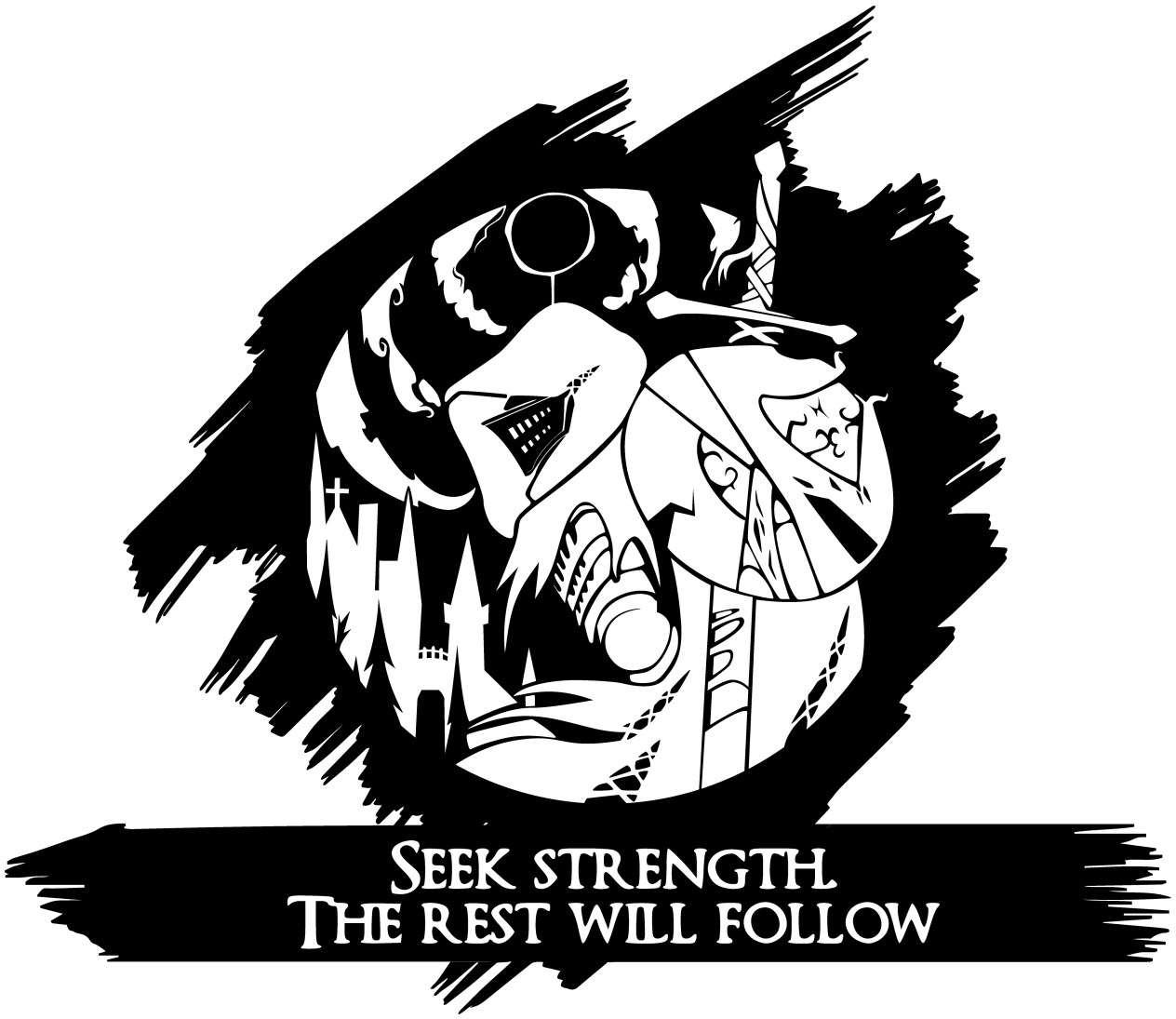
Malloc — Subject

version **#bfdaecd01cbf44dfbd7800b940343997d9ad7d73**



Assistants C/Unix 2025 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.forge.epita.fr

Obligations

*Obligations are **fundamental** rules shared by all subjects. They are non-negotiable and to not apply them means to face sanctions. Therefore, do not hesitate to ask for explanations if you do not understand one of these rules.*

Obligation #0: Cheating, as well as sharing source code, tests, test tools or coding-style correction tools is **strictly forbidden** and penalized by not being graded, being flagged as a cheater and reported to the academic staff.

