Pengfei Li (pl189)

Homework 2, ECE 590 & CS320 Software Reliability.

1.Code smell:

Long method

```
void bf_free(void * ptr){
    MemNode * cur = (MemNode *)((void *)ptr - sizeof(MemNode));
    cur->isFree = 1;
    free_space_segment_size += (cur->size+sizeof(MemNode));
    if(head==NULL){
        head = cur;
        tail = cur;
        return;
    }
    MemNode * curr = head;
    MemNode * prev = NULL;
    while(curr<cur){
        prev = curr;
        curr = curr->next;
        if(curr==NULL){
            prev->next = cur;
            cur->prev = prev;
            cur->next = NULL;
            tail = cur;
            return;
        }
    }
    if(prev==NULL){
        cur->next = head;
        head->prev = cur;
        cur->prev = NULL;
        head = cur;
    }else{
        prev->next = cur;
        curr->prev = cur;
        cur->next = curr;
        cur->prev = prev;
    }

    if(cur != tail && cur->next->isFree==1 && ((size_t)cur->next - cur->size - sizeof(MemNode)) == (size_t)cur){
        cur->size += (cur->next->size+sizeof(MemNode));
        removeNode(cur->next);
    }
    // if(cur!=head){
    //     printf("diff is %ld\n", (size_t)cur - cur->prev->size - sizeof(MemNode));
    // }
    if(cur != head && cur->prev->isFree==1 && ((size_t)cur - cur->prev->size - sizeof(MemNode)) == (size_t)cur->prev){
        cur->prev->size += (cur->size+sizeof(MemNode));
        cur = cur->prev;
        removeNode(cur->next);
    }
}
```

This method has almost 50 lines (the whole file has 200 lines)

Duplicated code

```
108  MemNode * change_old_mem(size_t size, MemNode * node){
109      //find the suit free space and change it/split it to not free and free space
110      size_t size_all = size + sizeof(MemNode);
111      assert(node->isFree==1 && node->size >= size);
112      node->isFree = 0;
113      if(node->size>size_all){ //need split
114          size_t rest_space = node->size + sizeof(MemNode) - size_all;
115          node->size = size;
116          free_space_segment_size -= size_all;
117          add_node(node, rest_space, 1, size_all);
118          removeNode(node);
119      }else{ // no split, just change it to not free
120          free_space_segment_size -= (node->size + sizeof(MemNode));
121          removeNode(node);
122      }
123      return node;
124  }
```

These two functions are called twice, but it only needs to call this function once after the if-else.

Excessively long line of code

```
83      if(cur != head && cur->prev->isFree==1 && ((size_t)cur - cur->prev->size - sizeof(MemNode)) == (size_t)cur->prev){
84          cur->prev->size += (cur->size+sizeof(MemNode));
85          cur = cur->prev;
86          removeNode(cur->next);
87      }
```

This line is too long. It is difficult for reading

Mysterious Name

```
void * helper(size_t size, int is_ff){
    if(head==NULL){
        data_segment_size += (size + sizeof(MemNode));
        return (void *)open_new_mem(size)+sizeof(MemNode);
    }
    MemNode * space = is_ff==1 ? find_free_space_ff(size):find_free_space_bf(size);
    if(space == NULL){
        data_segment_size += (size + sizeof(MemNode));
        return (void *)open_new_mem(size)+sizeof(MemNode);
    }
    return (void *)change_old_mem(size, space)+sizeof(MemNode);
}
```

We can use helper function, but we should name it in a way that others can know how it works. Just naming it as “helper” is not right.

