I. Databar Assignment Report

Metin Balaban
S131105
Danmarks Tekniske
Universitet
+4550299770
s131105@student.dtu.dk

ABSTRACT

In this report, I describe my design details of *embedded_malloc* and *embedded_free* methods. The implementation that I describe may not be too much efficient. However, it is easy to implement and enough for a small embedded system. I check my implementation in terms of consistency and give the details of test cases.

1. INTRODUCTION

Before giving the details of my work in this assignment, I want to give a brief description about the expectations of *embedded_malloc* and *embedded_free* routines. These functions are heavily used functions in an executable. In a ordinary executable, stack region and heap region are separate memory regions. However, in this assignment, I (have to) define a static memory array with fixed length. That means, heap is a part of data region. Particular to this case, the structure of executable in the memory is different than usual structure. Therefore reader should not expect any system calls such as brk() or sbrk(), which are used in common malloc implementations.

In section 2, I describe the organization scheme of the memory for heap-like allocation. In section 3, I define the algorithm that I use for allocation and subroutines of this algorithm. In section 4, I describe the deallocating strategy in detail. Lastly, in section 5, I give my testing strategy and demonstrations of all test cases that I added.

2. ORGANIZING THE MEMORY

The memory I allocate consists of continuous and successive chunks. For a moment we think an already allocated memory (see Figure 1).

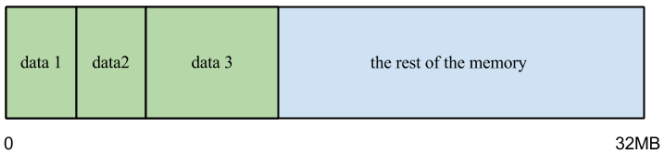

Figure 1. Visualisation of the memory of embedded hardware without meta-data

This is the most primitive way to using memory and pretty easy to implement. However, reallocation and freeing memory is impossible in this implementation. Therefore I need a more clever way to manage these data fields. Allocating more space than necessary and keeping information about properties of the data for each memory chunk is the simplest, pretty straightforward and effective solution (see Figure 2).

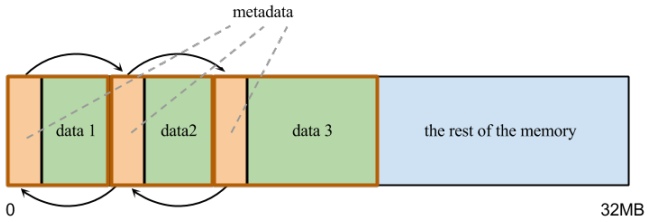

Figure 2. Visualisation of the memory of embedded hardware with meta-data

In this point we have to make a trade-off between the memory we spend on meta-data and implementation effectiveness. The more meta-data I get, the more pointers and availability I have.

Based on this concept I come up with a design such that every *embedded_malloc* call generates a chunk of data in memory. This chunk consists of a group of meta-data followed by a block of data. Reader may have a look at the C implementation of this meta-data block (see List 1).

```
typedef struct block * blockPtr;
struct block {
        int full;
        blockPtr prev;
        blockPtr next;
        size_t size;
};
```

List 1. block struct

This 16 byte struct consists of a flag representing whether the block is occupied or not, two pointers to previous and next blocks, and size information of data field in which the data is kept. Keeping a flag in a integer field seems like waste of space; However, the size of the block is to be multiple of four (since size of a pointer is four.) [1]. Using integer instead of char makes the struct already aligned. I made another trick to align the size of the data chunk. Size of the data section is also to be aligned (namely multiply of four). Using this formula (see Equation 1), I calculated the less number which is both greater than the size of the data region and multiple of four.

$$\text{Allocated size} = ((\text{Actual size} - 1) / 4) * 4 + 4$$

Equation 1.

3. MALLOC ALGORITHM: FIRST FIT

3.1 First Fit Algorithm

Since the simplest algorithm is first fit, I used it. As Tanenbaum stated in his book, Modern Operating Systems, the memory manager scans along the list of block until it finds a suitable place that is big enough for data plus its metadata. If the hole is bigger than necessary, then it is split into two pieces, one for the process and one for the unused memory, which is suitable for another *embedded_malloc* call [2]. I call this splitting process "splitting block". If it fails to find a suitable hole, memory manager creates a new block in the end of the double-linked list. I call this procedure "extending the list".

