Project 3a: Malloc and Free

DUE 03/17 at 11:59 PM

One late day allowed for submission without any penalty

Objectives

There are four 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.
- To do the above in a thread-safe manner.

Note

Useful to read: Chapter 17 from the free operating systems book.

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): malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared in the standard malloc. In this project there is a specific case where the memory you return should be cleared to zeros. See below.

- 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, your implementations of Mem_Alloc(int size) and Mem_Free(void *ptr) 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 **mymem.h** (which is available at ~cs537-1/public/mymem.h) ; you should include this header file in your code to ensure that you are adhering to the specification exactly.

You should not change mymem.h in any way!

We now define each of these routines more precisely.

- **void * Mem_Init(int sizeOfRegion, int slabSize):** Mem_Init is called one time by a process using your routines. sizeOfRegion is the number of bytes that you should request from the OS using mmap(). See below for the discussion of slabSize.

  We promise to request total memory blocks in multiples of the page size.

  Note also that you need to use this allocated memory for your own data structures as well; that is, your infrastructure for tracking free space has to be placed in this region too.

  You are not allowed to use 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 NULL on a failure. Return the base address of the memory provided to you by mmap on success. Cases where Mem_Init should return a failure: Mem_Init is called more than once.

- **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. The function returns NULL if there is not enough contiguous free space available.

- **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.

  It is an error to attempt to Mem_Free a ptr outside the mmap'ed memory. Your program should print "SEGFAULT\n" to stdout and exit with an error return of -1.

  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(). Our testing requires you coalesce during each free operation.

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

Allocators

You will be writing **two** allocators. These are: slab and next fit.

One quarter of your allocated memory region will be devoted to a slab allocator. Three quarters of your allocated memory region will be devoted to a next fit allocator. We promise to request you initialize only regions which are a multiple of four in size. The slabSize parameter to Mem_Init sets a "special size." Anytime you receive a request to allocate that specific number of bytes, you should first try your slab allocator. In the event you run out of available slabs, your alloc function must fall back to your next fit allocator.

Any memory matching the requested "special size" must be initialized to zero. This is different from the behavior of the standard malloc but is more in line with the functions of a slab allocator.

Your next fit allocator is responsible for managing three quarters of the memory you've set aside (slab manages the other quarter). Note you must properly handle coalescing of free space.

Note again: Your free space management is to be implemented in free memory not in a data structure outside your mmap'ed memory.

Note again: Allocation requests for slabSize sized blocks should come first from the slab allocator and from next fit only if all slabs are allocated. We will know the difference!

For performance reasons, Mem_Alloc() should return 16-byte aligned chunks of memory. For example if a user allocates 1 byte of memory, your Mem_Alloc() implementation should return 16 bytes of memory so that the next free block will be 16-byte alligned too. To figure out whether you return 16-byte aligned pointers, you could print the pointer this way `printf("%p", ptr)` . The last digit should be a multiple of 16 (i.e. 0 or 16). We promise that the slab element size that is requested will be greater than or equal to 16 though not necessarily a multiple of 16.

Thread safe

We will drive your code from our own test programs which will employ multiple threads calling your allocate and free functions asynchronously.

Produce a shared library

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:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:.
```

If the setenv command returns an error "LD_LIBRARY_PATH: Undefined variable", do not panic. The error implies that your shell has not defined the environment variable. In this case, you simply need to run:

```
setenv LD_LIBRARY_PATH .
```

Note that setenv is what you use in tcsh; if you are using bash, you'll have to figure out the alternative command to set the environment.

Grading and Teams

Your implementation will be graded on functionality. Sanity check and a subset of test cases will be provided.

Teams of two are encouraged. Hold you partner responsible for any delays! Teams may span the sections of the class.