Memory Allocator with FIFO Free List

Create your GitHub repo for this lab here: https://classroom.github.com/a/eMu4D7oX

In this lab, we'll build our own memory allocator that serves as an alternative to the default `malloc` family of functions. If we take a look at the `man` pages for `malloc` et al., we'll find that we need to implement four functions:


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

…eate memory out of thin air: it needs to request memory from the …e the `mmap` *system call*. `mmap` works very similarly to `malloc`, but …nally, its counterpart to release memory, `munmap`, requires that we …be released. (And as you'll recall, all `free` needs is a pointer to the …

…ces, we'll *prefix* each allocation with a struct that contains some …piece of metadata will be the allocation size, which we'll pass to …y. The other information we need to maintain is whether or not the …emory back to the OS every time `free` is called, our allocator won't …execute, so we want to reduce communication with the OS as much …d list that tracks all the free blocks that haven't been released back to the OS yet.

This means our allocations will look something like this:

```
[Header][Data] --> [Header][Data] --> [Header][Data] --> etc.
```


You are given the basic skeleton of a mostly-functioning memory allocator and must enhance it to support:

1. A free list (linked list of memory blocks)
   - When an allocation is freed, it is added to the head of the list rather than unmapped immediately
2. Block reuse capability
   - When servicing a `malloc` request, you should **first** check the free list for a viable block before `mmap()`ing a new one
   - Blocks should be removed from the **end** or **tail** of the list. Since additions happen at the head and removals happen at the tail, we are effectively creating a FIFO cache of free blocks.
3. Threshold-based memory release
   - If a large number of blocks have been freed, you should release memory back to the OS with `munmap`. We define a "large number" with a configurable threshold.
   - By default, the threshold is 100 blocks. If the free list contains more than 100 blocks, then subsequent `free` calls will unmap the block instead of adding it to the free list.
   - The user may supply their own threshold with an environment variable, `ALLOC_THRESH`. You can use the `getenv` function to get the value of this environment variable, and if it is set then you should use the provided threshold instead of the default.

Look through the starter code to familiarize yourself with memory allocation basics. To build the allocator, you'll compile like so:


```
cc allocator.c -Wall -fPIC -shared -o allocator.so
```

And then to actually use it:

```
LD_PRELOAD=$(pwd)/allocator.so ls
```


(In this example, we're running `ls` with our allocator).

Implementation Hints

- You have freedom in designing this lab, but remember to break functionality up into separate functions.
- You may want to test your linked list functions outside the context of the memory allocator; to do this, you can temporarily add a `main` to your code.
- To create the FIFO cache of free blocks, you'll need to implement a doubly-linked list, i.e., each node in the list should contain a link to its previous and next neighbors.
- Keep track of the head and tail of the list so you don't need to iterate through it to find either end.
- There are `TODO` notes in the starter code to remind you what needs to be done.

Instrumenting Your Code

Use the `TRACE` macro to evaluate whether your allocator is functioning properly; the events you must trace are:

- `malloc()`:
  - Allocating a new block
  - Reusing an existing free block (instead of allocating a new one)
- `free()`:
  - Adding a block to the free list
  - Unmapping a block because the free list is already full
- `realloc()`:
  - When a block is already large enough to accommodate the realloc request
  - When a block is "resized" (allocate a new block, copy the old data to it, and then free the old block)

You can see an example of how this might look in the test run below.

Testing your allocator

You should be able to run the `find` command with your memory allocator on a large directory. For example, run:

```
ALLOC_THRESH=10 LD_PRELOAD=$(pwd)/allocator.so find /
```

To execute `find` over the entire file system tree. You may want to compile your allocator with logging and tracing turned off first so it finishes faster:

```
cc allocator.c -shared -fPIC -DLOGGER=0 -DTRACE_ON=0 -o allocator.so
```

