Operating Systems 2
Writing a multi-threaded memory allocator

Zeger Use, Thomas Van Hecke, Milo Regnard

November 21 2025

Abstract

...

Contents

1 MyAlloctorImpl

At the beginning of this lab we were given a working allocator, however this allocator is poorly designed. The are two major problems with this allocator. Firstly the memory management it provides and secondly how it is synchronized.

1.1 Management

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

1.2 Synchronization

Every allocator interface method is synchronized on "this". This means that only one thread can use the allocator. This is therefore a single threaded allocator (STA). 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 want to synchronize when two related threads want to use the allocator.

2 Allocator

We have implemented three different allocators. Each of our allocators is optimized for a specific purpose and therefore designed a certain way.

2.1 Design

All our allocators implement the provided allocator interface. The interface methods are: allocate, free, reAllocate, isAccessible. The allocators differ in how they implement these methods and which resources are shared. The two most essential methods are the allocate method and the free method.

2.1.1 Best Fit Allocator

This allocator uses a NavigableMap and two TreeSets. The two sets are in sync and sorted on custom criteria. These sets contain chuncks of memory called blocks. Sorting these sets results in faster lookups. Mention the use of skiplists and treeset

2.1.2 Buddy Allocator

This allocator uses a freelist to keep track of all the blocks that havent been allocated. This freelist is implemented using a java List. A map is used to map allocated base addresses to the order of a block. An order is value of the power.

2.1.3 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.

2.2 Purpose

The BFA optimizes to minimize internal fragmentation, whereas the buddy allocator optimizes to minimize internal and external fragmentation but also be performant. The SSA is analog to BA. Explain here which allocator is optimized for what metric

3 Process

When programming a multi threaded allocator (MTA) we want to avoid global synchronization. Global synchronization is when a thread takes the lock on the entire allocator. This is bad for performance, because two unrelated threads still have to wait on each other. To avoid global synchronization the java programming language provides concurrency and synchronization primitives. We made use of a ConcurrentHashMap (java concurreny primitive) and the synchronized keyword (java synchronization primitive). When using the synchronized keyword, we synchronize on shared resources instead of this (entire instance) and our synchronize granularity is smaller. The synchronize granularity is the size of a synchronization block.

4 Overhead

When optimizing our allocator we had to make tradeoffs. When using a freelist we need to update this freelist after every allocate and free call. This extra effort is called overhead

4.1 Direct

The BFA uses a TreeSet, under the hood this is balanced tree. The time complexity is O(log(n)), compared to O(1) like Hashmaps this provides some overhead if we want to scale this.

The BA uses a LinkedList which has a time complexity of O(n), this scales even worse than O(log(n)).

The SSA uses a LinkedList and a TreeMap in every arena. These provide the same overhead as metioned earlier.

4.2 Indirect

All the allocators use locks on shared resources. These locks can cause lock contention which bottlenecks performance.

5 Benchmarks

Provide benchmarks here in graphs. Use references in text. ...