CPS 510 - Operating Systems -

courses.cs.duke.edu/fall22/compsci510/memory.html

Dynamic memory manager

For this project you will implement a basic heap manager. The standard C runtime library provides a standard heap manager through the malloc and free library calls. For this project you will implement your own version of this interface.

Recommended reading

DMM Lab

We use a different repo for the DMM lab. Please clone the repo via one of these addresses (depending on if you want HTTPS or SSH authentication):

```
$ git clone https://
OR
$ git clone git@gitl
Cloning into 'dmm-fa
```

You need to run the la docker we provide (the same xv6 docker

```
$ cd <PathToBaseRepo
$ docker run -it --r env
```

You only need to mod

To test your solution, p rint out “passed!” or “Fail”. You need to

To submit your solutio r `submission.zip` to gradescope.

Introduction

First you should understand the interface, purpose, and function of a heap manager. On any properly installed Unix system typing “man malloc” in a shell or terminal window will output documentation for the interface.

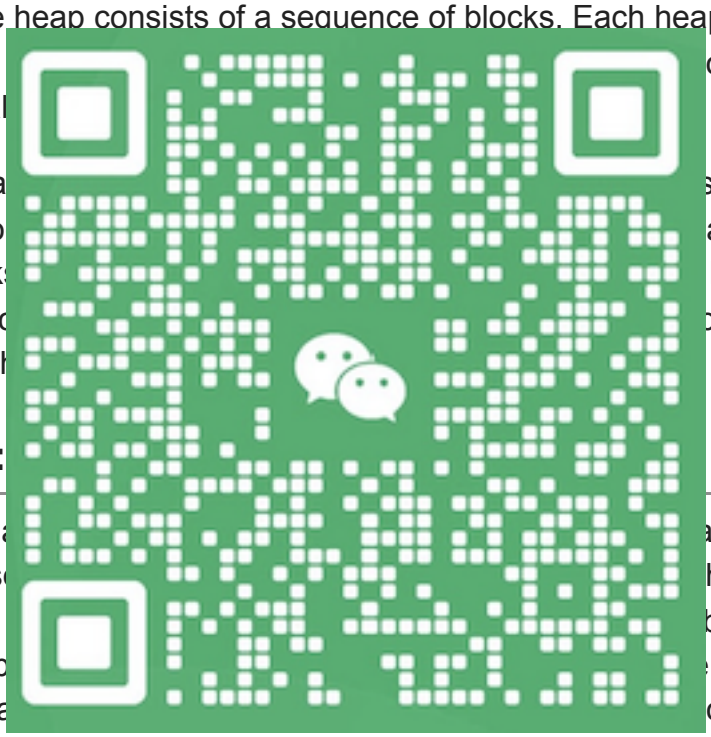
You will implement a the heap API operators dmalloc and dfree. From the perspective of the user program (the program using your heap manager), the behavior should be equivalent to the standard malloc and free. We have supplied some code to get you started, including a header file called dmm.h. Please use that header file and API in your solution, and do not change it.

The lab is designed to build your understanding of how data is laid out in memory, and how operating systems manage memory. Your heap manager should be designed to use memory efficiently: you should understand the issues related to memory fragmentation, and some techniques for minimizing fragmentation and achieving good memory utilization.

As a side benefit, the lab will you get better at system programming in the C language, and manipulating data structures “underneath” the type system. We strongly encourage you to start early and familiarize yourself with C and its pitfalls and with the C/Unix development environment. You will need to know some basic Unix command line tools: man, cd, ls, cat, more/less, pwd, cp, mv, rm, diff, make, and an editor/IDE of some kind. Also, debugging will go much more easily if you use a debugger such as gdb, at least a little.

Dynamic memory allocation

At any given time, the heap consists of a sequence of blocks. Each heap block is a contiguous sequence one heap block. Each block is either al



Heap blocks are varia s blocks are allocated (with dmallo anager splits and coalesces heap block heap manager must be careful to trac llowing subsections discuss th

Block metadata:

The heap manager pl ap space. A block’s header repres he block’s size. In general “metadata” is but it is not user data. The rest of the b user program does not see the meta of the heap manager.

