Comp 304 Project 3 - Custom Memory Allocator

14 January 2024

Team:
Kemal Barış Kesen
Mehmet Eren Kılıç

Assignment Description: This project provides a custom implementation of a memory allocator in C. It includes functions like the standard malloc, calloc, realloc, and free.

Parts That Work:

- **Valid Submission: 1/1** Submissions are valid.
- **Malloc: 4/4** All malloc parts work correctly.
- **Calloc: 1/1** Calloc works properly.
- **Free: 4/4** All free parts work properly.
- **Realloc: 3/3** Realloc works properly.
- **Batching: 1/1** Batch part also work properly.
- **Fragmentation: 0/2** We could not manage to add fragmentation. Our overhead is way higher than 17.
- **Locality: 0/1** We could not add locality.
- **Unmap: 1/1** Implementation gives back to the memory.
- **Alternative Design: 0/1** We could not use in-band metadata.
- **System Malloc: 3/4** malloc, preload ls, preload python works fine but not grep.
- **Dynamic Heap Size: 1/2** 128K heap works but not 256M heap.
- **Compiler Warnings: No Warning**
  Total: 8/13 points

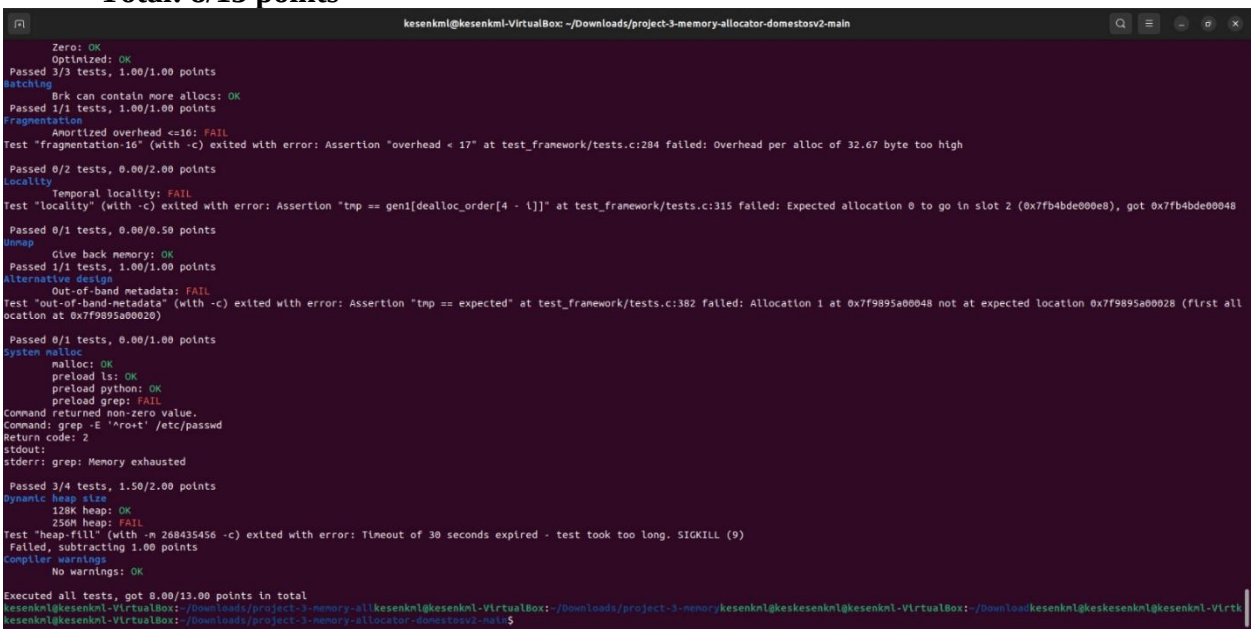
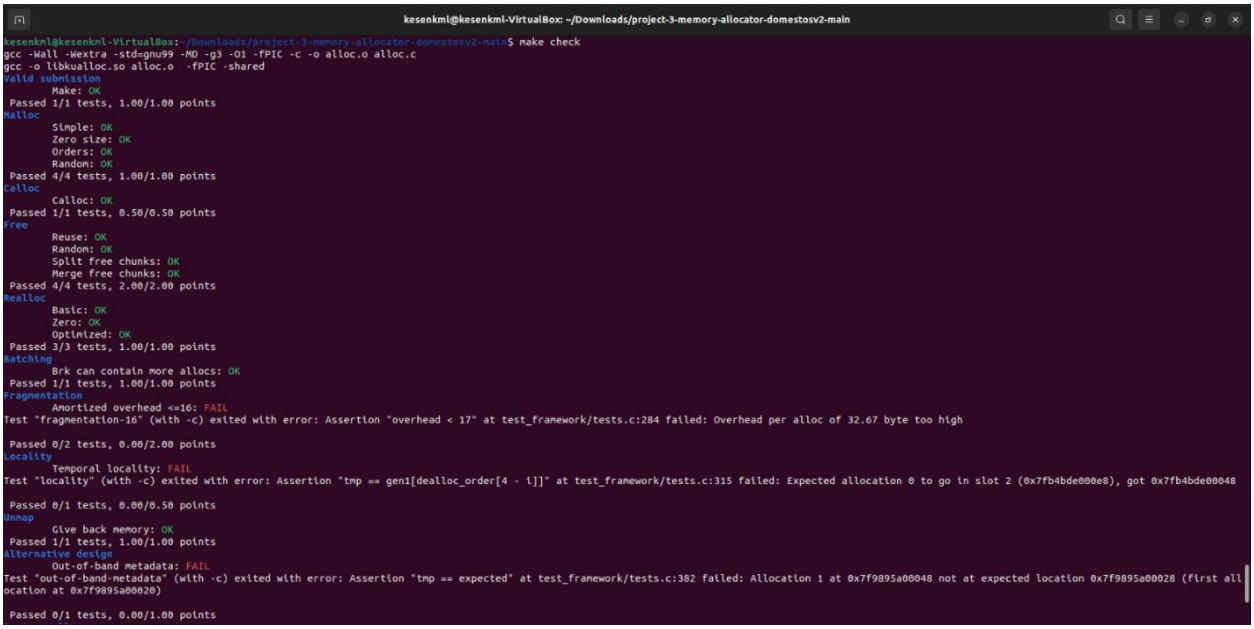


