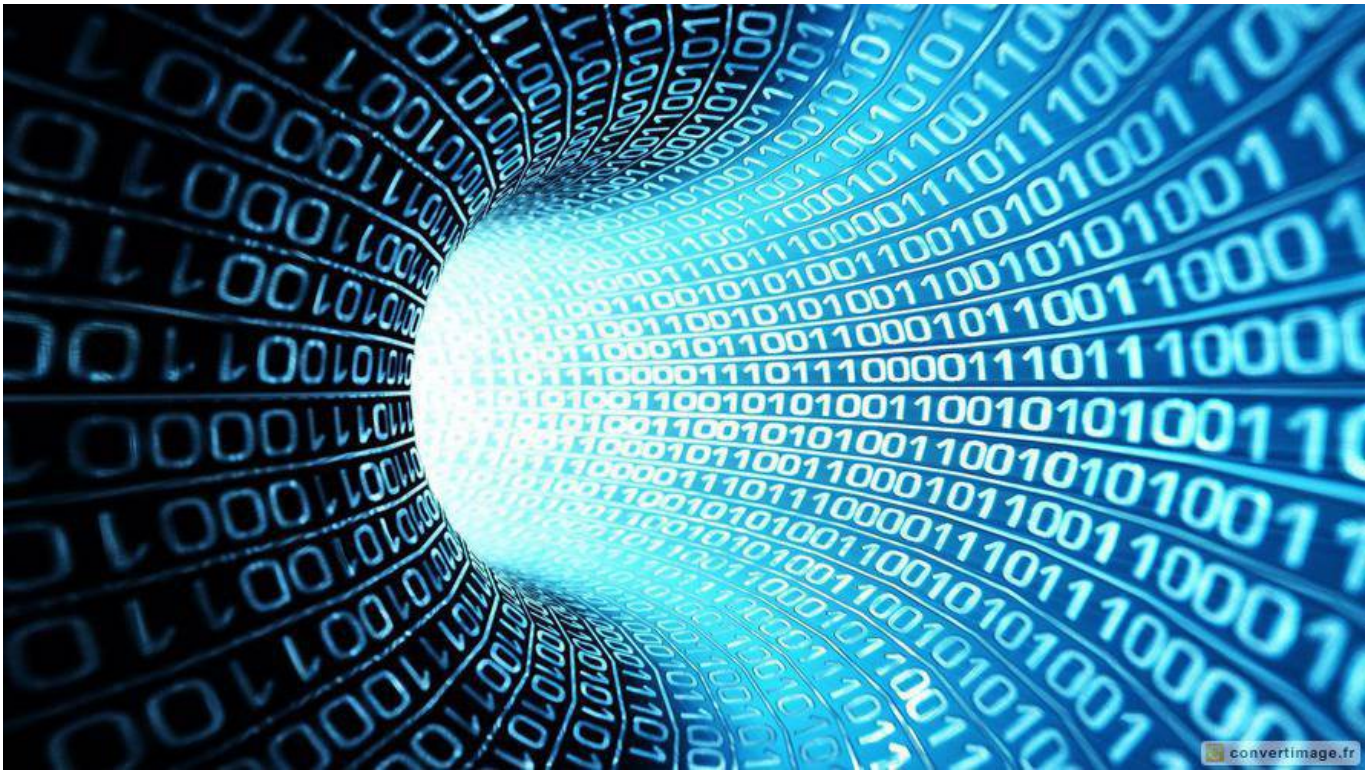
André Carvalho

141 Followers About Follow

Implementing malloc and free

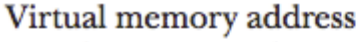
 André Carvalho Jun 14, 2017 · 5 min read

