malloc implementation?

I'm trying to implement `malloc` and `free` for C, and I am not sure how to reuse memory. I currently have a `struct` that looks like this:

```
typedef struct _mem_dictionary {
    void *addr;
    size_t size;
    int freed;
} mem_dictionary;
```

My `malloc` looks like this:

```
void *malloc(size_t size) {
    void *return_ptr = sbrk(size);
    if (dictionary == NULL)
        dictionary = sbrk(1024 * sizeof(mem_dictionary));

    dictionary[dictionary_ct].addr = return_ptr;
    dictionary[dictionary_ct].size = size;
    dictionary[dictionary_ct].freed = 1;
    dictionary_ct++;

    return return_ptr;
}
```

When I free memory, I would just mark the address as `0` (that would indicate that it is free). In my `malloc`, I would then use a for loop to look for any value in the array to equal `0` and then allocate memory to that address. I'm kind of confused how to implement this.

c | memory-management | malloc | free

edited Jul 28 '16 at 21:21
chqrlie
50.2k ● 7 ● 40 ● 95

asked Mar 24 '11 at 16:04
user675257
126 ● 1 ● 2 ● 3

2 There's a nice writeup about dlmalloc here: g.oswego.edu/dl/html/malloc.html – ninjalj Mar 24 '11 at 21:24

3 Answers

The easiest way to do it is to keep a linked list of free block. In `malloc`, if the list is not empty, you search for a block large enough to satisfy the request and return it. If the list is empty or if no such block can be found, you call `sbrk` to allocate some memory from the operating system. in `free`, you simply add the memory chunk to the list of free block. As bonus, you can try to merge contiguous freed block, and you can change the policy for choosing the block to return (first fit, best fit, ...). You can also choose to split the block if it is larger than the request.

Some sample implementation (it is not tested, and is obviously not thread-safe, use at your own risk):

```
typedef struct free_block {
    size_t size;
    struct free_block* next;
} free_block;

static free_block free_block_list_head = { 0, 0 };
static const size_t overhead = sizeof(size_t);
static const size_t align_to = 16;

void* malloc(size_t size) {
    size = (size + sizeof(size_t) + (align_to - 1)) & ~ (align_to - 1);
    free_block* block = free_block_list_head.next;
    free_block** head = &(free_block_list_head.next);
    while (block != 0) {
        if (block->size >= size) {
            *head = block->next;
            return ((char*)block) + sizeof(size_t);
        }
        head = &(block->next);
        block = block->next;
    }

    block = (free_block*)sbrk(size);
    block->size = size;
```

```
void free(void* ptr) {
    free_block* block = (free_block*)(((char*)ptr) - sizeof(size_t));
    block->next = free_block_list_head.next;
    free_block_list_head.next = block;
}
```

Note: `(n + align_to - 1) & ~ (align_to - 1)` is a trick to round `n` to the nearest multiple of `align_to` that is larger than `n`. This only works when `align_to` is a power of two and depends on the binary representation of numbers.

When `align_to` is a power of two, it only has one bit set, and thus `align_to - 1` has all the lowest bit sets (ie. `align_to` is of the form 000...010...0, and `align_to - 1` is of the form `000...001...1`). This means that `~ (align_to - 1)` has all the high bit set, and the low bit unset (ie. it is of the form `111...110...0`). So `x & ~ (align_to - 1)` will set to zero all the low bits of `x` and round it down to the nearest multiple of `align_to`.

Finally, adding `align_to - 1` to `size` ensure that we round-up to the nearest multiple of `align_to` (unless `size` is already a multiple of `align_to` in which case we want to get `size`).

edited Nov 4 '12 at 19:28

answered Mar 24 '11 at 16:33

Sylvain Defresne

29k ● 6 ● 57 ● 78

what does the magic in the first line of the malloc function do? – iGbanam Nov 4 '12 at 1:07

It rounds `(size + sizeof(size_t))` to the nearest multiple of `align_to` that is larger than `(size + sizeof(size_t))`. This only works if `align_to` is a power of two. – Sylvain Defresne Nov 4 '12 at 19:19

A similar technique of using a linked list cache to hold onto allocated memory (instead of allocating all over again) to speed up a graphics program (which is spending too much time on malloc) is used as an example in the first part of Column 9: Code Tuning in the book Programming Pearls by Jon Bentley. The book, sadly, does not contain code in its example, so seeing code like this was especially useful to me. – Drake Sobania Feb 2 '15 at 1:14

You don't want to set the `size` field of the dictionary entry to zero -- you will need that information for re-use. Instead, set `freed=1` only when the block is freed.

