TECHNION
Israel Institute
of Technology

Computer
Science
Department

Operating Systems – 234123

Homework Exercise 4 – Wet

Teaching Assistant in charge:

Sami Zreik

Assignment Subjects & Relevant Course material:

Virtual Memory & Memory Management

Recitations 10-11, Lecture 7

Abstract and Assignment Objectives

After a long wait of 8 clock cycles, the Night King has finally arrived at Linux, and he's taken the *libc* dynamic memory allocation unit to the Army of the Dead! Your immediate help is required. You must implement a new memory allocation unit and save Linux from the terrors of the cold, memory-less night!

On a more serious note - in this homework you will implement a simple memory allocation library, which will eventually 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 could grant you extra credit. The homework slowly increases in difficulty, with each part relying on the understanding and completion of the previous parts. Therefore, you must follow this homework step-by-step.

To make things even more interesting, we will show you how you can load your memory allocation code as a shared library. Loading your library as a shared library will enable you to use above functions instead of their corresponding *libc* implementation. This means that you will be able to run any code or program that uses memory allocation, such as the `'ls'` program in bash, and force it to use your library instead of *libc*

This homework will hopefully provide you with knowledge and skills in **virtual memory**, **memory regions**, **dynamic loading** and ***libc***. To better understand the requirements of this exercise, we suggest first thoroughly reviewing Recitations 10 & 11.

Introduction

About libc and Memory Management

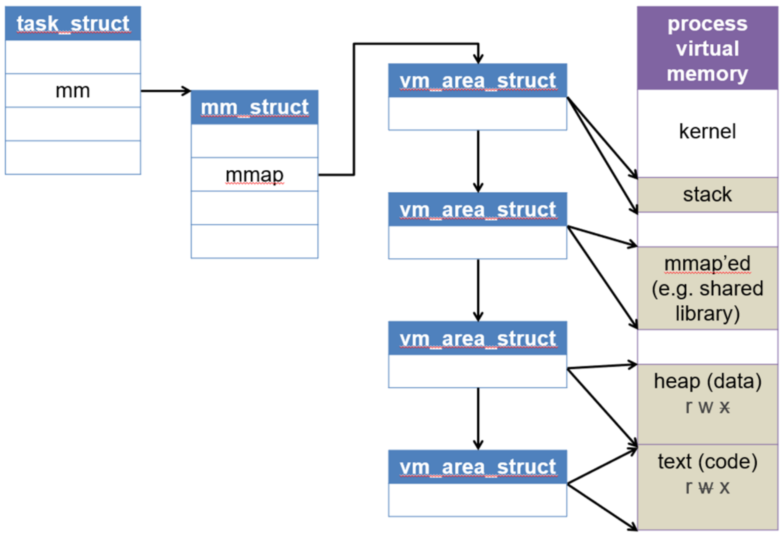
Libc refers to any standard library for the C programming language, that complies with the ANSI C standard. Those libraries are so integrated within operating systems, that they are often even considered to be a part of them. The application programming interface (API) of *libc* is declared in several 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 memory. The four most common memory management functions in C are `malloc()`, `free()`, `calloc()` and `realloc()`, and are defined the *"stdlib.h"* interface.

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

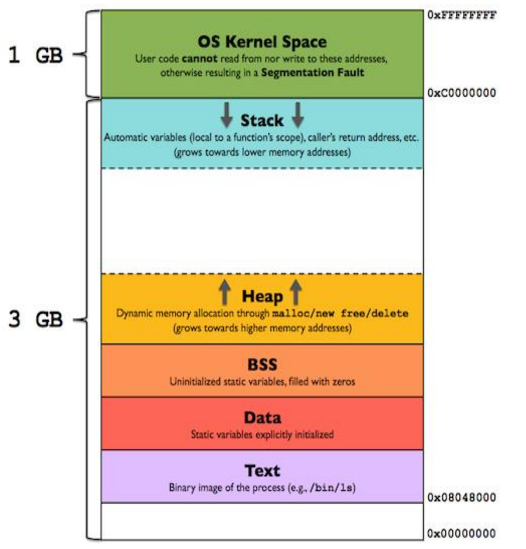
About memory regions

In Linux, we use the **heap** memory region to manage dynamic memory. The *heap* is one of several memory regions in a process (such as the stack and the memory mapped region), as you have seen in Recitation 11. Memory regions within a process vary from one another in size and access permissions, but are all (except the kernel region) governed by a the same struct – the **`vm_area_struct`**. If we were to examine the `vm_area_structs` of a single process, it should look something like this:



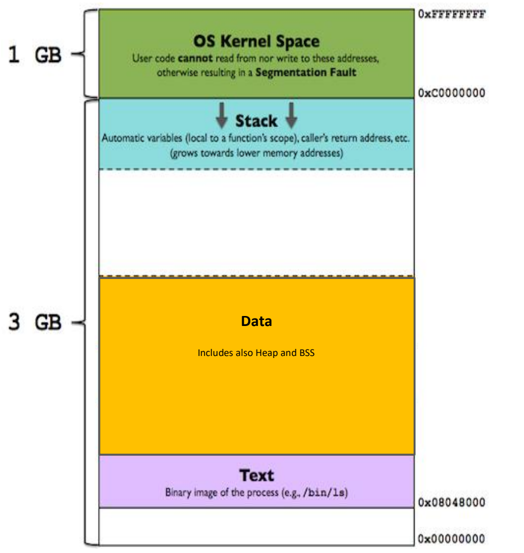
About the “data” segment

There is certain confusion about the definition of data segment. Some say it is an independent segment (right picture), while others tend to put the Data, BSS and Heap segments all together into one big segment, **also** called the ‘Data’ segment. You may read about the reasons for this online, but it is not important for the exercise. Frankly, there is no right answer here, as it primarily depends on the context of the conversation.



1 GB
3 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

*Figure SEQ Figure * ARABIC 3 – a more detailed*



1 GB
3 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Data
Includes also Heap and BSS
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

*Figure SEQ Figure * ARABIC 2 – a simplistic illustration of*

It is important to depict those differences, as you might come across them while reading about the subject on the internet.

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

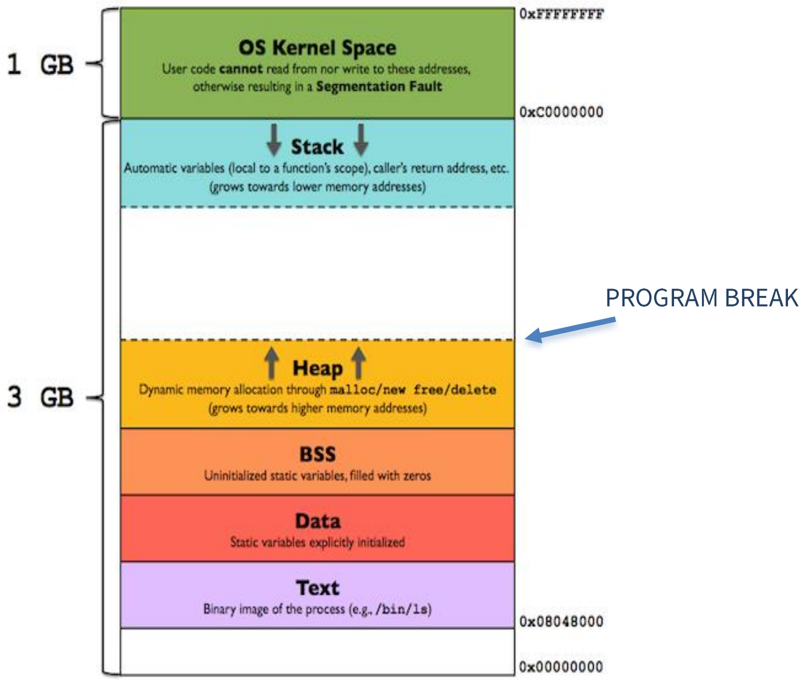
In accordance to this definition, we will focus primarily on the **heap**. The heap segment is where dynamic memory is usually managed. In fact, if a program does not use any form of dynamic allocation (e.g., malloc/new), there will be no area defined as a heap.