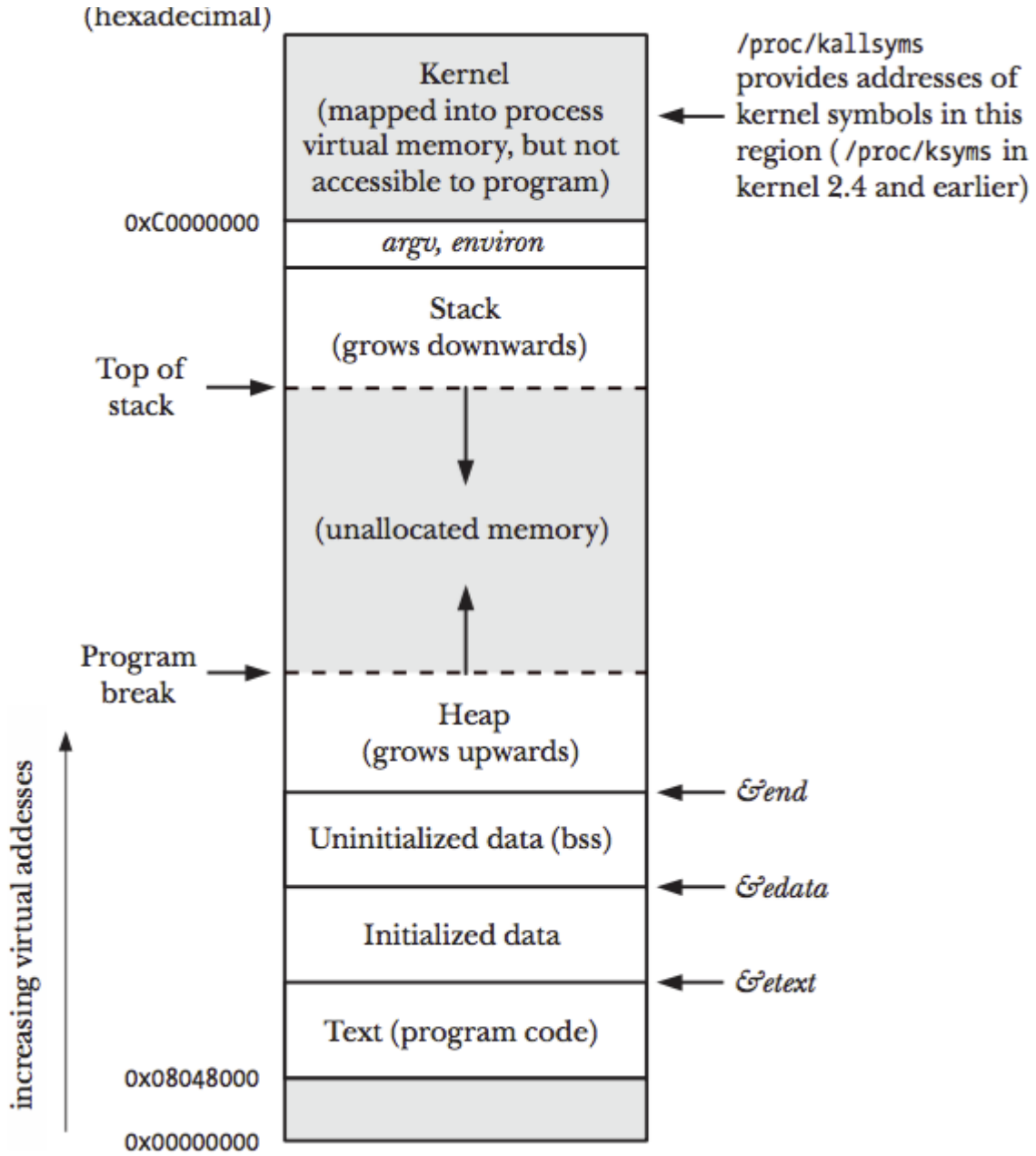


Chapter 7 of "The Linux Programming Interface" is about memory allocation. One of the exercises, marked as advanced, asks the reader to implement **malloc**. I've decided to give it a shot. My full implementation is available on Github; I will try to break down some of my reasoning on the next sections and include some snippets from the code.

Memory layout of a Process

The memory allocated to each process is composed of multiple segments, as can be seen on the following image:





Source: https://github.com/shichao-an/notes/blob/master/docs/tlpi/figure_6-1.png

We are particularly interested on the heap (also known as the data segment), an area from which memory can be dynamically allocated at run time. The top end of the heap is called the **program break**.

Adjusting the program break

We can move the program break on our C program by using sbrk() and brk().

```
int brk(void *addr);

void *sbrk(intptr_t increment);
```

The first, moves the program break to the address pointed by *addr*, while the latter increments the program break by *increment* bytes. Their man pages can give more

information about their implementation on Linux and other systems, but those are the basic building blocks that our malloc implementation will rely on. On Linux, **sbrk** relies on **brk**.

