Preliminaries

Preliminary tests

First check the following items :

- There is indeed a rendering (in the git repository)
- Valid author file
- The Makefile is present and has the requested rules
- No fault of norm, the Norminette being authentic
- No cheating (unauthorized functions...)
- 2 globals are allowed: one to manage allocations, and one to manage thread-safe

If an element does not conform to the subject, the notation stops there. You are encouraged to continue discussing the project, but the scale is not applied.

Yes | No

Compilation of the library

We will start by checking that the compilation of the library generates the files requested in the subject, 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 uses HOSTTYPE to define the name of the library (libft_malloc_$HOSSTYPE.so) and creates a symbolic link libft_malloc.so pointing to libft_malloc_$HOSSTYPE.so ?

If this is not the case, the defense stops.

Yes | No

Export of functions

Check with nm that the library exports the malloc, free, realloc and show_alloc_mem functions.

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

The functions exported by the library are marked with a T, those used with a U (the addresses have been replaced by 0, they change from one library to another, as does the order of the lines).

If the functions are not exported, the defense stops.

Functionality tests

Start by making a launch script that only modifies the environment variables for the time it takes to launch a test program.
He will have to call himself 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 going to make a first test program that does not do malloc, in order to have a basis for comparison:

```
$> cat test0.c
#include <stdlib.h>

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

  i = 0;
 while (i < 1024)
  {
   i++;
  }
  return (0);
 }
$> cc -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 ensure that the memory page is properly allocated by physical memory by the MMU.

Sometimes, the system really allocates the memory of a page only when we write from inside, so we write inside just to be sure.

This must allow to have a correct "page reclaims".

```
$> cat test1.c
#include <stdlib. h="">

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

  i = 0;
  while (i < 1024)
  {
   addr = (char*)malloc(1024);
   addr[0] = 42;
   i++;
  }
  return (0);
}
$> cc -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 see this by making the difference with the testO program :
- either between the lines "maximum resident set size", we get a little more than 1 Mbyte
- either between the lines page reclaims that we will multiply by the value of getpagesize(3)

Now let's test both programs with our library:

```
$>./run.sh /usr/bin/time -l ./testO
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 ./test1
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 note in this example that this malloc has used 1024 pages, i.e. 4 MBytes to store 1 MByte.
Suffice to say that it is not really optimized.

Ask the student what are his limits of the TINY and SMALL zones, in order to see if the 1024 byte blocks are TINY, SMALL, or WIDE.
In the case of LARGE, use the claims page difference between test1 and testO. In the other 2 cases, add the size used in the pre-allocated area.

Count the number of pages used and adjust the rating as follows:
- less than 255 pages, the reserved memory is insufficient: 0
- 1024 pages and more, the malloc works but consumes a minimum page at each allocation: 1
- between 512 pages and 1023 pages, the malloc works but the overhead is too large: 2 (1 page for 2 alloc)
- between 342 pages and 511 pages, the malloc works but the overhead is very important: 3 (1 page for 3 alloc)
- between 291 pages and 341 pages, the malloc works but the overhead is important: 4
- between 255 and 290 pages, the malloc works and the overhead is reasonable: 5

Pre-allocated areas

Check in the source code that the pre-allocated areas according to the different sizes of malloc make it possible to store at least 100 times the max size for this type of area.
Also check that the size of the areas is a multiple of getpagesize().

If one of these points is missing, leave No.

Yes | No

Tests of free

We will simply add a free to our test program:

```
$> cat test2.c
#Include <stdllb. h="">

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

 i = 0;
 while (i < 1024)
 {
  addr = (char*)malloc(1024);
  addr[0] = 42;
  free(addr);
  i++;
 }
 return (0);
}
$> cc -o test2 test2.c
$>./run.sh /usr/bin/time -l ./test2
```

We will compare the number of "page reclaims" to that of test0 and test1. If there are as many "page reclaims" or more than test1, the free does not work (unless everyone is placed in a pre-allocated zone, in which case the claims page must not change between tests, test 1 and test 2; but this is rather unlikely as a situation, and an unwise choice of N and M).

Does the free work? (fewer "reclaims pages" than test1)

Yes | No

Quality of the free

test2 has a maximum of 3 "reclaims page" in addition to testO?

Yes | No

Realloc test

```
$> cat test3.c
#include <strings.h>
#Include <stdllb. h="">

#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, "Hello\n");
  print(addr1);
  addr3 = (char*) realloc(addr1, 128*M);
  addr3[127*M] = 42;
  print(addr3);
  return (0);
}
$> cc -o test3 test3.c
$> ./run.sh ./test3
Hello
Hello
$>.
```

Does it work as in the example?

Yes | No

Testing of realloc ++

In test3.c, modify the body of the main function in the following way:

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

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

Error handling

Test for special cases and errors.

```
$> cat test4.c
#include <unistd.h>
#include <string. h="">
#Include <stdllb. h="">

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("Hello\n");
}
$> cc -o test4 test4.c
$> ./run/sh ./test4
Hello
```

In case of error, realloc must return NULL. Is the "Hello" displayed as in the example?

If the program reacts in an unhealthy way (segfault or others), the defense stops and you must select Crash at the top of the bareme.

Yes | No

Test of show_alloc_mem

```
$> cat test5.c
#include <stdlib. h="">

int main()
{
  malloc(1024);
  malloc(1024*32);
  malloc(1024*1024);
  malloc(1024 * 1024 * 16 );
  malloc(1024 * 1024 * 128 );
  show_alloc_mem();
  return (0);
}
$> cc -o test5 test5.c -L. -lft_malloc
$> ./test5
```

The display corresponds to the topic and the TINY/SMALL/LARGE distribution of the project?

Yes | No

Bonus

Competitive access

The project manages concurrent thread accesses thanks to the pthread library and mutexes.

Count the applicable cases :

- a mutex prevents several threads from entering the malloc function simultaneously
- a mutex prevents several threads from entering the free function simultaneously
- a mutex prevents multiple threads from entering the realloc function simultaneously
- a mutex prevents several threads from entering the show_alloc_mem function simultaneously

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

Other bonuses

If there are other bonuses, count them here. The bonuses must be 100% functional and a minimum useful (at the discretion of the proofreader).

Bonus example:

-During a free, the project "defragments" the free memory by grouping the concomitant free blocks into a single
- Malloc has debug environment variables
- A function allows to make a hexa dump of the allocated areas
- A function allows to display a history of the memory allocations made
- ...

Rate it from O (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

Standard

Cheat

Crash

Forbidden function