TECHNION
Israel Institute
of Technology

Computer
Science
Department

Operating Systems – 234123

<u>Homework Exercise 4 – Wet</u>

Teaching Assistant in charge:

Or Keret

Assignment Subjects & Relevant Course material:

Virtual Memory & Memory Management

Recitations 10-11, Lecture 8-9

Abstract and Assignment Objectives

The infamous rivalry between the EE and CS departments has intensified. Last week, while it was dark and rainy outside, the EE operating systems students hacked into our systems and stole the memory allocation units from your Linux virtual machines!

Now, the `malloc()` family of functions no longer exists.

It is time to fight back. You must implement a new memory allocation unit and show the EE OS students your strong OS skills.

The CS department's honor and future are now in your hands!

On a more serious note:

In this homework you will implement a simple memory allocation library, which will include your own implementation of the notorious `malloc()`, `free()`, `calloc()` and `realloc()` functions.

The homework consists of **four parts**, out of which **three are mandatory** and **one is optional**. The optional part can grant you extra credit. The homework increases in difficulty, with each part relying on the understanding and implementation of the previous parts. Therefore, you are encouraged to follow it step-by-step (**recommendation – do not start reading one part until you finish implementing the one before it**). At the end of this homework you will have gained knowledge and skills in **virtual memory**, **memory regions**, **memory-related system calls,** and ***libc***.

Throughout this homework, you cannot use any of `malloc()`, `free()`, `calloc()`, `realloc()`, `operator new`, `operator delete`, or any library/language-specific memory allocating function. Failure to comply with this may result in the disqualification of your homework.

Introduction

About libc and Memory Management

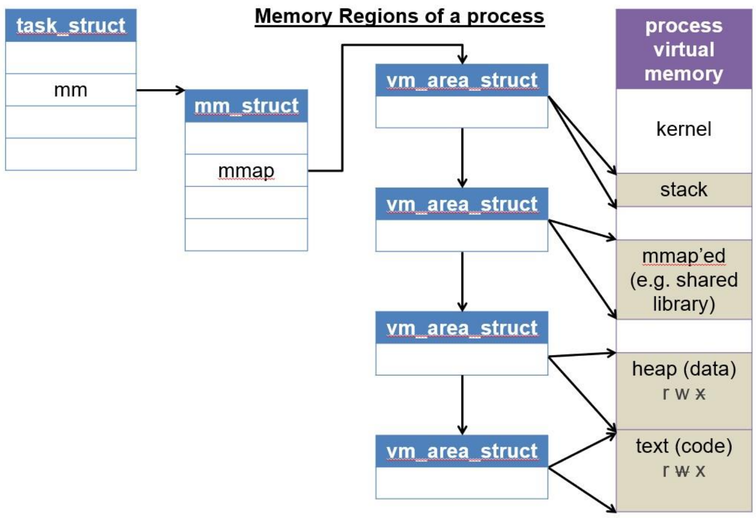
Libc refers to any standard library for the C programming language that complies with the ANSI C standard. Those libraries are so integrated within operating systems, that they are often even considered to be a part of them. The application programming interface (API) of *libc* is declared in several header files, such as *assert.h, stdlib.h, stdio.h*, etc.

Memory Management in this context refers to the unit of functions within *libc* that provides functions that **manage** dynamic memory. The four most common functions in the C programming language are `malloc()`, `free()`, `calloc()` and `realloc()`,which are defined in the *"stdlib.h"* interface.

Note: Different operating systems and compilers use different *libc* implementations. In Linux, for example, we use the GNU C Library (also referred to as *glibc*).

About memory regions

As you've learned, every process In Linux has multiple memory regions, which are maintained and managed by the OS. Memory regions within a process vary from one another in size and access permissions but are all, except the kernel region, governed by the same struct – the **`vm_area_struct`**. The memory regions of a process look something like this:



NOTE: This illustration is imprecise: there may be multiple mmap'ed regions and not necessarily a single region. In fact, each `mmap()` is served with a unique region.

