Principles of Operating Systems and Concurrent Programming, Fall 2022

Report of Laboratory 2: Memory Allocator

Prepared by: Louis Choules, Dagvanorov Lkhagvajav

Date: 10 October 2022

Step 1. Fast Pool(s):

A Fast Pool is made of blocks with a fixed size. In this lab task, we are given 3 fast pools with specific block sizes, A, B, and C respectively.

The following Import design decisions are needed to be made to implement the memory allocator for a Fast Pool(s):

1. Free Block Organization

   In order to track the free blocks in the pool, we use a linked list. All free blocks are linked to each other as a linked list. On the initialization, we make a chunk of memory using my_mmap. After we create the chunk of memory, we split the memory into a smaller blocks based on the maximum size of the pool. Then, we mark the first block as the pool->start and pool->first_free. We also mark the last block that we have as pool->end.

2. Placement

   The placement of the memory in the fast pool is to take the pool->first_free as the memory that will be allocated. We will check if that memory is NULL or not, if it is not NULL then we move the pool->first_free into the next pool->first_free and allocate that memory for our usage. But if it is NULL we just end the placement process.

3. Splitting

   In a Fast pool, there will not happen any external fragmentation. So we will use all the blocks of memory, even though we don't need to use them all. We will just have an internal fragmentation and because of that, we will use a whole block when allocating the memory.

4. Coalescing

   In a Fast Pool, there is no external fragmentation, so there is also no memory coalescing in the fast pool. We just free the block and put the freed block at the start of the pool->start. After that change, the pointer of pool->start to the one that is freed. In other words, the LIFO method is used to store the free list.

Step 2. Standard Pool:

Each block of a standard pool contains a header and a footer, at the beginning and end of the block, respectively. The header and footer store the same value, “size of the block” and “boolean value represent if the block is free or allocated”.

1. Free Block Organization

Same as the fast pool, we used a linked list to track free blocks in a standard pool. But, the difference is that the blocks in the free list are stored in address ascending order. Thus, the position of a newly freed block in the linked list will be determined by its address in the memory.

A new initialized standard pool will have a linked list with one block (node).

2. Placement

In this step, we utilized the “First Fit” policy to manage a new allocation request. The first block in the linked list of free blocks with a size, which equals to or more than the request size, will be allocated. The pre-implemented function of a header(footer) is used to extract the size of a free block. If the current block in the linked list has a smaller size than the request, the next block will be examined. This search process will be repeated until it finds the block with the desired size or reaches the end of the linked list.

3. Splitting

After allocating memory to a block, the block will be removed from the linked list of free blocks. To prevent internal fragmentation, we have implemented a strategy that splits the allocated block, if the remaining free size is higher than or equal to 32 bytes. By considering the fact that a free block should contain a header, a footer (16 bytes) and 2 pointers (16 bytes), we set this threshold. But, it should be mentioned that despite preventing internal fragmentations, it leads to more computation time.

4. Coalescing

A new freed block will be immediately merged with neighbor blocks that are free. We have implemented it in the following way. First, the allocator checks if the next block of the freed block is free. If the next is free, the blocks will be merged into a single free block. Then, the allocator checks if the previous block is free. This algorithm verifies that there will be no free adjacent blocks in the standard pool.

Utility functions:

```
void set_free_size(struct mem_std_free_block *address, size_t size){
```

```
    set_block_free(&address->header);
    set_block_size(&address->header, size);
}
```

This part is for setting the block free with the size that we propose.

```
void set_allocated_size(struct mem_std_allocated_block *address, size_t size){
    set_block_used(&address->header);
    set_block_size(&address->header, size);
}
```

This part is for set the block allocated with the size that we propose.

```
bool position_compare(struct mem_std_free_block *curr, struct mem_std_free_block *temp){
    return (void *)curr > (void *)temp;
}
```

This checks the position of the current block with the block next free block. It will return **true** if curr should be after temp based on the address. Otherwise, it will return **false**.

```
void sort_block(mem_pool_t *pool){
    struct mem_std_free_block *curr = pool->first_free;
    struct mem_std_free_block *temp = curr->next;
    bool compare = position_compare(curr, temp);
    bool isFirst = true;
    while(temp != NULL && compare){
        curr->next = temp->next;
        temp->prev = curr->prev;
        curr->prev = temp;
        temp->next = curr;
        if(isFirst){
            pool->first_free = (void *) temp;
            isFirst = false;
        }
        temp = curr->next;
        if(temp != NULL)
            temp->prev = curr;
        compare = position_compare(curr, temp);
    };
}
```

