Buddy Algorithm - ID1206

Youssef Taoudi, yousseft@kth.se

November 29, 2018

Abstract

For this seminar I implemented the Buddy Algorithm to use for memory allocation, benchmarked it and compared it to the built-in malloc allocation method for linux in terms of performance.

1 Introduction

The buddy algorithm is useful in memory allocation for avoiding external fragmentation since it does not leave any holes between memory blocks. The way this is done is by partitioning memory blocks to satisfy memory requests as suitably as possible. Blocks are partitioned by dividing blocks into two until it fits the requested size. This will of course give some internal fragmentation as the blocks will not fit perfectly to the users needs. The first task for this assignment was to implement balloc and bfree, the operations for allocating and freeing memory.

The second task was to benchmark the algorithm. We checked how much external fragmentation there was, how fast balloc and bfree ran and compared it to malloc. We also made a small optimization where blocks are given back to the OS when freed to decrease external fragmentation even more. This is of course also benchmarked and tested against the non-optimized balloc.

2 Implementation

Again, The two functions were implemented to allocate and free memory to and from the user: balloc and bfree.

Before diving into the implementation for the two functions, it is important to know that we keep track of free blocks in different freelists. We have eight freelists in total, one for each level of size. Our biggest size level for example, level 7 holds all free blocks of size 4096-2049. It is important to hold all free blocks in a structure where we can reach them when freeing as you will see in the implementation. There is also a head structure that comes before each block in memory. The head structure holds important information for a block including an integer for level, a an enum for status flag (Free or Taken), a pointer to next element in list and a pointer to previous element in list. The pointer value for the head itself is the address for the head.

2.1 balloc

The balloc functions takes one parameter, size and returns a pointer to a memory address. The size parameter is the size that the user needs and the program will check if there are any blocks of that size or of higher size in the freelists to give out. The return value will be the address of the block that the user can write memory into. The program checks which level the block should be according to the size input and we start looking through the freelists. The freelist for the given level is checked first. Only the first element of the list is checked since any block in the freelist would do. If the first element is NULL (i.e the list is empty) we will check through the freelist of the next level. If an element is found in this level, we will split it in half and give one half to the user, the other block will be put in the start of a freelist. If the freelist was empty, the program will check the next freelist and repeat the process recursively. This can go on until it hits level seven, if the freelist for level seven is empty, a new block will be fetched from the OS and will be split down (or given away depending on size asked for). Before any blocks are sent to a freelist or user, their level and status flag will be updated for future uses. One thing to note is that when inserting a free block into a freelist and the list is not empty, one must change the first elements previous pointer to point to the new free block and the new free block will become the new first block of the freelist. It is very important that the previous, next and freelist pointers are set correclty otherwise the freelist may be delinked or lost. The memory we give out will not include the size of the header.

2.2 bfree

There is not return value for the bfree variable but it takes a memory pointer as parameter. The memory that pointer points to will of course be a block that we have previously handed out. With some magic we can find the header of the memory block (we jump back 24 bytes to land on the header address). The program will now look to insert the block back into a freelist. The easy way to do this would be to add it into the freelist for the size level of the block but since our program strives to store as big blocks as possible to give out to future processes, we will check if we can merge the newly freed block with another block in memory. A block can only be freed with what is called a 'buddy'. A buddy is the block of memory right next to the newly freed block in the same level. It is found by toggling one bit on an index depending on level to be the opposite of the freeblock. If the buddy of the newly freed block is also free, they will be merged into a larger block on the next level. The buddy will have to be unlinked from it's freelist. We repeat the process recursively, so the newly merged block will also look for it's buddy and merge if possible. If the buddy is taken, the block will be inserted into the freelist for the level of the block. If a block is in level seven, it will be inserted into the freelist directly instead of looking for it's buddy since we are at max level. This will be done a little bit differently in our optimization where we will send all level seven blocks that are merged or freed back to the OS.

3 Data

3.1 Benchmarking

The data for the memory allocation and cost was measured through a benchmark program calling balloc and bfree for randomly generated sizes where sizes of each level were approximately as likely to appear. When memory was allocated, it was stored in a buffer in a random index. If the index in the buffer was already taken, bfree was called and the new memory address was inserted into the buffer. This means that bfree was called more the more elements had been stored in the buffer, in other words, the longer the program had ran, the more likely it would be for bfree to be called.

In the benchmark we test how big the external fragmentation is for regular balloc and how much it is optimized when sending back blocks to the OS. We also test how expensive the optimization is in terms of time for a balloc operation. The benchmark also tests how our balloc compares to malloc in terms of time performance depending on size of memory. There is also a graph showing how well our balloc would do compared to malloc in a simulation of a real application.

