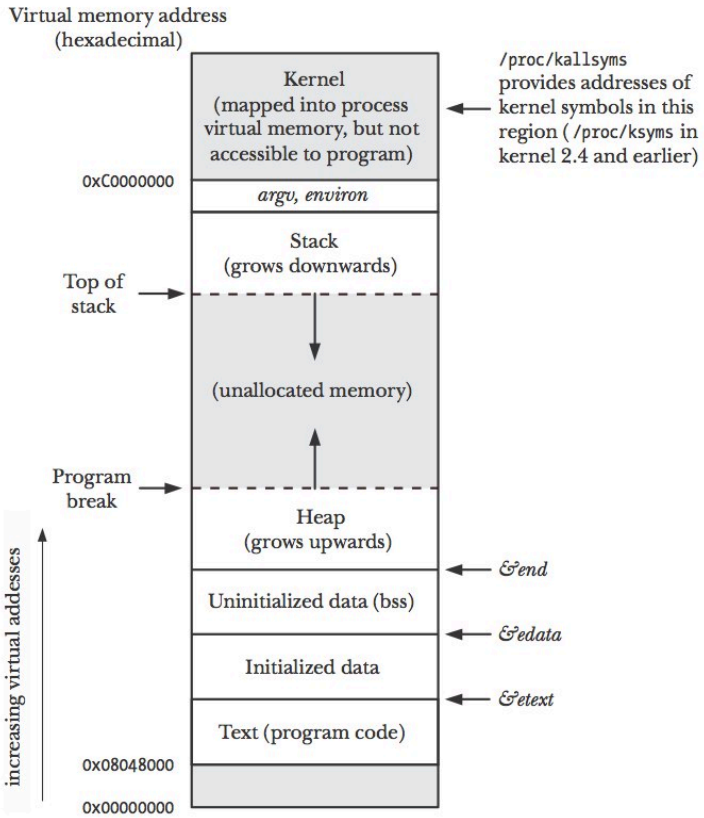
Programming Assignment #2: Memory Allocation Package)

Due: see My Courses for the deadline

What is required as part of this assignment?

In this assignment, you will develop a memory allocation library that is equivalent to the **`malloc`** library provided by Unix-like operating systems. The *malloc()* function allocates variable sized contiguous memory chunks on the heap, which is the memory segment just after the uninitialized data segment. The current end of the heap is given by the *program break* as shown in the figure below. The *malloc()* and related functions are built on top of *brk()* and *sbrk()*, which we discuss next.



To allocate memory on the heap, we need to tell the kernel that the process' program break is located at a different address such that there is additional room in the heap. Initially, the program break lies just past the end of the uninitialized data segment. After the program break is increased, the program can access the memory in the newly created heap space. The *brk()* system call sets the program break to the location specified by its argument. The *sbrk()* library routine is similar where an increment can be specified to increase the program break.

```
#include <unistd.h>

int brk(void *end_data_segment);
                                        Returns 0 on success, or −1 on error
void *sbrk(intptr_t increment);
            Returns previous program break on success, or (void *) −1 on error
```

The call sbrk(0) returns the current value of the program break without changing it. It can be useful to track the size of the heap to monitor the behavior of the memory allocation package.

Your memory allocation package needs to provide the following functions for managing the heap.

`void *my_malloc(int size)`

This function returns void pointer that we can assign to any C pointer. If memory could not be allocated, then my_malloc() returns NULL and sets a global error string variable given by the following declaration.

`extern char *my_malloc_error`

The possibility of error in memory allocation may be small, but it is important to provide mechanisms to handle the error conditions.

`void my_free(void *ptr)`

This function deallocates the block of memory pointed by the *ptr* argument. The *ptr* should be an address previously allocated by the Memory Allocation Package.

In UNIX/Linux, *free()* does not lower the program break. It adds the block of freed memory to the free list so it could be recycled by future calls to *malloc()*. This is done for several reasons.

- The block being freed could be somewhere in the middle of the heap.
- Minimize the number of *sbrk()* calls that the program must perform to minimize the number of system calls.
- In many cases, lowering the program break would not help programs that allocate large amounts of memory. Such programs tend to either hold onto the allocation for long periods of times or repeatedly allocate and deallocate memory segments.

You could consider these reasons when deciding what my_free() should be doing. If the argument of *my_free()* is NULL, then the call should not free any thing. Calling *my_free()* on a *ptr* value can lead to unpredictable results.

The *my_free()* should reduce the program break if the top free block is larger than 128 Kbytes.

`void my_mallopt(int policy)`

This function specifies the memory allocation policy. You need implement two policies as part of this assignment: first fit and best fit. Refer to the lecture slides for more information about these policies.

`void my_mallinfo()`

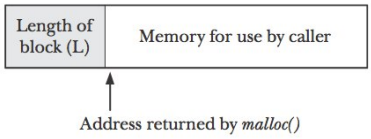
This function prints statistics about the memory allocation performed so far by the library. You need to include the following parameters: total number of bytes allocated, total free space, largest contiguous free space, and others of your choice.

