OS2 Lab 3: Writing a multi-threaded memory allocator

imec-DistriNet, KU Leuven Ghent Technology Campus

November 16, 2022

1 Introduction

In systems programming, directly requesting memory from the operating system using system calls, like `mmap` on Unix, is expensive (due to the mode switch) and too coarse-grained for immediate application use (returning page-sized chunks of memory at once). A memory allocator solves these problems by sitting between the application and the OS to serve allocation requests with more fine-grained chunks, e.g., unused parts of mapped pages or previously freed memory.

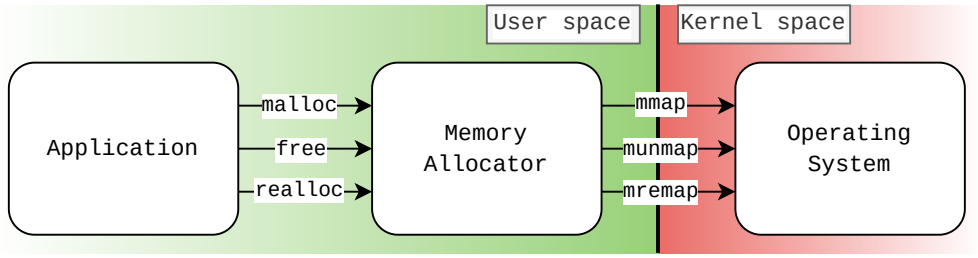

Figure 1: Position of memory allocator relative to application and OS.

Different types of memory allocators exist, that are typically tuned to perform well for a particular workload. The performance overhead of an allocator manifests both directly, i.e., the time it takes to serve a single allocation/deallocation request, and indirectly by determining the application memory layout, including the influence of that on, e.g., the processor's data cache. All the while, general-purpose memory allocators do not know about the memory access and allocation behavior of a specific application, sometimes resulting in noticeable performance losses. For this reason, some applications will use more custom allocators, or sometimes even a dedicated allocator for a given data structure for which the allocation/access pattern is highly predictable (e.g., Stack, Queue).

In multi-threaded applications, memory could be allocated, used, and freed again by different threads. This, again, increases the complexity of implementing a good general-purpose allocator. It is good practice to avoid synchronizing all memory allocation behavior across all threads, as it remains very common for memory to stay local to a single thread during its lifetime. This is usually achieved using thread-local allocators, that fall back to synchronizing behavior when handling a block of memory that was allocated from a different thread.

2 Implementing a Multi-threaded Memory Allocator in Java

The purpose of this lab is to implement a simulated high-performance multi-threaded memory allocator in Java. As a starting point, and a framework for testing, we have provided both an OS abstraction that implements the slow, coarse-grained, Unix-like `mmap` and `munmap` interface, as well as a multi-threaded application that allocates and frees memory in a semi-realistic fashion, using the standard C-like `malloc`,

`free`, and `realloc` interface. Your memory allocator should use the OS abstractions efficiently to serve the application's allocation requests as defined in the `Allocator.java` interface. A code skeleton, and an example of what *not* to do, is given by `MyAllocatorImpl.java`, which implements the `Allocator` interface but synchronizes every single request, and simply forwards to our OS abstraction in `BackingStore.java`, yielding a very coarse-grained result.

2.1 Memory Allocation Background

When talking about user-space memory management, there are at least 3 types of memory:

1. **Allocated memory**: Memory currently in use by the application.
2. **Available free memory**: Memory currently not in use by the application, but already obtained from the OS.
3. **Unmapped memory**: Memory that is currently not mapped, unavailable.

The point of an allocator is simply to distinguish between these types. The way in which they do it can differ, from maintaining an N-byte granular shadow memory, to keeping lists of free or allocated blocks, or a mix of multiple approaches. The general flow during allocation requests (`malloc`) is shown in Figure 2.

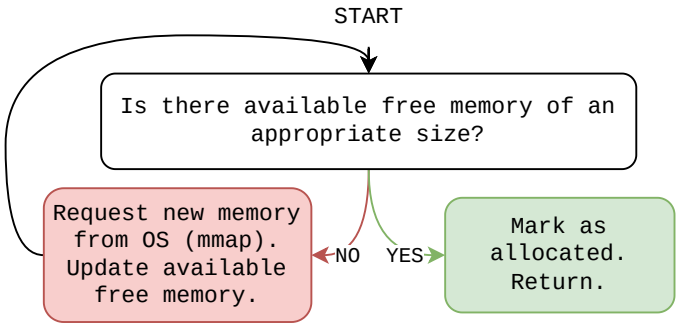

Figure 2: Flow of typical memory allocation routine.

