CS232 Operating Systems
Assignment 03: Implementing memory management routines

Hareem Feroz (hf04097)

Fall 2019

1 Question No. 1

Here is the code for the first question.

```
#include <inttypes.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mman.h>
#include <math.h>
#include "freeSpaceMange.h"

int BLOCK_SIZE = 1024* 1024;

node_t * head = NULL;
node_t* ptr_for_unint = NULL; //will be required for unint
node_t * MAGIC_NUMBER = (node_t * )-1;

int my_init()
{
   head = mmap(0, BLOCK_SIZE , PROT_READ|PROT_WRITE,MAP_ANON|
      MAP_PRIVATE, -1, 0);
   head-> size  = BLOCK_SIZE  - sizeof(node_t); //the main size
      upon initialzing
   head-> next = 0;
   if(head == MAP_FAILED) //if unable to initialize
   {
       return 0;
   }
   ptr_for_unint = head;
   return 1;
}

void my_coalesce()
{
    //We will be linearzing the memory. head will point at first
       free list node. the loop will run until temp2 is in the
       memory limit. if we find both free node at temp and
```

```
// and the the node next to temp that is temp2 then we will
   coalesce otherwise. temp2 will move forward in all cases.
   prev_temp will keep record of where what was being pointed
//temp.

head = NULL;
node_t* temp = ptr_for_unint;
node_t* temp2 = ptr_for_unint;
node_t* prev_temp = ptr_for_unint;

char* char_forsize_temp2 = (char*)temp;
char* char_forsize_prev = (char*)prev_temp;
char_forsize_temp2 = char_forsize_temp2 + temp2->size + sizeof
   (node_t);
temp2 = (node_t*)char_forsize_temp2;
int size_check= temp->size + sizeof(node_t);

while(temp2 < (node_t*)((char*)ptr_for_unint + BLOCK_SIZE))
{

    if (head == NULL && temp->next != MAGIC_NUMBER)
    {
        head = temp;
    }

    if(temp->next != MAGIC_NUMBER && temp2->next !=
       MAGIC_NUMBER)
    {
        temp->size = temp->size + temp2->size + sizeof(node_t)
           ;


    }
    else if (temp->next != MAGIC_NUMBER)
    {

        prev_temp ->next = temp;
        prev_temp = temp;
        temp = temp2;

    }
    else
    {
        //prev_temp = temp;
        temp = temp2;
        if (temp->next != MAGIC_NUMBER)
        {
            prev_temp ->next = temp;
            prev_temp = temp;
        }
    }


    size_check = size_check + temp->size;
    char_forsize_temp2 = (char*)temp2;
```

```
        char_forsize_temp2 = char_forsize_temp2 + temp2->size +
           sizeof(node_t);
        temp2 = (node_t*)char_forsize_temp2;

    }

    prev_temp->next = 0;
}

void * my_malloc(int size)
{
    node_t* temp = head->next;
    int real_size = head->size;
    void* sptr = NULL;

    if (head->size >= size + sizeof(node_t))
    {
        head->next = MAGIC_NUMBER;
        head->size = size;
        sptr = head + 1;  //+1 means skipping size and next's
           positions
        char* char_head = (char*)head;   //since adding directly
           the size won't work here we are using char which is of
            one byte
        char_head = char_head + size + sizeof(node_t);
        head = (node_t*)char_head;
        head->size = real_size - size - sizeof(node_t);
        head->next = temp;
    }

    else
    {
        node_t* temp = head;

        while (temp->next != NULL && temp->next->size < (size +
           sizeof(node_t)))
        {
            temp = temp->next; //until we get a size where
               allocation is possible

        }
        node_t* before_temp = temp;  //we will need this pointer
           because it's next will point at new freelist node
        temp = temp->next; //the node where we will allocate
        node_t* temp2 = temp;
        /* basically we are strong before allocation the size of
           the node that will be allocated and then moving the
           temp2's
         pointer forward where temp2'next  will point at where
            temp was pointing before allocation.
         This is basically slicing

        */
        int realsize = temp->size;
        node_t* realnext = temp->next;
```

```
    temp->size = size;
    temp->next = MAGIC_NUMBER;

    char* temp_temp2 = (char*)temp2;
    temp_temp2 = temp_temp2 + size +sizeof(node_t);
    temp2 = (node_t*)temp_temp2;
    temp2->size = realsize - temp->size - sizeof(node_t);
