Assignment 3
Deadline: Due at 11:59PM on April 20, 2024

In this assignment, you will work in a group of maximum 3. you will be implementing a memory allocator for the heap of a user-level process. Your tasks will be to build your own malloc() and free(). 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.

For this assignment, you will be implementing several different routines. You are provided with 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!** You can find more details here.

You will be working in a group of 3 students.

Overview

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).

Requirements and Specifications

1. Understanding Classic malloc() and free()

- **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.

2. Requirements

- **Implement mem_alloc() and mem_free().** For simplicity, your implementations of mem_alloc() and mem_free() should basically follow what malloc() and free() do.
- **Implement a supporting function, mem_dump().** You will provide a supporting function, mem_dump(); this routine simply prints which regions are currently free and should be used by you for debugging purposes.
- **Implement 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().

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 aligned 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 a debugging routine for your code. Have it print the regions of free memory to the screen.

```
./mem.out
Memory Dump After Allocations:
Free Memory Dump:
  [Size: 4048 bytes]

Memory Dump After Freeing ptr1:
Free Memory Dump:
  [Size: 16 bytes]
  [Size: 4048 bytes]

Final Memory Dump:
Free Memory Dump:
  [Size: 32 bytes]
  [Size: 16 bytes]
  [Size: 4048 bytes]
```

Note that this is just an example. Your results may differ.

3. How to compile and execute

- The Makefile for assignment3 is provided.
- The Makefile is supposed to work with mem.c, file so, make sure to name the source file accordingly.
- Run the following command in vs code Terminal to compile the code.
  `make`
- Run the following commands to run the code.
  `make run`
- Run the following command to clean the out file.
  `make clean`

4. Submission Guidelines

- Submit only one file mem.c.
- Include your student numbers with First and Last Name at the top of your code as comments including your partner's too.
- Any missing details will result in ZERO.

5. Grading

- Correct implementation of the functions defined in mem.h **(4 Pts)**
- Implementation of main function and displaying appropriate messages with regions of memory **(4 Pts)**
- Printing appropriate error messages to the console **(1 Pts)**
- Code organization, readability and documentation/comments in code **(2 Pts)**

Good luck with your assignment!