Project 3: Malloc and Free

Objectives

There are three objectives to this part of the assignment:

- To understand the nuances of building a memory allocator.
- To do so in a performance-efficient manner.
- To create a shared library.

Readings and Notes

At some point you will decide to use a header per each allocated block. The maximum size of such a header is 16 bytes.

Useful to read OSTEP Chapter 16.

Overview

In this project, you will be implementing a memory allocator for the heap of a user-level process. Your functions will be to build your own malloc() and free().

Memory allocators have two distinct tasks. First, the memory allocator asks the operating system to expand the heap portion of the process's address space by calling either sbrk or mmap. Second, the memory allocator doles out this memory to the calling process. This involves managing a free list of memory and finding a contiguous chunk of memory that is large enough for the user's request; when the user later frees memory, it is added back to this list.

This 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; that part is handled by the operating system.

When implementing this basic functionality in your project, we have a few guidelines. First, when requesting memory from the OS, you must use **mmap()** (which is easier to use than sbrk()). Second, although a real memory allocator requests more memory from the OS whenever it can't satisfy a request from the user, your memory allocator must call mmap() only **one** time (when it is first initialized).

Classic malloc() and free() are defined as follows:

- `void *malloc(size_t size)` , which allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.
- `void free(void *ptr)` , which 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, your implementations of `mem_alloc()` and `mem_free()` should basically follow what malloc() and free() do; see below for details.

You will also provide a supporting function, `mem_dump()` , described below; this routine simply prints which regions are currently free and should be used by you for debugging purposes.

Program Specifications

For this project, you will be implementing several different routines as part of a shared library. Note that you will not be writing a main() routine for the code that you handin (but you should implement one for your own testing). We have provided the prototypes for these functions in the file mem.h; you should include this header file in your code to ensure that you are adhering to the specification exactly. **You should not change mem.h in any way!** We now define each of these routines more precisely.

- `int mem_init(int size_of_region)` : mem_init is called one time by a process using your routines. `size_of_region` is the number of bytes that you should request from the OS using mmap().

  Note that you may need to round up this amount so that you request memory in units of the page size (see the man pages for `getpagesize()`). Note also that you need to use this

allocated memory for your own data structures as well; that is, your infrastructure for tracking the mapping from addresses to memory objects has to be placed in this region as well. You are **not** allowed to malloc(), or any other related function, in any of your routines! Similarly, you should **not** allocate global arrays. However, you may allocate a few global variables (e.g., a pointer to the head of your free list.)

Return 0 on a success (when call to mmap is successful). Otherwise, return -1 and set `m_error` to `E_BAD_ARGS`. Cases where mem_init should return a failure: mem_init is called more than once; size_of_region is less than or equal to 0.

- `void *mem_alloc(int size, int style)` : 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. The function returns NULL if there is not enough contiguous free space within `size_of_region` allocated by mem_init to satisfy this request (and sets `m_error` to `E_NO_SPACE`).

  The style parameter determines how to look through the list for a free space. It can be set to `M_BESTFIT` (BF) for the best-fit policy, `M_WORSTFIT` (WF) for worst-fit, and `M_FIRSTFIT` (FF) for first-fit. BF simply looks through your free list and finds the first free space that is smallest in size (but still can hold the requested amount) and returns the requested size (the first part of the chunk) to the user, keeping the rest of the chunk in its free list; WF looks for the largest chunk and allocates the requested space out of that; FF looks for the first chunk that fits and returns the requested space out of that.

  For performance reasons, `mem_alloc()` should return 8-byte aligned chunks of memory. For example if a user allocates 1 byte of memory, your `mem_alloc()` implementation should return 8 bytes of memory so that the next free block will be 8-byte alligned too. To figure out whether you return 8-byte aligned pointers, you could print the pointer this way `printf("%p", ptr)`. The last digit should be a multiple of 8 (i.e. 0 or 8).

- `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.

  Coalescing: mem_free() should make sure to coalesce free space. Coalescing rejoins neighboring freed blocks into one bigger free chunk, thus ensuring that big chunks remain free for subsequent calls to mem_alloc().

- `void mem_dump()` : This is just a debugging routine for your own use. Have it print the regions of free memory to the screen.

You must provide these routines in a shared library named `libmem.so`. Placing the routines in a shared library instead of a simple object file makes it easier for other programmers to link with your code. There are further advantages to shared (dynamic) libraries over static libraries.

When you link with a static library, the code for the entire library is merged with your object code to create your executable; if you link to many static libraries, your executable will be enormous. However, when you link to a shared library, the library's code is not merged with your program's object code; instead, a small amount of stub code is inserted into your object code and the stub code finds and invokes the library code when you 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 your 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 this library, you simply specify the base name of the library with `-lmem` and the path so that the linker can find the library `-L.`

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

Of course, these commands should be placed in a Makefile. Before you run "myprogram", you will need to set the environment variable, `LD_LIBRARY_PATH` , so that the system can find your library at run-time. Assuming you always run myprogram from this same directory, you can use the command:

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

Unix Hints

In this project, you will use mmap to map zero'd pages (i.e., allocate new pages) into the address space of the calling process. Note there are a number of different ways that you can call mmap to achieve this same goal; we give one example here:

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

// size_of_region (in bytes) needs to be evenly divisible by the page size
```

```
void *ptr = mmap(NULL, size_of_region, 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;
```

Hand In

- Source file `mem.c`
- Makefile which builds the memory allocation library.

Grading

Your implementation will be graded on functionality. However, we will also be comparing the performance of each of your projects, so try to be efficient!

Adapted from WISC CS537 by Remzi Arpaci-Dusseau