Introduction to Systems Programming & Computer Organization (CPSC 323) [Spring 2020]

Project 2: A Debugging Memory Allocator

Due: February 12, 2020

Objectives

Having full control over explicit allocation and deallocation of dynamic memory allows us to write fast code for machines. However, this also leads to writing programs that crash due to memory issues. At the same time, we can also build tools that can debug memory allocation problems.

In this problem set, you will write a debugging memory allocator in C or C++, that will provide many of the features of Valgrind. Specifically, your debugging allocator will:

1. Track memory usage,
2. Catch common programming errors (e.g., use after free, double free),
3. Detect writing off the end of dynamically allocated memory (e.g., writing 65 bytes into a 64-byte piece of memory), and
4. Catch less common, somewhat devious, programming errors.

You will also augment your debugging allocator with heavy hitter reporting that tells a programmer where most of the dynamically allocated memory is allocated.

Environment Details

Complete this assignment by yourself on the Zoo computers (this assignment is subject to the collaboration policy in the course syllabus). These machines are physically located in AKW on the 3rd Floor. You are responsible for ensuring that your programs compile and execute properly on these machines. You can connect to a Zoo machine using SSH. For example:

```
ssh <netID>@node.zoo.cs.yale.edu
```

We provide starter code and tests in `/c/cs323/Hwk2/StarterCode` on the Zoo; you should copy this directory somewhere into your home directory. For this assignment, you run the tests using the makefile in the starter code. To just compile your library and tests, run `make`. Use the following command to compile and run the tests the all the tests until one fails:

```
make check
```

To run the all the tests without stopping:

```
make check-all
```

To run a particular test:

```
make test001
./test001
```

Running the tests in this way will just show you your program's output. The expected output is at the bottom of each testNNN.cc file.

Background

Memory allocation in C

C-style memory allocation uses two basic functions, `malloc()` and `free()`

```
void* malloc(size_t size)
```

Allocate `size` bytes of memory and returns a pointer to it. This memory is not initialized to any value and can contain anything. Returns a null pointer if the allocation failed (for example, if the size was too big).

```
void free(void* ptr)
```

Free a single block of memory previously allocated by `malloc()`.

The rules of `malloc()` and `free()` are simple: Allocate once, then free once.

- Dynamically-allocated memory remains active until it explicitly freed with a call to `free()`.
- A successful call to `malloc(sz)` returns a pointer ptr to "new" dynamically allocated memory. This means that the sz bytes of storage starting at address ptr are guaranteed not to overlap with any other active objects, including the program's code and global variables, its stack, or with any other active dynamically allocated memory.
- The pointer argument in `free(ptr)` must either equal nullptr or a pointer to active dynamically-allocated memory. In particular:
  - It is not OK to call `free(ptr)` if ptr points to the program's code, or into its global variables, or into the stack.
  - It is not OK to call `free(ptr)` unless ptr was returned by a previous call to `malloc()`.
  - It is not OK to call `free(ptr)` if ptr is currently inactive (i.e., `free(ptr)` was previously called with the same pointer argument, and the ptr memory block was not reused by another `malloc()`).

  These errors are called **invalid frees**. The third error is also called a **double free**.

Some notes on boundary cases:

- In C++, `malloc(0)` should return a non-null pointer. If `ptr = malloc(0)`, then ptr does not overlap with any other allocation and can be passed to `free()`.
- `free(nullptr)` is allowed. It does nothing.

- `malloc(sz)` returns memory whose alignment works for any object (as described in Lecture 1, slide 62). On 64-bit x86 machines, this means that the address value returned by `malloc()` must be evenly divisible by 16.

A secondary memory allocation function, `calloc`, allocates memory for an array of objects and clears it to zero.

```
void* calloc(size_t nmemb, size_t sz) {
    void* ptr = malloc(sz * nmemb);
    if (ptr != nullptr) {
        memset(ptr, 0, sz * nmemb);     // clear memory to 0
    }
    return ptr;
}
```

There is a bug in this implementation of `calloc()`, our tests will catch it!

