Memory Management in GRUB2

Vincent Guffens
guffens@inma.ucl.ac.be

August 2, 2005

GRUB2 has been designed with extensibility in mind. Among others features that contribute to that goal, GRUB2 allows for dynamics memory allocation providing its programmers with an equivalence of the familiar malloc()/free() *system calls. This article explains how the GRUB2 memory management that supports this feature is implemented. The memory management of GRUB2 was written by Yoshinori K. Okuji, credit for other developers may be found in the GRUB2 THANKS file.*

Introduction

Memory management, also known as heap management is a classical problem and an extensive treatment of the subject can be found in the literature, for instance in Knuth, The Art of Computer Programming, vol. 1. However, GRUB2 implements a simple solution requiring a small amount of code and it may therefore be used as a practical case study that complements the concepts usually described in reference books.

Useful structures and functions

GRUB2 supports multiple non-continuous memory zones that are called regions. Each region is described by the following structure

```
typedef struct grub_mm_region
{
  struct grub_mm_header *first;
```

```
  struct grub_mm_region *next;
  grub_addr_t addr;
  grub_size_t size;
}
*grub_mm_region_t;
```

These regions are linked together using their *next* pointer and a static variable *base* declared in *kern/mm.c* points to the smallest regions. As these regions will be later inspected in that order to allocate new memory, this will ensure that the smallest memory regions are used first.

As memory blocks get allocated and de-allocated, the regions get fragmented and it is therefore required to divide them into smallest blocks, called chunks that will be marked as free or allocated. The important structure used to that end is :

```
typedef struct grub_mm_header
{
  struct grub_mm_header *next;
  grub_size_t size;
  grub_size_t magic;
  char padding[4];
}
*grub_mm_header_t;
```

which is displayed here for 32 bits machines.

The functions relevant to the memory allocation that are exported to be used by other modules are :

```
void *grub_malloc (grub_size_t size);
void grub_free (void *ptr);
void *grub_realloc (void *ptr, grub_size_t size);
void *grub_memalign (grub_size_t align, grub_size_t size);
```

These functions with their arguments are self-explanatory and are similar to their libc counterpart except for *grub_memalign* which returns a pointer with a given alignment. A pointer p is said to have an alignment $ALIGN$ if p is a multiple of $ALIGN$ or equivalently if

$$p\%ALIGN = 0$$

Alignment

It is important to note that every region and every chunk are always aligned on a *GRUB_MM_ALIGN* boundary. For 32 bits architectures, *GRUB_MM_ALIGN* is defined as $2^4 = 16$. One can easily check that the structures presented above also have a size of 16 bytes. As a consequence, every pointer returned by *grub_malloc* and every pointer passed as argument to *grub_free* will be a multiple of 16.

Initialisation

The initialisation of the memory regions is done during the initialisation of GRUB2 using the *grub_mm_init_region (void *addr, grub_size_t size)* which will initialise a free region starting at address *addr* and whose size is *size* (in bytes). The regions to be added are given by the BIOS using some assembly function defined in *kern/i386/pc/startup.S*. The region containing the modules to be loaded at startup is also added as a free region.

The address *addr* is first modified to be the next greater or equal multiple of *GRUB_MM_ALIGN* and a structure *grub_mm_region* is written at that memory address followed by a *grub_mm_header* which indicates that the entire region is a free chunk. The size held by the region structure is in bytes while the size held by the chunk header is in multiple of *GRUB_MM_ALIGN*.

After initialisation, the memory layout can be depicted as shown in Fig. 1.

