A Malloc Tutorial*

Marwan Burelle†
Laboratoire Système et Sécurité de l'EPITA (LSE)

February 16, 2009

Contents

*http://wiki-prog.kh405.net
†marwan.burelle@lse.epita.fr

1 Introduction

What is malloc ? If you don't even know the name, you might begin to learn C in the Unix environment prior to read this tutorial. For a programmer, malloc is the function to allocate memory blocks in a C program, most people don't know what is really behind, some even thinks its a syscall or language keyword. In fact malloc is nothing more than a simple function and can be understood with a little C skills and almost no system knowledge.

The purpose of this tutorial is to code a simple malloc function in order to understand the underlying concepts. We will not code an efficient malloc, just a basic one, but the concept behind can be useful to understand how memory is managed in every day processes and how-to deal with blocks allocation, reallocation and freeing.

From a pedagogical standpoint, this is a good C practice. It is also a good document to understand where your pointers come from and how things are organized in the heap.

What is malloc
malloc(3) is a *Standard C Library* function that *allocates* (*i.e.* reserves) memory chunks. It complies with the following rules:

- malloc allocates at least the number of bytes requested;
- The pointer returned by malloc points to an allocated space (*i.e.* a space where the program can read or write successfully;)
- No other call to malloc will allocate this space or any portion of it, unless the pointer has been freed before.
- malloc should be tractable: malloc must terminate in as soon as possible (it should not be NP-hard !;)
- malloc should also provide resizing and freeing.

The function obey to the following signature:

```
void*           malloc(size_t size);
```

Where size is the requested size. The returned pointer should be NULL in case of faillure (no space left.)

2 The Heap and the brk and sbrk syscalls

Prior to write a first malloc, we need to understand how memory is managed in most multitask systems. We will keep an abstract point of view for that part, since many details are system and hardware dependant.

2.1 The Process's Memory

Each process has its own virtual adress space dynamically translated into physical memory adress space by the MMU (and the kernel.) This space is devided in sevral part, all that we have to know is that we found at least some space for the code, a stack where local and volatile data are stored, some space for constant and global variables and an unorganized space for program's data called the heap.

The heap is a continuous (in terme of virtual adresses) space of memory with three *bounds*: a starting point, a maximum limit (managed through `sys/ressource.h`'s functions `getrlimit(2)` and `setrlimit(2)`) and an end point called the **break**. The break marks the end of the mapped memory space, that is, the part of the virtual adress space that has correspondance into real memory. Figure 1 sketches the memory organisation.

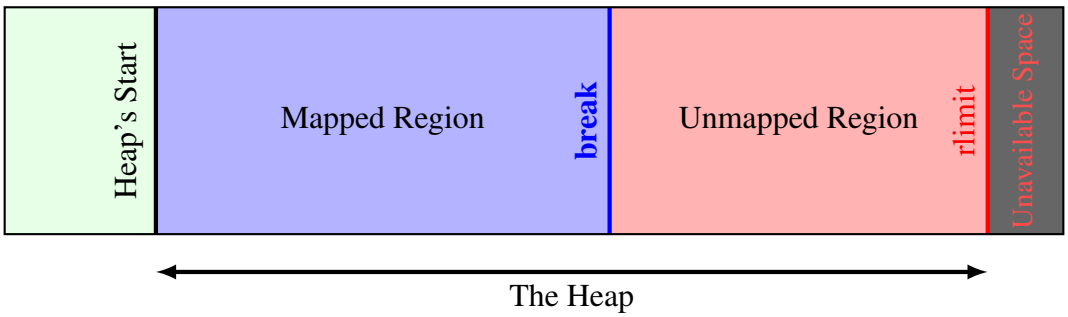

Figure 1: Memory organisation

In order to code a `malloc`, we need to know where the heap begin and the break position, and of course we need to be able to move the break. This the purpose of the two syscalls `brk` and `sbrk`.

2.2 `brk(2)` and `sbrk(2)`

We can find the description of these syscalls in their manual pages:

```
int             brk(const void *addr);
void*           sbrk(intptr_t incr);
```

`brk(2)` place the break at the given adress `addr` and return `0` if successful, `-1` otherwise. The global `errno` symbol indicate the nature of the error.
`sbrk(2)` move the break by the given increment (in bytes.) Depending on system implementation, it returns the previous or the new break adress. On failure, it returns `(void *)-1` and set `errno`. On some system `sbrk` accepts negative values (in order to free some mapped memory.)

