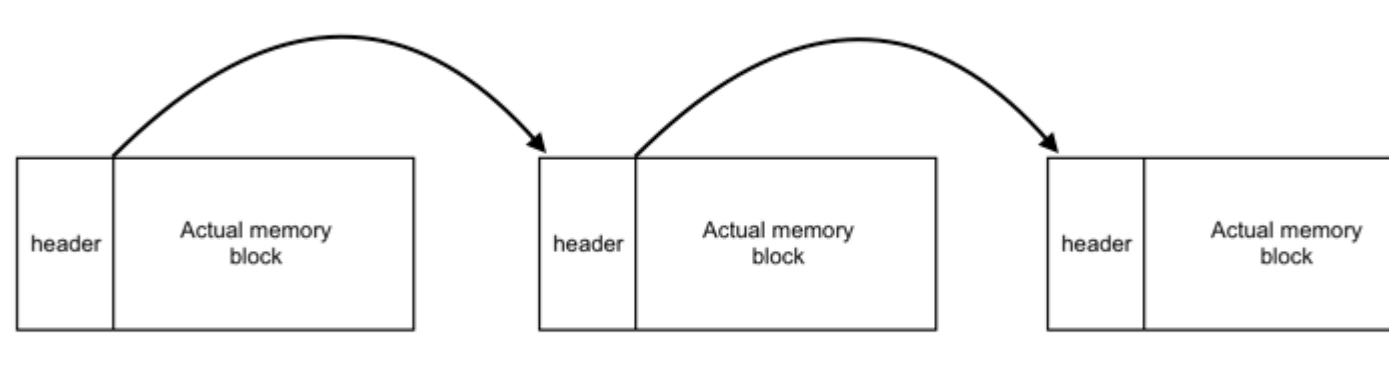
Now, each one of our memory blocks will look like:



We can't be completely sure the blocks of memory allocated by our malloc is contiguous. Imagine the calling program has a foreign sbrk(), or there's a section of memory *mmap()*ed in between our memory blocks. We also need a way to traverse through our blocks for memory (why traverse? we will get to know when we look at the implementation of *free()*). So to keep track of the memory allocated by our malloc, we will put them in a linked list. Our header will now look like:

```c
struct header_t {
	size_t size;
	unsigned is_free;
	struct header_t *next;
};
```

and the linked list of memory blocks like this:



Now, let's wrap the entire header struct in a `union` along with a stub variable of size 16 bytes. This makes the header end up on a memory address aligned to 16 bytes. Recall that the size of a union is the larger size of its members. So the union guarantees that the end of the header is memory aligned. The end of the header is where the actual memory block begins and therefore the memory provided to the caller by the allocator will be aligned to 16 bytes.

```c
typedef char ALIGN[16];

union header {
	struct {
		size_t size;
		unsigned is_free;
		union header *next;
	} s;
	ALIGN stub;
};
typedef union header header_t;
```

We will have a head and tail pointer to keep track of the list.

```c
header_t *head, *tail;
```

To prevent two or more threads from concurrently accessing memory, we will put a basic locking mechanism in place.

We'll have a global lock, and before every action on memory you have to acquire the lock, and once you are done you have to release the lock.

```c
pthread_mutex_t global_malloc_lock;
```

Our malloc is now modified to:

```c
void *malloc(size_t size)
{
	size_t total_size;
	void *block;
	header_t *header;
	if (!size)
		return NULL;
	pthread_mutex_lock(&global_malloc_lock);
	header = get_free_block(size);
	if (header) {
		header->s.is_free = 0;
		pthread_mutex_unlock(&global_malloc_lock);
		return (void*)(header + 1);
	}
	total_size = sizeof(header_t) + size;
	block = sbrk(total_size);
	if (block == (void*) -1) {
		pthread_mutex_unlock(&global_malloc_lock);
		return NULL;
	}
	header = block;
	header->s.size = size;
	header->s.is_free = 0;
	header->s.next = NULL;
	if (!head)
		head = header;
	if (tail)
		tail->s.next = header;
	tail = header;
	pthread_mutex_unlock(&global_malloc_lock);
	return (void*)(header + 1);
}

header_t *get_free_block(size_t size)
{
	header_t *curr = head;
	while(curr) {
		if (curr->s.is_free && curr->s.size >= size)
			return curr;
		curr = curr->s.next;
	}
	return NULL;
}
```

Let me explain the code:

We check if the requested size is zero. If it is, then we return `NULL`.
For a valid size, we first acquire the lock. The we call `get_free_block()` - it traverses the linked list and see if there already exist a block of memory that is marked as free and can accomodate the given size. Here, we take a first-fit approach in searching the linked list.

If a sufficiently large free block is found, we will simply mark that block as not-free, release the global lock, and then return a pointer to that block. In such a case, the *header* pointer will refer to the header part of the block of memory we just found by traversing the list. Remember, we have to hide the very existence of the header to an outside party. When we do `(header + 1)`, it points to the byte right after the end of the header. This is incidentally also the first byte of the actual memory block, the one the caller is interested in. This is cast to `(void*)` and returned.

If we have not found a sufficiently large free block, then we have to extend the heap by calling sbrk(). The heap has to be extended by a size that fits the requested *size* as well a header. For that, we first compute the total size: `total_size = sizeof(header_t) + size;`. Now, we request the OS to increment the program break: `sbrk(total_size)`.

