**Custom Memory Allocator Performance Analysis**

**Abstract**

This document aims to analyze and compare the performance differences between a custom memory allocator and the system's default memory allocators (malloc and free). By implementing a simple memory allocator based on mmap and munmap system calls, we explore the impact of custom memory management implementations on performance.

**Implementation Overview**

The custom memory allocator utilizes a linear probing hash table to track the size of each allocation, enabling precise memory deallocation. Designed as an educational tool rather than a production-ready competitor to system allocators, this implementation offers theoretical flexibility at the cost of speed and memory efficiency.

**Performance Comparison**

**Speed**

The custom allocator is noticeably slower in allocating and freeing memory compared to the system's malloc and free. The primary reasons include:

1. **Additional management overhead**: The custom allocator needs to update the hash table on each allocation or deallocation, adding extra computational overhead.
2. **System call overhead**: The custom allocator directly uses mmap and munmap, which are heavier system calls than malloc and free because they require interacting with the operating system kernel to manage virtual memory.

**Memory Efficiency**

The custom allocator also falls short in memory usage efficiency compared to system allocators. Key reasons are:

1. **Page alignment**: Memory allocated through mmap needs to be page-aligned, potentially leading to more unused space per allocation.
2. **Hash table overhead**: Maintaining a hash table to track memory allocations requires additional memory, especially as the table grows.

**Optimization Suggestions**

Although the simple memory allocator based on mmap and munmap shows performance limitations, the following approaches can help improve it:

1. **Memory pooling**: Pre-allocating a large block of memory and slicing it into smaller pieces to fulfill requests can reduce the number of system calls.
2. **Hash table optimization**: Employing more efficient data structures to track memory allocations, such as open addressing hash tables or locking mechanisms, can reduce management overhead.
3. **Delayed deallocation**: Delaying memory deallocation or reusing allocated blocks internally can reduce calls to munmap, thus improving performance.

**Conclusion**

While custom memory allocators are valuable for educational purposes and understanding the internals of memory management, they cannot compete with the system's malloc and free in terms of performance. Implementing the above optimization measures can enhance the performance of custom allocators, but designing such systems requires balancing development time, complexity, and performance.