Malloc

Mariam Tsilosani

David Awad

Design

The Malloc program was built to imitate the behavior of Malloc and how it distributes memory, except instead of using memory from the heap it uses a character array allocated on the stack.

It is designed with the hope of reducing fragmentation as much as possible. One solution to the fragmentation is to use three separate blocks* and allocate memory from them based on the sizes of the memory being asked for when the `malloc()` function is called. This will reduce fragmentation by keeping the smaller and larger chunks of memory in different arrays.

Implementation

The Malloc uses a linked list of memEntry structs to manage how much memory is allocated and how much we have remaining etc.

Below are the Struct definitions of the Linked List and Array Sizes.

```
/* malloc.h */
#define bigBLOCKSIZE 3000
#define medBLOCKSIZE 1500
#define smallBLOCKSIZE 500
//MemEntry Struct
typedef struct memEntry{
    int pattern;
    struct memEntry *prev;
    struct memEntry *next;
    char * file;
    int  line;
    int isfree;
    int size;
}memEntry;
```

Extra Credit We Implemented

Malloc has a lot of special features to improve it's usability and flexiblity for different edge cases.

1. Leak detection and reporting. We used atexit(function pointer) to go through each array and see if there are any pointers left that have not been fread, if said pointer exists we print out an error message and the LINE and FILE the pointer was allocated at.
2. calloc(size_t nmemb , size_t size) - Our `Calloc()` allocates new memory and zeroes it out before returning it back to the user.
3. realloc(void *ptr, size_t size) - our `Realloc()` If the new size is smaller than pointer size it shrinks

the pointer size and gives the left over size back to the array. If the new size is bigger that the pointer size it checks if the memory around the recieved pointer is free, if it is then it combines that memory with the memory pointer was pointing to, else, it frees the pointer and goes to find more memory in the arrays.
4. Corruption detection - We use a hex number that every memEntry struct has at the begining, everytime we recieve a pointer back we check if the hex number is stil there, if its not then it means it has been overwritten, corruption has been detected, the program exits.
5. Freeing non Heap Variables with `free(void *ptr)` will give an error message is the variable eeing freed has not been allocated on the heap.
6. free()ing some pointer that was not returned by previous malloc()/realloc() call. A little harder. Somebody has done some bogus pointer arithmetic and tries to return an altered pointer to somewhere in the heap.
7. Fragmentation reduction - We have three separate arrays as a means to reduce the amount of fragmentation because we can keep similarly sized blocks together.

Usage

The API for the malloc family of functions is below.

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Malloc comes with a very extensive Makefile that allows you to test individual functions in Main.c by using Conditional Compilation. We can test each specific function using `make <option>`. Where the options are the following.

**1. ** `make` will compile main.c and create an object file called my_malloc that has linked together malloc.c and malloc.h

**2. ** `make run` will compile and run main without debugging flags or any additional debugging information. This means that main can be edited with additional testcases if desired and can then be very quickly observed by using the command again.

**3. ** `make debug` will compile and run main.c with all debugging flags enables that will test all functions extensively.

4. other `<options>` include items such as `make call` which will compile main with only the test cases for `calloc()` and run them. The other options are `reall`, `leak`,and `corrupt`.

***The block sizes are determined based on the assignment specification telling us we have a maximum of 5000 bytes.**