Project 3 Design Document
Memory Allocation

Desmond Vehar - dvehar
Dave Lazell - dlazell
Neil Killeen - nkilleen
Melanie Dickinson - mldickin

May 21, 2014

Design

Our task was to write a wrapper around the malloc() and free() calls in order to detect problems with memory management such as leaking memory. By using a header file we include and by linking to our object file, a user program will be checked for common problems and various stats on the memory usage will be dumped upon exit.

Our program is split into three main sections:
1. slug_malloc(), the wrapper around malloc()
2. slug_free(), the wrapper around free()
3. slug_memstats(), for printing out stats on program exit

Via macro substitution in our header file, all calls to malloc() and free() will instead call our functions slug_memstats() and slug_free(). This will allow us to collect and store data about each memory allocation. The data stored for each allocation includes the current timestamp, the size of the allocated memory, the address of the allocated memory block, and the filename and line number of where the malloc() was located in the original program. All of this data is stored in a struct which is used as a node in a linked list.

A linked list structure was chosen as the internal data structure for conceptual simplicity and ease of implementation (dynamic modification: insertion, deletion). Alternative designs considered include a dynamically resized, sorted array, and a hash table. These were discarded for their high cost in dynamic resizing and overhead in authoring and memory, relative to a linked list. The speedier lookup gained from these structures was determined to be too infrequently utilized to outweigh these costs. More problematic memory management may cause any number of references to addresses that would translate into worst-case searches of a linked list, but only actually results in one such search, since our library invokes termination at the very first reference to an inactive or invalid address.

Our slug_malloc() simply creates a new node with the above information and adds this new node to our linked list. It does this by calling insert_node(). Our insert_node() function, in addition to adding the node to the linked list, also checks for a few errors: allocating memory exceeding 128 MiB, failure to allocate memory (out of memory), and attempting to allocate 0 bytes of memory. In all cases an error message is printed, and in the first two cases, the program exits. The tracked statistics (see slug_memstats()) are also updated here. Lastly, insert_node() sets up an atexit() callback on the first memory allocation. This callback will call our slug_memstats() function upon program exit.

When slug_free() is called, we check all of our allocations to determine if the address being freed exists in our list of allocations. If it doesn’t, that means that the memory has already been freed, the memory was never allocated in the first place, or the address freed is in the middle of a block of allocated memory. In all cases, the program prints an error message and exits. If the memory is allocated, we deallocate it, remove the node from out list, and update our stats.

On program exit, the atexit() handler calls slug_memstats(). This function prints information on each allocation unit that was not successfully deallocated by the user program (although not necessary, it also frees this memory). This information includes the address of the memory block, timestamp, file, and line number in which the allocation occurred. After this data (if applicable), we then print the number of allocations, total bytes allocated, the average memory allocated, and the standard deviation of the size of memory allocated. We do this for the memory blocks that are still unallocated at exit (if any), and also for the total memory allocations over the life of the program. This information is useful by the user program to determine if there are any memory leaks. This function also removes any nodes in the linked list that are currently allocated and frees any possible user leaks.

Errors Detected and Detection Limitations

Our program detects the following errors:

- Out of memory
- Attempting to allocate more than 128 MiB
- Attempting to allocate 0 bytes
- Attempting to deallocate memory that has already been deallocated
- Attempting to deallocate memory using an address that is not the first address of a memory block
- Attempting to deallocate memory twice
- Allocating and never deallocating memory (displayed at program exit)

Note that our program displays an error for the above cases, as per the assignment instructions. However, we do not store information on memory blocks that have been deallocated. To do this for programs that allocate and deallocate many small chunks would slowly increase our memory usage over time, much like a memory leak would have done. The consequence of this decision is that we are unable to differentiate between memory that has been allocated and then deallocated, and memory that that has never been allocated. Also, for speed reasons, we do not differentiate between attempted deallocation of memory inside a block, as opposed to the beginning address of a block.

Our library is limited to observation of the user program via just invocations of malloc(), free(), and program termination. It therefore does not aid in the debugging of errors or common issues dealing with memory access (e.g., off-by-one errors, read/writes just outside of allocated blocks, read/writes to uninitialized memory, read/writes to freed memory). This is consistent with the assignment specifications but severely limits the utility of a memory management aid.

Implementation

The implementation follows directly from our design, and is divided into two files, slug.h, and slug.c.

slug.h

This file is included by each program wishing to use our library to wrap malloc() and free() calls. This is accomplished by using preprocessor macros to substitute for our functions slug_malloc() and slug_free(), with added parameters for locating the calls in the user program, via filename and line number. Also, we expose slug_malloc(), slug_free(), and slug_memstats() for use by the user program. Usage documentation for each function is also included here.

slug.c

This file holds our implementation of slug_malloc(), slug_free(), and slug_memstats(). We also define a node structure for use in our linked list, helper functions, and global variables for tracking statistics of memory allocations. These globals include the number of allocations, the total size of memory allocated, the mean, and the standard deviation. Both active allocations and total allocations are tracked. The head of the linked list is also kept as a global variable, for a total of nine globals. We decided to make these global and static as consequence of their frequent use throughout the functions contained in slug.c. Making them global significantly simplifies the implementation of statistics collection and access to the internal data structure representing the core facility of slug.c, without danger of namespace pollution for the containing user directory.

void *slug_malloc (size_t size, char *WHERE)
This is the wrapper function for malloc(), called any time the user program calls malloc(). It returns the address of allocated memory. The parameter WHERE is a string constant that records the filename and line number of the caller, via the functionality of macros defined in slug.h. The private function split() is called to parse the name and number from WHERE, and the resulting information is fed into a call to insert_node().