This part of the code is for sorting the location of the free block to make it a linked list which address is sorted in ascending mode.

```
int get_full_size_of_block(void *addr){
   return get_block_size(&((struct mem_std_allocated_block*)addr)->header) +
(SIZE_OF_POINTER*2);
}
```

This return the total size of the block with the pointer.

```
void merge_block(struct mem_std_free_block *freed){
   //For Merge my Free Block with the next free block if it next to each other
   if(freed->next == (void *) freed + get_full_size_of_block(freed)){
      struct mem_std_free_block* temp = freed->next;
      freed->next = temp->next;
      if(temp->next != NULL)
         temp->next->prev = freed;

      int total_size = get_block_size(&freed->header)+ get_full_size_of_block(temp);

      set_block_used(&temp->header);
      set_free_size(freed, total_size);
   }

   //For Merge my Free Block with the previous free block if it next to each other
   struct mem_std_free_block *curr = freed->prev;
   if(curr != NULL && freed == (void *) curr + get_full_size_of_block(curr)){
      curr->next = freed->next;
      if(freed->next != NULL)
         freed->next->prev = curr;
      int total_size = get_block_size(&curr->header)+ get_full_size_of_block(freed);

      set_block_used(&freed->header);
      set_free_size(curr, total_size);
   }
}
```

This code is for merge the 2 or 3 free block that next to the block that we just freed before.

```
bool useableMemory(struct mem_std_free_block *curr, int size){
   return curr != NULL && get_block_size(&curr->header) >= size &&
is_block_free(&curr->header);
}
```

This code is for checking whether the block can be allocated with the specific criteria.

Step 3. Printing the state of the pools

1. Fast Pool

In order to identify the state of a pool, we utilize "size of the pool", "size of a block in the pool", "start address of the pool" and "linked list of the free blocks".

We can calculate the number of blocks in the pool by dividing "the pool size" by "size of a block". And, initialize an array with the size of "the number of blocks in the pool". The array contains only [0, 1] values, indicating "free" or "allocated" respectively.

And, we are aware that the distance between each block is fixed size.

*Let "**s**" be the size of the pool*
Let "pool_start" be the pointer to the start of the pool
*Let "**b**" be the size of a block in the pool.*
*Let "**n**" be the number of blocks where **n = s / b***
Let "first_free" be the pointer to the head of the linked list of free blocks.
Let "status_array" be an array with size "n" // Contains [0, 1] values. 0-free, 1-allocated.
Initialize the "status_array" as 1. // assume all blocks are allocated

BLOCKS IN THE POOL	BLOCK-ALLOCATED	BLOCK-FREE	BLOCK-FREE	BLOCK-ALLOCATED	BLOCK-FREE
ADDRESS -	ADDRESS-START	ADDRESS-START + b	ADDRESS-START + 2b	ADDRESS-START + 3b	ADDRESS-START + 4b
value of STATUS_ARRAY	1	1	1	1	1
index of STATUS_ARRAY	0	**1**	2	3	4
Example (block size is 4) Start address is 20.	Address - 20	**Address - 24**	Address - 28	Address - 32	Address 36
		FIRST_FREE			

With the help of “first_free” we can find the addresses of each free block in the pool.

The index of a free block in the “status_array” can be found with the following calculation.

```
current ← first_free
while(current)
        index = (address(current) - address(pool_start)) / b
        status_array[index] = 1
        current <- current.next
```

BLOCKS IN THE POOL	BLOCK-ALLOCATED	BLOCK-FREE	BLOCK-FREE	BLOCK-ALLOCATED	BLOCK-FREE
ADDRESS -	ADDRESS-START	ADDRESS-START + b	ADDRESS-START + 2b	ADDRESS-START + 3b	ADDRESS-START + 4b
value of STATUS_ARRAY	1	0	0	1	0
index of STATUS_ARRAY	0	**1**	2	3	4
Example (block size is 4) Start address is 20.	Address - 20	**Address - 24**	Address - 28	Address - 32	Address 36
Example calculation to find index in the status_array		**(24 - 20) / 4 = 1**	(28 - 20) / 4 = 2		(36 - 20) / 4 = 4
		FIRST_FREE			
RESULT	ALLOCATED	FREE	FREE	ALLOCATED	FREE

2. Standard Pool

In order to get the state of a standard pool, we utilize the “header of a block” and “starting address of the pool”.

