Computer Operating Systems

BLG 312E

Project 2 Report

ERENCULHACI
150220763
culhaci22@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 26.04.2024

0.1. Outputs

- In the main function, I call MyMalloc four times to allocate different sized memory space for 4 different example processes. Sizes for allocation requests are 300, 680, 1240, and 2560 as can be seen from the figure.

```
printf("\n Before allocation:");
DumpFreeList();

// Example processes
printf("\nProcess Allocation Status:\n");
void *P1 = MyMalloc(300, strategy);
if (P1) {
    printf("Process P1: Success, Address (16 byte is used for blockheader): %ld\n", (char*)P1 - (char*)heap_start);
} else {
    printf("Process P1: Failed\n");
}
//////////////////////////////////////////
void *P2 = MyMalloc(680, strategy);
if (P2) {
    printf("Process P2: Success, Address (16 byte is used for blockheader): %ld\n", (char*)P2 - (char*)heap_start);
} else {
    printf("Process P2: Failed\n");
}
//////////////////////////////////////////
void *P3 = MyMalloc(1248, strategy);
if (P3) {
    printf("Process P3: Success, Address (16 byte is used for blockheader): %ld\n", (char*)P3 - (char*)heap_start);
} else {
    printf("Process P3: Failed\n");
}
//////////////////////////////////////////
void *P4 = MyMalloc(2560, strategy);
if (P4) {
    printf("Process P4: Success, Address (16 byte is used for blockheader): %ld\n", (char*)P4 - (char*)heap_start);
} else {
    printf("Process P4: Failed\n");
}

printf("\n After allocation:");
DumpFreeList();

// Free allocated memory
MyFree(P1);
MyFree(P2);
MyFree(P3);
MyFree(P4);

printf("\n After deallocation:");
DumpFreeList();
```

Figure 0.1: Memory Allocations

- When I run my code, it first asks for heap size, if I type a value not exactly a multiple of the page size of the memory, It rounds up the value and initializes heap with that value as can be seen from the output, it rounds the 4000 input to the page size which is 4096. And when heap is initialized it prints the initialized size.
- Then it asks for the strategy to conduct memory allocation. Here I chose best fit with typing 0 and it prints that selected memory allocation is best fit.
- Then with the dumpfreelist function, I dump the free list before the allocations, It can be seen that it starts from 0 and has size 4096.
- Then I print the process allocation status for each process whether it is failed or succeded, for the process 4, since it's size is 2560, there is no more free space in free list after the first 3 allocations, so the memory allocation for P4 is failed. If we had initialized the heap with an 8192 heap size, it would have been succeded just like the other processes. Now I show this processes address where they are allocated in heap relatively. Since the header is 16byte for each block, it starts

with 16. So actually the header file starts with zero, and if we calculate the actual allocated space allocated for P1, Since P2's address starts with 332, and 16 byte is header, then P1 ends at 316, So we have 300 bytes allocated memory as we wanted.

- Then as it can be seen, I dump the free list again after all allocations and it shows that starting from the address 0, we have a full space sized 2276. If we sum the allocated spaces, we get 2276 too. Then it means it allocated the spaces correctly. And also it shows the remaining empty space blocks in the free list.
- After deallocation, I dump the free list again and it shows a single block with 4096 size. I use merging in MyFree() function, this ensures memory doesn't fragment too much. So we can see that after deallocation, we do not have fragmentation and have a single free block. You can see how it fragments if I disable and not use MergeBlocks() functions I created in Figure 0.7.

```
Enter the heap size: 4000

Heap successfully initialized with size : 4096

Please enter the strategy type (0: Bestfit, 1: Worstfit, 2: Firstfit, 3: NextFit): 0

Selected memory allocation strategy is : Best Fit

 Before allocation:
Addr    Size    Status
0       4096    Empty

Process Allocation Status:
Process P1: Success, Address (16 byte is used for blockheader): 16
Process P2: Success, Address (16 byte is used for blockheader): 332
Process P3: Success, Address (16 byte is used for blockheader): 1028

Memory allocation failed. No suitable block found for size: 2560
Process P4: Failed

 After allocation:
Addr    Size    Status
0       2276    Full
2276    1820    Empty

 After deallocation:
Addr    Size    Status
0       4096    Empty
root@vm_docker:/home/Ubuntu/hostVolume/COS/project2#
```

Figure 0.2: Running with Best Fit

- Here the output for worst fit can be seen, first it again asks for the heap size, and if we do not enter anything here, it initializes heap successfully with the default page size just like how I have done in this output for demonstration purposes.
- Then it can be seen that it allocates the first 3 process successfully and and fails for process 4 since there is no more space.

