CS 551 Systems Programming, Summer 2024
Programming Project 1

Out: 6/29/2024 Sat.
Due: 7/13/2024 Sat. 23:59:59

In this project your are going to implement a custom memory manager that manages heap memory allocation at program level. Here are the reasons why we need a custom memory manager in C.

- The C memory allocation functions `malloc` and `free` bring performance overhead: these calls may lead to a switch between user-space and kernel-space. For programs that do frequent memory allocation/deallocation, using `malloc`/`free` will degrade the performance.
- A custom memory manager allows for detection of memory misuses, such as memory overflow, memory leak and double deallocating a pointer.

The goal of this project is to allow students to practice on C programming, especially on bitwise operations, pointer operation, linked list, and memory management.

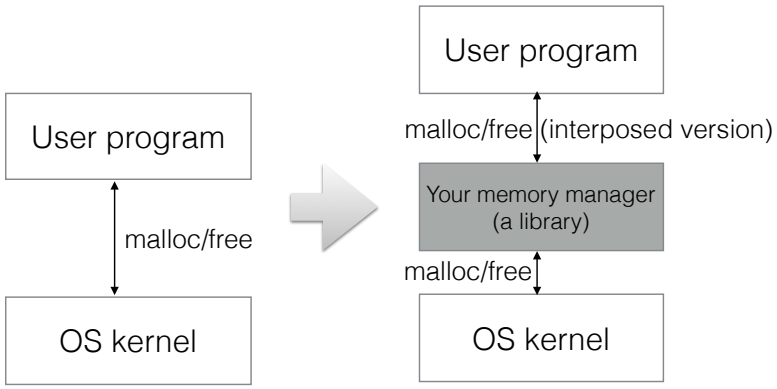

Figure 1: Overview

1 Overview

Figure 1 depicts where the memory manager locates in a system. The memory manager basically preallocates chunks of memory, and performs management on them based on the user program memory allocation/deallocation requests.

We use an example to illustrate how your memory manager is supposed to work. Suppose memory allocations in the memory manager are 8 bytes aligned[1].

- After the user program starts, the first memory allocation from the program requests 5 bytes of memory (`malloc`(5)). Prior to this first request, your memory manger is initialized with

[1]In other words, the memory manager always allocates memory that is a multiple of 8 bytes. In the base code, this is controlled by the `MEM_ALIGNMENT_BOUNDARY` macro defined in `memory_manager.h`

a batch of memory slots[2], each of which has a size of 8 bytes. Memory manager returns the first slot of the batch to the user program.

- Then the user program makes a second request (`malloc(6)`). Because the memory allocation is 8 bytes aligned, the memory manager should return a chunk of memory of 8 bytes. This time, because there are still available 8-byte-slots in the allocated batch, the memory manager simply return the second slot in the allocated batch to fulfill the request without interacting with the kernel.

- The user program makes 6 subsequent memory requests, all of which are for memory less than 8 bytes. The memory manager simply returns each of the rest 6 free 8-byte-slots to fulfill the requests. And for the implementation of this project, assume you will only get requests for less than or equal to 8 bytes memory. To get bonus of this project, you have to handle the larger size request which is explained in the Bonus section.

- The user program makes the 9th request (`malloc(7)`). Because there is no free slots available, the manager allocates another batch of memory slots of 8 bytes, and returns the first slot in this batch to the user program.

- The memory manger organizes memory batches that have the same slot size using linked list, and the resulting list is called a memory batch list, or a memory list.

- The manger uses a bitmap to track and manage slots allocation/deallocation for each memory batch list.

It is easy to see that with such a mechanism, the memory manger not only improves program performance by reducing the number of kernel/user space switches, but also tracks all the memory allocation/deallocation so that it can detect memory misuse such as double freeing. The memory manager can also add guard bytes at the end of each memory slot to detect memory overflow (just an example, adding guard bytes is not required by this project.)

2 What to do

2.1 Download the base code

Download the base code from Assignments section in the Brightspace. You will need to add your implementation into this base code.

2.2 Complete the bitmap operation functions (25 points)

Complete the implementation of the functions in `bitmap.c`.

- **int bitmap_find_first_bit(unsigned char * bitmap, int size, int val)**

  This function finds the position (starting from 0) of the first bit whose value is "val" in the "bitmap". The val could be either 0 or 1.

  Suppose

[2]In the base code, the number of memory slots in a batch is controlled by the `MEM_BATCH_SLOT_COUNT` macro defined in `memory_manager.h`

```
unsigned char bitmap[] = {0xF7, 0xFF};
```

Then a call as following

```
bitmap_find_first_bit(bitmap, sizeof(bitmap), 0);
```

finds the first bit whose value is 0. The return value should be 3 in this case.

