Project 4: Memory Allocator

Start Assignment

Due Apr 7 by 11:59pm **Points** 50 **Submitting** a file upload **File Types** c
Available Mar 11 at 12am - Apr 7 at 11:59pm 28 days

This project focuses on one of the major themes of Machine Organization and Programming classes taught for many years at Universities all over the world and has gained a reputation for being a challenging project. The hard part of this project for most students is understanding how memory allocators work. The code required to complete the project doesn't have to be long but it can be a little tricky. We strongly recommend making sure you are caught up on lecture and have studied section 9.9 to 9.9.6 of the CSAPP book before beginning.

The memory allocator we will develop for this project is very similar to the implicit free list model from the lecture and the book. However, we use a struct as the block header to store the size, allocated bit, and some extra information that we can use to analyze performance. All of the performance analysis has been done for you in the Mem_Dump function.

Corrections and Additions

None yet.

Learning Goals

The purpose of this project is to help you understand the nuances of building a memory allocator and further increase your C programming skills by working more with pointers.

Memory Allocator Specifications for the Project

- Header size is 8 bytes as BLOCK_HEADER struct (see below)
- Minimum block size = 16 (block size is header size + payload size + padding size)
- Block sizes must be multiples of 16.
- Pointers returned to users must be addresses that are divisible by 16.

- Headers are located 8 bytes before user pointers
- Use the First Fit model to allocate blocks.
- Use Immediate Coalescing for both blocks after and blocks before the one being freed.
- Your mem_functions.c file may not contain the word "malloc", even in comments.
- The project is divided across four files
  - a template file mem_functions.c where you will do your work to implement Mem_Alloc and Mem_Free.
  - mem.h the header file with the BLOCK_HEADER struct definition and all function prototypes
  - mem_init.c contains code provided to you for initializing the heap and setting up the first and last headers as well as the Mem_Dump function which prints out the contents of the heap.
  - Several sample drivers that call the memory allocator functions for testing.

Files

- mem.h ⤓ (https://canvas.wisc.edu/courses/280031/files/25219498/download?download_frd=1)
- mem_init.c ⤓ (https://canvas.wisc.edu/courses/280031/files/25219500/download?download_frd=1)
- mem_functions_template.c ⤓ (https://canvas.wisc.edu/courses/280031/files/25219499/download?download_frd=1) **(rename this to mem_functions.c)**
- driver_alloc_8.c ⤓ (https://canvas.wisc.edu/courses/280031/files/25219495/download?download_frd=1)
- driver_alloc_1.c ⤓ (https://canvas.wisc.edu/courses/280031/files/25219493/download?download_frd=1)
- driver_alloc_1_24.c ⤓ (https://canvas.wisc.edu/courses/280031/files/25219494/download?download_frd=1)
- driver_alloc_8_free.c ⤓ (https://canvas.wisc.edu/courses/280031/files/25219496/download?download_frd=1)
- driver_free.c ⤓ (https://canvas.wisc.edu/courses/280031/files/25219497/download?download_frd=1)

Compiling and Running the code

Download and move all of the files to the CSL machines. You will need three .c files when you compile every time to link together all of the pieces: mem_init, mem_functions, and one of the drivers.

Test your code with the driver files in the order listed above. The first three only require Mem_Alloc and the last two require both functions.

```
gcc -g mem_init.c mem_functions.c driver_alloc_8.c -Wall
./a.out
```

Block_Header

Please find the declaration for BLOCK_HEADER in mem.h. The block headers struct contain three pieces of data, packed into two variables. Extracting the information from this struct can be a little tricky. I recommend writing helper functions so you only need to get this right once and can reuse this code. The first data member is int size_alloc.

Size_alloc contains the total block size, which must be divisible by 16. That means the last four bits will always be zero, which allows us to borrow one of those bits (the least significant bit) to store the allocated bit. This works the same way as in the lecture and the textbook.

We're also storing a second int payload. This is the exact amount of bytes requested by the user. Memory allocators use meta information stored in the heap to find and free blocks. We're using this extra payload data to compute how efficiently our memory allocator manages the heap. For free blocks the payload contains the total bytes available to be allocated in this block.

