Lab 0 (p0):
Dynamic Memory: Heap Manager

1 Introduction

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

First you should understand the interface, purpose, and function of a heap manager. On any properly installed Unix system you should be able to type *man malloc* to a shell or terminal window, which will output documentation for the interface. (If you are using your own computer, follow the instructions to install a Unix/C development environment first.) You can also type *man malloc* into Google. You can also learn about heap managers from the readings on the course web.

You are asked to provide your own implementation of these heap API operators called `dmalloc` and `dfree`. From the perspective of the *user program* (the program using your heap manager in any given process), 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 that 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 exposes you to system programming in the C programming language, and manipulating data structures "underneath" the language and its type system. Many students find this difficult. We strongly encourage you to start early and familiarize yourself with C and its pitfalls. In particular, you must understand pointers and how they work within the virtual address space, and how to think about memory as a common storage pool for all of a program's data structures.

This lab also requires you to learn how to work with the C/Unix development environment, e.g., by playing around with the C examples on the course web. You also need to know some basic Unix command line tools: *man, cd, ls, cat, more/less, pwd, cp, mv, rm, diff* and an editor of some kind. Also, debugging will go much more easily if you use *gdb*, at least a little. See resources on the course web.

2 Dynamic Memory Allocation

At any given time, the heap consists of some sequence of *blocks*. Each heap block is a contiguous sequence of bytes. Every byte in the heap is part of exactly one heap block: the blocks are densely packed in the virtual memory, with no extra bytes between them. Each block is either allocated or free. The heap is "like a parking lot": some spaces are free and some are in use (i.e., they have cars parked in them), and the status changes as cars arrive and occupy space (malloc) and then depart (free).

The heap blocks are variably sized. The borders between the blocks shift as the application program allocates blocks (with `dmalloc`) and frees them (with `dfree`). In particular, the heap manager splits and coalesces heap blocks to satisfy the mix of heap requests efficiently. In doing this, your heap manager must be careful to track the borders and the status of each block. For example, your heap manager must ensure that no portion of any block is doubly allocated, i.e., in use as the result of more than one `dmalloc()` call at any given time. That is like two cars occupying the same space in a parking lot: the result is never good.

Now, there are many ways to build a heap manager. The source code and handout for this lab guide you toward a solution that is conceptually simple and easy to implement, but not as efficient as it could be. You are encouraged to understand its limitations and explore how to do it better. You are not restricted to use the structure summarized here, as long as your heap manager behaves correctly according to the heap manager "contract".

2.1 Block metadata: headers

The heap manager places a *header* at the start of each block of the heap space. A block's header represents some *metadata* information about the block, including the block's size. In general "metadata" is data about data: the header describes the block, but it is not user data. The rest of the block is available for use by the user program. The user program does not "see" the metadata headers, which are only for the internal use 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 has 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 contains the size of the data object or the amount
       * of free bytes
       */
        size_t size;
        struct metadata* next;
        struct metadata* prev;
} metadata_t;
```

The block headers let you track and locate the borders between heap blocks, since you know each block's starting address and size. The supplied metadata structure also makes it easy to link the headers of the free blocks into a list (the free list). Only the free blocks are linked into the list. The next/prev link pointers should be set to null for a block that is in use: that also enables your code to determine if a block is free or in use given a pointer to the block's header.

Your heap manager will operate on the block headers to *split* blocks to allocate space for a `dmalloc` request, and *coalesce* free heap blocks to form larger free blocks on `dfree`. To split a block you store a block header into some empty space in the middle of a free block, so that the block's storage becomes two smaller blocks each described by its own header. To coalesce two adjacent blocks you increase the size of the block at lower addresses so that it "swallows" the block at higher addresses and occupies its space. Splitting and coalescing are described below.

There are many ways to structure the metadata for a heap manager. The most efficient schemes also place a *footer* at the end of each block, with *prologue* and *epilogue* blocks at the start and end of the heap. You may use these if you wish, but they are not required.

2.2 Initialization: allocating a "slab" with a system call

When a heap manager initializes it obtains a large slab of virtual memory to "carve up" (by splitting) into heap blocks to store the individual data items or objects. It does this by requesting virtual memory using a special system call that exists for that purpose, e.g., `mmap` or `sbrk`. A system call is a protected call (trap) to the operating system kernel. The system call to allocate a slab causes a region of the virtual memory that was previously unused (and invalid) to be registered for use: the kernel sets up page tables for the region and arranges that each page of the region is zero-filled as it is first referenced. Then the kernel returns a pointer for the new region (the "slab") to the heap manager. Be sure that you understand these concepts.

Initially the heap consists of a single free heap block that contains the entire slab. Like every block, it starts with a metadata header. The supplied code casts the slab address to a pointer to a `metadata_t` struct and places it in a global pointer variable called `heap_region`, for example:

```
metadata_t* heap_region = (metadata_t*) sbrk(MAX_HEAP_SIZE);
```

Thus the `heap_region` pointer references the header for the first (and only) block in the heap, which is a free block, and is of a type that you can use to manipulate the block's header. Be sure that you understand these concepts: see the reading on C pointers and type casting.

You can control the size of your heap slab by changing the value of MAX_HEAP_SIZE. It will be useful to test your heap manager on small heaps so that you can verify that it works correctly when it runs out of memory, i.e., when the heap is full of allocated blocks. The heap manager cannot satisfy a `dmalloc` call when it is out of free memory. Such a call is not an error by the user (application) program or a failure condition. Rather, `dmalloc` must return null: both your heap manager and your test programs should handle this case appropriately. A "real" heap manager may obtain additional slabs from the kernel as needed. For this lab we limit you to `HEAP_SYSCALL_LIMIT` slabs. The default value is 1. All of the heap blocks you allocate with `dmalloc()` should be carved out of the one initial slab.

2.3 Heap Manager API: dmalloc and dfree

Whenever `dmalloc` allocates a block, it returns a pointer to the block for use by the user program. The returned pointer should "skip past" the block's header, so that the program does not try to use the header for its own data, which would likely cause it to accidentally overwrite the header. The returned pointer should be aligned on a longword boundary. Be sure that you understand what this means and why it is important.

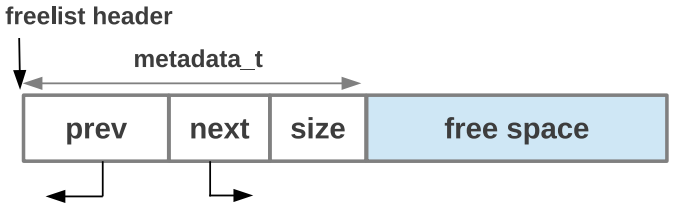

Figure 1: The initial state of the heap: a single block at the head of the `freelist`.

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`. Figure 1 shows the initial state of the heap, with the head of the freelist.

3 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 any desired size, within the address range of the original block before the split. You must implement a split operation: without an ability to split, 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` header in both the blocks is updated accordingly to reflect the sizes after splitting. Figure 2 shows the `freelist` after a split.

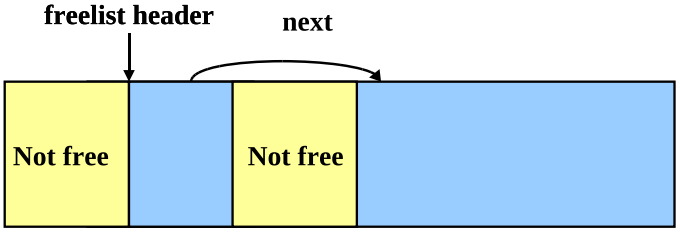

Figure 2: Structure of the `freelist` after a split.

4 Coalescing free space

The program frees an allocated heap block by calling `dfree()`, passing a pointer to the block to free. The heap manager reclaims the space, making it available for use by a future `dmalloc`. One solution is to insert the block back into the `freelist`.

As blocks are allocated and freed, over time you may end up with adjacent blocks that are both free. In that case, it is often useful and/or necessary to coalesce the adjacent free blocks to form a single contiguous free heap block. Coalescing makes it possible to reuse freed space as part of a future block of a larger size. Without coalescing, a program that fills its heap with small blocks could never allocate a large block, even if it frees all of the heap memory. We say that a heap is *fragmented* if its space is available only in small blocks.

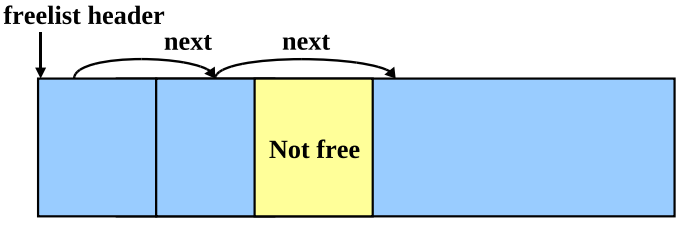

Figure 3: `freelist` after the first block is freed.

One optimization is to keep the `freelist` in sorted order w.r.t 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. See Figure 4.