You will work on our replacements for these functions:

- `dmalloc_malloc()`
- `dmalloc_free()`
- `dmalloc_calloc()`

Our versions of these functions simply call basic versions, `base_malloc()` and `base_free()`. The dmalloc functions take extra filename and line number arguments; you'll use these to track where memory was allocated and to report where errors occur. Our header file, `dmalloc.hh`, uses macros so that calls in the test programs supply these arguments automatically.

A note on undefined behaviour

Debugging allocators interact with undefined behavior. As we tell you in class, undefined behavior is a major no-no, because any program that invokes undefined behavior has no meaning. As far as the C language standard is concerned, once undefined behavior occurs, a program may do absolutely anything, such as force demons to fly out of your nose. Many of our tests explicitly invoke undefined behavior, and thus have no meaning. But helpful debuggers catch common bugs, and undefined-behavior bugs with malloc and free are common, so your helpful debugging allocator must produce specific warnings for these cases! Is that even possible?

Yes! Debugging allocators work by making certain undefined behaviors well-defined. For instance, when a debugging allocator is in force, a program with a simple double free will reliably print a specific error message and exit before any undefined behavior occurs.

To accomplish this magic trick, debugging allocators make use of properties of their underlying allocators. You should use our "base" allocator, `base_malloc()` and `base_free()`, which is defined in `basealloc.cc`. This allocator behaves like malloc and free, but has the following additional properties:

- `base_free()` does not modify freed memory (the contents of freed storage remain unchanged until the storage is reused by a future `base_malloc()`).
- `base_free()` never returns freed memory to the operating system.

This makes it much easier to write a debugging allocator with `base_malloc/free` than with C's default `malloc/free`. For instance, the following program is well-defined:

```c
int main(int argc, char* argv[]) {
    int* x = base_malloc(sizeof(int));
    *x = 10;
    base_free(x);
    assert(*x == 10); // will always succeed: base_free doesn't overwrite freed
   memory
}
```

In contrast, this program, which uses the system `malloc()` and `free()`, always exhibits undefined behavior (and the assertion will likely fail).

```c
int main(int argc, char* argv[]) {
    int* x = malloc(sizeof(int));
    *x = 10;
    free(x);
    assert(*x == 10); // use-after-free undefined behavior
}
```

This is because undefined behavior is imposed by language-defined interfaces, such as the system `malloc()` and `free()`. While `base_malloc()` and `base_free()` are used in the same way as `malloc()` and `free()`, they are not language-defined, so they can offer slightly more relaxed rules.

Using the base_allocator functions, your debugging allocator should run tests 1 through 26 completely safely, with no sanitizer warnings. (Use `make SAN=1 check-1-26` to check.) Even though some of the test programs, such as test020, appear to invoke undefined behavior, your debugging allocator library will catch the problems before the undefined behavior actually occurs.

Note that even for `base_malloc()` and `base_free()`, double frees, invalid frees, and wild writes can cause truly undefined behavior.

Part 1: Debugging allocator (80%)

This section describes the main functionality required for the debugging allocator, ordered from simpler to more complex, and from low-numbered to high-numbered tests. **Note: There are no private tests for this section.**

Heap Usage Statistics

Implement the following function:

```c
void dmalloc_get_statistics(dmalloc_statistics* stats);
```

Fill in the `stats` structure with overall statistics about memory allocations so far.

The `dmalloc_statistics` structure is defined like this:

```c
struct dmalloc_statistics {
    unsigned long long nactive;     // number of active allocations [#malloc - #
   free]
```

```
    unsigned long long active_size; // number of bytes in active allocations
    unsigned long long ntotal;      // number of allocations, total
    unsigned long long total_size;  // number of bytes in allocations, total
    unsigned long long nfail;       // number of failed allocation attempts
    unsigned long long fail_size;   // number of bytes in failed allocation
   attempts
    uintptr_t heap_min;             // smallest address in any region ever
   allocated
    uintptr_t heap_max;             // largest address in any region ever allocated
};
```

Most of these statistics are easy to track, and you should tackle them first. You can pass tests 1-5 and 7–10 without per-allocation metadata.