To manage dynamic memory, Linux uses two memory regions – the **heap** and the **memory mapped regions**.

About the "heap" and the "data" segment

There is certain confusion about the definition of data segment. Some define it as an independent segment (figure 2), while others tend to put the Data, BSS and Heap segments all together into one big segment, **also** called the 'Data' segment (figure 3).

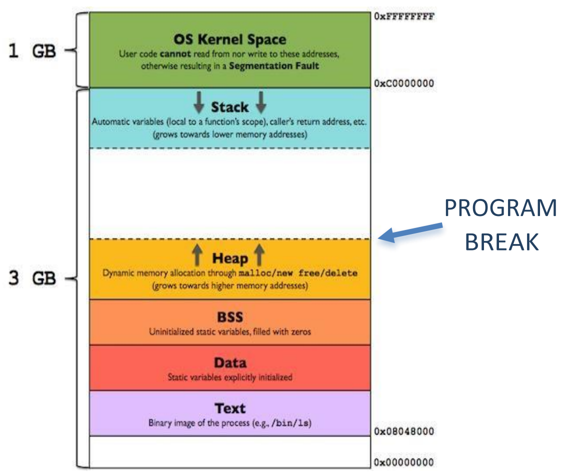


Figure 2 – a more detailed illustration of memory regions (Looking at 32bit system for ease of illustration regions)

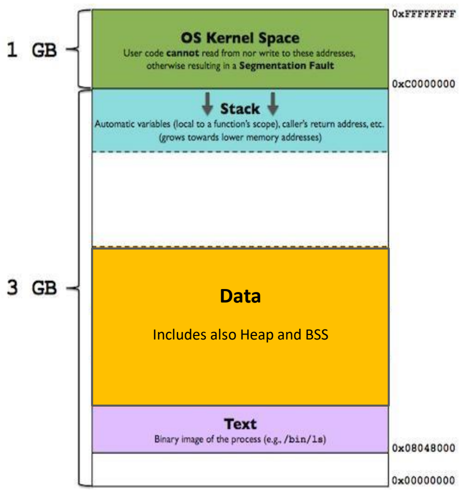


Figure 2 – a simplistic illustration of memory regions (Looking at 32bit system for ease of illustration regions)

There is no right or wrong answer here, as it primarily depends on the context of the conversation. We only depict those differences because you might come across them while reading on the subject online.

NOTE: In this homework and throughout the course we <u>will differentiate</u> between the Heap, BSS and the Data segments, and will use **figure 2** to discuss memory regions.

The *heap* area is defined by a component called the ***program break***, which is defined to be "the end of the heap segment" (refer to figure 2). This program break can be manipulated using **`sbrk()`**, a library function based on the **`brk()`** system call (you should **only** use **`sbrk()`**).

Memory mapped regions, on the other hand, are controlled by the **`mmap()`** system call. In <u>parts 1 & 2</u> of the homework, we will use the *heap* in our memory management units. In <u>part 3</u>, we will incorporate *memory mapped regions* as well. **DO NOT** use memory mapped regions in parts 1 & 2 of the homework.

NOTE: `malloc()` and friend functions are not system calls. They use system calls such as **`mmap`**`()` and **`sbrk`**`()` in their implementation, as you will in this homework.

To allocate dynamic space during runtime, we want to manipulate the program break to create space in the heap. **`sbrk()`**, which is declared in **`<unistd.h>`** is a perfect fit for the job -

`void* sbrk(intptr_t increment):`

- Increases or decreases the current program break by *increment* bytes.
- Note: `intptr_t` is like `long int`.
- Calling `sbrk(0)` is used to get the current program break.
- Return value:
    1. On <u>Success</u> - **previous program break**
    2. On <u>Failure</u> – **(void*)(-1)**, e.g. system out of memory/target address is bad/process out of heap memory.

