Akshay Thejaswi
03/05/14

Project 4, Option B, Malloc

Files:

Makefile – makefile for this project
malloc.c – Implementation of malloc, free, calloc, realloc, and the heap-checker
malloc.h – header file for malloc.c, contains declarations and data structures
test_malloc.c – test program for project
test_malloc.out – output from test program

To Build:

All: make
malloc.o: make malloc
test_malloc: make test_malloc

Running:

test_malloc: ./test_malloc

Heap-checker usage: Calls to the heap-checker are made in the test program at critical locations. To use the heap checker, any program linked to malloc can call heapchk() any time during execution.

Write up

Data Structures

I defined my two main data structures in malloc.h as "block_t" and "free_block_t". Throughout my implementation, I use the types block and free_block, which represent pointers to those structures.

A **block** is defined as an entire allocated structure, including the header struct, and the data given to the calling entity. The block has a field for the size of the data, and the checksum.

A **free_block** is the structure that represents a region of free memory available for allocation. The struct has size, and a pointer to the next and previous free blocks.

I frequently convert between these two structures by taking the value of one and casting it to the other, and then treating that region as the new type during computations. For example, a free_block is converted into a block by casting the free block to a block, then writing the size and checksum over it.

The free list is a linked list made of free_blocks that starts at malloc_head. For convenience, I made a macro called for_each_free(fb), that iterated through the list in a manner inspired by the linked list macros found in the kernel.

Allocation algorithm:

Malloc() is called with an argument size_t size. First, the 8-byte aligned size is computed using a simple macro defined in malloc.h, and the result is in size8. The total size that malloc need for that "block" (the allocated region) is calculated by adding size8 and the size of the block header struct.

If malloc_head is null, the heap is extended by alloc_heap , which uses sbrk(). Alloc_heap actually allocated twice the amount requested, and put the extra free memory on the free list pointed to by malloc_heap.

If the free list is not empty, the free list is iterated through to find a block large enough to use. If a larger block is found, malloc “carves” out a portion for the block, by reducing the size of the free_block by alloc_size and using the new space as a block. Malloc can also use free_block's that are the exact size needed, it just converts the free_block into a block.

If, after searching through the free list, no usable free region was found, malloc extends the heap further using alloc_heap.

Free algorithm:

When a pointer is passed to free, it get the block struct for it by backtracking by H_BLOCK and casting to a block struct. The block is validated by calling validate_checksum. If the block is in fact a malloc'd pointer, it is converted into a free block. To do this, the block pointer is casted to a free_block, and the size is set. It is then added to the free list by calling add_to_free.

Calloc & Realloc:

Calloc is a very simple implementation. It calls malloc() to get the allocation, and then calls bzero() to zero out the data portion of the block.

Realloc calculates the change in size, inc, but subtracting the block->size from the new size. If the size is decreasing by enough margin to create a free block (at least H_FREE, or 12 bytes), a free block is “carved” out of the end of the data block and added to the free list. The size field of the block is adjusted and the same pointer is returned.

Otherwise, realloc simply calls malloc with the new size, and uses bcopy to copy as much data as possible from the old block to the new one. The old block is then freed.

Checksum:

The checksum for this implementation is handled by the functions set_checksum and validate_checksum. The checksum algorithm is just the distance between the malloc function pointer and the block in bytes.

Heap-checker

My heap-checker is actually a utility function in malloc.c, any program using malloc can use the heap checker by calling heapchk(). Heapchk prints out formatted information about the heap along with verifying its integrity. It iterated through the heap by starting at the beginning and using the size fields of the blocks and free_blocks to jump forwards. If it ends up jumping to a block that is not a block or a free block, it recognizes it as “unmanaged” and prints out information about it. It iterated forward until the next valid block is found.

Heapchk also calls freechk, which makes sure that the free list is valid. It does this by iterating through it in both directions to make sure it is properly linked.

These functions print out color-coded debug info as well. When heapchk is called, it prints out the block in their memory order, so it is convenient for visualizing the heap. Freechk prints out a list of free_blocks and their information as well.

Test case and heap checker output

The output for the test program is in test_malloc.out. The test program test malloc, free, realloc and calloc and uses the heap-checker repeatedly to describe the output.

Malloc and free: This test makes several alocations using malloc, and frees them after. The heap-checker is run between function calls to show the heap state at each interval. At the end, a call to malloc

is made and the returned block is overwritten, causing a “memory leak”. The next heap checker outputs show that the heapchecker recognizes this region as unmanaged and continues to the rest of the heap.The second to last output shows that bocks are arranged free-block-free. After, the data in the block in the middle is free, and the next heap-checker output show that those blocks have been merged into one.

Calloc: The calloc test allocates and array, writes numbers to it, and frees it. Then, it allocates ,using calloc, another array of the same size, so that the free block from the first one is converted for this. This can be verified since the two pointers have the same address. The test also verifies that the returned memory is completley zeroed out because of calloc.

Realloc: The re-alloc test allocates an array, and uses re-alloc to increase the size of the array, which allocates a new area. Then it reallocates the array to be smaller, so that re-alloc uses the same block. You can see in the test output that the location of the array remains the same, but the size is reduced, while the free block after it grows as a result of freeing the extra space.

Extra credit
As mentioned in the test output section, the free() function correctly coalesces blocks adjacent to each other, and the realloc function re-uses the same block if possible.

Thanks,

Akshay