CS 213, Spring 2000
Lab Assignment L3: Implementing a Conservative Garbage Collector
Assigned: Thur., March 2, Due: Wed., March 15, 11:59PM

Angela Demke Brown (`demke+@cs.cmu.edu`) is the lead person for this lab.

Introduction

In this lab you will be writing a conservative mark-and-sweep garbage collector for C programs. Your collector will replace the `free` routine from the standard C library. Your task is to develop a collector that is correct, memory efficient, and fast. To get you started, we have supplied you with a number of data structures and helper functions, however you are free to modify or replace any of the code supplied to improve the performance of your solution.

The collector that you write needs to be conservative because the C language allows arbitrary values to be treated as pointers to regions of memory. Since you can't determine precisely what values are pointers, you need to treat anything that *looks* like a pointer into the heap area as a valid pointer. To simplify the task, we will be providing you with a block of memory to manage as a garbage-collected heap.

Logistics

As usual, you may work in a group of up to 2 people.

Any clarifications and revisions to the assignment will be posted on the class bboard and WWW page.

Your programs will be evaluated by their correctness and performance on the "fish" cluster. However, you can do your code development on any machine (you may need to edit the Makefile if you change platforms).

The tarfile

```
/afs/cs.cmu.edu/class/academic/15213-s00/L3/L3.tar
```

contains the files you'll need for this assignment. You will be turning in the file `malloc.c`, after filling in the empty functions and modifying existing functions to work with your implementation.

You can hand in the assignment via "`gmake handin NAME=username`", with "`VERSION=version_num`" if necessary for hand-ins after the initial one. **If you make changes to the supplied code in any other files, please edit the Makefile so that all of your code will be submitted by "make handin"**

Specification

Your task is simply to fill in the function:
`garbage_collect(int *regs, int pgm_stack)` in `malloc.c`.

This function is given the contents of the callee-saved registers and the stack base pointer (%ebp) at the time `gc_malloc` was called. You can use the function `get_data_area` to find the start address and length of the global data area.

You will need to define additional functions to help with the task of garbage collection. The specification of these functions is entirely up to you. You may also find it useful to define a stack type to hold pointers into the heap area that you find. You may make any changes you wish to the code supplied to you (including changing the interface to `garbage_collect()` if you find this useful) with the following exceptions:

1. You may not change the interface to `gc_malloc()`.
2. You MUST call `verify_complete()` at the end of `garbage_collect`.
3. You MUST call `verify_garbage()` with the address of the first allocated byte in the garbage-collected block (i.e., the address returned previously by `gc_malloc`) for each block of memory that you free using garbage collection. (If you do not, your program will not pass our correctness tests.)
4. You may change the `driver.c` to do your own testing, but remember that we will be testing your program with the original version.

You will also need to implement some form of coalescing of free blocks to address memory fragmentation. Your solution will not be considered correct if it fails to allocate memory when enough free memory exists in consecutive bytes.

You are allowed to use the standard C library malloc/free routines during garbage collection. (This may be useful for implementing the stack type) DO NOT call your own `gc_malloc` function!

Garbage Collection

The basic idea of a mark-and-sweep garbage collector is to first determine what blocks of heap-allocated memory are still accessible to the application program (mark phase) and then free all un-accessible blocks (sweep phase). We keep a tree of all allocated blocks, sorted by the address of the header to facilitate these actions. Each tree node includes a size field which is the number of bytes of allocated memory contained in the block. Because the allocated sizes are always a factor of 8, we can use the least significant bit in the size field for the mark.

Marking proceeds by first locating all potential heap pointers directly accessible to the application. That is, all such pointers stored in global data, the program stack, or the callee-saved registers. (Caller-saved registers should be found on the caller's stack) These pointers are called the *root set*. For each potential pointer in the root set, we first verify that the address is within some valid allocated memory and then scan the block containing that pointer looking for additional pointers to other blocks.

Once all reachable blocks have been marked, the unmarked blocks are freed and all the marks are cleared. In our implementation, splay trees are used to hold the allocated blocks, and deletions cause the tree to be re-balanced, so it is a good idea to first collect all the blocks to be deleted before actually performing any deletions.