Obligation #1: The coding-style needs to be respected at all times.

Obligation #2: If you do not submit your work before the deadline, it will not be graded.

Obligation #3: Anything that is not **explicitly** allowed is **disallowed**.

Obligation #4: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #5: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #6: Your submission repository must be **clean**. Except for special cases, which (if any) are **explicitely** mentioned in this document, an *unclean* repository may contain:

- binary files;[1]
- files with inappropriate privileges;
- forbidden files: `*~`, `*.swp`, `*.o`, `*.a`, `*.so`, `*.class`, `*.log`, `*.core`, etc.;
- a file tree that does not follow our specifications.

Advice

- If the slightest project-related problem arise, you can get in touch with the assistants.

  Post to the dedicated **newsgroup** (with the appropriate **tag**) for questions about this document, or send a **ticket** to **<assistants@tickets.assistants.epita.fr>** otherwise.
- Do **not** wait for the last minute to start your project!
- In examples, `42sh$` is our prompt: use it as a reference point.
- Read the *whole* subject.

[1] If an executable file is required, please provide its sources **only**. We will compile it ourselves.

1 Exercise

File Tree

```
malloc/
├── Makefile  (to submit)
├── src/
│   ├── *  (to submit)
│   └── malloc.c  (to submit)
└── tests/
    ├── *  (to submit)
    ├── corruptionproof
    ├── memoryfootprint
    └── speed
```

Compilation : Your code must compile with the following flags

- -std=c99 -pedantic -Werror -Wall -Wextra -Wvla -fvisibility=hidden -fPIC

Makefile

- library: Produce the libmalloc.so library
- check: Runs your testsuite
- clean: Deletes everything produced by make

Authorized functions : You are only allowed to use the following functions

- mmap(2)
- munmap(2)
- mremap(2)
- sysconf(3)

Authorized headers : You are only allowed to use the functions defined in the following headers

- string.h
- sys/mman.h
- pthread.h
- stdint.h
- err.h
- errno.h
- assert.h

- stddef.h

2 Introduction

This project is not as hard as you might think. It is split into many levels in order to simplify implementation.

Core features consist in a basic version of `malloc(3)`, `free(3)`, `calloc(3)` and `realloc(3)`, while advanced features consist in improved implementations. You are supposed to follow the behavior of each function as described in their man page.

Take time to read the whole subject before starting and think about your implementation.

Tips

- Thread safety is nice and surprisingly not hard to implement.
- Metadata structures matter for efficiency; think about speed and used space.
- Algorithm is also important.

Be careful!

Before trying to optimize your malloc implementation, try to have one that works well first!

Remember:

> *"Premature optimization is the root of all evil."*
>
> ---Donald Knuth

We give you a few binaries to help you for the levels they are named after. Remember: we gave them to you, you do not have to give them back to us.

We also provide a `Makefile` and a *C* file with the prototypes of the functions, made so that it produces a shared library with the correct symbols exported. You are free to modify them, they will not be overwritten.

For this project, you are exceptionally allowed to use five explicit casts. Use them wisely.

According to the coding style, you are allowed to use a global variable per translation unit.

3 Debugging

In this project you have to build a shared library. Thus, the debugging is not the same as for previous projects. Using the `debug` rule of the given Makefile is enough to build the library correctly. When the `libmalloc.so` is built, it has to be linked with the desired binary or program.

3.1 Linking

`LD_LIBRARY_PATH` is an environment variable which defines the path where the shared library should be found. Here is an example of how to use it:

```
42sh$ gcc main.c -L. -lmalloc -o main
42sh$ LD_LIBRARY_PATH=. ./main
```

`LD_PRELOAD` is also an environment variable but defines directly the library to preload instead of specifying the path. Here is an example of how to use it:

```
42sh$ LD_PRELOAD=./libmalloc.so ls
```

3.2 GDB

If you want to debug using gdb you must set environment variables inside gdb as follows:

```
42sh$ gdb ./main
Reading symbols from ./main...
(gdb) set env LD_LIBRARY_PATH=.
(gdb) start
```

```
42sh$ gdb ls
Reading symbols from ls...
(No debugging symbols found in ls)
(gdb) set exec-wrapper env 'LD_PRELOAD=./libmalloc.so'
(gdb) start
```

4 Core features

You have to generate a shared library named `libmalloc.so` at the root directory of your submission. It must **only** export the following symbols:

```c
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t number, size_t size);
void *realloc(void *ptr, size_t size);
```

Note that these functions do not include the usual "`my_`" prefix. That is because we want to **override** the functions provided by the standard library.