3.2 Splitting the List

First point that I want to cover is how empty block exist in the memory. Existence of these empty blocks among occupied memory is directly related to free function. Please consider a memory's state as represented in Figure 2. If free routine is called for the second data section in the figure, memory manager considers tagging the block "not full" instead of moving all full blocks back that are coming after the freed block. Moving operation is expensive, so by manipulating "full" tag, memory manager generates a double-linked list consisting of suitable and occupied blocks together.

If size of the empty block is big enough to be divided into two empty blocks one of which is large enough for allocation, I allocate the first chunk for request, and mark the second chunk (the remaining) not full. On the other hand, If the size of empty block is bigger than requested, and the difference of size of empty chunk and required memory is too small to create a new empty chunk from that, I simply allocate the whole empty space for the request.

3.3 Extending the List

The most important point to extend the double-linked list is to find the end of the list. I had to do it efficiently and I came up an clever idea. To understand it, one should know when the decision of extension of the list is made. The list is extended if memory manager searches through the list and could not find any suitable space for requested size. As one can notice, a search operation along the list is already done in that case. So I keep a global pointer that keeps an address of a block and I update it with the current block when I search along the list for a suitable place. When I fail to find (reach end of the list), then that global pointer keeps the last element of the list.

4. EMBEDDED FREE

As drawbacks of simple design of *embedded_malloc* and memory organization, free algorithm and its implementation are relatively complicated. Actually there is a straightforward algorithm of *embedded_free,* which is simply marking each block to-be-freed as "not-full". However, one can easily notice that it raises another problem called "fragmentation of memory". After heavily use of *embedded_free* and *embedded_malloc* routines, the fragmentations would be inevitable. So I come up with comprehensive design, which overcomes the fragmentation problem.

My methodology reveals that there are nine different state of deallocation of one memory block. The state is determined by status of previous block, and status of next block. Both previous and next blocks can be in three different statuses: Not Exist, Empty, Full. Therefore we have a 3x3 decision table for *free* routine.

blocks	NEXT BLOCK (notated as 'n' in the table)			
PREV-IOUS BLOCK (notated as 'p' in the table)	status	NOT EXISTS	EMPTY	FULL
	NOT EXISTS	base ←Null;	merge(this,n);	this->full ←0;
	EMPTY	p->next ←Null; free(p);	merge(this,n); merge(p,this);	merge(p, this);
	FULL	p->next ←Null;	merge(this,n);	this->full ←0;

Table 1. Decision table of *embedded_free* algorithm. *base* stands for the pointer keeping the first element of the double-linked list of memory chunks. *free()* represents recursive call of *embedded_free* routine.

While creating this table, I assume that it is impossible that a unoccupied memory block exists at the end of the list. Since the deallocating algorithm and "split the list" routine cannot create a free chunk at the end of the list, this assumption is always true. The main issue of implementing this algorithm is to obtain the previous block of the chunk to-be-freed. The most efficient solution is using a double-linked list instead of single-linked list. That is why I use double-linked list. The drawback of using double-linked list is obvious as I stated the trade-off in section 2.

Since some entries in the table are the same or almost the same, I made some groups of cases instead of dealing with each entry explicitly.

5. TESTING

embedded_malloc function is heavily used while testing *embedded_free* function. Therefore, in the test cases I have written for *embedded_free* function, consistency of *embedded_malloc* function is also checked. Any bugs resulted from allocating the memory can be detected.

5.1 Test Cases

I have prepared six extra test cases apart from the ones that are already prepared by instructor. Each of those six test cases focused on an 'if' or an 'else' branch in *embedded_free* function. Six test cases cover every possible branch in that routine.

5.1.1 Test_1

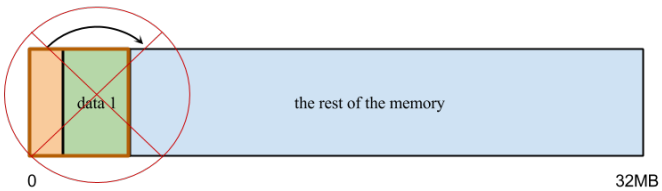

Figure 3. Test case 1. Each allocation is followed by deallocation. It is repeated one hundred thousand times for consistency check.

