Assignment 9: Advanced Memory Allocator

Due: Friday, April 1, 10pm

Starter code: See Assignment 9 on Canvas for the Github Classroom link.

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.

There will be no autograder for this assignment ahead of the deadline. Read the requirements and run tests locally.

For this assignment, we will write a safer and more efficient version of the memory allocator from Assignment 7. 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 7, you were asked to use the sbrk function/system call to change the size of the heap. This call is not the only way to request memory for our process.

Linux (as well as other modern Unix variants) provides another way to request memory 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`.

An "mmapped" region's size is always aligned along the boundary of a page. This means that one constraint of mmap 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.

A brief set of slides with some notes on `mmap` is provided with the starter code.

Because of concerns about fragmentation, mmap may not be optimal, especially if a program makes lots of little allocations. However, `mmap` has its advantages. It can be very fast because it maps directly to RAM. Remember, whether we are requesting memory with `mmap` or `sbrk`, at the end of the day, what we get is just a pointer to a block of memory.

Task

Our first task will be to try to get the best of both worlds. Extend the allocator from Assignment 7 to use `mmap` with the following simple heuristic:

- If the amount of memory needed (requested size + metadata) has a size of a page or more, always use `mmap`.
- If it is anything less, use `sbrk`.

Remember that the size of the memory region we get from `mmap` will be always a multiple of a page. If you are not using a full page, **you will need to split your blocks** to use the memory efficiently. That is, if there is left-over memory from a previous `mmap` when `malloc` is called, the allocator should use that block.

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.

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 7
    b) Initially, ignore splitting blocks. Use `sbrk` for small allocations and then mmap for large allocations.
    c) Move to multiple threads, adding locks around all allocations and frees

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`, `sbrk`, `malloc`, `calloc`, `free`, `realloc`, ...
- 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.
- 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.
- Have a second global lock around `sbrk` calls, especially if you are getting errors.
- Be conservative with your locks. That is, don't try to optimize too much by moving stuff inside and outside of locked sections–take the conservative approach first, then optimize later.
- You probably will need locks for your `free` function and `calloc` as well.
- 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 `sbrk` and `mmap` implementations, and other 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.

F.A.Q.

Q: Do I need to `munmap` the memory?
A: No–for our allocators we will not worry about unmapping the memory. In practice our operating system will do this for us.

Q: Can I just allocate for 10000 cpus? There is no possible way the instructors can test for that.
A: In the past students have done this, so we manually check if you statically allocate your lists with some fixed value. Nice try!

Q: Do I need to merge blocks?
A: It would be good to merge adjacent free blocks (right after you do a free), but is not required.

Q: If you need to split a block but the amount of remaining memory in the block is less than the amount of memory of a new block (which we need to split the remaining memory into a new block), what should we do?
A: In this case you do not need to do anything, that is an acceptable amount of fragmentation to live with.

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.

Changelog

03/30/2022

- Add a hint on how to get the current page size.