In the memory thus obtained from the OS, we first make space for the header. In C, there is no need to cast a `void*` to any other pointer type, it is always safely promoted. That's why we don't explicitly do: `header = (header_t *)block;`
We fill this header with the requested size (not the total size) and mark it as not-free. We update the *next* pointer, head and tail so to reflect the new state of the linked list. As explained earlier, we hide the header from the caller and hence return `(void*)(header + 1)`. We make sure we release the global lock as well.

free()

Now, we will look at what free() should do. free() has to first deterimine if the block-to-be-freed is at the end of the heap. If it is, we can release it to the OS. Otherwise, all we do is mark it 'free', hoping to reuse it later.

```c
void free(void *block)
{
	header_t *header, *tmp;
	void *programbreak;

	if (!block)
		return;
	pthread_mutex_lock(&global_malloc_lock);
	header = (header_t*)block - 1;

	programbreak = sbrk(0);
	if ((char*)block + header->s.size == programbreak) {
		if (head == tail) {
			head = tail = NULL;
		} else {
			tmp = head;
			while (tmp) {
				if(tmp->s.next == tail) {
					tmp->s.next = NULL;
					tail = tmp;
				}
				tmp = tmp->s.next;
			}
		}
		sbrk(0 - sizeof(header_t) - header->s.size);
		pthread_mutex_unlock(&global_malloc_lock);
		return;
	}
	header->s.is_free = 1;
	pthread_mutex_unlock(&global_malloc_lock);
}
```

Here, first we get the header of the block we want to free. All we need to do is get a pointer that is behind the block by a distance equalling the size of the header. So, we cast *block* to a header pointer type and move it behind by 1 unit.
`header = (header_t*)block - 1;`

`sbrk(0)` gives the current value of program break. To check if the block to be freed is at the end of the heap, we first find the end of the current block. The end can be computed as `(char*)block + header->s.size`. This is then compared with the program break.

If it is in fact at the end, then we could shrink the size of the heap and release memory to OS. We first reset our head and tail pointers to reflect the loss of the last block. Then the amount of memory to be released is calculated. This the sum of sizes of the header and the acutal block: `sizeof(header_t) + header->s.size`. To release this much amount of memory, we call sbrk() with the negative of this value.

In the case the block is not the last one in the linked list, we simply set the *is_free* field of its header. This is the field checked by *get_free_block()* before actually calling sbrk() on a malloc().

calloc()

The calloc(num, nsize) function allocates memory for an array of *num* elements of *nsize* bytes each and returns a pointer to the allocated memory. Additionally, the memory is all set to zeroes.

```c
void *calloc(size_t num, size_t nsize)
{
	size_t size;
	void *block;
	if (!num || !nsize)
		return NULL;
	size = num * nsize;
	/* check mul overflow */
	if (nsize != size / num)
		return NULL;
	block = malloc(size);
	if (!block)
		return NULL;
	memset(block, 0, size);
	return block;
}
```

Here, we do a quick check for multiplicative overflow, then call our malloc(),
and clears the allocated memory to all zeroes using memset().

realloc()

realloc() changes the size of the given memory block to the size given.

```c
void *realloc(void *block, size_t size)
{
	header_t *header;
	void *ret;
	if (!block || !size)
		return malloc(size);
	header = (header_t*)block - 1;
	if (header->s.size >= size)
		return block;
	ret = malloc(size);
	if (ret) {

		memcpy(ret, block, header->s.size);
		free(block);
	}
	return ret;
}
```

Here, we first get the block's header and see if the block already has the size to accomodate the requested size. If it does, there's nothing to be done.

If the current block does not have the requested size, then we call malloc() to get a block of the request size, and relocate contents to the new bigger block using memcpy(). The old memory block is then freed.

Compiling and using our memory allocator.

You can get the code from my github repository - memalloc.
We'll compile our memory allocator and then run a utility like **ls** using our memory allocator.

To do that, we will first compile it as a library file.

```
$ gcc -o memalloc.so -fPIC -shared memalloc.c
```

The *-fPIC* and *-shared* options makes sure the compiled output has position-independent code and tells the linker to produce a shared object suitable for dynamic linking.

On Linux, if you set the enivornment variable **LD_PRELOAD** to the path of a shared object, that file will be loaded before any other library. We could use this trick to load our compiled library file first, so that the later commands run in the shell will use our malloc(), free(), calloc() and realloc().

```
$ export LD_PRELOAD=$PWD/memalloc.so
```

Now,

```
$ ls
memalloc.c		memalloc.so
```

Voila! That's our memory allocator serving *ls*.
Print some debug message in malloc() and see it for yourself if you don't believe me.

Thank you for reading. All comments are welcome. Please report bugs if you find any.

Footnotes, References

See a list of memory allocators:
liballoc
Doug Lea's Memory Allocator.
TCMalloc
ptmalloc

[1] The GNU C Library: Malloc Tunable Parameters
OSDev - Memory allocation
Memory Allocators 101 - James Golick

1:11 AM 9th August 2015 0 notes

c memory Memory operating system

Ashwani Yadav
7 months ago

When you use sbrk(0) you get the current "break" address.
When you use sbrk(size) you get the previous "break" address, i.e. the one before the change.

reference: https://stackoverflow.com/q...