CS 214 Assignment 6: Error-detecting malloc and free

Kevin J. Sung and Joyce Wang

December 8, 2013

1 Implementation

For our implementations of malloc and free, we followed very closely the example given in Chapter 8, Section 7 of *The C Programming Language* by Kernighan and Ritchie. Our header structure is aligned properly, assuming that type **long** is the most restrictive type. The header contains the size of its block and a pointer to the next free block if it is free itself. We give out memory from a large static Header array.

Before any memory has been allocated, the array is a single free block that points to itself. When memory is allocated, the free blocks point to each other in ascending order of address, and the last free block in the array points to the first one. For technical reasons, there must always be at least one free block, so the maximum amount of memory that can be allocated is the entire array, minus one header unit. When a program calls malloc, the function scans the free list, starting at the point where the last block was allocated. If a big-enough block is found, then it is returned if it is exactly the right size. If it is bigger than requested, the tail end is returned so that only the block size in the header needs to be updated. If the entire free list is traversed and no big-enough block is found, an error is returned.

When a program calls free on a block, the function scans the free list to find the proper place in the free list to insert the block so that the order of the free list is maintained. If the block is adjacent to a free block on either side, they are merged together.

To be able to perform error-checking, we record the pointers that have been allocated by malloc in an array. Every time a block is allocated, we add the pointer to an empty spot in the array. When a program attemps to free a block, we check the array to see if the pointer is there. If it is not, the free is invalid and an error is returned. If the pointer is in the array, then the block is freed and the position of the pointer in the array is set to NULL.

2 Running time and memory usage

Since our functions allocate memory from a fixed-size array, the memory usage is a constant; when this array is filled, malloc can no longer satisfy requests for more memory. The only factor of the running time that is variable is the time it takes to traverse the free list. This time is linear in the number of free blocks. Hence, the running time of malloc or free is linear in the number of free blocks in the array at the time of the call.