5.1.2 Test_2

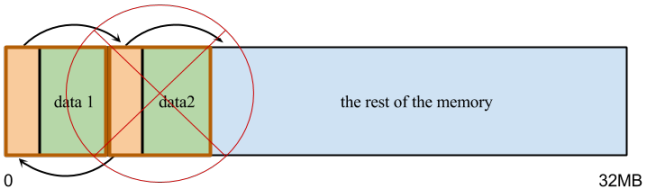

Figure 4. Test case 2. The second data block is allocated and deallocated repetitively. The contents of the first memory block is checked for defections.

5.1.3 Test_3

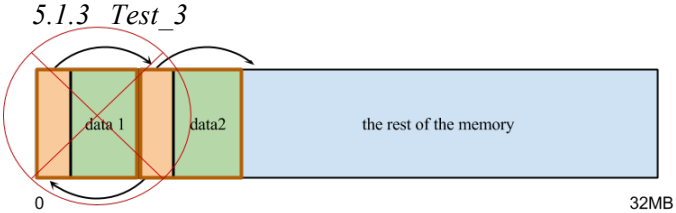

Figure 5. Test case 3. After two memory blocks allocated, then the first one freed. After that, a third one that is equal to the first one allocated. It is asserted that address of the second memory block is greater that third memory block.

5.1.4 Test_4

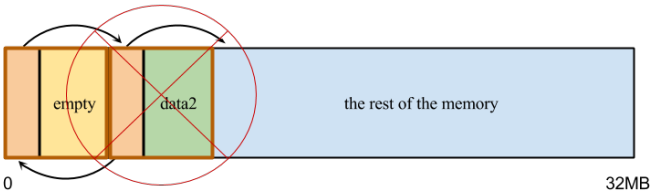

Figure 6. Test case 4. After two memory blocks allocated, then the first one and the second one freed respectively. After that, a third one that is equal to the first one allocated. It is asserted that address of the second memory block is greater than third memory block.

5.1.5 Test_5

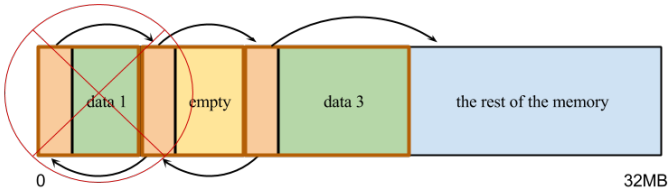

Figure 7. Test case 5. After three memory blocks allocated, then the second one freed. After that, the first one freed. Lastly, a fourth memory block whose size is bigger than sizes of both data 1 and data two, and smaller than the sum of sizes of data 1 and data 2 allocated. It is asserted that address of the third memory block is greater that fourth memory block.

5.1.6 Test_6

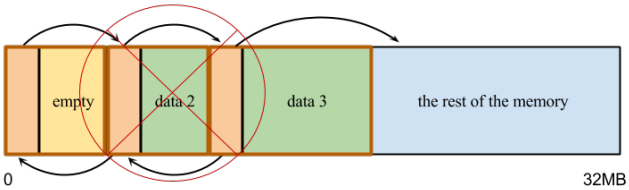

Figure 8. Test case 6. After three memory blocks allocated, then the first one freed. After that, the second one freed. Lastly, a fourth memory block whose size is bigger than sizes of both data 1 and data two, and smaller than the sum of sizes of data 1 and data 2 allocated. It is asserted that address of the third memory block is greater that fourth memory block.

6. ACKNOWLEDGMENTS

I want to thank to Husnu Sener, due to his inspiring statements, adored foresight, and encouraging personality.

7. REFERENCES

[1] Burelle, M. (2009). A malloc tutorial. Informally published manuscript, Laboratoire Systeme et Securite de l'EPITA (LSE), Graduate School of Computer Science, Paris, FRANCE. Retrieved from www.inf.udec.cl/~leo/Malloc_tutorial.pdf

[2] Tanenbaum, A. S. (2007). Modern operating systems . (3rd ed., p. 186). Upper Saddle River, NJ: Prentice Hall.