Part 1 – Naïve Malloc

In the previous discussion, you were provided with enough tools and information for you to begin working on your memory management unit. For this part, you are required to implement a naïve (simple) implementation of malloc. Open a new file, call it **malloc_1.cpp**, and implement the following function:

```
void* smalloc(size_t size)
```

- Tries to allocate '*size*' bytes.
- Return value:
  i. Success –a pointer to the first allocated byte within the allocated block.
  ii. Failure –
    a. If '*size*' is 0 returns NULL.
    b. If '*size*' is more than 10^8, return NULL.
    c. If sbrk fails, return NULL.

Notes:

- `size_t` is a typedef to unsigned int in 32-bit architectures, and to unsigned long long in 64-bit architectures. This means that trying to insert a negative value will result in compiler warning.
- You do not need to implement `free()`, `calloc()` or `realloc()` for this section.

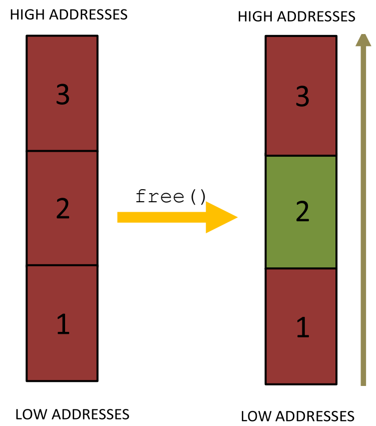
Discussion: Before proceeding, try discussing the current implementation with your partner. What's wrong with it? What's missing? Are we handling fragmentation? What would you do differently?

Part 2 – Basic Malloc

You've probably noticed that in the previous part you did not implement `free()`. However, to implement `free()`, we must understand what "freeing" means and how it works.

A few questions arise when thinking how to add support for `free()` :

1. **How do we know the size of the allocated space that was sent to free?**
   - We can allow the user to send the size of the block size to `free()` with the pointer. That, however, will not be optimal for the user.
2. **How could we mark a space that was just allocated as free?**
   - Ideally, we can adjust the heap with `sbrk()` and remove the allocated space. In other words, we would move the program break back to its location before the allocation.
   - But what if 3 consecutive allocations occurred, and the middle allocation is freed?
     In this situation, we can't just change the program break, as reducing the heap space will cause the top allocation to disappear as well, although it was not explicitly freed.
3. **Maybe we should simply reuse previously freed memory sectors? How would we do it?**



TIP: Before proceeding, challenge yourself by thinking how you can make your current implementation less wasteful (in memory management terms). How can you provide support for your own `free()` in the prior naïve implementation? **Think about the above questions before proceeding**.

Proposed Solution

In this part (part 2), you'll implement our proposed solution, which is a simplified version of the universal implementation. Below are the answers to the questions:

1. **How do we know the size of the allocated space that was sent to free?**
   - On each allocation, do allocate the required memory, but before it – append a **metadata** structure to the block. This means your **total allocation** would be the **requested** size + the **meta-data structure** size. The meta-data will contain an unsigned integer that will save the size of the **effective allocation** (i.e. the requested size).

2. **How could we mark a space that was just allocated as free?**
   - Add a Boolean to the meta-data structure – "is_free".

3. **How can we easily look-up and reuse previously freed memory sectors?**
   - We can save a **global** pointer to a list that will contain all the data sectors described before. We can use this list to search for freed spaces upon allocation requests, instead of increasing the program break again and enlarging the heap unnecessarily.
     The global pointer to the list will point to the first metadata structure (see metadata figure).

