ITP 30002-02 Operating System, 2023 Spring

Homework 3

Jooyoung Jang, 21800637 21800637@handong.ac.kr | Seongjoon Cho 21900699 21900699@handong.ac.kr

1. Introduction

Virtualization is the key feature that promotes the use of Operating Systems (OS). Among them, the virtualization of memory is one of the most important techniques that we need to focus on. The goal of virtualization in memory is to divide one chunk of physical memory into multiple blocks so it can be allocated to the process upon which it is required. Our goal for this homework is to allocate virtual memory based on the buddy memory allocation algorithm (**BMA**). Many studies were conducted to find the optimal allocation strategy, and BMA was one of them. The basic idea of this algorithm is to split one memory block into two equal-sized blocks until it finds its matching block for the proper allocation. It has its advantages of causing little external fragmentation, however, it has its disadvantages of causing internal fragmentation issues. Hence, it is a useful technique that can be used in certain circumstances.

In this homework, to implement the BMA. We had to allocate memory using system call mmap() and its various flags. This function is originally used to map a portion of its virtual address space to a file or device. However, for this homework, we used the MAP_ANONYMOUS flag to make an anonymous page so we can create virtual memory spaces that are not related to heap memory space. Once the initial memory block is allocated, we had to check the requested size and if our block can suit the requested size or not. If it was too big for a memory to be allocated it would cause a huge internal fragment issue. Thus, we had to divide the block into two equal-sized blocks. With these newly created blocks, we created a linked list to manage them. It was also our job to arrange or merge these blocks in case of freeing the memory by reallocating the size of the memory. Throughout this paper, we will discuss how we tackled each requirement that is listed above. We will demonstrate our ideas and the failures that soon evolved into a great success.

2. Approach

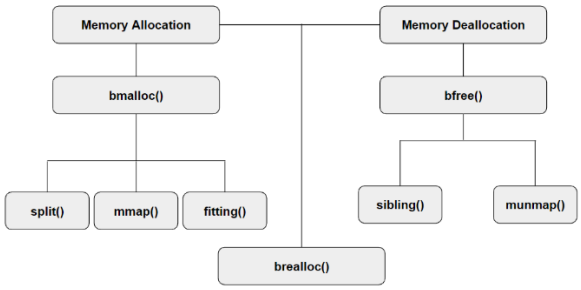

Figure 2.1 Approach diagram

The first thing we need to keep in mind is the memory structure of our buddy allocator. As it is requested it consists of two parts: the header and the payload. The header is composed of three things which are the used bit, the size bit, and the pointer which directs to the next block. It is a 9-byte header, and it holds the metadata of the block. Thus, the structure of our block looks like **figure 2.2** The key point of memory allocation is to use a pointer and point to the right address to keep them in a linked list. From now on we will discuss deeply our approach towards dealing with all the requirements in three easy pieces

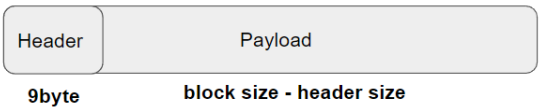

Figure 2.2 Block Structure

2.1 Memory Allocation: bmalloc()

In order to achieve BMA style we need to solve the bmalloc() function which is the allocator function. To implement this function, we used some helper functions. As you can see in **Figure 2.1** three functions are essential. First, as soon as bmalloc() is called it will check if the requested size is suitable for our allocator to handle. This was done through the fitting () function. Since our maximum block size is 4096, we could fit up to 4090. This was because we had to consider the header size upon allocation. After checking the block size suitability, it will return the size field value of a fitting block.

Then it will perform different procedures following the option deciding whether it will be the best fit or the first fit. The best-fit algorithm will compute the best matching block for the requested size while the first fit will find the first matching block. Once the best-fitting block has been selected it is time to check if the list of blocks is empty or not. If it is empty, it will initialize the list by allocating 4096-sized blocks with mmap and MAP_ANOYNOUS flag. Performing this sequence will make a virtual memory space. Then we will set the metadata for the block. To match the structure that is demonstrated in **Figure 2.2** we linked the block and the header to indicate the metadata. If the block already exists that means, we must split the blocks to find the fitting block for allocation.