The cost of running 100.000 ballocs is what is diplayed in the graph, not the cost of calling a single balloc.

3.2 Graphs

The measured graphs are shown in the next page because LaTeX is a pain to work with.

Figure 1: Plot for memory allocated in total by the OS and to the user for standard balloc

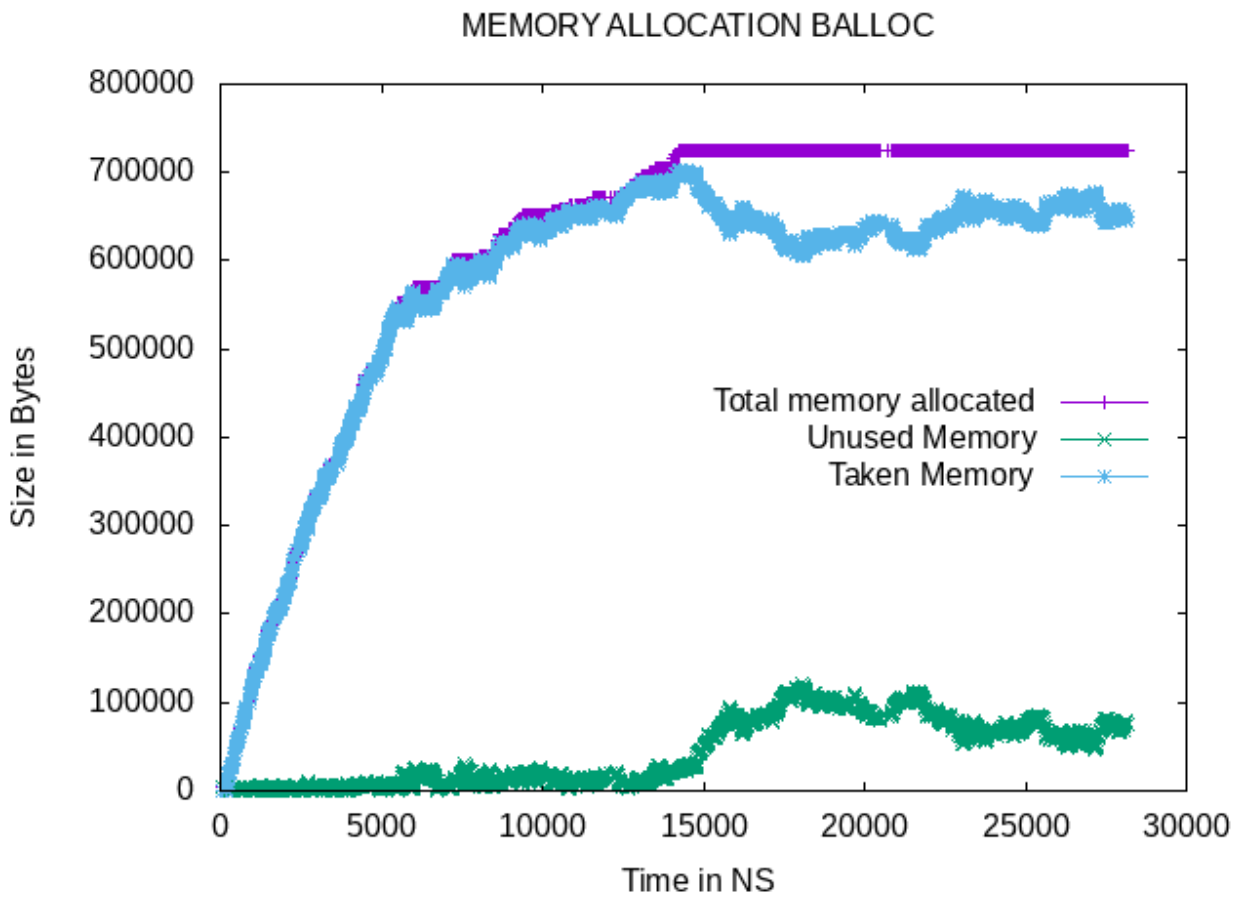


Figure 2: Plot for memory allocated in total by the OS and to the user for optimized balloc

