BLG312E Computer Operating Systems Spring 2024 - Homework 2

Nurselen Akın
150200087

***Abstract*—Efficient memory management is crucial for modern operating systems to guarantee peak performance and best use of available resources. This project investigates the creation of a customized memory management system that uses the mmap system call to create user-defined versions of the malloc and free functions instead of the standard ones. The main objective was to develop a system that manages dynamic memory allocation in a single process context while staying under a predetermined heap limit. For the purpose of allocating memory for simulated "processes" with varying memory requirements, the system supports the following allocation strategies: Best Fit, Worst Fit, First Fit, and Next Fit. A memory region must be initialized, memory blocks must be allocated and released via header tracking, and memory must be managed by keeping a free list. The design choices, difficulties faced, and performance implications of each technique are covered in this report.**

I. Introduction

A key component of operating system design, efficient memory management impacts system stability and performance. The fundamental capability for dynamic memory allocation in user space is provided by traditional memory management utilities such as malloc and free. But these tools have costs and restrictions, especially when it comes to managing fragmentation and adjusting memory utilization based on different application needs. In order to investigate different ways and improve teaching about low-level memory operations, this project entails creating a bespoke memory management system that uses the mmap system call to bypass standard library functions.

II. Differences Between *'malloc'* and *'mmap'*

The malloc function, which is designed for relatively small to medium-sized allocations, allots memory from a process's heap. In order to reduce fragmentation, memory is managed using a free list that handles both allocation and deallocation. However, the main purpose of mmap is to map big memory blocks, such files or devices, into the address space of the process. Applications that require large, contiguous memory regions can benefit from mmap's more direct control over memory compared to malloc, as it does not automatically manage fragmentation or maintain a free list.

III. Custom Free Implementation

In this project I used a function called MyFree, which is made to operate with memory allocated by mmap, in place of the usual free. MyFree works directly with our unique data structures that track memory allocations and deallocations, in contrast to free, which interfaces with the memory allocator of the C standard library. In order to reduce fragmentation, this function not only identifies memory blocks as free but also groups together neighboring free blocks. MyFree's goal is to emulate the safe memory release and fragmentation reduction of free, but it is only as good as the proper and careful handling of the memory blocks in the free list. These merging and tracking techniques must be implemented robustly for MyFree to be successful in releasing memory as safely and effectively as possible.

IV. Implementation

A. InitMyMalloc

The InitMyMalloc function, which initializes the memory system, is where I started. It uses mmap to allocate a block of memory after receiving a HeapSize as input, making sure the size is more than the system page size. To set the limits of the heap and mark the first memory block free, a header is built.

B. MyMalloc

I respond to memory requests in the MyMalloc function by first adding a header size to the requested size in order to account for our metadata. I then figure out how to select a suitable block from our free list using a strategy parameter. I used four distinct strategies: First Fit, Worst Fit, Best Fit, and Next Fit. Each strategy uses a different technique to choose a suitable free block. I split the block if needed to maximize memory consumption if a suitable block is discovered.

C. MyFree

MyFree was developed to modify pointers and declare a previously allocated block as free in order to release it. Crucially, in order to preserve more expansive, continuous free regions in the heap, I additionally merge nearby free blocks to lessen fragmentation.

D. Supporting Functions

- In order to reduce internal fragmentation, **SplitBlock(mem_block* block, size_t size)** splits a block into two if the block size exceeds the desired size plus the size of a header.

- **MergeFreeBlocks()**: This function reduces fragmentation and makes larger blocks available for allocation in the future by combining neighboring free blocks into a single, larger block.

- **FindFit(size_t size, int strategy, mem_block* last_allocated)**: Using the given strategy, it searches the free list for a block that matches the desired size. This function is essential for putting various allocation strategies into practice and has a big impact on memory usage and performance.

V. Outputs Recieved

The functions works as expected and allocation strategies returns the expected pointer. Here are some output examples.

Code can be run with:

g++ malloc.c -o malloc
./malloc

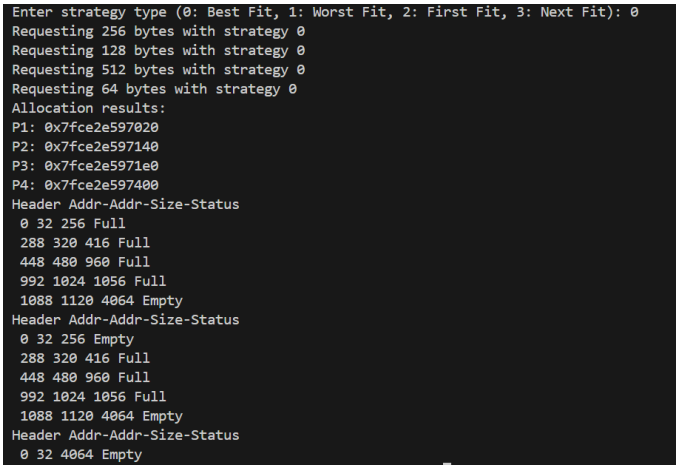

Fig. 1.

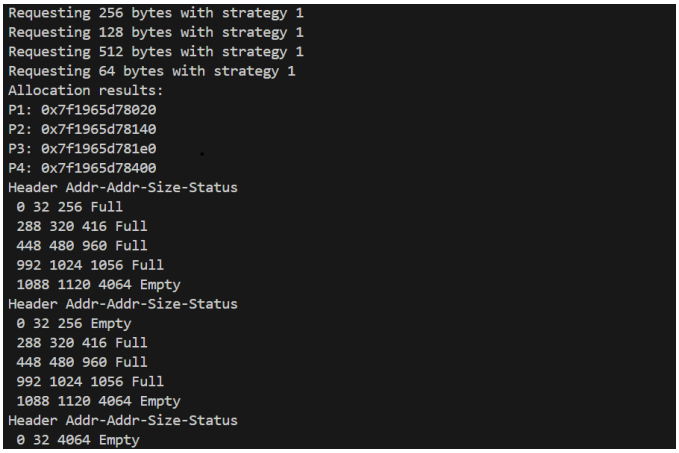

Fig. 2.