P2: Memory Management & Layering

Overview

Due to your excellent work handling the metadata for project Sky Skink, the great Silurian overload Reptar himself has commissioned your work in designing a new custom Memory Manager application for Reptilian. This new project, deemed **Project Repto**, is built using functionality from various layers of the OS to avoid human corruption. For extra security, the memory manager will write a log of its state when closed. If all goes well, your manager will be deployed all throughout the great Lizard Legion.

In this project, you will implement a memory manager in C++, whose features include initializing, tracking, allocating, and deallocating sections of memory. You will integrate the memory manager into a console program that we provide. The program will provide some simple unit testing to check for correctness but will **not** check for memory leaks and errors - *it is your responsibility to test for leaks/errors and ensure overall correctness!* You must also maintain a <u>valid listing of memory holes</u> and <u>combine those holes</u> as appropriate and discussed in the class and text. You'll then create a short video to demonstrate your code and submit the project via Canvas.

Structure

Modern operating systems are often built in layers with specific goals and limited privileges. Layering also facilitates the implementation of recognized standards by separating hardware and OS-specific implementations from generalized API calls. Your memory manager will make use of these standard functions so that it can be used in console applications.

The project is broken into three main parts:

1) Write a memory manager in C++ using basic system functionality.
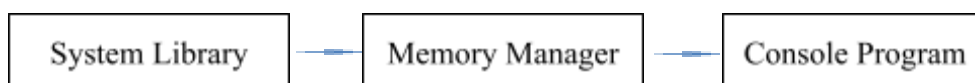2) Integrate the memory manager into the console-based testing program we provide.

Figure 1: Memory manager is instantiated from a console program.

While exact implementation may vary, *the library functions must match the signatures laid out in this document.*

Specification

The memory manager will incorporate the following class(es) and function(s).

Memory Manager Class

The Memory Manager will handle the allocation/deallocation of memory and provide details of its state. How the class keeps track of allocation/deallocation is implementation dependent and is left for the student to decide. `MemoryManager.h` and `MemoryManager.cpp` will contain declaration and definition, respectively.

public `MemoryManager(`*unsigned* `wordSize, std::`*function*`<int(int, void *)> allocator)`
Constructor; sets native word size (in bytes, for alignment) and default allocator for finding a memory hole.

public `~MemoryManager()`
Releases all memory allocated by this object *without leaking memory*.

public *void* `initialize(size_t sizeInWords)`
Instantiates block of requested size, no larger than 65536 words; cleans up previous block if applicable.

public *void* `shutdown()`
Releases memory block acquired during initialization, if any. This should only include memory created for long term use not those that returned such as *getList()* or *getBitmap()* as whatever is calling those should delete it instead.

public *void* `*allocate(size_t sizeInBytes)`
Allocates a memory using the allocator function. If no memory is available or size is invalid, returns `nullptr`.

public *void* `free(void *address)`
Frees the memory block within the memory manager so that it can be reused.

public *void* `setAllocator(std::`*function*`<int(int, void *)> allocator)`
Changes the allocation algorithm to identifying the memory hole to use for allocation.

public *int* `dumpMemoryMap(char *filename)`
Uses standard POSIX calls to write hole list to filename **as text**, returning **-1** on error and **0** if successful.

Format: "`[START, LENGTH] - [START, LENGTH] ...`", e.g., "`[0, 10] - [12, 2] - [20, 6]`"

public *void* `*getList()`
Returns an array of information (**in decimal**) about holes for use by the allocator function (*little-Endian*). Offset and length are in <u>words</u>. If no memory has been allocated, the function should return a *NULL* pointer.

Format:

NUMBER OF HOLES	HOLE 0 OFFSET	HOLE 0 LENGTH	HOLE 1 OFFSET	HOLE 1 LENGTH	...
2B	2B	2B	2B	2B	

Example: `[3, 0, 10, 12, 2, 20, 6]`

```
    public void *getBitmap()
```
Returns a bit-stream of bits in terms of an array representing whether words are used (**1**) or free (**0**). The first two bytes are the size of the bitmap (*little-Endian*); the rest is the bitmap, word-wise.
Note : In the following example B0, B2, and B4 are holes, B1 and B3 are allocated memory.

