

# SCALE FOR PROJECT MALLOC (/PROJECTS/MALLOC)

## Introduction

Please respect the following rules:

- Remain polite, courteous, respectful and constructive throughout the correction process. The well-being of the community depends on it.
- Identify with the person (or the group) graded the eventual dysfunctions of the work. Take the time to discuss and debate the problems you have identified.
- You must consider that there might be some difference in how your peers might have understood the project's instructions and the scope of its functionalities. Always keep an open mind and grade him/her as honestly as possible. The pedagogy is valid only and only if peer-evaluation is conducted seriously.

## Guidelines

You MUST run the requested tests.

Warning: This project is quite complex, the result and it's implementation are subjective. You have to keep in mind the aim of this project:

This project is about implementing a dynamic memory allocation mechanism.

## Attachments

 subject ([https://cdn.intra.42.fr/pdf/pdf/5644/ft\\_malloc.en.pdf](https://cdn.intra.42.fr/pdf/pdf/5644/ft_malloc.en.pdf))

Preview!!!

# Preliminaries

## Preliminary tests

First check the following elements:

- There is something in the git repository
- The author file is valid
- A Makefile is present and has all the requested rules
- No norm errors, Norminette is authoritative.
- No cheating (unauthorized functions...)
- 2 globals are authorised : one to manage the allocations, and one to manage the thread-safe

If an element of this list isn't respected, the grading ends.

Use the appropriate flag. You're allowed to debate some more about the project, but the grading will not be applied.

✓ Yes

✗ No

## Library compilation

First we will check that the compilation of the library does generate the requested files by modifying HOSTTYPE:

```
$> export HOSTTYPE=Testing
$> make re
...
$> ln -s libft_malloc_Testing.so libft_malloc.so
$> ls -l libft_malloc.so
```

```
libft_malloc.so -> libft_malloc_Testing.so $>
```

The Makefile does use HOSTTYPE to define the name of the library (libft\_malloc\_\${HOSTTYPE}.so) and does create a symbolic link libft\_malloc.so pointing towards libft\_malloc\_\${HOSTTYPE}.so ?

If that's not the case, the defense stops.

✓ Yes

✗ No

## Functions export

Check with nm that the library does export the functions malloc, free, realloc and show\_alloc\_mem.

```
$> nm libft_malloc.so
0000000000000000 T _free
0000000000000000 T _malloc
0000000000000000 T _realloc
0000000000000000 T _show_alloc_mem
U _mmap
U _munmap
U _getpagesize
U _write
U dyld_stub_binder
$>
```

The functions exported by the library are marked with a T, the used one with a U (addresses have been replaced by 0, they change from one library to the next, same as the order of the lines).

If the functions are not exported, defense stops.

☒ Yes

☐ No

## Feature's testing

Start by creating a script that will only modify the environment variables while you run a test program. It will be named run.sh, and be executable: `$> cat run.sh #!/bin/sh export DYLD_LIBRARY_PATH=. export DYLD_INSERT_LIBRARIES="libft_malloc.so" export DYLD_FORCE_FLAT_NAMESPACE=1 $@`

### Malloc test

We are first going to make a first test program that does not use malloc, so that we have a base to compare to:

```
$> cat test0.c
#include

int main()
{
    int i; char *addr;

    i = 0;
    while (i < 1024)
    {
```

```
i++;  
}  
return (0);  
}
```

```
$> gcc -o test0 test0.c
```

```
$> /usr/bin/time -l ./test0 0.00
```

```
real 0.00 user 0.00 sys  
491520 maximum resident set size  
0 average shared memory size  
0 average unshared data size  
0 average unshared stack size  
139 page reclaims  
0 page faults  
0 swaps  
0 block input operations  
0 block output operations  
0 messages sent  
0 messages received  
0 signals received  
0 voluntary context switches  
1 involuntary context switches  
$>
```