- Then also it can be seen that it deallocates memory successfully.

```
root@vm_docker:/home/Ubuntu/hostVolume/COS/project2# ./main

Enter the heap size:

No size entered, default heap size selected.

Heap successfully initialized with size : 4096

Please enter the strategy type (0: Bestfit, 1: Worstfit, 2: Firstfit, 3: NextFit): 1

Selected memory allocation strategy is : Worst Fit

 Before allocation:
Addr    Size    Status
0       4096    Empty

Process Allocation Status:
Process P1: Success, Address (16 byte is used for blockheader): 16
Process P2: Success, Address (16 byte is used for blockheader): 332
Process P3: Success, Address (16 byte is used for blockheader): 1028

Memory allocation failed. No suitable block found for size: 2560
Process P4: Failed

 After allocation:
Addr    Size    Status
0       2276    Full
2276    1820    Empty

 After deallocation:
Addr    Size    Status
0       4096    Empty
root@vm_docker:/home/Ubuntu/hostVolume/COS/project2#
```

Figure 0.3: Running with Worst Fit

- Here in the output I display below, the results with the first fit method can be seen.
- First it asks the heap size and I typed 5000. It rounds that value up to 8192 which is a multiple of the page size 4096.
- Then it asks for the strategy and I chose first fit.
- Here it allocates all of the request successfully since there is enough space in the heap. It shows the address in the same way before, and the it can be seen that allocation is successful by the dump free list after allocation.
- Dump Free list after allocation shows that starting from 0, there is a 4852 byte space which is full, so it means the allocation was successful. And it shows the remaining free space.
- And after deallocation, we can see that a single 8192 byte free block is in the free list.

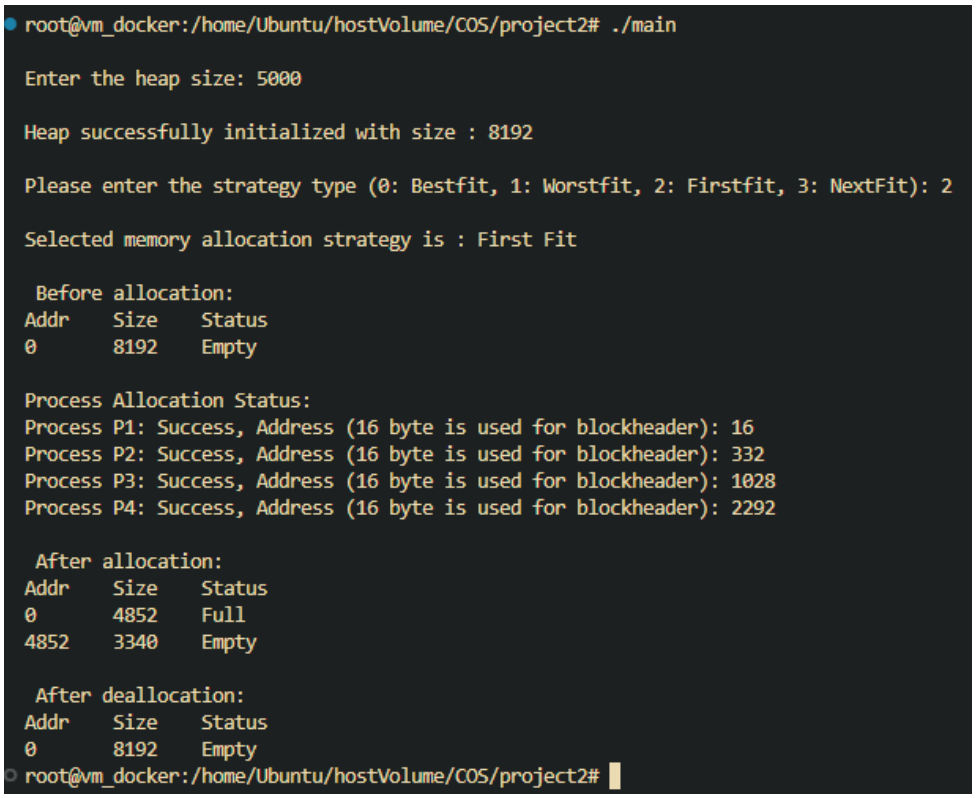

Figure 0.4: Running with First Fit

- Here, in the output shown below, I chose 3 for strategy which means next fit strategy, and it uses next fit for memory allocation. I selected 8500 heap size and it rounded up to 12288 just like before. And the allocation and deallocations are successfully made just like the previous steps.