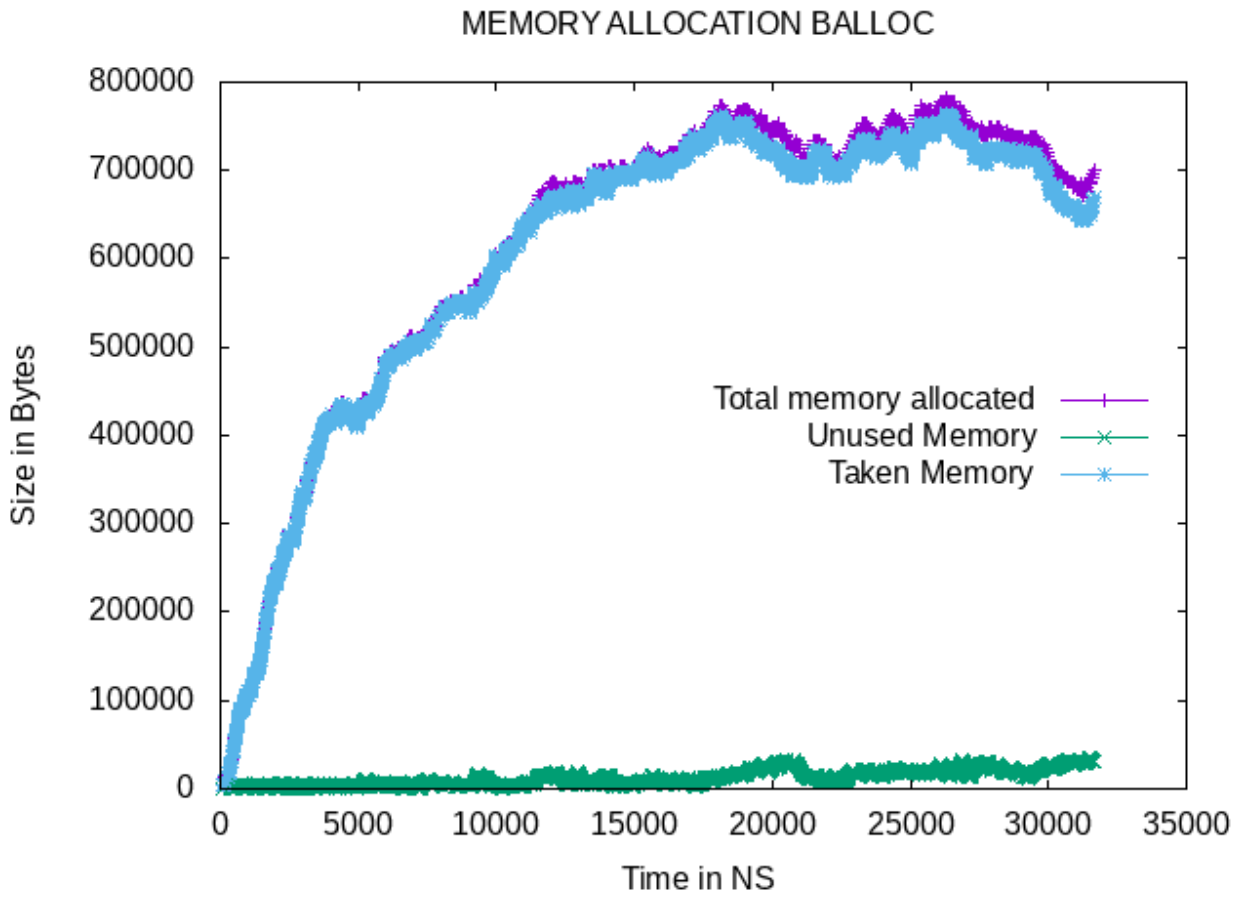


Figure 3: Cost of balloc operations in real life simulation (X-Axis is amount of times balloc has been called

