Computation Structures - Project 2
β-allocator: dynamic memory allocation for β-assembler programs

November 5, 2020

General information

- **Deadline: November 27, 2020, 23:59**.
- Project must be done by **groups of two people**.
- There will be a penalty for late submission.
- Questions will no longer be answered 24 hours before the deadline.
- English is strongly recommended (but French is allowed).
- Contact: `r.mormont@uliege.be`

1 Introduction

The goal of this project is to make you more familiar with the β-assembly language by getting your hands dirty and writing code. In this project, you will implement dynamic memory allocation for β-assembler code (similar to `malloc` in C).

1.1 Memory allocation

Usually, one distinguishes **static** and **dynamic memory** allocation. The former requires the amount memory to allocate to be known when the program starts. For instance, this is the case for static arrays which are either allocated on the stack or in the data memory segment of the program. In high-level languages, static memory is usually handled by the compiler which generates the necessary instructions for allocating and freeing it. However, this type of memory is sometimes too restrictive because one wants to allocate an amount of memory which can only be known during the program execution. Moreover, one might want to have more control on the lifetime of the allocated memory (e.g. when to free it). This is the purpose of *dynamic memory allocation*.

The β-assembly language already features *static memory allocation* through the macros `ALLOCATE(n)` and `STORAGE(n)`. The former allocates n words on the stack while the latter reserves n words at the current position in the memory, n being known at assembly time. As far as `ALLOCATE(n)` is concerned, it is the programmer's job to make sure that the allocated words are freed from the stack using `DEALLOCATE(n)`.

There is no native mechanism for *dynamic memory allocation* in β-assembly, and it is your task to implement such a mechanism.

1.2 Dynamic memory allocation

1.2.1 Interface

Like in C, the dynamic allocation will be available in β-assembly through procedures called `malloc` and `free` that you will have to implement:

- `int* malloc(int n)`: allocates n words and returns a pointer to the first word of the allocated space.[1]
- `void free(int* p)`: frees the allocated memory space pointed by p

1.2.2 Implementation: the β-allocator

We will call the implementation of the interface presented in Section 1.2.1 the β-allocator. It is a user-space storage allocator allowing dynamic memory allocation in β-assembly. The algorithms and data structures are largely inspired from the storage allocator presented at page 185 of the book "*The C programming language*" by Brian W. Kernighan and Dennis M. Ritchie.

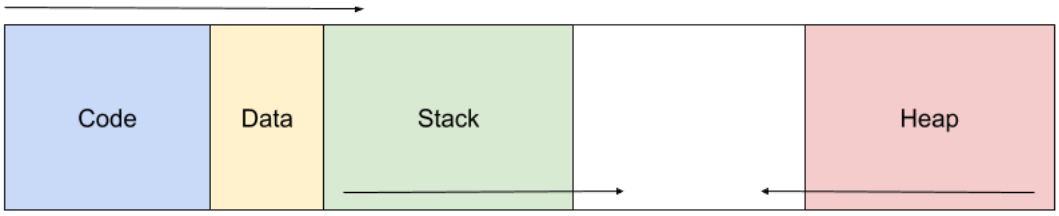

Figure 1: Memory segments with a heap segment for *dynamic memory allocation*. **Code and data** segments are located at the beginning of the memory. The **stack** starts at the end of the data segment and grows towards higher addresses. The **heap** starts at the end of the memory and grows towards lower addresses.

Figure 1 illustrates the memory structure we will use in this project. The **heap** is the area where the β-allocator will work ; it contains a set of memory blocks. A **memory block** is composed of a *two-words header* and a chunk of memory. The first word of the header contains the size (in words) of the block's chunk of memory. The second word contains the address of another block (i.e. a pointer to the header of another block) or the null pointer `NULL`. Note that the header is kept at the lowest address of the memory used by the block.

Each memory block can be either free (i.e. available for allocation) or allocated (i.e. currently used by the program). When `malloc` is called, we have to find a free block that can contain the amount of memory requested by the program. Therefore, all free blocks are organized as a linked list: each free block (actually the second word of the header of this block) points to the next free block in the heap. The last free block in the heap points to `NULL` (see Figure 2 for an example of free memory blocks). Used blocks are not organized as a list and thus the pointer in their header is not used and has the value `NULL`.

