Chris Jakins

1000802309

4/12/18

Malloc

Problem Statement:

With any running program, it is not always known beforehand how much memory will be need at execution time. Sometimes, memory is needed longer than what can be provided through the stack. Static allocation, therefore, cannot provide the flexibility of memory that most programs require. Through dynamic allocation, the developer has increased flexibility of how memory is managed throughout the lifetime of an application.

As a downside, memory also has to be more explicitly managed. This can create pressure on the developer, since bugs that occur as a product of dynamic allocation errors can be costly and hard to track down. Therefore, we must create a method that can provide all the needs of the user while also having the least amount of risk.

It is our goal to provide a system that is safe of data leakage from previous users, allows repeatable and constant time allocations, and where violations of the API are detected immediately.

Schedule:

|  |  |  |
| --- | --- | --- |
| **Tasks** | **Hours Required (est.)** | **Hours Expended** |
| Initialize project with user-level requirements scheme | 1 | 1 |
| Develop user-level requirements into a more formalized list of technical requirements | 2 | 1 |
| Begin process of designing the malloc system | 1.5 | 4 |
| Ensure that the malloc system design meets the requirements specified | 1 | 1 |
| Begin implementation and development of the malloc system | 3 | 3 |
| Validate the malloc system through testing | 2 | 2 |
| Verify that the malloc system implementation meets the requirements | 1 | 2.5 |

Requirements Analysis:

1. The User shall provide to malloc() how much memory they would like, specified in the number of bytes. (F)
2. The System shall allocate the number of bytes specified by the User off of the heap. (F)
3. The System shall record the location of the allocated memory in the Memory Management structure. (F)
4. The User shall receive a pointer to the beginning of the allocated memory specified in Req3. (F)
5. The System shall not initialize the memory allocated to the User. (NF)
6. The total size of memory to be managed is 0x0100000. (NF)
7. The System shall return NULL if the number of bytes specified by the User is 0. (NF)
8. The System shall determine the location for the allocated memory using the Worst Fit algorithm for memory management given by : https://www.geeksforgeeks.org/program-worst-fit-algorithm-memory-management/ (NF).
9. The System shall make use of the sbrk() system call to get the current top of the heap. (F)
10. The System shall make use of the sbrk() system call to update the heap pointer by the number of bytes specified by the User. (F)
11. The System shall return -1 if there was an error in allocating the memory. (F)
12. The System shall maintain, in the Memory Management structure, what blocks of memory are currently allocated to which User. (NF)
13. The System shall only accommodate one Users request for memory at a time. (NF)
14. The System shall detect if the number of bytes specified is an invalid number, or is not a number at all, at the start of execution. (NF)
15. The System shall be implemented in the C language. (NF)
16. The User shall provide to free() a pointer that they would like to deallocate. (F)
17. The System shall determine how much memory is allocated to the block specified by the pointer passed through free(), and deallocate the entire block accordingly. (F)
18. The System shall remove the entry of the deallocated memory in the Memory Management structure. (F)
19. The System shall return a success (1) or error (0) code to the User depending on the state of the deallocation. (NF)
20. The System shall only accommodate one Users request to free memory at a time. (NF)

Design:

In designing a malloc implementation, the first discussion needs to be what data structures should be involved. After some thought, I feel that the best data structure to use would be a linked list. This would easily keep track of the currently used/free blocks, and it provides linear time operations. Pointers can be used to keep track of the heads of blocks. When memory is requested, the blocks used are kept track of on this linked list and can be marked free when not needed any longer.

Alternatives:

A good alternative to the linked list is a bitmap where each bit represents a block of memory. Obviously, this would be very large and costly to represent, especially if there is a small amount of heap memory needed. However, this data structure choice is generally more time efficient, since lookups for freed/used blocks are simple bit manipulations through masking or indexing, etc.

Another possible implementation would be through the use of a hash/lookup table. Addresses could be used as the keys to hash, and the value they map to would include necessary information such as free/allocated and how many blocks. This is obviously very time efficient due to the nature of hash tables, but requires a lot of space overhead. Due to our constraints, I believe this to not be the best choice.

Constraints

1. External Fragmentation is the various spaced holes that are generated in our memory. This creates a problem when the user tries to request memory that we do not have contiguously available. Therefore, our design needs to either compact allocated memory every once in a while, or take care of this in some other way.
2. Since we only have a limited amount of memory on the system and are trying to save a large amount of that for the user, our design must be lightweight enough to not take up too much of this memory.
3. The design must keep security in mind, and make sure that any user can only access their data. Therefore, the design must make sure to clean up after a user is finished with the system.

Source Code

#include <stdlib.h>

#include <string.h> // for memset

char heap[10000000];

typedef struct list {

int size;

int free;

struct list \* next;

} List;

List \* freeBlocks = (void \*) heap;

// Utility methods

void init() {

freeBlocks->size = 10000000 - sizeof(List);

freeBlocks->free = 1;

freeBlocks->next = NULL;

}

void split(List \* block, int size) {

List \* new = block + size + sizeof(List);

new->size = block->size - size - sizeof(List);

new->free = 1;

new->next = block->next;

block->size = size;

block->free = 0;

block->next = new;

}

void merge() {

List \* curr, \* prev;

curr = freeBlocks;

while (curr && curr->next) {

if (curr->free && curr->next->free) {

curr->size += curr->next->size + sizeof(List);

curr->next = curr->next->next;

}

prev = curr;

curr = curr->next;

}

}

// Required methods

void \* \_malloc(int numBytes) {

List \* curr, \* prev;

void \* blockGiven;

if (!(freeBlocks->size)) {

init();

}

curr = freeBlocks;

while ((curr->size < numBytes || curr->free == 0) && curr->next) {

prev = curr;

curr = curr->next;

}

if (curr->size == numBytes) {

curr->free = 0;

blockGiven = (void \*) ++curr;

return blockGiven;

} else if (curr->size > numBytes + sizeof(List)) {

split(curr, numBytes);

blockGiven = (void \*) ++curr;

return blockGiven;

} else {

blockGiven = NULL;

return blockGiven;

}

}

void \_free(void \* ptr) {

if ((void \*) heap <= ptr && ptr <= (void \* ) (heap + 10000000)) {

List \* curr = ptr;

curr--;

curr->free = 1;

merge();

}

}

// same as malloc

// but with memsets

void \* \_calloc(size\_t nmemb, int size) {

List \* curr, \* prev;

void \* result;

if (!(freeBlocks->size)) {

init();

}

curr = freeBlocks;

while ((curr->size < size || curr->free == 0) && curr->next) {

prev = curr;

curr = curr->next;

}

if (curr->size == size) {

result = (void \*) ++curr;

memset(result, 0, size);

return result;

} else if (curr->size > size + sizeof(List)) {

split(curr, size);

result = (void \*) ++curr;

memset(result, 0, size);

return result;

} else {

result = NULL;

return result;

}

}

void \* realloc(void \* ptr, size\_t size) {

List \* curr, \* prev;

void \* result;

if (!(freeBlocks->size)) {

init();

}

curr = freeBlocks;

while ((curr->size < size || curr->free == 0) && curr->next) {

prev = curr;

curr = curr->next;

}

curr->size = size;

result = (void \*) ++curr;

return result;

}

Analysis:

All data averaged over 5 calls

Malloc Avg: .114s

Free Avg: .101s

Calloc Avg: .143s

Realloc Avg: .121s

In conclusion, the implementation presented runs in a consistent amount of time, and it also runs quickly. There are no anomalies in the data.