Operating Systems 2
Writing a multi-threaded memory allocator

Zeger Use, Thomas Van Hecke, Milo Regnard

November 21 2025

Contents

1 Introduction

...

2 Provided allocator

At the start of this lab we were given a working allocator. This allocator is poorly designed and therefore the performance was suboptimal. We first analyzed why it was bad, followed by creating our own implementations.

2.1 Management

An allocator should act as a memory management layer between a process and the operating system (OS). The provided allocator makes a memory call to the operating system for every allocate and free call. In essence this allocator serves no purpose. Ideally we want to limit the amout of OS-calls, because these are slow.

2.2 Synchronization

The provided allocator's methods are all synchronized on "this". Which means that only one thread can use the allocator. This is bad for performance, because two unrelated threads still have to wait on each other. Two unrelated threads are threads that use different resources, ideally we only want to synchronize when two threads are related.

3 Allocators

We have implemented three different allocators: Best-fit, buddy and single sized (arenas). Our approach to creating a working multi-threaded allocator is as follows: we start by creating a single threaded allocator (STA), then we refine the synchronization on the STA. We finish by creating a multi threaded allocator (MTA) that uses a pool of STA's.

3.1 Best Fit Allocator

This allocator tries to find the best possible block of memory to fit the request. This allocator is a dynamic allocator which means that is doesn't have internal fragmentation. Because of the best fit dynamic it also reduces external fragmentation. The allocator makes uses of 2 "lists": free list and allocated list. The free list is only used in allocation requests (alloc) and both are used in a deallocation request. We can structure the path so that only one of these lists needs to be aquired. This means that we already reduce lock contenion by around half. Internaly these "lists" are replaced with more performant data structures. For example when a allocation request comes in we need to quickly get a free block that is the smallest one that fits the request. Using a list we need to walk over all elements O(n). We can replace this list with a navigable set (Treeset) that is sorted on block size we can now do a 0(log(n)) search. For the free we also need a efficent way to get blocks before a to free block and after to see if we can merge. A list has a linear search which mean average lookup time of O(n). We can use a second data structure that we keep in sync with the other datastructure. This is also a treeset but sorted on addresses inplace of size.

3.2 Buddy Allocator

This allocator also keeps a free lists and an allocated list. This allocator is a semi-fixed size allocator. We can again program the allocator so that only at most one of these lists is used at a time reducing lock contention. To further improve lock contention we also split the freelist into multiple lists where each list is responble for a single buddy-order. This means that allocations that are far enough apart can fully independently fetch a free block. They still have a shared dependancy on the allocated list. This allocated list is in accuality a map once again for faster lookups.

3.3 Single Sized Allocator

This allocator uses arenas. An arena manages chunks of memory. Each chunk consists of multiple blocks that have a fixed size. The allocator uses two maps. One that maps the fixed size of a block to the arena, the other one maps base addresses to large allocation. An allocation is considered large if it is more than one page size (4096KB).

Each arena uses one freelist and two maps. The freelist keeps track of all the chunks that contain at least one free block. One map is used to map base addresses of a chunk to the free indices of the blocks in that chunk. The other maps base address to a bitset that represents the blocks.

The represent a block we only need two states, allocated (1) or not allocated (0). When working with an array of boolean there is still one byte per boolean. A BitSet is a Set where each element consumes one bit.

4 Multi Threaded Allocator

To convert our STA to a MTA we looked at 2 options: adding extra shared meta-data structures or implementing search mechanisms. Adding the extra shared resources adds a single shared object between threads which could cause lock contention. However lookups can happen more effiently. A search mechanism is when we iterate over every allocator to find the owner of the allocated memory. The search mechanism doesn't need extra memory but has the disatvangae that we need to obtain many locks of different allocators. This introduces extra lock-contention. This iterative approach is O(n) and won't scale well.

When using shared metadata structures we use Java's built in concurrency primitives, such as: ConcurrentHashMap.

We deliberatly chose to create a pool of 8-16 allocators, because this resembles the amount of core of a modern CPU. In that way each allocator can run on its own core. Though we also experimented by creating an allocator for each thread, but this proved negative in performance due to much heap allocation.

5 Overhead

When optimizing our allocator we had to make tradeoffs. Initially we used lists, but iterating over these list to find elements is O(n). If we want to search an element in O(log(n)) we can use a TreeMap, therefore we trade the LinkedList O(1) insertions, deletes and updates for O(log(n)) alternative. This extra effort is called overhead.

5.1 Direct

The BFA uses a sorted TreeSet, so for every insertion or deletion we have a small performance hit.

5.2 Indirect

6 Benchmarks

Allocator	ST [k tx/s]	MT [k tx/s]
BFA	70	200
SSA	240	400
BDA	200	400
Provided	45	-
None	440	-

Table 1: Benchmarks

We can see that the BFA hast the worst performance. SSA and BDA better, close theoretical performance and explain why