COSC 301: Operating Systems
Lab 6: The Mr. Forgetful malloc library
Due: 2 November 2011
Fall 2011

Contents

1 Overview

In this project, you will write a memory management library for user applications. Your library will implement the `malloc` and `free` functions that you have used in programs so far.

The goals for this lab are to learn how malloc and free actually work and how heap memory is managed, to learn how some memory corruption problems can be detected, and to understand better how pointers and pointer arithmetic work in C.

As with the past couple labs, work on this lab in a group of 2–3 students. Only one submission needs to be made for the group. Please make a note in the file(s) which students worked on it.

Important note: there are two stages to achieve in this lab. Each stage requires more sophisticated programming and provides more features in your malloc library. The stages are structured such that you can still achieve a good grade even if you only get through stage 1. Work through the stages sequentially, e.g., don't try to incorporate stage 2 features unless you're confident that everything works well for stage 1. There is also an opportunity for a little bit of bonus credit in this lab by augmenting your heap management library so that it can run underneath any arbitrary program.

2 Detailed Description

2.1 Getting Started

In this lab, you will create a library that implements the necessary calls to support heap memory allocation and deallocation for applications. We will implement our own versions of the standard `malloc` and `free` calls (as well as one other function). As a result, our test programs will look like "normal" programs that use `malloc` and `free`. We'll compile the library in two ways: one that makes it easy to test our library, and another that allows us to "insert"

our library between any program and the standard C library, thus intercepting all calls to `malloc` and `free`. Very subversive.

Because we are (re-)implementing `malloc` and `free` in this lab, you will not be able to use the built-in C-library versions of these functions! That is, if you wish to allocate heap memory for data structures used by your library, you must use your own home-grown mechanisms — that's the key point of this lab. We will instead use the `sbrk` system call (described below) to request a larger heap memory segment from the OS, and manage the heap within our library.

The following starter files are posted on Moodle:

- **SConstruct**: your scons configuration file. Note that the SConstruct file will actually create two final products: an executable test program (`forgettest`) and a shared library (`libforgettest.so`) that can be inserted between any program and the standard C library and OS. For all "normal" testing, you should use the `forgettest` program.
- **forgetful.c**: the source file for your malloc and free functions.
- **forgetful.h**: a header file for your library. You shouldn't have to modify this file at all.
- **forgettest.c**: a test program that you can use to exercise your malloc and free functions. You can feel free to modify this program to add or modify any test cases.

The test program supplied in the starter files has 4 different test cases for stage 1 functionality, each of increasing complexity, as well as 1 test case for stage 2. To run this program and specify different test cases, you can use the following command lines:

```
1 $ ./forgettest -t1  # runs test 1 (stage 1)
2 $ ./forgettest -t2  # runs test 2 (stage 1)
3 $ ./forgettest -t3  # runs test 3 (stage 1)
4 $ ./forgettest -t4  # runs test 4 (stage 1)
5 $ ./forgettest -t5  # runs test 5 (stage 2)
6 $
```

2.2 Stage 1 functionality: malloc, free, and dumpMemoryMap (worth 90%)

For stage 1 functionality, you will need to implement the following functions:

- `void *malloc(size_t)`.

  Basic malloc functionality, using the algorithm of your choice for allocating memory from a free list that you manage and maintain. (Suggestion: start with first fit. Keep it simple.)

  The type `size_t` behaves like the type unsigned int. Your `malloc` should return a pointer to a block of memory of the requested size as void *, or NULL if there's no memory left in the system.

- `void free(void *)`.

  Basic free functionality; it should free the block of memory pointed to by the given argument by returning the block of memory back to your free list.

  `free(NULL)` should have no effect.

- `void dumpMemoryMap(void)`.

  This function should print a "map" of allocated and free memory in the heap. (An example is shown below for what output from this function should look like.)

The main task in this lab is to keep track of free and allocated memory blocks in the heap segment of a process's address space. A good way to do this is with a "free list". *You may choose how to maintain your free list.* The "Tips and Hints" section below suggests one possible way; you are free to design your implementation in the way suggested in that section, or in any other way you like.

Initially, your heap segment will be of size zero. In order to obtain memory from the operating system to allocate to user processes, you will need to use the `sbrk` system call. Given a size (in bytes), `sbrk` increases the heap segment

size by that number of bytes and returns a pointer to the first byte in a block of memory of the requested size. That is, `sbrk(10)` will increase your heap segment size by 10 bytes and return a pointer to the first byte of a block of 10 bytes (or in other words, a pointer to the first byte of the last 10 bytes of the heap segment). You can use `sbrk(0)` to obtain the beginning (virtual) address of the heap segment. Note that you can never decrease the heap segment size; it only grows. (Thus within your implementation of `free()`, there won't be any calls to `sbrk()`.)

