CS 213, Fall 1998
Lab Assignment L2: Implementing a Dynamic Storage Allocator
Assigned: Oct. 8, Due: Wed., Oct. 21, 11:59PM

Kip Walker (`kwalker+ta@cs.cmu.edu`) is the lead person for this lab.

Introduction

In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version of the $malloc$ and $free$ routines from the standard C library. Your task is to develop an allocator that is correct, memory efficient, and fast. All implementation decisions are up to you! You will need to be very creative to write a good allocator.

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 class Alphas. However, you can do your code development on any machine.

The tarfile

```
/afs/cs.cmu.edu/class/academic/15213-f98/L2/L2.tar
```

contains the files `Makefile`, `malloc.h`, `malloc.c`, and `test_malloc.c`. You will be turning in the file `malloc.c`, after filling in the empty functions with your implementation. The file `test_malloc.c` is provided for you to write code for testing your `L2_malloc` and `L2_free`.

"`make mtest`" will build `mtest`, with the `main` routine from `test_malloc.c`.

As usual, handin via "`make handin NAME=username`", with "`VERSION=version_num`" if necessary for handins after the initial one.

Details

Your dynamic storage allocator consists of the following three functions, which are declared in `malloc.h` and defined in `malloc.c` with empty function bodies.

```
int   init_malloc(long int heap_size, char *heap);
char *L2_malloc(long int size);
void  L2_free(char *block);
```

You will fill in these empty function bodies (and possibly define other private functions) as your solution to this lab assignment. **You are not allowed to change the interfaces, nor to call any system routines that manage dynamic storage (e.g.,** ***malloc*****,** ***free*****,** ***sbrk*****, etc.)**

Your dynamic storage allocator interacts with an arbitrary application program in the following way: As part of its initialization phase, the application allocates a contiguous block of memory from the system heap using either the system *malloc* or *sbrk* routines. This block of memory is the *heap area* that your dynamic storage allocator will manage for the application using its implementations of *L2_malloc* and *L2_free*.

After it creates its heap area, and before it makes any calls to *L2_malloc* or *L2_free*, the application informs your dynamic storage allocator of the location and size of its heap area by calling the *init_malloc* function, which initializes the heap area. The application then makes a series of calls to *L2_malloc* and *L2_free*.

- *Init_malloc*. After the application program allocates its heap area, it calls *init_malloc* with the size (in bytes) and location of this heap area. The *init_malloc* routine initializes this heap area in whatever way is necessary for your implementation. We will grade your implementation in several phases. To facilitate our testing, write *init_malloc* such that it reinitializes *all* state when it is called. We will use it to reinitialize the heap and your dynamic storage allocator between each test phase.
- *L2_malloc*. The *L2_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 `heap` to (`heap`+`heap_size`-1]).
- *L2_free*. The *L2_free* routine is only guaranteed to work when it is passed pointers to allocated blocks that were returned by previous calls to *L2_malloc*. The *L2_free* routine should add the block to the pool of unallocated blocks, making the memory available to future *L2_malloc* calls.

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 (20 points)

To be correct, your *L2_malloc* routine must return *NULL* if it cannot find a sufficiently large free block. Otherwise, it must return a pointer, aligned to 8 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 original heap passed to *init_malloc*, and no part of the block may be returned by subsequent calls to *L2_malloc* until it has been released by a call to *L2_free*.

The correctness criteria are all or nothing. If your implementation is correct by this definition, you will receive all 20 points, otherwise you will receive 0 points.

Memory Efficiency (20 points)

There are several factors involved in evaluating the memory efficiency of your allocator:

1. **Coalescing (10 points).** Does your *L2_free* routine coalesce free blocks? You may either do immediate coalescing in *L2_free*, or do lazy coalescing - as long as *L2_malloc* never fails when enough memory is available in consecutive free blocks. There is no partial credit for this section.
2. **Overhead (5 points).** How much of the heap does your memory manager prevent the application from using due to overheads and internal fragmentation? This will be measured by performing several allocation sequences, and calculating the fraction of the heap that was returned in the form of allocated blocks. The overhead points will be awarded on a curve, with the cutoffs set after we see the distribution for the whole class.
3. **Minimum free block size (5 points)**. This is equivalent to the size of the smallest block that can be allocated.
   - 0 – 24 bytes: 5 points.
   - 24 – 40 bytes: 2 points.
   - $>$ 40 bytes: 0 points.

Running time (15 points)

The memory manager is such a critical part of a runtime system that it is very important to optimize 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.

We will award speed points on a curve, with the cutoffs set after we see the distribution for the entire class. If your allocator fails for any reason during our tests (i.e., you run out of heap space because of excessive fragmentation or failure to coalesce), then you won't receive any speed points. However, you will be able to run our tests yourself while you are developing your allocator, so there shouldn't be any surprises.

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

We will have an automated testing and grading system in place shortly. You will submit your `malloc.c` file in a directory; it will be tested for the above criteria, and the results will be posted to a web page. This will allow you to check your implementation for correctness and to gauge the performance of your implementation against those of other groups. Details on this system will be announced next week.

Hints

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of `(void *)` pointer references. They are difficult to program efficiently because running time is influenced by a number of factors, including the degree of fragmentation, the behavior of the application, the placement policy of the allocator, and the low-level mechanisms that implement the placement policy.

To help you better understand the behavior of your program, you might find a program called *Atom* to be helpful. Atom provides a simple but extremely powerful and general mechanism for building tools that navigate and instrument executable Alpha object files. You can write your own Atom tools from scratch (not recommended for this Lab), or you can use a wide variety of existing Atom tools. Examples include:

- *gprof:* procedure-level execution time profiling.
- *iprof:* procedure-level instruction profiling.
- *liprof:* basic-block level instruction profiling.
- *syscall:* system call performance summary.
- *3rd:* the *3rd Degree* memory checker and leak finder (similar to Purify).
- *pixie:* basic block profiling (like the pixie(1) command).

See `/afs/cs.cmu.edu/class/academic/15213-f98/L2/atom` for documentation and an example of an Atom tool (*ptrace*), which produces an instrumented version of "hello, world" (`hello.ptrace`) that traces every procedure call. The output is amazing!