TD3 : Formatted I/O Library

Master M1 MOSIG, Grenoble Universities

Lenka Kuníková   Lina Marsso

26/10/2014

1 Introduction

In this project we have to implement a memory allocator. We also explore different allocation strategies. And evaluate their performances in different applications.

Firstly in this report we give answers of questions in the subject. Then we present you our implementation. Then we speak about the safety checks implementation. Finnaly, we analyse the performance and conclude this project.

2 Subject questions

2.1 Question 1

We should keep track of the sizes of allocated blocks in order to release them. A linked list is not necessary because we dont need ordering. A better solution is to use a header at the beginning of each block, containing it's size. The exact position of the block can be determined using a list of free blocks.

2.2 Question 2

The user can use all addresses in our program if he has the pointer. But in real operating systems often it is not the case. Sometimes we can just access adresses divisible with 4.

2.3 Question 3

When we allocate a block we have to give the address of the block plus the size of the header containing the size of this block. For instance address + sizeof(int).

2.4 Question 4

The user should use the address that he gets after allocation. We need to decrease the address by the sizeof int to get the start of the block. When you free a block we need to check the position of the free block and decide if we have to do a merge, or create a new free block and add it to the list.

2.5 Question 5

At the beginning of each free block we need to store a free block structure and if the size of the free block is inferior to the size of this structure, we cant use it. In this case we give to the user the whole free block even if he wanted less.

3 Implementation

They are 3 folder :

- mem_alloc : The allocator with firstfit stategy, to perform the tests.
- mem_alloc_safe : The allocator with the safety checks integration.
- mem_alloc_frag : The allocator with tools for measuring the fragmentation.

1. We initialize the memory with the function memory_init
2. We alloc the new block with mem_alloc like that :
    - We find the address of new block allocated with one between the strategy 3 (First fit, Worse fit, best fit). We can choose. Basically, it's first fit.
    - We allocate the full block and update list of freeblock
3. We can free a block with free_block :
    - We find an adjacent block to the one to be freed and try to merge then
    - If no adjacent block : create a new block and insert it in the list

3.1 Analysis about test3

We can see with this test the first fit strategy favorises the fragmentation. In fact, in this example (test3) the smaller piece of memory (10 bytes) was allocated in a bigger free block. And this block became unusable. The next allocation asks for 20 bytes, this didn't fit into the first free block. We needed to skip it. And allocated another memory in the end. And we have a fragmentation. The best fit strategy would handle this case better.

3.2 Analysis about test4

We can see with this test that the best fit strategy is not for all cases a better strategy. Actually, best fit strategy puts the first allocated block in second free block of size 19. Then, 11 bytes become unusable. The second allocated block should be placed in the first free block.

Whereas using the first fit strategy, both allocated blocks can be placed in the first free block. And they fulfill the whole block. And we don't need to use another block.

4 Safety checks

We decided to implement some safety checks in our memory allocator. All the time when free() is called we check if the pointer is valid. We provide a function that can be called at the end and that check that memory was freed correctly. We also try to recognize corrupted data in our memory allocator, but it is not always possible.

4.1 Forgetting to free memory

We provide a function void checking_leaks() that can be run when the program is going to quit and witch checks if all allocated memory was successfully freed. It iterates through all available free blocks until it finds an existing full blocked that was not freed. We write the information in standard output about size of memory leak. Then the iterator continues examining other full blocks, if they exist, otherwise it would iterate through the left free blocks.

4.2 Calling free() incorrectly

Before the code of function free starts, we call a function int check() which will make sure the pointer it gets as an argument is really a pointer to start of a free block. It uses similar algorithm to previous function checking_leaks(). In case that pointers are not valid no memory is freed and a warning is displayed.

4.3 Corrupting the allocator metadata

We tried to implement basic control of data corruption. We are not able to detect every error that a user can cause, but we can avoid a large number of them. To do so, when we work with an iterator we always check if it really points to the piece of memory we are working with. When we use the size of a free/full block we check if size is greater then 0 but not bigger than the size of memory. For this purpose we use simple macro boundaries.
#define BOUNDARIES(a,b,c) ($a \leq b$ && $b \leq c$?1:0)

5 Performance evaluation

To measure the performance we have to chose betzeen 3 different stategies: first fit, best and worst fit. And we evaluate the fragmentation.

The figure 1 shows the amount of memory requested by the program gcc and the amount used by our allocator with the first_fit strategy. The same execution with a worst_fit policy is presented in the figure 2 and best_fit in the figure 3.

We can't see the difference between the different strategies with the program gcc because our allocator stoped before. For a small program, or the begining of allocation of a program the tree strategies have equal performances.

To compare the different allocation stategies we decided to measure the performance with an average at any memory operation of the ratio :

$$\left(\frac{Requested\_memory}{Sum\_of\_memory\_used}\right) \tag{1}$$

Figure 1: Memory requested and memory used with first fit strategy

Figure 2: Memory requested and memory used with worst fit strategy

Figure 3: Memory requested and memory used with best fit strategy

5.1 Table with fragmentation level of differents programs

program	worst_fit	best_fit	first_fit
ls	1.83	1.83	2.48
wc	0.96	0.96	0.96
ps	1.37	1.36	1.37
hostname	2.42	2.42	3.23

As seen in table 5.1, we always have fragmentation. The result shows the Best fit strategy is equal than the Worst fit strategy. And we can see again that First stategy is lower.

6 Conclusion

The goal of this project was to get familiar with the concept of memory managment and I think we succeeded. We managed to implement a simple version of memory allocator which allows users to allocate and free memory.

During the implementation we had to deal with several problems. For example we needed to find a way on how to determine positions of free blocks when we do not have linked lists of them, or think about the algorithm that helps us merge contiguous blocks of free space. Moreover, we implemented the three most common algorithms for finding an appropriate free block (first-fit, best-fit and worse-fit). We are now able to describe pros and cons for each of them.

In the second part, we added some other features to our memory allocator. All of them related to security checks. Now we know that implementing functions for free and alloc memory is not enough and we also need to deal with situations where users (on purpose or by chance) do not act the way we expect. The control we are making is not perfect but it helps us to deal with many cases that may happen.

Finnaly, we can see an application of our allocator. We can analyse it's performance with many apllications like ls, hostname... It was a very interesting project.