- **int bitmap_set_bit(unsigned char * bitmap, int size, int target_pos)**

  This function sets the "target_pos"-th bit (starting from 0) in the "bitmap" to 1.

  Suppose

  ```
  unsigned char bitmap[] = {0xF7, 0xFF};
  ```

  Then a call as following

  ```
  bitmap_set_bit(bitmap, sizeof(bitmap), 3);
  ```

  sets bit-3 to 1. After the call the content of the bitmap is {`0xFF, 0xFF`};

- **int bitmap_clear_bit(unsigned char * bitmap, int size, int target_pos)**

  This function sets the "target_pos"-th bit (starting from 0) in the "bitmap" to 0.

- **int bitmap_bit_is_set(unsigned char * bitmap, int size, int pos)**

  This function tests if the "pos"-th bit (starting from 0) in the "bitmap" is 1.

  Suppose

  ```
  unsigned char bitmap[] = {0xF7, 0xFF};
  ```

  Then a call as following

  ```
  bitmap_bit_is_set(bitmap, sizeof(bitmap), 3);
  ```

  returns 0, because the bit-3 in the bitmap is 0.

- **int bitmap_print_bitmap(unsigned char * bitmap, int size)**

  This function prints the content of a bitmap in starting from the first bit, and insert a space every 4 bits.

  Suppose

  ```
  unsigned char bitmap[] = {0xA7, 0xB5};
  ```

  Then a call to the function would print the content of the bitmap as

  ```
  1110  0101  1010  1101
  ```

  The implementation of this function is given.

2.3 Complete the memory manager implementation (70 points)

In `memory_manager.h` two important structures are defined:

- `struct _stru_mem_batch`: structure definition of the a memory batch (of memory slots).
- `struct _stru_mem_list`: structure definition of a memory list (of memory batches).

To better assist you understand how a memory manager is supposed to organize the memory using the two data structures, Figure 2 shows an example of a possible snapshot of the memory manger's data structures.

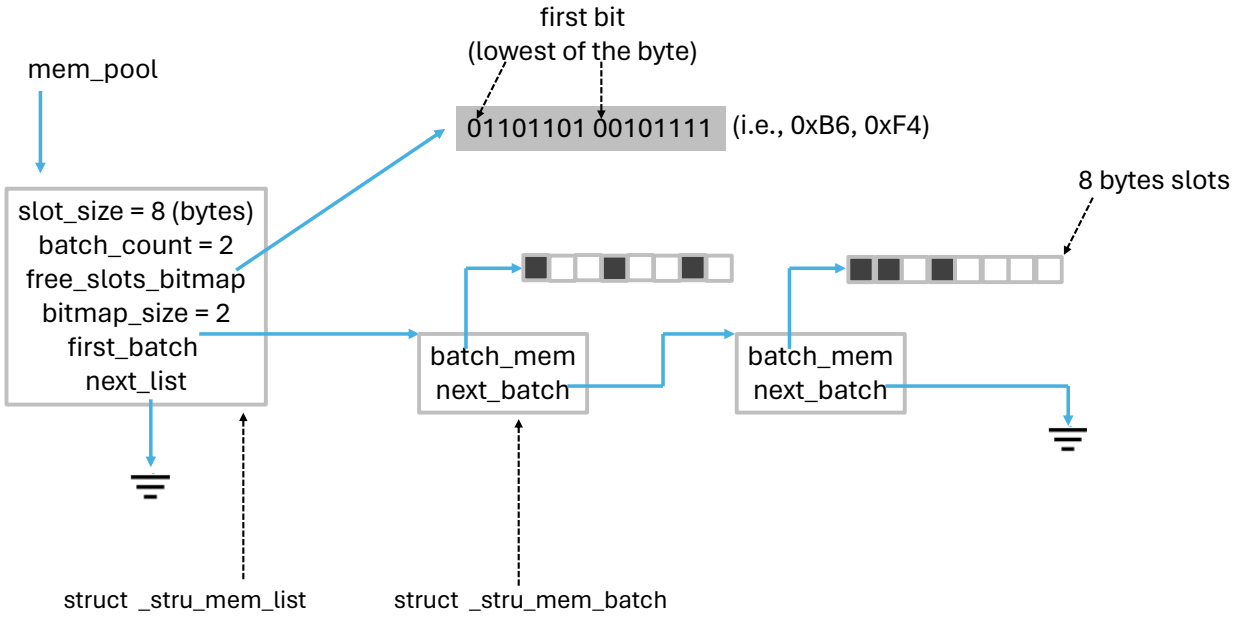

Figure 2: An example of data structures

Basically there are two kinds of linked list:

- A list of memory batches (with a certain slot size): as shown in the previous example, this list expands when there is no free slot available. The memory manager adds a new batch at the end of the list.
- A list of memory batch list: this list expands when a new slot size comes in. Be aware, in the base implementation of this project, assume this list won't expand. No new slot size is required to handle the requests.