Conclusion: to support your Basic Malloc unit, you will need to define a struct/class that will be attached to every allocation you make. It will contain meta-data for each allocation, which is for our own use only and the user shouldn't be aware of its existence. Below is an example of what the heap of a process could look like this after 3 consecutive allocations, alongside a meta-data structure you could use in your implementation:

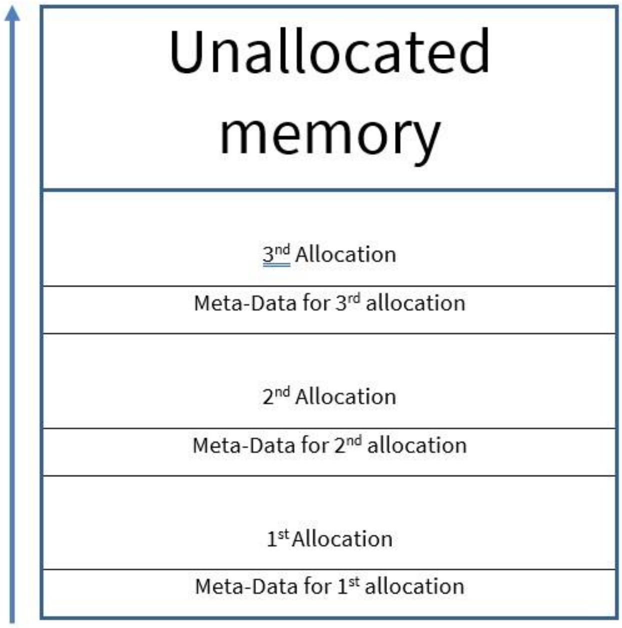


Figure 4 – possible heap of a process after 3 allocations

```
struct MallocMetadata {
    size_t size;
    bool is_free;
    MallocMetadata* next;
    MallocMetadata* prev;
};
```

Figure 5 – suggested meta-data structure, the global pointer to the list will point to the first metadata structure

Above discussion provided you with enough background information to improve your first memory management unit. Open a new file, call it **malloc_2.cpp** and in it implement the following functions:

1. **`void* smalloc(size_t size):`**
   - Searches for a free block with at least *'size'* bytes or allocates (**`sbrk()`**) one if none are found.
   - Return value:
     i. <u>Success</u> – returns pointer to the first byte in the allocated block (excluding the meta-data of course)
     ii. <u>Failure</u> –
        a. If size is 0 returns NULL.
        b. If *'size'* is more than 10^8, return NULL.
        c. If sbrk fails in allocating the needed space, return NULL.

2. **`void* scalloc(size_t num, size_t size):`**
   - Searches for a free block of at least *'num'* elements, each *'size'* bytes that are all set to 0 or allocates if none are found. In other words, find/allocate *size * num* bytes and set all bytes to 0.
   - Return value:
     i. <u>Success</u> - returns pointer to the first byte in the allocated block.
     ii. <u>Failure</u> –
        a. If size or num is 0 returns NULL.
        b. If *'size * num'* is more than 10^8, return NULL.
        c. If sbrk fails in allocating the needed space, return NULL.

3. `void sfree(void* p)`:

- Releases the usage of the block that starts with the pointer *'p'*.
- If *'p' is* NULL or already released, simply returns.
- Presume that all pointers 'p' truly points to the beginning of an allocated block.

4. `void* srealloc(void* oldp, size_t size)`:

- If '*size'* is smaller than or equal to the current block's size, reuses the same block. Otherwise, finds/allocates *'size'* bytes for a new space, copies content of oldp into the new allocated space and frees the oldp.
- Return value:
  - i. Success –
    - a. Returns pointer to the first byte in the (newly) allocated space.
    - b. If *'oldp'* is NULL, allocates space for 'size' bytes and returns a pointer to it.
  - ii. Failure –
    - a. If size is 0 returns NULL.
    - b. If '*size'* if more than 10^8, return NULL.
    - c. If sbrk fails in allocating the needed space, return NULL.
    - d. Do not free *'oldp'* if `srealloc()` fails.

On top of the memory allocation functions that you are defining, you are also required to define the following stats methods.

5. `size_t _num_free_blocks()`:

- Returns the number of allocated blocks in the heap that are currently free.

6. `size_t _num_free_bytes()`:

- Returns the number of **bytes** in all allocated blocks in the heap that are currently free, excluding the bytes used by the meta-data structs.

7. `size_t _num_allocated_blocks()`:

- Returns the overall (**free and used**) number of allocated blocks in the heap.

