Computer Operating Systems

Homework 2

Hakan Duran

Computer Engineering

Istanbul Technical University

Email: duranh20@itu.edu.tr

***Abstract*—In this homework, free-list implementation of operating systems is made by using mmap() system call, necessary data structures and algorithms.**

I. Compilation

The command for compilation is:

gcc -o app main.c MyLibrary.c

II. System

A. Introduction

I have a global variable called mem_init, which represents the start point of memory which gathered by mmap(). My memory system starts with InitMyMalloc function:

```
int InitMyMalloc (int HeapSize){
    mem_init = mmap(NULL, HeapSize, PROT_READ|
        PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS,
         -1, 0);

    // The first sizeof(void*) is reversed for
        pointer of head
    void* head = mem_init + sizeof(void*);
    int size = HeapSize - ( sizeof(int) +
        sizeof(void*) + sizeof(void*)) - sizeof
        (void*);
    void* next = NULL;
    void* prev = NULL;

    // Let's load our first header into memory
    memcpy(head, &size, sizeof(int));
    memcpy(head + sizeof(int), &next, sizeof(
        void*));
    memcpy(head + sizeof(int) + sizeof(void*),
        &prev, sizeof(void*));

    // The first reversed area is for free list
        's head
    memcpy(mem_init, &head, sizeof(void*));
}
```

The first 8 byte of memory is reserved area. In this area, there is a special pointer which points to free-list's start pointer, head.

Headers consists of size (int), next-header (void*), prev-header (void*). Each header took 20 bytes.

B. MyMalloc

MyMalloc starts with general methods. Strategies will be showed later with explanatory screenshots.

```
void *MyMalloc (int size, int strategy){

    // selected is pointer which will represent
         allocated space
    void* selected = NULL;

    if (strategy == 0){ // Best-fit }
    else if (strategy == 1){ // Worst-fit }
    else if (strategy == 2){ // First-fit }
    else if (strategy == 3){ // Next-fit }
    else {
        printf("Strategy type: %d, FAILED\n",
            strategy);
        exit(EXIT_FAILURE);
    }
```

After ”selected” pointer determined, there will be splitting in this area. t_selectedsize, t_selectednext, t_selectedprev are previous values for this area (before splitting). ”selected” pointer is for allocated space, its size is determined by function call, next and prev values are unneccessary. Because the free space splitted, the new pointer for free are should be determined. newptr is the new pointer. :

```
    /* The are for splitting is happening here
        */
    // selected : Allocated area
    // newptr : New free area

    void* nullptr = NULL;
    memcpy(selected, &size, sizeof(int));
    memcpy(selected + sizeof(int), &nullptr,
        sizeof(void*));
    memcpy(selected + sizeof(int) + sizeof(void
        *), &nullptr, sizeof(void*));

    void* newptr = selected + size + sizeof(int
        ) + sizeof(void*) + sizeof(void*);

    int newsize = t_selectedsize - size - (int)
         sizeof(int) - (int) sizeof(void*) - (
        int) sizeof(void*);
    memcpy(newptr, &newsize, sizeof(int));
    memcpy(newptr + sizeof(int), &
        t_selectednext, sizeof(void*));
    memcpy(newptr + sizeof(int) + sizeof(void*)
        , &t_selectedprev, sizeof(void*));

    // By using select pointer's prev pointer,
        we reach previous header and then
        change it's next value.
    if (t_selectedprev){
        memcpy(t_selectedprev + sizeof(int), &
            newptr, sizeof(void*));
    }

    // In a case where head's header changes
        its place
    if (head == selected){
        memcpy(mem_init, &newptr ,sizeof(void*)
            );
    }

    printf("Strategy type: %d, SUCCESS, %p\n\n"
        , strategy, selected);
    return selected;
}
```

There is also two important case here. If there is t_selectedprev, that means the free space whose next pointer is looking at that selected free space is changed its pointer. By using prev pointer, we can correct the previous header’s next area. The second important case is the chance of selected and head are same. In that case, inside of mem_init, which holds the head pointer should be corrected.

C. MyFree

MyFree starts with a short code. It makes the new freed area as head:

```
int MyFree (void *ptr){

    if (ptr == NULL){ return 0; }

    // The freed space is added in free-list
        and became head.
    // ptr->next = head;
    // head = ptr;

    memcpy(ptr + sizeof(int), mem_init, sizeof(
        void*));
    memcpy(mem_init, &ptr, sizeof(void*));

    void* head;
    memcpy(&head, mem_init, sizeof(void*));
```

1) Coalescion: Coalescion have some stages.

