Alexia Pappas
6013 A4 Malloc Assignment

In this project, I implemented a custom memory allocator - apMalloc - using mmap() and a linear probing hash table - LPHT - for tracking allocations. This write up analyzes the performance of apMalloc compared to the system's malloc() and explores potential optimizations.

When running the benchmark timings, they show that apMalloc is significantly slower than the standard malloc() - 0ms on average for the standard malloc() and 5 ms on averagefor my apMalloc. Some key reasons for this include:

- The system malloc() is highly optimized: The default memory allocator is implemented with techniques such as thread caching, making it much faster.
- The overhead of mmap(): the main bottleneck in apMalloc is its reliance on mmap(). Each allocation is rounded up to the nearest page (usually 4KB) and directly mapped using mmap(), which is costly in terms of system calls and memory fragmentation.
- Hash table lookup overhead: Every allocation and deallocation requires inserting into or searching through the hash table, adding an extra performance penalty compared to system allocators that use metadata stored inline.
- Lack of memory reuse: apMalloc does not reuse freed memory efficiently - apMalloc directly calls munmap() upon deallocation, meaning subsequent allocations require a new mmap() call, leading to inefficiency. In contrast, malloc() maintains free lists and bins to recycle allocations, reducing the number of system calls.
- Poor block handling: malloc() uses slab allocators and thread-local caches for small allocations, allowing quick memory retrieval without frequent system calls. In contrast, apMalloc treats all allocations the same, leading to excessive overhead.

Several optimizations could make my apMalloc more efficient.

- Use a Free List: instead of calling mmap() for every allocation, maintain a list of freed memory blocks that can be reused. This would drastically reduce the number of expensive mmap() and munmap() calls. This would also help to be able to merge adjacent free blocks in order to help prevent excessive fragmentation.
- Chunk allocations for small, medium, and large objects: implement a slab or bucked allocator that preallocates memory in chunks based on the size of the allocation.
- Reduce hash table overhead: store metadata inline with the allocated blocks instead of using an external hash table.
- Batch mmap() calls: instead of allocating one page per request, allocate large memory regions at once and subdivide them as needed. This would reduce the number of mmap() calls and allow faster small-block allocations without repeated system calls.
- Store metadata in allocated blocks: currently apMalloc stores allocation metadata in a separate hash table, which introduces lookup overhead. Instead, do something like malloc() does and embed the metadata within the allocated memory block.

My apMalloc allocator, while functional, is significantly slower than malloc() due to excessive mmap() calls, lack of memory reuse, hash table lookup overhead, and inefficient handling of small allocations. However, with optimizations such as free lists, batch allocations, inline metadata, and slab allocators, apMalloc could be improved to better compete with malloc(). While it is unlikely to match the performance of the system allocator, these changes would bring apMalloc closer in efficiency. This project has provided valuable insights into memory management and the trade-offs involved in designing a custom allocator.