Assignment 2: Allocator

For this assignment you will implement your own heap allocator similar to `ptmalloc2`, `jemalloc`, `tcmalloc`, and many others. These allocators are the underlying code of `malloc`. Heap allocators request chunks of memory from the operating system and place several (small) object inside these. Using the `free` call these memory objects can be freed up again, allowing for reuse by future `malloc` calls. Important performance considerations of a heap allocator include being fast, but also to reduce memory fragmentation.

Your allocator must implement its own `malloc`, `free` and `realloc` functions, and may not use the standard library versions of these functions. Your allocator may only use the `brk(2)` and `sbrk(2)` functions, which ask the kernel for more heap space. [1]

Description of the functions to implement

You should implement `mymalloc`, `mycalloc`, `myfree`, and `myrealloc` as described in the Linux man pages. [2] These functions should behave exactly as specified by their man-page description, although you can ignore their *Notes* section as these are implementation-specific. Your allocations (i.e., the pointer returned by `mymalloc`) should be aligned to `sizeof(long)` bytes.

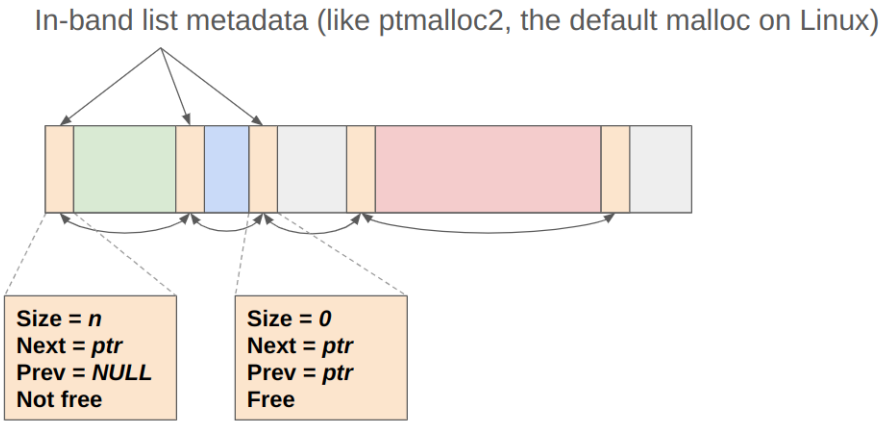
Your allocator may not place any restrictions on the maximum amount of memory supported or the maximum number of objects allocated. For example, your allocator should scale regardless of whether the maximum `brk` size is 64KB or 1TB.

Design

Multiple designs are possible for metadata management. For example, the Linux slab allocator manages metadata through a bitmap while `ptmalloc2` uses an **in-band freelists** design. We suggest the latter for this assignment.

But why do we need metadata? We need to store information about the state of each chunk of memory that the allocator returns to the user program. Metadata refers to this extra information.

The layout of an allocator based on in-band lists may look like the following:



Memory areas in orange are metadata. Adjacent to the metadata lies the data. When an application requests a memory buffer, the allocator reserves memory for both the metadata and the data. Note that when serving a memory request the allocator should return a pointer to the data since metadata management is transparent to the user application; metadata are not intended to be use from the user program.

In C, metadata may be represented by a struct as the following:

```
struct obj_metadata {
  size_t size;
  struct obj_metadata *next;
  struct obj_metadata *prev;
  int is_free;
};
```

The field `size` specified the number of bytes of the associaed data. `next` and `prev` point to the next and previous allocated chunk. `is_free` specifies if the chunk is actually in use by the user application or it has been released; It is possible to reuse a freed chunk to serve a subsequent request of the same size without having to reserve additional memory via `brk`.

As an example, your design may involve a list of free chunks (usually referred to as a "freelist"). The allocator could serve memory requests through the freelist rather than allocating more heap with `brk`.

In your allocator you should try to minimize the amount of memory dedicated to the metadata. Indeed, the resources used to manage metadata are not available to store user data and result in memory overhead!

Further requirement is that your allocator should scale with the maximum `brk` size (i.e.your medatata must be abl to grow). The following code shows an incorrect design.

```
include <stdint.h>

#define MAX_OBJ 1024
struct obj_metadata objs[MAX_OBJ];

void *malloc(size_t size) {
```

Metadata are statically defined and therefore there is a limit to the number of objects you can allocate. A correct implementation would look like this:

```
#include <stdint.h>

void *heap_start;
void *freelist;

void *malloc(size_t size) {
```

`heap_start` points to the first chunk. In this way you do not have an artificial limit to the number of objects.

Grading

This assignment is individual; you are not allowed to work in teams. Submissions should be made to the submission system before the deadline. Multiple submissions are encouraged to evaluate your submission on our system. Our system may differ from your local system (e.g., compiler version); points are only given for features that work on our system.

Your grade will be 1 if you did not submit your work on time, has an invalid format, or has errors during compilation.

If your submission is valid (on time, in correct format and compiles), your grade starts from 0, and the following tests determine your grade (in no particular order):

- +1.0pt if your make a valid submission that compiles.
- +1.0pt if your `malloc` returns a valid pointer to a new heap object. **Required**
- +0.5pt if your `calloc` returns a valid new heap pointer to zero-initialized memory.