The hard test is `active_size`, since to track it, your `dmalloc_free(ptr)` implementation must find the number of bytes allocated for `ptr`. The easiest, and probably best, way to do this is for your `dmalloc_malloc` code to request more space than the user when it calls `base_malloc`. The initial portion of that space will store metadata about the allocation, including the allocated size, using a structure you define yourself. Your `dmalloc_malloc` will initialize this metadata, and then return a pointer to the payload, which is the portion of the allocation following the metadata. Your `dmalloc_free` code will take the payload pointer as input, and then use address arithmetic to calculate the pointer to the corresponding metadata (possible because the metadata has fixed size). From that metadata it can read the size of the allocation. (But there are other techniques too. You could create a hash table that mapped pointer values to sizes. `dmalloc_malloc` would add an entry, and `dmalloc_free` would check this table and then remove the entry. You might try this first.)

Run `make check` to test your work. Test programs `test001.cc` through `test012.cc` test your overall statistics functionality. Open one of these programs and look at its code. You will notice some comments at the end of the file, such as this:

```
//!  alloc count:  active 0 total 0 fail 0
//!  alloc size :  active 0 total 0 fail 0
```

These lines define the expected output for the test. The `make check` command checks your actual output against the expected output and reports any discrepancies. (It does this by calling `check.pl`.) Note: In expected output, "???" can match any number of characters.

Integer overflow protection

Your debugging malloc library should support the `calloc` secondary allocation function. It also needs to be robust against integer overflow attacks. Our handout code's `dmalloc_calloc` function is not quite right. Fix this function and fix any other integer overflow errors you find. Test programs `test013.cc` through `test015.cc` check your work.

Invalid free and double-free detection

`dmalloc_free(ptr, file, line)` should print an error message and then call C's `abort()` function when `ptr` does not point to active dynamically-allocated memory. Some things to watch:

- Be careful of calls like `free((void*) 0x16)`, where the `ptr` argument is not `nullptr` but it also doesn't point to heap memory. Your debugging allocator should not crash when passed such a pointer. It should print an error message and exit in an orderly way. Test program `test017.cc` checks this.
- The test programs define the desired error message format. Here's our error message for test016:

```
MEMORY BUG: test016.cc:8:  invalid free of pointer
0xffffffffffffffe0, not in heap
```

- Error messages should be printed to standard error (using C's `fprintf(stderr, ...)`, or, equivalently, C++'s `std::cerr`).
- Different error situations require different error messages; the other test programs define the required messages.
- Include the file name and line number of the problematic call to `free()`.

Test programs `test016.cc` through `test024.cc` check your work.

Boundary write error detection

A *boundary error* is when a program reads or writes memory *beyond* the actual dimensions of an allocated memory block. An example boundary write error is to write the 11th entry in an array of size 10:

```
int* array = (int*) malloc(10 * sizeof(int));
...
for (int i = 0; i <= 10 /* WHOOPS */; ++i) {
    array[i] = calculate(i);
}
```

These kinds of errors are relatively common in practice. (Other errors can happen, such as writing to totally random locations in memory or writing to memory before the beginning of an allocated block, rather than after its end; but after-the-end boundary writes seem most common.)

A debugging memory allocator can't detect boundary read errors, but it can detect many boundary write errors. Your `dmalloc_free(ptr, file, line)` should print an error message and call `abort()` if it detects that the memory block associated with `ptr` suffered a boundary write error.

No debugging allocator can reliably detect all boundary write errors. For example, consider this:

```
int* array = (int*) malloc(10 * sizeof(int));
int secret = array[10];      // save boundary value
array[10] = 1384139431;      // boundary write error
array[10] = secret;          // restore old value! dmalloc can't tell there was an
   error!
int* array = (int*) malloc(10 * sizeof(int));
array[200000] = 0;           // a boundary write error, but very far from the
   boundary!
```

We're just expecting your code to catch common simple cases. You should definitely catch the case where the user writes one or more zero bytes directly after the allocated block. Test programs `test025.cc` through `test027.cc` check your work.

Memory leak reporting

A memory leak happens when code allocates a block of memory but forgets to free it. Memory leaks are not as serious as other memory errors, particularly in short-running programs. They don't cause a crash. (The operating system always reclaims all of a program's memory when the program exits.) But in long-running programs, such as your browser, memory leaks have serious effect and are important to avoid.

