Project Overview: Dynamic Storage Allocator Implementation

Summary

This project involved implementing a dynamic storage allocator for C programs, essentially creating custom versions of the `malloc`, `free`, and `realloc` routines. This task required understanding the internal workings of these routines, exploring different strategies for memory management, and implementing an efficient and fast allocator.

Key Learnings

1. Memory Allocation

The project provided a deep understanding of how dynamic memory allocation works in C. It involved the creation of the `mm_malloc` function, which is designed to allocate a block of memory of at least the requested size and return a pointer to this block. The function had to ensure that the allocated memory does not overlap with any other allocated chunk and always return 16-byte aligned pointers, just like the standard `malloc` function in C.

2. Memory Deallocation

Implementing the `mm_free` function required understanding the process of releasing previously allocated memory back to the system. The function takes a pointer to a block of memory and frees it, ensuring that the block is correctly removed from the list of allocated blocks and doesn't leave any memory leaks.

3. Memory Reallocation

The `mm_realloc` function taught about resizing previously allocated memory blocks. This function, which changes the size of the memory block pointed to by `ptr` to `size` bytes, necessitated understanding the mechanics of memory copying, memory expansion, and managing fragmentation.

4. Heap Initialization and Management

The `mm_init` function provided insights into the initialization of the heap memory, which is required before calling any of the memory management functions.

Additionally, it was necessary to write a heap checker (`mm_check`) to ensure the consistency of the heap. This included checks like ensuring every block in the free list is

marked as free, there are no contiguous free blocks that escaped coalescing, every free block is in actually in the free list, no allocated blocks overlap, and pointers in a heap block point to valid heap addresses. This checker provided valuable insight into how to maintain and validate the integrity of the memory heap.

5. Performance Considerations

The project also necessitated considerations of the allocator's performance, focusing on space utilization and throughput. The challenge was to design an allocator that balances between minimizing the space overhead and increasing the speed of memory allocation and deallocation operations. This required strategic thinking and careful design of the allocation strategy.

Conclusion

This project provided hands-on experience with low-level memory management in C, reinforcing theoretical knowledge with practical skills. By developing a dynamic storage allocator from scratch, understanding of how memory allocation works under the hood in high-level languages was deepened. The project highlighted the importance of efficient memory management in improving the performance of software applications.