2. negatively impact

Long method: programmer must spend a lot of time reading the long method, which means it is hard for them to figure out the complex logic in the method. And make it is hard to debug and maintain in the future. Usually, the long method contains more than one sub-function, which can be abstract as another method. For example, in the bf_free method, this code can be abstracted as a method to find the right position in the list and insert the node in this position.

```
MemNode * curr = head;
MemNode * prev = NULL;
while(curr<cur){
    prev = curr;
    curr = curr->next;
    if(curr==NULL){
        prev->next = cur;
        cur->prev = prev;
        cur->next = NULL;
        tail = cur;
        return;
    }
}
if(prev==NULL){
    cur->next = head;
    head->prev = cur;
    cur->prev = NULL;
    head = cur;
}else{
    prev->next = cur;
    curr->prev = cur;
    cur->next = curr;
    cur->prev = prev;
}
```

Duplicated code: if there is duplicated code in our program, when the programmer find a bug in this duplicated code, He needs to fix the bug repeatedly. If there is a bug in the duplicate code somewhere that is not fixed, the bug may reappear in the future

Excessively long line of code: it is difficult for reading and figuring out the logic in the statement. And it is easy to make some mistakes when we modify the statement.

Mysterious Name: when others read this name, he does not immediately understand what the method does, and he needs to spend time reading the code in the method to know what the method does. This defeats the purpose of abstraction.

3. develop/design principles

Single responsibility

This principle aims to avoid that one class/method has many jobs to do. This defeats the purpose of abstraction. It is easy to write long methods and makes it more difficult to maintain the code. Therefore, single responsibility improves software reliability.

Open/closed principle

modifying the old code every time when we need to add some new functions in the program will increase the complexity of the code and make it more difficult to maintain. It will potentially bring new bugs. Open/closed principle will solve this problem. We don't need to modify the existing code, We just need to extend the original class and method. Therefore, this principle improves software reliability.

Least surprise, better no surprises at all

This principle means we need to give method/variable/class names based on their job, and avoid some meaningless names. This will make it easier for programmers to read the code and maintain it. It will avoid some side effects caused by unexpected functions. Therefore, this principle improves software reliability.