Tips

In order to test whether your shared library exports any other symbols than `malloc`, `free`, `calloc` and `realloc`, use the following command:

```
42sh$ nm -C -D libmalloc.so
[...]
000000000000136a T calloc
[...]
0000000000000f87 T free
[...]
000000000000104a T malloc
[...]
00000000000010c3 T realloc
```

The symbols marked with `T` are symbols related to the `.text` section, which is the code section.

Notice how none of the auxiliary functions appear in the dynamic exported symbol table.

Be careful!

You **must** export these 4 symbols, even if the functions are not implemented yet.

Tips

It might be a good idea to create an interface between the `malloc`, `free`, ... functions by making them call usual `my_` prefixed functions, in order to be able to test your code while still using libc's malloc for printf (without having to preload your library).

Be careful!

Be aware that the malloc API is not limited to `malloc`, `free`, `calloc` and `realloc`. There are some additional functions such as `aligned_alloc`, `malloc_usable_size`, `memalign`, `posix_memalign`, `xmalloc`. Some programs, such as `vlc` or `cat` might use them, which might cause issues if you did not implement your own. You are allowed to implement them and export them if you wish, but they will not be evaluated.

Any symbol exported other than the 4 required and these 4 optional will be sanctioned.

4.1 malloc(3)

Implement the following function:

```
void *malloc(size_t size);
```

A successful call to `malloc` must return a pointer to a *valid* memory area, and this area must be *aligned* on `long double`. The caller should be able to write over the *whole* allocated block. If the memory cannot be allocated, `malloc` must return `NULL`.

You are expected to handle *several* pages at this stage. They might not be neighbors.

Tips

We strongly recommend that you write tests to verify your malloc's behavior. Afterwards, you can test the following commands:

- `factor 20 30 40 50 60 70 80 90`

- `cat Makefile` (but be mindful of the note on malloc API functions)
- `ip a`

Of course, this is not enough and will not guarantee that you will pass all our tests.

Be careful!

Beware, implementing a `malloc` that uses `mmap` at each call will result in a 0

4.2 free(3)

Implement the following function:

```
void free(void *ptr);
```

You need to implement `free` so that it releases the required memory space. Releasing a `NULL` pointer must not cause any error, and should not do *anything* either. Attempting to release an invalid pointer (i.e. a pointer that was not returned by your `{c,m,re}alloc`) may lead to an undefined behavior, and therefore will not be tested.

Tips

In order to test your `free(3)`, you can test the same commands listed for `malloc(3)`, since they all release the memory in the end.

Be careful!

Be careful, a memory space that has just been freed up must be reusable immediately by your library! Moreover a mapped page that no longer have any used memory blocks must be unmapped.

4.3 calloc(3)

Implement the following function:

```
void *calloc(size_t number, size_t size);
```

Please refer to its *man page* to fully understand its behavior.

Tips

In this function you will have to check for an overflow. We strongly recommend using the gcc builtin function. The online documentation is available here, read it and choose the right function to use.

4.4 realloc(3)

Implement the following function:

```
void *realloc(void *ptr, size_t size);
```

Once again, its *man page* will be an invaluable resource. For now, you can implement `realloc` as you want, but *be careful*: an upcoming level will require a wise implementation.

Tips

After testing, you can try running the following commands:

- `ls`
- `ls -la`
- `tar -cf malloc.tar libmalloc.so`
- `find /`
- `tree /`
- `od libmalloc.so`
- `git status`
- `less Makefile`
- `clang --help`

4.5 Recompile self

Now that you have implemented a basic version of the `malloc` library, you can try to use it to recompile your project. It basically consists in compiling your project, "preloading" your library and compiling it again.

5 Advanced features

5.1 Thread-safe

Now that you can compile your own project and test it at the same time, you need something new... You need a *thread-safe* `malloc`!

In a multi-threaded environment, your `malloc` can be called simultaneously from different threads.

Without thread safety, if a modification of your `malloc`'s internal data structures is happening in thread #1, and thread #2 starts modifying them before thread #1 finishes, your data structures will become corrupted, and your `malloc` will depend on false and corrupted data.

In order to solve this, you must make sure that while any operation is done on your internal data structures, these are modified by a single thread at a time.

For this purpose, we allow you to use some functions of the `pthread` library. You will need *mutexes* or *spinlocks* (or any other locking facility provided by the `pthread` library) to avoid data inconsistencies.