We will then add a malloc and write in each allocation to make sure that the memory page is allocated in physical memory by MMU. The system will only really allocate the memory of a page if you write in it, so even if we do a bigger mmap than the malloc request it won't modify the "page reclaims".

```
$> cat test1.c
```

```
#include  
  
int main()  
{  
    int i;  
    char *addr;  
  
    i = 0;  
    while (i < 1024)  
    {  
        addr = (char*)malloc(1024);  
        addr[0] = 42;  
        i++;  
    }  
}
```

```
}  
return (0);  
}
```

```
$> gcc -o test1 test1.c  
$> /usr/bin/time -l ./test1  
0.00 real 0.00 user 0.00 sys  
1544192 maximum resident set size  
0 average shared memory size  
0 average unshared data size  
0 average unshared stack size  
396 page reclaims  
0 page faults  
0 swaps  
0 block input operations  
0 block output operations  
0 messages sent  
0 messages received  
0 signals received  
0 voluntary context switches  
1 involuntary context switches  
$>
```

Our test1 program requested 1024 times 1024 bytes, so 1Mbyte. We can therefore check by doing the difference with the test0 program:

- either between the "maximum resident set size" lines, we obtain a little more than 1Mbyte
- or between the page reclaims lines that we will multiply by the value of `getpagesize(3)`

Let's test now both programs with our library:

```
$> ./run.sh /usr/bin/time -l ./test0  
0.01 real 0.00 user 0.00 sys  
708608 maximum resident set size  
0 average shared memory size  
0 average unshared data size  
0 average unshared stack size  
214 page reclaims  
0 page faults  
0 swaps  
0 block input operations  
1 block output operations  
0 messages sent
```

0 messages received  
0 signals received  
0 voluntary context switches  
1 involuntary context switches

```
$>./run.sh /usr/bin/time -l ./test 1
0.00 real 0.00 user 0.00 sys
4902912 maximum resident set size
0 average shared memory size
0 average unshared data size
0 average unshared stack size
1238 page reclaims
0 page faults
0 swaps
0 block input operations
0 block output operations
0 messages sent
0 messages received
0 signals received
0 voluntary context switches
2 involuntary context switches
$>
```

We notice in this example that this malloc has used 1024 pages  
ie: 4MBytes to store 1Mbyte.

Count the number of pages used and adjust the grade as such:

- less than 255 pages, allocated memory is insufficient: 0
- 1023 pages and over, the malloc works but consumes 1 page minimum for each allocation: 1
- between 513 pages and 1022 pages, malloc works but the overhead is too big: 2
- between 313 pages and 512 pages, malloc works but the overhead is very big: 3
- between 273 pages and 312 pages, malloc works but the overhead is big: 4
- between 255 and 272 pages, malloc works and the overhead is fine: 5

The defender is allowed to justify another method of counting allocated pages (using existing debug for example).

**Rate it from 0 (failed) through 5 (excellent)**



### Pre-allocated zones

Check inside the source code that the pre-allocated zones for the different malloc sizes allow to store at least 100 times the maximum size for this type of zone. Check also that the size of the zones is a multiple of `getpagesize()`.

If one of these points is missing, click NO.

☒ Yes

☐ No

### Tests of free

We will simply add a free to our test program:

```
$> cat test2.c
#include

int main()
{
    int i;
    char *addr;

    i = 0;
    while (i < 1024)
    {
        addr = (char*)malloc(1024);
        addr[0] = 42;
        free(addr);
        i++;
    }
    return (0);
}
```

```
$> gcc -o test2 test2.c
```

We will compare the number of "page reclaims" to those in test0 and test1. If there are as many or more "page reclaims" than test1, the free doesn't work.

```
$> ./run.sh /usr/bin/time -l ./test2
```

Does the free function? (less "pages reclaims" than test1)

☒ Yes

☐ No

### Quality of the free function

test 2 has at most 3 more "page reclaims" than test0?

