ITP 30002-02 Operating System, 2023 Spring

Homework 3

Shinhoo Kim and 21900136, 21900136@handong.ac.kr
Junghyun, An and 21900423, daniel7806@handong.ac.kr

1. Introduction

This problem is implementing malloc library with buddy algorithm to maintain the space. In order to implement it, there were the two parts we considered. The first one was how to request an extended free space in the heap memory from kernel. The second one was how to implement buddy algorithm and manage requested memory space using a proper data structure. We acheived the first consideration by using mmap() function without mapping any file in a storage so that we can just retrieve a desired size of empy page. However, in the user perspective, they do not know the application gets the page size of space from kernel but only the exact size they request. And in order to achieve the second consideration, we implemented an linked list data structure with a custom structure of a header which works as a metadata of a block(node of linked list) for buddy algorithm.

In order to retrieve an extended heap memory space from kernel, we used mmap() with MAP_ANONYMOUS flag to not map any file in the storage and MAP_PRIVATE flag to prevent the other processes from accessing the page. And in order to implement a buddy algorithm for malloc API, the three main functional requirements were required. The first one was that it could find a fitting block in the either best or first pick manner. The second one was that it could split and merge a block when they are needed. For splitting function, just like a binary tree, if a half of one block's size can accommodate the requested memory size, then the block can be splitted and there are 9 different block size options(2^4 to 2^12 bytes). So 16 bytes size is the minimum size of block which is atomic. For merging function, it could also find the one's sibling block to merge together. The last one was having a good data structure that can accommodate the above requirement in an efficient way. To use linked list, each block needed its description such as a metadata to store one block's size, a use bit, and an address of next adjacent node.

2. Approach

When a user calls bmalloc(size_t size) to allocate memory, in the bmalloc() logic, it calls fitting() helper function to find an address of block that accommodates the requested size. After it receive the address of header of the block from the fitting() helper function, the used bit is updated as 1 and then it returns the starting address of payload of the block but not the address of the header. In the fitting() helper function, we changed logic to achieve a simplicity in the client side (where it calls). The fitting() function takes the requested size and it travels the linked list to find the right block with the either best or first pick manner and store the found block address and its block size in the local variables. If it is best pick, it travels to the end of the linked list and keep updating the local variable to find the tightest(best) block but if it is the first pick manner, then as soon as it finds a possible block, then it breaks the loop. And the next step is to check if the target block that will serve the request is found or not by checking whether the local variable is null or not. If the target block is not found, then it means it needs another page from kernel. So it request another page. On the other hand, if the target block is found, then splitting logic is waiting for it. By continuously calling splitting() helper function until the half of the block size is bigger than the requested size. The reason why that the size of the block needs to be always bigger but not equals to the requested size is because the block's payload is actually smaller than the block size because of the size of the header. Once the splitting step is done, then it returns the address of the header of the block.

When a user calls bfree(void*p) to deallocate memory, in the bfree() logic, it receives the payload address and calculates header address by subtracting the size of the header to the given address value. If the header address is valid and the block is in not used, then the bfree() will not do anything. However, if the size of the block is the same size as the page size, then it does not perform merge() operation but just changes the used bit to zero and calls munmap() function to return the page.

In get_page() function, it allocates a new page of memory using mmap(). It requests a block of memory of size equal to the 4096 bytes. If mmap() is succeeded, the function initializes a bm_header for the allocated memory block. The 'next' field of the header is set to 0x0, it means this is the last block in the linked list. The 'used' field is set to 0, it means the block is not used. The 'size' field is set to PAGE_SIZE, it means the size of the block. the function returns the pointer to the allocated memory block.

In sibling(void*h), it iterates over the linked list of memory blocks, counting the number of blocks with the same size as h that precede h in the list. If it meets h, it determines the counting number is   an even or odd number. If the number is even, it returns a pointer to the block that precedes h in the list, if that block has the same size as h. If the number is odd, it returns a pointer to the block that follows h in the list, if that block has the same size as h. If it does not find a block with the same size as h, it returns a null pointer.

In split(void *p), it takes a pointer to a memory block header p and divides the block into two blocks. If the size of block is 4, the function returns the pointer to the block header and exits. Because the smallest block size is 16 bytes. And using get_splited_address() function and connecting it with the original block. 'used' field of the new block is initialized to 0, and the 'size' field is set to one less than the size of the original block. because 'size' is an exponent of 2, so size is -1, there is an effect of dividing by 2. The pointer to the block that comes immediately after the original block is saved in the 'next' field of the new block, and the 'next' field of the original block is set to point to the new block. When all the work is done, the function returns the pointer to the original block header. The newly split blocks can then be inserted into the memory block linked list to be allocated later.

And get_splied_address(void*p) is used as the helper function of the split function. it takes a pointer to a memory block header 'p' as an argument. 'p' pointer to the 'bm_headeer_ptr' type and stores it in the 'hp' variable. It then calculates the size of the block in bytes by raising 2 to the power of 'hp->size' and storing the result in the 'size_in_bytes' variable. To return a pointer to the header of the new block, the function adds ' size_in_bytes / 2' to the 'p' pointer and returns the resulting pointer.

