# Practical sessions 2, 3, 4: Memory Allocator

Master M1 MOSIG, Grenoble University, 2015-2016

## 1 Overview

This assignment is about implementing and evaluating the performance of classical dynamic memory allocators. To this end, we provide you with a tarball comprising a C skeleton, a Makefile and some non-regression tests.

The first part is devoted to implementing the classical *first-fit* memory allocator. The *first-fit* strategy consists in looking through the list of free memory zones and allocate the first zone which is big enough (meaning that it may be bigger than needed). In the provided sources, you should fill in the C skeleton with the help of the tests provided in the Makefile.
*Passing these tests is considered as mandatory.*

After finishing this part, you should be able to easily implement other allocation strategies like *best-fit* and *worst-fit*, and test them in similar situations (we only provide test cases for the *first-fit* allocation).

The last part of the assignment consists in evaluating the performance of your allocators using simple (but real) programs of your choice. We are interested in characterizing memory fragmentation which can be quantified using different metrics. The aim of this study is to check whether the relative performance of classical heuristics follows indeed what was announced during the lecture.

You should work by groups of two and turn in a `tar.gz` archive named **`Name1_Name2_td2.tar.gz`** (Name1 and Name2 being the names of the two students).

The archive should contain:

- A short design document explaining your choices and the results of your study. This should be a PDF file with your names on the top.
- Your clean and working code. Do not forget to *not include* temporary files, object files, executables, etc.

The archive should be turned in using the Moodle platform
**`http://imag-moodle.e.ujf-grenoble.fr/course/view.php?id=228`**.

# 2 Implementing a Dynamic Memory Allocator

## I. Memory Allocator Interface

We propose to study a system where a fixed amount of memory is allocated at the initialization. This fixed amount of memory (a static array of `char`s will be the only space available for dynamic allocation. The challenge is to provide a mechanism to manage this memory. Managing the memory means :

- to know where the free memory blocks are;
- to slice and allocate the memory for the user when he/she requests it;
- to free the memory blocks when the user indicates that he/she does not need them anymore.

Your futur allocator must implement the following methods :

- `void memory_init(char *mem, int size);`
  Initialize the list of free blocks with a single free block corresponding to the full array.

- `void *memory_alloc(int size);`
  This method allocates a block of size `size`. It returns a pointer to the allocated memory or `NULL` if the allocation failed.

- `void memory_free(void *zone);`
  This method frees the zone addressed by `zone`. It updates the list of the free blocks and merges contiguous blocks.

## II. Memory Allocator Management of Free Blocks

A memory allocator algorithm is based on the principle of linking free memory blocks. Each free block is associated with a descriptor that contains its size and a pointer to the next free block. *This descriptor must be placed inside the managed memory itself.*

The principle of the algorithm is the following :
When the user needs to allocate `block_size` bytes, the allocator must go through to lists of free blocks and find a free block that is big enough. Let `b` be one of these blocks. It may be chosen according to the following criteria :

- **First big enough free block (first fit) :** We choose the first block `b` so that `size(b) >= block_size`. This policy aims at having the fastest research.
- **Smallest waste (best fit) :** We choose the block `b` that has the smallest waste. In other words we choose the block `b` so that `size(b) - block_size` is as small as possible.
- **Biggest waste (worst fit) :** We choose the block so that `size(b) - block_size` is as big as possible.

## III. Allocator Implementation and Validation

The *memory zone* that your allocator will manage tis declared as follows:

```
#define MEMORY_SIZE 512
char memory[MEMORY_SIZE]
```

A pointer to this memory will be passed to the initializing function `memory_init()`. At the beginning, the list of free blocks will be made of a single big free block. The free block descriptor placed at the beginning of a free block is declared like this :

```
struct free_block {
  int size;
  struct free_block *next;
  /* ... */
};
```

**Question III.1:** *Do you have to keep a list for occupied blocks ? If no, explain why, if yes, provide a* `C` *definition of the structure.*

**Question III.2:** *As a user, can you use addresses 0,1,2 ? Generally speaking, can a user manipulate any address?*

**Question III.3:** *When a block is allocated, which address should be returned to the user ?*

**Question III.4:** *When a block is freed by the user, which address is used? What should be done in order to reintegrate the zone in the list of free blocks?*

**Question III.5:** *When a block is allocated inside a free memory zone, one must take care of how the memory is partitioned. In the most simple case, we allocate the beginning of the block for our needs, and the rest of it becomes a new free block. However, it is possible that the rest is too small. How is this possible ? How could you deal with this case?*

The initial code for your allocator is provided in the archive at :
`http://ginf41e0.forge.imag.fr/`
The archive contains :

- `mem_alloc.h`: The interface of your allocator.
- `mem_alloc.c`: A stub for your memory allocator library.
- `mem_shell.c`: A simple program to test interactively your allocator.
- `Makefile`: A Makefile to build and test your memory allocator.
- `README.tests`: A file containing a short description for each provided test.

- `allocX.in and allocX*out*`: `in` files contain respectively example inputs and outputs for your program.

Please note that the provided tests validate only the *first-fit* strategy, provided you keep the given free block structure and sort the list of free blocks by address.

### IV. Integrating Safety Checks

An application programmer can potentially introduce many bugs due to a wrong usage of the dynamic memory allocation primitives. In this section, we ask you to modify your implementation in order to strengthen your allocator against some of these very frequent and harmful bugs.

First of all, make a backup copy of your allocator's source code. Indeed, you will be asked to hand in two versions of your code: the first one without safety checks and the second one implementing safety checks.

The bugs to be considered are listed below:

**Forgetting to free memory** This common error is known as a *memory leak*. In long running applications (or systems, such as the OS itself), this is a big problem, as the accumulation of small leaks may eventually lead to running out of memory. Upon a program exit, your allocator must display a warning message if some of the dynamically allocated memory blocks have not been freed.

