PROJECT 2 - MEMORY ALLOCATOR
Computer Systems Principles

Emery Berger   Mark Corner

October 1, 2010

1 Overview

The purpose of this project is to acquaint you with how memory allocators provide virtual memory to programs.

Your assignment consists of two parts.

- Write a BiBOP-style memory allocator, which you will compile into a shared library that can be used with any existing program. You will need to implement the basic malloc API (malloc and free).
- Compare your allocator against the system allocator, comparing performance and space consumption for two benchmarks we will provide.

2 Allocator

2.1 Input

Your allocator will run as a layer between any program and the operating system. The program will attempt to allocate heap memory using calls such as `new`, `delete`, `malloc`, and `free`. These calls will be handled by your layer, which will in turn request memory from the operating system, using `mmap`.

The input to your program will be a sequence of calls to `malloc` and `free`, or `new` and `delete`. These calls will be intercepted by a "shim" layer that will then call your allocator. This shim will handle all memory allocation calls (`new`, `delete`, `malloc`, `calloc`, etc) and call your allocator. Note that it will only call `malloc` and `free`, as it will translate `new`, `delete`, etc. into calls to `malloc` and `free`.

2.2 Output

Your allocator should produce no output. If your allocator generates any output, the autograder will mark your solution as incorrect.

2.3 Implementing the Allocator

A BiBoP allocator manages memory in chunks of one page in size. The allocator must use segregated lists of different size objects (free and allocated). The objects themselves will not contain any headers—all metadata must appear at the start of each page, in a `BibopHeader`.

Your allocator will allocate memory in units of pages (4K) for small objects (segregated by different sizes, in powers of two from 8 to 1024), with all object metadata (like the size of all these objects) placed at the start of each page. Each page (4K chunk) should only contain objects of the same size. For objects larger than 1024 bytes, you should allocate a separate chunk of memory directly via `mmap()`.

To allocate memory from the operating system, you will use the following code (note: this will *not* work on a non-Unix system).

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <string.h>
#include <new>
#include "allocator.h"
using namespace std;

int fd = open("/dev/zero", O_RDWR);

void * ptr = mmap (NULL, sz, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
```

Refer to the man pages on the use of `mmap`. `mmap` requests will return page-aligned allocations, i.e., every mmap'ed area is guaranteed to start on some multiple of 4096.

To return memory to the operating system (when freeing a large object), call `munmap`. Please see the man page on how to use `munmap`.

The BiBop header needs to be located at the beginning of a page/area that you allocated with mmap. To do this you can use a "placement new":

```
bibop_ptr = new (page_ptr) BibopHeader;
```

which allocates a BibopHeader starting at the address given by page_ptr. You do not need to delete this header, as your munmap will take care of that for you.

Your allocator will be implemented as a C++ class conforming to the Allocator class in allocator.h. The header for the BiBop pages is also defined in allocator.h. The autograder will compile your project using our copy of allocator.h and you may not change any of the class or struct definitions.

2.4 Allocating Memory Inside the Allocator

Your allocator may not allocate any other memory from the operating system, except for giving to the user's program. This means that your allocator cannot call `new` or `malloc` directly, or indirectly. This also means that your allocator *may not use the STL*, which allocates memory from the heap.

Your allocator may use some global variables of reasonable size to implement the allocator. The storage for global variables is not in the heap.

You may not allocate extra memory for your allocator (other than that needed for the BiBop pages) using mmap. Part of getting this project right is making it space efficient and the autograder will count the number of times (and how large) your allocator calls mmap. Extra, or out-of-order calls will be marked as wrong. This specification is exact enough that any correct solution will call mmap and munmap the same number of times and in the same sequence.

Note that you can count calls to mmap and munmap yourself to try to determine if the allocator is space efficient.

2.5 Choosing Space to Allocate

In many cases there is more than one free object to choose to allocate. For instance, if two pages have free space you must choose one of the two pages at the next allocation. You must use the following algorithm for picking which page to allocate free space from:

- When allocating a new page, place it at the head of the list of pages of that size. The next allocation will occur from that page, even if other objects from pages in that list are freed before then.
- When a page becomes full, move it to the head of a separate list of pages that are full.
- When a page that was full has an object freed, move it to the head of the list of pages with space available. The next allocation will occur from that page. Leave it there until it becomes full again.
- You can choose any object inside the page to allocate.
- Using two lists is not necessary for large objects, as when they are freed you remove the entire object and header.

3 Compiling and Using Your Allocator

3.1 Compiling a shared library

Your will compile and run your code on Edlab Linux machines. To build your allocator as a shared library, you need to compile your code like this (the "-O2" and "-DNDEBUG" are optimization flags and should be used for timing experiments).

```
% g++ -g -O2 -DNDEBUG -Wall -shared allocator.cc -o libmymalloc.so libshim.a -ldl
```

3.2 Running Your Allocator with a Shared Library

Using the shared library you may run precompiled programs.

```
% LD_PRELOAD=./libmymalloc.so ./test
```

Your allocator will only work on relatively simple programs (for instance, single-threaded programs), so don't try this on Firefox.

3.3 Compiling Your Allocator as Part of a Program

If you want to debug your allocator, it is handy to compile it as part of a program, rather than as a shared library.

For instance:

```
% g++ -Wall -g -o test_alloc allocator.cc test.cc libshim.a -ldl
```

The "-g" tells the compiler to include debugging symbols. You may then run your program with `gdb`, `valgrind`, or other debuggers and profilers. Please see the TA for information on how to use these tools.

3.4 Testing your allocator

You should test your program against a number of real programs (things like `df` and `pwd` are at least sanity checkers). The example below shows how to run a benchmark and get timing and memory consumption results.

```
% export LD_PRELOAD=/your/directory/goes/here/libmymalloc.so
% /usr/bin/time benchmark
[...] (output omitted)
0.00user 0.00system 0:00.00elapsed 50%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+274minor)pagefaults 0swaps
```

Note that this will run `time` with your allocator as well, so you shouldn't do this until your allocator is really working.

The first set of results indicates how much time your program took (CPU time by your program, kernel time on behalf of your program, and total wall clock time). The second set indicates the number of page faults—the larger the number, the more pages of memory your program visited.

To run a benchmark using the default library, do this:

```
% export LD_PRELOAD=
% /usr/bin/time benchmark
```

4 Performance hints

- Finding a BibopHeader for a previous allocation should take constant time. (Hint: recall that `mmap` returns page aligned addresses).

- Finding a free page to allocate for any particular size should occur in constant time.
- If your allocator is no longer using a page, it should be returned to the operating system.
- Only open `/dev/zero` once.

My advice is to get the allocator to work, then optimize it.

5 Handing Project In

All of the files you need (`allocator.h`, `libshim.a`) can be found on the edlab in:

`/courses/cs200/cs291sp/cs291sp/malloc/`

Your project will be handed in using the autograding system. The autograder is checking for correctness and performance. Currently, the autograder will deem any allocator that is only three times slower than the solution allocator as “fast enough”. Please see the web page for details on how to submit your solution.