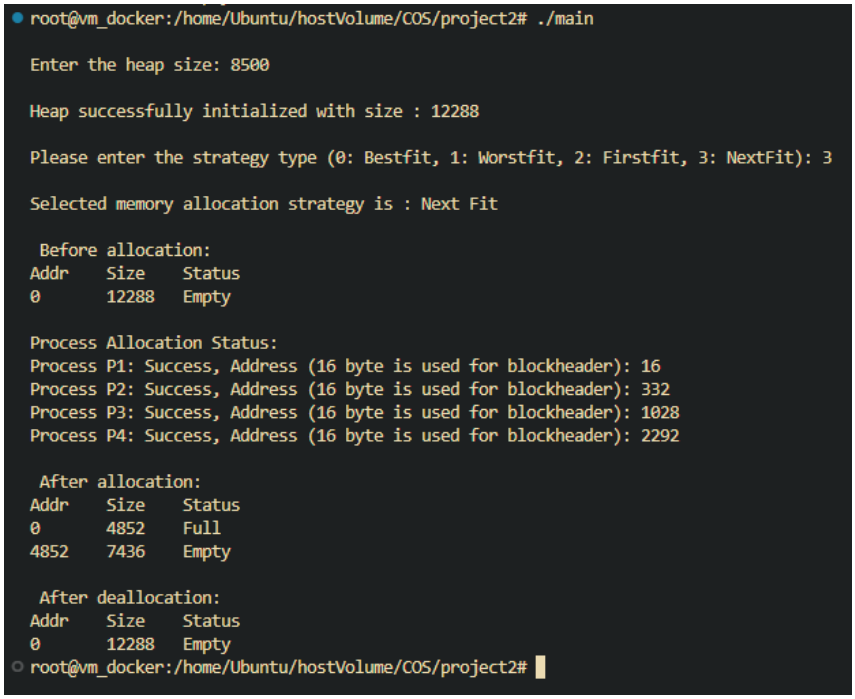

Figure 0.5: Running with Next Fit

```
    // Merge adjacent free blocks
    //MergeBlocks();
    return 0; // Free successful
}
```

Figure 0.6: Disabling Merge Function

- This code part is from MyFree() function, but differently, I disabled the MergeBlocks() function I created for demonstration purposes. Below you can see the fragmentation due to the lack of merging operation after free operations. It can be seen that after dealocations, there are many small and empty blocks which can be used for allocation. But if a bigger request comes, this small blocks won't be adequate. So I created the MergeBlocks() function to merge these small adjacent blocks into bigger blocks.

```
root@vm_docker:/home/Ubuntu/hostVolume/COS/project2# ./main

Enter the heap size:

No size entered, default heap size selected.

Heap successfully initialized with size : 4096

Please enter the strategy type (0: Bestfit, 1: Worstfit, 2: Firstfit, 3: NextFit): 1

Selected memory allocation strategy is : Worst Fit

 Before allocation:
Addr    Size    Status
0       4096    Empty

Process Allocation Status:
Process P1: Success, Address (16 byte is used for blockheader): 16
Process P2: Success, Address (16 byte is used for blockheader): 332
Process P3: Success, Address (16 byte is used for blockheader): 1028

Memory allocation failed. No suitable block found for size: 2560
Process P4: Failed

 After allocation:
Addr    Size    Status
0       2276    Full
2276    1820    Empty

 After deallocation:
Addr    Size    Status
0       316     Empty
316     696     Empty
1012    1264    Empty
2276    1820    Empty
root@vm_docker:/home/Ubuntu/hostVolume/COS/project2#
```

Figure 0.7: Free Space Fragmentation due to lack of Merging

0.2. Questions

1. **What are the main differences between “malloc” and “mmap”? If you had to make a decision between these two, which would you choose? Why?**

    - "malloc" is a standard library function in C. However, "mmap" is a system call in Unix like operating systems. "malloc" allocates a block of memory and returns a pointer to start of the block. This memory is initialized with garbage values. "malloc" is suitable for allocating generally small blocks. It can be resized with functions like "realloc". On the other hand, "mmap" maps a part of a file or device to memory and returns a pointer to that memory region. "mmap" can be used for communication between processes , memory-mapped I/O, and shared memory. This initialization is generally done with the contents of the mapped file or device unlike the garbage values in "malloc". It is generally used to allocate larger blocks.
    - If I need relatively small memory space to allocate dynamically and manage the space, I would choose "malloc", but if I want to allocate a larger space or if I want to map a file into memory or make processes share memory, "mmap" would be my choice since it is more efficient because it only loads the portions of the file that are accessed in memory. Also I can manage accessibility in "mmap". I can specify if the memory region should be read-only, read-write, or executable.

