Lab 5: Writing a Dynamic Storage Allocator

Assigned	Friday, May 24, 2013
Due Date	*Friday, June 7, 2013 at 5:00pm (absolute deadline June 10, 2013 at 5:00pm)*
Files	lab5.tar.gz

Overview

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.

Instructions

Start by extracting `lab5.tar.gz` to a directory on `attu` in which you plan to do your work, by typing:

```
wget cs.washington.edu/education/courses/cse351/13sp/labs/lab5/lab5.tar.gz
tar xzvf lab5.tar.gz
```

This will cause a number of files to be unpacked in a directory called lab5. The only file you will modify and turn in is `mm.c`.

(In the following instructions, we will assume that you are executing programs in your local directory on `attu`. For this lab, you can work anywhere there's a C compiler and `make`, but make sure your allocator works on `attu`, where we'll be testing it.)

Your dynamic storage allocator will consist of the following three functions (and several helper functions), which are declared in `mm.h` and defined in `mm.c`:

```
int   mm_init(void);
void* mm_malloc(size_t size);
```

```
void  mm_free(void* ptr);
```

The `mm.c` file we have given you partially implements an allocator using an explicit free list. Your job is to complete this implementation by filling out `mm_malloc()` and `mm_free()`. The three main memory management functions should work as follows:

- `mm_init()` (provided): Before calling `mm_malloc()` or `mm_free()`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls mm_init to perform any necessary initializations, such as allocating the initial heap area. The return value is -1 if there was a problem in performing the initialization, 0 otherwise.
- `mm_malloc()`: The `mm_malloc()` routine returns a pointer to an allocated block payload of at least `size` bytes. (`size_t` is a type for describing sizes; it's an unsigned integer that can represent a size spanning all of memory, so on x86_64 it is a 64-bit integer.) The entire allocated block should lie within the heap region and should not overlap with any other allocated block.
- `mm_free()`: The `mm_free()` routine frees the block pointed to by `ptr`. It returns nothing. This routine is only guaranteed to work when the passed pointer (`ptr`) was returned by an earlier call to `mm_malloc()` and has not yet been freed. These semantics match the the semantics of the corresponding malloc and free routines in libc. Type `man malloc` in the shell for complete documentation.

We will compare your implementation to the version of malloc supplied in the standard C library (libc). Since the libc malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers.

Provided Code

We define a `BlockInfo` struct designed to be used as a node in a doubly-linked explicit free list, and the following functions for manipulating free lists:

- `BlockInfo* searchFreeList(int reqSize)`: returns a block of at least the requested size if one exists (and `NULL` otherwise).
- `void insertFreeBlock(BlockInfo* blockInfo)`: inserts the given block in the free list in LIFO manner.
- `void removeFreeBlock(BlockInfo* blockInfo)`: removes the given block from the free list.

In addition, we implement `mm_init` and provide two helper functions implementing important parts of the allocator:

- `void requestMoreSpace(int incr)`: enlarges the heap by `incr` bytes (if enough memory is available on the machine to do so).
- `void coalesceFreeBlock(BlockInfo* oldBlock)`: coalesces any other free blocks adjacent in memory to `oldBlock` into a single new large block and updates the free list accordingly.

Finally, we use a number of C Preprocessor macros to extract common pieces of code (constants, annoying casts/pointer manipulation) that might be prone to error. Each is documented in the code. You are welcome to use macros as well, though the ones already included in `mm.c`are the only ones we used in our sample solution, so it's possible without more. For more info on macros, check the GCC manual.

- `FREE_LIST_HEAD`: returns a pointer to the first block in the free list (the head of the free list).
- `POINTER_ADD` and `POINTER_SUB`: useful for doing pointer arithmetic without worrying about the size of BlockInfo struct.
- Other short utilities for extracting the size field and determining block size.

Additionally, for debugging purposes, you may want to print the contents of the heap. Here's code for a procedure to do so:

```
/* Print the heap by iterating through it as an implicit free list. */
static void examine_heap() {
  BlockInfo *block;

  /* print to stderr so output isn't buffered and not output if we crash */
  fprintf(stderr, "FREE_LIST_HEAD: %p\n", (void *)FREE_LIST_HEAD);

  for(block = (BlockInfo *)POINTER_ADD(mem_heap_lo(), WORD_SIZE); /* first
block on heap */
      SIZE(block->sizeAndTags) != 0 && block < mem_heap_hi();
      block = (BlockInfo *)POINTER_ADD(block, SIZE(block->sizeAndTags))) {

    /* print out common block attributes */
    fprintf(stderr, "%p: %ld %ld %ld\t",
    (void *)block,
    SIZE(block->sizeAndTags),
    block->sizeAndTags & TAG_PRECEDING_USED,
    block->sizeAndTags & TAG_USED);

    /* and allocated/free specific data */
    if (block->sizeAndTags & TAG_USED) {
      fprintf(stderr, "ALLOCATED\n");
```

```
    } else {
      fprintf(stderr, "FREE\tnext: %p, prev: %p\n",
      (void *)block->next,
      (void *)block->prev);
    }
  }
  fprintf(stderr, "END OF HEAP\n\n");
}
```

Memory System

The `memlib.c` package simulates the memory system for your dynamic memory allocator. In your allocator, you can call the following functions (if you use the provided code for an explicit free list, most uses of the memory system calls are already covered).

