Memory Management & Buddy Memory Allocation

Louis Cameron Booth
louiscb@kth.se
9704181917

November 30, 2018

1 Introduction

In this report we will discuss the performance of the manual memory management for dynamic memory allocation of the C programming language. We will implement our own simplified versions of the *malloc* and *free* functions from the C standard library - *balloc* & *bfree*. Our functions will be utilising the buddy algorithm to manage the coalescing of free blocks in the memory. We will compare performance between both sets of functions using benchmarking testing.

2 Manual Memory Management

In this report we will be focusing on two functions from C's standard library.

2.1 Malloc

The *malloc* function stands for memory allocation and is used to request memory from the operating system. The *malloc* function works by reserving a block of memory of the user specified size in bytes. The function returns an address pointing to the space in memory reserved if successful, else it returns NULL. The *malloc* function dynamically reserves memory and is used to allocate memory during the execution of a program. The code below is an example of requesting 100 bytes of memory and returning the address to a variable.

```
int addr = malloc(100);
```

2.2 Bfree

Dynamically allocated memory created with *malloc* will remain in the operating system's memory until instructed to do so. This is where the *free* function is used. This function releases the specified block of memory back for use of the operating system. The *free* function accepts an address in memory as its only parameter and return NULL. The code below releases the specified address to the system.

```
free(addr);
```

3 The Buddy Algorithm

The buddy algorithm is a method of splitting and combining blocks of memory to return to a system memory request. When receiving a request, the algorithm will split a memory block into halves until it achieves a suitable size for the specified request. The algorithm will also coalesce free blocks of memory into larger blocks when it can in order to avoid a the memory being a large set of tiny blocks of memory. Each block is subdivided into smaller blocks with all blocks following a predefined range of block sizes. The level within this range of block sizes relates to the amount of bytes that the corresponding block takes up in memory as: 2^n where n is the level of the block. One of the buddy algorithm's benefits is that the total cost of the allocate and free operations are constant.

4 Our Implementation

You can view all of our code and run the program by downloading the code at: https://github.com/louiscb/Operating-Systems/tree/master/buddy

4.1 Balloc

Balloc is our version of the malloc function. We have followed the design of the standard C function as described in section 2 of this report. Our balloc function will accept a number of bytes as a parameter and will return an address in memory related to a block of memory of a size suitable to the request.

We keep track of block sizes by defining a maximum block size in bytes and creating a list of exponentially smaller block sizes to a defined minimum size. We call this list of block sizes *levels*, with the level of each byte size in the list equal to its index in the list. In our implementation we used the levels *32, 64, 128, 256, 512, 1024, 2048, 4096*, however one could have any list of base 2.

Our balloc function closely follows the design of the buddy algorithm in its implementation, with the core operation of the function being the splitting in half of larger blocks in order to obtain blocks of a suitable size. Our function recursively performs this operation of splitting memory until it reaches a block size that encompasses the requested number of bytes. When splitting

a block into two to obtain a single smaller block, an obvious side effect is that now you have a free unused block of the equal size. In order to not lose track of this block of memory and the resources spent on the splitting operation, we put a pointer to this free block of memory into a list. Each level with has its own list so we can keep track of block sizes for each level. Below is our code for this process (we have simplified the code to be easier to read):

```
struct head *find(int index) {
    if (index == LEVELS-1) {
        if (flists[LEVELS-1] != NULL) {
            return = flists[LEVELS-1];
        }
        return new();
    }

    if (flists[index] == NULL) {
        struct head *originalBlock = find(index+1);
        struct head *splitBlock = split(originalBlock);

        addToFlist(splitBlock);
        return buddy(splitBlock);
    } else {
        struct head *listBlock = flists[index];
        return listBlock;
    }
}
```

4.2 Bfree

Bfree is our version of the free function. We have followed the design of the standard C function as described in section 2 of this report. Our bfree function will accept an address as a parameter and returns NULL if it is successful. In order to free a block we do not have to deal with the data the address points to, we simply release that memory back into system by telling our program that this address space is now usable. We do this by adding the block to the list of free blocks for the next time the program requests a memory block of that size.

However we also need to implement the merge aspect of the buddy algorithm, where we coalesce two blocks of the same size that are 'buddies' in the memory (next to each other) to form one larger block. In our code we recursively merge blocks together until we reach the maximum level.

An issue with this merging method is that clearly we do not want to merge a block that is currently been allocated by the system - we only want to merge two buddies together if they are both free blocks of memory. One could find this out by looking at the free list we have created and checking if our buddy block is in the list and therefore a free block. However this is time consuming as in the worst case we have to traverse the entire list. As a result we have implemented a status that sits in the head of each block that tells us if the block is either TAKEN or FREE. Below is our code for this process (we have simplified the code to be easier to read):

```
void insert(struct head *block) {
    if (block->level == LEVELS-1) {
        flists[LEVELS-1] = block;
        return;
    }

    struct head *buddyBlock = buddy(block);

    if (buddyBlock->status == Taken) {
        addToFlist(block);
    } else {
        block->level++;
        removeFromFlist(buddyBlock);
        insert(block);
    }

    return;
}
```

5 Analysis

5.1 Malloc vs Balloc

Malloc outperforms our balloc program. In our benchmark testing we ran a series of calls to each program one after the other and timed how long each program ran for. We varied the amounts of memory we were allocating randomly in order to mimic a real world system environment. We discovered that malloc took a consistent amount of time no matter what size of bytes were requested, whilst our balloc code would fluctuate and generally increased in time the larger the block size. Please look at the appendix 7.1 for the results.

