(https://profile.intra.42.fr)

Remember that the quality of the defenses, hence the quality of the of the school on the labor market depends on you. The remote defences during the Covid crisis allows more flexibility so you can progress into your curriculum, but also brings more risks of cheat, injustice, laziness, that will harm everyone's skills development. We do count on your maturity and wisdom during these remote defenses for the benefits of the entire community.

SCALE FOR PROJECT MALLOC (/PROJECTS/42CURSUS-MALLOC)

You should evaluate 1 student in this team

Git repository

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

test4.c (/uploads/document/document/2487/test4.c)

test3.c (/uploads/document/document/2488/test3.c)

run_linux.sh (/uploads/document/document/2489/run_linux.sh)

test2.c (/uploads/document/document/2490/test2.c)

test6.c (/uploads/document/document/2491/test6.c)

test0.c (/uploads/document/document/2492/test0.c)

test5.c (/uploads/document/document/2493/test5.c)

test1.c (/uploads/document/document/2494/test1.c)

run_mac.sh (/uploads/document/document/2495/run_mac.sh)

subject.pdf (https://cdn.intra.42.fr/pdf/pdf/15965/en.subject.pdf)

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_$HOSSTYPE.so) and does create a symbolic link libft_malloc.so pointing towards libft_malloc_$HOSSTYPE.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 (adresses 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: FOR MAC: $> cat run.sh #!/bin/sh export DYLD_LIBRARY_PATH=. export DYLD_INSERT_LIBRARIES="libft_malloc.so" export DYLD_FORCE_FLAT_NAMESPACE=1 $@ $> FOR LINUX: $> cat run.sh #!/bin/sh export LD_LIBRARY_PATH=. export LD_PRELOAD=libft_malloc.so $@ $>

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.

Use test0.c file attached in the scale
WARNING: If you are using a linux vm, make sure that you are using the time binary that you can get with apt (sudo apt install time), else you won't have access to the -v option

MAC: $> gcc -o test0 test0.c
$> /usr/bin/time -l ./test0 0.00
LINUX: $> gcc -o test0 test0.c
$> /usr/bin/time -v ./test0 0.00
Command being timed: "./test0"
User time (seconds): 0.00
System time (seconds): 0.00
Percent of CPU this job got: 0%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 412
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 130
Voluntary context switches: 0
Involuntary context switches: 0
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
$>

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".

Using the test1.c file given as attachment, run the following tests

$> gcc -o test1 test1.c
$> /usr/bin/time -l ./test1
(Use -v for linux)
Command being timed: "./test1"
User time (seconds): 0.00
System time (seconds): 0.00
Percent of CPU this job got: 0%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0

Average total size (kbytes): 0
Maximum resident set size (kbytes): 1500
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 402
Voluntary context switches: 0
Involuntary context switches: 0
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0

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
(Use -v for linux)
Command being timed: "./test0"
User time (seconds): 0.00
System time (seconds): 0.00
Percent of CPU this job got: 0%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 412
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 130
Voluntary context switches: 0
Involuntary context switches: 0
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0

$>./run.sh /usr/bin/time -l ./test1

(Use -v for linux)
Command being timed: "./test1"
User time (seconds): 0.00
System time (seconds): 0.01
Percent of CPU this job got: 93%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.01
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 1984
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 525
Voluntary context switches: 0
Involuntary context switches: 0
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
$>

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

- less than 255 pages, allocated memory is insufficiant: 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)

2

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

$> 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
(Use -v for linux)

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

Using test3.c file given as attachment, test the following:
$> gcc -o test3 test3.c
$> ./run.sh ./test3 Bonjour Bonjour
$>

Does it work as expected?

☑ Yes ☒ No

Realloc test++

test4.c is a small modification of test3.c. Run it, does it still work ?

☑ Yes ☒ No

Error management

Using test5.c, test all the particular cases and errors:
$> gcc -o test5 test5.c
$> ./run.sh ./test5 Bonjour

In case of error, realloc must return NULL. Is "Bonjour" displayed as expected? 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

$> gcc -o test6 test6.c -L. -lft_malloc

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 simulteanously enter inside the malloc function
- a mutex prevents multiple threads to simulteanously enter inside the free function
- a mutex prevents multiple threads to simulteanously enter inside the realloc function
- a mutex prevents multiple threads to simulteanously 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.
- If there are other bonuses, add them up here. Bonuses must be 100% functional and a minimum useful (at the corrector's discretion)

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

4

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

Finish evaluation