- `void* mem_sbrk(int incr)`: Expands the heap by `incr` bytes, where `incr` is a positive non-zero integer and returns a pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix `sbrk` function, except that `mem_sbrk` accepts only a positive non-zero integer argument. (Run `man sbrk` if you want to learn more about what this does in Unix.)
- `void* mem_heap_lo()`: Returns a pointer to the first byte in the heap
- `void* mem_heap_hi()`: Returns a pointer to the last byte in the heap.
- `size_t mem_heapsize()`: Returns the current size of the heap in bytes.
- `size_t mem_pagesize()`: Returns the system's page size in bytes (4K on Linux systems).

The Trace-driven Driver Program

The driver program `mdriver.c` in the `lab5.tar.gz` distribution tests your `mm.c` package for correctness, space utilization, and throughput. Use the command `make` to generate the driver code and run it with the command `./mdriver -V` (the `-V` flag displays helpful summary information as described below).

The driver program is controlled by a set of *trace files* that are posted on `attu` (if you want to work on another computer, you can copy these files and then update the TRACEDIR path in `config.h`). Each trace file contains a sequence of allocate and free directions that instruct the driver to call your `mm_malloc` and `mm_free` routines in some sequence. The driver and the trace files are the same ones we will use when we grade your submitted `mm.c` file.

The `mdriver` executable accepts the following command line arguments:

- `-t <tracedir>`: Look for the default trace files in directory `tracedir` instead of the default directory defined in `config.h`.
- `-f <tracefile>`: Use one particular `tracefile` for testing instead of the default set of tracefiles.
- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc` malloc in addition to the student's malloc package.
- `-v`: Verbose output. Print a performance breakdown for each tracefile in a compact table.
- `-V`: More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing your malloc package to fail.

Programming Rules

- You should not change any of the interfaces in `mm.c`.
- You should not invoke any memory-management related library calls or system calls. This excludes the use of `malloc`, `calloc`, `free`,`realloc`, `sbrk`, `brk` or any variants of these calls in your code. (You may use all the functions in `memlib.c`, of course.)
- You are not allowed to define any global or `static` compound data structures such as arrays, structs, trees, or lists in your `mm.c`program. You *are* allowed to declare global scalar variables such as integers, floats, and pointers in `mm.c`, but try to keep these to a minimum. (It is possible to complete the implementation of the explicit free list without adding any global variables.)
- For consistency with the `malloc` implementation in `libc`, which returns blocks aligned on 8-byte boundaries, your allocator must always return pointers that are aligned to 8-byte boundaries. The driver will enforce this requirement for you.

Evaluation

Your grade will be calculated (as a percentage) out of a total of 60 points as follows:

- **Correctness (45 points).** You will receive 5 points for each test performed by the driver program that your solution passes. (9 tests)
- **Style (10 points).**
    - Your code should use as few global variables as possible (ideally none!).

    - Your code should be as clear and concise as possible.
    - Since some of the unstructured pointer manipulation inherent to allocators can be confusing, short inline comments on steps of the allocation algorithms are also recommended. (These will also help us give you partial credit if you have a partially working implementation.)
    - Each function should have a header comment that describes what it does and how it does it.
- **Performance (5 points).** Performance represents a small portion of your grade. We are most concerned about the correctness of your implementation. For the most part a correct implementation will yield reasonable performance. Two performance metrics will be used to evaluate your solution:
    - *Space utilization*: The peak ratio between the aggregate amount of memory used by the driver (i.e., allocated via `mm_malloc` but not yet freed via `mm_free`) and the size of the heap used by your allocator. The optimal ratio is 1. You should find good policies to minimize fragmentation in order to make this ratio as close as possible to the optimal.
    - *Throughput*: The average number of operations completed per second.