The speed of a call to `malloc` is determined by how fast it can find a suitable block of free memory to use for the new allocation. The fastest approach is called *first fit*, which returns as soon as it finds a correct block. However, the speed of an allocator is not only determined by the execution time of its allocation routines. The first found block may be slightly too big, cutting off a small part of free memory that is unlikely to be filled by another request. This causes memory *fragmentation*, and can have strong adverse performance effects on the application. An alternative solution is to use *best fit*, which tries to avoid fragmentation by searching for a perfectly fitting block of free memory.

Optimizing these searches for free memory is an active area of research. A very common solution is to maintain different contiguous memory regions for allocations of the same size (size *bins*), such that the size of an allocation can be stored more efficiently at the start of the region, and every block in a region is equally well-fitting. For instance, the allocator that comes with most Linux distributions, glibc's `ptmalloc`, uses 32 size bins, and allocation sizes are rounded up to the smallest size they fit in. For larger requests (multiple pages), it directly forwards to `mmap`. You can find more information at https://sourceware.org/glibc/wiki/MallocInternals.

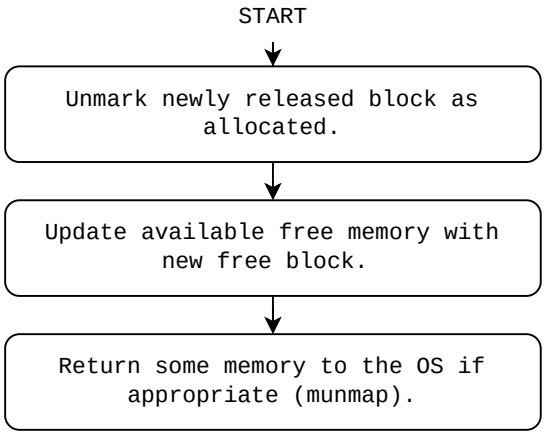

Figure 3: Flow of typical memory deallocation routine.

Figure 3 visualizes the flow of a `free` call. The implementation of `free` is tightly bound to `malloc`. Generally, allocators tend to focus on `malloc` performance, since `free` calls can often be batched more easily by the application.

Finally, `realloc` can be implemented naively, as in `MyAllocatorImpl`, by subsequently calling `free` and `malloc`. However, this is often a waste of precious CPU cycles. When an allocation is resized to a slightly smaller size, it's technically possible not to do anything, or simply only inform the rest of the allocator of the new size of the allocation: a smaller allocation is guaranteed to fit in the previous, bigger one. Even for `realloc` requests that want to enlarge the existing allocation (more common), the allocator could first check if there is any free memory adjacent to the already allocated block. If so, and if it is sufficiently big, that free memory could be used to expand the allocation into without falling back to the slower `free` → `malloc` path.

Note that our provided `MyAllocatorImpl` supports only the bare minimum of these operations. It can only distinguish between memory that is currently allocated, and memory that is not. For all `malloc` requests, it simply forwards to `mmap`, ignoring any excessive size overhead and memory usage that causes. It still keeps the originally requested size around, to know how much memory to `munmap` during `free` calls.

Multi-threaded allocators In a multi-threaded context, we want to accommodate the very common case of memory that stays local to the thread. Hence, it is undesirable to synchronize all allocation calls across all threads, since that will needlessly serialize unrelated allocation calls in unrelated threads. Similar to the `Account` locking problem from the previous banking simulation, we need to come up with a solution that only synchronizes allocation requests that interfere with each other. One very popular solution is to keep a separate allocator per thread. During allocation calls, the correct allocator is selected based on the current thread (`Thread.currentThread().getId()`). This prioritizes thread-local allocation patterns, and is an extremely common choice among many popular, high-performance allocators. On `free` and `realloc` requests, if the existing memory block is not found in the current thread list (was allocated from a different thread), we need to iteratively synchronize on and find the right block in the other thread's allocators. To speed this up, one possibility can be to define contiguous memory regions per thread, so we can find the origin thread of an allocation from its address.

Other complications exist, including threads starting and stopping throughout the lifetime of the application. These cases are not triggered by our sample MultiThreadedApplication, but we encourage you to think about them, discuss them in the report, or handle them in your implementation.

2.2 How to Start

We welcome your creative ideas to address the challenges of multi-threaded memory allocation. There are many available resources online about (multi-threaded) memory allocators, which you can draw inspiration from for this lab. Java supports many concurrency and synchronization primitives beyond the ones we used in the labs until now. We strongly encourage you to explore, among others, the `AtomicInteger`, `ThreadLocal`, `Lock`, and `ReadWriteLock` types when implementing your solution.