The code we provide defines metadata_t as a data structure template (a struct type) for the block headers. The intent is that each heap block will have a metadata_t structure at the top of it, whether the block is allocated or free. The metadata_t structure is defined in dmm.c as:

```
typedef struct metadata {

  size_t size;
  struct metadata* next;
  struct metadata* prev;

} metadata_t;
```

The block header is useful for two reasons. First, a block's header indicates whether the block is allocated or free. A heap manager must track that information so that it does not allocate the same region or overlapping regions of memory for two different dmalloc() calls by accident.

Second, the block headers help track and locate the borders between heap blocks. This makes it possible to coalesce free heap blocks to form larger blocks, which may be needed for later large dmalloc requests. The supplied metadata structure makes it easy to link the headers of the free blocks into a list (the free list).

There are many ways to implement a heap manager. The most efficient schemes also place a footer at the end of each block. You may use footers if you wish, but they are not required.

Initialization: sbrk

When a heap manager is initialized it obtains a large slab of virtual memory to carve up into blocks. It does thi system kernel using a system call, e

The supplied code us ddress space. The sbrk system call c eviously unused (and invalid) to be reg the region and zero-fills each page of ter to the new region, i.e., the "slab".

Initially the heap cons ntire slab. The supplied code casts th s it in a global pointer variable called

```
metadata_t* heap_reg
```

The heap_region poin ck in the heap, which initially is a free nal slabs as needed. For project 0 we limit the number of sbrk calls to HEAP_SYSCALL_LIMIT for evaluation purposes. The default value is 1.

API: dmalloc and dfree

All of the heap blocks you allocate with dmalloc() should come from the one initial slab. Whenever dmalloc allocates a block, it returns a pointer to the block for use by the client application. The returned pointer should skip past the block's header, so that the program does not overwrite it. The returned pointer should be aligned on a long-word boundary. Be sure that you understand what this means and why it is important by reading this.

The supplied code includes some macros to assist you in dmm.h. It also makes it easy to keep track of the available space using a doubly linked list of headers of the free heap blocks, called a freelist. At the start of the program, the freelist is initialized to contain a single large block, consisting of the entire slab pointed to by heap region.

Splitting a free heap block

It is often useful to split a free heap block on a call to dmalloc. The split produces two contiguous free heap blocks of variable size, within the address range of the original block before the split. You must implement a split operation: without splitting, the heap could never contain more than one block.

For a split, we first need to check whether the requested size is less than space available in the target block. If so, the most memory-efficient approach is to allocate a block of the requested size by splitting it off of the target block, leaving the rest of the target block free. The first block is returned to the caller and the second block remains in the freelist. The metadata headers in both blocks must be updated accordingly to reflect their sizes.

Freeing space: c

A client program frees g a pointer to the block to free. The hea omes available for use by a future dmallc the freelist.

As blocks are allocate that are both free. In that case, it is single contiguous heap block. This is ca euse freed space as part of a future bloc that fills its heap with small blocks coul f the heap memory. We say that in small blocks.

You have a couple of esce as you exit the dfree() call. Secon you are no longer able to find a sufficiently large block to satisfy a call to dmalloc.

One optimization we can perform is to keep the freelist in sorted order with respect to addresses so that you can do the coalescing in one pass of the list. For example, if your coalescing function were to start at the beginning of the freelist and iterate through the end, at any block it could look up its adjacent blocks on the left and the right (“above” and “below”). If free blocks are contiguous/adjacent, the blocks can be coalesced.

If we keep the freelist in sorted order, coalescing two blocks is simple. You add the space of the second block and its metadata to the space in the first block. In addition, you need to unlink the second block from the freelist since it has been absorbed by the first block.

Logistics

We recommend that you first implement dmalloc with splitting. Test it. Then implement dfree by inserting freed blocks into a sorted freelist. Test it, and be sure you can recycle heap blocks through the free list. Then add support for coalescing to reduce fragmentation. Make sure you pass all local tests first.