This is when the split () function takes its place. What the split function does is, it split the block until it finds the

matching block with the required size. This will generate the buddy block of our current block which later merges when the memory is freed. The split () will return the address of the block payload's starting point. Coming back to the bmalloc, if there is no unused fitting block among the linked lists, it will then allocate a new 4096-sized block to repeat the process listed above.

2.2 Memory Deallocation: bfree()

Freeing memory was a whole different story from memory allocation. Splitting and allocating was easy however, freeing them and merging the memory was rather more like dealing with data structure. The first task we had done was to receive the block's address and adjust the pointer to the starting point of the block. Then using this address and block structure we turned off the used bit to zero and cleared out the block by setting the memory to zero.

Now the freeing requested memory was done it was time to merge the current block with its sibling block. We ran a while loop to check its size and made sure it does not exceed the maximum. By using the sibling function, we were able to obtain the sibling block. Our sibling function made a mathematical approach. It adds up all the block sizes in the list and divided them with the initial block size. Then performed a modulo operation to see if it was located on the left or right. This gave made us index the blocks and made it much easier to obtain the correct sibling block's address. Then to merge the current block and its sibling block we first removed the sibling block from the linked list and adjusted the current block's size so it can merge into a bigger block. We repeated this process until we found a used sibling block or reached the maximum-sized block. Once the block has merged to one single block, we checked whether it was being used or not. If it was in its unused state we used munmap() function which unmapps the block that was created by mmap() function.

2.3 The others

brealloc(): The brealloc() function's task was to reallocate memory spaces' size. This implied that we had to figure out how to rearrange the size and the address of the block. The steps that we took are simple. First, we received the address of the current block and its reallocating size. We created a header pointer to deal with this block and if the current block can fit into the new size. We freed the current block and reallocated the block using the bmalloc() function and copied the contents of the previous block to the newly created block.

bmprint(): Apart from the template we were provided with we had to include additional information while printing out the information about the current status. We traversed through the linked list to see if the block was surd or not. If it was being used, we added the size of the allocated block to the user_mem variable. This variable keeps the number of allocated blocks for the users. If it was not used, we added them to the avail_mem variable to show the amount of free memory upon every allocated block out there. To show the payload size of each block while traversing the linked list we simply subtracted the header size from the block size.

3. Evaluation

To evaluate this homework, there are 3 test cases that were provided beforehand, test1.c, test2.c and test3. c. All the requirements that were required to be provided by the bmalloc library have accomplished all tasks successfully in test1. The first allocation starts off with bmalloc(2000). Since the requested size is 2000, it is smaller than 2048 even if the header of the memory space is considered, thus the result will be like the follwing:

```
bmalloc(2000):0x7f4333dcb010
==================== bm_list ====================
 0:0x7f4333dcb010:1        11      2032:00 00 00 00 00 00 00 00
 1:0x7f4333dcb810:0        11      2032:00 00 00 00 00 00 00 00
=================================================
=================================================
==================== stats ====================
total given memory:             4096
total given memory to user:     2048
total available memory:         2048
=================================================
```

2048 memory is given to the user to allocate a block with the size of 2000, and the rest will be marked as available memory for future allocations. Similarly, the second memory allocation is called right afterward, but this time a block with the size of 2500 is requested, which obviously cannot fit into the already free space of 2048, so a new block of 4096 is created for it. Because 4096 is the best fitting block that can fit a 2500-sized memory, the remaining total available memory space will remain as 2048, even if a lot of internal fragmentation might happen. After this, test1.c calls the bfree function to free the memory space allocated to the block with the size of 2000. The bmalloc library proceeds to free the memory space as requested joins the now free 2048 block and the already free 2048 block to create a new 4096 block. However, since this 4096-sized block is now just an empty block, it gets deallocated completely, leaving us with memory space allocated with just one 4096 blocks. Test1.c continues to allocate new blocks 2 more times, and the bmalloc library does the job without any problems.