2. **What was the method/call you used in MyFree() instead of free()? Do you think it is as successful as free() at correctly and safely releasing the memory back? Why?**

    - The method I used to free the allocated memory is to create free_list which is a linked list and keep all the free memory block’s there after they are freed by MyFree() operation. After free operation, I use MergeBlocks() function I created to reduce fragmentation. This function merges the adjacent free blocks in the free list. And even if there are newly created adjacent blocks after one merge operation, it merges them either. So I think it is a reliable and successful free operation but of course, since standard free() function is tested and optimized over the time, it is obviously more reliable.

3. *Amount of free memory is shown in the first part of the report by using DumpFreeList() function and displaying the amount of free list before and after allocations.*

4. *Addresses for the processes relative to heap are shown in the first part of the report.*

5. **Prove that when you release the allocated memory in your code using MyFree(), the memory is actually freed:**

Figure 0.8: Addition of a new process with 4000byte request

- After freeing the memory which was full before, I created a new example process named P5 with a 4000byte malloc request in the main and dumped the free list again. This part will not show my main.c upload, it is just changed now for proving purposes, you can add this proof to check.
- Below you can see that first I initialized a default sized heap with 4096 byte size. Then I used worst fit strategy. (Strategy doesn't change anything for this proof).
- Then I displayed the free space by DumpFreeList() before allocation.
- After that, I allocated 3 of the processes request successfully. Their sizes are 300, 680 and 1248. And these allocated space addresses start from 16, 332, 1028 respectively. Total I have 2276 sized full space as can be seen from after allocation DumpFreeList() call. And this full space starts from 0.
- Lastly, I deallocated the space which is required for these three processes which means I used MyFree() on all these three processes. Then after freeing these processes, I created a new P5 process with 4000 size for this proof. If this new process allocates memory in the same place where I freed the previous processes' allocations, then my MyFree() actually frees the memory and I can allocate new space on these freed locations.
- So after free operations I allocated 4000byte for P5 and used DumpFreeList, as it can be seen, P5 successfully allocated and starts from address 16 which is the previously freed location. (The 16byte is for header, in DumpFreeList(), it will start from 0 not 16 since it should say that from 0 - 16 it is also full.)

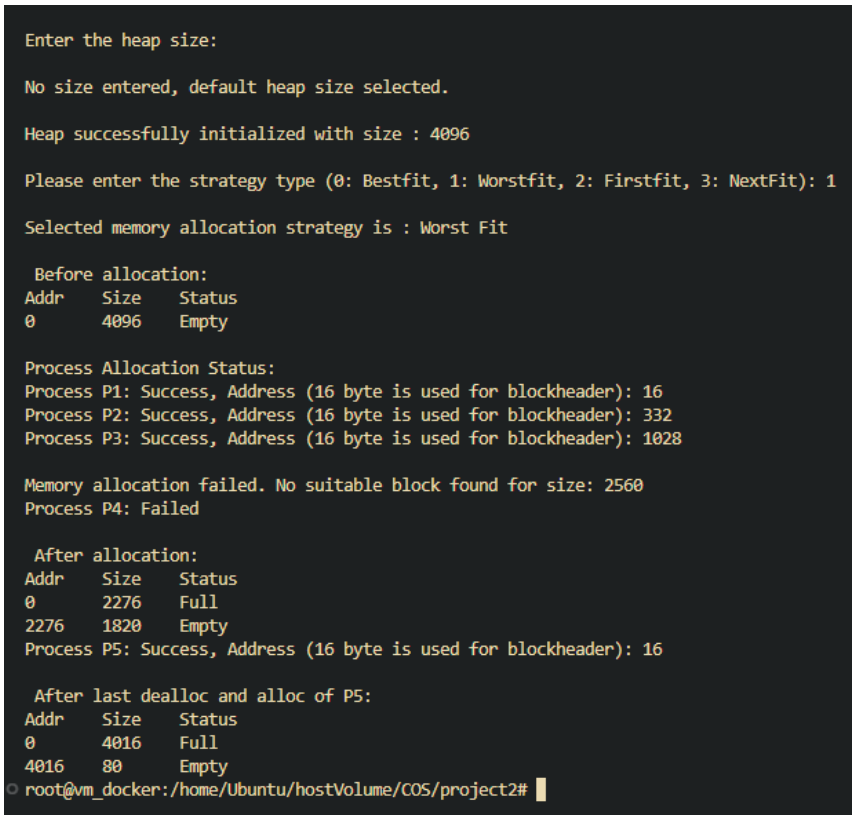

Figure 0.9: Proof of MyFree() actually frees the memory

0.3. How to Compile and Run?

To compile the code, use this command in the Linux command line:

gcc -o main main.c

To run the compiled object, use this command in the Linux command line:

./main