Since `sbrk`'s specification does not fix the meaning of its result, we won't use the returned value when moving the break. But, we can use a special case of `sbrk`: when increment is nul (*i.e.* `sbrk(0)`), the returned value is the actual break adress (the previous and the new break adresses are the same.) `sbrk` is thus used to retrieve the begining of the heap which is the initial position of the break.

We will use `sbrk` as our main tool to implement `malloc`. All we have to do is to acquire more space (if needed) to fulfil the query.

2.3 Unmapped Region and No-Man's Land

We saw earlier that the break mark the end of the mapped virtual adress space: accessing adresses above the break should trigger a *bus error*. The remaining space between the break and the maximum limit of the heap is not associated to physical memory by the virtual memory manager of the system (the MMU and the dedicated part of the kernel.)

But, if you know a little about virtual memory (and even if use your brain 5 seconds), you know that memory is mapped by pages: physical memory and virtual memory is organize in pages (frames for the physical memory) of fixed size (most of the time.) The page size is by far bigger than one byte (on most actual system a page size is 4096 bytes.) Thus, the break may not be on pages boundary.

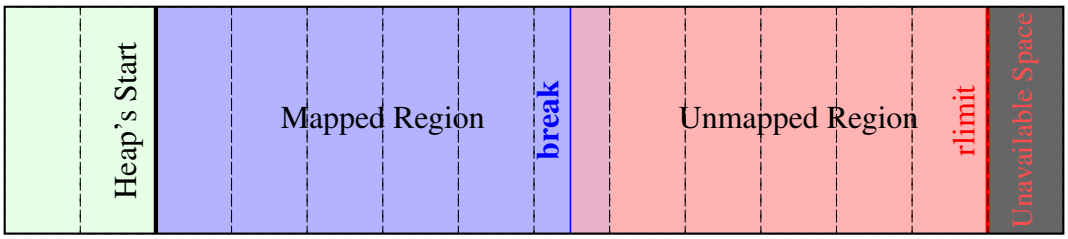

Figure 2: Pages and Heap

Figure 2 presents the previous memory organisation with page boundaries. We can see the break may not correspond to a page boundary. What is the status of the memory betwee the break and the next page boundary ? In fact, this space is available ! You can access (for reading or writing) bytes in this space. The issue is that you don't have any clue on the position of the next boundary, you can find it but it is system dependant and badly advise.

This *no-man's land* is often a root of bugs: some wrong manipulations of pointer outside of the heap can success most of the time with small tests and fail only when manipulating larger amount of data.

2.4 `mmap(2)`

Even if we won't use it in this tutorial, you should pay attention to the `mmap(2)` syscall. It has an annonymous mode (`mmap(2)` is usualy used to directly map files in memory) which can used to implement `malloc` (completely or for some specific cases.)

`mmap` in annonymous mode can allocate a specific amount of memory (by page size granularity) and `munmap` can free it. It is often simpler and more efficient than classical `sbrk` based `malloc`. Many `malloc` implementation use `mmap` for large allocation (more than one page.) The OpenBSD's `malloc` uses only `mmap` with some quirks in order to improve security (prefering page's borders for allocation with holes between pages.)

3 Dummy malloc

First, we will play with `sbrk(2)` to code a dummy `malloc`. This `malloc` is probabily the worst one, even if it is the simplest and quiet the fastest one.

3.1 Principle

The idea is very simple, each time `malloc` is called we move the break by the amount of space required and return the previous address of the break. It is simple and fast, it takes only three lines, but we cannot do a real `free` and of course `realloc` is impossible.

This `malloc` waste a lot of space in obsolete memory chunks. It is only here for educationnal purpose and to try the `sbrk(2)` syscall. For educationnal purposes, we will also add some error management to our `malloc`.

3.2 Implementation

```
1  /* An horrible dummy malloc */
2
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  void        *malloc(size_t size)
7  {
8    void *p;
9    p = sbrk(0);
10   /* If sbrk fails, we return NULL */
11   if (sbrk(size) == (void*)-1)
12     return NULL;
13   return p;
14 }
```

4 Organizing the Heap

In Section 3 on the preceding page, we present a first attempt to code a `malloc` function, but we failed to fulfil all the requirements. In this section we will try to find an organisation of the heap so that we can have an efficient `malloc` but also a `free` and a `realloc`.