Exceptional cases can always happen, to test this we have made an additional test case branching out from the test1.c. We made situations where the user allocates memory exceeding its capacity (eg. bmalloc(4081)). Plus, the situations where user frees the unallocated memory address. Both cases can lead the segmentation fault issue. Thus, we made a safeguard for this type of issue my printing out the error message and stop the code from executing. The user can test this by running the test case(test4_M.c) that was provided with the rest of the library.

While test1.c was about using available library functions only, test2.c and test3.c were about applying the buddy memory allocation strategy to more complicated data structures like trees and linked lists.
In the case of a tree data structure, the user has to specify what the data in each node has to be. Because only the memory for each node is allocated, each node does not take up a lot of memory in the BMA. No matter what node the user puts in, bmalloc is called in order to allocate a single node each time a new node is inserted. This will make test2 results with several small allocations, where each small memory block will be used to allocate one node in the tree. The test3.c however, was intended to raise seg fault. If you review the code carefully to achieve its proper execution. We must rearrange code in a way that does not follow for test 1 and 2. Hence no matter what we do we could not get rid of the seg fault issue.

4. Discussion

In the development of bmalloc functions, the focus was on efficient memory management using basic data structures. It was surprising how the concepts of binary trees and ked lists were employed in this algorithm. When splitting blocks, we used principles similar to splitting nodes in a binary tree. This allowed us to divide memory blocks efficiently while maintaining the hierarchical structure of the memory system. Such as the red-black tree that we learned in our class time.

The BMA algorithm provided insights into the trade-offs between internal and external memory fragmentation. The algorithm minimizes internal fragmentation by recursively splitting memory blocks in half until a block is created that can fit the requested memory while being the smallest possible size. External fragmentation is addressed by merging unused blocks of the same size. Freeing memory blocks requires careful handling of data structures and the implementation of merging algorithms to consolidate adjacent unused blocks.

Despite efforts to minimize internal and external fragmentation, there were cases where memory allocation was inefficient. For example, if a user requested a block with a size of 2049 (including the header), the algorithm would allocate a whole 4096-sized block, wasting nearly half of the given block. In poorly managed external fragmentation cases, where there is one used block of 1024, we may end up with two free blocks: the buddy block of the used one and another with a size of 2048. However, if a user then requested a new block with a size of 2500, even though there is a total of 3072 available memory, the algorithm would create a new page of 4096 since the split blocks cannot accommodate a 2500-sized memory block.

During the development of the bmalloc library, we were able to compare this memory allocation strategy with other strategies, such as slab allocation. Slab allocation divides memory pages into fixed size "slabs" and aims to minimize external fragmentation by allocating blocks from pre-allocated slabs, ensuring there are no unused spaces between blocks. However, this approach may result in a higher degree of internal fragmentation. In contrast, bmalloc splits memory blocks into sizes that best fit the requested size, rather than using fixed-sized blocks. Although some internal fragmentation may occur, the strategy minimizes it by recursively splitting larger blocks into smaller ones.

Every strategy has its pros and cons. In the case of buddy allocation, the advantages include minimizing internal fragmentation by creating the best-fitting blocks for requested sizes. However, there are some drawbacks, such as the time efficiency required for merging blocks when freeing memory and the issue of external fragmentation due to the creation of numerous small blocks during splitting.

5. Conclusion

The goal of a buddy memory allocation algorithm is to divide one chunk of physical memory into multiple blocks to be allocated to processes as needed. We have demonstrated the steps to implement BMA throughout this paper.

The evaluation of this homework was done using three test cases: test1.c, test2.c and test3. c. Although each test handled different block sizes and with the usage of different data structures like linked lists and binary trees, the bmalloc library provided efficient memory management by splitting blocks and merging unused blocks at the same time.

The trade-offs between internal and external fragmentation in the buddy memory allocation algorithm were discussed. By splitting bigger chunks of memory into a best-fitting block for a requested size, this approach might be a good way to minimize internal fragmentation. However, because the algorithm creates best-fitting blocks by splitting bigger blocks, a lot of residual smaller blocks are left over, which might lead to external fragmentation.

This homework successfully implemented the buddy memory allocator and provided insights into memory management and fragmentation issues. The bmalloc library provided functions that accomplished all the required tasks and demonstrated the effectiveness of the buddy memory allocation algorithm in certain scenarios.