You cannot coalesce adjacent blocks because there may have been intervening calls to `sbrk()`, so that makes this easier. You just need a `for` loop which searches for a large enough freed block:

```
typedef struct _mem_dictionary
{
    void *addr;
    size_t size;
    int freed;
} mem_dictionary;


void *malloc(size_t size)
{
     void *return_ptr = NULL;
     int i;

     if (dictionary == NULL) {
         dictionary = sbrk(1024 * sizeof(mem_dictionary));
         memset(dictionary, 0, 1024 * sizeof(mem_dictionary));
     }

     for (i = 0; i < dictionary_ct; i++)
         if (dictionary[i].size >= size
          && dictionary[i].freed)
     {
         dictionary[i].freed = 0;
         return dictionary[i].addr;
     }

     return_ptr = sbrk(size);

     dictionary[dictionary_ct].addr = return_ptr;
     dictionary[dictionary_ct].size = size;
     dictionary[dictionary_ct].freed = 0;
     dictionary_ct++;

     return return_ptr;
}

void free(void *ptr)
{
    int i;

    if (!dictionary)
        return;

    for (i = 0; i < dictionary_ct; i++ )
    {
        if (dictionary[i].addr == ptr)
        {
```

```
            dictionary[i].freed = 1;
            return;
        }
    }
}
```

This is not a great `malloc()` implementation. In fact, most `malloc` / `free` implementations will allocate a small header for each block returned by malloc. The header might start at the address eight (8) bytes *less* than the returned pointer, for example. In those bytes you can store a pointer to the `mem_dictionary` entry owning the block. This avoids the O(N) operation in `free`. You can avoid the O(N) in `malloc()` by implementing a priority queue of freed blocks. Consider using a binomial *heap*, with block size as the index.

edited Mar 24 '11 at 16:34

answered Mar 24 '11 at 16:17

Heath Hunnicutt

13.8k ● 1 ● 31 ● 55

Sorry, I'm relatively new to C, but what is the dictionary variable in malloc()? – no92 Sep 11 '13 at 10:46

@no92 -- I should have named that "journal" instead of "dictionary." Remember, this is my *example* and trivial implementation of `malloc`. It has at least one obvious flaw: there can never be more than 1024 blocks allocated at one time. The only purpose of giving this example is to give the reader a *starting point* for implementing their own `malloc`. This is just a conceptual basis for implementing a `malloc` / `free` library. It does not even implement `realloc` as one other glaring deficiency. It might not even be the best example. :) – Heath Hunnicutt Sep 12 '13 at 17:12

I am borrowing code from Sylvain's response. He seems to have missed calculating the size of the free_block* ini calculating the overhead.

In overall the code works by prepending this free_block as a header to the allocated memory. 1. When user calls malloc, malloc returns the address of the payload, right after this header. 2. when free is called, the address of the starting of the header for the block is calculated (by subtracting the header size from the block address) and that is added to the free block pool.

```
typedef struct free_block {
    size_t size;
    struct free_block* next;
} free_block;

static free_block free_block_list_head = { 0, 0 };

// static const size_t overhead = sizeof(size_t);

static const size_t align_to = 16;

void* malloc(size_t size) {
    size = (size + sizeof(free_block) + (align_to - 1)) & ~ (align_to - 1);
    free_block* block = free_block_list_head.next;
    free_block** head = &(free_block_list_head.next);
    while (block != 0) {
        if (block->size >= size) {
            *head = block->next;
            return ((char*)block) + sizeof(free_block);
        }
        head = &(block->next);
        block = block->next;
    }

    block = (free_block*)sbrk(size);
    block->size = size;

    return ((char*)block) + sizeof(free_block);
}

void free(void* ptr) {
    free_block* block = (free_block*)(((char*)ptr) - sizeof(free_block ));
    block->next = free_block_list_head.next;
    free_block_list_head.next = block;
}
```

answered Oct 7 '12 at 23:16

Kingkong Jnr

506 ● 2 ● 8 ● 24

1 Thank you, I think this response is slightly more correct than Sylvain's response, since I was just wondering about this. The overhead variable is a very good idea, just not correctly calculated or even used. – bbill Oct 18 '14 at 22:05

Can anyone tell me the use of head in malloc function? (`free_block** head = &(free_block_list_head.next);`) I don't see it being used anywhere. In addition, why do we add `sizeof(free_block)` before returning? – user1719160 Mar 8 '16 at 6:22

`head` is a pointer to an address containing a pointer. It is used in the while-loop to unlink the memory block returned to the user. Adding and subtracting `sizeof(free_block)` is a common and neat trick to "hide" the

metadata from the caller. – Björn Lindqvist Apr 13 '16 at 4:34

There is also a small error in the free() implementation as `free(NULL)` will segfault. – Björn Lindqvist Apr 13 '16 at 4:35

Sylvain uses sizeof(size_t) instead of sizeof(free_block), because the "next pointer" is only used in freed blocks, otherwise it can be overwritten by the user. – bjoernz Jul 25 '16 at 17:00