CODE:

```
//
//  my_malloc.c
//  ece650Hw1
//
//  Created by Pengfei Li on 2022/1/11.
//

#include "my_malloc.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>

void * ff_malloc(size_t size){
    //the only diff between two method is their find method
    //therefore, the second Parameter is used to control the find method
    return helper(size, 1);
}
void ff_free(void * ptr){
    //there is no diff between ff and bf free method
    bf_free(ptr);
}

MemNode * find_free_space_ff(size_t size){
    //for FF, find the first place which size is larger than given space
    MemNode * cur = head;
    MemNode * res = NULL;
    while(cur != NULL && cur->size<size){
        cur = cur->next;
    }
    if(cur!=NULL){
        res = cur;
    }
    return res;
}

//Best Fit malloc/free
void * bf_malloc(size_t size){
    return helper(size, 0);
}

void bf_free(void * ptr){
```

```
    MemNode * cur = (MemNode *)((void *)ptr - sizeof(MemNode));
    cur->isFree = 1;
    free_space_segment_size += (cur->size+sizeof(MemNode));
    if(head==NULL){
        head = cur;
        tail = cur;
        return;
    }
    MemNode * curr = head;
    MemNode * prev = NULL;
    while(curr<cur){
        prev = curr;
        curr = curr->next;
        if(curr==NULL){
            prev->next = cur;
            cur->prev = prev;
            cur->next = NULL;
            tail = cur;
            return;
        }
    }
    if(prev==NULL){
        cur->next = head;
        head->prev = cur;
        cur->prev = NULL;
        head = cur;
    }else{
        prev->next = cur;
        curr->prev = cur;
        cur->next = curr;
        cur->prev = prev;
    }
    if(cur != tail && cur->next->isFree==1 && ((size_t)cur->next - cur->size -
sizeof(MemNode)) == (size_t)cur){
        cur->size += (cur->next->size+sizeof(MemNode));
        removeNode(cur->next);
    }
    // if(cur!=head){
    //     printf("diff is %ld\n", (size_t)cur - cur->prev->size -
sizeof(MemNode));
    // }
    if(cur != head && cur->prev->isFree==1 && ((size_t)cur - cur->prev->size -
sizeof(MemNode)) == (size_t)cur->prev){
        cur->prev->size += (cur->size+sizeof(MemNode));
```

```
        cur = cur->prev;
        removeNode(cur->next);
    }
}

MemNode * find_free_space_bf(size_t size){
    //for BF, find the smallest place which size is larger than given space
    MemNode * bestFitNode = NULL;
    MemNode * cur = head;
    while(cur!=NULL){
        if(cur->size >= size){
            if(bestFitNode == NULL || bestFitNode->size > cur->size){
                bestFitNode = cur;
            }
        }
        if(bestFitNode!=NULL && bestFitNode->size == size){
            break;
        }
        cur = cur->next;
    }
    return bestFitNode;
}

MemNode * change_old_mem(size_t size, MemNode * node){
    //find the suit free space and change it/split it to not free and free space
    size_t size_all = size + sizeof(MemNode);
    assert(node->isFree==1 && node->size >= size);
    node->isFree = 0;
    if(node->size>size_all){ //need split
        size_t rest_space = node->size + sizeof(MemNode) - size_all;
        node->size = size;
        free_space_segment_size -= size_all;
        add_node(node, rest_space, 1, size_all);
        removeNode(node);
    }else{ // no split, just change it to not free
        free_space_segment_size -= (node->size + sizeof(MemNode));
        removeNode(node);
    }
    return node;
}

void * add_node(MemNode * node, size_t size, int status, size_t offset){
    //split space or add a new space by calling sbrk, which is controlled by
offset
```

```
    //if we need to call sbrk, we just need to pass 0 for offset.
    //offset is used to split a space. status is 1 when it should be free, 0 is
not free.
    //if(size == 0) return NULL;
    MemNode * newNode = NULL;
    if(offset==0){
        newNode = sbrk(size);
    }else{
        newNode = (MemNode *)((void *)node + offset);
    }
    newNode->size = size-sizeof(MemNode);
    newNode->isFree = status;
    newNode->next = NULL;
    newNode->prev = NULL;
    // printf("size is : %ld, node address is %ld\n", newNode->size,
(size_t)newNode);
    if(node==NULL){
        return (void *)newNode;
    }
    newNode->next = node->next;
    newNode->prev = node;
    if(node->next != NULL){
        node->next->prev = newNode;
    }else{
        tail = newNode;
    }
    node->next = newNode;
    // debug code, print the list and find out bugs
    // MemNode * curr = head;
    // while(curr!=NULL){
    //     printf("^^^^^^^add^^^^^^^\n");
    //     printf("size is : %ld, node address is %ld\n", curr->size,
(size_t)curr);
    //     printf("real size is %ld\n", curr->size);
    //     printf("isfree is %d\n", curr->isFree);
    //     printf("^^^^^^^^^^^^^^^^^^\n");
    //     curr = curr->next;
    // }
    return (void *)newNode;
}

void * open_new_mem(size_t size){
    //open a new space and pass 0 for offset when calling add_node
    size_t size_all = size + sizeof(MemNode);
```

```
    return add_node(NULL, size_all, 0, 0);;
}

void removeNode(MemNode * node){
    //remove any required node from a list
    if(head == node && tail == node){
        head = NULL;
        tail = NULL;
    }else if(head == node){
        node->next->prev = NULL;
        head = node->next;
    }else if(tail == node){
        node->prev->next = NULL;
        tail = node->prev;
    }else{
        node->next->prev = node->prev;
        node->prev->next = node->next;
    }
    node->next = NULL;
    node->prev = NULL;
}

void * helper(size_t size, int is_ff){
    if(head==NULL){
        data_segment_size += (size + sizeof(MemNode));
        return (void *)open_new_mem(size)+sizeof(MemNode);
    }
    MemNode * space = is_ff==1 ?
find_free_space_ff(size):find_free_space_bf(size);
    if(space == NULL){
        data_segment_size += (size + sizeof(MemNode));
        return (void *)open_new_mem(size)+sizeof(MemNode);
    }
    return (void *)change_old_mem(size, space)+sizeof(MemNode);
}

// get the size of data_segment
unsigned long get_data_segment_size(void){
    return data_segment_size;
}
// get the size of free_spaces
unsigned long get_data_segment_free_space_size(void){
    return free_space_segment_size;
}
```