  The driver program summarizes the performance of your allocator by computing a **performance index**, P, which is a weighted sum of the space utilization and throughput:

```
P = 0.6U + 0.4 min (1, T/Tlibc)
```

  where U is your space utilization, T is your throughput, and T_{libc} is the estimated throughput of `libc` malloc on your system on the default traces. The performance index favors space utilization over throughput. You will receive 5(P+ 0.1) points, rounded **up** to the closest whole point. For example, a solution with a performance index of 0.63 or 63% will receive 4 performance points. Our complete version of the explicit free list allocator has a performance index between 0.7 and 0.8; it would receive 5 points. Observing that both memory and CPU cycles are expensive system resources, we adopt this formula to encourage balanced optimization of both memory utilization and throughput. Ideally, the performance index will reach P = 1 or 100% . To receive a good performance score, you must achieve a balance between utilization and throughput.

Hints

Getting Started

- Read these instructions.
- Read over the provided code.
- Take notes while doing the above.
- Draw some diagrams of how the data structures should look before and after various operations.

Debugging

- Use the `mdriver -f` option. During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (`short1-bal.rep` and `short2-bal.rep`) that you can use for initial debugging.
- Use the `mdriver -v` and `-V` options. The `-v` option will give you a detailed summary for each trace file. The `-V` will also indicate when each trace file is read, which will help you isolate errors.
- Compile with `gcc -g` and use `gdb`. The `-g` flag tells `gcc` to include debugging symbols, so `gdb` can follow the source code as it steps through the executable. The `Makefile` should already be set up to do this. A debugger will help you isolate and identify out of bounds memory references. You can specify any command line arguments for `mdriver` after the `run` command in `gdb` e.g. `run -f short1-bal.rep`.
- Understand every line of the malloc implementation in the textbook. The textbook has a detailed example of a simple allocator based on an implicit free list. Use this is a point of departure. Don't start working on your allocator until you understand everything about the simple implicit list allocator.
- Write a function that treats the heap as an implicit list, and prints all header information from all the blocks in the heap. Using `fprintf`to print to `stderr` is helpful here because standard error is not buffered so you will get output from your print statements even if the next statement crashes your program.
- Encapsulate your pointer arithmetic in C preprocessor macros. Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. We have supplied maros that do this:
  see `POINTER_ADD` and `POINTER_SUB`.
- Use a profiler. You may find the `gprof` tool helpful for optimizing performance. (`man gprof` or searching online for `gprof` documentation will get you the basics.) If you use `gprof`, see the hint about debugging above for how to pass extra arguments to GCC in the`Makefile`.

- Start early! It is possible to write an efficient malloc package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far in your career. So start early, and good luck!

Heap Consistency Checker

Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of untyped pointer manipulation. In addition to the usual debugging techniques, you may find it helpful to write a heap checker that scans the heap and checks it for consistency.

Some examples of what a heap checker might check are:

- Is every block in the free list marked as free?
- Are there any contiguous free blocks that somehow escaped coalescing?
- Is every free block actually in the free list?
- Do the pointers in the free list point to valid free blocks?
- Do any allocated blocks overlap?
- Do the pointers in a heap block point to valid heap addresses?

Your heap checker will consist of the function `int mm_check(void)` in `mm.c`. Feel free to rename it, break it into several functions, and call it wherever you want. It should check any invariants or consistency conditions you consider prudent. It returns a nonzero value if and only if your heap is consistent. This is not required, but may prove useful. When you submit `mm.c`, make sure to remove any calls to `mm_check` as they will slow down your throughput.

Extra Credit

As optional extra credit, implement a final memory allocation-related function: `mm_realloc`. The signature for this function, which you should add to your `mm.h` file, is:

```
extern void* mm_realloc(void* ptr, size_t size);
```

Similarly, you should add the following to your `mm.c` file:

```
void* mm_realloc(void* ptr, size_t size) {
  // ... implementation here ...
}
```

To receive credit, you should follow the contract of the C library's `realloc` exactly (pretending that `malloc` and `free` are `mm_malloc` and`mm_free`, etc.). The man page entry for `realloc` says:

```
The realloc() function changes the size of the memory block pointed to
by ptr to size bytes. The contents will be unchanged in the range from
the start of the region up to the minimum of the old and new sizes. If
the new size is larger than the old size, the added memory will not be
initialized. If ptr is NULL, then the call is equivalent to
malloc(size), for all values of size; if size is equal to zero, and
ptr is not NULL, then the call is equivalent to free(ptr). Unless ptr
is NULL, it must have been returned by an earlier call to malloc(),
calloc() or realloc(). If the area pointed to was moved, a free(ptr)
is done.
```

A good test would be to compare the behavior of your `mm_realloc` to that of `realloc`, checking each of the above cases. Your implementation of `mm_malloc` should also be performant. Avoid copying memory if possible, making use of nearby free blocks. You should not use `memcpy` to copy memory; instead, copy `WORD_SIZE` bytes at a time to the new destination while iterating over the existing data.

Submitting Your Work

Submit your `mm.c` file to the Catalyst Drop Box for this assignment.