4.1 What do we need ?

If we consider the problem outside of the programming context, we can infer what kind of information we need to solve our issues. Let's take analogy: you own a field and partition it to rent portion of it. Clients ask for different length (you divide your field using only one dimension) which they expect to be continous. When they have finished they give you back their porpotion, so you can rent it again.

On one side of the field you have a road with a programmable car: you enter the distance between the begin of the field and the destination (the begining of your portion.) So we need to know where each portion begin (this the pointer returned by `malloc`) and when we are at a begining of a portion we need the *address* of the next one.

A solution is to put a sign at the begining of each where we indicate the address of the next one (and the size of the current one to avoid unecessary computing.) We also add a mark on free portion (we put that mark when the client give it back.) Now, when a client want a portion of a certain size we take the car and travel from sign to sign. When we find a portion marked as

free and wide enough we give it to the client and remove the mark from the sign. If we reach the last portion (its sign have no next portion address) we simply go to the end of the portion and add a new sign.

Now we can transpose this idea to the memory: we need extra-information at begining of each chunks indicating at least the size of the chunk, the address of the next one and whether its free or not.

4.2 How to represent block information

So what we need is a small block at the begining of each chunk containing the extra-information, called meta-data. This block contains at least a pointer to the next chunk, a flag to mark free chunks and the size of the data of the chunk. Of course, this block of information is before the pointer returned by `malloc`.

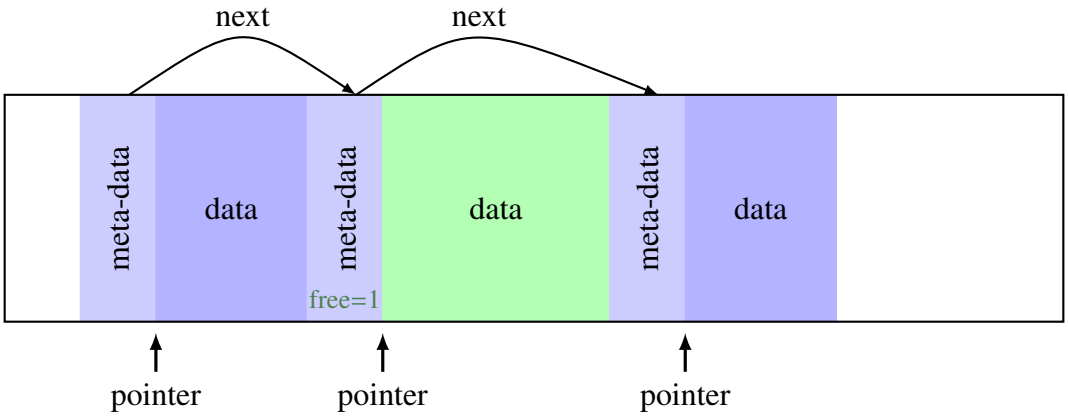

Figure 3: Heap's Chunks Structures

Figure 3 presents an example of heap organisation with meta-data in front of the allocated block. Each chunck of data is composed of a block of meta-data followed by the block of data. The pointer returned by `malloc` is indicated in the lower part of the diagram, note that it points on the data block, not on the complete chunk.

Now, how we translate this into C code ? This look like a classic linked list (and this is a linked list), so we write a type for linked list with the needed information inside member of the list. The type definition is self describing and presents no surprise:

```
1 typedef struct s_block *t_block;
2
3 struct s_block {
4   size_t        size;
5   t_block       next;
6   int           free;
7 };
```

Note the use of a `typedef` to simplify the use of the type. Since we will never use the type without a pointer we include it in the `typedef`, this is in fact the good practice for linked list since the list is the pointer, not the block (an empty list is a `NULL` pointer.)

It may seems a waste of space to use an int as a flag, but since `struct` are aligned by default, it won't have changed anything, we will see later how we can shrink the size of meta-data. Another point that we will see later is that `malloc` must return aligned address.