1) In the first, find the head area's first and last pointers.
2) Start iterate over free-list. Find first and last pointers of each space.
3) Compare and find superpositions.
4) Merge.

Here the code can be seen:

```
    /* COALESCION */

    void* iter = NULL;
    void* old_iter = head;

    // start_ptr and end_ptr are head's start
        and end points.
    // Since the new freed space is now head,
        it is enough to check
    // head's previous and next areas.

    void* start_ptr = head;
    void* end_ptr = head + headsize + sizeof(
        int) + 2*sizeof(void*);

    while(iter){
        int itersize;
        memcpy(&itersize, iter , sizeof(int));

        // i_start_ptr and i_end_ptr are areas
            where iter's start and end points.
        void* i_start_ptr = iter;
        void* i_end_ptr = iter + itersize +
            sizeof(int) + 2*sizeof(void*);

        if (i_start_ptr == end_ptr){

            // If iter's start is same with
                head's end:

            memcpy(old_iter + sizeof(int), iter
                 + sizeof(int) , sizeof(void*))
                ;
            int sum = headsize + itersize +
                sizeof(int) + 2*sizeof(void*);
            memcpy(head , &sum, sizeof(int));
        }

        if (i_end_ptr == start_ptr){

            // If iter's end is same with head'
                s start:

            memcpy(old_iter + sizeof(int), iter
                 + sizeof(int) , sizeof(void*))
                ;
            memcpy(&iter, &head, sizeof(void*))
                ;
            int sum = headsize + itersize +
                sizeof(int) + 2*sizeof(void*);
            memcpy(head , &sum, sizeof(int));
        }

        old_iter = iter;
        memcpy(&iter, iter + sizeof(int),
```

```
            sizeof(void*));

    }
}
```

III. Coalescing algorithms and memory allocation strategies

A. Coalescing

Example for showing coalescing:

```
void* ptr = MyMalloc(100, 2);
DumpFreeList();
void* ptr2 = MyMalloc(200, 2);
void* ptr3 = MyMalloc(300, 2);
void* ptr4 = MyMalloc(300, 2);
void* ptr5 = MyMalloc(300, 2);
DumpFreeList();
MyFree(ptr4);
DumpFreeList();
MyFree(ptr3);
DumpFreeList();
```

In here, coalescing can be observed. In 24th line, allocated area was started at 8 and its size was 660. Empty area starts at 668 and its size was 320. After calling MyFree(ptr3), as it can be seen from 31th line, allocated area's size became 340 and empty area's space is 650. It shows that coalescing has succesfully carried out.

```
Strategy type: 2, SUCCESS, 0x75ded246e008

Addr    Size    Status
0       8       Reversed for head
8       120     Full
128     3968    Empty

Strategy type: 2, SUCCESS, 0x75ded246e080

Strategy type: 2, SUCCESS, 0x75ded246e15c

Strategy type: 2, SUCCESS, 0x75ded246e29c

Strategy type: 2, SUCCESS, 0x75ded246e3dc

Addr    Size    Status
0       8       Reversed for head
8       1300    Full
1308    2788    Empty

Addr    Size    Status
0       8       Reversed for head
8       660     Full
668     320     Empty
988     320     Full
1308    2788    Empty

Addr    Size    Status
0       8       Reversed for head
8       340     Full
348     640     Empty
988     320     Full
1308    2788    Empty
```

B. Best-fit

Example for showing best-fit:

```
void* ptr = MyMalloc(100, 2);
void* ptr2 = MyMalloc(250, 2);
void* ptr3 = MyMalloc(100, 2);
void* ptr4 = MyMalloc(200, 2);
void* ptr5 = MyMalloc(100, 2);
void* ptr6 = MyMalloc(50, 2);
void* ptr7 = MyMalloc(100, 2);
MyFree(ptr6);
MyFree(ptr4);
MyFree(ptr2);
DumpFreeList();
void* ptr8 = MyMalloc(120, 0);
DumpFreeList();
```

It can be seen that there are empty areas whose sizes sequeantally are 270, 120 and 70. Since we want to allocate 120 byte area (120+20, its header), best-

fit should select the least area which will meet the requirements:

```
Addr    Size    Status
0       8       Reversed for head
8       120     Full
128     270     Empty
398     120     Full
518     220     Empty
738     120     Full
858     70      Empty
928     120     Full
1048    3048    Empty

Strategy type: 0, SUCCESS, 0x7f503dde8206

Addr    Size    Status
0       8       Reversed for head
8       120     Full
128     270     Empty
398     260     Full
658     80      Empty
738     120     Full
858     70      Empty
928     120     Full
1048    3048    Empty
```

In the 18th line, it is seen full space's are became 260, it was 120. It means it selected the area which is the best fit for him.