Figure 1: Output



```
kesenkml@kesenkml-VirtualBox: ~/Downloads/project-3-memory-allocator-domestosv2-main
kesenkml@kesenkml-VirtualBox:~/Downloads/project-3-memory-allocator-domestosv2-main$ make check
gcc -Wall -Wextra -std=gnu99 -MD -g3 -O1 -fPIC -c -o alloc.o alloc.c
gcc -o libkualloc.so alloc.o  -fPIC -shared
Valid submission
        Make: OK
 Passed 1/1 tests, 1.00/1.00 points
Malloc
        Simple: OK
        Zero size: OK
        Orders: OK
        Random: OK
 Passed 4/4 tests, 1.00/1.00 points
Calloc
        Calloc: OK
 Passed 1/1 tests, 0.50/0.50 points
Free
        Reuse: OK
        Random: OK
        Split free chunks: OK
        Merge free chunks: OK
 Passed 4/4 tests, 2.00/2.00 points
Realloc
        Basic: OK
        Zero: OK
        Optimized: OK
 Passed 3/3 tests, 1.00/1.00 points
Batching
        Brk can contain more allocs: OK
 Passed 1/1 tests, 1.00/1.00 points
Fragmentation
        Amortized overhead <=16: FAIL
Test "fragmentation-16" (with -c) exited with error: Assertion "overhead < 17" at test_framework/tests.c:284 failed: Overhead per alloc of 32.67 byte too high

 Passed 0/2 tests, 0.00/2.00 points
Locality
        Temporal locality: FAIL
Test "locality" (with -c) exited with error: Assertion "tmp == gen1[dealloc_order[4 - i]]" at test_framework/tests.c:315 failed: Expected allocation 0 to go in slot 2 (0x7fb4bde000e8), got 0x7fb4bde00048

 Passed 0/1 tests, 0.00/0.50 points
Unmap
        Give back memory: OK
 Passed 1/1 tests, 1.00/1.00 points
Alternative design
        Out-of-band metadata: FAIL
Test "out-of-band-metadata" (with -c) exited with error: Assertion "tmp == expected" at test_framework/tests.c:382 failed: Allocation 1 at 0x7f9895a00048 not at expected location 0x7f9895a00028 (first all
ocation at 0x7f9895a00020)

 Passed 0/1 tests, 0.00/1.00 points
```

