A rather UNIX project

malloc

Summary: *This project is about implementing a dynamic memory allocation mechanism.*

Version: 6.3

Contents

Chapter I

Foreword

Here is what Wikipedia has to say about sensory memory.

During every moment of an organism's life, sensory information is being taken in by sensory receptors and processed by the nervous system. Sensory information is stored in sensory memory just long enough to be transferred to short-term memory.[1] Humans have five traditional senses: sight, hearing, taste, smell, touch.

Sensory memory (SM) allows individuals to retain impressions of sensory information after the original stimulus has ceased.[2] A common demonstration of SM is a child's ability to write letters and make circles by twirling a sparkler at night. When the sparkler is spun fast enough, it appears to leave a trail which forms a continuous image. This "light trail" is the image that is represented in the visual sensory store known as iconic memory.

The other two types of SM that have been most extensively studied are echoic memory, and haptic memory; however, it is reasonable to assume that each physiological sense has a corresponding memory store. Children for example have been shown to remember specific "sweet" tastes during incidental learning trials but the nature of this gustatory store is still unclear.

Chapter II

General Instructions

- This project will be corrected by humans only. You're allowed to organise and name your files as you see fit, but you must follow the following rules.

- The library must be named `libft_malloc_$HOSTTYPE.so`.

- A `Makefile` or something similar must compile the project and must contain the usual rules. It must recompile and re-link the program only if necessary.

- Your Makefile must verify the existence of the environment variable `$HOSTTYPE`. If this variable is empty or undefined, it should be assigned the following value:
  `'uname -m'_'uname -s'`

```
        ifeq ($(HOSTTYPE),)
            HOSTTYPE := $(shell uname -m)_$(shell uname -s)
        endif
```

- Your Makefile will have to create a symbolic link `libft_malloc.so` pointing to `libft_malloc_$HOSTTYPE.so` so for example:
  libft_malloc.so -> libft_malloc_intel-mac.so

- If you are clever, you will use your library for your `malloc`. Submit also your folder `libft` including its own `Makefile` at the root of your repository. Your `Makefile` will have to compile the library, and then compile your project.

- You are allowed one global variable to manage your allocations and one for thread-safety.

- Your project must be clean code even without the norm. If it's ugly, you will get 0.

- You must handle errors carefully. In no way can your functions lead to undefined behavior or segfaults.

- Within the mandatory part, you are allowed to use the following functions:

  - mmap(2)
  - munmap(2)
  - getpagesize under OSX or sysconf(_SC_PAGESIZE) under linux
  - getrlimit(2)

- ○ The authorized functions within your libft (write(2) for example)

- ○ The functions from libpthread

- You are allowed to use other functions to complete the bonus part as long as their use is justified during your defence. Be smart!

- You can ask your questions on the forum, on slack...

Chapter III

Mandatory part

This mini project is about writing a dynamic allocation memory management library. So that you can use it with some programs already in use without modifying them or recompiling, you must rewrite the following libc functions malloc(3), free(3) and realloc(3).

Your functions will be prototyped like the system ones:

```
#include <stdlib.h>

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

- The malloc() function allocates "size" bytes of memory and returns a pointer to the allocated memory.

- The realloc() function tries to change the size of the allocation pointed to by "ptr" to "size", and returns "ptr". If there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by "ptr" as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.

- The free() function deallocates the memory allocation pointed to by ptr. If ptr is a NULL pointer, no operation is performed.

- If there is an error, the malloc() and realloc() functions return a NULL pointer.

- You must use the mmap(2) and munmap(2) system calls to request and return memory zones to the system.

- You must manage your own memory allocations for the internal functioning of your project without using the libc malloc function.

- With performance in mind, you must limit the number of calls to mmap(), but also to munmap(). You have to "pre-allocate" some memory zones to store your "small" and "medium" malloc.

- The size of these zones must be a multiple of getpagesize() under macOS or sysconf(_SC_PAGESIZ under Linux.

- Each zone must contain at least 100 allocations.

  - "TINY" mallocs, from 1 to n bytes, will be stored in N bytes big zones.

  - "SMALL" mallocs, from (n+1) to m bytes, will be stored in M bytes big zones.

  - "LARGE" mallocs, from $(m + 1)$ bytes and above, will be allocated outside the standard memory zones. This means they will be handled using `mmap()`, placing them in their own separate memory zone.

- It's up to you to define the size of n, m, N and M so that you find a good compromise between speed (saving on system recall) and saving memory.

You also must write a function that allows visualization of the state of the allocated memory zones. It needs to be prototyped as follows:

```
void    show_alloc_mem();
```

The visual will be formatted by increasing addresses such as:

```
TINY : 0xA0000
0xA0020 - 0xA004A : 42 bytes
0xA006A - 0xA00BE : 84 bytes
SMALL : 0xAD000
0xAD020 - 0xADEAD : 3725 bytes
LARGE : 0xB0000
0xB0020 - 0xBBEEF : 48847 bytes
Total : 52698 bytes
```

> ⚠ You must properly align the memory returned by your malloc

Chapter IV

Bonus part

The following is the first bonus in this subject:

- Manage the use of your malloc in a multi-threaded program (make it "thread-safe" using the pthread library).

In order to get the maximum score you have to implement some additional functions (non exhaustive list) such as:

- Manage malloc debug environment variables. You can imitate those from the system malloc or invent your own.

- Implement a `show_alloc_mem_ex()` function that provides extended details, such as a history of memory allocations or a hexadecimal dump of the allocated memory zones.

- Defragment the freed memory.

> ⚠ The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter V

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.

Chapter VI

Let's laugh a little

In a not so distant past, the malloc project had to be made using brk(2) and sbrk(2) instead of mmap(2) and munmap(2). Here is what brk(2) and sbrk(2)'s man has to say on dinosaur's era:

```
$> man 2 brk
...
DESCRIPTION
The brk and sbrk functions are historical curiosities left over from
earlier days before the advent of virtual memory management.
...
4th Berkeley Distribution   December 11, 1993   4th Berkeley Distribution
$>
```

From that description result most probably the brevity of brk(2)'s implementation on Mac Os X:

```
void *brk(void *x)
{
  errno = ENOMEM;
  return((void *)-1);
}
```