ECE 2230: Computer Systems Engineering
Machine Problem 4

Spring 2021

Due: 11:59 pm, Friday April 02

1 Introduction

All prior MPs have depended on the ability to dynamically allocate memory. As we learned in class, dynamic memory is allocated on the heap and managed by the operating system. This MP allows us to better understand the heap and issues related to dynamic memory management. The goal of this machine problem is to investigate strategies for managing a memory heap. In particular, we use a free list to manage free blocks of memory. The free list will provide memory regions to satisfy memory allocation calls.

The code must consist of the following files:

`lab4.c` – This file is provided and contains a few test drivers for testing your code. You must add additional test drivers to this file, and you must document your tests in your test plan.

`mem.c` – Your implementation of `Mem_alloc`, `Mem_free`, and supporting functions to manage the memory heap.

`mem.h` – The data structures and prototype definitions for the memory heap.

`makefile` – Compiler commands for all code files. An example is provided.

Two additional documents should be included. The first required document is a **test plan** that describes details of your implementation and how you plan to verify that the code works correctly. The second required document is a **test log** that demonstrates how you verified that the code works correctly. You must add at least three new unit test drivers to `lab4.c`.

2 Management of the memory heap

You are tasked to implement three procedures to manage the memory heap.

`void *Mem_alloc(int nbytes);` – Returns a pointer to space for an object of positive size `nbytes`, or `NULL` if the request cannot be satisfied. The space is uninitialized. It should first check your free list to determine if the requested memory can be allocated from the free list. If there is no memory block available to accommodate the request, then a new *segment* of memory should be requested using the `morecore()` function. This function will internally request a new segment of memory from the operating system using the system call `sbrk()`. The memory returned from `morecore()` should be put in the free list. After adding the additional memory to the free list, a memory block of the correct size can now be found. Therefore, the `Mem_alloc` function will return `NULL` only if the `morecore()` call fails because the system runs out of memory. In all other cases, this function must return the pointer to user's requested memory space.

`void Mem_free(void *return_ptr);` – Deallocates the space pointed to by `return_ptr` by returning the memory block to the free list; it does nothing if `return_ptr` is `NULL`. You can assume that `return_ptr` is a pointer to space previously allocated by `Mem_alloc`. **Do not** provide this routine with a pointer allocated with `malloc` or any other `cstdlib` memory allocation routines!

`void Mem_stats(void);` – Prints statistics about the current free list at the time the function is called. At the time the function is called, scan the free list and determine the following information and populate the appropriate entries in the statistics structure.

- Number of items in the list
- Min, max, and average size (in bytes) of the chunks of memory in the free list
- Total memory stored in the free list (in bytes)

In addition to the statistics that you compute, we keep track of:

- Number of calls to sbrk()
- Number of pages requested

If the total amount of memory in the free list is equal to the number of pages requested multiplied by the `PAGESIZE`, then the message "all memory is in the heap – no leaks are possible" is printed.

3 Memory segments

When a new segment of memory is needed use the `morecore` function to request more memory via the `sbrk` system function. The `sbrk` function expands the size of the data segment of the process. The amount of requested memory must be an integer multiple of a `PAGESIZE` (we have defined this size as 4096 bytes). If your `Mem_alloc` function needs a memory allocation that is larger than one page, request the next larger multiple of the page size. First, put the memory returned by `morecore` into the free list, and then get the appropriate sized memory block to return from `Mem_alloc`. The `morecore` function is already implemented in `mem.c`.