[1] For simplicity, we assume here that `malloc(n)` allocates `n` words, unlike the C function which actually allocates `n` bytes.

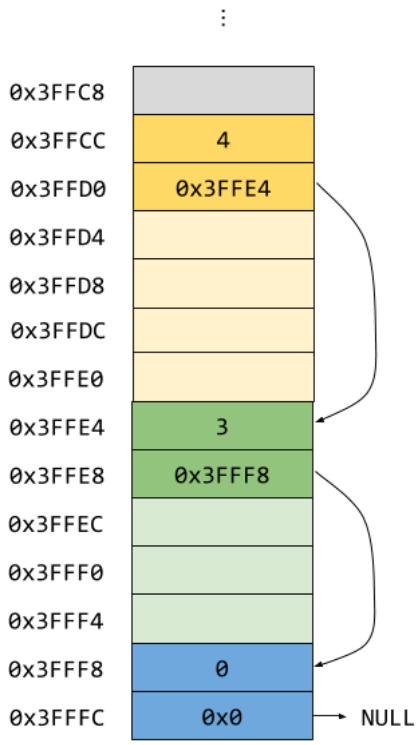

Figure 2: Example of contiguous free memory blocks.

In order to be able to easily retrieve the first block of the heap (i.e. the lowest memory address used by the heap) and first free block of the heap, we will use *two new dedicated registers*: **Base Block Pointer** (`BBP = r26`) and **Free Pointer** (`FP = r25`) (see Figure 3).

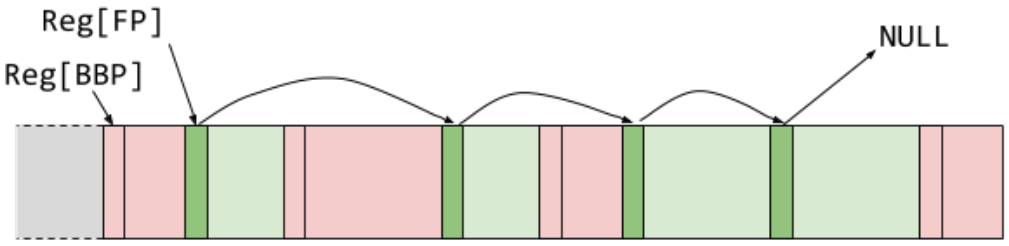

Figure 3: Example of heap with the list of free blocks. The free blocks are green while the currently used blocks are red.

1.2.3 Initialization

The heap will be allocated down from word address `0x3FFFC`. When the program starts, the heap is initialized with a single memory block of chunk size zero starting at address `0x3FFF8` and with its next pointer set to `NULL`. `BBP` and `FP` are set to point both to the header of this block (`0x3FFF8`).

1.2.4 Allocation: `malloc`

As explained in Section 1.2.2, when `malloc` is called, one has to scan the free list to find a memory block large enough for storing the n requested words[2]. We can already distinguish two cases:

1. There is a **block large enough** in the free list. We consider that "large enough" means exactly of the required size or at least 2 words larger than the required size. Then the first valid block can be used for the current allocation.

2. There is **no block large enough** in the free list. Then a new block must be created at the beginning of the heap.

If there is a valid block, either its size is exactly equal to the requested size or it is larger. In the first case, one just has to remove the block from the free list. Otherwise, one has to split this block into an allocated block of the requested size and a free block with the remaining memory, which is added to the free list. The pointer to the first word of the memory chunk of the allocated block (i.e. the address of the header plus 8) is then returned by the `malloc` function.

See functions `malloc` and `try_use_block` in `malloc.c` for an example implementation.

1.2.5 Cleaning: `free`

When `free` is called, one first has to find the previous and the next free blocks of the memory chunk pointed by `p`. This can be done by running through the linked list until the address of the header of the current block is greater than `p`. After that, one can insert the block in the free list between the current and previous blocks of the list.