Write an `dmalloc_print_leak_report()` function that, when called, prints a report about every allocated object in the system. This report should list every object that has been `malloc`ed but not `free`d. Print the report to standard output (not standard error). A report should look like this:

```
LEAK CHECK: test033.cc:23:  allocated object 0x9b811e0 with size 19
LEAK CHECK: test033.cc:21:  allocated object 0x9b81170 with size 17
LEAK CHECK: test033.cc:20:  allocated object 0x9b81140 with size 16
LEAK CHECK: test033.cc:19:  allocated object 0x9b81110 with size 15
LEAK CHECK: test033.cc:18:  allocated object 0x9b810e0 with size 14
LEAK CHECK: test033.cc:16:  allocated object 0x9b81080 with size 12
LEAK CHECK: test033.cc:15:  allocated object 0x9b81050 with size 11
```

A programmer would use this leak checker by calling `dmalloc_print_leak_report()` before exiting the program, after cleaning up all the memory they could using `free()` calls. Any missing `free`s would show up in the leak report.

To implement a leak checker, you'll need to keep track of every active allocated block of memory. It's easiest to do this by adding more information to the block metadata. You will use the `file` and `line` arguments to `dmalloc_malloc()`.

Note: You may assume that the `file` argument to these functions has static storage duration. This means you don't need to copy the string's contents into your block metadata—it is safe to use the string pointer.

Test programs `test028.cc` through `test030.cc` check your work.

Advanced reports and checking Test programs `test031.cc`, `test032.cc`, and `test033.cc` require you to update your reporting and error detection code to print better information and defend against more diabolically invalid frees. You will need to read the test code and understand what is being tested to defend against it.

Update your `invalid free` message. After determining that a pointer is invalid, your code should check whether the pointer is inside a different allocated block. This will use the same structures you created for the leak checker. If the invalid pointer is inside another block, print out that block, like so:

```
MEMORY BUG: test031.cc:10:  invalid free of pointer 0x833306c, not allocated
  test031.cc:9:  0x833306c is 100 bytes inside a 2001 byte region allocated here
```

And make sure your invalid free detector can handle all situations in the other tests.

Performance and C++ integration

Finally, test programs `test034.cc` and up test other situations. `test034` calls `dmalloc_malloc` 500,000 times, supplying tens of thousands of different filename/line-number pairs. Your solution should run `test034` in a second or less. Our solution, with heavy hitters, runs `test034` in about 0.2 seconds on a modern Mac laptop. Some of the other tests repeat earlier tests, but using C++-style memory allocation instead of C-style memory allocation.

Part 2: Heavy hitter reports (20%)

Note: the tests for this section are all private. We provide a stress testing tool for you to use: hhtest. You will need to compile it: `make hhtest`. More information is given below.

Memory allocation is expensive, and you can speed up a program a lot by optimizing how that program uses malloc and by optimizing malloc itself (Google and Facebook use their own internal version of malloc; tcmalloc is google's version of malloc).

But before optimizing a program, we must collect data. Programmer intuition is frequently wrong: programmers tend to assume the slowest code is either the code they found most difficult to write or the last thing they worked on. This recommends a memory allocation profiler — a tool that tracks and reports potential memory allocation problems.

Your job is to design and implement a particular kind of profiling, heavy hitter reports, for your memory allocator. This has two parts. You will:

1. Track the heaviest users of malloc() by code location (file and line). A "heavy" location is a location that is responsible for allocating many payload bytes.
2. Generate a readable report that summarizes this information, using this exact format (each location on one line, sorted in descending order by percentage):

```
HEAVY HITTER: hhtest.cc:48:  817311692 bytes (~25.0%)
HEAVY HITTER: hhtest.cc:47:  403156951 bytes (~12.4%)
```

Rule 1: If a program makes lots of allocations, and a single line of code is responsible for 20% or more of the total payload bytes allocated by a program, then your heavy-hitter report should mention that line of code (possibly among others).

Rule 2: Your design should handle both large numbers of allocations and large numbers of allocation sites. In particular, you should be able to handle a program that calls `malloc()` at tens of thousands of different file-line pairs (such as `test034`).