8. `size_t _num_allocated_bytes()`:

- Returns the overall number (**free and used**) of allocated **bytes** in the heap, excluding the bytes used by the meta-data structs.

9. `size_t _num_meta_data_bytes();`

- Returns the overall number of meta-data bytes currently in the heap.

10. `size_t _size_meta_data()`:

- Returns the number of bytes of a single meta-data structure in your system.

Important Notes:

1. Note that once **size** field in the metadata is set for a block in this section in the metadata, it's not going to change.
2. You should always search for empty blocks in an **ascending manner**. This means if there are two free (and large enough) pre-allocated blocks at 0x1000 and at 0x2000, you should

choose the block that starts at 0x1000. **Hint:** you should **maintain a sorted list** of all the allocations in the system, as described in the proposed solution. You can use large freed blocks for small allocations. This might cause fragmentation but **ignore it for now.**

3. An initial underline in function names means "hidden" or "private" functions in programmer lingo - these are not meant to be called directly by the user. We will use these in our testing, and so should you.
4. Wrong usage of `sfree()` and `srealloc()` (e.g. sending bad pointers) is not your responsibility, it is the library user's problem. Therefore, such action is undefined and there's no need to check for it. In other words, assume that the pointers sent to these functions are legal pointers that point to the first allocated byte, the same ones that are returned by the allocation functions.
5. In this part we will not look at optimizations other than reusing pre-allocated areas. If you come up with optimization ideas, keep them up for the next parts.
6. You should use [`std::memmove`]() for copying data in `srealloc()`.
   a. Self-reading: read about 'std::memmove' and [`std::memcopy`](), what is the difference? Could you have used 'std::memcopy' instead? Why not?
7. You should use [`std::memset`]() for setting values to 0 in your `scalloc()`.
8. You are **NOT PERMITTED** to use **STL** data structures for this part (e.g. std::vector or std::list). Use only primitive arrays or linked lists that you implemented by yourselves.
9. A "block" in the context above means **both** the meta-data structure and the usable memory next to it.
10. You should not count un-allocated space that's not been added to the heap by `sbrk()`.
11. You are not required to "narrow down" the heap (e.g. use `sbrk()` with a negative value).
12. If your algorithm chooses a large block (e.g. 1000 bytes) for a small allocation (e.g. 10 bytes), you should mark the entire block as used. This means that if the system had "X free bytes" before such allocation, it should have "$X - 1000$" free bytes after the allocation.
    This should be reflected in your `_num_free_bytes()` function.
13. You should not perform any alignments in this part.

Part 3 – Better Malloc

Our current implementation has a few **fragmentation** issues. Below are some which you might have noticed while working on the previous section (with their solutions). In this section you will work on solutions for some of those issues.
Open a new file, call it **malloc_3.cpp**, copy the content from malloc_2.cpp into it, and in it implement the following changes:

- **Challenge 0** (Memory utilization):
  As mentioned before, searching for an empty block in ascending order will cause internal fragmentation. To mitigate this problem, we shall allocate the smallest block possible that fits the requested memory, that way the internal fragmentation caused by this allocation will be as small as possible.
  Solution: change your current implementation, such that you maintain a **sorted** list (by size in ascending order, and if sizes are equal, then sort by the memory address) of all the free memory regions, such that allocations will use the 'tightest' fit possible i.e. the memory region with the minimal size that can fit the requested allocation.

- **Challenge 1** (Memory utilization):
  If we reuse freed memory sectors with bigger sizes than required, we'll be wasting memory (internal fragmentation).
  Solution: Implement a function that `smalloc()` will use, such that if a pre-allocated block is reused and is ***large enough***, the function will cut the block into two smaller blocks with two separate meta-data structs. One will serve the current allocation, and another will remain unused for later (marked free and added to the list).
  Definition of "large enough": After splitting, the remaining block (the one that is not used) has at least **128** bytes of free memory, **excluding** the size of your meta-data structure.
  Note: Once again, you are not requested to find the "best" free block for this section, but the first block that satisfies the allocation defined above.