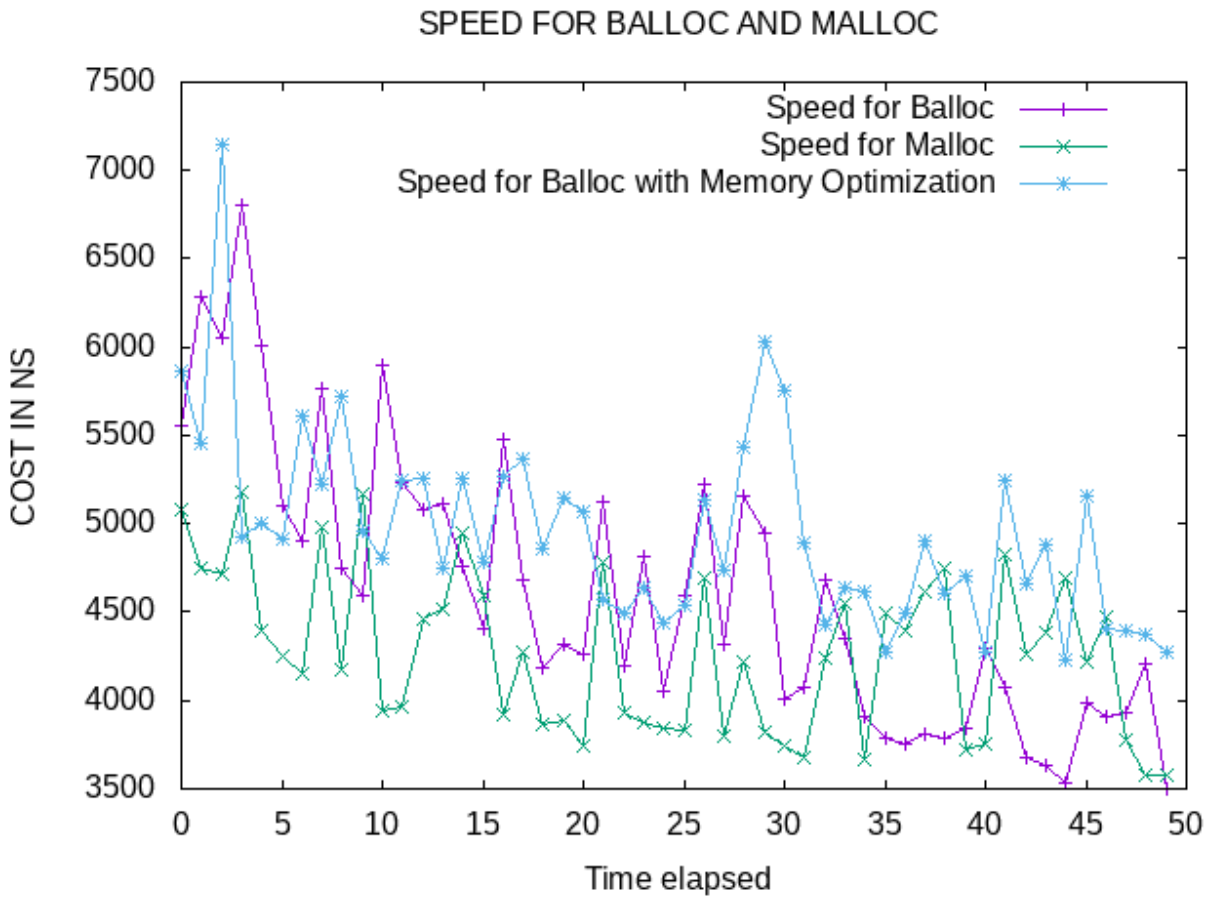
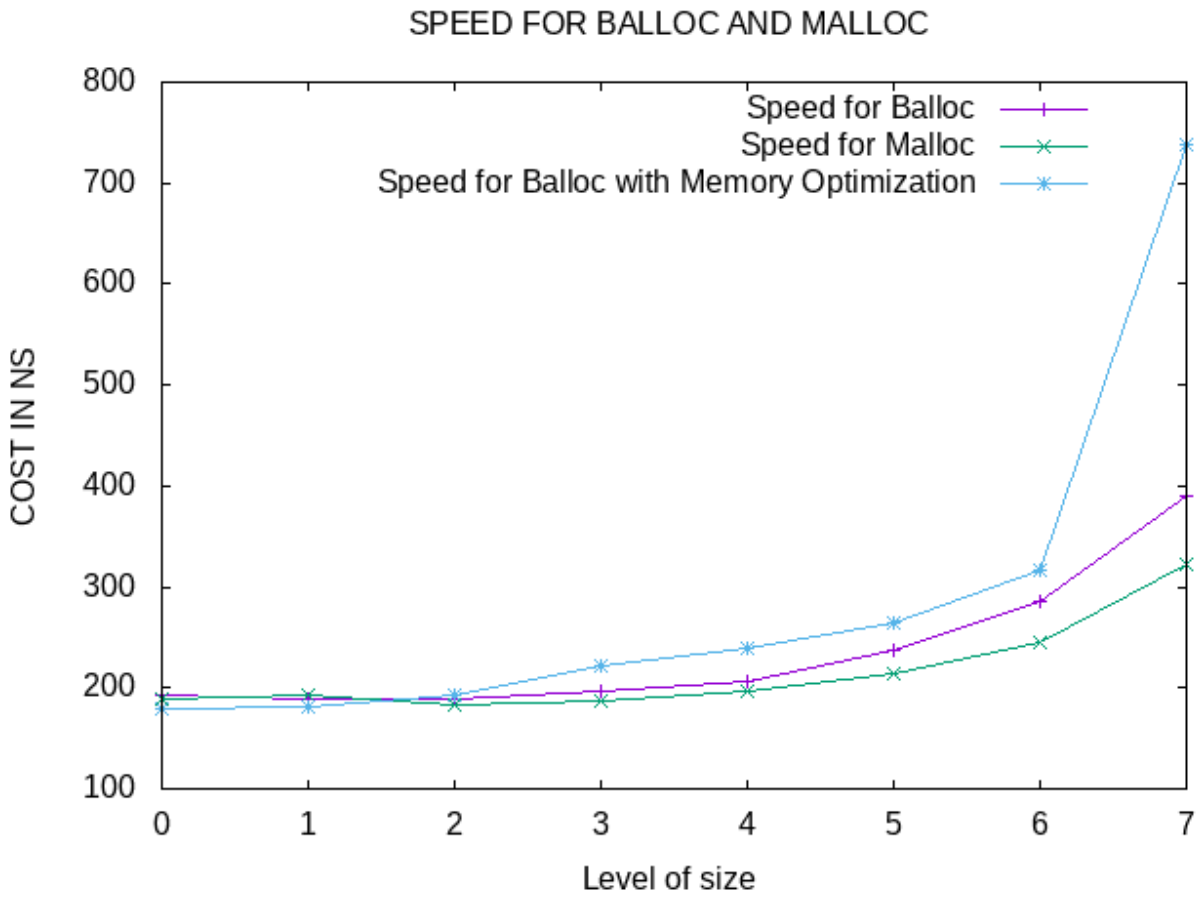