There are two ways in which we can manipulate the heap – either by changing something called a ***program break*** using certain system calls (discussed in the next section), or by using the `mmap()` system call to create anonymous mappings (covered partially in the lectures and recitations). Advanced implementations of malloc often use both.

NOTE: `malloc()` and cousins are NOT system calls. They do, however, use system calls in their implementations, such as `mmap()`, `brk()` and `sbrk()` (defined shortly).

About the program break

The **program break** is defined to be "the end of the heap segment" (sources may define it as the end of the data segment, refer to the discussion above). If we were to add the program break to the previous figure, it would look something like this:



1 GB
3 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000
PROGRAM BREAK

One would want to manipulate the program break in order to allocate dynamic space during runtime. Linux provides two system calls that can change the program break, both declared in **<unistd.h>** –

1. `int brk(void *addr)`
    - Sets the program break to the value specified by *addr*.
    - Return value:
        1. On <u>Success</u> – **(0)**
        2. On <u>Failure</u> – **(-1)**, e.g. system out of memory/*addr* is bad/process out of heap memory.

2. `void* sbrk(intptr_t increment)`
    - Increases the current program break by *increment* bytes.
    - Note: `intptr_t` is like `void*`.
    - 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.

NOTE: Consider using `sbrk()` in your implementation.

A small demonstration:

Let us demonstrate how the `malloc()` in *glibc* in our virtual machines uses both *brk()* and *mmap()*. Fire up your **VMWares** and open a new .cpp file. In it insert the code:

```
#include <iostream>
#include <cstdlib>
#define SIZE 10000 //WE WILL CHANGE THIS SOON
using namespace std;

int main() {
    cout << "BEFORE MALLOC" << endl;
    malloc(SIZE);
    cout << "AFTER MALLOC" << endl;
    return 0;
}
```

Now we will use a Linux utility called "**strace**", which you have seen before (R9). This utility traces and prints the system calls for a program in Linux. In this demonstration we will see what system calls are used when we call `malloc()`. Many system calls are called from the moment we begin our tracing, therefore, to detect the system calls called only by malloc, we will trap malloc between two prints ("BEFORE MALLOC" and "AFTER MALLOC"). Each print should use the system call "`write()`", therefore, the system calls called by malloc will be the one between two writes.

Now do the following:

1. Compile above code with – `g++ ./[FILENAME].cpp -o tst`
2. Run compiled code with the command – `strace ./tst`

For the current *SIZE* , you should notice that there is a call to **`brk()`** between the two writes.

3. Change the *SIZE* to 100,000.
4. Compile
5. Run with strace.

This time, between the two writes you should see a call to **`mmap()`.**

Conclusions: memory allocators (specifically the one in the *libc* of our Linux distribution) can use both the **program break** and **memory mapping**. Try running this with various sizes on the CSL3 server as well and see what happens.

NOTE: In this homework, we will create a custom version of the above `malloc`. Our version **only** manipulates the program break, no matter the size of the input (this means, we will not utilize the `mmap()` system call, as in the standard `malloc()` implementation). This is true for all sections but the optional section.