- **Challenge 2** (Memory utilization):
  Many allocations and de-allocations might cause two **adjacent** blocks to be free, but separate.
  Solution: Implement a function that `sfree()` will use, such that if one adjacent block (next or previous) was free, the function will automatically combine both free blocks (the current one and the adjacent one) into one large free block. On the corner case where both the next and previous blocks are free, you should combine all 3 of them into one large block.

- **Challenge 3** (Memory utilization):
  Define the "Wilderness" chunk as the topmost **allocated** chunk. Let's presume this chunk is free, and all others are full. It is possible that the new allocation requested is bigger than the wilderness block, thus requiring us to call `sbrk()` once more – but now, it is easier to simply enlarge the wilderness block, saving us an addition of a meta-data structure.
  Solution: Change your current implementation, such that if:
    1. A new request has arrived, and no free memory chunk was found big enough.
    2. And the wilderness chunk is free.

  Then enlarge the wilderness chunk enough to store the new request.

- **Challenge 4** (Large allocations):
  Recall from our first discussion that modern dynamic memory managers not only use `sbrk()` but also `mmap()`. This process helps reduce the negative effects of memory fragmentation when large blocks of memory are freed but locked by smaller, more recently allocated blocks lying between them and the end of the allocated space. In this case, had the block been allocated with `sbrk()`, it would have probably remained unused by the system for some time (or at least most of it).
  Solution: Change your current implementation, by looking up how you can use `mmap()` and `munmap()` instead of `sbrk()` for your memory allocation unit. Use this **only** for allocations that require **128kb space or more (128*1024 B).**

- **Challenge 5** (Safety & Security):
  Consider the following case – a **buffer overflow** happens in the heap memory area (either on accident or on purpose), and this overflow overwrites the metadata bytes of some allocation with arbitrary junk (or worse). Think – which problems can happen if we access this overwritten metadata?
  Solution: We can detect (but not prevent) heap overflows using "cookies" – **32bit** integers that are placed in the metadata of each allocation. If an overflow happens, the cookie value will change and we can detect this before accessing the allocation's metadata by comparing the cookie value with the expected cookie value.
  You are required to add cookies to the allocations' metadata.
  Note that cookie values should be randomized – otherwise they could be maliciously overwritten with that same constant value to avoid overwrite detection. You can choose a global random value for all the cookies used in the same process.
  Change your current implementation, such that before every metadata access, you should check if the relevant metadata cookie has changed. In case of overwrite detection, you should immediately call **exit(0xdeadbeef)**, as the process memory is corrupted and it cannot continue (not recommended in practice).
  Things to consider –
    1. When should you choose the random value?
    2. Where should the cookie be placed in the metadata? (most buffer overflows happen from lower addresses to higher addresses)

Note: You are not requested to "narrow" down the heap anywhere in this section. The only exception for allowing free memory to go back to the system is in challenge 4, when using `munmap()`.

Note: As opposed to the previous section, the 'size' field in the metadata for blocks here changes.

Notes about `srealloc()`:

`srealloc()` requires some complicated edge-case treatment now. Use the following guidelines:

1. If `srealloc()` is called on a block and you find that this block and one of or both neighboring blocks are large enough to contain the request, merge and use them. Prioritize as follows:
    a. Try to reuse the current block without any merging.
    b. Try to merge with the adjacent block with the <u>lower address</u>.
        - If the block is the wilderness chunk, enlarge it after merging if needed.
    c. If the block is the wilderness chunk, enlarge it.
    d. Try to merge with the adjacent block with the <u>higher address</u>.
    e. Try to merge all those <u>three adjacent blocks</u> together.
    f. If the wilderness chunk is the adjacent block with the <u>higher address</u>:

i. Try to merge with the lower and upper blocks (such as in e), and enlarge the wilderness block as needed.

ii. Try to merge only with higher address (the wilderness chunk), and enlarge it as needed.

g. Try to find a different block that's large enough to contain the request (don't forget that you need to free the current block, therefore you should, if possible, merge it with neighboring blocks before proceeding).

h. Allocate a new block with `sbrk()`.

