Performance Engineering of Software Systems
Massachusetts Institute of Technology 6.172
Prof. Charles E. Leiserson and Prof. Julian Shun Handout 10

Homework 6: Custom Memory Allocators

[Note: This assignment makes use of AWS and/or Git features which may not be available to OCW users.]

1 Introduction

Project 3 requires you to examine the complex real-world problem of high-performance memory management. You will implement a serial memory allocator that implements the `malloc()`, `free()`, and `realloc()` functions (the C memory management API). In this homework, you will implement different versions of such an allocator.

Then, you will explore extensions to the memory management API for cases in which a custom memory allocator is useful. In particular, you will complete implementations for a "wrapped allocator," a "packed allocator," a "fixed aligned allocator," and a "smart allocator."

You should start installing OpenTuner immediately; it will take about 10 minutes to install, and starting it before you look at the code will let you have it ready once you need it. Please ensure that you are logged into your Amazon VM.

```
$ sudo apt-get install python-setuptools

$ sudo easy_install pip

$ sudo ./install_opentuner.sh
```

More detailed instructions are in the `README`. You can open up another `ssh` connection to your VM to work with while OpenTuner is being installed.

Performance

Unlike in previous assignments, you will be evaluating your custom allocators in terms of space utilization and throughput.

- ***Space utilization*** is the peak ratio between the aggregate amount of currently allocated memory (M) (i.e., allocated via your custom `malloc()` and not yet freed via your custom `free()`) and the size of the heap (H) used by your allocator. The optimal ratio is, of course, 1. The space utilization U is calculated as follows:

$$U = \max\{M, 40\text{KB}\} / \max\{H, 40\text{KB}\}$$

- ***Throughput*** is the average number of operations completed per second.

To summarize these two performance metrics for your allocator, we define a ***performance index*** P to be a weighted sum of the space utilization and throughput:

$$P = wU + (1 - w) \min\{1, T/T_{libc}\}$$

where U is your space utilization, T is your throughput, and T_{libc} is the estimated throughput of `libc`'s `malloc()` on your system on the default traces.

Running the code

Your allocator implementations will be tested on ***traces***, which are text files that encode a series of calls to `malloc()` and `free()`. You can use the provided `mdriver` program to test and evaluate your custom allocator on a given trace. Here is an example of how to compile and run the driver program on a particular trace:

```
$ make clean mdriver; awsrun ./mdriver -g -v -B -f traces/trace_c0_v0
```

You can also run the driver on all traces in the `traces` directory as follows:

```
$ awsrun ./mdriver -g -v -B
```

The `mdriver` program accepts the following command-line arguments:

- `-t <tracedir>`: Look for the default trace files in directory `<tracedir>` instead of the default directory (`./traces`).
- `-f <tracefile>`: Use one particular trace file for testing instead of the default set of trace files.
- `-h`: Print a summary of the command line arguments.
- `-l`: Run and measure `libc`'s `malloc()` in addition to the custom `malloc()` implementation.
- `-g`: Generate summary info for the autograder.

- `-v`: Verbose output. Print a performance breakdown for each trace file in a compact table.
- `-V`: More verbose output. Prints additional diagnostic information as each trace file is processed. Useful during debugging for determining which trace file is causing you to fail.
- `-B`: Use the custom "simple allocator."
- `-W`: Use the custom "wrapped allocator."
- `-P`: Use the custom "packed allocator."
- `-F`: Use the custom "fixed aligned allocator."
- `-S`: Use the custom "smart allocator."

2 Code layout

This homework uses code that resembles the code that you will use for Project 3. Let's review the functions whose implementations you will complete to implement your custom allocators, as well as some methods that you will use in those implementations.

Heap memory allocator interface

Your storage allocators will implement different versions of `init()`, `malloc()`, and `free()` using various allocation strategies. These functions are described below and (among other functions) are declared in `allocator_interface.h`. The specific versions of the functions to implement and modify are specified in each question.

