Haleemah Amisu

Project Proposal: A Custom Memory Allocator

Overview:

The goal of this project is to implement a custom Memory Allocation System to manage dynamic memory in C, serving as an alternative to the standard malloc() function. This custom allocator, tentatively named malloy(), will explore more efficient memory management strategies to address the limitations of existing methods, such as fragmentation, performance bottlenecks in multithreaded environments, and limited memory tracking features. A detailed performance comparison with malloc() will highlight the potential improvements in speed, fragmentation handling, and overall efficiency.

Objectives:

Design a Custom Memory Allocator: Develop a custom memory allocator, malloy(), that can dynamically allocate and free memory using an optimized strategy (e.g., memory pooling, the buddy system, etc.).

Optimized Memory Management: Tackle traditional allocator issues like fragmentation and allocation speed by using techniques such as block merging, efficient memory reuse, and handling varying request sizes. This will also aim to enhance memory allocation efficiency in multithreaded environments.

Memory Debugging Features: Integrate features for debugging memory usage that are typically provided by external tools like Valgrind. These features will include memory leak detection, buffer overflow detection, and (optional) runtime inefficiency analysis.

Implementation Plan:

- **Design the Architecture for malloy():**

  Create the fundamental structure for memory blocks and free lists, which will serve as the foundation for the allocator. This structure will include metadata to track the size of allocated and free blocks, as well as pointers for memory management.

- **Implement Basic Allocation and Deallocation Features:**

  Develop the core functionality for memory allocation and deallocation, using a hybrid strategy combining the buddy system and best fit allocation. This approach aims to reduce fragmentation and improve the handling of various request sizes.

- **Set Up a Memory Management Structure:**

  Use data structures such as linked lists to manage free and allocated memory blocks. This will allow for efficient traversal, coalescing of adjacent free blocks, and tracking of allocated memory chunks.

- **Error Handling:**

  Implement robust error handling for common allocation issues, such as failed memory requests, invalid deallocations, and out-of-memory scenarios.

- **Fragmentation-Reducing Techniques:**

Focus on block merging and coalescing free blocks to reduce fragmentation over time. Additionally, integrate splitting large free blocks to accommodate small requests and prevent wasted space.

- **Comparison with malloc():**

  Conduct performance benchmarks comparing malloy() with the standard malloc() function, evaluating metrics such as allocation speed, fragmentation, and memory usage over time. These comparisons will help identify the strengths and limitations of the custom allocator.

Conclusion:

This project aims to create a custom memory allocator that not only competes with malloc() but also introduces advanced memory management techniques to reduce fragmentation, improve performance, and enhance debugging capabilities. By leveraging a hybrid allocation strategy and implementing fragmentation-reducing techniques, this allocator will offer a potential alternative for performance-critical applications.