A frequent question at this point is: how can we create a `struct` without a working `malloc` ? The answer is simple, you just need to know what realy is a `struct`. In memory, a `struct` is simply the concatenation of its fields, so in our example a `struct s_block` is just 12 bytes (with 32 bit integer) the first four correspond to `size`, the next four to the pointer to next block and finaly the last four are the integer `free`. When the compiler encounter an access to `struct` field (like `s.free` or `p->free`) it translate it to the base address of the `struct` plus the sum of the length of the previous field (so `p->free` is equivalent to `*((char*)p+8)` and `s.free` is equivalent to `*((char*)&s + 8)`.) All you have to do is to allocate enough space with `sbrk` for the chunck (so the size of the meta-data plus the size of the data block) and put the address of the old break in a variable of type `t_block`:

```
/* Example of using t_block without malloc      */
t_block         b;
/* save the old break in b                      */
b = sbrk(0);
/* add the needed space                         */
/* size is the parameter of malloc              */
sbrk(sizeof(struct s_block) + size);
b->size = size;
/* ...                                          */
```

5 A First Fit Malloc

In this section we will implement the classic *first fit* `malloc`. The first fit algorithm is quiet simple: we traverse the chunks list and stop when we find a free block with enough space for the requested allocation.

5.1 Aligned Pointers

It is often required that pointers be aligned to the integer size (which is also the pointer size.) In out case we will consider only 32 bit case. So, our pointer must be a multiple of 4 (`32 bits = 4 bytes`, of course.) Since our meta-data block is already aligned, the only thing we need is to align the size of the data block.

How do we do this ? There are several ways, on of the more efficient is to add a preprocessor macro using arithmetic trick.

First, the arithmetic trick: given any positive integer dividing (integer division) it by four and then multiplying it by four again results in the nearest smaller multiple of four, thus to obtain the nearest greater multiple of four we only need to add four to it. This is quiet simple, but it don't work with an integer multiple of four, since it results with the next multiple of four. But, let's play with arithmetic again, let x be an integer such that $x = 4 \times p + q$ with $0 \leq q \leq 3$, then if x is a multiple of four, $q = 0$ and $x-1 = 4\times(p-1)+3$, so $((x-1)/4)\times 4+4 = 4\times p = x$. If x is not a multiple of four, then $q \neq 0$ and $x - 1 = 4 \times p + (q - 1)$ and $0 \leq q - 1 \leq 2$, so $(x-1)/4\times 4+4 = 4\times p+4 = x/4\times 4+4$. And thus, we have our formula since $(x-1)/4\times 4+4$ always results to the nearest greater or equal multiple of four.

So, how do we do that in C ? First, note that dividing and multiplying by four can be express using right and left bit-shift (>> and << in C) which are faster than simple multiplication. So our formula can be write in C like that: `(((x-1)>>2)<<2)+4`, to have a proper macro we need some extra parenthesis: `(((((x)-1)>>2)<<2)+4)`. And now we can write our macro:

```
#define align4(x) (((((x)-1)>>2)<<2)+4)
```

5.2 Finding a chunk: the First Fit Algorithm

Finding a free sufficiently wide chunk is quite simple: We begin at the base address of the heap (saved somehow in our code, we will see that later) test the current chunk, if it fit our need we just return its address, otherwise we continue to the next chunk until we find a fitting one or the end of the head. The only *trick* is to keep the last visited chunk, so the `malloc` function can easily extends the end of the heap if we found no fitting chunk. The code is straightforward, `base` is a global pointer to the starting point of our heap:

```
1 t_block find_block(t_block *last, size_t size){
2   t_block b=base;
3   while (b && !(b->free && b->size >= size)) {
4     *last = b;
5     b = b->next;
6   }
7   return (b);
8 }
```

The function returns a fitting chunk, or `NULL` if none where found. After the execution, the argument `last` points to the last visited chunk.

5.3 Extending the heap

Now, we won't always have a fitting chunk, and sometimes (especially at the begining of the program using our `malloc`) we need to extends the heap. This is quite simple: we move the break and initialize a new block, of course we need to update the `next` field of the last block on the heap.

