ITP 30002-{01, 02} Operating Systems, Spring 2023

Homework 3. `bmalloc`: buddy heap memory allocator

Shin Hong
hongshin@handong.edu

1. Introduction

This homework asks you to construct `bmalloc`, a heap memory allocation library using a buddy algorithm. To serve a user's request for allocating *m* bytes, a buddy algorithm creates a new page using `mmap()`, and then partitions the page into multiple blocks such that the size of each block is a power of two bytes (e.g., 16 bytes, 32 bytes, 64 bytes). A buddy algorithm prevents external fragmentation issues by providing fixed-size memory to each memory allocation request and merging adjacent unused memory blocks into a larger one (Section 2).

A specific description of a buddy algorithm (Section 2.1) and baseline code, `bmalloc.c`, are given for you to implement the buddy algorithm as a C library (Section 2.2). You need to complete the missing parts in `bmalloc.c`, and then demonstrate your implementation works as expected.

This homework is for teamwork: each member must work together with the other and takes an equal amount of work for all activities of the homework. All your results must be submitted by 9 PM, May 12 (Fri).

2. Buddy memory allocation

2.1. Algorithm

A buddy memory allocation is a free memory management technique that recursively divides a continuous memory region into smaller blocks to serve a memory allocation request. A block consists of the *header* where the block metadata is written, and the *payload* where a requested memory will be allocated. Once a used block is reclaimed, a buddy algorithm merges unused blocks into a larger one, if possible, to prepare for future demands for memory allocation.

Specifically, the buddy memory allocator for this homework must work as follows:

- The allocator can add a new page of a fixed size if more memory is needed.
- Initially, a whole page is defined as a single block.
- A block can be divided into two blocks of the same size,
- For a *m*-bytes allocation request, the allocator must provide a block of the smallest size that can accommodate *m*-bytes in its payload (calling it *fitting* block). If such a fitting block is available, the algorithm uses the existing block. Otherwise, the algorithm gets a fitting block by dividing a larger block and uses it.
- when a used block is becoming free (i.e., released), the allocator merges the block with its sibling block (i.e., another block created together) if the sibling block is free. Merging must be cascaded up to restore a block as large as possible.
- All blocks are forming a linked list, such that every block can be reached if one iterates over the linked list.

Note that the buddy memory allocator provides certain fixed-size blocks because a block can be derived only by halving an existing block. Consequently, the buddy memory allocator overuses memory when the payload of the smallest feasible block is larger than the requested memory.

2.2 The `bmalloc` library

The `bmalloc` library implements aforementioned buddy memory allocator for managing heap memory on a 64-bit x86 running Linux. You can find the baseline code of `bmalloc` at `https://github.com/hongshin/OperatingSystem/tree/hw3`. `bmalloc` has the followings APIs as declared in `bmalloc.h`:

- `void * bmalloc (size_t s)`: allocates a buffer of `s`-bytes and returns its starting address.
- `void bfree (void * p)`: free the allocated buffer starting at pointer `p`.
- `void * brealloc (void * p, size_t s)`: resize the allocated memory buffer into `s` bytes. As the result of this operation, the data may be immigrated to a different address, as like `realloc` possibly does.
- `void bmconfig (bm_option opt)`: set the space management scheme as `BestFit`, or `FirstFit`.
- `void bmprint ()`: print out the internal status of the block list to the standard output.

The followings are the design, the expected behaviors, and the assumptions of `bmalloc`:

- When memory is needed, `bmalloc` allocates a new page using `mmap() with MAP_ANONYMOUS`. A page size must be always 4096 ($=2^{12}$) bytes.
- The size of block is either 16 bytes, 32 bytes, 64 bytes, 128 bytes, 256 bytes, 512 bytes, 1024 bytes, 2048 bytes, or 4096 bytes. Total 9 different size options exist.
- The `bm_header` structure defines the header of a block. This header must be placed at the beginning of a block, and the payload must begin right after the header. A `header` object has one byte to contain the `used` field and the `size` field. `used` is `1` if and only if the block is in-use, otherwise `0`. `size` is the exponent of the block size when base is 2. For example, if the block size is 4096, size is defined as `12`. The `next` field points to the address where the immediate next block begins, or `NULL` if the next block does not exit.
- `bmalloc` uses a fitting block for given `s` if there exists. If a fitting block does not exist, `bmalloc` may behave differently based on `bm_mode`. When `bm_mode` is `BestFit` (by default), among unused blocks larger than the fitting block size, `bmalloc` selects and divides a smallest block among all unused blocks. When `bm_mode` is `FirstFit`, `bmalloc` selects the first feasible block in the linked list (i.e., via `next`), and divides it to get a fitting block.
- `bfree` switches the used block to unused, and then merge it with its sibling if both are unused, and this step must be repeated with the merging result upward, if possible. `bfree` unmaps (releases) a page if whole page becomes unused.