You will need to implement the functions in `memory_manager.h`:

- **void * mem_mngr_alloc(size_t size)**
  - This is the memory allocation function of the memory manager. It is used in the same way as `malloc()`.
  - Provide your implementation.

- The macro `MEM_ALIGNMENT_BOUNDARY` (defined in `memory_manger.h`) controls how memory allocation is aligned. For this project, we use 8 byte aligned. But your implementation should be able to handle any alignment. One reason to define the alignment as a macro is to allow for easy configuration. When grading we will test alignment that is multiples of 8 by changing the definition of the macro `MEM_ALIGNMENT_BOUNDARY` to 8, 16, 24, ...
- The macro `MEM_BATCH_SLOT_COUNT` (defined in `memory_manger.h`) controls the number of slots in a memory batch. For this project this number is set to 8. But your implementation should also work if we change to the macro `MEM_BATCH_SLOT_COUNT` to other values. When grading, we will test the cases when `MEM_BATCH_SLOT_COUNT` is set to multiples of 8 (i.e., 8, 16, 24, ...).
- When there are multiple free slots, return the slot using the first-fit policy(i.e, the one with smallest address).
- Remember to clear the corresponding bit in `free_slots_bitmap` after a slot is allocated to user program.
- Do not use array for bitmap, because you never now how many bits you will need. Use heap memory instead.
- Remember to expand the bitmap when a new batch is added to a list, and keep the old content after expansion.

- **void mem_mngr_free(void * ptr)**
  - This is the memory free function of the memory manager. It is used in the same way as `free()`.
  - Provide your implementation.
  - The `ptr` should be a starting address of an assigned slot, report(print out) error if it is not (three possible cases:
    1: `ptr` is the starting address of an unassigned slot - double freeing;
    2: `ptr` is outside of memory managed by the manager;
    3: `ptr` is not the starting address of any slot).
  - Remeber to set the corresponding bit in `free_slots_bitmap` after a slot is freed, so that it can be used again.

- **void mem_mngr_init(void)**
  - This function is called by user program when it starts.
  - Initialize the lists of memory batch list with one default batch list. The slot size of this default batch list is 8 bytes. Assume allocation requests are always smaller than or equal to 8 bytes.
  - Initialize this default list with one batch of memory slots.
  - Initialize the bitmap of this default list with all bits set to 1, which suggests that all slots in the batch are free to be allocated.

- **void mem_mngr_leave(void)**

- This function is called by user program when it quits. It basically frees all the heap memory allocated.
- Provide your implementation.

The following function has already been implemented. It prints out the current status of the memory manager. Reading this function may help you understand how the memory manager organizes the memory. Do not change the implementation of this function. It will be used to help the grading.

void mem_mngr_print_snapshot(void)

2.4 Bonus (20 points)

In the base implementation, you are only required to deal with memory allocation requests with a maximum size of 8 bytes. For those seeking to challenge themselves and earn additional points, your code should be capable to handle any requests for memory allocation greater than 8 bytes, such as 16 bytes, 32 bytes or larger.

To get a better understanding of this, recall the example in the overview section, where all 9 memory allocation requests are for sizes less than or equal to 8 bytes.

- Now, suppose the 10th memory request from the user program is `malloc(28)`. The manager should return a memory chunk of 32 bytes (remember memory allocation is 8 bytes aligned). Because there is no memory list of 32-byte-slots, the manger has to allocate a batch of memory slots of 32 bytes, and returns the first slot to fulfill this request. At this moment, the memory list of 32-byte-slots has only one memory batch.
- Memory lists are also linked together using linked list. So the default memory list with slot size of 8 bytes is linked with the newly created 32 bytes slot size memory list.

Figure 3 shows an extended example of the memory manger's data structures.

In fact, the base implementation is just one specific case of the bonus code. Therefore, ensure your code is compatible with both, seamlessly handling memory requests up to 8 bytes as part of its capability to manage larger memory sizes.

Consider these potential changes for the required functions.

