The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

COMPUTER SYSTEM PROGRAMMING
ECE454 – FALL 2023
ASSIGNMENT-3

Dues are defined at the end of this guideline.

Objectives

- Understanding/Implementing Dynamic Memory Management and Allocation Processes
- Understanding/Working with Memory Reference methods and finding relevant Access Times
- Finding Performance issues and solutions

Note- This assignment has two parts (1 & 2) and the following is Part-1. The guideline for the Part-2 will be posted right after the related lecture being provided. Any clarification and revisions to this assignment will be posted on Quercus,

3.1 Implementing Dynamic Memory Allocation (A3 – Part-1)

In this Lab, you will explore the Dynamic Memory Management and Allocation techniques on Heap Segment, which are implemented through various functions and system calls. You should simulate those major functions to find out about the challenges in implementing each of them and about the performance issues in managing /allocating the memory dynamically.

3.1.1 Dynamic Memory Allocation Processes

In this section, you will define/code the major methods in your program to simulate the Dynamic Memory Allocation Processes. These methods should be defined based on the following prototypes and description in your file "a3_malloc.h", and should be defined in your file "a3_malloc.c": [20 Marks]

Note- These methods are coded initially at this point, but will be improved at the next steps in this assignment. So feel free to improve your code at the next steps as you wish.

- `int m_init(void);`
  This method should be called to initialize the heap area in your program, before calling any other methods.
  It returns 0 for a successful initializing of the Heap and returns not 0 in case of any problems.
  Use this method to make an initial size for the Heap segment. All memory allocation for dynamic usage should be done within this part.

- `void *m_malloc(size_t size);`
  This method returns a pointer to the allocated block in the Heap with `size` bytes (at least). This is an entire separated block within the Heap and should not have any overlap with the other allocated blocks.
  The method should search to find the best fit in the Heap for that given size and allocate it. In this process the memory has already been divided into various size of blocks. To manage these blocks a mechanism is provided at section 3.1.3 using a linked-list. Each node in this linked-list, naming `h_Node`, is connected to a block in the Heap and contains its information. Thus for tracking the information about each block, its relevant `h_Node` should be checked.

- `void m_free(void *ptr);`
  This method returns the allocated block pointed to by `ptr` to the free space of the Heap.
  This method should work if and only if the `ptr` is valid, meaning that the `m_free(ptr)` should be called following a call to `m_malloc()` or `m_realloc()` that already generated its `ptr` parameter.
  Like the real free() method, the `m_fee()` should set the allocated block as a free block now, and this should be done by modifying the information in the relevant `h_Node` of that block. Moreover, it should also check if the previous and/or next blocks are free or not, and if yes, join them to the current free block to make a bigger free block and finally update all the relevant `h_Nodes`.

- `void *m_realloc(void *ptr, size_t size);`
  This method returns a pointer to the new allocated block in the Heap with `size` bytes (at least) after resizing an old block pointed to by `ptr`. This is an entire separated block within the Heap and should not have any overlap with the other allocated blocks.
  This method changes the size of the old block pointed to by `ptr.` The address of the new `ptr` could be the same as the old `ptr` or be different. Note that the content of the common parts between the old and the new blocks should be the same. For example if the old block has the size=200 bytes and the new block has the size=250 bytes, this means that the first 200 bytes of both blocks are the same (actually the old block will be replaced by the new one, but the contents should be kept for the common size).

Note- You cannot use library functions: malloc(), calloc(), realloc() and free(), which are all defined in the <stdlib.h> header file. Use man (Unix help) to see the semantics of these real methods in the C library.

3.1.2 Managing/Controlling the Heap Blocks

In this section, you will define a Linked-List "`h_List`" to be used for managing/controlling the blocks in the Heap. Each Node in this list has the following structure "`h_Node`": [15 Marks]

```
struct h_Node{
  int STATUS;              defines the status of the block: 0 ifit is free and 1 if it is blocked
  size_t SIZE;             defines the size of the block in bytes
  void   *c_blk;           defines the starting address of the current block
  void   *n_blk;           defines the starting address of the next block
  struct h_Node * NEXT;    points to the next h_Node containing the data for the next block linked to this one
};
```

Each block has a corresponding h_Node containing its information. The link list of all h_Nodes contains the information about the whole Heap. Following each task of: m_malloc(), m_realloc(), and m_free() the content of the relevant h_Node and the structure of this Linked-list should be updated.

The following method should be implemented to display the layout of the Heap after each modification:

```
void h_layout(struct h_Node *ptr);
```

3.1.3 Supporting Methods

In this program you should use `sbrk()` and/or `brk()` functions, which are the basic memory management system calls used in Unix operating systems in order to control the amount of memory allocated to the data segment in the process. These functions are typically called from a higher-level memory management library function such as `malloc()`, which is being simulated in your program.