- +2.0pt if a region of memory can be reused after freeing it with `free`. **Required**
- +1.0pt if `realloc` behaves as described on its man-page and only allocates a new object when needed.
- +1.0pt if your allocator batches `brk` calls, i.e., it does not need to request memory from the kernel for every allocation.
- +2.0pt if your amortized overhead per allocation is on average 8 bytes or less.
- +0.5pt if your allocator tries to optimize for locality (reuse recently freed memory).
- +1.0pt if your allocator gives back memory to the kernel (using `brk`) when a large portion of the allocated memory has been freed up.
- +2.0pt if your allocation functions work correctly without the `my` prefix too (see *Notes* below).
- -2.0pt if your allocator cannot scale with the maximum `brk` size.
- -1.0pt if `gcc -Wall -Wextra` reports warnings when compiling your code.
- -1.0pt if your source files are not neatly indented or formatted.

If you do not implement an item marked with **Required** you cannot obtain any further points. This means you need to implement at least a simple allocator that can do malloc and free with reuse.

The grade will be maximized at 10, so you do not need to implement all features to get a top grade. Some features might be mutually exclusive with each other, depending on your allocator design.

Note: Your allocator will be evaluated largely automatically. This means features only get a positive grade if they work perfectly, and there will be no half grade for "effort".

Evaluation environment

For setting up a local development environment, refer to the setup document. In short, on Linux you should install `build-essential python3`, on Windows you should use WSL2, and on macOS you should use Docker.

To test your implementation, the file `test_framework/tests.c` contains a number of (automated) test cases that evaluate the different aspects of your allocator. It can be invoked manually via `./test <test name>`. Running `make check` (or `make docker-check`) will run all test cases, and additionally check your work for other errors that would lead to deducted points during grading.

Additionally you should test your work on our server. Remember to try this as often as your like, as your local environment may be different than ours. Points are only awarded based on what works on our server. The final submission before the deadline is used for grading.

Attempts to exploit, bypass or cheat the infrastructure and automated grading system will result in a 1 for this assignment.

Notes

- While you can edit the test framework locally to debug issues, you should not modify `alloc.h` or any file in `test_framework/`. During submission and grading any modifications made to these files will be thrown away.
- If you add definitions for `malloc` etc. to your `alloc.c`, you should also keep the original set of `my` functions for grading. Sample code that makes enables these functions is included in the skeleton `alloc.c`.
- If you have added support for replacing the system allocator (i.e., by adding non `my` prefixed functions) you can use your allocator for any existing program on your system. You can do this by prefixing any command with `LD_PRELOAD=/path/to/libmyalloc.so`. For example, `LD_PRELOAD=./libmyalloc.so ls` will run `ls` with your allocator.

- Calling your functions `malloc` instead of `mymalloc` not only redirects all calls inside **your** code to your `malloc`, but will also cause all internal libc calls to go to your allocator instead of the built-in libc `malloc`. Many libc functions, such as `printf`, internally make calls to `malloc`, and as such using `printf` inside your allocation code would cause an infinite loop. Therefore we prefix our allocator functions with `my` in this assignment.

Notes on debugging

- You cannot use ASan to compile your code. ASan would intercept your malloc calls and won't work properly.
- You will most likely get segfaults, framework assertions, unexpected bahavior.

Some tips:

- Run `./test` with `-v` to see what the test is trying to do.
- Use debugging prints and GDB
- Dump your freelist before/after every allocation

Example of using gdb:

```
$ gdb -q --args ./test -v malloc-simple
Reading symbols from ./test...done.

(gdb) run
Starting program: /home/koen/git/vu-os-alloc/framework/test -v malloc-simple

[test_framework/checked_alloc.c:136] Allocating 1 bytes
Program received signal SIGSEGV, Segmentation fault.
chunk_is_free (chunk=chunk@entry=0x0) at alloc.c:81
81      return chunk->_is_free;

(gdb)
```

You can now print any variable (local or global) in your C code:

```
(gdb) print chunk
$1 = (struct chunk *) 0x0
```

and display the stack trace:

```
(gdb) backtrace
#0  chunk_is_free (chunk=chunk@entry=0x0) at alloc.c:81
#1  0x00007ffff79cf000 in chunk_data (chunk=0x0) at alloc.c:85
#2  0x00007ffff79cf486 in mymalloc (size=8, size@entry=1) at alloc.c:274
#3  0x0000555555556eef in _checked_alloc (nmemb=1, size=size@entry=1, allocator=allocator@entry=ALLOC_DEFAULT) at test_framework/checked_alloc.c:61
#4  0x00005555555573de in checked_alloc (size=size@entry=1) at test_framework/checked_alloc.c:141
#5  0x000055555555574a in test_malloc_simple () at test_framework/tests.c:38
#6  0x0000555555555643 in run_test (name=0x7fffffffe29b "malloc-simple") at test_framework/main.c:69
#7  main (argc=<optimized out>, argv=<optimized out>) at test_framework/main.c:120
```

---

1 Normal heap allocators may also use `mmap` to request memory from the kernel. For this assignment you should only use `brk` (or `sbrk`).

2 https://linux.die.net/man/3/malloc