5.2 Free vs Bfree

As expected, free also outperforms our bfree program. In our benchmark testing we ran a series of calls to each program one after the other and timed how long each program ran for. We varied the amounts of memory we were allocating randomly in order to mimic a real world system environment. We requested a chunk of memory and then at random points within our benchmark program would request that each address be freed. Similarily to Malloc vs Balloc, the C function free consistently produced the same results whilst our program increased with size. See appendix 7.2 for the results.

We believe that the main area of improvement for our program is the traversal of the free list. Once the program has been running for a while, the free list will be quite long filled with blocks and as a result the time taken to go through that list will exponentially increase the time taken for our program.

6 Conclusions

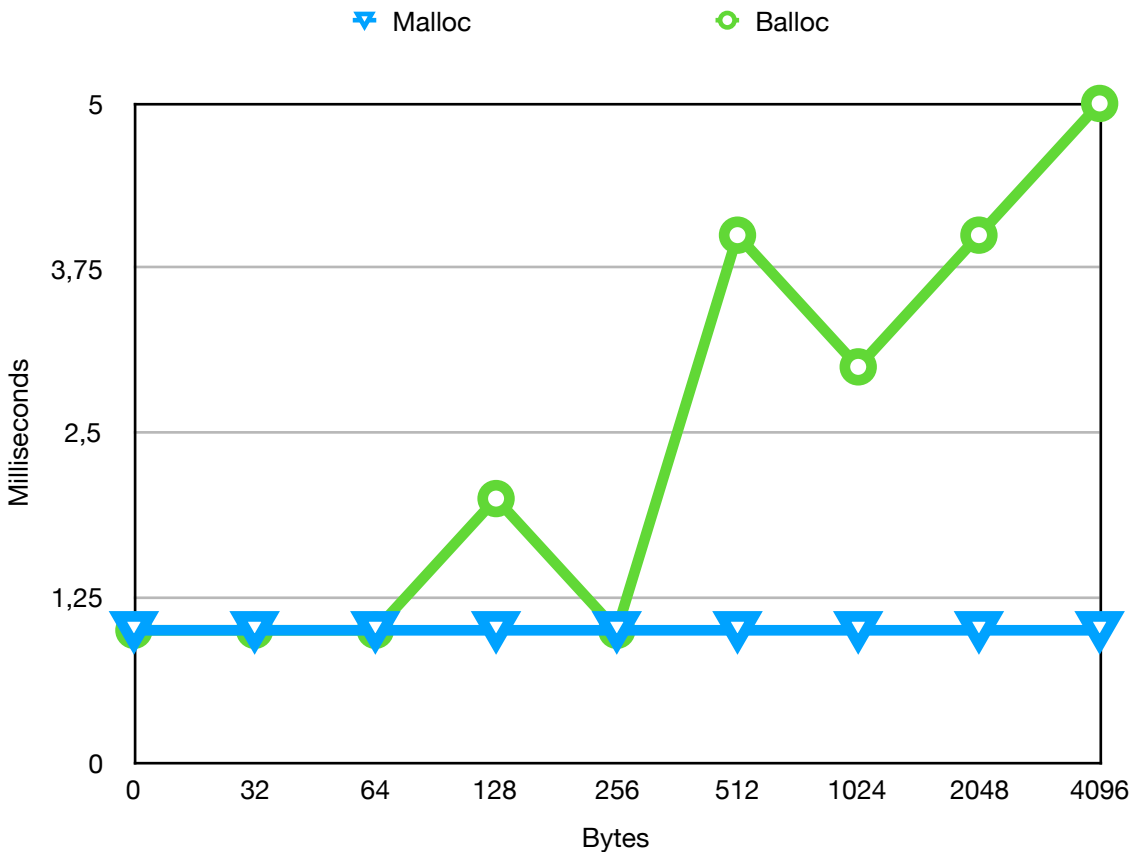
The standard C library functions have been optimised and worked on over many years in order to provide programs that are reliable and can work across many styles of hardware. Looking at the source code for malloc we can see that it is over 5000 lines of code compare to a couple dozen for our balloc.

As a result the malloc & free program perform extremely efficiently and from our benchmarking was hardly bothered by any size of memory we asked

it to reserve. These functions form part of the underlying core of the operating system and if their overhead were not minuscule it would impact all operations on a system.

7. Appendix

7.1 Malloc vs Balloc



7.2 Free vs Bfree