These functions dynamically change the amount of space allocated for the data segment of the application by resetting the program break in the process. The program break is the address of the first location beyond the current end of the data segment. When the break value is increased, the amount of available space will be increased. The available space can be initialized to a value of zero. It is recommended to start working with these functions (mostly with `sbrk()`), using the following `test.c`: [5 Marks]

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
      void *c_break = sbrk(0);
      printf("1- %p \n", c_break);
      c_break = sbrk(0);
      printf("2- %p \n", c_break);
      c_break = sbrk(0);
      printf("3- %p \n", c_break); }
```

It is recommended to change `test.c` to see the various break addresses and the way it works which you might find it useful before working on this assignment.

3.1.4 Checking for Heap Consistency

In this section, you will define/code a method to check the consistency among the blocks with the following prototype:
[5 Marks]

```
        int m_check(void);
```

This method will check if the Heap is consistent or not. In other words, it will check if the next and/or previous blocks are free to join them with the current block, after a free method called for the current block (the Coalescing concept).
It returns 0 for a successful consistency checking of the Heap, and returns -1 (or none zero) in case of any problems.

3.1.5 Test-Driver for Testing Heap Performance

-Use the following driver in the main() in “a3_malloc.c”, to test your simulated Heap and its methods. Also write 2 more drivers to show the behavior of your simulated Heap, in one of them making your `m_malloc()` function to fail.
- Find the “Space Utilization” for each driver above, as the ratio below and compare them with each other:
    (total amount of memory allocated via m_malloc() or m_realloc() but not yet freed by m_free() / size-of-heap)
-Find the number of operations per second executed through each driver
[5 Mark]

```
        …
        h_layout(h_List);
        char  *pt1 = m_malloc(2000);
        h_layout(h_List);
        char  *pt2 = m_malloc(500);
        h_layout(h_List);
        char  *pt3 = m_malloc(300);
        h_layout(h_List);
        m_free(pt2);
        h_layout(h_List);
        char  *pt3 = m_malloc(1500);
        h_layout(h_List);
        …
```

Note – Feel free to add/define other methods, structures and variables to complete your Heap Processes simulation in this assignment.

Deliverables & Submission for Assignment-3 (Part-1) [Total 50 Marks = weight 10%]

-The teams should submit their final report including: source codes with a brief description for each function you add in the program (what are input parameters, function process and outputs that can be in the comments), header files and any other relevant files, through the submission folder on Quercus by Thursday Oct26, at 8:59am
And then present their work in the labs (on the same day) to be evaluated by the TAs through Q/A [30 Marks]
The TAs will also evaluate your partial work (3.1.2 and 3.1.3, [20 Marks]) in the Labs on Thursday Oct12

-Make sure to add the names and student numbers of your Team members, and the Lab section your team attends at the top part of your final report. Only ONE final submission is required from each team.
-The teams should save all their answers in a final report (plain text format) under this naming rule:
“PRAxx_Tyy_Az.txt” where “xx” is the Lab section (01 for 12-3pm, and 02 for 9am-12pm), “yy is the Team number and “z” is the Assignment number, e.g. “PRA01_T06_A3.txt” is the name for submission from Team06 for A3 working in the Lab on Thursday 12-3pm.

Good Luck.

Useful Resources:

-All C programming language and libraries references
-All GCC compiler references
-All Unix/Linux references

3.2 Data Alignment in Memory Management (A3 – Part-2)

In this Lab, you will explore the Data Alignment process in Memory Management and techniques on Optimizing this process.

Objectives

- Understanding/working with Data Alignment in Memory Management
- Optimizing CPU access to Memory through Alignment Process

A data object (variable) has 2 properties: the storage location (its address in memory) and the value (its content in the memory based on its type). Data Alignment means that the address of a data object can be evenly divisible by any power of 2, or data object can have alignment by any power of 2. For instance, if the address of a data is 0x1A4 (420 in decimal), then it has 4-byte alignment because the address can be evenly divisible by 4. It can be divided by 2 or 1 as well, but 4 is the highest number that is divisible evenly.

In each access to memory, CPU can refer to an address of 4-byte or 8-byte boundary instead of 1-byte boundary, which means read and write higher amount of data in a faster mood. And this ends up to a better performance for CPU each time accessing to memory. Otherwise, if the data is misaligned of 4-byte boundary (for example), CPU has to perform extra work to access that data at each time: load data in 2-times access, take out unwanted bytes and then combine them together to have the entire data required. And this process definitely slows down the CPU performance and wastes the CPU cycles each time to have access to the right data from memory.

To work with address alignment and its optimizing process, complete and run the following code as instructed:

```
//Define/Declare structure a_test1 and its array container as:
struct a_test1 {
  short a;
  int x;
  short b;
} table1[10];
struct a_test1 *p1, *q1;

// a-Print the sizeof each member type in struct a_test1 (here short and int)

// b-Print the seize of table1[0] and then the sizeof the whole table1

// c-Print the alignment and address of table1[0]

// d-In a loop, print the aligned addresses of the previous and the current elements in table1 (starting with table1[1]),
// and then the difference between their addresses showing the length of the previous element in the array



//Optimization: Add a new structure a_test2 and its array container as:
struct a_test2 {
  short a;
  short b;
  int x;
} table2[10];
  struct a_test2 *p2, *q2;

//Apply the same instructions above (a-d) on the new a_test2 struct,
//save the results in proper tables and describe about them in your report
// your optimized structure will be compared with its initial format
```

```
//define a new structure: a-test3 as below and apply the same instructions above (a-d) on it
Struct a_test3 {
   char c;
   short s;
   int k;
   long l;
   float f;
   double d; } table3[10];
struct a_test3 *p3, *q3;

// Try to optimize it, and save the results in proper tables and describe about them in your report
// your optimized structure will be compared with its initial format
```

Deliverables & Submission for Assignment-3 (Part-2) [Total 10 Marks = weight 5%]

-The teams should submit their final report including: source codes with a brief description for each function you add in the program (what are input parameters, function process and outputs that can be in the comments), header files and any other relevant files, through the submission folder on Quercus by Thursday Oct26, at 8:59am

(You should have separate submissions for part-1 and part-2)

And then present their work in the labs (on the same day) to be evaluated by the TAs through Q/A [10 Marks]

Good Luck.

Useful Resources:

-All C programming language and libraries references

-All GCC compiler references

-All Unix/Linux references