- `int init(void);`

  Before calling the corresponding `malloc()` or `free()`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `init()`. You may use this function to perform any necessary initialization, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization and 0 if everything went smoothly. The specific versions you will encounter in this homework are

  ```
  int simple_init(void);
  int wrapped_init(void);
  int packed_init(void);
  int fixed_aligned_init(void);
  int smart_init(void);
  ```

- `void *malloc(size_t size);`

  This call must return a pointer to a contiguous block of newly allocated memory which is at least `size` bytes long. This entire block must lie within the heap region and must not overlap

any other currently allocated chunk. The pointers returned by `malloc()` must always be aligned to 8-byte boundaries; you'll notice that the `libc` implementation of `malloc()` does the same. If the requested size is zero or an error occurs and the requested block cannot be allocated, a `NULL` pointer must be returned. The specific versions you will encounter in this homework are

```
void *simple_malloc(size_t size);
void *wrapped_malloc(size_t size);
void *packed_malloc(size_t size);
void *fixed_aligned_malloc(size_t size);
void *smart_malloc(size_t size);
```

- `void free(void *ptr);`

  This call notifies your storage allocator that a currently allocated block of memory should be deallocated. The argument must be a pointer previously returned by `malloc()` and not previously freed. You are not required to detect or handle either of these error cases. However, you should handle freeing a `NULL` pointer – it is defined to have no effect. The specific versions you will encounter in this homework are

```
void simple_free(void *ptr);
void wrapped_free(void *ptr);
void packed_free(void *ptr);
void fixed_aligned_free(void *ptr);
void smart_free(void *ptr);
```

All of this behavior matches the semantics of the corresponding `libc` routines. Type `man malloc` at the shell to see additional documentation, if you're curious.

The provided memory allocator in `simple_allocator.c` is very fast. On `simple_malloc()`, it increases the heap size and returns the newly allocated memory, while on `simple_free()`, it does nothing. Compile and run it using `mdriver` on `rec_traces/trace_c0_v0`. Unsurprisingly, the reference allocator has nearly 0% space utilization (because it doesn't reuse freed memory) and 100% throughput.

Support routines

The code in `memlib.c` simulates the memory system for your dynamic memory allocators. You can invoke the following functions in `memlib.c`:

- `void* mem_sbrk(int incr);`

  Expands the heap by `incr` bytes, where `incr` is a positive non-zero integer and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix `sbrk()` function, except that `mem_sbrk()` accepts only a positive non-zero integer argument.

- `void* mem_heap_lo(void);`

  Returns a generic pointer to the first byte in the heap.

- `void* mem_heap_hi(void);`

  Returns a generic pointer to the last byte in the heap.

- `size_t mem_heapsize(void);`

  Returns the current size of the heap in bytes.

- `size_t mem_pagesize(void);`

  Returns the system page size in bytes (4 KB on Linux systems).

In addition to these functions, there are several macros defined in `allocator_interface.h` to help you implement your custom allocators.

3 Fixed-size blocks

To improve the space utilization, let's implement the fixed-size allocation strategy from Lecture 11: Storage Allocation. In this strategy, all blocks are allocated with the same size. The allocator uses a free list to track the freed blocks. The free list can be implemented as a singly linked list, with the `next` pointers stored inside the freed blocks.

For this first part, assume that the fixed block size that we use is 1024 bytes. Add the following lines to the top of `simple_allocator.c`:

```
#ifndef BLOCK_SIZE
#define BLOCK_SIZE 1024  // default value
#endif
```

You should use the variable `BLOCK_SIZE` in your code rather than hardcoding 1024 as you will be modifying `BLOCK_SIZE` in later parts. Notice that next to each block, the provided memory allocator stores the size of the block because it is needed for memory reallocation. Even though you are not using any reallocation functionality in this homework (that will be in Project 3!), you can choose whether or not you still want to store the size next to each block (which requires allocating slightly extra memory).