Mem_Alloc and Mem_Free

Write the code to implement Mem_Alloc() and Mem_Free(). Use the first fit algorithm when allocating blocks with Mem_Alloc(). When freeing memory, always coalesce both adjacent memory blocks if they are free.

```
void *Mem_Alloc(int size);
```

Mem_Alloc() is similar to the library function malloc(). Mem_Alloc() takes as an input parameter the size in bytes of the memory space to be allocated, and it returns a pointer to the start of that memory space (i.e, this means, a pointer to the start of the first useful byte, after the header). The function returns NULL if there is not enough contiguous free space to satisfy this request. Mem_Alloc() is required to return a memory address aligned to 16-byte data. This means that the address returned to the user should be divisible by 16.

Once the appropriate free block is located, we could use the entire block for the allocation. However, the disadvantage is that it causes internal fragmentation and wastes space. So, if possible, we will split the block in two. The first part becomes the allocated block, and the remainder becomes a new free block. Before splitting the block, verify that there is enough space left over for a new free block, i.e. 16 bytes; otherwise, do not split the block.

The payload variable of the BLOCK_HEADER stores the exact size requested by the user. Note that this is different than the allocator blocks we saw in the lecture. We use this to analyze memory usage.

```
int Mem_Free(void *ptr):
```

Mem_Free() frees the memory object that ptr points to. First, traverse the list of blocks to verify that the pointer could have been allocated with Mem_Alloc. If the pointer could not have been allocated with Mem_Alloc, return -1.

Then free the block. For the block being freed, always coalesce with its adjacent blocks if either or both of them are free. Return 0 upon success.

Mem_Init() This function is provided for you and creates sets up the heap. Study this function to see how the BLOCK_HEADERS are created and used.

Mem_Dump() This function prints a table of all blocks in the heap managed by our memory allocator. Study this code for more clues about accessing the data. If you see an infinite loop when the table is printed - edit this function to break out of the loop after the first few lines are printed.

Test the Code

We are providing a limited test suite of drivers for this project. Please write more drivers to completely test your code. Make sure you test allocations that are too big, almost too big, small enough to require splitting, big enough to require not splitting, and coalescing before, after, and both.

The Infinite Loop

The number one mistake you are most likely to see is an infinite loop that manifests during when the Mem_Dump is called. This function prints out the status of the heap. The blocks in our memory allocator work like a linked list and if there is an error finding the next block you may visit the first block over and over again. If you see Mem_Dump printing in an infinite loop - edit the function to break out of the loop when the variable ID is greater than a small number (like 5). That will allow you to see the first few lines of the table which will help you identify your error.

Hints

1. Invest the time to understand the BLOCK_HEADER struct and the data that it stores.
2. BLOCK_HEADER vs BLOCK_HEADER* vs void*.
    1. Remember the C treats all pointers as the base address of an array of that type.

    2. So if you have a pointer to a BLOCK_HEADER, it treats this as an array of BLOCK_HEADERS
    3. means that C will “help” you with your arithmetic and include the size of the header for you.
    4. new_header = header + 16 will result in new_header containing an address that is 128 bytes larger – not 16 bytes larger
    5. To get around this use casting. To do the arithmetic yourself without the compiler “helping” cast all pointer addresses to (unsigned long) before doing math and then cast them back to (BLOCK_HEADER *) when finished.
    6. Check out the examples in Mem_Init and Mem_Dump
3. Write helper functions so you only need to get the hard stuff correct once. Here are some suggested function names:
    1. Print_Block_Data
    2. Is_Free
    3. Set_Free
    4. Set_Allocated
    5. Get_User_Pointer
    6. Get_Next_Header
    7. Get_Padding
4. Mem_Init() and Mem_Dump() are provided for you. There are significant clues as to how to work with the headers within these functions. One of the first things you should do is study these functions to see how they access the data.

<u>Turn in</u>

- Upload your mem_functions.c file to Canvas