2. After the process described in the previous section, if one of the options 'a' to 'd' worked, and the unused section of the block is **large enough**, split the block (according to the instructions in challenge 1)!
3. You can assume that we will not test cases where we will reallocate an mmap() allocated block to be resized to a block (excluding the meta-data) that's less than 128kb.
4. You can assume that we will not test cases where we will reallocate a normally allocated block to be resized to a block (excluding the meta-data) that's more than 128kb.
5. When srealloc() is called on an mmaped block, you are never to re-use previous blocks, meaning that a new block must be allocated (unless old_size==new_size).

Notes about `mmap()` **:**

1. It is recommended to have another list for `mmap()` allocated blocks, separate from the list of other allocations.
2. To find whether the block was allocated with `mmap()` or regularly, you can either add a new field to the meta-data, or simply check if the 'size' field is greater than 128kb or not.
3. Remember to add support for your debug functions (function 5-10). Note that functions 5-6 should not consider `munmapp()`'ed areas as free.

Part 4 (OPTIONAL) – A (More) Efficient Implementation

There are many other optimizations that could be added to the system. The following changes, for example, could improve the performance of your memory allocation unit:

- **Challenge 6** (TLB efficiency):
  As we have seen in class, the TLB plays a large role in accelerating translations of memory addresses, therefore optimizing it can lead to significant performance improvements. when allocating some memory block, we will often use multiple entries of that block sequentially, e.g., consider a block allocated for some class, we often use more than one variable in close temporal proximity to each other. some blocks are large enough that the different entries span different memory pages\frames, and thus the TLB becomes ineffective in such cases, to counter this, one can use HugePages, so that the TLB coverage can improve. deciding when to do this is an engineering trade-off, for the sake of this homework we shall implement a simple threshold solution.
  Proposed Solution: for any allocation originally allocated by 'smalloc' (as opposed to 'scalloc', if the allocation request (not including the meta-data) is equal-to or larger than $4MB$, allocate a hugepage.
  If the allocation is originally done by means of 'scalloc', only do this if the size of one block is larger than $2MB$.
  Otherwise, allocate memory as you normally would have (according to part3's instructions).
  Hint 1: you might have to change the meta-data structure.
  Hint 2: you should be able to implement this using the memory allocation functions we learned in class (sbrk, mmap).
- **Challenge 7** (Memory allocation performance):
  Memory Alignment - recall that the **load** & **store** operations work by the granularity of a **memory word** (in 64-bit architectures – 64 bits).
  When a CPU calls for an unaligned memory access, more CPU cycles are required than a call for an aligned memory access (i.e. more load/store operations are needed). Aligned access could also increase cache hits in L1/L2.
  Solution: Change your current implementation, such that each request for new memory is aligned. Because we're working with 64-bit machines, we must align every request for a memory access to a **multiplication of 8** (explained below).
  You should make sure that both the meta-data and the data provided to the user reside between aligned (multiplication of 8) addresses.

  More explanation: As we know, each address stores one byte (8 bits) of information. Since we work with 64-bit systems, each load/store operation loads/stores 64-bits (information from 8 consecutive addresses) together. This limitation makes the system work faster, thus forcibly loading or storing information from or to unaligned memory requires more CPU cycles.
  NOTE: This might 'waste' several bytes for each allocation (internal fragmentation), but overall, it's negligible in typical allocation sequences.
  NOTE: Make sure that the size of your meta-data structure is a multiple of 8 Bytes.

If you wish to implement this part, open a new file, name it **malloc_4.cpp**, copy the current content from malloc_3.cpp and implement the suggested change.

NOTE 1: This is an optional section, so little to no support will be provided on the Piazza Forum. Try finding answers to your questions alone before submitting them to the staff.