☒ Yes

☐ No

### Realloc test

```
$> cat test3.c
#include
#include

#define M (1024 * 1024)

void print(char *s)
{
    write(1, s, strlen(s));
}

int main()
{
    char *addr1;
    char *addr3;

    addr1 = (char*)malloc(16 * M);
    strcpy(addr1, "Bonjour\n");
    print(addr1);
    addr3 = (char*)realloc(addr1, 128 * M);
    addr3[127 * M] = 42;
    print(addr3);
    return (0);
}

$> gcc -o test3 test3.c
$> ./run.sh ./test3
Bonjour
Bonjour
$>
```

Does it work like in the example?

☒ Yes

☐ No



### Realloc test++

In test3.c, modify the main function's body as such:

```
int main()
{
char *addr1;
char *addr2;
char *addr3;

addr1 = (char*)malloc(16 * M);
strcpy(addr1, "Bonjour\n");
print(addr1);
addr2 = (char*)malloc(16 * M);
addr3 = (char*)realloc(addr1, 128 * M);
addr3[127 * M] = 42; print(addr3);
return (0);
}
```

Does it still work ?

☒ Yes

☐ No

### Gestions des erreurs

Test all the particular cases and errors:

```
$> cat test4.c
#include
#include
#include

void print(char *s)
{
write(1, s, strlen(s));
}

int main()
{
char *addr;

addr = malloc(16);
free(NULL);
free((void *)addr + 5);
if (realloc((void *)addr + 5, 10) == NULL)
print("Bonjour\n");
```

```
}  
$> gcc -o test4 test4.c  
$> ./run.sh ./test4  
Bonjour
```

In case of error, realloc must return NULL. Is "Bonjour" displayed like in the example? If the program reacts badly (segfault or something else), defense stops and you need to select the "crash" flag.

☒ Yes

☐ No

---

### Show\_alloc\_mem test

```
$> cat test5.c  
#include  
  
int main()  
{  
  malloc(1024);  
  malloc(1024 * 32);  
  malloc(1024 * 1024);  
  malloc(1024 * 1024 * 16);  
  malloc(1024 * 1024 * 128);  
  show_alloc_mem();  
  return (0);  
}
```

```
$> gcc -o test5 test5.c -L. -lft_malloc  
$> ./test5
```

Does the display corresponds the topic and the TINY/SMALL/LARGE allocation of the project?

☒ Yes

☐ No

---

## Bonus

---

### Competitive access

The project manages the competitive access of the threads with the support of the pthread library and with mutexes.

Count the applicable cases:

- a mutex prevents multiple threads to simultaneously enter inside the malloc function
- a mutex prevents multiple threads to simultaneously enter inside the free function
- a mutex prevents multiple threads to simultaneously enter inside the realloc function
- a mutex prevents multiple threads to simultaneously enter inside the show\_alloc\_mem function

Rate it from 0 (failed) through 5 (excellent)



### Additional bonuses

If there are more bonuses, grade them here. Bonuses must be 100% functional and a minimum useful. (up to the grader)

Bonus example:

- During a free, the projet "defragments" the free memory while regrouping the available simultaneous blocks.
- Malloc has debugging environnement variables
- A function allows to make an hexadecimal dump of the allocated zones
- A fonction allows to display an history of the memory allocations done.
- ...

S'il y a d'autres bonus, comptez-les ici. Les bonus doivent être 100% fonctionnels et un minimum utiles (à la discrétion du correcteur).

Rate it from 0 (failed) through 5 (excellent)



## Ratings

Don't forget to check the flag corresponding to the defense

✓ Ok

★ Outstanding project

📄 Empty work

📄 Incomplete work

💬 No author file

🧠 Invalid compilation

📖 Norme

📖 Cheat

💥 Crash

🚫 Forbidden function

## Conclusion

Leave a comment on this evaluation

General term of use of the site  
(<https://signin.intra.42.fr/legal/terms/6>)

Privacy policy  
(<https://signin.intra.42.fr/legal/terms/5>)

Legal notices  
(<https://signin.intra.42.fr/legal/terms/3>)

Declaration on the use of cookies  
(<https://signin.intra.42.fr/legal/terms/2>)

Terms of use for video surveillance  
(<https://signin.intra.42.fr/legal/terms/1>)

Terms of use for the site  
(<https://signin.intra.42.fr/legal/terms/4>)