The implementation

The entire code for the implementation is available at github. Beware that this implementation is full of bugs (some are discussed below, some are obvious from reading the real malloc implementation). The following code is the malloc function implementation.

```
1   void           *
2   _malloc(size_t size)
3   {
4           void           *block_mem;
5           block_t        *ptr, *newptr;
6           size_t          alloc_size = size >= ALLOC_UNIT ? size + sizeof(block_t)
7                   : ALLOC_UNIT;
8           ptr = head;
9           while (ptr) {
10                  if (ptr->size >= size + sizeof(block_t)) {
11                          block_mem = BLOCK_MEM(ptr);
12                          fl_remove(ptr);
13                          if (ptr->size == size) {
14                                  // we found a perfect sized block, return it
15                                  return block_mem;
16                          }
17                          // our block is bigger then requested, split it and add
18                          // the spare to our free list
19                          newptr = split(ptr, size);
20                          fl_add(newptr);
21                          return block_mem;
22                  } else {
23                          ptr = ptr->next;
24                  }
25          }
26          /* We are unable to find a free block on our free list, so we
27           * should ask the OS for memory using sbrk. We will alloc
28           * more alloc_size bytes (probably way more than requested) and then
29           * split the newly allocated block to keep the spare space on our free
30           * list */
31          ptr = sbrk(alloc_size);
32          if (!ptr) {
33                  printf("failed to alloc %ld\n", alloc_size);
34                  return NULL;
35          }
36          ptr->next = NULL;
37          ptr->prev = NULL;
```

```
38          ptr->size = alloc_size - sizeof(block_t);
39          if (alloc_size > size + sizeof(block_t)) {
40                  newptr = split(ptr, size);
41                  fl_add(newptr);
42          }
43          return BLOCK_MEM(ptr);
44  }
```

_malloc.c hosted with ❤️ by GitHub view raw

Our implementation keeps a doubly linked listed of free memory blocks and every time **_malloc** gets called, we traverse the linked list looking for a block with at least the size requested by the user (lines 8–25). If a block with the exact requested size exists, we remove it from the list and return its address to the user (lines 11–16); if the block is larger, we split it into two blocks, return the one with the requested size to the user and adds the newly created block to the list (lines 19–21).

If we are unable to find a block on the list, we must "ask" the OS for more memory, by using the **sbrk** function (lines 31–35). To reduce the number of calls to sbrk, we alloc a fixed number of bytes that is a multiple of the memory page size, defined as:

```
#define ALLOC_UNIT 3 * sysconf(_SC_PAGESIZE)
```

After the call to sbrk (where our program break changes value) we create a new block with the allocated size. The metadata on this block contains the size, next and previous blocks and is allocated on the first 24 bytes of the block (this is our overhead) (lines 36–38). Since we may have allocated much more memory then the user requested, we split this new block and return the one with the exact same size as requested (lines 39–43).

The BLOCK_MEM macro, defined as:

```
#define BLOCK_MEM(ptr) ((void *)((unsigned long)ptr +
sizeof(block_t)))
```

returns skips the metadata at given **ptr** and returns the address of the memory area that is available for the user.

The _free function is quite straightforward, given a pointer that was previously "malloced" to the user, we must find its metadata (by using the BLOCK_HEADER macro)

