Operating System TD1

Armando Ochoa, Alessandro Fenicio

8 October 2014

Abstract

This document was written to show our progress in understanding how to deal with memory allocation. The following sections will present our teamwork and will explain our solutions to the several topics that were proposed.

1 Introduction

First of all we start showing the problem. Thanks to the C skeleton was given we have the possibility to implement a kind of fake memory that could be allocated using the function we implemented. Allocate memory means reserve some space inside that fake memory, to distinguish between this to concept of free and allocated space we have to main structures: the **free block** and the **busy block**. We will start presenting the different way to allocate memory than we switch to the questions to make the reader more comfortable with our solutions and finally we will discuss about implementation and evaluation of fragmentation

1.1 First Best and Worst fit

To explain this three approaches we will use that pictures: The above black block of size 16k is going to be allocated. Now the question is where it can fit? For sure it cannot be allocated in the block of 8k or 14k because there is not enought space. So we have the free block of 22k 16k and 48k are the possible candidates:

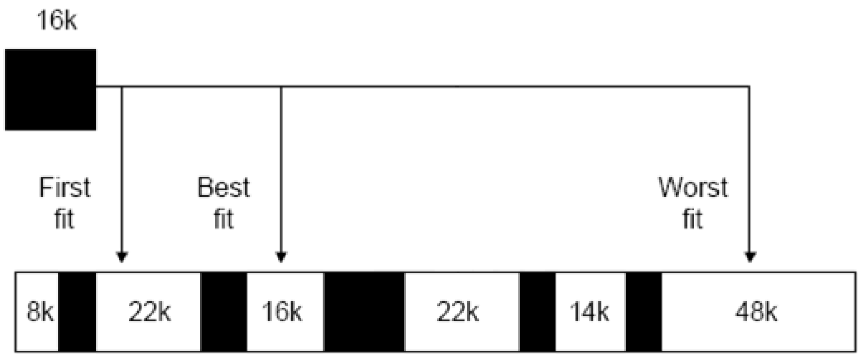


Figure 1: First best and worst fit

First fit
We choose the 22k block. This block represents simply the first candidate discovered.

Best fit
We choose the 16k block. This block represents the candidate whose size is closer to the one of we are going to allocate. In this case is exactly the same!

Worst fit
We choose the 48k block. This block is the candidate who has the bigger available size.

First fit

1.2 Questions

Do you have to keep a list for occupied blocks ? If no, explain why, if yes, provide a C definition of the structure. No, it's not necessary, Why?

We don't keep a list for occupied bloks. Initially we thought at how to handle the problem of exploring free and busy block, and we figured out that we would need at least the free block list. The busy block one is straight forward of the free block list: let's say we want to iterate the busy block list, if the first free is not the first memory address that means that is a busy block. So we can read the first busy header and sum its size to go to the next block. If that address is not in the freeblock list, that means that is a busy block...so we continue again exploring.

As a user, can you use addresses 0,1,2 ? Generally speaking, can a user manipulate any address?

No. When allocating memory for the user, only the payload's address is returned because he's not allowed to modify the free_block header, since that would corrupt the whole list of free blocks, and possibly lead to losing information.

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?

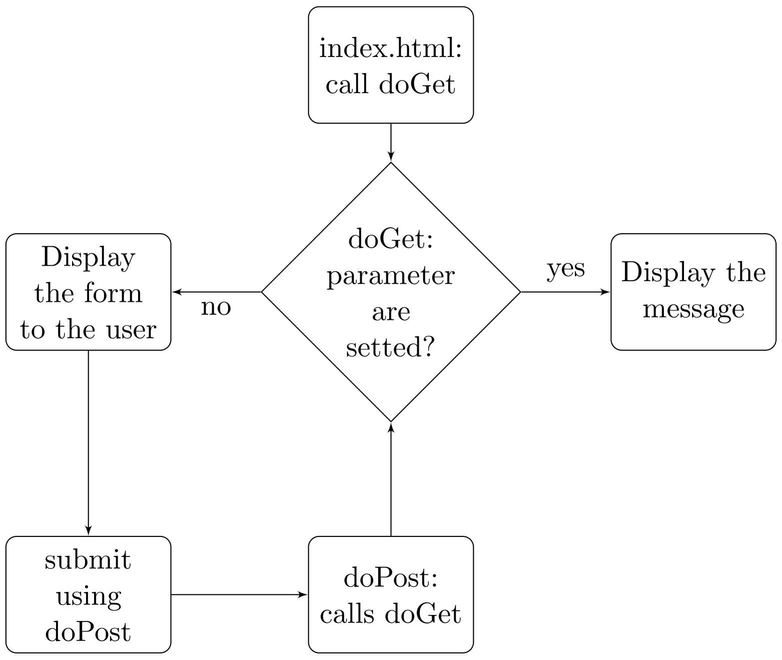
The address where the payload starts. There is no reason for the user to know the initial address of the busy block, so it's the memory free function's responsability to find it. In order to reintegrate the zone to the list of free blocks, we need to find the closest free block before the new free zone and make it the closest block's next. We also need to make sure that to set it's next as the next the closest free block had before. In short words, link the zone inside the free block's list in it's corresponding position.

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?

It happens when the amount of free space is not enough to hold at least the header of a free block. One possible solution is to allocate the remaining space as part of the block when it's to small.

2 Material and Methods

To implement a memory allocator means define how **initialize**, **allocate** and **free** the memory using one of the approach presented in the first chapter. The given C skeleton is capable to redirect the call to the *real* malloc (the one used by the operating system) to our implementation, and it has also a makefile that compare the executions of our program with some expected outputs.



3 Experiment

4 Conclusion