Malloc Lab

Author Clayton Knittel

Overview

You'll be implementing dynamic memory allocation via the C `malloc` API for 64-bit virtual address spaces. You will be implementing the following four functions:

```
void* malloc(size_t size);
void* calloc(size_t nmemb, size_t size);
void* realloc(void* ptr, size_t size);
void free(void* ptr);
```

`malloc` is used to allocate blocks of contiguous memory of a requested size in bytes, returning a pointer to the starting address of that block. The block must be deallocated by calling `free` on that pointer before the program terminates, otherwise the program is considered to have a memory leak. This is a contract requested by `malloc`, but it is up to the callers of `malloc` to uphold this contract. Tools like valgrind are used to catch bugs in code like memory leaks, or writes to unallocated memory.

As the author of `malloc`, you are allowed to assume the invariants requested by the API, which are the following:

1. For every pointer `p` returned from `malloc/calloc/realloc`, there is a call to `free(p)` or `realloc(p, _)` some time in the future, with no calls to `free(p)` or `realloc(p, _)` between those two calls.
2. For every call to `free(p)` or `realloc(p, _)` on pointer `p`, there is a `malloc/calloc/realloc` which previosuly returned `p`.

In other words, you may assume all calls to `free/realloc` are passed pointers to allocated blocks.

There are some invariants that you as the `malloc` author must uphold:

1. All pointers returned from allocation functions lie within memory allocated by the kernel and can be written to in full by the caller.
2. All pointers returned from allocation functions are aligned to 16 bytes, unless the requested size is 8 bytes or smaller, in which case they must be aligned to 8 bytes.
3. Blocks of allocated memory must not overlap.
4. Blocks of allocated memory must not be written to by `malloc` code (i.e. you are not allowed to dirty allocated blocks).

Each of the `malloc` functions are described in detail below:

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

`malloc` returns a pointer to an allocated block of *uninitialized* memory of the requested size. If `size` is 0, then `nullptr` is returned.

```
void* calloc(size_t nmemb, size_t size);
```

`calloc` returns a pointer to an allocated block of size $nmemb * size$ fully initialized to 0. Similarly to `malloc`, if `size` is 0, then `nullptr` is returned.

```
void* realloc(void* ptr, size_t size);
```

`realloc` changes the size of an allocated block to `size`, returning a pointer to a block of memory of the new size, with the contents of the old block passed to `realloc` copied up to the minimum of the length of the old and new block. *Note*: `realloc` may return the same block which is passed to it.

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

`free` deallocates the block of memory passed to it, allowing it to be allocated by a future allocation call. This block must have been returned by a previous call to one of the allocation functions. If `nullptr` is passed, `free` should do nothing.

Git instructions

You should fork the repository at https://github.com/ClaytonKnittel/malloc-bench. The functions you must fill in are at `/src/allocator_interface.h`. Please feel free to make more files/directories to nicely organize your code.

Requesting memory from the kernel

When your program loads, no memory has yet been allocated by the kernel for the heap (where dynamic memory lives). Normally, you would request the kernel to allocate a range of virtual addresses with `mmap`, but for the purposes of testing, I have provided a heap interface that you should allocate through.

This heap management interface contains only one memory "arena" (i.e. contiguous region of memory) which "grows upward". The region of memory starts at some address *heap_start*, and continues up through *heap_start*+ *heap_size*. When `sbrk(n_bytes)` is called, *heap_start* + *heap_size* is returned, and *heap_size* is increased by *n_bytes*. Note that *heap_start* + *heap_size* is calculated before *heap_size* is modified.

To request memory from the heap manager, you may call:

```
// #include "src/singleton_heap.h"
void* prior_heap_end = SingletonHeap::GlobalInstance()->sbrk(n_bytes);
```

This function returns the previous end of the heap, before being extended by `n_bytes`. On the first call to this function, the start of the heap is returned.

Note that the benchmarking tool measures heap fragmentation by looking at how much memory the program requested with `sbrk`. This means you may want to wait to `sbrk` more memory for the heap until you need it, since requesting too much may lead to a poorer fragmentation score.

Programming Restrictions

- This may be obvious, but you are not allowed to invoke system dynamic `malloc`/`free`, nor `mmap` or `sbrk` not directly through the `SingletonHeap` interface.
- You may not define large global data (i.e. compound global data structures to hold heap metadata). You will need to use global data (i.e. a pointer to somewhere on the heap where your metadata starts), but all other bookkeeping must be done on the heap.

Running the correctness checker and benchmarking tools

You can test your code against all traces by running

```
bazel run -c opt //src:driver
```

This will check for correctness on each trace first, and if your code is correct, it will measure throughput and memory utilization. When running your code for scoring, make sure you use `-c opt` so throughput is measured accurately.

While writing your code, you will probably want to run on specific small traces first. You can run the driver on a single trace in debugging mode with:

```
bazel run -c dbg //src:driver -- --trace=traces/<trace>
```

(replace `<trace>` with the name of the trace you want to run). The short traces typically contain "test", "simple" or "_short".

How throughput and utilization are measured

Throughput is measured in mega ops per second, or million `malloc`/`free`/etc. calls per second. There is some overhead in the calling code, but it is very small.

Utilization measures the overhead of metadata and fragmentation. It simply checks how much total memory you requested with `sbrk` compared to the peak amount of memory allocated to the user at once over the course of a trace, and reports this difference as a percentage.

Heap Checker

It is highly recommended that you write a heap check function which checks for consistency of the whole heap. Don't worry about how slow this method is, it will likely need to iterate over the entire heap. It will be invaluable for testing, as dynamic memory allocators are notoriously difficult to debug when something breaks.

I recommend either using macro guards, a global variable, or something similar to enable heap checking for debugging purposes. You want it to be configurable so you can disable it after you finish and want to measure throughput (also, some of the longer traces may take a long time to complete with heap checking enabled).

Tips and Tricks

You are only given access to one memory region (heap/arena), so your heap metadata and allocated blocks will likely interleave. It is often possible to use the extra space in unallocated blocks for metadata, which doesn't waste any space (since this extra space is caused by fragmentation and is unavoidable).

Additionally, you may assume that all pointers passed to `free` are valid, so you may be able to avoid tracking allocated blocks in the same way you will freed blocks (i.e. you will never need to find an allocated block at any point, you can wait until it is freed before inserting it back into your data structures).

Small allocation sizes are much more common than large, and certain sizes (i.e. 8, 16, 32, 64) are typically most common (note that cache lines are usually 64 bytes). You may want to treat small allocation sizes differently from large.