In later development we will need to do some trick with the size of `struct s_block`, so we define a macro holding the size of meta-data block, for now it is define as:

```
#define BLOCK_SIZE sizeof(struct s_block)
```

Nothing surprising in this code, we just return `NULL` if `sbrk` fails (and we don't try to understand why.) Note also that since we're not sure that `sbrk` returns the previous break, we first save it and then move the break. We could have compute it using `last` and `last->size`.

```
1 t_block extend_heap(t_block last, size_t s){
2   t_block        b;
3   b = sbrk(0);
4   if (sbrk(BLOCK_SIZE + s) == (void*)-1)
5     /* sbrk fails, go to die */
6     return (NULL);
7   b->size = s;
8   b->next = NULL;
9   if (last)
```

```
10      last->next = b;
11    b->free = 0;
12    return (b);
13  }
```

5.4 Spliting blocks

You may have notice that we use the first available block regardless of its size (provide that it's wide enough.) If we do that we will loose a lot of space (think of it: you ask for 2 bytes and find a block of 256 bytes.) A first solution is to split blocks: when a chunk is wide enough to held the asked size plus a new chunk (at least `BLOCK_SIZE + 4`), we insert a new chunk in the list.

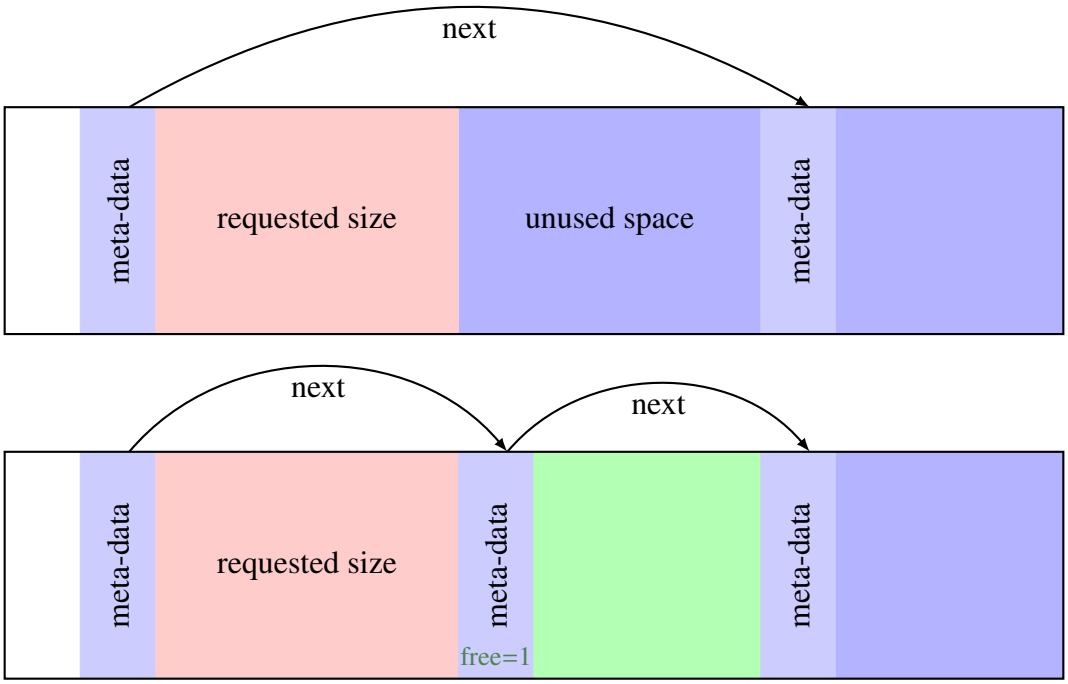

Figure 4: Splitting blocks

The following function is called only if space is available. The provided size must also be aligned. In this function we'll have to do some pointer arithmetic, to prevent errors, we will use a little trick to be sure that all our operations are done with one byte precision (remember `p+1` depends on the type pointed to by `p`.)

We just add another field to `struct s_block` of type characters array. Arrays in structure are simply *put there*: the array lies directly in the whole memory block of the structure at the point where the field is defined, thus for us, the array's pointer indicate the end of the meta-data. C forbid zero length array, so we define a one byte long array and this why we need to specify a macro for the size of `struct s_block`.

```
struct s_block {
  size_t        size;
  t_block       next;
```

```
  int           free;
  char          data[1];
};

/* Of course, update the BLOCK_SIZE macro */
#define BLOCK_SIZE 12 /* 3*4 ... */
```

Note that this extension does not require any modification to `extend_heap` since the new field will not be used directly.