Checkoff Item 1: Implement the fixed-size block allocation strategy in `simple_allocator.c` by modifying `simple_init()`, `simple_malloc()`, and `simple_free()` as necessary. Report that the space utilization (and score) increases to over 99% when run on `rec_traces/trace_c0_v0`. *Hint:* Make a `struct` for the nodes of the free list, and remember to initialize the head in `simple_init()`. Use the `-B` flag when you run `mdriver`.

4 Autotuning

Unlike `rec_traces/trace_c0_v0`, trace `rec_traces/trace_c1_v0` only requests memory of a fixed size `BLOCK_SIZE = 4096`. Your allocator should support all allocation sizes less than or equal to `BLOCK_SIZE`. Add the following lines to the top of the `simple_malloc()` function in `simple_allocator.c`:

```
if (size > BLOCK_SIZE)
  return NULL;
else  // size <= BLOCK_SIZE
  size = BLOCK_SIZE;
```

The `simple_malloc()` function is now able to handle all sizes less than or equal to `BLOCK_SIZE`, and it returns an error (as a `NULL` pointer) for sizes greater than `BLOCK_SIZE`. (You may argue that this error checking shouldn't be necessary in `simple_malloc()`: someone else should check that it's broken if it returns a block of size `BLOCK_SIZE` when a size greater than `BLOCK_SIZE` was requested. We agree, and that will be your job in Project 3 when you fill in `validator.h`.) Confirm that there is an error when you recompile and run on `rec_traces/trace_c1_v0`.

```
$ make clean mdriver; ./mdriver -g -v -B -f rec_traces/trace_c1_v0
```

Next, let's manually override `BLOCK_SIZE` and confirm that there is no longer an error.

```
$ make clean mdriver PARAMS="-D BLOCK_SIZE=4096"

$ ./mdriver -g -v -B -f rec_traces/trace_c1_v0
```

`BLOCK_SIZE` is an example of a tunable parameter of the code, but how do we determine the best value for `BLOCK_SIZE`? In this case, we could easily determine the value by manually inspecting the traces, but in general, it can be very difficult to tune these parameters by hand, particularly when there are multiple different parameters (as you will see in Project 3). This is where autotuning is useful.

OpenTuner is a autotuning tool that, by running an optimization, automatically finds the best values for the parameters that you tell it about. You will use OpenTuner next to see if you can automatically determine the appropriate values for `BLOCK_SIZE` on different traces.

Checkoff Item 2: Use OpenTuner to find the best value of `BLOCK_SIZE` for `rec_traces/trace_c0_v0` and `rec_traces/trace_c1_v0`.

1. Add `BLOCK_SIZE` as a power-of-two parameter in `opentuner_params.py`, varying it from 2^5 to 2^{15}.

2. Run the OpenTuner script (which takes 1–2 minutes):

   ```
   $ ./opentuner_run.py --test-limit=300 --no-dups --display-frequency=20 \
   --trace-file=<trace-file>
   ```

   How did OpenTuner know that `BLOCK_SIZE` should be `1024` and `4096`, respectively? Is it just really good at reading traces, or is the value of `BLOCK_SIZE` somehow affecting the value of the optimization's objective function?

3. Add a target `autotune` to your Makefile so that you can run OpenTuner by running

   ```
   $ make clean autotune TRACE_FILE=<trace_dir>/<tracefile>
   ```

   Hint: Your Makefile target needs to run the `opentuner_run.py` script with the necessary flags shown above, as well as the trace file passed in on the command line.

Checkoff

Commit your changes to your local repository, then verify your work using `verifier.py` and check your code quality by running `clint.py`. If these scripts pass, show your work to a TA or UTA to complete the checkoff for the recitation.

5 Cache-friendly allocation

Aligning objects on a cache-line boundary limits the number of cache lines needed to access an object. For randomly accessed objects, ensuring that an access uses the fewest cache lines possible is especially important.

