CS 744 Autumn 2019

Assignment 3: Writing your own Memory Allocator

In this Assignment, you have to write your own version of malloc, free and realloc using **`sbrk`** system call. Your implementation will be evaluated based on correctness, throughput and space utilisation.

Your dynamic memory allocator will consist of the following four functions, which are declared in **mm.h** and defined in **mm.c**.

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

`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 allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, and 0 otherwise.
The driver will call **`mm_init`** before running each trace (and after resetting the brk pointer). Therefore, your **`mm_init`** function should be able to reinitialize all state in your allocator each time it is called. In other words, you should not assume that it will only be called once.

`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.

`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.

`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.

Testing Your Allocator

To test your implementations, we are providing here (with the permission of the original author) a test harness with a driver. The malloc driver is a trace-driven program, called "**mdriver**", that checks your **`mm.c`** malloc package for correctness, space utilization, and throughput.
The driver expresses overall performance as an index in the range [0..100]. Malloc implementations with larger performance indices have better space utilization and throughput than those with smaller performance indices.

The performance index (**pindex**) is a linear combination of space utilization and throughput.
It is computed as

```
p1 = UTIL_WEIGHT * avg_mm_util;
if (avg_mm_throughput > AVG_LIBC_THRUPUT) {
      p2 = (double)(1.0 - UTIL_WEIGHT);
}
else {
      p2 = ((double) (1.0 - UTIL_WEIGHT)) *
                  (avg_mm_throughput/AVG_LIBC_THRUPUT);
}
pindex = (p1 + p2) * 100.0;
```

- **UTIL_WEIGHT** is a value between 0 and 1 that gives the contribution of space utilization (UTIL_WEIGHT) and throughput (1-UTIL_WEIGHT) to the performance index. Through trial and error, we have found that a UTIL_WEIGHT of around 0.6 works well; optimizing too much for the speed at the expense of space utilization leads to rather stupid

implementations. A higher value of UTIL_WEIGHT tends to discourage this, with the result that more intelligent implementations get better scores.

- **AVG_LIBC_THRUPUT** is an estimate of the throughput of the libc malloc on your system. Its only purpose is to cap the contribution of throughput to the performance index. Once you surpass the AVG_LIBC_THRUPUT, they get no further benefit to their score. We use a constant here rather than a measured value to make the index more stable.
- **avg_mm_util** is the average measured space utilization of your malloc package. The idea is to remember the high water mark "hwm" of the heap for an optimal allocator, i.e., where the heap has no gaps and no internal fragmentation. Utilization then is the ratio hwm/heapsize, where heapsize is the size of the heap in bytes after running the student's malloc package on the trace. Note that our implementation of mem_sbrk() doesn't allow your to decrement the **`brk`** pointer, so **`brk`** is always the high watermark of the heap.
- **avg_mm_throughput** is the average measured throughput (ops/second) of your malloc package on all of the traces.

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>

is balanced. It has a recommended heap size of 20000 bytes (ignored), three distinct request ids (0, 1, and 2), eight different requests (one per line), and a weight of 1 (ignored).

Your code will be tested on

1. **Correctness** by checking that block must be aligned properly, and must not overlap any currently allocated block.
2. **Space utilisation** by executing with different trace files which will check that you are reallocating previously allocated block, whether performing coalescing etc.
3. **Throughput** by calculating the number of operations per second.

Submission

You must do this assignment in teams of two. In other words, you may NOT submit this alone!

Just turn in one submission against the roll number of any student of the team on Moodle. Submit just two files mm.h and mm.c in a tarball named

`<rollnumber1>-<rollnumber2>-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.