Now the `split_block`: this function cut the block passed in argument to make data block of the wanted size. As for previous function `s` is already alined. The Figure 4 on the preceding page shows what is done here.

```
1 void split_block(t_block b, size_t s){
2   t_block        new;
3   new       = b->data + s;
4   new->size = b->size - s - BLOCK_SIZE;
5   new->next = b->next;
6   new->free = 1;
7   b->size   = s;
8   b->next   = new;
9 }
```

Note the use of `b->data` in pointer arithmetic on line 3. Since the field `data` is of type `char[]` we are sure that the sum is done by byte precision.

5.5 The `malloc` function

Now, we can do our `malloc` function. It is mostly a wrapper around previous functions. We have to aligne the required size, test if we are the first called `malloc` or not and every thing else have already been told.

First, remember in Section 5.2 on page 8 the function `find_block` use a global variable `base`. This variable is defined as follow:

```
void *base=NULL;
```

It is a `void` pointer and it is `NULL` initialized. The first thing our `malloc` does is to test `base`, if it `NULL` then this the first time we're called, otherwise we start the previously described algorithm.

Our `malloc` follow these lines:

- First align the requested size;
- If `base` is initialized:
  - Search for a free chunk wide enough;
  - If we found a chunk:
    * Try to split the block (the difference between the requested size and the size of the block is enough to store the meta-data and a minimal block (4 bytes;)
    * Mark the chunk as used (`b->free=0`;)

  - Otherwise: we extend the heap.

    Note the use of the `last`: `find_block` put the pointer to the last visited chunk in `last`, so we can access it during the extension without traversing the whole list again.

- Otherwise: we extended the heap (which is *empty* at that point.)

  Note that our function `extend_heap` works here with `last=NULL`.

Note also that each time we fail, we silently returns `NULL` as expected by the specification of `malloc`. The Listing 1 presents the code.

Listing 1: The `malloc` function

```
1  void *malloc(size_t size){
2    t_block                b,last;
3    size_t                s;
4    s = align4(size);
5    if (base) {
6      /* First find a block */
7      last = base;
8      b = find_block(&last,s);
9      if (b) {
10       /* can we split */
11       if ((b->size - s) >= (BLOCK_SIZE + 4))
12         split_block(b,s);
13       b->free=0;
14     } else {
15       /* No fitting block, extend the heap */
16       b = extend_heap(last,s);
17       if (!b)
18         return(NULL);
19     }
20   } else {
21     /* first time */
22     b = extend_heap(NULL,s);
23     if (!b)
24       return(NULL);
25     base = b;
26   }
27   return(b->data);
28 }
```

6 `calloc`, `free` and `realloc`

6.1 `calloc`

`calloc(3)` is quite straightforward:

- First do the `malloc` with right size (product of the two operands);

- Put `0` on any bytes in the block.

We just play a little trick: the data block size in the chunk is always a multiple of 4, so iterate by 4 bytes steps. For this, we just use our new pointer as an unsigned integer array. The code is straightforward (see listing 2.)

Listing 2: The `calloc` function

```
1  void *calloc(size_t number, size_t size){
2    size_t       *new;
3    size_t        s4,i;
4    new = malloc(number * size);
5    if (new) {
6      s4 = align4(number * size) << 2;
7      for (i=0; i<s4 ; i++)
8        new[i] = 0;
9    }
10   return (new);
11 }
```

6.2 `free`

A quick implementation of `free(3)` can be quiet simple, but this does not mean that it is convenient. We have two issues: find the chunk to be freed and avoiding space fragmentation.

6.2.1 Fragmentation: the `malloc` illness

A major issue of `malloc` is fragmentation: after several use of `malloc` and `free`, we end with a heap divided in many chunks individually to small to satify *big* `malloc` while the whole free space would have been sufficient. This issue is known as the space fragmentation problem. While we cannot go against the extra-fragmentation due to our algorithm without changing it, some other sources of fragmentation can be avoided.

When we select a free chunk wider enough to hold the requested allocation and another chunk, we split the current chunk. While this offer a better usage of the memory (the new chunk is free for further reservation) it introduces more fragmentation.