Allocating memory

The *grub_malloc()* function simply calls *grub_memalign()* with an alignment of 0. This function first calculate the numbers of blocks n (in multiple of 16) that must be allocated and add one block for the header as follows :

```
n = ((size + GRUB_MM_ALIGN - 1) >> GRUB_MM_ALIGN_LOG2) + 1;
```

Using the *base* pointer, the memory allocation is tried in each region until it succeed. If none of the regions has room for the allocation, *grub_memalign()* tries to recover some free memory by invalidating the disk cache and unloading some unused modules. The actual memory allocation takes place in

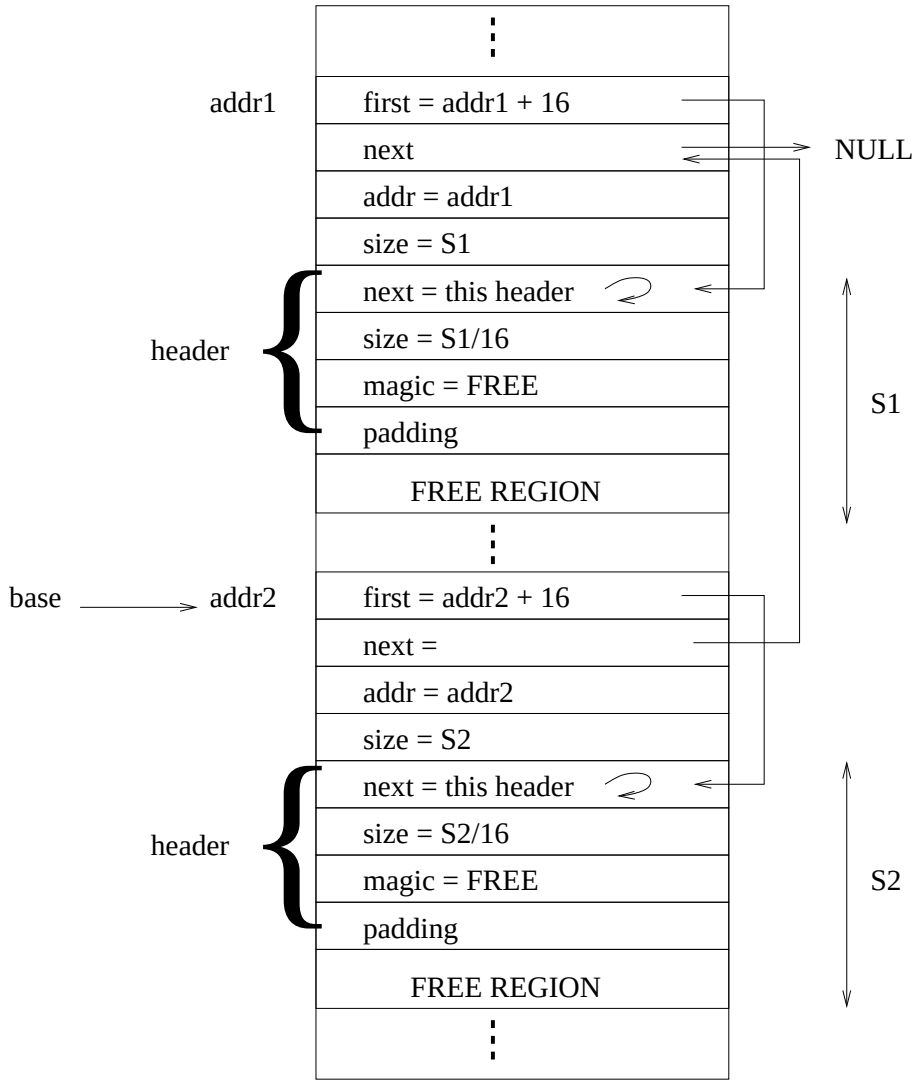

Figure 1: Memory layout after initialisation of 2 non-continuous free regions of size S_1 and S_2 bytes ($S_2 < S_1$). The pointers *addr1*, *addr2* as well as the addresses of the two depicted headers are multiple of 16.

```
grub_real_malloc (grub_mm_header_t *first,
                  grub_size_t n, grub_size_t align)
```

The principle is to maintain a circular linked list of all the free chunks in a region. Allocated chunks are marked with a header which is not in a linked list but which can be referenced by the pointer returned by *grub_malloc*.

Allocation without special alignment

The allocation of a memory block of size m (in multiple of 16) is represented in Fig. 2. Using the *first* pointer of the memory region, the free chunk list is inspected until a chunk of size greater than m is found. If the size of this chunk is N_2, a new header is then written at an offset $N_2 - m$ and marked as being allocated. The memory layout after this operation can be seen in Fig. 3. The header of the free chunks that has just been used is modified to reflect its new size which is now $N_2 - m$.

