Mohan Guo 郭默函

Introduction

In the task1, I completed the basic realization of the requirements. I learned how to use free and vmstat tools to check the change of memory allocation in system. Through transferring and allocating two functions *“void *_malloc(size_t size)” and “void _free(void *ptr)”* defined by myself, I could implement a simple management of the memory successfully.

Implementation Overview and Design Decisions

Command-line argument handling: The program accepts a command-line argument specifying the amount of memory to allocate (in megabytes), allowing users to flexibly control the program's memory usage.

Memory allocation: Uses malloc to dynamically allocate memory of the specified size, malloc is a standard C library function that provides the flexibility to allocate memory of different sizes[1].

Memory initialization: Uses a value (‘A’) to fill the array representing the allocated memory to ensure actually the memory is used by the system. This could verify the memory allocation is successful.

Memory usage verification: Uses a while loop to access the allocated memory frequently to prevent the memory from being released. This helps to monitor the memory usage for a long term.

Error handling: Adds checks for invalid command-line arguments and memory allocation failures, providing detailed error messages. This improves the program's reliability.

Memory release: Despite the program's endless execution, it incorporates a call to free to guarantee proper memory deallocation at termination, so averting memory leaks.

Memory block management: Uses an array of blocks to manage memory blocks, and each block records its address, size, and availability status. This method may lead to memory fragmentation, so, memory merging is necessary in this case.

Memory splitting: In the _malloc function, if the identified memory block exceeds the required size, the surplus portion is partitioned into a new memory block, therefore preventing over-allocation and enhancing memory efficiency.

Memory merging: In the _free function, after releasing a memory block, adjacent available memory blocks are merged, reducing memory fragmentation and improving the efficiency of subsequent memory allocations.

Memory alignment: Ensures that the returned pointer is aligned for any variable type, achieved through the alignment mechanism of malloc and can be further optimized in _malloc.

Main design principles and code explanation

1) In the *memory_consumption.c*

Dynamic memory allocation: Use malloc to dynamically allocate memory of specified size.

Memory usage verification: Ensure memory is actually used by filling data and continuously traversing arrays.

Error handling: Check command line arguments and memory allocation success, and return error messages on failure.

CPU usage control: Use the sleep(1) function to control CPU usage and avoid overloading the system[2].

2) In the *memory_management.c* and *memory_management.h*

-init_mem_manager: Initializes the memory manager, allocates heap memory, and sets up an array of memory blocks, recording the starting and ending addresses of the heap memory.

- _malloc: Searches for an appropriate available memory block based on the requested size, marks it as used and returns the starting address. If the block is larger than needed, the excess part is split into a new available block. If the array is full or there is insufficient heap memory, it returns NULL.

- _free: Releases the specified memory block and marks it as available. If the pointer is invalid or the block has already been released, it returns an error message. And then, merge the adjacent available blocks to reduce memory fragmentation.

- print_memory_status: Prints information about the current memory blocks, including addresses, sizes, and availability. It displays the total number of memory blocks and available memory blocks. This helps to verifying the correctness of the memory manager.

Study Reflection

I learned the following points:

Memory allocation and release: I comprehended the use of malloc and free, as well as the management of memory allocation failure scenarios.

Memory usage verification: By filling data and continuously traversing arrays, I ensured that memory is actually being used.

System monitoring tools: I became familiar with the free and vmstat tools, which can monitor the conditions of memory usage in the system. And this is very useful for debugging and performance analysis.

Error handling: I added error handling mechanisms to the program which improved the safety of the program.

CPU usage control: I used the sleep function to control CPU usage which could prevent the program from imposing too great a load on the system.

This work deepended my understanding of the significance of memory management in operating systems. Meanwhile, I acquired proficiency in the fundamental principles of memory allocation and release and used arrays for memory block management, In addition, I try to implement the splitting and merging of memory blocks. These competencies are essential for myself to build a stable and efficient memory allocation system in the future.

References:

1. GeeksforGeeks. Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc(). Available online at:

https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/ Last Updated 11th Oct 2024

2. Aman Kumar. sleep() Function in C. Available online at: https://www.scaler.com/topics/sleep-function-c/ Last Updated 24th May 2024