BLG 312E
Computer Operating Systems
Project 2 Report

Mustafa Can Çalışkan
Computer Engineering Department
Istanbul Technical University
Istanbul, Turkey
Email: caliskanmu20@itu.edu.tr

***Abstract*—In this project, developed within the scope of the operating systems course, a focus is placed on custom malloc and free functions. The project aims to evaluate the feasibility of different strategies for memory management, encompassing best fit, next fit, worst fit, and next fit strategies. Each strategy offers unique approaches in memory allocation and deallocation operations. Following the development of these functions, comprehensive tests were conducted to ensure their correctness. This study aims to provide valuable insights into the applicability and effectiveness of memory management strategies.**

I. Introduction

This project focuses on custom memory allocation functions (InitMyMalloc, MyMalloc, MyFree) within the course assignment. Various strategies (best fit, worst fit, first fit, next fit) are explored and evaluated for managing memory resources efficiently.

A custom memory allocation scheme is utilized, where memory is allocated from a pre-allocated heap using specified strategies. InitMyMalloc initializes the heap, MyMalloc handles allocation requests, and MyFree deallocates memory, also performing memory coalescing for efficient management of free blocks.

Extensive testing is conducted across different scenarios and strategies to validate correctness and efficiency. Additionally, a mechanism to dump the state of the free memory list before and after allocations provides insights into memory utilization and fragmentation.

The report provides an overview of the implemented functions, rationale behind strategy choices, design, implementation details, and testing results. The aim is to deepen understanding of memory management principles and their practical application in software systems.

II. Memory Allocation Strategies

In the project, 4 strategies (best fit, worst fit, first fit, and next fit) have been employed for memory allocation. Best fit involves selecting the smallest available block of memory, worst fit selects the largest available block, and first fit selects the first available block it encounters. Next fit is based on the implementation of first fit, but unlike the other 3 strategies, it starts searching for available blocks from the previous available node rather than the head node of the list. This helps to keep the list more uniform[1] and prevents the majority of splits from occurring at the beginning of the list.

Figures 1, 2, and 3 provide examples of the application of the best and worst fit strategies respectively.

Fig. 1. Before Split[1]

Fig. 2. After Split Using Best Fit[1]

Fig. 3. After Split Using Worst Fit[1]

III. Design and Implementation of Memory Allocation and Deallocation Functions

A. Header File

A ListNode struct has been created within the header file with the purpose of facilitating the division of the heap into nodes, forming a linked list through allocation and deallocation operations, in addition to declarations of functions.

```
typedef struct ListNode {
    size_t size;
    struct ListNode* next;
    unsigned int isFree;
} ListNode;
```

Listing 1. ListNode definition

In addition to all of these, the header file contains the declaration of pointers to the head node of the list and the last selected node for the next fit strategy.

B. InitMyMalloc

In the InitMyMalloc function, the heap to be utilized is initially obtained from the operating system through the mmap system call. Subsequently, this space is added to the linked list as the head node and marked as free. It is imperative to note that if `X` byte of data is obtained using the mmap system call, attention must be paid to ensure that there is available space for use `X - sizeof(ListNode)`.

According to the assignment description, the entered heap size will be adjusted to the nearest multiples of the page size by rounding up. For instance, if the provided heap size is 512 bytes, it will be rounded up to the nearest multiple of the page size, which could vary depending on the system but is assumed to be 4096 bytes.

C. MyMalloc

In the MyMalloc function, first, the available node that is greater than or equal to the requested size from the allocated space is selected according to the desired strategy. Then:

1) If the size of the chosen node is greater than the requested size and has enough available space to create a new node (`sizeOfChosenNode > size + sizeof(ListNode)`), the chosen node is split into two parts, the new created node is left with the remaining space, and the starting address of the requested space is returned to the user.
2) If the first condition is not satisfied, i.e., the available node is larger than the requested size but not suitable for creating a new empty node, it is treated as if there is not enough space available to prevent fragmentation.
3) If the selected node exactly matches the requested size, the starting address of that node is returned without any further partitioning.

The point to be noted is that the "starting address" referred to is the next available address from the node's metadata.

D. MyFree

In the MyFree function, initially, the node with the starting address provided as a parameter is located, and then that node is marked as free. Subsequently, the list is traversed, and if two consecutive free nodes are found, these nodes are merged.