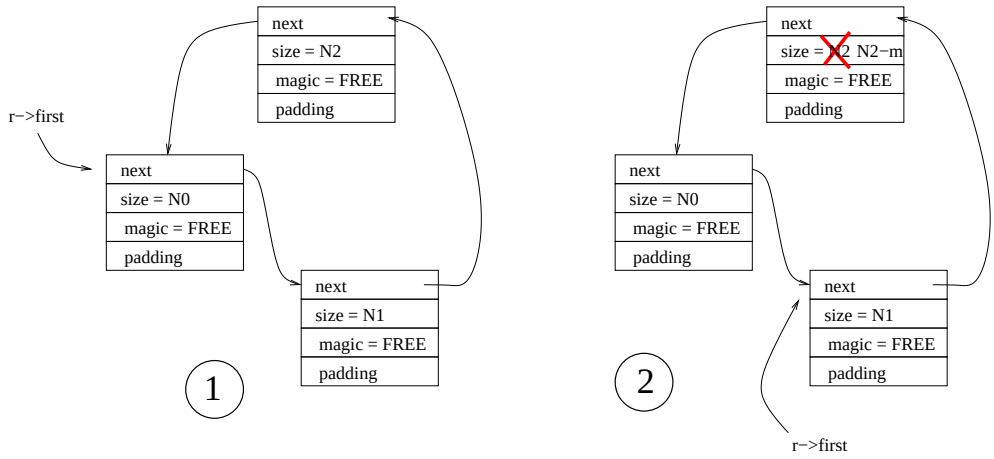

Figure 2: Allocation of a block of size m in a ring having 3 free chunks of size N_0, N_1, N_2 with $N_1 < m < N_2$.

The region pointer that indicates the first free chunk of the region ($r->first$) is advanced in order to avoid the accumulation of small free chunks at the head of the list. This would impact the efficiency of the algorithm as this list of small chunks is traversed for each new allocation.

Some special allocation cases are :