In implementing the allocators in this section, assume that objects never need to be freed with `wrapped_free()`. You can evaluate your allocators empirically on `traces/trace_c0_nofree`. Any existing allocator can be used to ensure that objects start at a cache-line boundary with sufficient padding.

Write-up 1: In `wrapped_allocator.c`, wrap the call to `unaligned_malloc()` inside of `wrapped_malloc()` to ensure that each object starts at a cache-line boundary. The wrapper code cannot assume anything about the state of `unaligned_malloc()`. The macros in `allocator_interface.h` may prove useful. From analyzing your allocator's code and looking at its utilization when run through `mdriver`, argue about how much memory is wasted for aligned allocations. Use the `-W` flag when you run `mdriver`.

Although each object will need fewer cache lines, inefficient cache utilization can lead to *more* cache loads overall.

Now consider allocating memory in a more cache-friendly way that allows for a more compact packing. In other words, you can pack multiple objects into a single cache-line. Even though some of your objects may no longer be cache-aligned, don't forget that objects still need to be 8-byte aligned. The allocator in `packed_allocator.c` is targeting a single-threaded workload and aims to ensure that each object spans a minimal number of cache lines. The provided memory allocator stores the size of a block next to that block. Let's keep this block header and think about where we need to store it in relation to the pointer returned to the caller and what we might need to store in it.

Write-up 2: Implement `packed_malloc()` in `packed_allocator.c`. Where did you allocate the block header? Report the utilization and performance scores. Based on analyzing your code and running it through `mdriver`, how much memory is wasted overall by your aligned allocator? Use the `-P` flag when you run `mdriver`.

6 Allocator overheads

Write-up 3: Do we need to allocate the header in `packed_allocator.c` if, instead of the classic `free(p)` interface, programmers were in charge of passing the original size as in `free(p, size)`?

Write-up 4: Assume that we have only one object size for our next allocator: 64 bytes. Do we need to allocate a size header at all? Fill in the `fixed_aligned_init()`, `fixed_aligned_malloc()`, and `fixed_aligned_free()` methods in `fixed_aligned_allocator.c` to implement a cache-aligned fixed-size allocator. (In particular, make sure you implement a free list this time.) Show the utilization and performance of your allocator. Use the `-F` flag when you run `mdriver`.

Now assume that we need to support large (64-byte) and small (32-byte) object sizes. We still wish for these objects to be cache-aligned, that is, at least one endpoint of each object must lie on a cache-line boundary. Although we could use large allocations for all objects, this internal fragmentation can double the memory requirements. Can we support allocation and deallocation of two sizes with **zero** space overhead?

Let's implement these ideas in `smart_allocator.c`. Suppose that you can change the interface for your allocator such that any pointer returned by `smart_malloc()` needs to be accessed through the `SMART_PTR()` macro.

Write-up 5: Where and how can you store the size of each allocation? Following this assumption, the `smart_free()` implementation in `smart_allocator.c` uses the `SMART_PTR()` and `IS_SMALL()` macros to operate on a given pointer. Implement the `SMART_PTR()` and `IS_SMALL()` macro definitions in `allocator_interface.h`.

Write-up 6: Using a single allocator for both different sizes might still suffer from external fragmentation. If the small and large objects need to be cache-aligned, then we might need to waste space to allocate a 64 byte object on a cache line boundary. What would you do with the "wasted space"? Implement `alloc_aligned()` and `smart_malloc()` in `smart_allocator.c`, and show your utilization. (Use the `-S` flag with `mdriver`.)

Write-up 7: *(No implementation required.)* What can you do when you need to allocate a 32-byte object when you have free-list entries of size 64? What would happen once you have run out of space and keep breaking up large objects, but never coalesce small, adjacent objects into large ones? What can you do when you have run out of space and need to allocate a 64-byte object when you have free list entries of size 32?

Write-up 8: *(No implementation required.)* Allocating space has to be done in contiguous regions of memory. Is it possible to coalesce two 32-byte objects into a 64-byte object if the 32-byte objects are not adjacent in memory? How would you implement this coalescing if possible?