Malloc - Apping — Subject

version **#9ea0dcc-dirty**

Assistants C/Unix 2020 `<assistants@tickets.assistants.epita.fr>`

Copyright

Contents

*https://intra.assistants.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: If you do not submit your work before the deadline, it will not be graded.

Obligation #2: 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 (see **??**).

Obligation #3: All your files must be encoded in ASCII or UTF-8 without BOM.

Obligation #4: When examples demonstrate the use of an output format, you must follow it scrupulously.

Obligation #5: Your code must compile with the flags ”-pedantic -Werror -Wall -Wextra -std=c99 -fno-builtin”.

Obligation #6: If a function, a command or a library is not *explicitly* authorized, it is **forbidden**. If you use a forbidden symbol, your work will not be graded.

Advice

- ▷ Read the *whole* subject.
- ▷ 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 **<acu@tickets.assistants.epita.fr>** otherwise.
- ▷ In examples, `42sh$` is our prompt: use it as a reference point.
- ▷ Do **not** wait for the last minute to start your project!

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

Project data

Instructors:

- Lionel Auroux `<lionel@assistants.epita.fr>`

Dedicated newsgroup: `assistants.apprentis.projets with [MLL]`

Members per team: 1

1 Malloc

Files to submit:

- `./Makefile`
- `./README`
- `./src/*`
- `./tests/*`
- `./TODO`

Provided files:

- `./memoryfootprint`
- `./corruptionproof`
- `./speed.py`
- `./call_trace.c`
- `./Makefile`
- `./malloc.c`

Coding Style Changes:

- The number of allowed C cast is 5.

Main function: None

Makefile: Your makefile should define at least the following targets:

- **libmalloc.so**: Produces the libmalloc.so shared library
- **check**: Run your testsuite
- **clean**: Delete everything produced by make

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

- **C Standard Library (3)**: sysconf

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

- `string.h`
- `sys/mman.h`
- `pthread.h`
- `stdint.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

- Metadata structures matter for efficiency; think about speed and used space
- Algorithm is also important
- Thread safety is nice
- *premature optimization is the root of all evil* – Donald Knuth

We give you a few binaries and a python script. They are your objectives for the levels they are named after. Remember we gave them to you, so you don't need 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 of course free to modify them, they won't be overwritten.

For this project, you are allowed to use 5 explicit casts without triggering coding style errors. Use them wisely.

Be careful!

Beware, implementing a `malloc` that only calls `mmap` each time will be considered as cheating.

3 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
[...]
00000000000013d4 T _fini
0000000000000f87 T free
[...]
00000000000006d8 T _init
[...]
000000000000104a T malloc
[...]
00000000000010c3 T realloc
```

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

Of course, `_fini` and `_init` are automatically exported by your compiler, and you shouldn't bother getting rid of them, as they will be ignored by us.

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

Be careful!

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

Tips

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`. Some programs, such as `vlc` might use them, which might cause issues if you didn't implement your own. You are allowed to implement them and export them if you wish, but they won't be evaluated.

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

3.1 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 `size_t`. 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

After unit testing, you can test the following commands:

- *factor 20 30 40 50 60 70 80 90*
- *cat Makefile*

- *ip a*

Of course, this won't guarantee that you will pass all our tests.

3.2 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.

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.

3.3 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 an overflow. We strongly recommend to use the gcc builtin function. Here is an example to check an addition:

```
#include <stdio.h>

void add_check_overflow(int a, int b)
{
    int res = 0;
    if (__builtin_sadd_overflow(a, b, &res))
        printf("Overflow detected with operands %u and %u.\n", a, b);
    else
        printf("%d + %d = %d\n", a, b, res);
}

int main(void)
{
    add_check_overflow(25, 42);
    add_check_overflow(1234567890, 2048);
    return 0;
}
```

```
42sh$ gcc -Wall -Wextra -Werror -std=c99 -pedantic -o main main.c
42sh$ ./main
25 + 42 = 67
Overflow detected with operands 1234567890 and 1234567890.
```

You can check online documentation for more information.

3.4 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` the way you want, but **be careful**: an upcoming level will require a wise implementation.

Tips

After unit testing, you can test the following commands:

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

4 Advanced features

4.1 Fragmentation

For this level, you have to take **fragmentation** into account. Your `malloc` must be able to return some memory space that was previously released, and that may be located between two used areas.

4.2 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.

4.3 Advanced realloc(3)

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

4.4 Speed

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

4.5 Memory footprint

To pass this level, you need to execute the given binary `memoryfootprint` correctly. 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.

4.6 Recompile self

You will pass this level if you can use your own `malloc` implementation to recompile your project. This will be tested during your defense. It basically consists in compiling your project, “preloading” your library and compiling it again.

4.7 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 to modify them before thread #1 finished, 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 any operations done on your internal data structures 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

After unit testing, you can test the following commands:

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

Of course, this won't guarantee that you will pass all our tests.

4.8 Real-world allocator

To pass this level, you must be able to "preload" your library, and then launch and use various memory-greedy applications. This will be tested during your defense.

Keep in mind that some applications, like `firefox`, don't 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 won't be able to test your malloc with those applications.

4.9 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 he only has 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 to test this feature. It can return these values:

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

Note: 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.

I never really sleep. Got one eye open, always.