- When you allocate a block, you should: search the free list for a large enough block to satisfy the request. If a large enough block is found, you should update the free list. Note that if a block on the free list is somewhat larger that the allocation request, you should split the block in two, leaving the difference between the free block size and the allocation request on the free list. It is up to you to decide on thresholds to determine when and how to split a larger free block in two.
- If there is not a large enough block on the free list to satisfy the request, you will have to increase the heap segment size with the `sbrk` function as discussed above. Note that the first allocation request should *always* cause the heap segment size to increase.
- Upon a call to free(), you should add the block to the free list, leaving the free list ordered by memory address. Adjacent free blocks should be coalesced into a single free block.

Your `dumpMemoryMap` function should contain the capability to print a representation of the heap, including free blocks and allocated blocks of memory. (You should identify each free block, but you do not need to identify each allocated block; it is sufficient to identify a "region" of allocated memory.)

An example of what this function might print out (you can choose your own format) is:

```
block  size  alloc?  addresses
-----  ----  ------  ---------
    0    10      no  0-9
    1    20     yes  10-29
    2   100      no  30-129
    3   300     yes  130-429
```

You may wish to invoke this function at different points while testing in order to verify that your malloc and free functions are working correctly. Your library should register the `dumpMemoryMap` function to be called automatically when a test program exits. There is a system call that enables this: `atexit()`. Note that `atexit()` should only be called once from within your library. (Note that there is already some code in the starter file that does this.)

2.3 Stage 2 functionality: limited detection of memory corruption (worth 10%)

For stage 2, you will add some functionality to your library to detect buffer overruns and other invalid writes to memory. To accomplish this, you will add *guard bytes* to the beginning and end of an allocated memory block and set them to known values. The guard bytes should essentially bracket the bytes allocated for the user. If a user program writes something other than the known value into one of these guard bytes, you will be able to detect that when the memory block is freed.

Adding the guard bytes and setting them to a particular known value should be specified by setting two environment variables: `FORGET_GUARD_SIZE` and `FORGET_FILL_BYTE`.

For example, consider the following variable settings:

```
$ export FORGET_GUARD_SIZE=2
$ export FORGET_FILL_BYTE=0xde
```

`FORGET_GUARD_SIZE` specifies the number of bytes to be added to the beginning and to the end of a memory block. `FORGET_FILL_BYTE` specifies the value that these bytes should be initialized to. With the above settings, if a malloc request for 10 bytes is made, your library should actually allocate: 10 (user request) + 2 * 2 (guard bytes at the beginning and end of the block) + any memory required for free list bookkeeping.

If `FORGET_GUARD_SIZE` is 0 or less, you should not add any guard bytes. If `FORGET_GUARD_SIZE` is greater than zero but `FORGET_FILL_BYTE` is not set, you should use 0xde (the hexadecimal byte 0xde) as the fill byte value. Otherwise, you should use the value specified by `FORGET_FILL_BYTE` as the guard byte value.

To get the values of process environment variables to find out what `FORGET_GUARD_SIZE` and `FORGET_FILL_BYTE` are set to (or whether they're set), you can use the `getenv()` C library call. See the man page for details. You can assume that these two environment variables will not change during the execution of a user test program.

To set the value of each byte in a sequence of bytes to a particular value, the `memset()` system call can be used. Again, see the man page.

When a user memory block is freed, you should check the guard begin and guard end bytes to see whether they still have the value specified in the `FORGET_FILL_BYTE` environment variable (or the default 0xde if that variable isn't set). If they don't, you should print a warning message, indicating that the user program has modified memory that it doesn't own.

2.4 Bonus functionality: supporting all malloc-related calls

For 5% bonus credit you can extend your malloc library to support *all* the malloc-related library calls. In particular, you'll have to support `realloc` and `calloc`. You should read the relevant man page(s) to see exactly what these functions should do, what they take as parameters, and what they should return. The overall goal for adding this functionality is to interpose your malloc library underneath an arbitrary program, such as `gvim`, so that the program uses *your* malloc implementation rather than the C library's implementation.

Intercepting calls to malloc() and free() (and other malloc-related calls) can be accomplished by telling Linux to load your shared library *before* any other system shared library. Your library will get loaded into the address space of your test program, and when the malloc() and free() calls are "resolved" in your program, they will refer to the first occurence of those functions, which will be in your own shared library.

To specify that your library should be loaded before any program is executed, you can set the environment variable `LD_PRELOAD`, as follows:

```
$ export LD_PRELOAD=‘pwd‘/libforgetful.so
```

The `LD_PRELOAD` environment variable tells Linux to load the given shared library *before* any program is executed. Thus, when any program is run, it will be using your malloc() and free(), not the standard C library malloc/free. (This can be dangerous! If your library has bugs, don't be surprised if your shell becomes unusable.)

