OS(H) — Memory Management Lab

jeremy.singer@glasgow.ac.uk

20 Feb 2017

1 What this lab involves

Do you remember using `malloc` and `free` in your Advanced Programming lectures, when you were learning how to code in C? Did you ever wonder how the runtime system lays out and organizes its memory underneath this abstraction? The answer involves **free space management**, which we will explore in this lab exercise.

2 Intended learning outcomes

By the end of this lab, students should be able to:

1. **recall** the semantics of explicit dynamic memory management for C programs
2. **implement** a simple freelist based memory allocator for uniformly sized memory blocks
3. **appreciate** the underlying need for `free` operations
4. **appraise** various data structures and metadata formats for free space management
5. **sketch** an outline implementation of a memory allocator for varied size blocks

3 String sorting program: What you need to do

Download and unzip mem_lab.zip. Examine the file sort.c — this program reads in a textfile, finds each sequence of 15 character strings and then sorts these in an array. It skips over strings that contain the substring `''Once''`.

Look at the `#define` macros at the start of sort.c — notice the three macros `MALLOC`, `FREE` and `INIT_MEM`. Currently these are set to use the default C memory allocation routines. So you can run the program (compile it with gcc or similar) and observe the sorted output as follows:

```
[jsinger@narnia lab_mem]$ gcc sort15.c
[jsinger@narnia lab_mem]$ ./a.out
A mouse took a
Alice was begin
Dorothy lived i
Here is Edward
```

```
If you went too
In a hole in th
It all began wi
Mr Sherlock Hol
Mr and Mrs Durs
Mr. and Mrs. Br
Roger, aged sev
Squire Trelawne
The Mole had be
The boy with fa
When Mr Bilbo B
[jsinger@narnia lab_mem]$
```

Now, you need to define your own memory management routines in the header file `my_memory_allocator.h`.

Follow the sequence of steps below, to create a fixed size freelist based memory allocator. There are comments in `my_memory_allocator.h` to show you where to make modifications. You can search online for more details about freelist allocation, e.g. at `https://en.wikipedia.org/wiki/Free_list`.

3.1 Define your type carefully

You are effectively setting up a linked list. Each cell of the linked list has a next pointer, but can also be a string of 16 characters (including the null char at the end of the string). Ideally, you should use a `union` datatype, which has two alternatives, a `next` pointer and a fixed length char array.

Perhaps use a `typedef` to give your union a tidy name, like "cell".

3.2 Initialize a large block of memory

Do this in the `init_mem_pool` routine. I recommend you use the `sbrk` library routine to allocate a block of memory. Check its semantics online or via `man sbrk`.

What will you store in this large block of memory? Presumably a sequence of cells (the union data type you defined above). You could set up the linked list now. Iterate over all the cells and store the pointer to the next cell in the current cell. You will need a special SENTINEL value (I suggest `(void *)-1`) for the end of the list.

This is the freelist: it should should contain all the available memory from the pool, in discrete sized cells.

3.3 Write your malloc routine

You'll need a global variable to point to the first free entry in your freelist. Your `my_malloc` routine should take off this cell and return its return pointer. Although malloc takes a size argument, you won't actually use it for fixed size freelist allocation. You could ignore it or make an assertion for now, since we are only able to allocate fixed size cells. When you allocate the cell to a client, you need to update your freelist head entry, i.e. the global variable that points to the first free cell.

3.4 Write your free routine

What happens when a client program wants to return a cell to the freelist? This freed cell can simply be prepended onto the existing freelist? A metadata update is required, to update appropriate pointers in the freelist.

3.5 Put it all together

Given the macro definitions, you should be able to compile and run the sort.c program now. Does it give you sensible output? Use gdb if you are getting segfaults, or similar. Ask the tutors to help you if you are stuck!

Make sure that you have set up the sizes correctly, for the individual cells. For this problem, each cell needs to be 16 bytes. For the overall freelist, you will need just under 20 nodes in the initial list. What is the minimum size of freelist you can run without the program running out of memory?

4 Binary Trees Program: What you need to do

Now we are going to have a competition. The binarytrees.c program is a *micro-benchmark*— i.e. a program that people use to test the performance of their memory allocators. Compile and run binarytrees.c with the default memory allocation routines. You should see output similar to this:

```
[jsinger@narnia lab_mem]$ gcc binarytrees.c
[jsinger@narnia lab_mem]$ ./a.out
Memory Allocator Test
 Live storage will peak at 10291408 bytes.

 Stretching memory with a binary tree of depth 18
 Creating a long-lived binary tree of depth 16
Creating 33824 trees of depth 4
  Top down construction took 82 msec
  Bottom up construction took 82 msec
Creating 8256 trees of depth 6
  Top down construction took 82 msec
  Bottom up construction took 83 msec
Creating 2052 trees of depth 8
  Top down construction took 84 msec
  Bottom up construction took 84 msec
Creating 512 trees of depth 10
  Top down construction took 83 msec
  Bottom up construction took 84 msec
Creating 128 trees of depth 12
  Top down construction took 85 msec
  Bottom up construction took 85 msec
Creating 32 trees of depth 14
  Top down construction took 86 msec
  Bottom up construction took 86 msec
Creating 8 trees of depth 16
  Top down construction took 92 msec
```

```
  Bottom up construction took 97 msec
Completed in 1262 msec
[jsinger@narnia lab_mem]$
```

Now adapt the progrma to use the routines you have defined in your header file `my_memory_allocator.h`. On an x86-64 Linux machine, each cell needs to be 24 bytes in size, and you will need around 1 million cells.

When you run the program, it should create lots of binary tree data structures of varying depths in memory. The final line of output reports how long the execution takes. How fast can you go, by optimizing your memory management library? We will have a class competition in the lab to see who can get the fastest execution of the binarytrees benchmark.

5 Things to consider

OK - so this is one way to implement a fixed size memory allocation system. Can you think of other techniques. Perhaps involving bitmaps, or low-order bits in pointers?

Further, how would you implement a more flexible `malloc` routine that allows you to allocate blocks with different sizes?

For some ideas, check out something like *buddy memory allocation*, e.g. see `https://en.wikipedia.org/wiki/Buddy_memory_allocation`. If you are very serious, check out Knuth's *The Art of Computer Programming*, vol. 1, chapter 2, section on *Dynamic Storage Allocation*.