It is known that a block in a standard pool contains “header (**8 byte**)”, “footer (**8 byte**)” and “size of a block (**s bytes**)” in any case (allocated or free). We can extract whether the block is free or allocated and size of it using header/footer. Using this information we can visit every block in the pool repeatedly.

Block	Block-1 (free)	Block-2 (allocated)	Block-3 (free)	Block-4 (allocated)
Address	Start_address	Start_address + (16 bytes + s1)	Start_address + (16 bytes + s1) + (16 bytes + s2)	Start_address + (16 bytes + s1) + (16 bytes + s2) + (16 bytes + s3)
Header	Status: Free	Status: Allocated	Status: Free	Status: Allocated

	Size: s1	Size: s2	Size: s3	Size: s4

Step 4. Standard Pool with Best fit

In this step, we have implemented everything the same as the Step 2 - First Fit policy, except the placement strategy. While the first fit policy chooses the first block which satisfies the request size, the best fit chooses the smallest block which satisfies the request size.

The best fit policy checks every block in the linked list of free blocks. And, we have utilized a helper pointer to save the desired block. If the next free block has a smaller size but meets the request size, the helper pointer will point to the next block.

Step 5. Safety Checks

In our code, we implement safety checks for two things. Forgetting to free memory and Calling free() incorrectly.

Forgetting to free memory/memory leak is a common error when using dynamic memory allocation. To solve that problem, we add a code to check if there is any pool that has a block that is not freed. We tried to add the checker at function **run_at_exit()**. But it's not working, so we changed the code a bit in the **mem_shell.c** to check the memory state first before we exit the program. If the code is found there a pool that has at least one allocated block, it will print the warning message and also show the memory state for that pool.

```
int memory_checker(){
    bool needToFree = false;
    for(int i = 0; i < NB_MEM_POOLS; i++){
        if(used_memories[i] != 0){
            needToFree = true;
            fprintf(stderr, "[!] You need to free memory in pool %d\n",
i);
            print_memory_states(i);
        }
    }
    if(needToFree)
        return -1;
    return 0;
}
```

Test case 10

a8000
q
a5
q
f1
f2
q

Normally, when we tried this one, the program is just gonna stop at first q. But after we implement the safety check for Forgetting to free memory, the program will not stop but will send the warning message to free the memory. You can not quit the program before you free the block.

Calling free() incorrectly is an error where we tried to pass a wrong address where that address is already allocated or that address is not used by the program. We add a checker when we are free the block to know whether the address is already freed or the address is not even allocated,

```
void memory_free(void *p)
{
    int i;
    bool flag = false, error = false;

    debug_printf("enter p = %p\n", p);
    i = find_pool_from_block_address(p);

    switch (mem_pools[i].pool_type)
    {
    case FAST_POOL:
        flag = mem_free_fast_pool(&(mem_pools[i]), p);
        break;
    case STANDARD_POOL:
        flag = mem_free_standard_pool(&(mem_pools[i]), p);
        break;
    default: /* we should never reach this case */
        // assert(0);
        //TODO Change assert into something
        fprintf(stderr, "[!] The memory has not been allocated for
that\n");
        error = true;
    }
```

```
    if(flag){
        used_memories[i]--;
        print_free_info(p);
    }
    else if(!error)
        fprintf(stderr, "[!] The memory is already freed\n");
    debug_printf("exit\n");
}
```

Test case 11
a8000
a5
f1
f1
f2
q

Test Case 12
a8000
f1
f5
q

Normally when you tried to free the block twice (second f1 from test case 11), it will just print you free the block at x position, but after we update a bit in the code, it will print a warning error when you tried to free the address when you already freed. And if you tried to free the memory that is not available (f5 from test case 12) the program will just abort the message the program will terminate. We also change the code a bit so it will not terminate the program but it will print the warning message.