Custom Memory Management System

Project Overview

This project implements a custom memory management system that mimics dynamic memory allocation (`malloc`), deallocation (`free`), and memory resizing (`realloc`) functions. The system uses a fixed-size heap and maintains a free list to track available memory blocks. It provides basic functionality for memory management, allowing allocation and deallocation of memory blocks, as well as reallocation to resize blocks.

Features

- **Custom malloc (`my_malloc`)**: Allocates memory blocks of a specified size.
- **Custom free (`my_free`)**: Frees allocated memory, making it available for future allocations.
- **Custom realloc (`my_realloc`)**: Resizes allocated memory blocks to a new size.
- **Free list management**: Keeps track of free memory blocks to efficiently allocate and deallocate memory.
- **Heap initialization**: Initializes a static heap of 1MB, with metadata to manage memory blocks.

Memory Management Structure

Heap Layout

The heap is simulated using a static array of fixed size (1MB), defined as `heap`. Memory blocks within the heap are managed using a linked list, where each node in the list represents a memory block. The metadata for each block is stored in the `Block` structure, which contains:

- **Size**: The size of the memory block (excluding the metadata).
- **Next**: Pointer to the next block in the list.
- **is_free**: Flag to indicate whether the block is free (`1`) or allocated (`0`).

Block Structure

Each memory block is represented by the following structure:

```
c
Copy code
typedef struct Block {
    size_t size;          // Size of the memory block
    struct Block* next;   // Pointer to the next block in the free list
    int is_free;          // Indicates whether the block is free (1) or allocated (0)
} Block;
```

The `Block` structure is stored in the heap alongside the memory that it manages. When a block is allocated, its metadata is stored just before the allocated memory.

Free List

The free list is a linked list of memory blocks, with each node in the list representing a block. The list is used to track available memory and allocate it efficiently. The free list is initialized at the beginning of the heap, and as memory is allocated and freed, the list is updated accordingly.

Heap Size

The total heap size is defined as `1024 * 1024` bytes (1MB):

```
c
Copy code
#define HEAP_SIZE 1024 * 1024
```

Function Descriptions

1. Heap Initialization

`void initialize_heap()`

This function initializes the heap by setting up the free list. It assigns the entire heap as a single free block. The block size is set to the heap size minus the size of the `Block` metadata.

```
c
Copy code
void initialize_heap() {
    free_list->size = HEAP_SIZE - sizeof(Block);
    free_list->is_free = 1;
    free_list->next = NULL;
}
```

2. Custom Malloc Implementation

`void* my_malloc(size_t size)`

This function allocates a memory block of at least `size` bytes. It searches the free list for a block that is free and large enough to satisfy the request. If a suitable block is found, the block is marked as allocated (i.e., `is_free` is set to `0`), and the pointer to the usable memory (just after the block's metadata) is returned. If no suitable block is found, `NULL` is returned.

```
c
Copy code
void* my_malloc(size_t size) {
    Block* current = free_list;
    while (current != NULL) {
        if (current->is_free && current->size >= size) {
            current->is_free = 0;
            return (void*)(current + 1);
        }
        current = current->next;
    }
    return NULL;  // No suitable block found
}
```

3. Custom Free Implementation

`void my_free(void* ptr)`

This function frees an allocated memory block. It sets the block's `is_free` flag to `1`, making the block available for future allocations. The function also ensures that the pointer passed to it is not `NULL`.

```
c
Copy code
void my_free(void* ptr) {
    if (!ptr) return;

    Block* block = (Block*)ptr - 1; // Get the block metadata
    block->is_free = 1;
```

```
}
```

4. Custom Realloc Implementation

`void* my_realloc(void* ptr, size_t size)`

This function resizes an allocated memory block. If the current block's size is already large enough to accommodate the new size, the function returns the same pointer. If the block is too small, a new block is allocated using `my_malloc`, and the contents of the old block are copied to the new one. The old block is then freed using `my_free`.

```
c
Copy code
void* my_realloc(void* ptr, size_t size) {
    if (!ptr) return my_malloc(size);

    Block* block = (Block*)ptr - 1;
    if (block->size >= size) return ptr;

    void* new_ptr = my_malloc(size);
    if (new_ptr) {
        memcpy(new_ptr, ptr, block->size);
        my_free(ptr);
    }
    return new_ptr;
}
```

5. Print Free List

`void print_free_list()`

This function prints the current state of the free list, displaying each block's memory address, size, and free status. It is used for debugging purposes to visualize the state of the heap and free list.

```
c
Copy code
void print_free_list() {
    Block* current = free_list;
    printf("Free List:\n");
    while (current != NULL) {
        printf("Block at %p: Size = %zu, Is Free = %d\n", (void*)current, current-
>size, current->is_free);
        current = current->next;
    }
}
```

Example Workflow

Here's an example of how the memory management functions are used in the `main` function:

1. The heap is initialized with `initialize_heap()`.
2. Two memory blocks of 100 and 200 bytes are allocated using `my_malloc`.
3. The free list is printed to show the current state.
4. The first block (`ptr1`) is resized using `my_realloc` to 150 bytes.
5. Both blocks are freed using `my_free`.
6. The free list is printed again to show the state after freeing.

```
c
Copy code
int main() {
```

```
    initialize_heap();

    void* ptr1 = my_malloc(100);
    void* ptr2 = my_malloc(200);

    print_free_list();

    ptr1 = my_realloc(ptr1, 150);
    my_free(ptr1);
    my_free(ptr2);

    print_free_list();

    return 0;
}
```

Limitations and Future Enhancements

1. **Fragmentation**: The current implementation does not handle memory fragmentation, which may occur after multiple allocations and deallocations.
2. **Coalescing Free Blocks**: The system could be improved by merging adjacent free blocks to reduce fragmentation and make larger blocks available for allocation.
3. **Error Handling**: More comprehensive error handling (e.g., handling memory exhaustion) could be added for robustness.
4. **Splitting Blocks**: When a block larger than the requested size is found, the system could split the block and return only the required portion, keeping the remainder in the free list.

Conclusion

This project provides a basic implementation of a custom memory management system with essential functions for dynamic memory allocation, deallocation, and resizing. It serves as a foundation for further enhancements, including better memory management strategies, coalescing of free blocks, and splitting of large blocks.