and add it to our free linked list. After that, the function scan_merge() is called to do some cleaning:

```
/* scan_merge scans the free list in order to find
 * continuous free blocks that can be merged and also
 * checks if our last free block ends where the program
 * break is. If it does, and the free block is larger then
 * MIN_DEALLOC then the block is released to the OS, by
 * calling brk to set the program break to the begin of
 * the block */
void
scan_merge()
{
        block_t        *curr = head;
        unsigned long   header_curr, header_next;
        unsigned long   program_break = (unsigned long)sbrk(0);
        if (program_break == 0) {
                printf("failed to retrieve program break\n");
                return;
        }
        while (curr->next) {
                header_curr = (unsigned long)curr;
                header_next = (unsigned long)curr->next;
                if (header_curr + curr->size + sizeof(block_t) == header_next) {
                        /* found two continuous addressed blocks, merge them
                         * and create a new block with the sum of their sizes */
                        curr->size += curr->next->size + sizeof(block_t);
                        curr->next = curr->next->next;
                        if (curr->next) {
                                curr->next->prev = curr;
                        } else {
                                break;
                        }
                }
                curr = curr->next;
        }
        stats("after merge");
        header_curr = (unsigned long)curr;
        /* last check if our last free block ends on the program break and is
         * big enough to be released to the OS (this check is to reduce the
         * number of calls to sbrk/brk */
        if (header_curr + curr->size + sizeof(block_t) == program_break
            && curr->size >= MIN_DEALLOC) {
                fl_remove(curr);
                if (brk(curr) != 0) {
                        printf("error freeing memory\n");
                }
        }
```

```
46  }
```

scan_merge.c hosted with ❤ by GitHub view raw

scan_merge() first traverses the linked list looking for continuous blocks (two different memory blocks that are free and correspond to continuous addresses). We keep the blocks sorted by address to make this step easier. For every two continuous blocks found, we merge both blocks to reduce our total overhead (less metadata to keep) (lines 18–33).

After finding the last block on our free list, we check if this blocks ends on the program break (line 39). If that is true, and the block is big enough (where "big" is defined as MIN_DEALLOC, also a multiple of the page size), we remove the block from our list and move the program break to the beginning of the block, by calling **brk**.

How is malloc actually implemented?

Before diving into the real malloc code, I decided to write a simple test program and trace it's execution using **strace**. I used the following code:

```
1  #include <malloc.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5   malloc(atoi(argv[1]));
6  }
```

malloc_test.c hosted with ❤ by GitHub view raw

I decided to trace the executing testing different sizes.

```
$ strace ./malloc 1
...
brk(NULL)                               = 0x5585209f2000
brk(0x558520a13000)                     = 0x558520a13000
exit_group(0)                           = ?
+++ exited with 0 +++

$ strace ./malloc 100000
...
brk(NULL)                               = 0x55b45a386000
brk(0x55b45a3bf000)                     = 0x55b45a3bf000
exit_group(0)                           = ?

$ strace ./malloc 1000000
...
mmap(NULL, 1003520, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
```

```
-1, 0) = 0x7f05f7cbf000
exit_group(0)                           = ?
```

The first thing I noticed between executions is that when asking for a large size, malloc actually used **mmap** instead of **brk** to allocate the memory. I wasn't sure why, since I have yet to study mmap. But checking the code for glibc's malloc I've found a pretty nice writeup made by the authors explaining their implementation. I highly recommend reading it. Regarding the use of mmap, from the source code comments:

```
This backup strategy generally applies only when systems have
"holes" in address space, so sbrk cannot perform contiguous
expansion, but there is still space available on system.   On
systems for which this is known to be useful (i.e. most linux
kernels), this occurs only when programs allocate huge amounts of
memory.  Between this, and the fact that mmap regions tend to be
limited, the size should be large, to avoid too many mmap calls and
thus avoid running out of kernel resources.
```

The documentation also explains their design goals, motivations, how the blocks are organized (by using bins for the different sizes, instead of keeping a linked list sorted by address, as I did) and lots of other details. I learned a lot reading it.

Adding a call to free on my test program does not change the syscalls made by at, as the memory is not released to the OS (some people rely on this behavior "mallocing" a large chunk of memory and freeing it on the start of a program to reserve the memory). One can control this behavior by defining M_TRIM_THREASHOLD:

```
M_TRIM_THRESHOLD is the maximum amount of unused top-most memory  to
keep before releasing via malloc_trim in free().  Automatic trimming
is mainly useful in long-lived programs.  Because trimming via sbrk
can be slow on some systems, and can  sometimes be wasteful (in
cases where programs immediately  afterward allocate more large
chunks) the value should be high  enough so that your overall system
performance would improve by  releasing this much memory.
```

This blog post is part of a series of posts that I intent to write while reading "The Linux Programming Interface" to make sure I'm actually learning something, as a way to practice and to share knowledge.

Thanks to Guilherme B. Rezende.

Programming Linux C Operating Systems