Figure 4: Cost of 100.000 balloc operations depending on size of block allocated



4 Analysis

4.1 Optimization

Figure 1 shows us how small the external fragmentation is. The purple line is the total memory allocated to the program by the OS, the blue line is how much memory we have given to the user and the green line is the free memory in our free lists. The green line therefore shows us our external fragmentation. The graph shows us that the external fragmentation gets higher the longer the program has been running, this is because we call the bfree method more often at the end of the running time. As expected, our external fragmentation is fairly low but we will test if it is possible to make it even lower.

To try to reduce external reduction, we introduced our memory optimization where we send level seven blocks back to the OS whenever possible. As seen in figure 2, this has massively improved external fragmentation. The amount of free memory is close to 0, even when the buffer is full. While this looks amazing, it might not look as good when we test the cost of the balloc operations.

Figure 3 displays how well our optimization runs a balloc function compared to balloc and malloc in a real program simulation. The blue line is the memory optimized balloc, purple line is regular balloc and green is malloc. The cost in the graph is the time it takes to run a very large amount of allocations (I have forgotten how many) Here the problems of our optimization start showing themselves, it generally takes longer to complete a balloc with our optimization than without it. This is further shown to be true in figure 4 where the run time of 100.000 ballocs are measured for each block size level. Not only does the memory optimized balloc take longer time in general than the other allocations, it takes significantly longer time for level seven. This is of course because we have to ask for blocks from the OS more often than regular balloc because we generally do not have as much free memory to use from. Since the call to the OS is fairly costly, our memory optimized balloc will be slow.

4.2 Balloc vs Malloc

In linux, memory is allocated using the malloc function. How does our balloc compare to the build in malloc in linux?

Well, looking at figure 3 where the cost of a large amount of ballocs and mallocs are called in multiple rounds, the built in malloc generally performs better than our balloc especially in the beginning. It is of course expected that the linux built in method would be better than the implemented balloc. The reason that our balloc performs poorly in the first rounds is because we have no free blocks to work with and thus need to call the OS for more, which is expensive.

In figure 4, both malloc and balloc get more costly as we request higher sized blocks. Once again however, our malloc performs better. The reason our balloc performs worse with higher size blocks is because it is less likely that we have a block available the bigger sized block we need to use. When a free block of wanted size is unavailable, the OS has to be contacted for more and that costly as stated time and time before in this report.

5 Conclusion

The balloc implementation was not too bad in performance and had fairly little external fragmentation. Calling the OS (mmap in linux) for more blocks when there was not enough memory free drove the cost of balloc up.

Our memory optimized balloc had next to no external fragmentation but lacked significantly in performance since it never had any free memory to use and had to rely on the OS all the time. This optimization could be used in systems where memory is limited and performance is not very important, or in other words, never.

The built in memory allocation in linux was superior to balloc as expected.

6 Discussion

The implementation for the buddy algorithm was not very difficult, especially when more than half the implementation was given before hand. Debugging in C however is something I would not wish upon my worst enemy. Benchmarking and plotting using gnuplot was also quite tricky, most plots I tried producing did not look very good, including figure 3 in this report.