Simple Memory Allocator

Check My Courses for Due Date

High-Level Description

Most of your programs rely on dynamic memory allocation to implement complex data structures. By default, in Unix/Linux you use the **malloc** library for dynamic memory allocation. You could install other libraries instead of **malloc** for dynamic memory allocation. In this assignment, you will develop a library called Simple Memory Allocator (SMA) for dynamic memory management. Because your library is working at the user-level, your library would grab memory in bulk from the kernel and then distribute it to the application requests as they come in. The application program is going to use the API provided by your Simple Memory Allocator to manage dynamic memory.

Here is an example program using the SMA.

```c
#include <stdio.h>
#include "sma.h"

int main(int argc, char *argv[])
{
      int i;
      char *c[32];
      // Allocating 32 kbytes of memory..
      for(i=0; i<32; i++)
            c[i] = (char*)sma_malloc(1024);
      use_memory(c);
      // Now deallocating
      for(i=0; i<32; i++)
            sma_free(c[i]);

      puts("Print some information..");
      sma_mallinfo();

      return(0);
}
```

In the above program, we are allocating 32 KB of memory and passing the memory block (held by the array of pointers) to a function. After the function returns, we don't need the memory block any further. We call the free function provided by SMA to release the memory. Notice that we are not initializing the SMA library before using it. The library is initialized appropriately at first use in the implementation. The last SMA call (`sma_mallinfo`) is asking the library to print out information about its internal operation.

More Details on SMA

The SMA library you develop is equivalent to the **malloc** library provided by Unix-like operating systems. Like the **malloc** library, the SMA 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 denoted by the ***program break*** as shown in the figure below. We use the `brk()`

and `sbrk()` calls to get memory in bulk (i.e., large chunks) from the heap. The SMA is going to hold the heap memory that it got from the OS and use it to allocate the requests from the application. This means the actual heap is not going to grow and shrink in direct relation to allocations and deallocations of memory by the application. The heap growth and shrinkage depend on the SMA actions as well. For example, the application might release lot of memory, but the SMA might still be holding it. As a result, the heap will not shrink to correspond to the memory release operations.
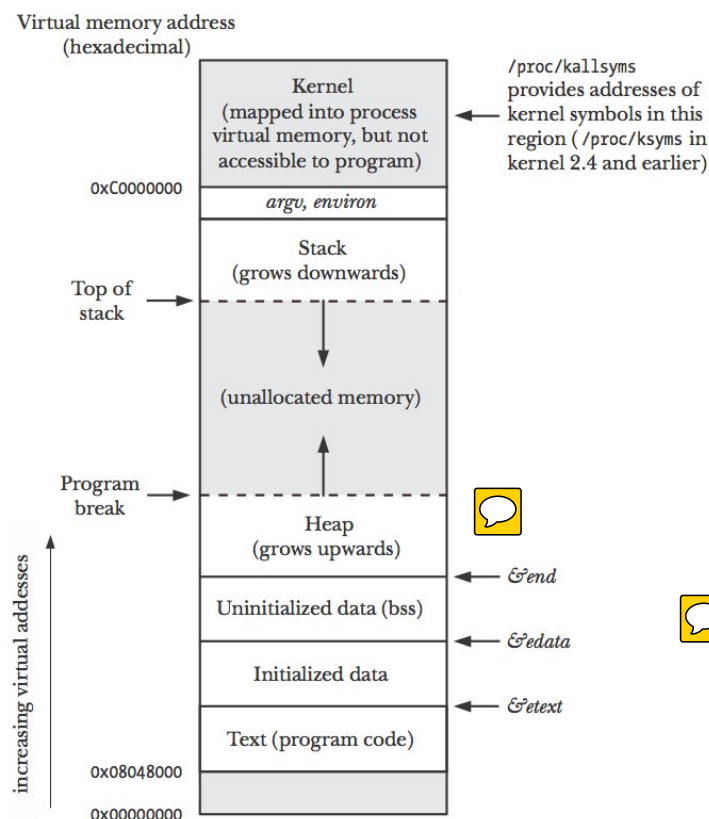
To allocate memory on the heap, we need to tell the kernel that the process' program break is located at a different address such that there is additional room in the heap. Initially, the program break lies just past the end of the uninitialized data segment. After the program break is increased, the 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.

