Lab 1: Dynamic Memory: Heap Manager

Introduction to Systems, Duke University

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 will expose you to system programming in the C programming language, and manipulating data structures "underneath" the type system. Many students find this to be quite difficult. We strongly encourage you to start early and familiarize yourself with the C and its pitfalls and with the C/Unix development environment, e.g., by playing around with the C examples on the course web. You will 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.

Follow the instructions on the lab webpage to submit your solution.

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. Each block is either allocated or free.

The heap blocks are variably sized. The borders between the blocks shift as blocks are allocated (with `dmalloc`) and freed (with `dfree`). In particular, the heap manager splits and coalesces heap blocks to

satisfy the mix of heap requests efficiently. In doing this, the heap manager must be careful to track the borders and the status of each block. The following subsections discuss the implementation in more detail.

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 provided 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 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 header is useful for two reasons. First, a block's header represents 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 to 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.

2.2 Initialization: The sbrk system call

When a heap manager initializes it obtains a large slab of virtual memory to "carve up" into heap blocks to store the individual items or objects. It does this by requesting virtual memory from the operating system kernel using a system call, e.g., `mmap` or `sbrk`. This is described in the various text resources on the course web. The supplied code uses `sbrk` to extend the data segment of the virtual address space. The `sbrk` system call 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 referenced. Then `sbrk` returns a pointer to the new region (the "slab").

Initially the heap consists of a single free heap block that contains the entire slab. 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`:

```
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.

A "real" heap manager may obtain additional slabs as needed. For this lab we limit the number of `sbrk` calls to `HEAP_SYSCALL_LIMIT` for evaluation purposes. The default value is 1.

2.3 Heap Manager 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 application program. The returned pointer should "skip past" the block's header, so that the program does not 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.

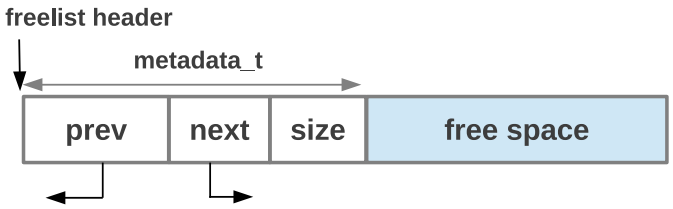

Figure 1: The initial state of the heap: a single block at the head of the `freelist`.

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.

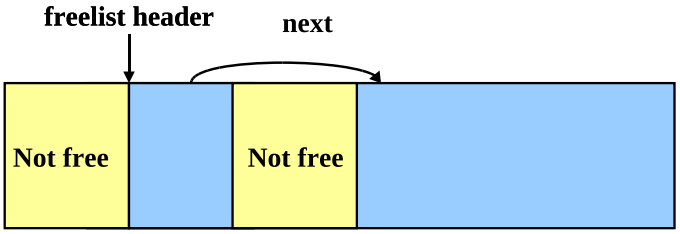

Figure 2: Structure of the `freelist` after a split.

4 Freeing space: coalescing

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 just insert the block back into the `freelist`.

As blocks are allocate 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 combine the adjacent blocks into a single contiguous heap block. This is called *coalescing*. 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.

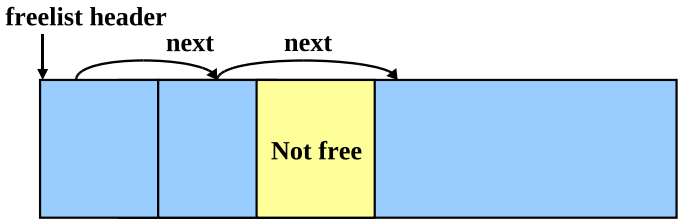

Figure 3: `freelist` after the first block is freed.

You have a couple of options to perform coalescing: First, you can coalesce as you exit the `dfree()` call. Second, you can periodically or explicitly invoke the coalesce function when 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 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.

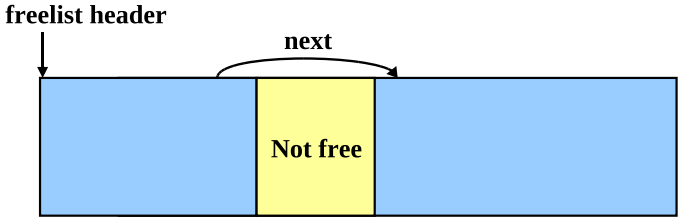

Figure 4: `freelist` after the first two blocks are coalesced.

5 Getting started

Fetch the source code files from the course web into a directory, cd into that directory, and type "make". Read the 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. You can debug/execute all the test cases by running "make debug" and "make test".

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.)

7 What to submit

In `dmm.c`, you implement two functions:

1. `dmalloc()`
2. `dfree()`

according to the API in `dmm.h`. Submit a single `dmm.c` file along with a `README` file documenting the implementation choices you made (more in section 8), the results, the amount of time you spent on the lab, and also include your name along with the collaborators involved. Use the websubmit to submit your code as directed on the course web. Submission instructions will be announced on the course page before the deadline.

You may implement the code any way you like. In particular, you may either coalesce when you run out of space or when `dfree()` is terminating. It must behave in the same way as `malloc()` does.

As mentioned earlier, we will use the provided version of `dmm.h` 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.

In addition, we have provided supplementary files to get you started: `Makefile` contains cases for `compile`, `debug`, and `test`. `test_basic, test_coalesce, test_stress1, test_stress2` contain test cases scenarios which can be helpful while debugging. The stress test cases generate variable sized objects and will randomly call `dmalloc` and `dfree` functions. Note: Passing `test_basic, test_coalesce` does not mean your code is functional. You should aim to pass `test_stress2` with high success rate. You may want to know that the final test cases need not be the same cases provided to you.

8 Grading

The grading is done on the scale of 100 and you have an opportunity to earn 25 extra points.

- Correctness: 50 points

- Efficiency: 25 points (improve the efficiency using first-fit/best-fit algorithms using the `metadata` structure provided in the `dmm.c` file; efficiency is measured as a combination of utilization and throughput as provided in [1]; with the provided metadata structure you should at least see a success rate above 80%)
- Readability: 10 points (comment your code where necessary)
- README: 15 points (Your README should contain evaluation of the choices you made in the design of your heap manager. For example, it should contain about the list of information you store in the `metadata` structure, how the `coalesce`, `free`, and `split` operations benefited from the `metadata` structure, the space and time overheads and tradeoffs, the results of your test cases, the amount of time you spend on the lab, your name and any collaborators involved, references to any additional resources used, and your feedback on the lab).
- Extra credit: 25 points (With address order sorting, the efficiency of all operations is O(#free blocks). Improve the efficiency of `free` and `coalescing` operations to a constant time, i.e., O(1). Useful reference [1])

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`.