E. DumpFreeList

This function was created for debugging purposes. The address part was adjusted to fit the format specified in the project description as `(void*)currentNode - (void*)listHead`. In order to indicate that each node also has space for its metadata in addition to its size, the starting address of each node was adjusted to show `startingAddress = SizeOfPreviousNode + sizeof(ListHead)`. The sample output is as follows:

```
Addr    Size    Status
0       512     Full
536     1024    Full
1584    256     Full
1864    768     Full
2656    1416    Empty
```

IV. Compiling and Testing

A. Compiling

The source code of the program can be compiled using the GCC compiler along with the following flags:

```
gcc -Wall -Werror *.c -o main
```

After compilation, when executing the program, the size of the heap obtained from mmap should be provided as a command-line argument. For example, if 4096 bytes are required, the program should be executed as follows:

```
./main 4096
```

B. Testing

For testing functions, 4 processes were defined in the main function. The page size was given as 4096 bytes. Prior to allocation, the list was observed as follows using the DumpFreeList function:

```
Dumping Free List before allocations:
Addr    Size    Status
0       4072    Empty
```

As seen above, the available space in the heap was determined to be `4096 - sizeof(ListNode)`. Here, the size of the ListNode object is 24 bytes.

Afterwards, child processes were created for each process using the fork() system call. Each child process called the MyMalloc function with pre-determined sizes and strategies for themselves to obtain the starting addresses of the allocated space. The output obtained after allocation is as follows:

```
Memory allocation for P1 (PID: 10659)
succeeded at address using strategy 0:
0x744161aed018
Memory allocation for P2 (PID: 10660)
succeeded at address using strategy 1:
0x744161aed230
Memory allocation for P3 (PID: 10661)
succeeded at address using strategy 2:
0x744161aed548
Memory allocation for P4 (PID: 10662)
succeeded at address using strategy 3:
0x744161aed960

Allocation for each process is completed.

Dumping Free List after allocations:
Addr    Size    Status
0       512     Full
536     768     Full
1328    1024    Full
2376    1280    Full
3680    392     Empty
```

Since the parent process cannot see the pointer obtained by the child process, each child, after the MyMalloc function,

uses the pipe() system call to send the obtained pointer to the parent process.

After completing the allocation of all child processes, the parent process performs deallocation using the MyFree function with the pointers obtained through the pipe() system call. The output obtained is as follows.

```
Memory for P1 (PID: 10659) deallocated.
Memory for P2 (PID: 10660) deallocated.
Memory for P3 (PID: 10661) deallocated.
Memory for P4 (PID: 10662) deallocated.

All Deallocations are completed.
Dumping Free List after deallocations:
Addr    Size    Status
0       4072    Empty
```

If an error is encountered during the allocation stage, for example, if an allocation cannot be performed due to insufficient available space, the output obtained is as follows:

```
Memory allocation for P1 (PID: 17841)
succeeded at address using strategy 0:
0x770184c49018
Memory allocation for P2 (PID: 17842)
succeeded at address using strategy 1:
0x770184c49230
Memory allocation for P3 (PID: 17843)
succeeded at address using strategy 2:
0x770184c49648
Not enough space for PID: 17844.
Memory allocation for P4 (PID: 17844)
failed using strategy 3.

Memory for P1 (PID: 17841) deallocated.
Memory for P2 (PID: 17842) deallocated.
Memory for P3 (PID: 17843) deallocated.
NULL pointer deallocation for PID: 17844.
```

V. Results and Discussion

In conclusion, the InitMyMalloc, MyMalloc, and MyFree functions were implemented and tested using various strategies. The debugging phase was observed with the DumpFreeList function.

If a comparison between malloc and mmap is necessary, it is pertinent to note that malloc() is a standard library function for dynamic memory allocation, operating within the heap segment of a process's virtual memory. It provides simplicity and flexibility but lacks fine-grained control and advanced security features. In contrast, mmap() is a system call enabling memory mapping, offering more control over memory management, efficient file I/O operations, and enhanced security features like read-only and execute-only permissions. While malloc() is commonly used for general-purpose memory allocation, mmap() is preferred in scenarios requiring advanced memory management and stronger security measures.

Due to being beyond the scope of the assignment, lock mechanisms were not added to the MyFree function to make access by multiple threads safe. Therefore, unlike the free() function, the MyFree function may lead to various concurrency issues.

VI. Conclusion

The project delves into custom memory allocation functions developed within an operating systems course, evaluating strategies such as best fit, worst fit, first fit, and next fit. Through meticulous design and implementation, the study explores the feasibility of these strategies, ensuring correctness and efficiency through comprehensive testing. Insights gained from the project shed light on memory management principles and their practical application, offering valuable contributions to software performance optimization.

References

[1] R. H. Arpaci-Dusseau, *OPERATING SYSTEMS: three easy pieces*. S.L.: Createspace, 2018. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/