Proposed approach

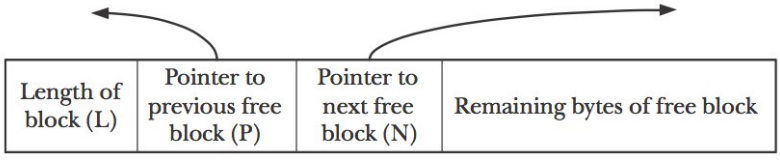
DO NOT USE malloc, calloc, or other variations offered by the operating system in this assignment. Remember you are using lower level system calls and library routines for implementing the memory management library. Using malloc routines will change the heap configuration and destroy your setup. Some library routines like printf() also use malloc, so you need to avoid their use as well. Use puts() and sprintf() instead of printf().

The implementation of the memory allocation package is straightforward. It first scans the list of free memory blocks previously released by *my_free()* to find one whose size is large than or equal to the one to be allocated. You can use different strategies such as first-fit or best-fit for scanning the free list. If a larger block is found, then it is split and a portion is returned as the allocation while the other portion remains in the free list. If no block on the free list is large enough, *sbrk()* is used to allocate more memory. To reduce the calls to *sbrk()*, rather than allocating exactly the number of required bytes, *my_malloc()* increases the program break in larger units and puts the excess memory onto the free list. The excess memory cannot be more than 128 Kbytes to be consistent with the way *my_free()* works.

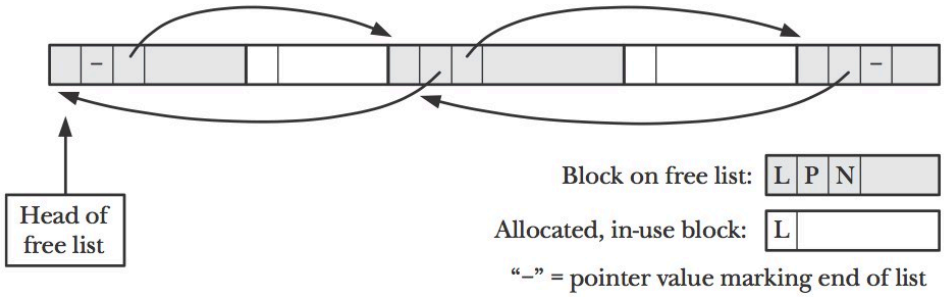
One of the challenges is deallocating the correct number of bytes when my_free() is called. To know the correct number of bytes to place in the free list, my_free() gets a little help from my_malloc(). When my_malloc() allocates the block, it allocates extra bytes to hold an integer containing the size of the block. The integer is located at the beginning of the block; the address actually returned to the caller points to the location just past this length value, as shown in the figure below.



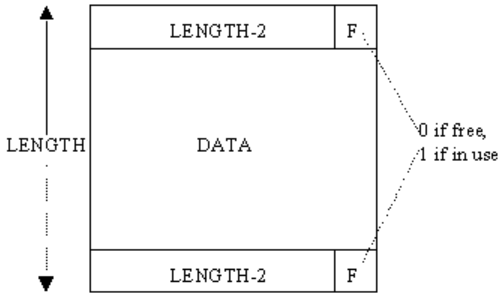
When a block is placed on the double linked free list, my_free() uses a structure such as the following to represent the free block. It should be noted that all the free blocks are connected in a doubly linked list while the allocated one are not.



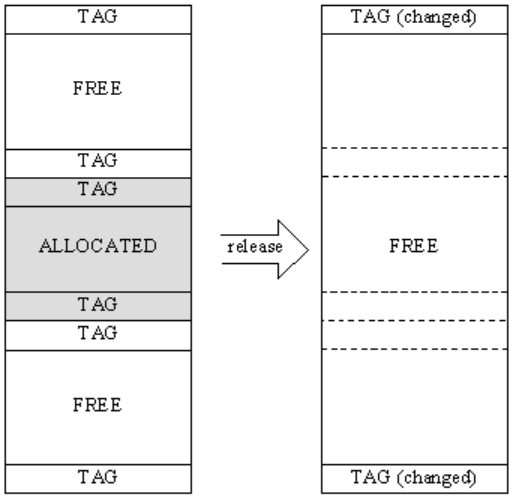
As blocks are deallocated and reallocated over time, the blocks of the free list will become intermingled with the blocks of the allocated memory as shown below.



When *my_free()* runs, it is essential to check adjacent blocks for their status. If free blocks are next to the block being freed, you need to merge them into a larger free block. To easily detect adjacent free blocks, a boundary-tag scheme could be implemented. The boundary tag is a slight variation of the scheme shown above. The figure below shows an example boundary-tag scheme.



(a) Structure of a segment



(b) Coalescence of adjacent segments

You may return a slightly larger block than what is requested from *my_malloc()* to reduce external fragmentation.

What to Hand in

Submit the following files separately:

1. A README file that explains any design decisions the TA should be aware of when grading your assignment.
2. If you have not fully implemented all, then list the parts that work so that you can be sure to receive credit for the parts you do have working. Indicate any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit.
3. All source files needed to compile, run and test your code. If multiple source files are present, provide a Makefile for compiling them. Do not submit object or executable files.
4. Output from your testing of your program.
5. You need to submit the testing routine that you used to test the assignment.