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 implement a naïve (simple) implementation of malloc. Open a new file, call it **malloc_1.cpp**, include *stdlib*, and implement the following function:

```
void* malloc(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*' if more than 10^8, return NULL.
     c. If sbrk or brk fail, 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. Insertion of negative value without complying to the warning will cause '*size*' to have an undefined value. You should not worry about this problem.
- You do not need to implement `free()`, `calloc()` or `realloc()` for this section.
- To compile and test this file, please read the two sections about compilation and loading.

Submission: Once you have finished this part, make sure you save it as **malloc_1.cpp**. This **MUST** be submitted alongside the 2 other mandatory parts of this homework.

Conclusion questions for this part:

From now on, consider yourselves maintainers/gate keepers of every virtual memory space, along with the single physical space.

- How should a proper malloc implementation look like? What is wrong with the current?
- As a memory manger, what ideals should you protect? What would a "correctly" handled set of virtual spaces look like? (**Hint**: external/internal fragmentation, handling of disk accesses, OS intervention etc).

Part 2 – Basic Malloc

You've probably noticed that in the previous part you did not implement `free()`. Well let's start with a simple question, what is the meaning of free? We somehow need to provide a mechanism that, given a certain pointer to the beginning of a previously allocated block, will free the block.

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 let the user of malloc send the size to `free()`, but that restricts the user.
2. **How could we mark a space that was just allocated as free?**
   - If the **last** memory block allocated was also the one freed, we can move the program break and return it to its location from before the allocation.
   - What if 3 allocations occur, 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 is not free.
3. **How can we easily look-up and reuse previously freed memory sectors?**

TIP: Before proceeding, challenge yourself by thinking how you can make your current implementation less wasteful (in memory management terms you defined on the last page) and how you can provide support for your very own `free()` to the prior naïve implementation. **Please try solving the above questions**.

Proposed Solution

In this part, you'll implement our proposed solution, which is the universal (but simplistic in our exercise) implementation to solve the above problems. Our answers of the above questions are:

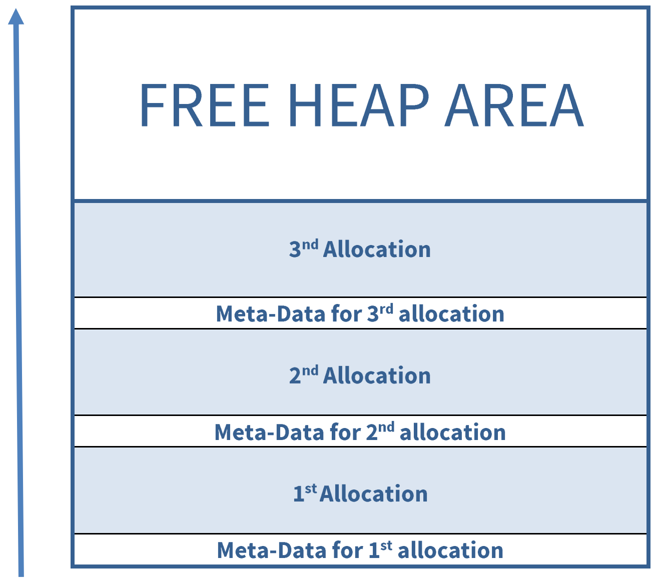
1. **How do we know the size of the allocated space that was sent to free?**
   - On each allocation, allocate the required memory, but before it – append a **meta-data** structure. This means, your **total allocation** is the **requested** size + the meta data structure size. The data will contain an unsigned integer that will save the size of the actual **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 above structure - **is_free**.

3. **How can we easily look-up and reuse previously freed memory sectors?**
   - We can save a **global** 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 enlarge the heap unnecessarily.

So the conclusion is, to support your Basic Malloc unit, you will need to define a struct/class that will be attached to every allocation you make and contain meta-data for each allocation.

After 3 consecutive allocations, your heap could look something like this:


FREE HEAP AREA
3rd Allocation
Meta-Data for 3rd allocation
2nd Allocation
Meta-Data for 2nd allocation
1st Allocation
Meta-Data for 1st allocation

The above discussion should provide you with enough tools and 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* malloc(size_t size):`**
   - Allocates *'size'* bytes.
   - Return value:
     i. <u>Success</u> - returns pointer to the first byte in the allocated block.
     ii. <u>Failure</u> –
        a. If size is 0 returns NULL.
        b. If *'size'* if more than 10^8, return NULL.
        c. If sbrk or brk fail in allocating the needed space, return NULL.

2. **`void* calloc(size_t num, size_t size):`**
   - Allocates a block of *'num'* elements, each *'size'* bytes that are all set to 0. In other words, 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 is 0 returns NULL.
        b. If *'size'* if more than 10^8, return NULL.
        c. If sbrk or brk fail in allocating the needed space, return NULL.

3. **`void free(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 point to the beginning of an allocated block.

4. **`void* realloc(void* oldp, size_t size):`**
   - Allocates *'size'* bytes for a new space, copies content of old space into new space and frees the old space.
   - **Hint:** This function **should not** allocate unnecessary space if it doesn't need to.
   - Return value:
     i. <u>Success</u> –
        a. Returns pointer to the first byte in the (newly) allocated space.
        b. If *'oldp'* is NULL, allocate space for 'size' bytes and return a pointer to it.
     ii. <u>Failure</u> –
        a. If size is 0 returns NULL.