[*] Last update: 12 May, 2023

- `bmprint` display the statistics and the status of each block along the linked list. As statistics, `bmprint` must show (1) the total amount of all given memory, (2) the total amount of memory given to the users (i.e., total amount of used blocks), (3) the total amount of available memory, ~~(4) total amount of the internal fragmentation~~. For each block, `bmprint` must show whether it is used or unused, the size of the block, and the size of the payload. Note that `bmprint` will be used for testing your program in the evaluation.
- Beside the API functions, `bmalloc` must have helper functions that other API functions can use for convenience. `void * sibling (void * h)` returns the header address of the suspected sibling block of `h`. Note that `sibling(h)` may not be the sibling of h when their block sizes are different. `int fitting(size_t s)` returns the `size` field value of a fitting block to accommodate `s` bytes.

2.3 Example

Suppose that a user program uses `bmalloc` as follows:

```
1:  #include <stdio.h>
2:  #include "bmalloc.h"

3:  int main () {
4:       void *p1, *p2, *p3, *p4 ;

5:       p1 = bmalloc(1000) ;
6:       p2 = bmalloc(2000) ;
7:       bfree(p1) ;
8:       p3 = bmalloc(2500) ;
9:       p4 = bmalloc(120) ;
10:      bfree(p2) ;
11:      bfree(p3) ;
12:      bfree(p4) ;  }
```

Initially, `bmalloc` does not allocate any page. As a response to Line 5, a page is allocated, and then the page is divided into two 1024-byte blocks and one 2048-byte block, and subsequently, a 1024-byte block is used to allocate the requested 1000 bytes.

For Line 6, `bmalloc` uses the 2048-byte block to allocate the 2000 bytes. After executing Line 6, among given 4096 bytes, 3000 bytes are given to the user, 1015 bytes are available for the user, and the total internal fragmentation is 54 bytes.

As `p1` is released at Line 7, `bmalloc` merges two 1024-byte blocks and create one 2048-byte used block. For Line 8, since the current page has no room to allocate 2500 bytes, `bmalloc` allocates another page and then create a 4096-byte block and allocates 2500 bytes from its payload.

Finally, to allocate 120 bytes as requested by Line 9, `bmalloc` divides the merged 2048-byte block into one 1024-byte block, one 512-byte block and two 256-byte blocks, and then uses one 256-byte block to accommodate 120 bytes.

3. Tasks

3.1. Programming

Fill out the missing definitions of the functions in `bmalloc.c` to fulfill all functionalities given at Section 2. In the implementation, it is strictly prohibited to modify `bmalloc.h`. Note that your program will be desk rejected at evaluation if any content of these immutable parts were changed. Your program must be built and run successfully on the peace server (`peace.handong.edu`) because your program will be tested on this environment.

3.2. Report writing

You must write a report on your homework results within 3 pages of the given template. Your report is expected to show the followings:

- Explanations on the logics of important operations
- Demonstration of multiple test scenarios that different aspects of your program work correctly
- Discussion on your results including challenges you have faced and/or questions you have bear in in doing this homework, new ideas for improvements, etc. You can achieve extra points if you write interesting discussions.

Evaluation will be primary based on your report, thus, try best to deliver your results via your writing. The comprehensiveness of your demonstration will be carefully checked to see how many requirements were properly addressed in your solution. In addition, your program will be run against new test cases to see if it behaves as expected, and as consistently as your report.

4. Submission Instruction

Upload a zip file to HDLMS, containing all results including all files of your implementation and your report.

Together with `bmalloc.c`, your submission must contain test cases that you used in the report, a build script (e.g., `Makefile`) and documentation (e.g., `README.md`) on how to build and run the program. Your report must be in PDF for compatibility. The submission deadline is **9 PM, Mon 15 May**. This is a strict deadline, and no late submissions will be accepted.