A solution to eliminate some fragmentation is to *fusion* free chunks. When we free a chunk, and its neighbors are also free, we can fusion them in one bigger chunk. All we have to do is to test the next and previous chunks. But, how to find the previous block ? We have several solitions:

- search from the begin, particulary slow (especially if we already do a search for the freed chunk)
- if we already do a search for the current chunk, we could keep a pointer on the last visited chunk (as for `find_block`)
- Double link our list.

We choose the last solution, it's simpler and let's us do some trick to find the target chunk. So, we modify (again) our `struct s_block`, but since we have another pending modification (see the next section) we will present the modification later.

So, all we have to do now is the fusion. We first write a simple fusion function which fusion a chunk and its successor. Fusionning with the predecessor will just be a test and a call with the right chunk. In the code we use our new field `prev` for the predecessor.

```
1  t_block fusion(t_block b){
2    if (b->next && b->next->free){
3      b->size += BLOCK_SIZE + b->next->size;
4      b->next = b->next->next;
5      if (b->next)
6        b->next->prev = b;
7    }
8    return (b);
9  }
```

Fusion is straightforward: if the next chunk is free, we sum the sizes of the current chunk and the next one, plus the meta-data size. Then, the we make `next` field point to the successor of our successor and if this successor exists we update its predecessor.

6.2.2 Finding the right chunk

The other `free`'s issue is to find, efficiently, the correct chunk with only the pointer returned by `malloc`. In fact, there are several issues here:

- Validating the input pointer (is it realy a *malloc'ed* pointer ?)
- Finding the meta-data pointer

We can eliminate most of invalid pointers by a quick range test: if the pointer is outside the heap, it can not be a valid pointer. The remaining cases are related to the last case. How can we be sure that a pointer was obtained by a `malloc` ?

A solution is to have some *magic number* inside the block structure. And better thant a magic number, we could use the pointer itself. I explain: say we have field `ptr` pointing to the field `data`, if `b->ptr == b->data`, then `b` is probably (very probably) a valid block. So, here is the extended structure and functions that verify and access the bloc corresponding to a given pointer:

```
1  /* block struct                                          */
2  struct s_block {
3    size_t             size;
4    struct s_block    *next;
5    struct s_block    *prev;
6    int                free;
7    void              *ptr;
8    /* A pointer to the allocated block */
9    char                        data[1];
10 };
11 typedef struct s_block *t_block;
12
13 /* Get the block from and addr                    */
14 t_block get_block(void *p)
```

```
15 {
16   char *tmp;
17   tmp = p;
18   return (p = tmp -= BLOCK_SIZE);
19 }
20
21 /* Valid addr for free                              */
22 int valid_addr(void *p)
23 {
24   if (base)
25     {
26       if ( p>base && p<sbrk(0))
27         {
28           return (p == (get_block(p))->ptr);
29         }
30     }
31   return (0);
32 }
```

6.2.3 The `free` function

`free` is now straightforward: we verify the pointer and get the corresponding chunk, we mark it free and fusion it if possible. We also try to release memory if we're at the end of the heap.

Releasing memory, is quite simple: if we are at the end of the heap, we just have to put the break at the chunk position with a simple call to `brk(2)`.

the listing 3 on the following page presents our implementation, it follows this structure:

- If the pointer is valid:
  - we get the block address
  - we mark it free
  - if the previous exists and is free, we step backward in the block list and fusion the two blocks.
  - we also try fusion with then next block
  - if we're the last block we release memory.
  - if there's no more block, we go back the original state (set `base` to `NULL`.)
- If the pointer is not valid, we silently do nothing.

6.3 Resizing chunks with `realloc`

The `realloc(3)` function is almost as straightforward as `calloc(3)`. Basically, we only need a memory copy operation. We won't use the provided `memcpy`, since we can do better (size are available in blocks and are aligned.)

The copy is straightforward:

Listing 3: The `free` function

```
/* The free                                                  */
/* See free(3)                                               */
void free(void *p)
{
  t_block b;
  if (valid_addr(p))
    {
      b = get_block(p);
      b->free = 1;
      /* fusion with previous if possible */
      if(b->prev && b->prev->free)
        b = fusion(b->prev);
      /* then fusion with next */
      if (b->next)
        fusion(b);
      else
        {
          /* free the end of the heap */
          if (b->prev)
            b->prev->next = NULL;
          else
            /* No more block !*/
            base = NULL;
          brk(b);
        }
    }
}
```

```
/* Copy data from block to block                  */
void copy_block(t_block src, t_block dst)
{
  int           *sdata,*ddata;
  size_t         i;
  sdata = src->ptr;
  ddata = dst->ptr;
  for (i=0; i*4<src->size && i*4<dst->size; i++)
    ddata[i] = sdata[i];
}
```

A very naive (but working) `realloc`, just follow this algorithm:

- Allocate a new bloc of the given size using `malloc`;
- Copy data from the old one to the new one;
- Free the old block;
- Return the new pointer.

