Lab #5: Building a Memory Allocation System

October 23rd

Introduction

In this lab we look at managing free memory: we learn how an operating system allocates and frees memory.

due 9 am November 11th

Part I Linux memory allocation

1. Read the entire man page for the `sbrk()` function and describe using your own words (without copying and pasting) what this function does. You may also want to read this article.

2. Locate the "`lab5n1.c`" file. It uses the `malloc()` and `sbrk()` functions:
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
   char* start = sbrk( 0 );
   printf("sbrk says: %p\n", start);
   printf("Calling malloc ...\n");
   size_t size       = 16384; //bytes
   char*  b = (char*) malloc(size);
   printf(" --> malloc result is at %p\n", b);

   char* end = sbrk( 0 );
   printf("sbrk says: %p\n", end);
   free( b );
}
```
Run the program 3 times and show the output of each run. In your report, explain the results:
   - what do the various numbers mean?
   - what changes with each run and what stays the same for the 3 runs?

   After the above 3 runs, modify this program to also call the `sbrk()` function after the call to `free()` and print the new `sbrk` return value. Run the modified program three times and explain the new results.

3. Read the man page for the `mmap()` function. Describe using your own words (no copying and pasting) what the `mmap` and `munmap()` functions do. This article provides more information on mmap.
In the `lab5n2.c` program shown below, explain what the three components of the expression **`MAP_ANON | MAP_PRIVATE`** mean and what the result of this expression is.

4. Locate the "`lab5n2.c`" file. It uses the `mmap()` function:
```
#include <stdio.h>
#include <sys/mman.h>

int main(){
   printf("Calling mmap ...\n");
   size_t size       = 16384; //bytes
   int    protection = PROT_READ|PROT_WRITE; // rw
   int    flags      = MAP_ANON|MAP_PRIVATE; // anonymous and private
   int    nofd       = 0;
   char*  a = mmap( NULL, size, protection, flags, nofd, 0);

   printf(" --> result of mmap is at %p\n", a);
}
```
What do the two least significant bytes of the mmap result tell you about where the mmap'ed memory is?

Modify this program to allocate heap memory using `malloc` after the call to `mmap` and print the location in the heap from where the memory was allocated.
Show the output of your program as it `malloc`s 10, 100, 1000, 10,000, 100,000, 1,000,000 and 10,000,000 bytes. For each `malloc` size, run the program 3 times to see what the differences are.

Add code to free any malloc'ed memory and `munmap` any `mmap`'ed memory and print out the result of a call to `sbrk(0)` at the end of `main`().

Run the program 3 times and show the run outputs. In your report, explain what the outputs tell you about where the heap is in virtual memory. Submit the modified program with your prelab report.

Part II Creating our own heap memory allocation

In this part, we will implement a **heap memory allocator**. We have three objectives in this lab:

- To understand how to build a memory allocator.
- To do so in a performance-efficient manner.
- To create a shared library for our allocator.

Our job will be to build our own `malloc`() and `free`().

A memory allocator has two distinct tasks.

- It asks the operating system to expand the heap portion of the process's address space by calling either `sbrk` or `mmap` to ask for a "slab" of memory.
- It doles out memory from this slab to user processes that request heap memory. This involves managing a "free" list (one per slab) of memory and finding a contiguous chunk of memory that is large enough for the user's request; when the user later frees memory, the memory to be freed is added back to this free list.

Such a memory allocator is usually provided as part of a standard library and is not part of the OS. To be clear, the memory allocator operates entirely within the virtual address space of a single process and knows nothing about which physical pages have been allocated to this process or the mapping from logical addresses to physical addresses.

When implementing this basic functionality, there are a few requirements:

- When requesting a "slab" of memory from the OS, use **`mmap()`** (which is easier to use than `sbrk`()). Subsequent allocations should be doled out from this slab and we cannot use malloc
- Our memory allocator must call `mmap`() only once for each slab (see "Mem_Init" below). If a slab turns out to be too small, the user should be able to ask for another bigger slab and the old slab can stay where it is.

  (*Real memory allocator requests more memory from the OS whenever it can't satisfy a request from the user*)

The classic `malloc()` and `free()` are defined as follows:

- `void *malloc(size_t size)`: malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.
- `void free(void *ptr)`: free() frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc() (or calloc() or realloc()). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

For simplicity, our implementations of `Mem_Alloc(int size)` and `Mem_Free(void *ptr)` should mainly follow what `malloc()` and `free()` do. See below for details.

Provide a supporting function, `Mem_Dump()`, which will print which regions are currently free. We can use this function for debugging purposes.

Program Specifications

Use the provided **`mem.h`** header file. Include this header file in your code to ensure that you are adhering to the specifications exactly. **Do not change `mem.h`** in any way!

Create at least two files: **`mem.c`** and a **`main.c`** and a **Makefile**.

- The `main`() function for the executable should be in `main.c` and it should use and test the functions in mem.c. `#include` the `mem.h` file in `main.c`.
- `mem.c` should contain your implementation of memory initialization, allocation and deallocation functions. See the prototypes for these functions in **`mem.h`**. Here are the more precise definitions of each of these routines:

- **`void *Mem_Init(int sizeOfRegion)`**
  `Mem_Init` is to be called once for a slab of memory and subsequent Allocs and Frees should use this "slab" using our routines.
  `sizeOfRegion` is the number of bytes that we should request from the OS using `mmap`.
  `Mem_Init` must return the address of the region returned by `mmap`. If `mmap` fails, return `NULL`.
  Outside of the mmap'd region, we are allowed only one `static` variable used to point to the region. Note that this means we need to use the allocated memory region returned by `mmap` for our own data structures as well; that is, our infrastructure for tracking the mapping from addresses to memory objects has to be placed in this region as well.
  We are not allowed to use `malloc()`, or any other related function, in any of our routines! Similarly, we cannot allocate global arrays!
  Note that we may need to round up `sizeOfRegion` so that we request memory in units of the page size (see the man pages for `getpagesize()`).
- **`void *Mem_Alloc(int size)`**
  `Mem_Alloc()` is similar to the library function `malloc()`.
  `Mem_Alloc` takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object within the most recently initialized slab of memory. The function returns `NULL` if there is not enough contiguous free space within the slab of size **`sizeOfRegion`**.
  Use the **Best Fit** allocation algorithm. You can implement First Fit and Worst Fit for testing but the final submitted version should use Best Fit.
- **`int Mem_Free(void *ptr)`**
  `Mem_Free()` frees the memory object that ptr points to. Just like with the standard `free()`, if ptr is `NULL`, then no operation is performed. The function returns 0 on success, and -1 otherwise.
  Avoid errors and bugs by making sure the pointer is valid and within the current slab.
- **`void Mem_Dump()`**
  This is a debugging routine for our use in checking the above functions. The above three functions should not print anything to the console/terminal. `Mem_Dump` is the only one that can use `printf()`. Have this function print the regions of free memory to the screen.

You are not required to coalesce memory. However, `free`'d regions should be reusable for future allocations that are equal or smaller.

Shared Library

Your `mem.c` should be used to create a shared library – to mimic the way glibc functions work. Your main.c should be linked to a shared library named "`libmem.so`".

Placing the memory handling routines in a shared library instead of a simple object file makes it easier for other programmers to link with our code. There are further advantages to shared (dynamic) libraries over static libraries. When we link with a static library, the code for the entire library is merged with our object code to create your executable; if we link to many static libraries, our executable will be enormous. However, when we link to a shared library, the library's code is not merged with our program's object code; instead, a small amount of stub code is inserted into our object code and the stub code finds and invokes the library code when we execute the program. Therefore, shared libraries have two advantages: they lead to smaller executables and they enable users to use the most recent version of the library at run-time.

To create a shared library named libmem.so, use the following commands (assuming our library code is in a single file "`mem.c`"):