        Hole-0   Hole-1   Hole-2   ⌐─B4─⌐   ⌐B2⌐ ⌐────B0───⌐   ⌐Size (4)⌐ ⌐This is Bitmap in Hex⌐
Example: **[0,10]-[12,2]-[20,6]** **[00 00001111 11001100 00000000]**   **[0x04,0x00,0x00,0xCC,0x0F,0x00]**
                                         └─*B3*─┘   └*B1*┘
Returned Array: **[0x04,0x00,0x00,0xCC,0x0F,0x00]** or **[4,0,0,204,15,0]**

```
    public unsigned getWordSize()
```
Returns the word size used for alignment.

```
    public void *getMemoryStart()
```
Returns the byte-wise memory address of the beginning of the memory block.

```
    public unsigned getMemoryLimit()
```
Returns the byte limit of the current memory block.

Note: The following two functions should not be part of the Memory Manager Class.
    <u>Memory Allocation Algorithms</u>
```
    int bestFit(int sizeInWords, void *list)
```
Returns **word offset** of hole selected by the best fit memory allocation algorithm, and **-1** if there is no fit.

```
    int worstFit(int sizeInWords, void *list)
```
Returns **word offset** of hole selected by the worst fit memory allocation algorithm, and **-1** if there is no fit.

    <u>Testing</u>
Your code must also compile at the command line and function with provided sample files. These sample files cover a base level functionality for the functions above. ***It is critical that you test for memory leaks and errors!*** The code we provide will not test for this. Submissions with memory leaks/errors will be marked <u>**zero**</u> for functionality.

Extra Credit

In the above implementation of `MemoryManager.initialize(...)`, you most likely made use of the **new** operator to acquire your initial block of memory. Your job is to now figure out how to allocate your initial block of memory without the use of **new** or any stdlib functions, e.g. `malloc, calloc`, etc.

Submissions

You will submit the following at the end of this project:

- Report (**MemoryManager.txt**) in man format on Canvas, including link to unlisted YouTube screencast
- Compressed tar archive (**MemoryManager.tgz**) containing source/build files for text program on Canvas
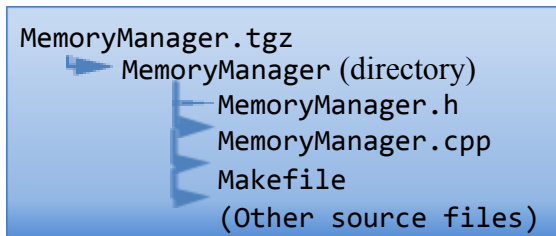
Report

Your report will explain how you implemented all your functions and tested them. It will describe how testing was performed, any known bugs, and why the output is correct. The report should be created using **man** format saved as a .txt. The report should be no more than 500 words (about two pages in man format), cover all relevant aspects of the project, and be organized and formatted professionally – ***this is not a memo!***

Screencast

Students should submit a screencast (with audio) walking through each class and function. It should include showing/demoing your changes in action (no more than 5 minutes). Audio speed-up is prohibited. Video cannot have watermarks and <u>must be an unlisted YouTube video!</u>

Compressed Archive (MemoryManager.tgz)

Your compressed tar file should have the following directory/file structure:

```
MemoryManager.tgz
    ➤ MemoryManager (directory)
        ── MemoryManager.h
        ▶ MemoryManager.cpp
        ▶ Makefile
        ▶ (Other source files)
```

To build your program, we will execute these commands:

```
$ tar zxvf MemoryManager.tgz
$ cd MemoryManager
$ make
$ cd ..
```

To link against the library, we will execute this command:

```
$ c++ -std=c++17 -o program_name sourcefile.c -L ./MemoryManager -lMemoryManager
```

Please test your functions before submission! If your code does not compile it will result in **<u>zero credit</u>** (0, none, goose-egg) for that portion of the project.

Helpful Links

You may find these resources useful as you develop and test your memory manager.

Development

https://www.cs.rit.edu/~ark/lectures/gc/03_00_00.html

https://www.ibm.com/developerworks/library/pa-dalign/index.html

https://en.cppreference.com/w/cpp/utility/functional/function

Testing

http://valgrind.org/

https://github.com/catchorg/Catch2