A1: Umalloc!

Due Nov 22, 2022 by 11:59pm **Points** 100 **Submitting** a file upload
Available Nov 6, 2022 at 10pm - Nov 23, 2022 at 6am

This assignment was locked Nov 23, 2022 at 6am.

Abstract:

From the system's point of view, memory is little more than a massive array of type 'byte'. All the constraints seen from the programmer's point of view like type, region you are allowed to access, stack vs heap vs global, are entirely a matter of policy.

In this project you will create a memory allocator using malloc() as the template. Since you will be allocating the memory, you will need to decide how it is organized, updated and manipulated. You will learn about some of the same organizational issues faced when implementing a memory allocator.

Introduction:

As a user you have very little influence over memory. You are locked solidly within your segment and page sizing and faulting is handled for you. In this case you will bulid a library that will load with a 10MB static char array and you will serve malloc() calls from that. Your library will be accompanied with a macro that will alter all malloc() calls to be calls to your library, so you will stand in for the system. This may sound intimidating, however inspect the interface to the calls you must implement:

void* malloc(size_t size)
Malloc takes an integer from 0 to MAX_INT and returns an untyped pointer.

void free(void*)
Free takes a pointer to dynamically-allocated memory and returns nothing.

Methodology:

>Note: Keep in mind that size you are implemeting malloc(), there should be no calls to malloc() anywhere in your code. You are the memory allocator.

In your library define:
- a static char array named mem that is 10MB long
- a char named init with a default value
- the function void* umalloc(size_t bytes)

the function void ufree(void* ptr)

You will need to first determine if you have been run already and are initialized with user data present, or if memory needs to be initialized. This is the purpose of init. The first thing umalloc should do is check init. If init is the default value, you must intialize memory, if not, presume memory has been initialized.

The act of allocating memory is fairly simple. You must return a pointer to a region of memory at least as large as what was requested. That region must not overlap with any other allocation. If that is not possible, then you return a NULL pointer. You are responsible for managing the space. Since you are the memory manager, your metadata must reside in the same space as all user data. Free()ing memory is updating your metadata to denote an allocated area is now unallocated and free to be used to serve another request - that is all.

A. Prototyping:

Before trying to support user code transparently you should implement your allocator as a standalone library where you call in to umalloc() and ufree() directly.

You first need to decide on a discipline for your metadata and how you will allocate. Your code will be evaluated for efficiency in allocation and metadata. You should be able to handle an allocation of any size from 1B up to nearly 10MB and you should also be able to support many small allocations.

You should implement a pure segmentation strategy and a first-fit selection algorithm:
represent your memory as one massive free block
on a request break it in to:
- an allocated region
(whose address is passed back)
- a free region
(whose size is that of the previous region minus the allocation, minus any new metadata)
on a free:
- mark the region pointed to as not in use
(check the adjacent regions, if either or both are free, meld them all in to one free block)

There are essentially two ways to handle metadata: all in one place, or per allocation unit. Each strategy has its own characteristics. Your goal is to keep your metadata as small as possible.

B. Basic Integration

Update your "umalloc()" and "ufree()" to have the following prototype:

void* umalloc(size_t, char*, int)
void ufree(void*, char*, int)

Add two macros to your .h file to replace all calls to "malloc(X)" and "free(Y)" with calls to "umalloc(X, __FILE__, __LINE__)" and "ufree(Y, __FILE__, __LINE__)"

Write a "memgrind.c" that implements the following:

0. Consistency:
   allocate a small block (1 to 10B), cast it to a type, write to it, free it
   allocate a block of the same size, cast it to the same type, then check to see if the address of the pointers are the same

1. Maximization: (simple coalescence)
   allocate 1B, if the result is not NULL, free it, double the size and try again
   on NULL, halve the size and try again
   on success after a NULL, stop
   free it

2. Basic Coalescence:
   allocate one half of your maximal allocation (see test 1)
   allocate one quarter of your maximal allocation
   free the first pointer
   free the second pointer
   attempt to allocate your maximal allocation - it should work
   free it

3. Saturation:
   do 9K 1KB allocations (i.e. do 9216 1024 byte allocations)
   switch to 1B allocations until malloc responds with NULL, that is your saturation of space

4. Time Overhead:
   saturate your memory (i.e. immediately after test 3)
   free the last 1B block
   get the current time
   allocate 1B
   get the current time, compute the elapsed time, that is your max time overhead

5. Intermediate Coalescence
   saturate your memory (i.e. immediately after test 4)
   free each allocation, one by one

attempt to allocate your maximal allocation - it should work
free all memory

C. Error Detection

Since you have a direct view of all memory, you can determine when certain operations are good and bad ideas. Enhance your ufree() to detect the following errors:

Free()ing addresses that are not pointers:

```
int x;
free( (int*)x );
```

Free()ing pointers that were not allocated by malloc():

```
p = (char *)malloc( 200 );
free( p + 10 );
    - or -
int * x;
free( x );
```

Redundant free()ing of the same pointer:

```
p = (char*)malloc(100);
free( p );
free( p );
    ... is an error, but:

p = (char *)malloc( 100 );
free( p );
p = (char *)malloc( 100 );
free( p );
... is perfectly valid
```

On discovery of an improper use of malloc() or free(), you should output a diagnostic error message using the input you get from __LINE__ and __FILE__:

e.g.: Error on free(): attempted to free() a value not from the allocatable region in line 47 of memtest.c

You should disambiguate between all of the errors above on free().

You should also detect and output a response from malloc() when:

there is no free memory

memory is not full but there is not enough free memory for the allocation
there is enough free memory, but there is no block large enough for the allocation
the amount requested makes no sense (i.e. is 0)

Results:
Be sure to fully comment your code.

Report:
Prepare a report.pdf with your partners' NetIDs.
Document your design and implementation details, including but not limited to your metadata structure and values determined from the tests in Part B.

Submission:
Prepare and submit a A1.tgz that holds:
umalloc.h
umalloc.c
Makefile
memgrind.c
.. any other source files needed that you wrote

Your Makefile should build your library code and memgrind with a "make all" and delete all files it created with "make clean".
Your Makefile should build your library as a compiled object.

We will inspect and test code for at least the maximization tests and may award additional points for optimal implementations.

Note:
I would strongly recommend you write a function to traverse your memory and 'pretty print' it.
e.g.:

```
Starting memory dump:
***
initialized
1048576B in use
9437184B free
***
0xAB0912, 2B, in use
0xAB0914, 10B, free
```

```
0xAB091D, 1026B, in use
... etc
```

This way you can verify the location your allocations start, how long they are and whether they are free or in use.

If and when bad things happen, not having a picture of memory will only make them harder to debug.