        b. If *'size'* if more than 10^8, return NULL.
        c. If sbrk or brk fail in allocating the needed space, return NULL.
        d. Do not free *'oldp'* if `realloc()` fails.

Note: An initial underline in function names usually 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 you should too.

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.

Notes:

1. Wrong usage of `free()` and `realloc()` (A.K.A 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.
2. This part 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.
3. You should use `std::memcpy` for copying data in `realloc()`.
4. You should use `std::memset` for setting values to 0 in your `calloc()`.
5. You CANNOT use the **STL** library for this part besides for the functions described above (remember that std::vector and the like use memory allocation themselves). Please use simple data structures like primitive arrays or linked lists you implemented by yourselves.
6. When asked to return the number of blocks, you should refer to a single block as a combination of **both** the meta-data structure and the usable memory attached to it.
7. You should not count un-allocated space that's not been added to the heap by `brk()/sbrk()`.
8. You can use larger freed blocks for smaller allocations. Use the first large enough block you find. This might cause fragmentation but ignore it for now.

Part 3 – Better Malloc

Our current implementation has a lot of **fragmentation** and **performance** issues. Below are issues which you might have noticed this in the previous section (with their solutions). 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:

1. **Problem 1** (Memory utilization):
   If we reuse freed memory sectors that have a bigger size than required, we'll be wasting memory (internal fragmentation).
   Solution: Implement a function that `malloc()` 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.
   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.

2. **Problem 2** (Memory utilization):
   Many allocations and de-allocations might cause two **adjacent** blocks to be free, but separate.
   Solution: Implement a function that `free()` will use, such that if two adjacent blocks have both become free, the function will automatically combine both free blocks into one large free block.

3. **Problem 3** (Memory utilization):
   Define the "Wilderness" 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. The wilderness chunk is free.

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

4. **Problem 4** (Memory allocation performance):
   Our current implementation faces many performance issues. A crucial one we should address is memory alignment. Remember that **load** & **store** work by the granularity of a **memory word** (in 32-bit architectures – a 32-bit line in memory).
   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 32-bit processors (on VMWare), we must align every request for a memory access to a **multiplication of 4**. You should

make sure that both the meta-data and the data provided to the user reside between aligned (multiplication of 4) addresses.
NOTE: This might waste several bytes for each allocation, but its overall negligible.

Part 4 (OPTIONAL) – Optimized Malloc

Until now we've worked on several time and space optimizations, but there's much more that can be done. There are two very important additions that are implemented in advanced memory allocators today:

1. (Up to **5** bonus points) Recall that certain memory allocation units use both `brk()/sbrk()` and `mmap()` and not only `brk()/sbrk()`. Look up why using `mmap()` is useful, and how you can use it (a system call we've avoided this far), in your current memory allocation unit. Add `mmap()` support for any call that requests **256kb space or more**.
2. (Up to **10** bonus points) Recall that to search for pre-allocated free chunks we currently use utilizes a **single** list that contains meta-data for chunks of different size. Think of ways to make this faster. Look up how Doug Lea Malloc (**dlmalloc** for short) uses bins for fast look-up and implement the scheme.

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 one or both suggested additions above.

NOTE: This is a freestyle section; you must come up with and read the details for your optimization. Little to no support will be provided on the Piazza forum for this part, as it is optional and is meant to challenge you.

NOTE: You cannot remove the additions you made in parts 2 and 3. You can change the way they work.

Testing

There are two ways to test this your code. The first one is the normal one - you can include your .cpp files in the tests and run them (without including <stdlib.h> anywhere, so there wouldn't be confusion with *libc* malloc). This, however, can restrict you as including other libraries that contain declaration of *stdlib* might include *stdlib*. To overcome this, you can temporarily change the name of your memory allocation functions (e.g. from `malloc()` to `my_malloc()`).

We're going to present a new way, in which you can "overload" the existing *libc* functions. Make sure your implementations have *stdlib* included and then:

Compilation:

Compile your malloc_x.cpp implementation, where $x \in \{1,2,3,4\}$ with the following line:

```
g++ -fPIC -shared ./malloc_x.cpp -o malloc_x.so
```

The flags -fPIC and -shared tell the compiler that the compiled output is a **shared** library. Feel free to use any other debugging flags as well. Shared libraries are usually compiled into a *.so file and not *.o (short for Shared Object).

Loading your libraries:

Congratulations, you've just created your (probably) first couple of dynamic libraries! To load your custom libraries, and let other processes use them, we will use the **LD_PRELOAD** trick.

Explanation:

The macro **LD_PRELOAD**, defined in your shells, tells the Linux's dynamic loader which libraries to load first, before loading the usual shared libraries (e.g. *libc*).

You can view which libraries your code will load by using the shell command '`ldd` [SOME EXECUTABLE]'. Try compiling a simple code and running '`ldd`' on it.

Now, using LD_PRELOAD, you will be able to load your own custom libraries, and tell your compiler "listen! look for certain functions through my custom library first, and only then begin looking for them in other libraries".

For example, if you use LD_PRELOAD on your `malloc.so`, you will tell the code you're running to use your implementation of `malloc()`, `free()`, `realloc()` and `calloc()`, instead of those implemented in *libc*.

TIP: Please read this to understand this subject more thoroughly.

Using LD_PRELOAD (there’s 2 ways):

1. The first method is to tell a SPECIFIC program to run with your malloc.so:
   ‘LD_PRELOAD=[PATH_TO_MALLOC_X.SO] [SPECIFIC_PROGRAM_OR_TEST]’
2. The second method will all programs to run using your malloc.so:
   ‘export LD_PRELOAD=[PATH_TO_MALLOC_X.SO]’
   To revert this change so that all programs use *libc* allocation instead:
   ‘unset LD_PRELOAD’

Now all you need to do is load your malloc.so as a shared library for a test program and run it. Remember that all programs that use allocation can now use your implementation, this includes bash commands. If the library works fine you should be able to run the ‘ls’ command. Try playing around by running other shell commands you know. You could probably pull out your MATAM or Data Structure course tests and run them with your malloc.so. You might even be able to run the g++ compiler, which uses malloc() if your implementation is good enough (we will **not test** this).

Advice and Grading Policy

1. The maximum grade for the wet part of this homework is **115**, where a grade out of 100 will be given to your work on parts 1, 2 and 3, and up to **additional 15** points will be given for a good 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 should NOT 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 (VMWare)**. You should not test it on any of the department servers (e.g. CSL3). You could run it there and see if it works (for fun). 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).
6. Note that things like printing and opening files or pipes inside malloc() could be problematic. Therefore, for debugging inside your allocation functions you can use assertions.
7. Note that to test functions that begin with underscore (using the LD_PRELOAD technique), you'll need to have a header file for their declarations. Simply open a new .h file, put the declarations of functions that begin in underscore there and add the header file in your test files. You do not need to submit that header file; we will use our own during testing.

Questions & Answers

- 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.

Several guidelines for using the forum:

- 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://goo.gl/forms/Im4tFXNLq4uznBzM2

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 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: We know this is the first time such a homework is posted; therefore, changes may occur after the homework has been release. Make sure you keep an eye on the Piazza, as if we see that a change is required, we will give you the heads up there.

We hope you will manage to save Linux & libc from the terrors of the dark night!