Supplied functions

Your garbage collector will work in combination with an allocator that performs allocations and manages free space in your heap area. An initial implemenation of the allocator is supplied for you. It consists of the following functions, which are declared in `malloc.h` and defined in `malloc.c`.

```
int   gc_init(void);
char *gc_malloc(size_t size);
void  gc_remove_free(char *addr, list_t *list);
void  gc_add_free(char *addr, list_t *list);
```

These functions initialize the heap area for use by the allocator, perform allocations and manage the free list. Brief descriptions of each are as follows:

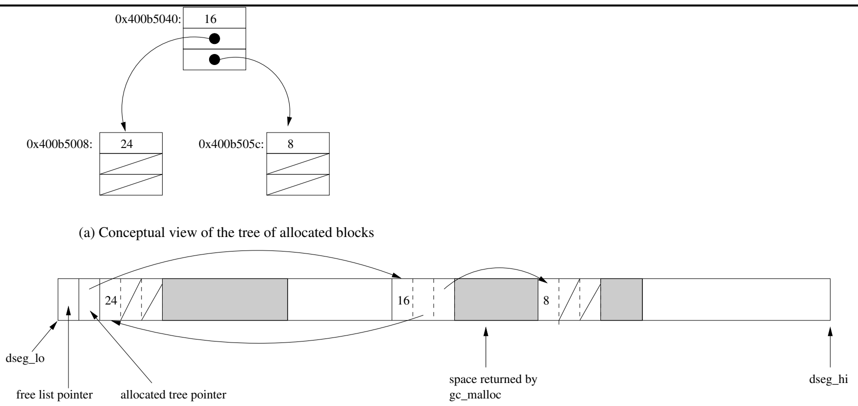
- `gc_init`: Before calling `gc_malloc`, the application program calls `gc_init` to perform any necessary initializations, including the allocation of the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise. We will grade your implementation in several phases. To facilitate our testing, `gc_init` reinitializes *all* state when it is called. We will use it to reinitialize your dynamic storage allocator between each test phase.
- `gc_malloc`: The `gc_malloc` routine returns a pointer to an allocated region of at least `size` bytes. The pointer must be aligned to 8 bytes, and the entire allocated region should lie within the memory region from `dseg_lo` to `dseg_hi`.
- `gc_remove_free`: The `gc_remove_free` routine takes a pointer to the head of the free list and a pointer to the block to remove from the free list. It simply removes the block from the list, and is used by `gc_malloc` to remove the newly allocated block from the free list. (This function is local to `malloc.c`.)
- `gc_add_free`: Given a pointer to the head of the free list, and a pointer to the block to add, `gc_add_free` places the block on the free list. It is used by `gc_malloc` when the size of the heap needs to be expanded, and by your collector to place garbage blocks back on the free list. (This function is local to `malloc.c`.)

The supplied dynamic storage allocator interacts with an arbitrary application program in the following way: As part of its initialization phase, the application calls the `gc_init` function to perform initialization of the heap, which allocates the necessary initial heap area and initializes all structures you need. The application then makes a series of calls to `gc_malloc`. There are no explicit application calls to `free`. Your garbage collector is the only means of reclaiming unused memory.

The functions you have been given in `malloc.c` make use of a number of routines from `memlib.c`.

- `mem_sbrk`: You use this function to expand the heap area. The lower and upper boundaries of the heap area are contained in `dseg_lo` and `dseg_hi` respectively. You are allowed to read these variables, but you should not modify them in any way. You must call `mem_sbrk` in order to change the upper bound. This function accepts a positive integer argument, which is the amount of bytes by which the upper bound should be expanded. The return value is the begining of the newly allocated heap area, or `NULL` if there wasn't any memory left. The interface of `mem_sbrk` is very similar to that of the `sbrk` system call, but you should use `mem_sbrk`. You cannot decrease the heap area in size, only increase it. In effect, each time you call `mem_sbrk`, the value of `dseg_hi` is incremented by the amount you request, but the actual memory allocated is always in multiples of the system page size. `gc_malloc` always calls `mem_sbrk` with an increment that is a multiple of the page size. You can call `mem_pagesize()` to find out the system page size.

- `mem_usage`: This is simply a shorthand that returns the current size of the heap in bytes.



(a) Conceptual view of the tree of allocated blocks

(b) Layout of tree and allocated memory in the heap area

Figure 1: Tree of allocated blocks, conceptually and laid out in the heap.

