Operating Systems: Malloc

1 Introduction

In this lab, we'll work to uncover the inner workings of **malloc** through a series of experimental programs. You can find all the necessary code in the **malloc** directory on the operating-systems-course GitHub repository.

2 Background:

Before exploring how **malloc** works internally, let's first review the necessary APIs for this lab. Dynamic memory allocation in C enables programs to request memory during execution. Functions like `calloc`, `malloc`, and `realloc` are essential tools for efficient memory management.

2.1 calloc

The `calloc` (contiguous allocation) function allocates memory for an array of elements, initializes all bytes in the allocated memory to zero, and returns a pointer to the allocated memory.

```
void *calloc(size_t nmemb, size_t size);
```

Arguments:

- `nmemb`: The number of elements to allocate.
- `size`: The size in bytes of each element.

Return Value:

- If the allocation is successful, `calloc` returns a pointer to the allocated memory, with all bits initialized to zero. If it fails, it returns `NULL`.

2.2 malloc

The `malloc` (memory allocation) function allocates a specified amount of memory and returns a pointer to the allocated memory.

```
void *malloc(size_t size);
```

Arguments:

- `size`: The number of bytes to allocate.

Return Value:

- If the allocation is successful, `malloc` returns a pointer to the allocated memory. If it fails, it returns `NULL`.

2.3 realloc

The `realloc` function resizes previously allocated memory. It may move the memory block to a new location if necessary.

```
void *realloc(void *ptr, size_t new_size);
```

Arguments:

- `ptr`: Pointer to the previously allocated memory block. If `NULL`, it behaves like `malloc`.
- `new_size`: The new size in bytes for the memory block.

Return Value:

- If the allocation is successful, `realloc` returns a pointer to the allocated memory. If there isn't enough available memory to expand the block to the given size, the original block is left unchanged, and NULL is returned.

3 Programs

3.1 calloc.c

This program demonstrates dynamic memory allocation using `calloc`. It prompts the user to enter the number of elements for an array, then allocates memory for those elements dynamically using `calloc`. After checking if the memory allocation was successful, it displays the array elements, which are automatically initialized to zero by `calloc` (unlike `malloc`, which leaves memory uninitialized). Finally, it frees the allocated memory to prevent memory leaks.

3.1.1 Conclusion:

This lab shows how to use the `calloc` API for dynamic memory allocation with zero initialization. It demonstrates the key difference between `calloc` and `malloc`: `calloc` automatically initializes all allocated memory to zero, which is useful when you need a clean slate for your data structures without manual initialization.

3.2 malloc.c

This program demonstrates basic dynamic memory allocation using `malloc`. The program first asks the user to specify the number of elements for an integer array. It then allocates the required memory dynamically based on this input, ensuring proper memory allocation by multiplying the number of elements by the size of an integer. After validating that memory allocation was successful, the user is prompted to enter values for each array element. The program then displays all entered values and properly releases the memory using 'free' to prevent memory leaks.

3.2.1 Conclusion:

This lab demonstrates the fundamental memory allocation workflow using the `malloc` API: allocating memory dynamically, checking allocation success, utilizing the memory and properly freeing it when no longer needed. It shows how programs can request memory at runtime based on user input rather than having fixed, compile-time allocations.

3.3 realloc.c

This program demonstrates memory reallocation using `realloc`. It first allocates memory for an array of integers using `malloc` based on user input for the initial size. After populating the array with sequential values (1, 2, 3, ...) and displaying these values, the program asks the user for a new size. It then uses `realloc` to resize the array to the new dimensions. If the array is expanded, the program initializes the new elements to zero. Finally, the program displays the contents of the resized array and properly frees the allocated memory.

3.3.1 Conclusion:

This lab demonstrates using the `realloc` API to dynamically resize previously allocated memory. It shows that `realloc` preserves the existing data in the memory block when possible, which is important when growing or shrinking data structures at runtime. The program also illustrates proper memory management by checking for allocation failures and freeing memory when it's no longer needed.

4 Takeaways:

Experiment with different allocation sizes, quantities, and sequences to gain deeper insights into how `calloc`, `malloc`, and `realloc` function internally.