- **void * mem_mngr_alloc(size_t size)**
  - Create a new memory list to handle the request if none of the existing memory list can deal with the requested size.
  - Add this memory list into the list of memory lists.
- **void mem_mngr_free(void * ptr)**
  - Search through all the memory lists to find out which memory batch the `ptr` associated slot belongs to.
- **void mem_mngr_leave(void)**
  - Don't forget to free all the memory lists.

Make sure your code can meet the criteria described in the base implementation, then move forward with the bonus implementation.

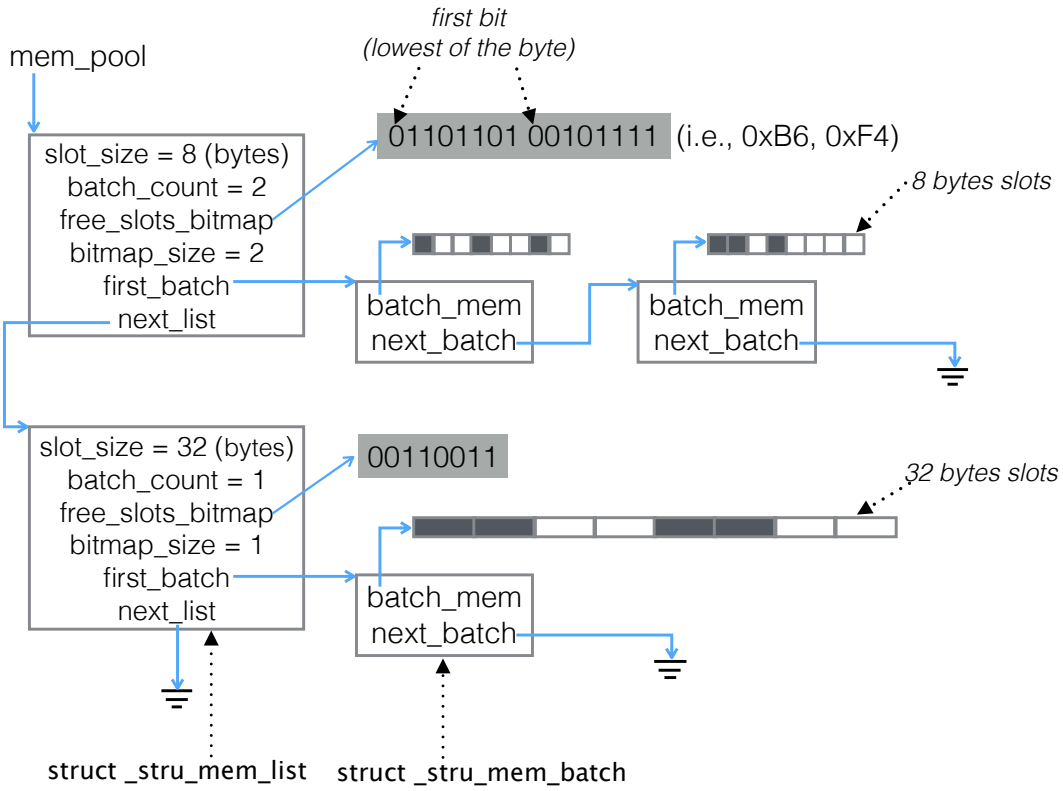

Figure 3: An example of data structures

2.5 Writing a makefile (5 points)

Write a makefile to generate

- `memory_manager.o`
- `bitmap.o`
- a static library `memory_manager.a`, which contains the previous relocatable object files. Your library will be linked to our test program for testing.

2.6 Log and submit your work

Log your work: besides the files needed to build your project, you must also include a README file which minimally contains your name and B-number. Additionally, it can contain the following:

- The status of your program (especially, if not fully complete).
- Bonus is implemented or not.
- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Possibly a log of test cases which work and which don't work.
- Any other material you believe is relevant to the grading of your project.

Compress the files: compress the following into a ZIP file:

- bitmap.c
- common.h
- interposition.h
- memory_manager.c
- memory_manager.h
- Makefile
- README

Name the ZIP file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the zip file should be "proj1_abc.zip".

Submission: submit the ZIP file to Brightspace before the deadline.

2.7 Grading guidelines

(1) Prepare the ZIP file on a Linux machine. If your zip file cannot be uncompressed, 5 points off.

(2) If the submitted ZIP file/source code files included in the ZIP file are not named as specified above (so that it causes problems for TA's automated grading scripts), 10 points off.

(3) If the submitted code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(4) If the code is not working as required in this spec, the TA should take points based on the assigned full points of the task and the actual problem.

(5) Late penalty: Day1 10%, Day2 20%, Day3 40%, Day4 60%, Day5 80%

(6) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with your fellow students, but code should absolutely be kept private.