When a user calls fitting(size_t s), it takes a size ‘s’ as an argument and searches the memory block linked list for a block that is large enough to accommodate the requested size ‘s’. It initializes a ‘best_block_ptr’ to NULL and a size ‘best_fit_size’ to value greater than the maximum size. It then iterates over the memory block linked list, starting from the next block after the ‘bm_list_head’ block, and checks each block to see if it is not used and has a size greater than ‘s’, and is either the best fit so far or the first fit found. If the block meets these conditions, the function updates ‘best_block_ptr’ and ‘best_fit_size’ to point to this block and size. After the loop finishes, the function returns ‘best_block_ptr’, which points to the header of the best fit or first fit block that can be used to allocate the requested size ‘s’. If a block is not found, the function returns NULL.

After finding a fit block of memory, the function splits the selected block into two blocks until it finds a block that is just large enough to hold the requested size ‘s’. It returns a pointer to the selected block.

In merge(void *p), it call a helper function mergex(void *p, void *q). mergex function takes two pointers to memory blocks p and q and merges them into a single memory block. If the two blocks are adjacent in memory, it updates the header of the first block to reflect the increased size and its new next pointer. If the blocks are not adjacent, the function raises an error message and aborts. merge(void *p) is a role that merges memory blocks. The pointer p, passed as an argument, points to the memory block to be merged. Call the sibling function to obtain the pointer to the sibling block. If there is no sibling block or it is already in use, it cannot be merged, so 0x0 is returned. If merge is possible, the mergex function is called to merge the two blocks. It connects the two blocks to create one block and returns the pointer to the merged block.

3. Evaluation

We evaluated the program by the three test cases with the fixed inputs for each test case.

We can say the first test case was successful. In the test case, the user request bmalloc 4 times and free one of them. Through the test case, we could evaluate that the bmalloc library in the three aspects. The first one we could evaluate is that it can allocate memory in address space using mmap() and deallocate using munmap(). The second one is that it can pick the desired block depending on the picking policy (best pick / first pick) and split the block as smaller as possible to reduce the internal fragment. The last one is that it can a sibling block of a block and merge them if they are both unused in order to prepare to serve a request of bigger memory size. So, in the first test case, we could evaluate the general functional requirement of bmalloc library by testing bmalloc(), bfree(), and their helper functions which were merge(), split(), and fitting().

We can say the second test case was successful. In the test case, the user designed an linked list that contains unique elements. Through the test case, we could evaluate the user can put integer values and utilize it as an linked list.

We can say the third test case was not successful. In the test case, the user designed an linked list that contains character arrays. Through the test case, we could evaluate the user can use the pointer value as a header of character array. But in a process of an initialization of the user defined linked list, the second block in the linked list in bmalloc acted weird. The size information in the header initialized as 0 according to bmprint(). But the user still could store, read, and delete blocks that contains character arrays.

So, our bmalloc library was successfully running in the first two cases but acting weird in the last case.

4. Discussion

The first topic is what can make mmap fail. We looked up in which conditions mmap system call can fail and we found the four major reasons. The first case is when fragmentation occurs due to too many files mapped on a address space. The second case is when address space is just too small to map large files especially on 32-bit system. For example, let say a address space is 4GB and 2GB is reserved by the kernel and only 2GB is available. Since users mappings have to share space with the program;s code and stracks and heap, the file may not fit in the address space. The third case is when there is a lack of swap space when users create a private copy-on-write mapping. The last case is when user tries to map a file that is not permitted.

The second topic is a protection of accessing header address in bmalloc. Currently, there is no protection that bmalloc library provides in order to prevent users from accessing a header address of a block. If users access there and overwrite any value there, then linked list structure will be damaged and a lot of blocks may be lost which can cause memory leaks in heap memory. And the worst thins is there could be any unexpected bugs. So, we discussed about this problem and suggested restoring system when the linked list structure is damaged by saving all header information in extra memory space. But it still does not solve the main problem which is the user can access the invalid memory. We would like to handle this problem in kernel level but since the bmalloc library handles the memory in application layer, we could not find a better solution such as making those address as inaccessible by users.

5. Conclusion

We implemented bmalloc library in this assignment. In order to implement it, there were some steps we needed to take.

The first step was to request free memory in address space to kernel by using mmap() function.

The second step was to implement an linked list data structure to manage given space in a buddy algorithm manner to prevent internal fragment. In the linked list structure, one block is consisted of a header part and payload part. A header contains an information about the block like a metadata and a payload. In the buddy algorithm, it provides 9 different size of blocks from 2^4 bytes to 2^12 bytes and manages a given space from mmap() by splitting and merging the blocks. And there is the helper function “fitting()” which returns a fitting block that can service requested size. In order to merge two blocks, the program needs to figure one’s sibling block out. This is where a helper function “sibling()” is introduced.

So, bmalloc library can help user to allocate and deallocate memory in heap space efficiently by using a buddy algorithm.