4 Structure for the free list

Here is the definition of the `chunk_t` using a one-way linked list:

```
typedef struct chunk_tag {
        struct chunk_tag *next;   /* next memory chunk in free list */
        long size;                /* size of chunk in units, not bytes */
} chunk_t;
```

A memory block in the free list contains a pointer to the next block in the linked free list, a record of the size of the block, and then the remaining free space itself; the information at the beginning is called the **header** (i.e., the fields `next` and `size` of `chunk_t`). To simplify memory alignment, the size of a block must be a multiple of the header size (i.e., multiple of `sizeof(chunk_t)`). In our implementation, the size of the `chunk_t` structure is 16 bytes (Note: we use a `long` instead of an `int` to ensure the page size is a multiple of the `chunk_t` size). The parameter for `Mem_alloc(nbytes)` is the number of bytes of memory that is requested. One of the first steps is to convert the parameter `nbytes` to an integer number of header-sized units, called `nunits`, where `nunits` is the smallest number of units that provides at least `nbytes` and contains one additional header-sized unit, for the header itself, and the value recorded in the size field of the header is the total number of header-sized units (i.e., `nunits`). These size of the block that the user will use for their memory allocation is therefore `nunits`-1. The pointer returned by `Mem_alloc` points to the start of the space for the user, which is one unit past the header as shown in Figure 1.

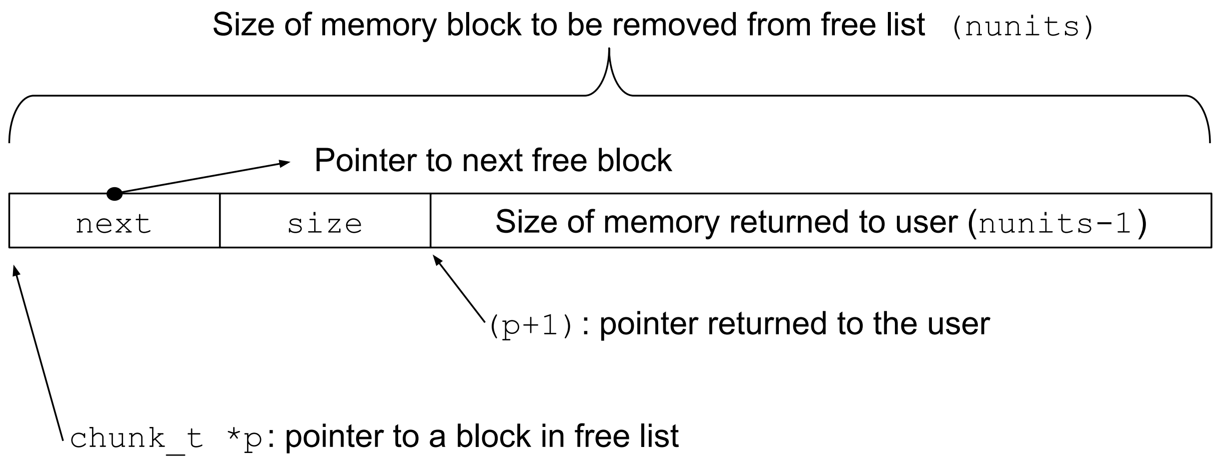
Size of memory block to be removed from free list (nunits)
Pointer to next free block
next
size
Size of memory returned to user (nunits-1)
(p+1) : pointer returned to the user
chunk_t *p: pointer to a block in free list

Figure 1: Logical breakdown of block to be removed from free list.

The free list is implemented using a circular linked list. The list must contain one dummy block (red block in Figure 2). The dummy block is one `chunk_t` memory block for which the size field is set to zero. This guarantees that the dummy block can **never** be removed from the list. Thus, the list always contains at least one item. Because the design of our memory module does not include a header block, we must use a static global variable (with scope limited to `mem.c` only) for the roving pointer and dummy block. Here is an example of the free list with the dummy block and two additional memory blocks.

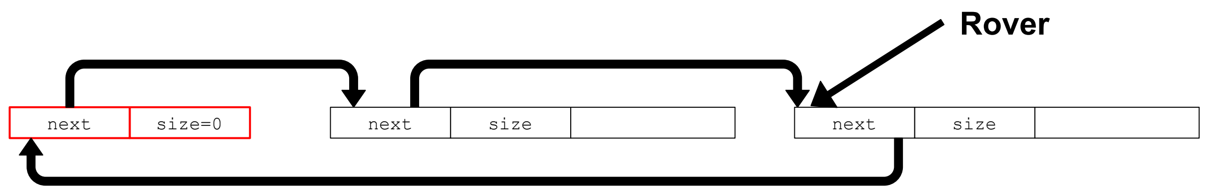

Figure 2: Example free list with two free blocks. Dummy block for the list is the red node with `size` 0.