```
gcc -c -fpic mem.c -Wall -Werror
gcc -shared -o libmem.so mem.o
```

To link with the “**`libmem.so`**” library file, we use the "**`-l`**" (that’s a letter L, not the number 1) flag and specify the base name of the library: "**`-lmem`**". We can specify the library path using the "`-L`" flag so that the linker used by `gcc` can find library files. If the library file is in the current directory, the library path option will be "**`-L.`**":

```
gcc -L. -o myprogram mytestmain.c -Wall -Werror -lmem
```

Place these commands in a `Makefile`. Before we run "`myprogram`", we will need to set the environment variable, `LD_LIBRARY_PATH`, so that the system can find our library at run-time. Assuming we always run `myprogram` from this same directory, we can use the command:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

The `export` command is what the `bash` shell uses. If you are not using a different shell, you will have to figure out the alternative command to set the environment.

Unix implementation hints

We can use mmap to map zeroed pages (i.e., allocate new pages) into the address space of the calling process. There are many ways that we can call mmap to achieve this same goal. Here is one example:

```
// open the /dev/zero device
int fd = open("/dev/zero", O_RDWR);

// sizeOfRegion (in bytes) needs to be evenly divisible by the page size
void *ptr = mmap(NULL, sizeOfRegion, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd,
0);
if (ptr == MAP_FAILED) { perror("mmap"); exit(1); }

// close the device (don't worry, mapping should be unaffected)
close(fd);
return 0;
```

Testing

Internal checking: make good use of `errno` and the `perror` and `strerror` functions and check the return values of all calls to Mem_Alloc and Mem_Free.

External checking: You should write tests that will crash your program and then fix any bugs and try to make it crash-proof.

Allocate as much of the memory that you set up. Then free the memory and make sure you can reallocate memory (that was previously allocated and then freed).

Ideally, a malloc/free pair should not affect future mallocs or frees. Write tests that will test this ideal situation.

In your Lab Report document explain how you tested your `malloc` and `free` functions:

- What sequence of `malloc` and `free` function calls did you use and what sizes did you use?
- What do these sequences test? Are you testing whether you can allocate memory that was previously allocated?
- Which sequences of calls will crash a program?

Grading

Your implementation will be graded mainly on functionality and testing. However, I will also be looking at the relative performance of your programs, so try to be efficient!

Lab 5 Submission

Create a simple **Makefile** mainly to ensure that I am able to compile your C code correctly.
Do **not** submit any .o, .so, or .a files. Make sure that your code runs correctly on your Linux VM.

Turn in, as a group, in Canvas a single .tar.gz file containing:

- a report document with:
  - answers to the questions
  - a list of the ways you tested the submitted version of your functions
- all requested source files with
  - a Makefile for building all the C programs using a single make command
  - **required:** the allocator part of the code should compile into a shared library.