Assignment 7: A Better Memory Allocator

> **Due:** Friday, March 24, 10pm
>
> **Starter code:** Use your Assignment 5 code as a starter for this assignment.
>
> **Submission:** This is a **pair assignment**, but you can work alone, if you so choose.
>
> Submit the contents of your repository via Gradescope. See Deliverables below for what to submit. If you are working with a partner, do not forget to include their name with the submission. **Only submit one solution for both partners.**
>
> There will be no autograder for this assignment ahead of the deadline. Read the requirements and run tests locally. Note, that the sbrk stats from Assignment 5 are not useful anymore.

For this assignment, we will write a safer and more efficient version of the memory allocator from Assignment 5. Our goal is to get performance closer to that of the actual `malloc`, while making it safe for use by multiple threads.

Task 1: Use `mmap` to Allocate Page-sized Blocks

Understanding `mmap`

For the basic allocator implemented in Assignment 5, you were asked to use the sbrk system call to change the size of the heap. This call is not the only way to request memory for our process. In fact, the `sbrk` syscall is considered *deprecated* and its use is discouraged. We have used it in our first allocator because it provides a simple and easy-to-use interface for requesting heap memory from the OS.

So what does a modern Unix/Linux want us to use for memory allocation? The modern way to allocate memory is to directly request *pages* for a process using the mmap system call. In general, `mmap` allows us to map a file or a device into memory, meaning that any reads/writes within the memory region returned from a successful `mmap` call are reflected in the file (more on this in the file system project). In other words, `mmap` allows us to allocate memory from the operating system, backed by a particular file. On Linux and some other operating systems, we can even request "anonymous" mappings, meaning that the returned region is not backed by any file. In this mode, `mmap` can behave like `malloc` that gives out pages of memory.

An "mmapped" region's size is always aligned along the boundary of a page. This means that one constraint is that allocations with `mmap` should be made in multiples the size of a page. Even if you specify a size that is not a multiple of the system's page size, `mmap` will round up to the nearest page. For example, if our page size is 4KB (4096 bytes), and we only use 12 bytes in total during our program's execution, we have quite a bit of waste! In practice, for large desktop applications this is not a major issue.

Task 1:

The first task is to update our allocator implementation to use `mmap` instead of `sbrk`. Remember that, unlike `sbrk`, `mmap` will return a pointer to the beginning of a page of memory. Here is the overall updated strategy for `mymalloc` and `myfree`. You might not need to update `mycalloc` at all. However, see the last hint.

mymalloc

(`BLOCK_SIZE` is the size of your block header)

a) For requests of size $<$ `PAGE_SIZE` $-$ `BLOCK_SIZE`:

  - Use your free list to see if there is a large enough block to accommodate the request.
  - If a block is found, select it and remove it from the free list (marking it as "not free")
  - If a large enough block is not found, use `mmap` to request a new **page** and set it up as a block.
  - If the block is larger than the requested memory size, determine whether it is feasible to split it using the criterion: After "shrinking" the original block, is the "leftover" big enough to include a block header and at least 8 bytes of user memory?
  - If splitting, set up the leftover as a new memory block and add it to your free list.
  - Return the selected block's user memory pointer.

b) For requests of size $\geq$ `PAGE_SIZE` $-$ `BLOCK_SIZE`:

  - Compute the number of pages needed to satisfy the request (including a block header) and allocate that many pages with mmap.
  - Set up the block header and set the size to the no. of pages $\times$ `PAGE_SIZE`.
  - Return the block's user memory pointer.

myfree

a) For a block that occupies less than `PAGE_SIZE` bytes of memory in total: add the block to your free list and coalesce (see below) if needed.

b) For a block that occupies `PAGE_SIZE` or more bytes of memory in total: use `munmap` to unmap it.

Coalescing Free Blocks

When adding a block into your free list, keep the list sorted by the memory address of the blocks. This will allow *coalescing*: whenever two blocks in the free list form a continuous area of memory, they should be merged into one block (coalesced).

Task 2: A Thread-safe Allocator

Data races can affect memory allocators too. In a multi-threaded environment, we cannot simply make requests to our `malloc` and `free` functions based on our previous implementation. We could have a scenario where two or more threads request memory at the same time, and potentially all allocate the same block of memory in the heap. This would certainly be unlucky!

Luckily, we have mutexes to enforce mutual exclusion and help protect against data races. Remember, when we use pthread_mutex_lock and pthread_mutex_unlock, this creates a critical section where only one thread that has acquired the lock can execute a section of code, thus enforcing sequential execution over shared data.