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 *sma_malloc(int size)
```

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

`extern char *sma_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 sma_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 `sma_free()` should be doing. If the argument of `sma_free()` is NULL, then the call should not free anything. Calling `sma_free()` on a `ptr` value can lead to unpredictable results.

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

`void sma_mallopt(int policy)`

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

`void sma_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.

`void *sma_realloc(void *ptr, int size)`

This function is similar to `sma_malloc()`. The major difference is instead of freshly allocating memory, it is resizing the memory pointed by `ptr`, which was previously allocated by `sma_malloc()`. The freshly reallocated block of memory can be in a different memory area because the previous block could not be expanded. In this case, the library needs to copy the data from the old memory block to the new block.

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 SMA library is straightforward. It maintains a list of free memory blocks at all times. When a memory allocation request comes in (i.e., `sma_malloc()` call), it will find one of the memory blocks from this list that is suitable for the request. The suitability of a free memory block for satisfying a request depends on the memory allocation policy that is in effect at the allocation time. In this assignment, you are expected to support **next fit** and **worst fit**.

The available free memory blocks is due to previous deallocations via `sma_free()` calls. In an allocation call (i.e., `sma_malloc()`, we are looking for a free block that is going to satisfy the request according to the allocation policy. If a free block is found and it is larger than the request, we would split the block and a portion is returned as the allocation while the other portion remains in the free list. If no block is found on the free memory block list, *sbrk()* is used to allocate more memory. To reduce the number of calls to *sbrk()*, rather than allocating exactly the number of required bytes, `sma_malloc()` increases the program break in larger units and puts the excess memory into the free list. The excess memory cannot be more than 128 Kbytes to be consistent with the way `sma_free()` works.

One of the challenges is deallocating the correct number of bytes when `sma_free()` is called. To know the correct number of bytes to place in the free list, `sma_free()` gets a little help from `sma_malloc()`. When `sma_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.
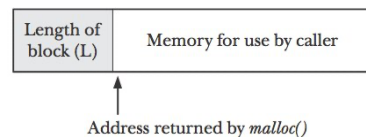

When a block is placed on the double linked free list, `sma_free()` uses a structure such as the following to represent the free block. It should be noted that all the free blocks are connected in a doubly linked list while the allocated ones are not.

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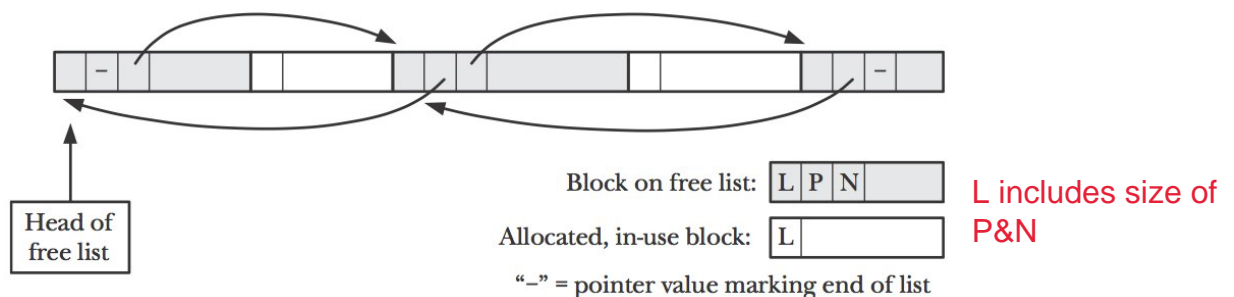
When `sma_free()` runs, it is essential to check adjacent blocks for their status. If free blocks are next to the block being freed, you need to merge them into a larger free block. To easily detect adjacent free blocks, a boundary-tag scheme could be implemented. The boundary tag is a slight variation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

(a) Structure of a segment

merge everything

(b) Coalescence of adjacent segments
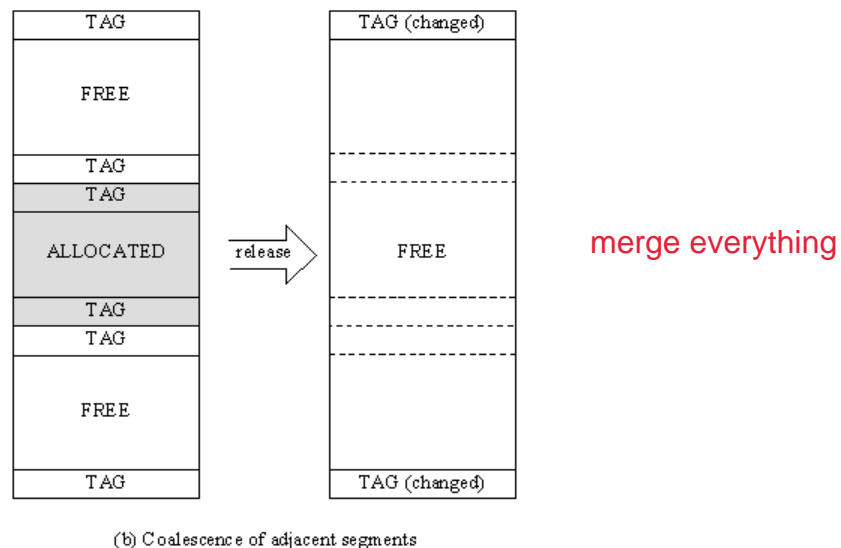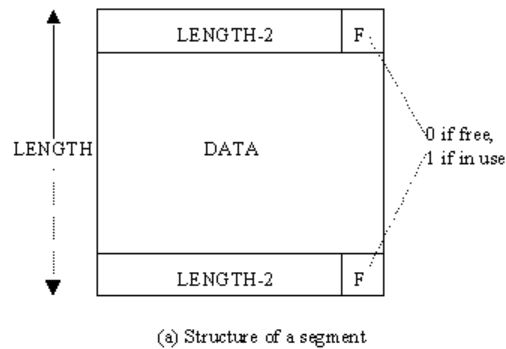
You may return a slightly larger block than what is requested from `sma_malloc()` to reduce external fragmentation.

What to Handin?

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.