Here is an example of the free list when it is empty:

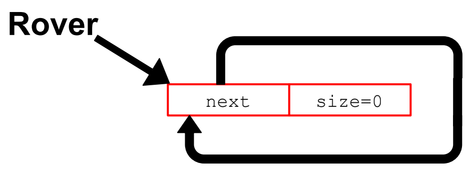

Figure 3: Empty free list contains only the dummy block with `size` 0.

5 Memory package requirements and options

1. You must implement a **first-fit**, **best-fit**, and **worst-fit** policy for searching for a memory block on the free list. When a memory block has been found and removed from the free list, the roving pointer must point to the next memory block in the free list (i.e. the next block to the block removed or carved). The template for `lab4.c` includes a command line option to specify the search policy. The policy is specified using the *-f {first | best | worst}* command line flag.

2. You must implement two strategies for selecting the starting point of the search in the free list. The starting location is specified using the `-h {rover | head}` command line flag. The `rover` option must use a roving pointer. The `head` option causes the location of the roving pointer to be reset to the address of the dummy block at the start of any search of the free list.

3. You are not permitted to use `malloc()` or `free()`. Instead you are developing the procedures to replace these functions.

4. The rover always points to the free block just after the block used to satisfy a `Mem_alloc.` If the the free block is an exact match, then the rover points at that block's `next`. Otherwise, there is a remainder on the selected block, and the rover points to next of the remainder of the selected free block when it is reinserted into the free list.

6 Testing

Write unit test drivers to test your library extensively and write a detailed description in your **test plan**. Remember to record their output and justification for success in the **test log**. A unit test driver performs a systematic sequence of tests. Here are a few of the tests you should perform and document (but the details depend on your design):

- Test special cases such as boundary conditions for memory block sizes. For example
  - Allocate blocks 1, 2, and 3
  - Print the free list
  - Free blocks 1 and 3
  - Print the free list and verify the hole between blocks 1 and 3
  - Extend the above with other patterns and sizes
  - Have one trial request a whole page (or a whole page minus space for a header)
  - Have one trial remove all memory from the free list and show the list is empty
- Show that your roving pointer spreads the allocation of memory blocks throughout the free list.
- Call `Mem_stats` at the end of each driver when all memory has been returned to your free list. Verify that the total memory stored in the free list (in bytes) matches the number of pages requested times the page size (in bytes) and that the message "*all memory is in the heap -- no leaks are possible*" is printed.

Your tests must be added as drivers to the file `lab4.c` and documented in your test plan. Each driver is enabled using the `-u` command line argument (see notes in `lab4.c` about command line arguments).

7 Optional Bonus: Coalescing (10 points)

In order to receive bonus points you must implement a second policy for freeing memory called **coalescing**. The default option is without coalescing, and coalescing is enabled at run time using the command line with the `-c` flag. Coalescing requires you to maintain the free list in sorted order where the order is determined by the addresses of the memory blocks. When a new block is put in the free list, coalesce the block if the previous or next blocks form a larger block of continuous memory. If you choose to implement coalescing you must also add **three** extra unit tests and update the test log and test plan to test coalescing.

8 Submission

Command line arguments must be used to modify parameters for the test drivers, and options for the memory library. See the comments in lab4.c for the command line options and their meanings.

Compress all code, test plan, and optional performance evaluation into a ZIP file, and submit the ZIP file to Canvas by the deadline. **Your last submission is the one that will be graded.**

See the ECE 2230 Programming Guide for additional requirements that apply to all programming assignments. Work must be completed by each individual student, and see the course syllabus for additional policies.