Brandon Mountan

03/05/2025

CS6013 – Systems I

Why is my custom allocator (MyMalloc) slower than the system allocator?

System Call Overhead. MyMalloc uses the system calls mmap() and munmap() for every allocation and deallocation which involves switching between the user mode and the kernal mode and is therefore expensive. The system allocator uses both mmap() and sbrk() for large allocations and maintains a pool of pre-allocated memory for small allocations which reduces the frequency of system calls.

Internal Fragmentation. MyMalloc rounds up every allocation to the nearest multiple of page size which is 4 KB. So, for small allocations like 1 KB, this would waste 3 KB of memory. The system allocator uses advanced algorithms to minimize this fragmentation.

Hash Table Overhead. MyMalloc uses a hash table to keep track of allocated memory addresses and sizes. Hash table lookups and insertions are slower than direct pointer arithmetic. Linear probing for collision resolution can degrade performance as the hash table fills up. The system allocator embeds metadata directly into the memory blocks or uses more efficient data structures which avoids the hash table overhead.

Lack of Memory Pooling. MyMalloc does not reuse freed memory while the system allocator maintains a pool of freed memory blocks and reuses them which reduces need for frequent system calls.

No Optimization for Small Allocations. MyMalloc treats all allocations the same which can be inefficient for small allocations. The system allocator uses specialized techniques (slab allocation) to handle small allocations efficiently.

How to improve performance of my custom allocator (MyMalloc)?

Memory pooling. Instead of immediately deallocating memory with munmap(), maintain a pool of freed memory blocks and reuse them. Reduces frequency of system calls and improves performance of programs that have frequent allocations/deallocations.

Slab Allocation for Small Allocations. Slab allocator pre-allocates large blocks of memory and divides them into fixed-size chunks, reducing fragmentation and overhead.

Segregated Free Lists. Maintain separate free lists for different class sizes (e.g., 16, 32, 64 bytes, etc.). Results in faster allocation/deallocation by reducing search time for memory blocks.

Optimized Hash Table. Look into a chained hash table or balanced binary tree to replace the linear probing hash table. Could also embed metadata like allocation size directly into the allocated memory blocks to avoid need for separate hash table.

Batch Allocations. Allocate memory in larger chunks and split them into smaller blocks as needed. Reduces frequency of mmap() calls.

Tread-Local Storage. Maintains separate memory pools for each thread.

Lazy Deallocation. Mark memory as free and reuse it instead of immediately deallocating it with munmap().