Note: You should only count bytes requested by the program; don't include allocator overhead such as metadata space.

How should you implement this? We leave this up to you. You can implement this using something very simple if you like. Here are some tips -

- **Sampling is acceptable**. It would be okay, for example, to report information extrapolated from a sample of the allocations. This could cut down the amount of data you need to store.

  - Make sure that you still follow Rule 1 with very high probability.
  - You could sample exactly every Nth allocation, but random sampling is usually better, since it avoids synchronization effects. (For instance, if the program cycled among 4 different allocation sites, then sampling every 20th allocation would miss 75% of the allocation sites!) For random sampling you'll need a source of randomness. Use `random()` or `drand48()`.
- Here are some papers describing algorithms that catch all heavy hitters with O(1) space and simple data structures. YOU DO NOT NEED TO USE THESE ALGORITHMS! Try to take a look, they are surprisingly simple.
  - A Simple Algorithm for Finding Frequent Elements in Streams and Bags, Karp, Shenker, and Papadimitriou
  - Frequency Estimation of Internet Packet Streams with Limited Space, Demaine, López-Ortiz, and Munro. The paper's context doesn't matter; the relevant algorithms, "Algorithm MAJORITY" and "Algorithm FREQUENT," appear on pages 6-7, where they are simply and concisely presented. (You want FREQUENT, but MAJORITY is helpful for understanding.)

We provide a test program for you to test heavy hitter reports, `hhtest.cc`. This program contains 40 different allocators that allocate regions of different sizes. Its first argument, the skew, varies the relative probabilities that each allocator is run. Before exiting, it calls `dmalloc_print_heavy_hitters()`, a function defined in `dmalloc.cc`.

Running `./hhtest 0` will call every allocator with equal probability. But allocator #39 allocates twice as much data as any other. So when we run our simple heavy hitter detector against `./hhtest 0`, it reports:

```
HEAVY HITTER: hhtest.cc:49:  1643786191 bytes (~50.1%)
HEAVY HITTER: hhtest.cc:48:  817311692 bytes (~25.0%)
HEAVY HITTER: hhtest.cc:47:  403156951 bytes (~12.4%)
```

If we run `./hhtest 1`, however, then the first allocator (`hhtest.cc:10`) is called twice as often as the next allocator, which is called twice as often as the next allocator, and so forth. There is almost no chance that allocator #39 is called at all. The report for `./hhtest 1` is:

```
HEAVY HITTER: hhtest.cc:10:  499043 bytes (~50.0%)
HEAVY HITTER: hhtest.cc:11:  249136 bytes (~25.0%)
HEAVY HITTER: hhtest.cc:12:  123995 bytes (~12.5%)
```

At some intermediate skews, though, and there may be no heavy hitters at all. Our code reports nothing when run against `./hhtest 0.4`.

Negative skews call the large allocators more frequently. `./hhtest -0.4`:

```
HEAVY HITTER: hhtest.cc:49:  15862542908 bytes (~62.1%)
HEAVY HITTER: hhtest.cc:48:  6004585020 bytes (~23.5%)
```

Try `./hhtest -help` to get a full description of `hhtest`'s arguments. You should test with many different arguments; for instance, make sure you try different allocation "phases." A great software engineer would also create tests of her own; we encourage you to do this!

Style and Code Organization:
Your code should be clean, clear, correct, and consistent. The most important style guideline is consistency. Don't write code that changes style from line to line. In addition, as a general rule, this assignment will be easier to write if you break up your program into smaller (1-50 lines), reusable functions with readable and unambiguous names.

Grading

We will provide a set of 38 public tests for part 1, and we will use a small set of additional private tests to grade your code Heavy Hitter code. You will lose additional points for other compilation issues (including warnings).

Submission

To submit your code you will need to run the submission tool from the directory that contains your dmalloc.cc file. The tool will submit ONLY your dmalloc.cc file. YOU MAY NOT USE EXTRA FILES FOR THIS ASSIGNMENT.

```
/c/cs323/Hwk2/submit
```

You can submit any number of times; we will only look at the most recent submission.