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.

V003 : Adjusted header size maximum to 32 bytes
V002 : Added limitation for header size.
V001 : Initial release, expect a few 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
In addition you will write a utility to dump the state of your currently allocated memory and you will also need to write an initialization routine that establishes the allocation policy.

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.
You will implement the following routines in your code

umalloc : has the same interface as malloc
ufree : has the same interface as free
Described in more detail below
umemdump : 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 you will not be writing a main() routine for the code that you hand in (but you should implement one for your own testing). Do not include a main function in your umem.c file; you should implement this only in your testing files.

We have provided the prototypes for these functions in the file umem.h. Download 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. Extra credit points can be earned for implementing a buddy allocator.
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 contunues 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: Your headers are limited to 32 bytes max. We will assume this size as maximum in our testing. This means that the header structures that you use in your embedded lists should not be larger than 32 bytes. You probably don't need 32 bytes for your headers, but we want to give enough room for a number of different solutions. What this means is that we will be testing your allocator with requests large enough that a 32 byte header size will not be a problem.

int ufree(void *ptr): ufree() 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: ufree() 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 umalloc().

void umemdump: This is primarily a debugging routine for your own use. Have it print the regions of free memory to the screen. The routine will not be tested extensively by script. However, sizes and addresses of blocks must be printed and will be checked.

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. Some small number of points may be based on the cleanliness of code, particularly with respect to functional decomposition.

Upload your umem.c file. We do not need your testing files, nor do we need you to upload umem.h