Lab 1: Dynamic Memory Management
COMPSCI 210: Introduction to Operating Systems

1 Introduction

For this lab, you implement a basic heap manager, which is memory management wrapper using the existing system call API (application programming interface). Your implementation should mimic the standard dynamic memory operators `malloc` and `free` found in `libc` library by providing your own implementation, and your version of the operators are called `dmalloc` and `dfree` as defined in `dmm.h`[1]. You must use the header file we provided, `dmm.h`, in your solution. If you change the header file, your solution however functional, cannot be evaluated correctly using autograder tools; hence, you will not receive any credit. Learning to rely on abstractions or APIs provided by a 3rd party is a necessary skill for implementing any systems project.

The lab is designed to introduce you to the low-level concepts such as pointers, linked lists, and issues related to memory alignment using C programming. Manipulating pointers and managing memory—explicitly—are major sources of difficulty while programming in C, especially if you have programmed only in automatic garbage-collected languages such as Java before. At a high level, you learn about heap managers and the challenges they face, how they are used, and their interaction with the operating system kernel. Your heap manager should be designed to use memory efficiently: you need to understand the issues related to memory fragmentation, some techniques for minimizing fragmentation and achieving good memory utilization. The goal of the assignment is really to understand these issues, rather than C programming per se (although learning C is part of it).

To help you get started, we have provided some source code. You may find OSTEP and CSAPP sections on dynamic memory management helpful [6, 4, 5]. We have also listed several freely available material for learning C/C++ [7, 8, 9]. For this lab, we recommend C; however, if you have any prior experience with C++, you can implement using C++. Check the `Makefile` for the changes necessary to compile your C++ code.

You work *independently* for this lab. Submit the code using the web tool at [1]. Once you open the link, choose the course label as "compsci210" and then the lab label as "lab1". You can submit the files multiple times before the deadline and we will consider only the last submission for final grading. Remember to include README in the mandated format as well as all the relevant files in your final submission. Any violations to the submission policy will be automatically given zero score.

Start early!

2 Dynamic Memory Allocation

Managing data items in C can be categorized into three types: static, automatic, and dynamic. Static storage and automatic storage are allocated on stack. The static variables persist for the program lifetime,

[1] The lab names are commonly prefixed with `d`, which—as you might have guessed—can be read as `duke/devil`, or with other appropriate connotation: for e.g., `dynamic` in this context.

i.e., at launch until exit, whereas automatic variables persist within a code block: for e.g., variables declared in a function will be reclaimed when the function exits. Automatic variables are convenient for the programmer if the data is no longer required on a function exit. Finally, the dynamic storage is allocated from a heap, and the life of a data item starts with a `malloc` call and persists until `free` is called.

For this lab, you implement a simplified version of a heap allocator, which may not be the most efficient compared to what is provided by `libc`. However, your implementation will be complete—i.e., the `dmalloc` and `dfree` which you implement will exactly function like the standard implementation of `malloc` and `free`. `malloc` is actually implemented using the system calls `mmap` and `sbrk` internally. For this lab, you use the `sbrk` system call. To simplify debugging, we fetch a large chunk of continuous (virtual) memory using `sbrk` (invoking `sbrk` is an expensive operation) into a buffer called `heap_region`. In your program, you need to allocate this buffer as an array of bytes, the size of which is defined by `MAX_HEAP_SIZE`, aligned accordingly. This array should be defined globally and not in the local functions which you implement. For example,

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

will allocate a region of program's address space of size `MAX_HEAP_SIZE` and assign it to the `heap_region`. All of the space you allocate with `dmalloc()` should come from this region. Whenever you allocate a block, you need to ensure that you are returning a pointer that is memory aligned.

`metadata_t` is the structure that keep the header information. The metadata is useful for two reasons. First, metadata will need to keep track of free and allocated memory so that you do not allocate the same region of memory to two different `dmalloc()` calls. Second, as free blocks are given away and then released (via `dfree()`), your buffer will become *fragmented* as parts of it remain allocated and have yet to be freed. You will store the list of available free blocks and perform periodic *coalescing* to avoid excessive fragmentation. You learn about fragmentation and coalescing in the next sections.