When performing garbage collection, a key task is to determine if a pointer into the heap (i.e. one that lies between `dseg_lo` and `dseg_hi`) points to allocated memory or not. Further, for any pointer into allocated memory, we need to be able to locate the start of the header for that block. To facilitate this task, we keep a tree of allocated blocks, sorted by address. Each tree node stores the size of the allocated memory returned by `gc_malloc`, as well as pointers to its left and right children. The address of each node is used as the key value for insertions, deletions, and searches. The tree nodes are themselves stored in the heap area that you will be managing, each node forms a *header* for an allocated block. Figure 1a shows a conceptual diagram of a tree with three nodes, while Figure 1b shows how this tree is stored in the heap.

The implementation we have given you uses splay trees to keep track of allocated blocks. The functions you may call are in `splay_tree.c`, and are defined in `useful.h`. Note that searching the splay tree causes the tree to change, so you will need to be careful to keep your pointer to the root of the tree (stored in the heap area) up to date. You are free to replace this implementation with your own if you wish.

Test driver

The file `driver.c` contains the actual driver program we will use to partially test your garbage collector. The driver program uses a series of artificial traces that allocate memory and clear any pointers to the allocated memory. For these traces, all of the valid pointers will be found in the root set (i.e. in the stack or global data). The driver program also calls several real program tests that use dynamically allocated memory. The subdirectory `15213-s00/L3/tests` contains the `.o` files for these tests. Feel free to use any other testing method you wish while developing your code. The test driver should provide you with some useful information for debugging your program. The command line options it accepts are as follows:

`-f` *tracefile* ...	Use a particular tracefile for testing; can repeat this option to load multiple tracefiles. If no tracefiles are specified, the default set of tracefiles is used.
`-v`	Verbose mode; print out some detailed debugging info (default).
`-q`	Quiet mode.
`-h`	Print a help (usage) message.
`-d`	Generate a dump of your allocator's internal operation (e.g. the pointers it returns and the `mem_sbrk` calls it initiates) into a text file.
`-t` *tolerance*	Specify an error tolerance for the time measurements (default: 0.05)

Grading Criteria

Your dynamic storage allocator will be evaluated in four areas: correctness, memory efficiency, running time, and style. There are a total of 60 points.

Correctness (50 points)

To be correct, your garbage collector must satisfy three primary conditions:

1. No memory which is still in use by the application program may be garbage collected.
2. `gc_malloc` must not fail when there is enough unused memory in consecutive bytes to satisfy the request.
3. The garbage collector must collect ALL unreachable blocks when it run.

Further, your implementation must ensure that `gc_malloc` continues to satisfy its correctness constraints. In particular, any modifications you make to implement coalescing should not break `gc_malloc`. To be correct, your `gc_malloc` routine must return `NULL` if it cannot find a sufficiently large free block. Otherwise, it must return a pointer, aligned to 4 bytes, to an allocated block of at least the requested size (the block might be larger because of alignment constraints or placement policies in your allocator). The block must be located within the allocated heap (between `dseg_lo` and `dseg_hi`), and no part of the block may be returned by subsequent calls to `gc_malloc` until it has been garbage collected.

You will receive 20 points if your garbage collector passes all of the synthetic tests used by `driver.c`. The remaining 30 points are attained by passing each of the real program tests, with 5 points awarded for each test your program passes.

Performance (5 points)

There are two main aspects to the performance of the garbage collector: space utilization and running time. Your implementation will be evaluated on both. A number of traces will be used to test your allocator in addition to real programs; some are artificially generated for the purpose of testing the behaviour of your code in various situations and others have been obtained from real-world applications. We are providing you with all the traces that we will use to evaluate your allocator. These can be found at `/afs/cs/academic/class/15213-s00/L3/traces/`.

We will use two performance metrics to evaluate your allocator. The first metric is *space utilization* and the second is *execution time*.

Space utilization is defined as the aggregate amount of memory requested by the driver (via gc_malloc) and still accessible to the size of the heap used by your allocator (dseg_hi-dseg_lo). Space utilization fluctuates during the execution of the tests as the heap is expanded and as memory is allocated and collected. The performance metric we will use for space utilization is the peak utilization, U, defined as the maximum amount of memory allocated (and still in use) by the application at any point in time during its execution to the final heap size.

The optimal space utilization is, of course, $U_{opt} = 1$, which corresponds to 0% space lost to overhead and fragmentation, and the limits of conservative collection. Although U_{opt} is unachievable, you should come quite close to it. There are several factors that influence space utilization, including fragmentation, overhead, and frequency of garbage collection. One way to ensure that fragmentation does not get out of hand is to coalesce adjacent free blocks. You can either do immediate coalescing in gc_add_free, or do lazy coalescing - as long as gc_malloc never fails when enough memory is available in consecutive free blocks. Space utilization is also influenced by the amount of overhead, (the space used by your allocator for its internal housekeeping), and by your decision of when to garbage collect.