Next, you can use this program as a test case. Compile your allocator with `-DTRACE_ON=1` and then run it:

```
$ cc allocator-test.c -o allocator-test
$ ALLOC_THRESH=5 LD_PRELOAD=$(pwd)/allocator.so ./allocator-test
[TRACE] malloc(): Allocated block [0x7f2516018000]: 25 bytes
[TRACE] malloc(): Allocated block [0x7f2516017000]: 27 bytes
[TRACE] malloc(): Allocated block [0x7f2516016000]: 29 bytes
[TRACE] malloc(): Allocated block [0x7f2516015000]: 31 bytes
[TRACE] malloc(): Allocated block [0x7f2516014000]: 33 bytes
[TRACE] malloc(): Allocated block [0x7f2516013000]: 35 bytes
[TRACE] free(): Cached free block [0x7f2516018000]: 25 bytes
[TRACE] free(): Cached free block [0x7f2516017000]: 27 bytes
[TRACE] free(): Cached free block [0x7f2516016000]: 29 bytes
[TRACE] free(): Cached free block [0x7f2516015000]: 31 bytes
[TRACE] free(): Cached free block [0x7f2516014000]: 33 bytes
[TRACE] free(): Unmapped block -- [0x7f2516013000]: 35 bytes
[TRACE] malloc(): Reused block -- [0x7f2516018000]: 25 bytes
[TRACE] malloc(): Reused block -- [0x7f2516016000]: 29 bytes
[TRACE] malloc(): Allocated block [0x7f2516013000]: 36 bytes
[TRACE] realloc(): Unchanged ---- [0x7f2516013000]: 36 bytes
[TRACE] malloc(): Allocated block [0x7f2516012000]: 1024 bytes
[TRACE] free(): Cached free block [0x7f2516013000]: 36 bytes
[TRACE] realloc(): Resized block  [0x7f2516013000]: 36 bytes -> 1024 bytes
[TRACE] malloc(): Allocated block [0x7f2516011000]: 1048 bytes
X is: 99
[TRACE] malloc(): Reused block -- [0x7f2516017000]: 27 bytes
[TRACE] free(): Cached free block [0x7f2516018000]: 25 bytes
[TRACE] free(): Cached free block [0x7f2516016000]: 29 bytes
[TRACE] free(): Unmapped block -- [0x7f2516012000]: 1024 bytes
[TRACE] free(): Unmapped block -- [0x7f2516017000]: 27 bytes
```

Benchmarking

The whole premise of this assignment is that less system calls will mean less overhead, and therefore give us better performance. But you shouldn't trust any theory without testing it first. We'll collect some empirical data on just how effective our free list is.

First, install the `time` command (we're not going to use the one that is already built into your shell):

```
$ sudo pacman -Syu time
```

Now that we have the time command installed, you can measure performance with the following:

```
$ ALLOC_THRESH=100 LD_PRELOAD=$(pwd)/allocator.so /usr/bin/time find / > /dev/null
```

NOTE: compile first **without** logging and tracing turned on!

You'll get output that looks something like:

```
0.96user 1.56system 0:02.58elapsed 97%CPU (0avgtext+0avgdata 83532maxresident)
128inputs+0outputs (0major+200778minor)pagefaults 0swaps
```

So, this took 2.58 seconds to run. You can modify the value of `ALLOC_THRESH` (including setting it to 0 to effectively disable caching) and measure how long each run takes. We need to collect data for a range of thresholds, so the following shell script is a good starting point. It will output the total elapsed time to a file named (threshold).out for each threshold, allowing us to find the best run time.

```
#!/usr/bin/env bash

rm -f *.out
echo "Starting benchmark"
for (( i = 0; i <= 40; i += 10 )); do
    export ALLOC_THRESH=$i
    output_file=$(printf "%03d.out" "${i}")
    echo -n "${i}..."
    LD_PRELOAD=$(pwd)/allocator.so /usr/bin/time -q -f "%e" \
        -o "${output_file}" find / &> /dev/null
    sed -i "1s/^/${i}\t/" "${output_file}" # Add threshold to beginning of file
done
echo 'done!'

echo 'Combining outputs into benchmark.out...'
cat *.out > benchmark.out
```

Put the script in a file, such as `run.sh`, make it executable with `chmod +x run.sh`, and run it. You'll notice that the script only runs through the first few thresholds, so you should modify it to do a more exhaustive set of benchmarks.

After doing this, we can find the absolute best run time with:

```
sort -n -k 2 benchmark.out | head -n1
```

But absolute best may not be exactly what we want. What are the tradeoffs here?

Finally, to get fancier, you can plot this in your terminal using termgraph:

```
$ sudo pacman -Sy python-pip
$ python3 -m pip install termgraph
$ ~/.local/bin/termgraph benchmark.out

0  : ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 2.93
10 : ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 1.85
20 : ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 1.61
30 : ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 1.51
40 : ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 1.41
50 : ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 1.37
```

What to Turn In

- Your memory allocator, `allocator.c`
- A Makefile that builds `allocator.so`
- Results from your benchmarks and a short discussion on what the best threshold might be.