int insert_node (int linenr, size_t mem_size, char* file_name, void** address)
The private function insert_node() contains the primary functionality of slug_malloc(), including the actual call to malloc(), insertion of allocation information into the internal data structure, updates to running statistics calculations, and basic checks for usage correctness. If the size parameter passed to malloc() is zero, it is reported to stderr as an unusual operation. If the input is excessively large (more than 128 MiB) it is reported as an error to stderr and the program is terminated. The function then updates the global variables tracking mean and standard deviation, then records the address, length, current timestamp, and location of the call in a node structure, which it finally adds to the linked list.

void slug_free (void *addr, char *WHERE)
This function first checks that the addr is the start of a valid memory region that is currently allocated by looking through the linked list. If not, an error is shown and the program terminated. If it is valid, then free() is called and the node unlinked from the linked list. We also update the global variables tracking the running mean and standard deviation. Also, the important task of setting the onexit() handler is done upon the first allocation. This causes the program to call slug_memstats() on program exit.

void slug_memstats (void)
This function traverses the linked list kept by slug_malloc() and slug_free() and prints out information about all current allocations. Each allocation report includes the size of the allocation, a timestamp for when the allocation took place, the actual address of the allocation, and file and line number in the test program where the allocation happened. In addition, a summary of all allocations is reported that includes the total number of allocations done, the number of current active allocations, the total amount of memory currently allocated, and the mean and standard deviation of sizes that have been allocated (both for all allocations not deallocated, and for total allocations made during the user program’s lifetime).

Determining the mean and standard deviation

We keep a running calculation of the mean and standard deviations of allocation sizes, since we do not keep information about individual memory allocations after they have been freed (see 1.1). To aggregate statistics covering the duration of a user program’s lifetime without incurring exponential memory expenses for long-running or rapid-fire examples, we use an incremental algorithm written by Donald Knuth [http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance]. In addition to saving space, this approach prevents a large computational cost from amassing at the end of the calling program.

Standard deviation is found by taking the square root of the mean of the squared differences of the values from their average value.

In slug_malloc(), we add to the global variables *total_size_allocated*, which is a total size of all the allocated memory, *total_alloc_count*, which is the number of total allocations, *total_mean*, which is the total mean, and *total_m2*, which is the sum of all the squares of the differences from the mean. We also do the same for the active allocations. The difference is that in slug_free(), we decrease the globals associated with the active allocations. By keeping track of these values in slug_malloc and slug_free(), we are able to take the square root of the sum of the averages squared divided by the number of allocations to obtain the standard deviation. We do that in slug_memstats() when we report it.

Testing

The goals of our testing process were as follows.

- Show a test program that correctly uses allocation in a nontrivial way behaves correctly.
- Show that after allocating and deallocating memory, trying to deallocate an invalid address is immediately detected.
- Show that after allocating and deallocating memory, trying to deallocate an already freed region is immediately detected.
- Show that after allocating and deallocating memory, trying to deallocate a valid region by passing in a pointer inside the region is immediately detected.
- Show that allocating memory and then exiting triggers the leak detector and shows where the leak occurred.
- Show that allocating memory of inappropriate size (zero or more than 128 MiB) is immediately detected, and results in a warning message or immediate termination, respectively.

Strategy

Six C programs were written to test our library's facilities on common cases of memory mismanagement. Most of these cases will cause our library to report an error to stderr, which specifies the filename and line number of the offending call, and invoke immediate program termination at the first problematic call, as per the assignment specifications. This includes deallocation of invalid addresses, deallocation of already freed memory, and deallocation of an address inside a block (as opposed to the first address). In implementation, all three cases correspond to the situation of the address passed to free() not being found in the linked list of active allocations. Memory leak is detected and reported at program termination. Additionally, running the Valgrind utility on all test files provokes the appropriate memory leak (or lack of leak) records. No memory leak is incurred by the library functions alone.

The six programs created were:

- test1 - This program shows that after allocating and deallocating memory, trying to deallocate an invalid address is immediately detected
- test2 - This program shows that after allocating and deallocating memory, trying to deallocate an an already freed address is immediately detected
- test3 - This program shows that after allocating and deallocating memory, trying to deallocate with an address inside the block, but not the address of the beginning of the block, is immediately detected
- test4 - This program creates memory leaks by allocating memory, and not deallocating it. This is shows by the slug_memstats() report on program exit
- test5 - This program allocates and deallocates memory, showing that a normal program executes correctly and shows no memory leaks
- test6 - This program attempt to allocate 0 bytes, and then 128,000,001 bytes to show that both produce errors, and the second large allocation exits the program.

Results

test1: Running this program correctly produces an error that the address attempting to be freed was the not first byte of memory allocated. (deallocating an invalid address)

test2: Running this program correctly produces an error that the address attempting to be freed was the not first byte of memory allocated. (deallocating an already freed address)

test3: Running this program correctly produces an error that the address attempting to be freed was the not first byte of memory allocated. (trying to deallocate with an address inside the block, but not the address of the beginning of the block, is immediately detected). Note that this program shows memory leaks, because per the assignment, when we detect the error, we exit before we free the memory.

test4: Running this program correctly produces a list of all the memory blocks allocated and not deallocated.

test5: Running this program correctly shows a report showing that no memory leaks were found.

test6: Running this program correctly shows a message stating the we attempted to allocate 0 bytes, and an error stating that we attempted to allocate 180,000,001 bytes, which exits correctly exits the program.

Conclusion

Our design lead to an implementation which tests show to fulfill all goals of the assignment.