YTU CE

OPERATING SYSTEMS

ASSIGNMENT 2:

Manual memory allocation, reallocation and deallocation

STUDENT:

Görkem Şahin

15011087

DATE:

27.12.2017

Introduction

In this assignment we were asked to create our own **malloc**, **realloc** and **free** functions.

Since using the functions provided by the OS would defeat the whole purpose of this homework, we had to design a memory block structure to keep the allocated memory and info about the allocated memory within them, an expandable and shrinkable linked list to keep track of these blocks and a first-fit algorithm to allocate requested sizes of memory into suitable blocks. A mutex lock was also required to make sure that this program is thread-safe.

Implementation

Mutex Lock

First of all, memory allocations, reallocations and deallocations are supposed to be thread-safe and not interrupted. If that's not the case, a failed or corrupted memory allocation attempt can cause serious problems.

Easiest and quickest way of ensuring safety was implementing a Mutex Lock. I declared a global variable named **global_malloc_lock** of type **pthread_mutex_t**. Locking and unlocking it at beginning and end of memory allocations worked out pretty well for this program.

Block-Header Structure

Blocks and headers are needed for dynamic memory reallocation and deallocation operations. Portions of memory can be allocated without headers with **mm_malloc**, but without headers, we wouldn't be able to keep track of blocks to reallocate or deallocate them later on during run time.

Key points of having a header per block is to keep record of how much data had been allocated for that block, and to know whether that block is still being used or already freed previously. The variable **size** of type **size_t** states the size of the memory allocated for the related block. **is_free** variable of type **unsigned** lets us deduce the current state of the block. If it's a 0

then this block is already allocated. If it's a 1 then it can be used to provide memory for new variables.

Finally, In order to link all the blocks to each other and create a linked list of memory blocks, a **next** pointer should be present within the header section of the block. This allows us to traverse through the list.

void* mm_malloc(size_t)

First of all, if the parameter **size** equals to 0, there's no point of allocating 0 memory so function returns a NULL value.

If not, access to memory from other threads is locked using the **global_malloc_lock**.

Before increasing the size of the heap for this new allocation, all existing memory blocks are traversed. If a suitable, deallocated block that is also big enough is found, necessary settings are made on the header and pointer to this block is returned. If not, heap is increased using the **brk** call. The amount of increase should equal to the sum of the **size** requested by the user and the size of a header structure. Assuming the increase in heap didn't fail, this new block is placed on the linked list, mutex lock is unlocked and the pointer pointing to the block (one byte further from the header) is returned. If heap can not be incremented, a NULL value is returned.

void mm_free(void* block)

If the **block** variable sent to the function as an argument turns out to be already null, function terminates itself as there is nothing to do.

Else, this function will have to deallocate the block so it begins with locking the **global_malloc_lock**.

Next step is to figure out where the header of this block is. Going back one byte from where the data block begins is how we find the header. Then, using the **sbrk(0)** call, current ending point of the heap is acquired. Therefore we can compare it to the ending point of the block. If they're equals, program can simply shrink the heap by calling the **sbrk()** call after adjusting the linked list.

If this block that we want to deallocate is not the last one on the heap, we can't shrink the heap since heaps basically work like stacks. Instead, all that needs to be done is marking the header of the block as "free" by changing **is_free** to 1. From now on, this block will function as a deallocated block, available to be occupied by other variables next time **mm_malloc** is called.

Finally, **global_malloc_lock** is unlocked and function is terminated.

void* mm_realloc(void*, size_t)

In this function, first thing to do is checking the arguments. If the **block** is NULL or **size** equals 0, **mm_malloc(size)** is called.

Otherwise, we acquire the header of the block and see if the size of this block is bigger than or equal to requested **size**. If so, **block** is returned as it is, without editing anything.

If the requested **size** is bigger than the size of the block, we create a new block as big as the requested **size.** If this allocation is successful, content of the old block is copied into the new one using the **memcpy** function and the old block is freed by calling **mm_free(block).**

Now that we have a new block with the requested **size** and all the content of the old block, function returns the new block.

Analysis & Conclusion

```
gorkem@ubuntu:~/Desktop/odev$ ./mm_test

data: 354
malloc test successful!

start_addr      size    free    next
==============================
HEAD OF LL 93960010272768
93960010272768  4       0       93960010272796
93960010272796  1       0       93960010272821
93960010272821  4       0       0
gorkem@ubuntu:~/Desktop/odev$
```

This program I coded works as it is supposed to without any errors, although there are two warnings as shown above.

My biggest concern is that this algorithm called **first-fit** is not very efficient. Even when the requested memory to be allocated is one single byte, if the first available block this algorithm comes across is, let's say, a gigabyte, this block would be allocated to store a single byte, wasting almost a thousand megabytes. Embracing a more advanced strategy such as **buddy allocator** could have been better.

Thread safety is ensured by using a **mutex lock**, locking whole access to the memory before the allocation until the allocation phase is over, but implementing a protection that locks access to memory only when more than one threads are trying to allocate the same amount of memory would be an overall better solution.

mm_realloc() function creates a new block of memory and copies the content of the old one instead of extending it, which is less efficient.

I also considered using a list of free blocks for each allocation size (as suggested in the PDF) to prevent the iteration of the whole linked list when an allocation is needed to be done but I decided to stick with the classic list.

Overall, this memory allocator is quite reliable and works as it is supposed to, without any fatal errors, although it could be improved in a few ways in terms of efficiency.

Görkem Şahin

No: 15011087