Of course, we want something a little bit more efficient: we don't want a new allocation if we have enough room where we are. The different cases are thus:

- If the size doesn't change, or the extra-available size (do to alignement constraint, or if the ramainning size was to small to split) is sufficient, we do nothing;
- If we shrink the block, we try a split;
- If the next block is free and provide enough space, we fusion and try to split if necessary.

The listing 4 on the next page presents our implementation. Don't forget, that the call `realloc(NULL,s)` is valid and should be replaced by `malloc(s)`.

Listing 4: The `realloc` function

```
/* The realloc                                          */
/* See realloc(3)                                       */
void *realloc(void *p, size_t size)
{
  size_t        s;
  t_block       b, new;
  void          *newp;
  if (!p)
    return (malloc(size));
  if (valid_addr(p))
    {
      s = align4(size);
      b = get_block(p);
      if (b->size >= s)
        {
          if (b->size - s >= (BLOCK_SIZE + 4))
            split_block(b,s);
        }
      else
        {
          /* Try fusion with next if possible */
          if (b->next && b->next->free
              && (b->size + BLOCK_SIZE + b->next->size) >= s)
            {
              fusion(b);
              if (b->size - s >= (BLOCK_SIZE + 4))
                split_block(b,s);
            }
          else
            {
              /* good old realloc with a new block */
              newp = malloc(s);
              if (!newp)
                /* we're doomed ! */
                return (NULL);
              /* I assume this work ! */
              new = get_block(newp);
              /* Copy data */
              copy_block(b,new);
              /* free the old one */
              free(p);
              return (newp);
            }
        }
      return (p);
    }
  return (NULL);
}
```

6.3.1 FreeBSD's `reallocf`

FreeBSD provides another `realloc` function: `reallocf(3)` which free the given pointer in any case (even if the reallocation fails.) This just a call to `realloc` and a `free` if we get a `NULL` pointer. The listing 5 presents the code.

Listing 5: The `reallocf` function

```
1  /* The reallocf                                        */
2  /* See reallocf(3)                                     */
3  void *reallocf(void *p, size_t size)
4  {
5    void *newp;
6    newp = realloc(p,size);
7    if (!newp)
8      free(p);
9    return (newp);
10 }
```

6.4 Putting things together

All need, now is to integrated modification done to the block structure in previous code. We only need to *rewrite* `split_block` and `extend_heap` and redefine `BLOCK_SIZE`.

Listing 6: Putting things together

```
/* block struct                                                */
struct s_block {
  size_t             size;
  struct s_block    *next;
  struct s_block    *prev;
  int                free;
  void              *ptr;
  /* A pointer to the allocated block */
  char                         data[1];
};
typedef struct s_block *t_block;

/* Define the block size since the sizeof will be wrong        */
#define BLOCK_SIZE 20

/* Split block according to size.                              */
/* The b block must exist.                                     */
void split_block(t_block b, size_t s)
{
  t_block       new;
  new       = (t_block)(b->data + s);
  new->size = b->size - s - BLOCK_SIZE;
  new->next = b->next;
  new->prev = b;
  new->free = 1;
  new->ptr  = new->data;
  b->size   = s;
  b->next   = new;
  if (new->next)
    new->next->prev = new;
}

/* Add a new block at the of heap                              */
/* return NULL if things go wrong                              */
t_block extend_heap(t_block last, size_t s)
{
  int           sb;
  t_block       b;
  b       = sbrk(0);
  sb      = (int)sbrk(BLOCK_SIZE + s);
  if (sb < 0)
    return (NULL);
  b->size = s;
  b->next = NULL;
  b->prev = last;
  b->ptr  = b->data;
  if (last)
    last->next = b;
  b->free = 0;
  return (b);
}
```

List of Figures

Listings