One way to define metadata is to augment each block with its size information. We also will store pointers that are helpful to reach the neighboring blocks. In the code we provided, 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;
```

Figure 1 shows an allocated block along with the metadata information. See Bryant and O'Hallaron [6], Chapter 9 for more efficient structure. You are encouraged to improve the metadata structure for better efficiency. In addition, you can earn extra credit for implementing all the operations in constant time, i.e., O(1).

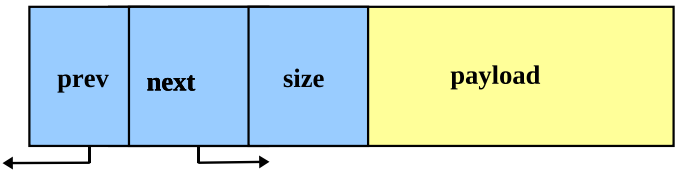

Figure 1: Structure of an allocated block.

You initialize the `freelist` structure from the `heap_region`. You have two choices during initialization. You can append prologue and epilogue blocks to the start and the end of the `freelist`, or initialize

the `freelist` pointers to `NULL`.

3 Splitting the freelist

Now that we initialized the `freelist` to a large contiguous chunk of memory, we need to split the chunk when a process requests memory through `dmalloc`. First, we need to check whether the requested size is less than space available, and if so, we need to split `freelist` into two blocks: The first block is assigned to the requested process moving the header of the `freelist` to the second block, which is free. Figure 2 shows `freelist` after a split. The `metadata` in both the blocks is updated accordingly to reflect the size after splitting.

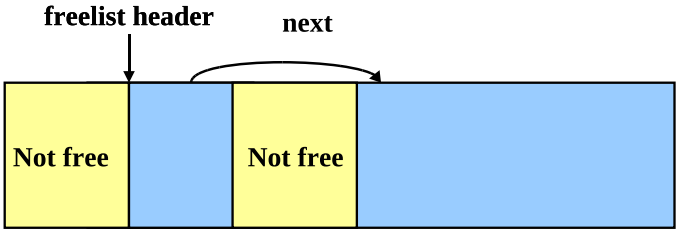

Figure 2: Structure of `freelist` after a split.

4 Freeing space

The allocated space can be freed by a call to `dfree()`. You must reclaim the space by appropriately inserting the block into the `freelist`. For example, consider the case when an allocated block is freed. The function `dfree()` must link that block into the `freelist`. See Figure 3.

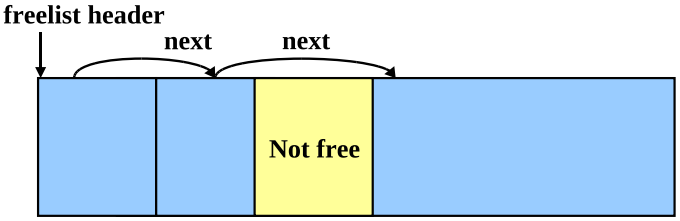

Figure 3: `freelist` after the first block is freed. The header should now point to the first block.

5 Coalescing free space

As the blocks will be freed, over a time you may end up with adjacent blocks that are free and contiguous. In other words, the free space can be fragmented and although you have requested memory in sum, you may not be able to allocate the memory due to fragmentation. You need to do periodic cleanup by merging the contiguous free blocks to form a large contiguous block. The process is called coalescing.

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 no longer able to find the requested memory.

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. If the two blocks are contiguous, the blocks can be coalesced. What are the other possible cases for coalescing?

If we keep the `freelist` in sorted order, coalescing two blocks is simple. You choose one block to absorb the other. If your `freelist` is sorted smallest address to the biggest, choosing the block occurring earlier in the list is a better choice. If you do, then the procedure is to 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 (the one you have removed) from the `freelist` since its space has been absorbed by the first block. Notice that by choosing the first block as the absorber, you can now consider your new list without starting over at the beginning. That is, you can look at your new big block and the block that comes after it on your new free list to see if you can do any further merging. If you cannot, you move on down the list. See Figure 4.

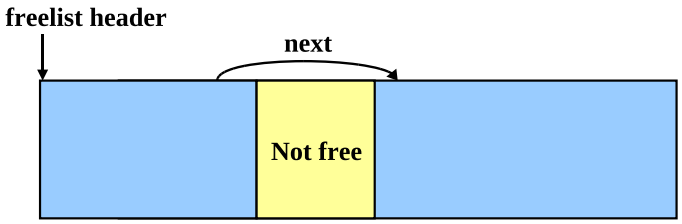

Figure 4: `freelist` after the first two blocks are coalesced.

6 Finding the right fit for the requested bytes

Finding the right fit is hard since it requires you to predict the pattern of *future* requests. For example, suppose after n requests all the blocks are exactly four words in size. Does this heap suffer from fragmentation? The answer depends on future requests. Check the reference [6] for more material on this topic.

While, there is lot of research done in finding the right fit algorithms, we recommend to start with the simplest choice: the first fit algorithm, which essentially traverses the `freelist` from the head until it finds the spot. Once you are done with first fit and you have a working code, it is easier to implement best fit or next fit algorithms.

7 Getting started

The source code is available at `http://www.cs.duke.edu/courses/spring13/compsci210/projects/lab1/`. Copy the source code files 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".

8 Roadblocks and hints

You are encouraged to post your questions or issues to piazza [2]. 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.

9 What to submit

First, fetch the source code available at [3]. 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 10), the results, the amount of time you spent on the lab, and also include your name along with the collaborators involved. Use the webtool [1] to submit your code by selecting the course as "compsci210" and then the lab label as "lab1". You can submit the files multiple times before the deadline and we will treat the latest submission as the final submission for grading.

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 in any way. If your code depends on a change to this header file and would not work otherwise, it is view as non functional.

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 the same cases which were provided to you.

10 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 [6]; 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 (improve the efficiency of `free` and `coalescing` operations to a constant time, i.e., O(1). Useful reference [6])

References

[1] Lab submission page. `https://www.cs.duke.edu/csed/websubmit`.

[2] Piazza discussion page. `https://piazza.com/class#spring2013/compsci210`.

[3] Source repository for lab1. `http://www.cs.duke.edu/courses/spring13/compsci210/projects/lab1/`.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Free-Space Management, Chatper 16 from Operating Systems: Three Easy Pieces*. `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf`.

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Interlude-Memory API, Chatper 13 from Operating Systems: Three Easy Pieces*. `http://pages.cs.wisc.edu/~remzi/OSTEP/vm-api.pdf`.

[6] 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/spring13/compsci210/internal/dynamicmem.pdf`.

[7] Mark Corner. 377 student guide to c++, 2005. `http://people.cs.umass.edu/~emery/classes/cmpsci377-fall2005/papers/intro_to_c++.pdf`.

[8] Tim Love. Ansi c for programmers on unix systems, 1996. `http://www.cs.duke.edu/courses/spring13/compsci210/internal/LoveC.pdf`.

[9] Nick Parlante. A brief introduction to c programming, 2003. `http://www.cs.duke.edu/courses/spring13/compsci210/internal/essentialC.pdf`.