CS 744 Autumn 2020
Programming Assignment 3
Memory allocation and management

Introduction

You are familiar with dynamic memory allocation and management via calls to malloc, free and realloc. In this assignment, you will be implementing your own dynamic memory allocator. By this we mean that you will implement the three functions of allocation, deallocation and reallocation that our test drivers will use to manage dynamic memory. Rather than having to start from scratch, we have provided a starting point of a naive implementation of these three functions that you will need to optimise. Recall that the basic tradeoff in managing heap memory is latency/throughput of handing requests Vs the memory utilization efficiency(defined as the ratio of memory allocated to size of the heap being managed). Your implementation will therefore be evaluated based on correctness, throughput and utilisation efficiency.

The handout contains a naive implementation of malloc, free and realloc. You have to optimize the given code to increase its throughput and memory utilization. Remember, the design points are those we discussed in class such as which free block to use (first fit, best fit, worst fit etc.), how to know how much memory to free, whether to coalesce adjacent free blocks, how to traverse free blocks (explicit vs implicit free lists) and so on. You have to think about these design choices and make the ones you believe will fit the widest variety of workloads. By workload we mean the pattern of requests of malloc, free and realloc that our test driver will make to your library. An example is malloc will be called in increasing sizes of heap to be allocated up to some point and then a series of free calls are made and then further malloc calls. Think of what workloads can look like while you design your allocator.

The assignment

We have provided starter code as a handout including a driver to test your memory management implementation. The naive implementation is provided in **mm.c** Only make changes to **`mm1.c` and `mm2.c`** (or to a copy of it). ***Do not change any other files.***

`mem_sbrk(size_t size:` This is a wrapper function for the sbrk() system call.
(Use mem_sbrk() instead of sbrk() otherwise the mdriver.c will give wrong evaluation results**)**

All three files mm.c, mm1.c and mm2.c share the same header file mm.h
The code to change will consist of the following four functions, which are declared in **mm.h** and defined in **mm1.c and mm2.c**

```
int mm_init(void);
void* mm_malloc(size_t size);
void mm_free(void*ptr);
void* mm_realloc(void*ptr, size_t size);
```

The driver code will call these instead of the corresponding equivalents in libc.

Do not change the interface of the above four functions. You are to change their implementation according to your design.
The following is needed as part of the assignment,

1. The implementation of a memory manager via the four functions mentioned. (this is already provided as an example in mm.c, no need to add any implementation). This implementation will serve as one of the base cases of comparison.

2. Implement a best-fit + two-sided coalescing memory manager in **mm1.c**. This will need boundary tags of metadata at both ends of each block.

3. Design another memory management solution that you think is better than #2. Argue for it in your reporting. Implement the design. Implementation of this should be submitted as **mm2.c. Think about using some form of balanced tree as a data structure to hold the metadata.**

4. Comparison of the 4 schemes --- glibc's malloc implementation, the naive implementation and the two designs that you will implement using the given driver and test workloads. You can design your own workloads as well for testing purposes.

The descriptions of the four functions that will have to be implemented for custom memory management are as follows:

`mm_init` : Before calling mm_malloc, mm_realloc, or mm_free, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls mm_init to perform any necessary initialization, such as resetting the heap, setting all the data structures and variables to their starting values. The return value should be -1 if there was a problem in performing the initialization, and 0 otherwise. The mdriver.c will call mm_init before running each trace input. The mm_init function should include reinitialization of all required state (as part of the memory manager implementation) each time it is called for every trace file by the driver program (mdriver.c).

`mm_malloc` : The mm_malloc routine returns a pointer to an allocated block with a payload of at-least size bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk. We will compare your implementation to the version of malloc supplied in the standard C library (libc). Since the libc malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers.

The sample code provided in mm.c uses mem_sbrk() for each call to mm_malloc().

`mm_free` : The mm_free routine frees the block pointed to by ptr . It returns nothing. This routine is only guaranteed to work when the passed pointer (ptr) was returned by an earlier call to mm_malloc or mm_realloc and has not yet been freed. The freeing of memory will also intersect with other memory management actions which manage information related to free lists, allocated lists etc.

`mm_realloc` : The mm_realloc routine returns a pointer to an allocated block with a payload of at least size bytes with the following constraints.

- if ptr is NULL, the effect of the call is equivalent to mm_malloc(size);
- if size is equal to zero, the effect of the call is equivalent to mm_free(ptr) and the return value is NULL;
- if ptr is not NULL, it must have been returned by an earlier call to mm_malloc or mm_realloc .

The call to mm_realloc changes the size of the memory block pointed to by ptr (the old block) to provide a payload of size bytes and returns the address of the new block. The address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the realloc request.

The contents of the new block are the same as those of the old ptr block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block.

Other functions can be added in mm.c according to your needs

Trace file Description :
A trace file is an ASCII file. It begins with a 4-line header:

```
<sugg_heapsize>        /* suggested heap size (unused) */
<num_ids>              /* number of request id's */
<num_ops>              /* number of requests (operations) */
<weight>               /* weight for this trace (unused)
```

The header is followed by num_ops text lines. Each line denotes either an allocate [a], reallocate [r], or free [f] request. The <alloc_id> is an integer that uniquely identifies an allocate or reallocate request.

```
a <id> <bytes>         /* ptr_<id> = malloc(<bytes>) */
r <id> <bytes>         /* realloc(ptr_<id>, <bytes>) */
f <id>                 /* free(ptr_<id>) */
```

For example, the following trace file:

```
<beginning of file>
20000
3
8
1
a 0 512
a 1 128
r 0 640
a 2 128
f 1
r 0 768
f 0
f 2
<end of file>
```

Your code will be tested as follows:

- Correctness by checking that block must be aligned properly, and must not overlap any currently allocated block.
- Space utilisation by executing with different trace files which will check that you are reallocating previously allocated block, whether performing coalescing etc. It is a ratio calculated as **(optimal_heap_size / heap_size)** where:

    - **optimal_heap_size** = heap size that will be used without any holes or any overheads for book-keeping.
    - **heap_size** = the heap size used by your mm_malloc
- Throughput by calculating the number of operations per second. It is represented in **Kops** (Kilo operations per second).
- The traces are designed to test your malloc under high memory load and to check whether coalescing is happening properly
- If the max_heap_size is exceeded by your mm_malloc then an error "`ERROR: mem_sbrk failed. Ran out of memory...`" will be printed on the terminal and the testing of the corresponding trace will terminate without calculating the throughput and space utilization. The next trace will be used for subsequent testing.

You should be able explain all of the submitted code---in the template and your own implementation to the TAs/instructor. We will also put your code through a plagiarism detector to ensure that you do independent work. The penalty for plagiarism as always will be a fail grade in the course.

Submission

Submit the three files mm.h, mm1.c and mm2.c in a tarball named **<rollnumber>-PA3-malloc.tar.gz**

We will compile your file along with our version of the driver and run a number of traces that have not been provided in the handout.

Grading Rubric:

1. Implementing mm1.c: 30 points
2. Implementing mm2.c: 70 points
    a. Design explanation: Arguing for why you believe your design choices are optimal for the widest variety of workloads (5 points)
    b. Correctness (15 points)
    c. Throughput and Efficiency (50 points)