The functions in `pthread.h` are authorized, including the following:

- `pthread_mutex_init(3)`
- `pthread_mutex_lock(3)`
- `pthread_mutex_trylock(3)`
- `pthread_mutex_unlock(3)`
- `pthread_mutex_destroy(3)`

To test this level, we will spawn multiple threads doing numerous calls to `malloc`, `free` and `realloc`. The program must run smoothly, and must not make any invalid memory access.

If you want to have performance at the same time as thread safety, you have to think of a better allocation strategy for multi-threaded programs, that is also well suited for single-threaded programs.

Tips

We strongly recommend you to write tests to verify your malloc behavior. After testing, you can try running the following commands:

- `gimp`
- `chromium-browser`
- `vlc` (but be mindful of the note on malloc API functions)

Of course, this is not enough and will not guarantee that you will pass all our tests.

5.2 Extended realloc(3)

Use a smart implementation for the `realloc` function; that is, extend the current memory space whenever possible rather than relocating it. If the space is extended, the pointer returned by `realloc` is the first argument of the function.

Your `realloc` must support realloc-intensive binaries: be prepared to handle every case.

5.3 Speed

To pass this level, you must execute `speed` correctly. This test ensures that it ends in a fixed number of seconds, so your `malloc` implementation needs to be fast enough.

The binary does not take any arguments, your library just has to be preloaded correctly. It returns 0 if it is fast enough, 1 otherwise.

```
42sh$ ls
libmalloc.so  speed
42sh$ LD_PRELOAD=./libmalloc.so ./speed; echo $?
0
```

5.4 Corruption proof

As you may have realized, the behavior of `malloc` relies entirely on the metadata. If those are corrupted (i.e. do not contain what they are supposed to), the behavior of malloc may be undefined. If the metadata is located next to the allocated block, it is simple for the user to corrupt the metadata since they only have to write before or after the allocated block.

To succeed, your implementation must be able to run correctly even if we try to override the metadata.

The given binary `corruptionproof` will help you test this feature. It can return these values:

- 0 Good
- 1 `malloc` returned `NULL`
- 2 `malloc` returned `NULL` after corruption

```
42sh$ ls
libmalloc.so  corruptionproof
42sh$ LD_PRELOAD=./libmalloc.so ./corruptionproof; echo $?
0
```

Tips

This is an additional feature, it is not mandatory to get the maximum grade. You can see it as a challenge and it should not break your implementation.

5.5 Memory footprint

This level is separated into two tests. The first one check the fragmentation and the second one check the allocation ratio between metadata and real data.

To help you, we gave you a `memoryfootprint` binary. This binary will do many allocations, but your `mmap` calls are limited. To succeed, you will have to optimize the memory internally needed by `malloc` in order to reduce your memory footprint.

The usage of `memoryfootprint` is the following: you have to give it a single argument, which is the allocation size it will use. It will then return the percentage of occupation of the allocated data: the higher the better.

```
42sh$ ls
libmalloc.so  memoryfootprint
42sh$ LD_PRELOAD=./libmalloc.so ./memoryfootprint 512; echo $?
50
42sh$ LD_PRELOAD=./libmalloc.so ./memoryfootprint 32768; echo $?
87
```

5.6 Real-world allocator

Now you must be able to “preload” your library, and then launch and use various memory-greedy applications.

Keep in mind that some applications, like `firefox`, do not use the `malloc` from the `libc`, but they implement their own. You will not be able to override their implementation using `LD_PRELOAD`, so you will not be able to test your malloc with those applications.

6 Documentation

As you may have guessed there are a lot of ways to implement your allocator.

But there are also a lot of research papers about memory allocation.

These papers may be interesting to you if you decide to go on and implement a more complex algorithm:

- [PN77] Buddy Systems - James L. Peterson, Theodore A. Norman
- [BON94] The Slab Allocator: An Object-Caching Kernel Memory Allocator - Jeff Bonwick
- [JW98] The Memory Fragmentation Problem: Solved? - Mark S. Johnstone, Paul R. Wilson
- Memory Allocation with Lazy Fits - Yoo C. Chung, Soo-Mook Moon
- Fast Efficient Fixed-Size Memory Pool: No Loops and No Overhead - Benjamin Kenwright

Tips

As these papers tend to be very technical and the presented algorithms may be difficult to implement, it may be a good idea to have a working algorithm before going on with these papers.

Seek strength. The rest will follow.