You can immediately observe the impact of poorly implemented memory allocation in our provided skeleton code, by running it with allocations enabled and disabled, using the `ENABLE_ALLOCATOR` constant in `Transaction.java`, and comparing the transaction rates. This is not an entirely fair assessment, since apart from their synchronization behavior, allocation requests also do some additional work, but this work is insufficient to explain the drastic performance impact you observe.

Your allocator should implement the interface shown in `Allocator.java`. You can change which allocator is used by the `MultiThreadedApplication` by changing the initialization of the `Allocator.instance` singleton variable.

2.3 How to Test

The `Allocator` interface you implement contains the `isAccessible` methods, which are called by our testing application to ensure that memory that *should* be allocated, is in fact allocated, and vice versa. You should implement these yourself since they are specific to your allocator design. After submission, we will use these methods again to test your application more rigorously.

Testing a multi-threaded allocator can be hard, especially once it allows a high degree of concurrency. For one, verifying that recently freed memory is in fact not accessible is often not possible reliably, as it might already have been re-used by a new allocation request. For this reason, we encourage you to write your own testing code beyond the application we provide, including single-threaded tests.

2.4 Relation to Previous Lab

As you might have noticed, the multi-threaded application we provide strongly resembles the extended Banking simulation of the previous lab. Although there will be no dedicated feedback for the previous lab, this allows you to compare your own solution to ours, and learn from the differences between them. Feel free to ask any questions about the previous lab at any point.

We deviate from the previous assignment in the following ways to aid this lab:

1. We avoided implementing a *Stop The World* mechanism to log the total balance every 5000 transactions, since that needlessly slowed down the transaction rate.

2. We now log transaction insertion rate, transaction processing rate, and the number of in-flight transactions every second in a dedicated `Logger` thread. None of its operations noticeably impact the `Transferer` and `Worker`, yet the statistics it provides suffice to estimate the saturation rate of the `Bank`, and hint at potential bottlenecks.

3. The single, shared transaction buffer between the `Transferer` and `Worker` threads proved to be too much of a contention point. We now switched to a load balancing mode where each `Worker` has its own, private buffer.

4. Throughout the lifetime of a `Transaction`, it allocates, frees and reallocates memory at various points, in various threads, to help test your allocator.

2.5 Overview of the Skeleton Code

The `.zip` file on Toledo contains two Java packages: `Allocator` and `BankingSimulation`. The `MultiThreadedApplication.java` file contains the `Main` method to run the skeleton code in its default configuration.

- The `BackingStore` class implements OS abstractions your allocator should build upon. It handles memory in 4096B-sized chunks and is fully, globally synchronized across all threads. You should consider calls to this interface slow, and try to avoid holding locks while calling them: your critical sections should be as fast as possible to reduce lock contention.
- We encourage you to include testing code in the implementation of your allocator. Sometimes, this can be as simple as asserting that some conditions about the input parameters to your internal allocator functions always hold true. You can use Java's `assert` keyword for that, but keep in mind that these assertions are disabled by default in most JVMs. Use the `-ea` command line options when invoking Java to enable them.
- Our provided implementation of the `realloc` function is extremely basic. You are encouraged to include a more optimal implementation in your own allocator, i.e., using in-place resizing.
- Although the current `MyAllocatorImpl` is fully synchronized (which is bad), we did not decorate the methods themselves with the `synchronized` keyword. On the one hand, this is to hint at you that a fully synchronized allocator interface is not ideal. On the other hand, Java has some unintuitive behavior about `synchronized` methods implementing non-synchronized interfaces and vice versa. You should not run into any issues as long as you do not make `malloc`, `free`, `realloc` or any of the `isAccessible` methods synchronized.

3 Report

The report for this lab should mainly focus on the steps you took to avoid globally synchronizing allocation/deallocation requests. Compare your solution with alternatives you considered, and with our naive `MyAllocatorImpl`. Why was our allocator design so bad?

In addition to this, the report should outline the design and rationale behind your allocator, and the kind of memory access/allocation patterns it optimizes for. Include a discussion of your expectations for its direct and indirect overhead.

You are encouraged to include benchmark results and performance measurements in your report. You can use your own benchmarking tools for this, or the provided `MultiThreadedApplication` in various configurations: without allocations, with your allocator, with our naive `MyAllocatorImpl` version, etc.