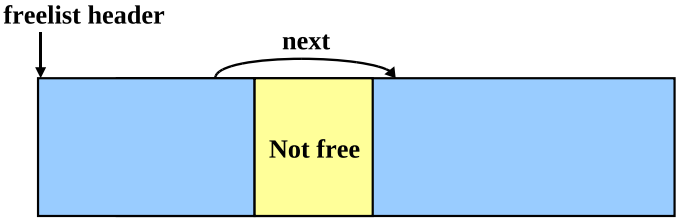

Figure 4: `freelist` after the first two blocks are coalesced.

5 Getting started

Fetch the source code files from the course web into a directory, cd into that directory, and type "make". Read this handout. Modify the code as directed in the handout. Type "make" again. Test by running the test programs. (Just type the name of the program preceded by ./.) Repeat. Use a debugger (e.g., *gdb*) to save time during debugging.

We recommend that you first implement `d`malloc with splitting. Test it. Then implement `d`free 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. Then consider alternative ways to make your heap manager more efficient.

6 Roadblocks and hints

You are encouraged to post your questions or issues to piazza. The instructor or the TAs will post tips that may point you and others in the right direction.

One general advice is to write code in small steps and understand the functionality of provided header files, macros, and code completely before starting the implementation. Use `assert` and `debug` macros aggressively. If you crash, use *gdb* to figure out where. It's not hard! (See the resources page on the course web.)

Think carefully about modularity and what procedures might be useful. In particular, try to avoid repeating code: do it once, do it right.

7 What to submit

Submit a single `dmm.c` file along with a `README` file. All projects should have a README saying at least the amount of time you spent on the lab, the list of group members who contributed, and a list of any other sources of assistance or source code. You should feel free to add remarks about your approach or your thoughts on the lab, but this is no longer required. It is likely that we will auto-grade your lab and assign a score without ever looking at the README. But the README is required. In particular, you must acknowledge any sources of assistance or code per the course policies.

As mentioned earlier, we will use the provided version of `dmm.h` for auto-grading so it is important that you do not change this header file, except maybe to change the `MAX_HEAP_SIZE` as needed for your own testing. The supplied code is intended to support a "simple" implementation of the heap manager: you don't have to do it that way, but if you need a different header structure please define it in your `dmm.c` file and not in the header file.

8 Grading

The grading is done on the scale of 100. Grading is done automatically with a "secret" stress testing program. We may provide you some test programs, but these are only to get you started and come with no warranties expressed or implied. You should figure out how to do your own testing.

In some semesters we give extra credit for "constant time" implementations that meet a minimum standard for space efficiency under a reasonable (secret) workload mix. Building a "constant time" heap manager requires a different approach and careful attention to detail, and a few tricks: see the discussion of heap managers in CS:APP [1]. Extra credit is at the discretion of the instructor and TA: ask.

References

[1] Randal E. Bryant and David R. O'Hallaron. *Dynamic Memory Management, Chapter 9 from Computer Systems: A Programmer's Perspective, 2/E (CS:APP2e)*. `http://www.cs.duke.edu/courses/compsci310/current/internal/dynamicmem.pdf`.