    temp2->next = realnext;
    before_temp->next = temp2;

    sptr = temp +1;

}

if (sptr == NULL)
{
    my_coalesce();
    node_t* temp = head->next;
    int real_size = head->size;
    //void* sptr = NULL;

    if (head->size >= size + sizeof(node_t))
    {
        head->next = MAGIC_NUMBER;
        head->size = size;
        sptr = head + 1;  //+1 means skipping size and next's
           positions
        char* char_head = (char*)head;   //since adding
           directly the size won't work here we are using
           char which is of one byte
        char_head = char_head + size + sizeof(node_t);
        head = (node_t*)char_head;
        head->size = real_size - size - sizeof(node_t);
        head->next = temp;
    }

    else
    {
        node_t* temp = head;

        while (temp->next != NULL && temp->next->size < (size
           + sizeof(node_t)))
        {
            temp = temp->next; //until we get a size where
               allocation is possible

        }
        node_t* before_temp = temp;  //we will need this
           pointer because it's next will point at new
           freelist node
        temp = temp->next; //the node where we will allocate
        node_t* temp2 = temp;
        /* basically we are strong before allocation the size
           of the node that will be allocated and then moving
            the temp2's
```

```
             pointer forward where temp2'next  will point at where
                 temp was pointing before allocation.
             This is basically slicing

            */
            int realsize = temp->size;
            node_t* realnext = temp->next;
            temp->size = size;
            temp->next = MAGIC_NUMBER;

            char* temp_temp2 = (char*)temp2;
            temp_temp2 = temp_temp2 + size +sizeof(node_t);
            temp2 = (node_t*)temp_temp2;
            temp2->size = realsize - temp->size - sizeof(node_t);
            temp2->next = realnext;
            before_temp->next = temp2;

            sptr = temp +1;

        }

    }

    return sptr;
}

void my_free (void *ptr)
{
        node_t* temp_ptr = ptr;
        node_t* temp2 = temp_ptr - 1;
        if (temp2->next == MAGIC_NUMBER) //checking if even the
           ptr is allocated or no
        {
                temp2->next = head;
                head=temp2;
        }
}

void my_showfreelist()
{
    int node_no = 1;
    node_t* temp_iter = head;
    while (temp_iter!=NULL)
    {
        printf("%d:%d:%p", node_no, temp_iter->size, temp_iter);
        node_no++;
        temp_iter = temp_iter->next;
        printf("\n");
    }
}

void * my_calloc(int num, int size)
{
     void  *calloc_ptr = my_malloc(num * size);
      //calloc_ptr =
```

```
      if (calloc_ptr == NULL) //if enough space was not available
      {
        return (NULL);
      }
      memset(calloc_ptr, 0, num* size);
    return (calloc_ptr);
}

int minimum(int a,int b)
{
    if(a<b)
    {
        return a;
    }

    return b;

}

void * my_realloc(void * ptr, int size)
{

    if(ptr == NULL) // If ptr is NULL, then the call is equivalent
        to malloc(size)
    {
        ptr = my_malloc(size);
        return ptr;
    }
    if(size == 0 && ptr != NULL) // if size is equal to zero, and
       ptr is not NULL, then the call is equivalent to free(ptr).
    {
        my_free(ptr);
    }

    else //The contents will be unchanged in the range from the
       start of the region up to the minimum of the old and new
       sizes
    {
        node_t* ptr1 = (node_t*)ptr;
        ptr1 = ptr;
        ptr1 = ptr1 - 1;
        int old_size = ptr1->size;

        int min_size = minimum(old_size,size);
        void* realloc_ptr = my_malloc(size);
        memcpy(realloc_ptr,ptr,min_size);
        my_free(ptr);
        return realloc_ptr;
    }

}



void my_uninit()
```

```
{
    int check = munmap(ptr_for_unint,BLOCK_SIZE);
    if(check == 0)
    {
        ptr_for_unint = NULL;
        head = NULL;
    }

}
```

2 Resouces used

- Had discussions with Huda Feroz and Faraz Ahmed about the assignment.
- Referred to Chapter 17 of Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.
- referred to manpages of realloc,calloc,mmap,munap to understand what they do