**Calling `free()` incorrectly** Such an error can have different incarnations. For instance, a wrongly initialized pointer passed to `free()` may result in an attempt to free a currently unallocated memory zone, or only a fraction of an allocated zone. Such calls may jeopardize the consistency of the allocator's internal data structures. Your allocator must address these erroneous calls (either by ignoring them or by immediately exiting the program and displaying an error message).

**Corrupting the allocator metadata** Arbitrary writes in the memory heap (due to wrong pointers) may potentially jeopardize the consistency of the data structures maintained by the allocator. Although it is not possible to detect all such bugs, a robust allocator can nonetheless notice some inconsistencies (for example, strange addresses in the linked list pointers or strange sizes in the block headers). When the allocator detects such an issue, it must immediately exit the program.

Along with your modified implementation of the allocator, you are asked to provide/describe a set of test programs illustrating the fact that your safety checks work as expected.

## 3 Measuring Fragmentation

To evaluate the fragmentation resulting from the allocation strategies, one should use sound workload [WJNB95]. Therefore, in your tests, you will run *real* programs e.g Firefox, ... and enforce the usage of your memory allocator.

To do so, you will use the provided Makefile to create the `libmalloc.so` library. Then, you will use the `LD_PRELOAD` option) to enable programs to run with your own version of malloc/free/realloc instead of the ones provided by the libc.

Yet, before starting any evaluation, we will need a way to measure fragmentation. A few metrics have been proposed in the literature [JW98]. We reproduce a part the work of Johnstone and Wilson explaining possible approaches:

> *There are a number of legitimate ways to measure fragmentation. Figure 1 illustrates four of these. Figure 1 is a trace of the memory usage of the GCC compiler using a given allocator. The lower line is the amount of live memory requested by GCC (in kilobytes). The upper line is the amount of memory actually used by the allocator to satisfy GCC's memory requests.*
>
> *The four ways to measure fragmentation for a program which we considered are:*
>
> 1. *The amount of memory used by the allocator relative to the amount of memory requested by the program, averaged across all points in time. In Figure 1, this is equivalent to averaging the fragmentation (i.e. the ratio) for each corresponding point on the upper and lower lines for the entire run of the program. For the program of Figure 1,this measure yields 258% of fragmentation. The problem with this measure of fragmentation is that it tends to hide the spikes in memory usage, and it is at these spikes where fragmentation is most likely to be a problem.*
> 2. *The amount of memory used by the allocator relative to the maximum amount of memory requested by the program at the point of maximum live memory. In Figure 1 this corresponds to the amount of memory at point 1 relative to the amount of memory at point 2. In this example, this measure yields 39.8% fragmentation. The problem with this measure of fragmentation is that the point of maximum live memory is usually not the most important point in the run of a program. The most important point is likely to be a point where the allocator must request more memory from the operating system.*
> 3. *The maximum amount of memory used by the allocator relative to the amount of memory requested by the program at the point of maximal memory usage. In Figure 1 this corresponds to the amount of memory at point 3 relative to the amount of memory at point 4. On this example, this measure yields 462% fragmentation. The problem with this measure of fragmentation is that it will tend to report high fragmentation for programs that use only slightly more memory than they request if the extra memory is used at a point where only a minimal amount of memory is live.*
> 4. *The maximum amount of memory used by the allocator relative to the maximum amount of live memory. These two points do not necessarily occur at the same point in the run of the program. In Figure 1 this cor- responds to the amount of memory at point 3 relative to the amount of memory at point 2. On this example, this measure yields 100% fragmentation. The problem with this measure of fragmentation is that it can yield a number that is too low if the point of maximal memory usage is a point with a small amount of live memory and is also the point where the amount of memory used becomes problematic.*

We obviously do not expect from you a study as thorough and deep as the one provided by Johnstone and Wilson. Yet, you should try to evaluate the performance of a few real programs

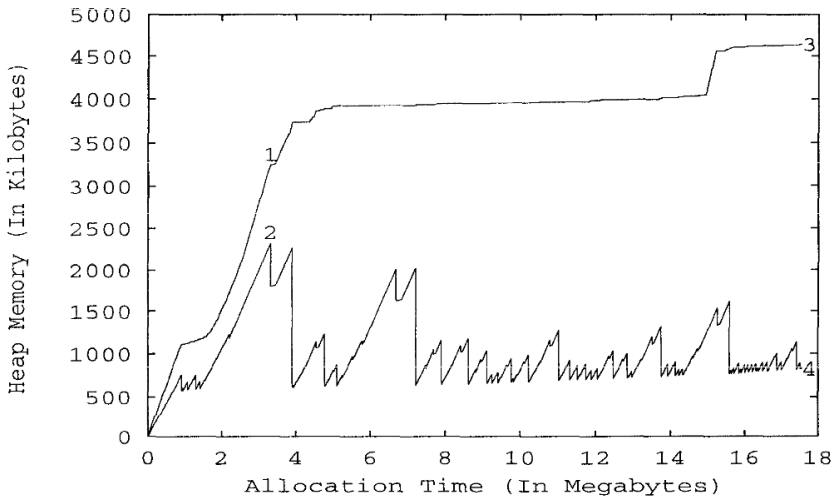


Figure 1: Measuring fragmentation

(say sort, head, ls, dmesg, ...) with one (or more) of the previous metrics or even one of your own invention. This should enable you to check whether "Best-Fit $\approx$ First Fit $\gg$ Worst fit" or not.

## References

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved. In *Proceedings of the First International Symposium on Memory Management, ACM*. Press, 1998. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.4610`.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. pages 1–116. Springer-Verlag, 1995. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.111.8237`.