The second metric is execution time, T. To do well on this metric, you have to expend a minimal number of instructions when allocating and freeing memory in the common case. The memory manager is a critical part of the runtime system. Therefore, it is very important to optimize it in every way possible. We will be measuring the speed of your implementation using a number of different workloads. You should think about how to write the code in such a way as to minimize the number of instructions required for the common case. When designing your implementation, try to make choices that simplify the code, e.g. that result in fewer instructions, need fewer conditionals, etc.

The performance of your allocator is summarized by a performance index, P, which is a linear sum of the utilization and execution time metrics. The index is biased towards the second (default $w = .25$):

$$P = w\frac{U}{U_{opt}} + (1-w)Min(1, \frac{T_{ref}}{T}) = wU + (1-w)Min(1, \frac{T_{ref}}{T})$$

The first part of the performance index, $w \cdot U$, is the contribution of the space utilization metric. At best, this term is equal to w, which occurs when your space utilization U is 1.

The second part of the index, $(1-w) \cdot Min(1, \frac{T}{T_{ref}})$ is contributed by the execution time metric. T_{ref} is the execution time of our reference implementation. At best, when your throughput equals or exceeds that of our implementation, T_{ref}, this part is equal to $1-w$. Thus, ideally the performance index is $P = w+(1-w) = 1$ or 100%.

The reason for the Min in the second term is to make sure that a very fast solution that exceeds our reference's execution time while doing poorly on space utilization does not get a high performance index. The motivation is that space and time are both expensive resources and your solution should be balanced in optimizing for both. Because the summary performance index depends both on the space and time performance, **you should not optimize speed at the expense of memory overhead, or vice-versa.**

The driver program reports the performance index, P, of your allocator as a percentage.

Over the entire set of default traces and tests, your execution time should approximate T_{ref}, or even excede it.

The final grading will be done on a curve, after we have reviewed the performance results from all your implementations. You can see how good your implementation is by checking the statistics web page. Note that if you fail the correctness tests, you will not get any points for performance.

Style (5 points)

Your code should be readable and commented. Define macros or subroutines as necessary to make the code more understandable. Keep in mind that when your code gets more and more complicated, your performance is likely to suffer. Smart design decisions and optimizations will tend to make your code smaller.

Automated Testing/Grading System

Feel free to modify `driver.c` to do your own testing and debugging, but it would be a good idea if you used `driver.c` located in `/afs/cs/academic/class/15213-s00/L3/src/` for your final tests.

You can evaluate the performance of your allocator by running the tests locally. Note that due to variability in load on the cluster machines, the performance reported by the local driver may be inaccurate (usually slightly slower) than what it would be when your solution is graded.

We will be using a Web-based automated testing and grading system. You can submit your `malloc.c` to this testing and grading system at any time, and as many times as you wish, by doing a "`gmake update NAME=username`". Your code will be tested for the above criteria, and the results will be posted to a web page every few minutes. This will allow you to check your implementation for correctness and to gauge the performance of your implementation against those of other groups.

NOTE: If you have modified any files other than `malloc.c` for your solution, the automated testing and grading system will not work.

You can hand in your assignment by doing "`gmake handin user=USERNAME`", with `version=VERSION`" if necessary for hand-ins after the initial one.

Hints

- Debugging: Dynamic memory allocators and garbage collectors are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of `(void *)` pointer references. It may be helpful to write functions that print the state of your collector's data structures which you can use when debugging your program.
- Dumps: For debugging purposes, you may find the `-d` option helpful. Feel free to modify the dump routines to report more information if you like.
- Traces: During initial development, using shorter traces may simplify debugging and testing. We have placed two short trace files in `L3/traces/short/short{1,2}.rep` that you can use by invoking the `-f` option of the driver.
- Performance: When optimizing performance, you may find the `gprof` tool helpful. This tool produces an execution profile of your program. It calculates the amount of time spent in each routine. To use `gprof`, you will need to turn on the `gprof` flags when compiling your program:

  ```
  bass>gmake clean
  bass>gmake GPROF=-pg
  bass>gmake GPROF=-pg
  ```

  When you run your executable, say `malloc`, a file named `gmon.out` is created in your current working current directory. To view the profile information in this file, you can invoke `gprof` as follows:

```
bass>gprof malloc gmon.out
```