Task

Implement locking mechanisms such that, whenever there is an allocation (`malloc` or `calloc`) or deallocation (`free`), a lock protects that section of code from being run by another thread. You should lock as little code as possible, while maintaining thread safety.

Assignment Strategy

1. You have a previous implementation of an allocator. Structurally many pieces will look the same, so you do not need to build things from scratch.
2. You do not have to work on this assignment in the order of the tasks. Task 1 and Task 2 are independent of each other. Start with Task 2 if you have no idea where to begin with Task 1.
3. One way to approach the assignment:

    a) Start single threaded, with your code from Assignment 5
    b) Initially, ignore splitting and coalescing blocks, and just accept wasted memory when using `mmap`
    c) Move to multiple threads, adding locks around all allocations and frees
    d) Add splitting of blocks
    e) Add coalescing of blocks

Deliverables

All Tasks Implement your memory allocator in `mymalloc.c` and include any additional `.c` and `.h` files your implementation relies on. For example, you might want to compile your helper data structure separately.

Commit the code to your repository. Do not include any executables, `.o` files, or other binary, temporary, or hidden files.

Once you are done, remember to submit your solution to Gradescope and *do not forget to include your partner*.

> ### Hints & Tips
>
> - Check out the man pages of `mmap`, `munmap`, `malloc`, `calloc`, `free`, `realloc`, ...
> - `mmap` arguments. Request multiples of `PAGE_SIZE`. You'll want an *anonymous private* mapping, that is both *readable* and *writable*. The flags you're looking for are `MAP_PRIVATE` and `MAP_ANONYMOUS`, and the protection `PROT_READ` and `PROT_WRITE`. For anonymous mappings, use `-1` as the file descriptor. Offset should be `0`.
> - Compile and test *often*.
> - Use `assert` to check that your assumptions about state are valid.
> - **Write your own (unit) tests.** Doing so will save you time in the long run, especially in conjunction with the debugger. In office hours, the instructors or the TAs may ask you to show how you tested code that fails.
> - Unit tests can be just a bunch of functions and a main, with `asserts` to check expected results. Use our tests for queue/vector from Assignment 4 as an example.
> - Follow good coding practices. Make sure your function prototypes (signatures) are correct and always provide purpose statements. Add comments where appropriate to document your thinking, although strive to write self-documenting code.
> - Split code into short functions. Avoid producing "spaghetti code". A multi-branch **if-else if-else** or a multi-case **switch** should be the only reason to go beyond 40-50 lines per function. Even so, the body of each branch/case should be at most 3-5 lines long. Consider applying the *Single Level of Abstraction* principle
> - It is ok to use global variables, but be judicious with their use. Give them meaningful names and explain their purpose.
> - If multiple threads are reading and writing the same variable, consider using `pthread_mutex_lock` and `pthread_mutex_unlock` to ensure consistency.
> - Have one global lock for initializing your free list.
> - At first, be conservative with your locks. That is, don't try to optimize too much from the get-go by moving stuff inside and outside of locked sections–take the conservative approach first, then optimize later.
> - Make sure when you `memset` (if you are using `memset` in `calloc`) that you are not `memset`'ing over your block.
> - Write additional tests to examine high volumes of `mallocs` and `frees`, `mallocs` of a wide range of sizes to exercise the two block size strategies, in particular the edge cases.
> - If your program hangs up at some point and does not continue running, it is likely a deadlock. Check the source that every time you lock, you unlock too.
> - Use `sysconf(_SC_PAGE_SIZE)` to get the OS's page size. Check the manpage for `sysconf` to see which header you need to include.
> - On Linux, anonymously mapped pages are guaranteed to be initialized to 0, meaning a freshly mmapped page does not need to be `memset` to initialize it. Could you use this fact to improve performance somewhere?

Going Further and Additional Resources

- A nice survey of dynamic memory allocation strategies: http://www.cs.northwestern.edu/~pdinda/ics-s05/doc/dsa.pdf

- Investigate the 'buddy allocator' and 'arena allocators'
- John Lakos Memory Allocator talks

    - part 1: https://www.youtube.com/watch?v=nZNd5FjSquk
    - part 2: https://www.youtube.com/watch?v=CFzuFNSpycI

- Some high performance memory allocators are

    - tcmalloc
    - jemalloc

- Pool Allocators: https://blog.molecular-matters.com/2012/09/17/memory-allocation-strategies-a-pool-allocator/
- Our allocator works on the 'heap' working with dynamic memory. Other memory may work on the 'stack' which is for really *hot* code. Read up on alloca (in alloca.h) and understand that we could also write a custom stack allocator.