NOTE 2: we know that we have not taught you how to allocate hugepages in class, finding this out is a part of this section, we recommend you try finding out alone to simulate real-life situations where you will need to learn how to use different functions by reading the documentation on your own.

NOTE 3: When scalloc() is called, the total size should be aligned to a multiple of 8, but you should **NOT** align each element on its own.

Advice and Grading Policy

1. The maximum grade for the wet part of this homework is **110**, where a grade out of 100 will be given to your work on parts 1, 2 and 3, and up to **additional 10** points will be given for a successful submission of part 4.

2. Each part will be graded and tested individually. This means you have overall 2 options:
   a. Submit `malloc_1.cpp, malloc_2.cpp, malloc_3.cpp`
   b. Submit `malloc_1.cpp, malloc_2.cpp, malloc_3.cpp, malloc_4.cpp`

3. You CANNOT #include malloc_x.cpp in malloc_y.cpp (when $x, y \in \{1,2,3,4\}$), even if you must rewrite similar lines of code in the two files. Each part to its own.

4. You should write, compile and test **ALL your code on your virtual machines (that was installed in HW0)**. You should not test it on any of the department servers (e.g. CSL3). The reason for this is that the servers contain different versions of *glibc*, allocation functions and system calls and therefore what works on them may not work on your virtual machine.

5. You must implement this exercise in C++. You may use C, but your code will be compiled and tested with `g++` (C++ compiler).

Questions & Answers / Piazza

- The Q&A for the exercise will take place at a public forum Piazza **only**. Please **DO NOT** send questions to the private email addresses of any of the TAs.
- Critical updates about the HW will be published in **pinned** notes in the piazza forum. These notes are mandatory, and it is your responsibility to stay updated.
- Read **previous Q&A** carefully before asking the question; repeated questions will probably go without answers.
- Be **polite**, remember that course staff does this as a service for the students.
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you must discuss such a matter, please come to the reception hours.
- When posting questions regarding **hw4**, put them in the **hw4** folder

Late Days

- Please **DO NOT** send postponement requests to the TA responsible for this assignment.

Only the **TA in charge** can authorize postponements. In case you need a postponement, please fill out the attached form : https://forms.office.com/r/NLBU1nykjE

Submission

- You are to electronically submit a single zip file named **XXX_YYY.zip** where XXX and YYY are the IDs of the participating students.
- The zip should contain all source files you wrote **with no subdirectories of any kind.**
- Make sure to also add to the zip a file named **submitters.txt** which includes the ID, name and email of the participating students. The following format should be used:

```
Linus Torvalds linus@gmail.com 234567890
Ken Thompson ken@belllabs.com 345678901
```

Important Note: Make the outlined zip structure exactly. In particular, the zip should contain only the following files (no subdirectories):

```
zipfile -+
|
+- malloc_1.cpp
|
+- malloc_2.cpp
|
+- malloc_3.cpp
|
+- malloc_4.cpp (for potential extra credit)
|
+- submitters.txt
```

Important Note: when you submit, **retain your confirmation code and a copy of the file(s)**, in case of technical failure. Your confirmation code is **the only valid proof** that you submitted your assignment when you did.

Important Note: Make sure you keep an eye on the Piazza. If we see that a change is required, we will give you the heads up there.

important Note: make sure that your code compiles without additional header files (it is common for some student tests to use a .h file, make sure that your code doesn't need one to be compiled properly).

Important Note: Your code is assumed to have functions defined in accordance with the next declarations.

```
void* smalloc(size_t size);
void* scalloc(size_t num, size_t size);
void sfree(void* p);
void* srealloc(void* oldp, size_t size);
size_t _num_free_blocks();
size_t _num_free_bytes();
size_t _num_allocated_blocks();
size_t _num_allocated_bytes();
size_t _num_meta_data_bytes();
size_t _size_meta_data();
```

We hope you managed to fight back and restore what was stolen from us by the EE students! Next semester – we'll be stealing Ohm's law from the EE department.