C. *Worst-fit*

Example for showing worst-fit (the same allocation used with best-fit code except allocation of ptr8 is made by strategy 1):

```
Addr    Size    Status
0       8       Reversed for head
8       120     Full
128     270     Empty
398     120     Full
518     220     Empty
738     120     Full
858     70      Empty
928     120     Full
1048    3048    Empty

Strategy type: 1, SUCCESS, 0x71f0d27cd418

Addr    Size    Status
0       8       Reversed for head
8       120     Full
128     270     Empty
398     120     Full
518     220     Empty
738     120     Full
858     70      Empty
928     260     Full
1188    2908    Empty
```

It can be seen the biggest area selected, which is big empty chunk at the bottom.

D. *First-fit*

Example for showing worst-fit (the same allocation used with best-fit code except allocation of ptr8 is made by strategy 2):

```
Addr    Size    Status
0       8       Reversed for head
8       120     Full
128     270     Empty
398     120     Full
518     220     Empty
738     120     Full
858     70      Empty
928     120     Full
1048    3048    Empty

Strategy type: 2, SUCCESS, 0x724a5baf8080

Addr    Size    Status
0       8       Reversed for head
8       260     Full
268     130     Empty
398     120     Full
```

```
518     220     Empty
738     120     Full
858     70      Empty
928     120     Full
1048    3048    Empty
```

IV. Example operation with 4 process

I've created 4 processes by using fork(). In P1, i've called MyMalloc(100, 0). MyMalloc(200,1) for P2, MyMalloc(300,2) for P3, MyMalloc(400, 3) for P4:

```
Child PID: 76094
Please enter the strategy type (0,1,2,3): 0

Strategy type: 0, SUCCESS, 0x7eb494dd9008

Addr    Size    Status
0       8       Reserved for head
8       120     Full
128     3968    Empty


Child PID: 76100
Please enter the strategy type (0,1,2,3): 1

Strategy type: 1, SUCCESS, 0x7eb494dd9080

Addr    Size    Status
0       8       Reserved for head
8       340     Full
348     3748    Empty


Child PID: 76101
Please enter the strategy type (0,1,2,3): 2

Strategy type: 2, SUCCESS, 0x7eb494dd915c

Addr    Size    Status
0       8       Reserved for head
8       660     Full
668     3428    Empty


Child PID: 76102
Please enter the strategy type (0,1,2,3): 3

Strategy type: 3, SUCCESS, 0x7eb494dd929c

Addr    Size    Status
0       8       Reserved for head
8       1080    Full
1088    3008    Empty
```

It can be seen allocation is successful in all processes.

V. Munmap()

In order to free the memory space which gathered by mmap(), we should use munmap() function. Details can be found in mmap()'s man page.

```
// Free all the memory
munmap(mem_init, memsize);

##############################################

part of man page in mmap():

 munmap()
       The munmap() system call deletes the
           mappings for the specified
       address range, and causes further
           references to addresses within
       the range to generate invalid memory
           references.  The region is
       also automatically unmapped when the
           process is terminated.  On
       the other hand, closing the file
           descriptor does not unmap the
       region.
```

VI. Questions to be answered

– What are the main differences between "malloc" and "mmap"? If you had to make a decision between these two, which would you choose? Why?

The first difference between malloc and mmap is the complexity. I think malloc function is easier to use. Second difference is their usage area. malloc function is for allocatin memory in heap for data structures. mmap function is giving you a size of area and usage of it is depend on the developer. While mmap function's usage area is much more related with mapping files or devices into the memory, in this homework, mmap's usage was looking like malloc since it was an anonymous call. Another point is inter-process communication, chunk of memory given by mmap() is useful when we want the communication of processes. If i would need to make a decision between mmap and malloc, my decision would depend on the requirements of the project. My decision would've consist of the consideration of statements above.

– 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?

Since free() function is developed by UNIX system developers, it is clear that using free() is much more correct and safe. But free() function is not working in mmap() area, we need to create our specialized free() function. MyFree() function was taking in the specified area's pointer and then make it "head". By using this way, the specified area was started to called as empty block. The mapping's deletion was made by munmap(), which i used as the last statement in my code.