Following the insertion, one final operation is needed to avoid fragmentation. Fragmentation is the process of splitting blocks into smaller and smaller chunks. At one point, the free blocks are so small that they cannot be used anymore for most allocations. Therefore, the storage allocator ends up wasting a lot of memory.

To alleviate the problem of fragmentation, a simple solution consists in merging contiguous free blocks in the list. Indeed, if two contiguous blocks are both free, they can be merged into one larger free block. In our implementation, every time a block is freed and inserted in the list, we will attempt to merge it with the next and previous blocks of the list if they are contiguous.

See functions `free` and `try_merge` in `malloc.c` for an implementation.

2 Project

For this project, you must implement the interface presented in Section 1.2.1 using the mechanisms defined in Section 1.2.2 (implementing another storage allocator policy will result in a penalty). You are provided with `malloc.c`, a C implementation of the interface, that you can use as a basis for your own. You must write your code in the provided file `malloc.asm`. This file already contains the definition of:

[2]We apply a *first-fit* strategy, that we can oppose to a *best-fit* strategy. The latter selects the smallest block that can satisfy the request instead of the first.

- identifiers for the new dedicated registers `BBP` and `FP`
- an identifier `NULL` (for null-pointers)
- the labels for the procedures `malloc` and `free` that you have to implement
- some macros that can be used to call `malloc` and `free`

The names of those macros, identifiers and labels **cannot** be changed and doing so will result in a penalty. You can however create any new label, identifier or macro needed for your implementation.

You are also provided with an example main file `main.asm`. It defines a simple program that initializes the stack and the β-allocator, and then calls `malloc` and `free`.

3 Additional guidelines

3.1 Practical organization

This project will be done by groups of two people. A report of **maximum two pages** can be provided if you want to explain things that are not easy to understand by just looking at the code and comments. Providing a report does not necessarily mean that you will get a better grade; it should be provided only if it brings something that is not mentioned clearly elsewhere (e.g. in the comments).

Plagiarism is of course not tolerated and will be severely punished. Any detected attempt will result in the grade 0/20 for all who have participated in this practice. Any non-detected attempt will result in eternal shame. All help obtained should be properly reported in the report.

You will include your completed `malloc.asm` and your (optional) report (PDF only) in a ZIP archive named `sXXXXXX_NAME1_sYYYYYY_NAME2.zip` where `sXXXXXX` is the student ID of the first student in the group and `NAME1` is his surname in uppercase (similarly, `sYYYYYY` and `NAME2` are respectively the ID and name of the second student). Insert your `malloc.asm` in a ZIP archive even if you do not provide a report. Naming your files differently or submitting other files **will result in a penalty**.

Submit your archive to the **Montefiore Submission Platform**[3] (course INFO0012-2), after having created an account if necessary. If you encounter any problem with the platform, let me know. However problems that unexpectedly and mysteriously appear five minutes before the deadline will not be considered. **Do not send your work by e-mail; it will not be read**.

If you cannot find a partner for the project, you should first post a message on the course forum on e-Campus before sending me an email. Doing so will help other students in the same situation to know that they can team up with you.

3.2 Code guidelines

Choose a coding style and stick to it. You are advised to use the coding style used in `main.asm` and `malloc.asm`. The goal here is not to write code which is as compact and

[3] http://submit.montefiore.ulg.ac.be/

efficient as possible, but to learn the concepts of β-assembly. However, your code should not be unreasonably long and inefficient.

3.3 Documentation

One of the challenges when writing assembly code is to write a program which is relatively easy to understand. Thus, the second most important element taken into account for your grade (after correctness) will be your code's readability. Use comments extensively (your comments can be larger than your code), but don't be verbose : explain the non obvious, not the immediately apparent.

In addition to the comments written alongside your code, all your procedures should be documented using pre- and post-conditions:

- Arguments have to be properly defined
- Any return value must be documented
- Any side effect (e.g. modification of the dynamic memory) must be documented

Your are free (and advised) to use macros to reduce the redundancy in your code. Those macros should also be documented (arguments and side effects).

Good luck and have fun !