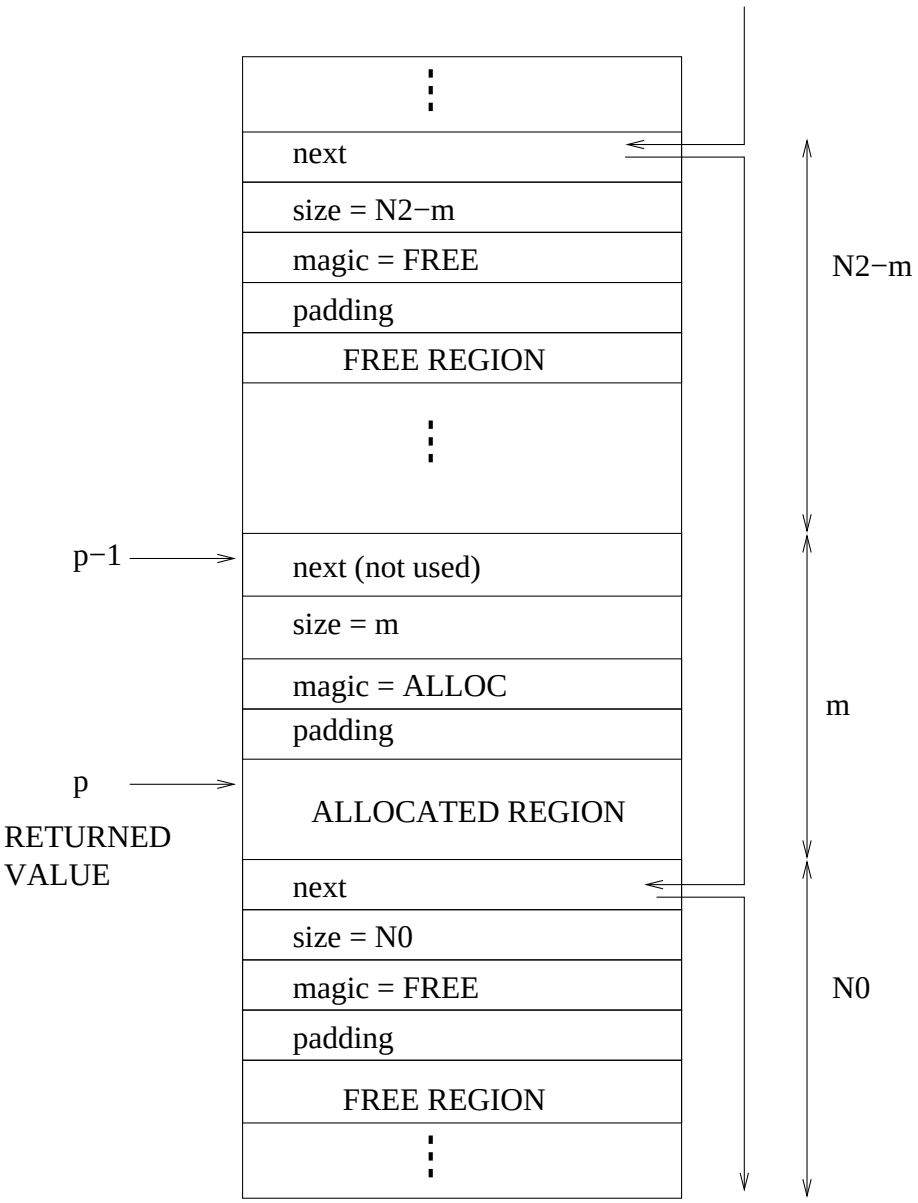

Figure 3: Memory layout after the allocation shown in Fig. 2.

- Allocation of a block that fit exactly in a free chunk : In that case, the free header is simply marked as allocated and the header is removed from the circular buffer list.
- Allocation of an entire region : In that case, there is a single free header in the circular list and this header is simply marked as allocated. This is the only case where a header in the free list may be marked as allocated.

Allocation with alignment requirement

The function *grub_memalign()* may be used to allocate memory having a special alignment requirement. As already mentioned, each allocated chunk is necessarily aligned on a 16 bytes boundary and therefore the alignment requirement is divided by 16 before being passed to the *grub_real_malloc()* function.

The offset needed to satisfy the alignment *align* is first calculated as follows :

```
extra = ((grub_addr_t) (p + 1) >> GRUB_MM_ALIGN_LOG2) % align;
if (extra)
  extra = align - extra;
```

where p is the address of the free header which is being tried. Let's say that the size of the memory block that must be allocated is n and the size of the free chunk is $m + n + extra$ with $m > 0$. The correct alignment is then obtained by splitting the original free chunk into 3 smaller chunks as shown in Fig. 4. The first chunk receives a size equal to $extra$ so that the next chunk begins at address $p + 1 + extra$ which has the correct alignment. The last chunk receives the remaining free space. As above, there is a special case when $m = 0$ as in this case, the third region does not exist.

Freeing memory

Freeing memory is conceptually very simple as it suffice to mark the allocated chunk as free and add this new free chunk into the circular linked list. The problem however is worsened a little bit as the previous free chunk which should be used to add the freshly freed chunk is not known, even after the header of the chunk to free has been recovered. Therefore, GRUB2 uses a function

