PROJECT :: Memory Allocator V2 - Part 1

For this assignment, you will be writing your own memory allocator. Writing a custom memory allocator is something you might do if you work on performance sensitive systems (games, graphics, quantitative finance, embedded devices or any application you want to run fast!). Malloc and free are general purpose functions written to manage memory in the average use case quite well, but they can always be optimized for a given workload. That said, a lot of smart people have worked on making malloc/free quite performant over a wide range of workloads. Optimization aside, you might write an allocator to add in debugging features, and swap it in as needed.

Release History

You are always responsible for the latest release. The release may be updated at any time to fix bugs or add clarification. Get used to it, programmers are expected to adapt to changing requirements.

V002 : Clarified and removed ambiguity with respect to the ufree definition to match free and to exit on a double free.
V001 : Updated for FA24, expect bugs!

Introduction

For this assignment, you will implement a portion of a custom memory allocator for the C language. You will write your own versions of:

- malloc
- free
- realloc

In addition you will also need to write an initialization routine that establishes the allocation policy and you will write a function `umemstats` that prints statistics of the current state of the memory allocator.

It's tempting to think of malloc() and free() as OS system calls, because we usually think of memory management as being a low-level activity, but they are actually part of the C standard library. When you call malloc(), you are actually calling a C routine that may make system calls to ask the OS to modify the heap portion of the process's virtual address space if necessary.

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.

> 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 the sbrk or mmap system call. Second, the memory allocator doles out this memory to the calling process. To do this, the allocator maintains a free list of available memory. When the process calls malloc(), the allocator searches the free list to find a contiguous chunk of memory large enough to satisfy the user's request. Freeing memory adds the chunk back into the free list, making it available to be allocated again by a future call to malloc().
>
> 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). This is a simplification to keep this project manageable.
>
> The functions `malloc()` and `free()` are defined as given below.
>
> - `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.
> - `void *realloc(void *ptr, size_t size)` : This function resizes a previously allocated memory block.
>
> You will implement the following routines in your code
>
> - umalloc : has the same interface as malloc
> - urealloc : has the same interface as realloc
> - ufree : has the same interface as free
> - Described in more detail below
>   - void umemstats(void) : prints which regions are currently free and is primarily to aid your debugging.
>   - umeminit : initializes your memory allocator.

Program Specifications

For this project, you will be implementing several different routines. Note that your main() routine must be in a separate file called `main.c`. ***Do not include a main function in your `umem.c` file; you should implement this only in your testing file main.c.*** You will submit both files for grading. Your `umem.c` file will be evaluated for functionality and will be linked to our testing code. Your `main.c` file will be evaluated for thoroughness and attention to detail in terms of testing.

We have provided the prototypes for these functions in the file **umem.h.** you should include this header file in your code to ensure that you are adhering to the specification exactly. **You should not change umem.h in any way!**

- `int umeminit (size_t sizeOfRegion, int allocationAlgo):` umeminit 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(). Do not define your own constants for the allocation algorithm but used the defined constants provided in umem.h. This is not an object oriented programming course, your allocator may simply use a switch statement based on the state of a global variable initialized in this function. I do suggest that you break up your code into smaller meaningfully named functions to aid in debugging. Some points will be, wait for it, allocated, for code quality.

  You must 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, as described in your textbook. 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, and the aforementioned algorithm state variable.)

  Return 0 on a success (when call to mmap is successful). Otherwise it should return -1. The umeminit routine should return a failure if is called more than once or sizeOfRegion is less than or equal to 0.

- `void *umalloc(size_t size)`: **umalloc**() is similar to the library function malloc(). umalloc 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 **sizeOfRegion** allocated by umeminit to satisfy this request. Given a request of a certain size, there are many possible strategies that might be used to search the free list for an satisfactory piece of memory. For this project, you are required to implement BEST_FIT, WORST_FIT, FIRST_FIT, and NEXT_FIT.
  - BEST_FIT searches the list for the smallest chunk of free space that is large enough to accommodate the requested amount of memory, then returns the requested amount to the user starting from the beginning of the chunk. If there are multiple chunks of the same size, the BESTFIT allocator uses the first one in the list to satisfy the request.
  - WORST_FIT searches the list for the largest chunk of free space
  - FIRST_FIT finds the first block that is large enough and splits this block
  - NEXT_FIT is FIRST_FIT except that the search continues from where it left off at the last request.

  - For performance reasons, `umalloc()` should return 8-byte aligned chunks of memory. For example if a user allocates 1 byte of memory, your `umalloc()` 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).

  - **Note:** You are required to use the headers as defined in umem.h. These are almost identical to the headers in the book except that they have been adapted to work more cleanly with a 64 bit system, which is required. You may not add anything to these headers and you may not redefine them in your C files.

