ECE 223: Computer Systems Engineering **Fall 2015**
Machine Problem 4 **Due: 11:59 pm, Thursday, October 15**

The goal of this machine problem is to investigate strategies for managing a memory heap.

The code must consist of the following files (additional files can be included and the design must be documented in the test plan):

`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.
`list.c` – the two-way linked list ADT from assignments 2 and 3. lab4.c requires the use of list_remove(list_ptr, index_ptr) for index_ptr pointing to any node in the list (not just the first or last). You may need to fix your implementation of list_remove so it can remove an item from any position. The list ADT is used to support a test driver in lab4.c. **However, none of the list ADT functions can be used in mem.c**.

`datatypes.h` – Key definitions for list.c. An example is provided.
`mem.h` – The data structures and prototype definitions for the memory heap.
`list.h` – The data structures and prototype definitions for the two-way linked list ADT.

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

Two additional documents should be submitted. One is a **test plan** that describes details of your implementation and demonstrates how you verified that the code works correctly. You **must** add at least one new unit driver to lab4.c. The second document describes your **performance evaluation**, and the details are described below.

Management of memory heap

You are to design four procedures to manage the memory heap.

```
void *Mem_alloc(const int nbytes);
```

Returns a pointer to space for an object of 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 from the system using the system call `sbrk()` and the memory 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. So, the Mem_alloc function will return NULL only if the `sbrk()` call fails because the system runs out of memory.

```
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. return_ptr must be a pointer to space previously allocated by Mem_alloc

```
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. (You can print additional information)

- 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). Define this to be M.
- number of calls to sbrk() and the total number of pages requested (define as P)
- If M is equal to P * PAGESIZE, then print the message "all memory is in the heap – no leaks are possible"

`void Mem_print(void);`
Print a table of the memory in the free list. Here is an example for the format, but you must modify this to suit your design. The pointer "`chunk_t *p`" points to one memory chunk in the list

```
printf("p=%p, size=%d, end=%p, next=%p\n", p, p->size, p + p->size, p->next);
```

Memory segments

When a new segment of memory is needed, use the unix system command `sbrk` to request the memory from the system. Include the following header,

```
#include <unistd.h>
```

This header contains the prototype for the `sbrk` command:

```
int sbrk(int increment);
```

The `sbrk` command adds `increment` bytes to the data segment of the process. The value for `increment` must be an integer multiple of a page size (assume a page is 4096 bytes). If your Mem_alloc command 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 `sbrk` into the free list, and then get the appropriate size memory block to return from Mem_alloc. The return value from `sbrk` is either a pointer to the memory segment or -1 if there was no space. You are required to use the following function called `morecore()`. Because the return value for an error is -1 instead of NULL, the test for an error must be handled like this:

```
#define PAGESIZE 4096
chunk_t *morecore(int new_bytes)
{
      char *cp;
      chunk_t *new_p;
      assert(new_bytes % PAGESIZE == 0 && new_bytes > 0);
      cp = sbrk(new_bytes);
      if (cp == (char *) -1)  /* no space available */
            return NULL;
      new_p = (chunk_t *) cp;
      // add something like: C++;  P += new_bytes/PAGESIZE;
      return new_p;
}
```