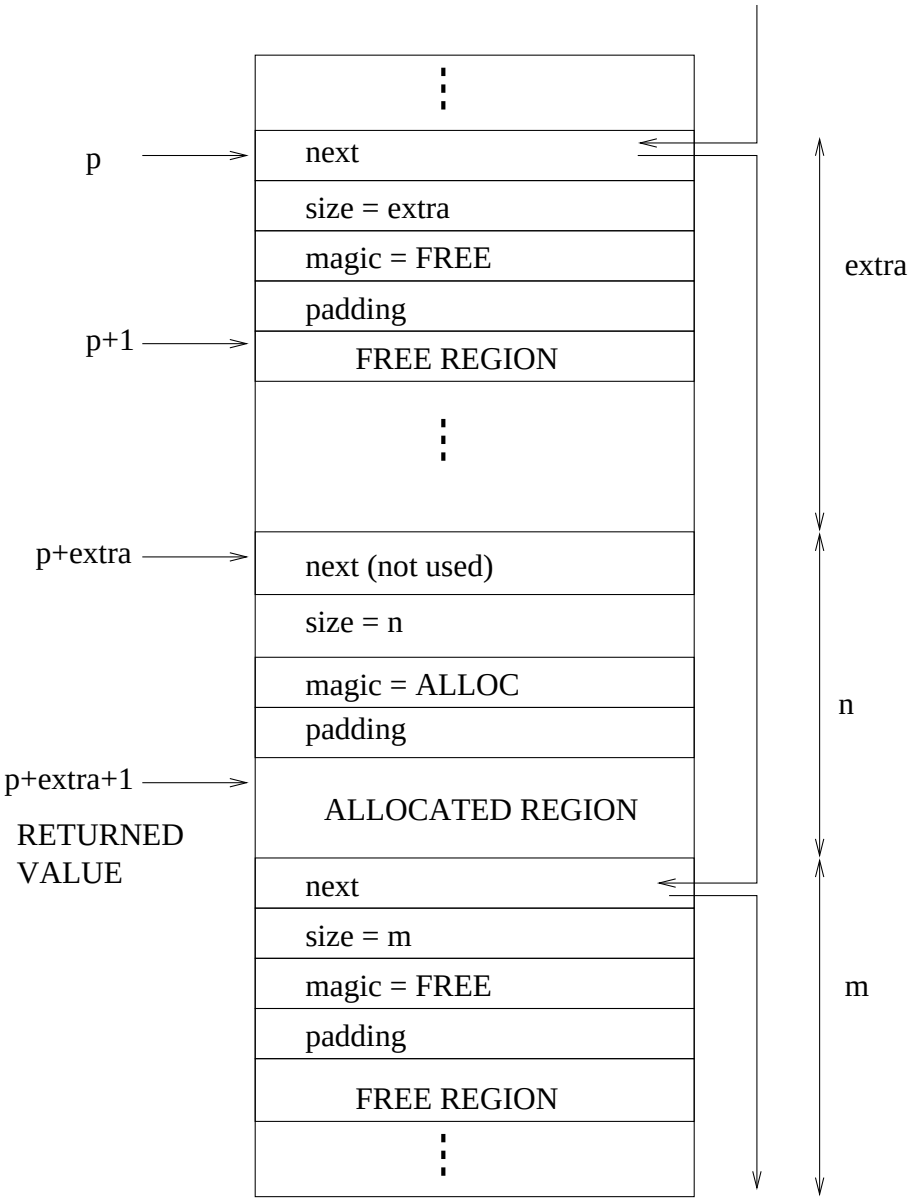

Figure 4: Allocation of a memory block of size n with special alignment in a free chunk of size $N_0 = extra + n + m$.

```
get_header_from_pointer (void *ptr, grub_mm_header_t *p,
                         grub_mm_region_t *r)
```

which, given an allocated buffer *ptr*, returns its header p and the its region r. As already mentioned, the header can be simply recovered using $p = ptr - 1$. The region is found by going through the region list and finding in which one does the *ptr* pointer belongs. Knowing the region, it is possible to go through the circular list of free chunks until one finds the free chunk to be used. Let's called q a pointer toward the header of this chunk and p a pointer toward the header of the chunk to be freed. These pointers satisfy

$$(q < p)\ \&\&\ (q- > next > p)$$

or

$$q >= q- > next\ \&\&\ (q < p\ ||\ q- > next > p)$$

It is then very simple to free this chunk and add it to the ring as follows :

```
p->magic = GRUB_MM_FREE_MAGIC;
p->next = q->next;
q->next = p;
```

The second complication arises from the fact that this new free chunk might be contiguous with others free chunks. This situation occurs if

$$(p + p- > size == p- > next)\ ||\ (q + q- > size == p)$$

In these cases, the adjacent header chunk is unlinked and the size of the remaining free chunk is updated accordingly.

Reallocating memory

The reallocation strategy is quite simple. It allocates a new buffer of suitable size (if the new size is strictly greater than the old one), copies the content of the old buffer into it and finally frees the old buffer.