  - **Note:** You are required to use the **MAGIC** constant defined in `umem.h` for the `magic` field in the `header_t` structure. This value will be used for integrity checks to detect memory corruption. When freeing or reallocating memory, your implementation must verify that the `magic` field has not been altered. If it has, an error should be raised to indicate memory corruption. If a corruption is detected, print an error message and exit the program immediately with `exit(1)`. This ensures that the allocator stops execution in case of serious issues like memory corruption, which could lead to undefined behavior or more severe errors later on.

    1. **Print an error message** to `stderr` detailing the corruption. For example:

       ```
       fprintf(stderr, "Error: Memory corruption detected at block %p\n", ptr);
       ```

    2. **Exit the program** immediately with a return code of `1` using `exit(1)` to signal a failure. This prevents the allocator from continuing with corrupted memory, which could lead to undefined behavior.

- **void ufree(void *ptr):** ufree() frees the memory object that ptr points to. If `ptr` is NULL, no operation is performed, For valid pointers, the function attempts to coalesce adjacent free blocks, combining neighboring blocks into larger chunks to reduce fragmentation and maintain memory availability.

  To handle cases of double-freeing, `ufree` should verify that the block being freed is not already in the free list. If a double-free is detected, print an error message to `stderr` and exit the program immediately to avoid memory corruption. This ensures that memory safety is enforced without complex error handling.

- `void *urealloc(void *ptr, size_t size)`: This function resizes a previously allocated memory block. If `ptr` is NULL, it behaves like `umalloc(size)`. If `size` is 0, it behaves like `ufree(ptr)`. Otherwise, it attempts to resize the memory block pointed to by `ptr` to the new size. If there is sufficient contiguous space after the block, it should resize in place; otherwise, it must allocate a new block, copy the data from the old block to the new one, free the old block, and return a pointer to the new block. If reallocating fails due to insufficient memory, return `NULL`.

  This function introduces complexity by handling cases where the block needs to grow or shrink. Be mindful of maintaining 8-byte alignment in your allocations. You should handle the edge cases, such as `ptr` being `NULL` or `size` being zero, and ensure that the function works seamlessly with your existing `umalloc` and `ufree` implementations.

- `void umemstats(void)`: This function will compute and printvarious statistics related to your memory allocator. The required statistics include:
  - The total number of successful allocations made (`umalloc` or `urealloc` calls).
  - The total number of successful deallocations (`ufree` calls).
  - The current amount of allocated memory.
  - The current amount of free memory.
  - The total memory fragmentation (i.e., the percentage of memory in small free blocks that cannot be easily reused for larger allocations).
    - **Memory Fragmentation Explanation:**

      Memory fragmentation refers to the condition where available free memory is divided into small, non-contiguous blocks, making it difficult to satisfy larger memory allocation requests. In your implementation, fragmentation occurs when there are many small, free blocks scattered throughout the memory region, which can't be coalesced into a larger contiguous block.

      To calculate the percentage of fragmentation, identify the total amount of free memory and determine how much of that free memory is made up of blocks that are too small to be useful for larger allocations.

      A potential formula for fragmentation might be:

      Fragmentation (%) = (Memory in small free blocks / Total free memory) * 100

      Where "small free blocks" could be defined as blocks that are smaller than a certain threshold (e.g., less than half the size of the largest free block). This percentage gives an idea of how fragmented the memory is and how likely it is that future allocations may fail despite having free memory available.

  - You must make use of the following function to print the stats. Call this function from within your `umemstats()` function. Note that this function is defined as a macro in the header file already. You may directly use the macro, or, for testing comment it out and define this function. Keep in mind that if you do that, your code won't compile in my test scripts which will not make use of any changes that you make to `umem.h`. So, if you do edit umem.h, make sure that you revert to using the macro before submission.

```
void printumemstats(int total_allocations, int total_deallocations, size_t allocated_memory, size_t free_memory, double fragmenta
tion) {
    printf("Memory Allocation Statistics:\n");
    printf("Total Allocations: %d\n", total_allocations);
    printf("Total Deallocations: %d\n", total_deallocations);
    printf("Currently Allocated Memory: %zu bytes\n", allocated_memory);
    printf("Currently Free Memory: %zu bytes\n", free_memory);
    printf("Memory Fragmentation: %.2f%%\n", fragmentation);
}
```

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);

// 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;
```

Grading and Submission

Your implementation will be graded on functionality by script for `umem.c`, and by thoroughness of testing effort and documentation and code quality in `main.c`. Some number of points may be based on the cleanliness of code, particularly with respect to functional decomposition. Note that your main.c file must be extensively documented. Each test case that you implement should be thoroughly described in terms of how it works, what it's testing, and how it achieves the goal of testing that feature. This will force you to engage with the material and will, in fact, help you write the code for the different algorithms.

Upload your `umem.c` and your `main.c` file. We do not need you to upload `umem.h`. Note that you may not make any changes to `umem.h` and any uploaded copy of `umem.h` will be ignored by the grader.