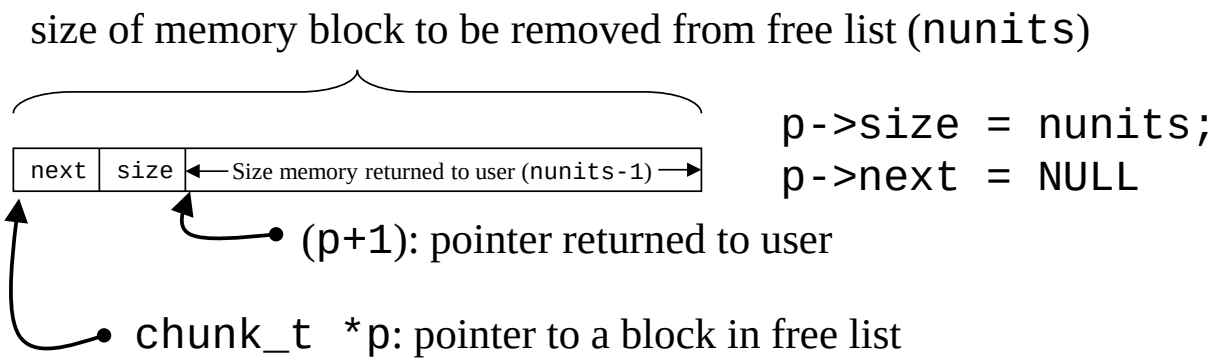
Structure for the free list

Here is an example of how to define `chunk_t` for a one-way linked list:

```
typedef struct chunk_tag {
      struct chunk_tag *next;  /* next memory chunk in free list */
      int 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` for the example `chunk_t` given here). To simplify memory alignment, the size of a block must be a multiple of the header size (i.e., multiple of `sizeof(chunk_t)`). 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, which is the smallest number of units that provides at least `nbytes`. The block that is to be allocated contains one additional header-sized unit, for the header itself. So the total number of units that must be extracted from the free list is **`nunits`**. The pointer returned by `Mem_alloc` points to the start of the space for the user, which is one unit past the header.

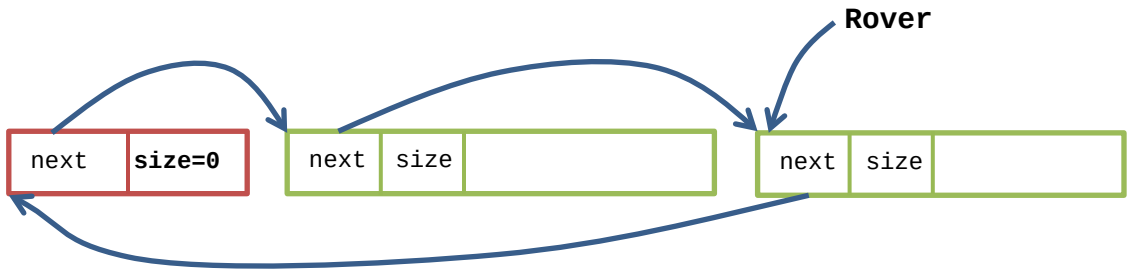


Notes on alignment of memory.

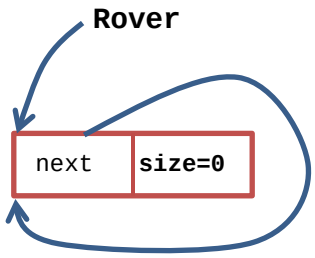
1. `sizeof(chunk_t)` must evenly divide into the page size (i.e., 4096)
2. The member `size` in `chunk_t` must be equal to `i * sizeof(chunk_t)` for an integer `i` = 1, 2, … Size is not measured in bytes but in multiples of `sizeof(chunk_t)`. For example, here is how to request `num_pages` of new memory from the OS and add it to the free list.

```
chunk_t *p, *q;
new_bytes = PAGESIZE * num_pages;              /* num_pages is an int */
p = morecore(new_bytes);
p->next = NULL;
p->size = new_bytes / sizeof(chunk_t);       /* num header-sized units */
q = p + 1;                        /* move to one unit past start of block */
Mem_free( q );                              /* add new segment to free list */
```

The free list must be implemented using a circular linked list. The list must contain one dummy block. 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 pointer to a header block for the list, you must use a static global variable (with scope limited to `mem.c` only) for the roving pointer. Here is an example of the free list with the dummy block and two additional memory blocks.



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



Memory package requirements

1. You are required to implement the free list with a circular linked list. It is your choice on whether to use a one-way or two-way linked list.
2. You must implement two policies for freeing memory. One policy simply places the memory block that is returned at the location of the roving pointer. The second policy is 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.
3. You must implement three policies for searching for a memory block in the free list: first-fit, best-fit, and worst-fit. The search policies all begin at the roving pointer (do not start searching at the dummy block). 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. The first-fit policy stops at the first memory block that can satisfy the request. The best-fit policy finds the memory block that is closest in size to the request but that is also large enough. In the best case it finds a memory block that is exactly the correct size. The worst-fit policy finds the largest block in the free list that satisfies the request. The effect of worst-fit is that the fragment that is left in the list is as large as possible. An optional improvement to worst-fit is to find either an exact match (in which case there is no fragmentation) or the largest memory block (so the remaining fragment is as large as possible). 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.
4. You must implement a roving pointer. After an allocation, the roving pointer is always left pointing to the memory block immediately after the memory block that was removed from the list. After a free, the roving pointer is pointing to the memory block that was just returned to the free list.
5. You are not permitted to use `malloc()` or `free()`. Instead you are developing the procedures to replace these functions.

Testing and performance evaluation

Write unit drivers to test your library extensively and write a detailed description in your **test plan**. A unit 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 (and a whole page minus space for a header)

    - Have one trial remove all memory from the free list and show the list is empty
    - Repeat all the above tests but with coalescing enabled and show all coalescing patterns
- 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). Do not use the equilibrium driver found in `lab4.c` for testing (use this driver for performance evaluation). Do not describe results from the equilibrium driver in your test plan.

For the **performance analysis** document use the equilibrium driver to evaluate the performance of your design for dynamic memory.

- Describe the advantages and disadvantages of the options you selected to implement for your dynamic memory allocation package. Examples of items you should discuss include
    - discuss why you selected your `chunk_t` structure, including implementing a one-way versus two-way linked list and how that effects the efficiency of searching for and allocating memory
    - discuss the advantages and disadvantages of first-fit, best-fit, and worst-fit search policies, and how the search policy and the roving pointer effect fragmentation of the free list
    - discuss the effect of coalescing on fragmentation and run time
- Use the equilibrium driver included in `lab4.c` to evaluate the performance of your memory library.
    - Compare the performance with and without coalescing by considering both the total size of memory used for the heap, the number of chunks in the free list, the average size of the chunks in the free list, and the time to complete the equilibrium phase of the driver. Use the default values for the equilibrium driver, but also consider other values.
    - Consider the special case that the size of each allocation request is the same. That is, use the option '**`-r 0`**'.
    - Compare the performance of your search policies to the default memory library available with standard C. The **`-d`** command line flag forces the equilibrium driver to use `malloc/free` instead of `Mem_alloc/Mem_free`.

Notes

Before making any modifications to `mem.c`, first test that the template code works.

```
./lab4 -e -d
```

This verifies that your list ADT (`list.c`) will work correctly with the equilibrium driver.

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.

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. All code and documentation files must be turned in to Blackboard.

Work must be completed by each individual student, and see the course syllabus for additional policies.