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 valid listing of memory holes and combine those holes 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 class in C++ using basic system functionality.
2) Write two memory allocation algorithms to be used by your memory manager.
3) Package your class and algorithms together into a static library to be used by our console testing program.

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 65535 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 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 that starts at the given address within the memory manager so that it can be reused.

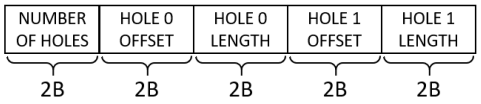
```
public void setAllocator(std::function<int(int, void *)> allocator)
```
Changes the algorithm used in allocation to find a memory hole.

```
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 words. If no memory has been allocated, the function should return a ***NULL*** pointer.

Format:



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.

Example: `[0,10]-[12,2]-[20,6]` → `[00 00001111 11001100 00000000]` → `[0x04,0x00,0x00,0xCC,0x0F,0x00]`

Hole-0 *Hole-1* *Hole-2* — *B4* — *B2* — *B0* — *Size (4)* — *This is Bitmap in Hex* — *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 acquired during initialization.

```
public unsigned getMemoryLimit()
```
Returns the byte limit (total size, in bytes) of the current memory block.

Note: The following two functions should not be part of the Memory Manager Class.

Memory Allocation Algorithms

```
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.

Testing

Your code must compile and function with the provided testing file. ***It is critical that you test for memory leaks and errors!*** The code we provide only covers base level functionality for the functions above and will not test for these. Submissions with memory leaks/errors will be marked **zero** for functionality. Make sure to remove any stray cout or print calls prior to submitting, as they may interfere with output comparisons.

Makefile

You will write a Makefile that creates **`libMemoryManager.a`** and **`MemoryManager.o`**. It may create other .o files if necessary for your implementation. Your Makefile must only build your library, make sure it does not build any executables (like CommandLineTest.cpp).

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. You may still use **`new`** and stdlib functions anywhere else in your implementation, just not in **`initialize`**.

Submissions

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

- Plain text file containing YouTube link to unlisted screencast video (**`screencast.txt`**)
- Compressed tar archive files for **`MemoryManager`** library (**`MemoryManager.tgz`**)

Screencast

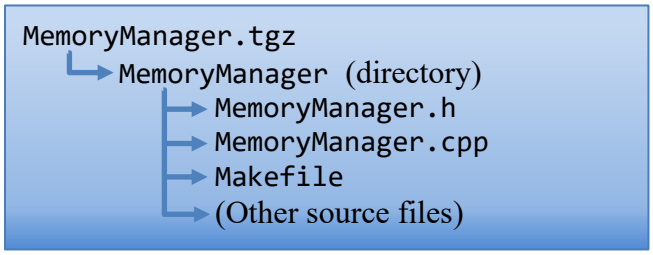
You will record a screencast (with audio) walking through all your library code. As you explain something, you should be showing the corresponding file/code. You will upload the recording as an unlisted YouTube video and submit a text file containing a link to this video.

Your screencast should:

- Walk through all your library code, including the allocation algorithms and any custom helper functions.
  - Make sure to explain your logic thoroughly!
- Explain how your code tracks holes and allocated regions, and what data structure(s) you used to do so.
  - Be sure to cover how holes & allocated regions are created/destroyed, and how holes are merged.
- If you completed the extra credit, explain how (say more than just “I used x,” explain how you used x).
- Demo functionality by running the provided test suite and checking for memory leaks and errors.
- Be at most **5 minutes** long. (Audio speed-up is prohibited)

Compressed Archive (MemoryManager.tgz)

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



Do NOT include any extraneous files in your compressed archive, such as output files (*.a, *.o), *.sh scripts, CommandLineTest.cpp, etc.

To create the archive, go to the directory containing your MemoryManager folder and use the command:

```
$ tar -zcvf MemoryManager.tgz MemoryManager
```

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:

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

Please test your functions before submission! If your code does not compile it will result in **zero credit** (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