3 Tips and Hints

Some general tips for this lab:

- Implement the simplest free list allocation strategy: first fit. You aren't being graded on efficiency in this lab, so take the path of least resistance. You can easily modify this later to do next fit (rotating first fit) if you wish, but initially, keep it simple.
- Initially, don't worry about coalescing adjacent free blocks. Deal with that later, once you get some of the basics working.
- Be sure that you understand pointer arithmetic. Understanding how pointer addition works in C is essential to getting this lab done correctly. The Kernighan and Ritchie C book has excellent information. You may also wish to write some simple test programs to verify that you understand how pointer math works.
- Draw out specific corner cases and various scenarios for allocation and freeing in order to understand how your free list information will change. For example:
  - Free list is NULL; allocate one block; free that block. Free list should be NULL again.
  - Free list has only one block on it; allocate a new block with request exactly same size as free list; free list should now be NULL.
  - Free list is NULL; allocate two blocks; free the first; free list should point to first block; free the second; free list should now contain both blocks. Test with freeing in reverse order (free second, then first); free list should point to the blocks in correct order (first block then second).

- Use `assert` function calls copiously in your library code to check for unexpected conditions and to ensure that certain invariants hold (e.g., a pointer is non-NULL or similar). These error checks are essential for debugging libraries and help to flag error conditions early on in the development process. Once you `#include <assert.h>`, you can call the assert function and pass a boolean expression, e.g., `assert (x == 0);`. If `x` is not zero, your program will crash — this is the "feature" of the assert function! So, by sprinkling `assert` statements here and there, you can verify whether things that should be true, indeed are true, and cause your program to crash if they don't.

- Declare all internal functions (those that are not called by clients of the library) "static" to prevent naming conflicts with programs that link with your malloc library. For example:

```
1  static void malloc_helper(void)
2  {
3      // a helper function that user programs shouldn't (and can't) touch
4  }
```

  In C, this use of the keyword static means that only functions within the current file (e.g., forgetful.c) can access the static functions. Similar things can be done with private library variables.

An example way to implement a free list

The following sketches one way that you could maintain your free list. However, you're welcome to use any free list implementation you like.

For each memory block, you could maintain a *header* that contains two items: the size of the block and an offset (pointer) to the next free block. Each of these items should be 4 bytes in length, giving a total header size of 8 bytes. The block size should be stored in number of bytes as an integer. The offset could also be stored as an int and should contain the distance (offset) between the first byte of the current block and the first byte of the next free block. Alternatively, you could keep *all* memory blocks, both allocated and free, on a list and add a flag to the header to indicate whether a block is free or not.

Note that with this basic design you will never allocate a block for less than 8 bytes due to the overhead of the block header. When you receive a malloc request for 10 bytes, you will actually need to allocate 18 bytes to satisfy the request. The first 8 bytes will be used for library bookkeeping and the last 10 bytes will be for the user. *Your malloc function should return a pointer to the first byte of the user bytes* — not to the header. To accomplish this (and other necessary bookkeeping tasks within your library), you will have to become familiar with "pointer arithmetic".

In C, if p is a pointer to a block of heap memory, the expression `p + i` actually means `p + i * sizeof(p)`. Thus, if p is a pointer to an int, p+1 will yield the address of p + 4 bytes (assuming an int consumes 4 bytes). If p is a pointer to char, p + 1 will yield the address of p + 1 byte (the char type consumes one byte). Note that you cannot use pointer arithmetic with pointers to void (i.e., void *) since sizeof(void) is nonsensical.

So, to allocate a block of size 10 and return that to the user, all while maintaining our library bookkeeping, we might do something like the following. (Assume here that our free list is NULL and that we need to increase the size of our heap segment).

```
1  int requestSize = 10;
2  int internalRequestSize = requestSize + sizeof(int)*2;
3        // space for two header items plus the actual request
4
5  void *vptr = sbrk(internalRequestSize);
6        // vptr points to the first byte of a block of 18 bytes
7        // (10 + 2 * sizeof(int) = 18)
8
9  int *iptr = (int*)vptr;
10       // cast vptr something more useful
11
12 *(iptr + 0) = requestSize; // store the (user) block size in the first
13                           // 4 bytes of the header as an int
14 *(iptr + 1) = 0;           // set the offset to the next free block as
15                           // zero, initially.
16 void *rv = (void *)(iptr + 2);
17                           // rv points to the first byte of the block
18                           // of 10 bytes that the requester wants.
```

```
19                          // return type of malloc() is void*
20  return rv;
```

Your free list should always be ordered by address. The offset part of the header should point to the next block in the free list, or zero if the current block is the end of the free list. For the offset, the value represents the number of bytes between the first bytes of consecutive free blocks. Using pointer arithmetic, it is relatively simple to move from one block to the next along the list:

```
1   //
2   // assume that free list has been declared as something like:
3   //    static void *freeList;
4   // inside the library.
5   //
6
7   int *iptmp = (int *)freeList;
8   int offset = *(iptmp + 1);
9
10  char *firstFreeBlock = freeList;
11  char *secondFreeBlock = NULL;
12  if (offset > 0)
13  {
14      // non-zero offset gives distance to next free block
15      secondFreeBlock = firstFreeBlock + offset;
16
17      int *tmp2 = (int *)secondFreeBlock;
18      int blockSize = *(tmp2 + 0);
19      int nextOffset = *(tmp2 + 1);
20  }
```

4 Submission

Just submit your forgetful.c file to Moodle. This should be the only file to which you make any changes related to implementing the `malloc`, `free`, and `dumpMemoryMap` functions required for this lab.