Aligned Dynamic Memory Allocation

Abstract

You are required to implement an aligned version of the standard **malloc()** and **free()** function calls provided by the standard C library. The aligned version should allow user-space programs to allocate 'aligned' memory chunks, where the base pointer of an allocated chunk is aligned to a specific alignment value defined by the user-space program. Your program will typically run in an x86_64 GNU/Linux environment where gcc and/or clang shall be available.

Introduction

Dynamic memory allocation is the functionality that allows user-space programs to allocate heap memory space in **run-time**. In the C programming language, there are two main functions that provide dynamic memory allocation functionality (defined in **<stdlib.h>**):

- `void* malloc(size_t size);`

  malloc() function call is used by user-space programs to allocate a memory chunk of at least "**size**" in the heap segment. **malloc()** returns a pointer to the base address of the allocated memory chunk.

  Internally, **malloc()** might need to allocate more than "**size**" in order to store information about the allocated chunk and to enable optimization.

- `Void free(void* ptr);`

  free() function call is used by user-space programs to release a previously-allocated memory chunk. '**ptr**' is basically a pointer to the base address of a previously-allocated memory chunk (i.e. the pointer that was returned by **malloc()** when it was called to allocate this chunk).

  The **free()** function implementation will look into the metadata that is associated with the memory chunk itself (metada is usually stored in memory just before the allocated

memory chunk). Metadata would include information about the size of the chunk, so you don't need to pass this information again when you call **free()**.

Problem Statement

Using the dynamic memory allocation routines provided by the standard C library (**malloc()** and **free()**), you are required to implement **amalloc**() and **afree**() functions, which are defined as follows:

- `void* amalloc(size_t alignment, size_t min_size);`

  This function is an aligned version of **malloc()**. It allocates '**size**' bytes of uninitialized storage whose alignment is specified by '**alignment**'.

  For example, if '**alignment**' is 16, then the return pointer is guaranteed to be a multiple of 16. Valid examples are:
  - 0
  - 16
  - 32
  - 64
  - 0x1AB0
  - 0x3000

  That being said, if you decrease the alignment parameter by 1, then the returned pointer should evaluate to zero when it is bit-wise anded with the decreased value:

  ((uint64_t) ptr) & ((uint64_t) (alignment – 1)) → should evaluate to zero

  '**min_size**' parameter defines the least number of memory bytes that the caller wants to allocate. The function **amalloc()** might need to allocate more than '**min_size'** in order to achieve the alignment criterion and to store its own internal metadata.

  The function **amalloc()** would internally call the standard **malloc()** function provided by **<stdlib.h>** in order to do the actual heap-space allocation. The **amalloc()** function would typically return **NULL** if **malloc()** cannot allocate the required size, or the parameters are invalid.

- `void afree(void* ptr);`

  This function is used to deallocate what has been allocated by **amalloc()**. It only takes

'**ptr**' as a parameter, which shall be an aligned pointer that was previously returned by **amalloc()**.

afree() would internally call **free()** to deallocate all the spaces that **amalloc()** had previously allocated for '**ptr**', this includes any additional space allocated to store metadata.

afree() would typically have an internal mechanism to determine where the metadata is stored in memory and what parameters should be passed to **free()**.

To be compliant with the standard function **free(),** your implementation of **afree()** should accept a **NULL** value for the parameter '**ptr**'. If '**ptr**' is **NULL**, you should simply return without doing any action (it is not an error).

Testing your Implementation

You need to test your implementation and make sure that:

- Your implementation doesn't throw unexpected exceptions (like segmentation fault) for different test cases.
- Your implementation doesn't take much time to execute (the optimal solution is **O(cM)** where **c** is a constant, and **O(M)** is the order of the standard **malloc()**/**free()** functions). That being said, your solution should run in constant time regardless of **malloc()** and **free()** complexity.
- Your **amalloc()** implementation should always return an aligned pointer, or return **NULL** if it cannot allocate the required space or the arguments are invalid.
- Your **afree()** implementation should always free up the aligned pointer. If an invalid pointer (which is not equal to **NULL**) is passed to **afree()**, **afree()** should report that to **stderr** without throwing and exception. **afree()** should also be able to detect double frees (i.e. calling **afree()** again with the same ptr although it has already been deallocated) and should handle this case without throwing an exception.

You can implement your unit test cases in an external file (namely **main.c**), and implement your function calls in other files (**amalloc.c** and **afree.c**). It's totally up to you to define your unit tests, but make sure that they cover the cases described above.

References

We have used the following man pages in order to prepare this document:

- http://man7.org/linux/man-pages/man3/malloc.3.html