Figure 2: Output

Helper Functions:
alignSize(size_t size): Returns aligned size.
splitBlock(MemoryBlock *block, size_t size): Splits the large memory block into two smaller blocks.
findBestFitFree(MemoryBlock **last, size_t size): Checks free blocks and returns best-fit block.
extendHeap(MemoryBlock *last, size_t size): When no suitable memory block found, this function increases program's data segment.
mergeBlocks(MemoryBlock *block): Merges adjacent memory blocks to create larger block.
zeroization(void *inputblock, size_t inputSize): Sets memory blocks to zero to be used in calloc.
getBlockAddress(void *ptr): Returns the block address from the pointer.

Implementation:
kumalloc: First aligns the requested size using alignSize function. If it is the first allocation using extendHeap, extends the heap with given size. This block becomes the head of the linked list. If it is not the first allocation, the function uses findBestFitFree method to find suitable block. If found, splits the block if necessary. Thus, allocates memory of the given size.

```c
void *kumalloc(size_t size)
{
    if (size <= 0)
    {
        return NULL;
    }

    size_t correctedSize = alignSize(size);
    MemoryBlock *block;

    if (!head)
    {

        block = extendHeap(NULL, correctedSize);
        if (!block)
        {
            return NULL;
        }
        head = block;
    }
    else
    {

        MemoryBlock *last = head;
        block = findBestFitFree(&last, correctedSize);

        if (!block)
        {

            block = extendHeap(last, correctedSize);
            if (!block)
            {
                return NULL;
            }
        }
        else
        {

            block->isFree = 0;

            if (block->size > correctedSize + SIZEOFBLOCK)
            {
                splitBlock(block, correctedSize);
            }
        }
    }
    return (void *)(block + 1);
}
```

Figure 3: kumalloc implementation

kucalloc: Takes number of elements and element sizes and multiply them to find the find total size. After allocating total memory from the kumalloc, uses zeroization to create these elements from that total sized block.

```c
void *kucalloc(size_t numElements, size_t elementSize)
{
    if (numElements == 0 || elementSize == 0)
    {
        return NULL;
    }

    size_t totalSize = numElements * elementSize;
    size_t correctedSize = alignSize(totalSize);
    void *newBlock = kumalloc(correctedSize);

    if (newBlock)
    {
        zeroization(newBlock, correctedSize);
    }

    return newBlock;
}
```

Figure 4: kucalloc implementation

kurealloc: Takes the pointer and new size and if new size is sufficient, returns the pointer. If the current block smaller than the requested size using kumalloc, the function asks for a new sufficient block. Then copies from the old data to the new block and frees the old block using kufree.

```c
void *kurealloc(void *ptr, size_t size)
{
    if (size == 0)
    {
        kufree(ptr);
        return NULL;
    }

    if (!ptr)
    {
        return kumalloc(size);
    }

    MemoryBlock *block = (MemoryBlock *)ptr - 1;
    if (block->size >= size)
    {
        return ptr;
    }

    void *newPtr = kumalloc(size);
    if (newPtr)
    {
        memcpy(newPtr, ptr, block->size);
        kufree(ptr);
    }

    return newPtr;
}
```

Figure 5: kurealloc implementation

kufree: Takes the pointer of the data block to be freed. It gets the data block by getBlockAddress function, then sets its free value to 1. Then uses mergeBlocks to merge it if it applicable. If freed block is the last block in the memory pool, it uses brk to reduce the heap size.

```c
void kufree(void *ptr)
{
    if (!ptr)
    {
        return;
    }

    MemoryBlock *block = getBlockAddress(ptr);
    block->isFree = 1;
    block = mergeBlocks(block);

    if (block->next == NULL && block->isFree)
    {

        if (block->prev)
        {
            block->prev->next = NULL;
        }
        else
        {
            head = NULL;
        }

        if (brk(block) == -1)
        {

            fprintf(stderr, "Reduce Error!!!\n");
        }
    }
}
```

Figure 6: kufree implementation