KAIST
EE 209: Programming Structures for EE

Assignment 5: A Dynamic Memory Manager Module

(This assignment is borrowed and slightly modified from Princeton COS 217)

Purpose

The purpose of this assignment is to help you understand how dynamic memory management works in C. It also will give you more opportunity to use the GNU/Unix programming tools, especially `bash`, `emacs`, `gcc209`, `gdb`, and `make`.

Rules

"Checking invariants" (as described below) is the "on your own" part of this assignment. That is, when doing that part you may consult with your partner, but must not consult with the course instructors, lab TAs, Moodle, etc., except perhaps to clarify requirements. That part is worth 15% of this assignment.

Background

A standard C programming environment contains four functions that allow management of the runtime heap: `malloc`, `free`, `calloc`, and `realloc`.

Those heap management functions are used heavily in many C programs.

Section 8.7 of the book *The C Programming Language* (Kernighan and Ritchie) shows an implementation of the `malloc` and `free` functions. That book is on reserve at KAIST library. The key data structure in that implementation is a circular singly-linked list; each free memory "chunk" is stored in that list. Each memory chunk contains a header which specifies its size and, if free, the address of the next chunk in the list. Although elegant in its simplicity, that implementation can be inefficient.

The web page http://gee.cs.oswego.edu/dl/html/malloc.html (Doug Lea) describes how one can enhance such an implementation so it is more efficient. The key data structure is an array of non-circular doubly-linked lists, that is, an array of "bins." Each bin contains all free chunks of a prescribed size. The use of multiple bins instead of a single linked list allows `malloc` to be more efficient.

Moreover, each memory chunk contains both a *header* and a *footer*. The header contains three fields: the size of the chunk, an indication of whether the chunk is free, and, if free, a pointer to the next free chunk in its bin. The footer contains two fields: the size of the chunk, and, if free, a pointer to the previous free chunk in its bin. That chunk structure allows `free` to be more efficient.

A more thorough description of the pertinent data structures and algorithms will be provided in lectures and precepts.

Your Task

You are given the interface of a HeapMgr (heap manager) module in a file named heapmgr.h. It declares two functions:

```
void *HeapMgr_malloc(size_t uiSize);
void  HeapMgr_free(void *pv);
```

You also are given three implementations of the HeapMgr module:

- heapmgrgnu.c is an implementation that simply calls the GNU malloc and free functions provided with our lab machine development environment.
- heapmgrkr.c is the Kernighan and Ritchie implementation, with small modifications for the sake of simplicity.
- heapmgrbase.c is an implementation that you will find useful as a baseline for your implementations.

Your task is to create two additional implementations of the HeapMgr module. Your first implementation, heapmgr1.c, should enhance heapmgrbase.c so it is reasonably efficient. To do that it should use a single doubly-linked list and chunks that contains headers and footers (as described above, in lectures, and in precepts). Unlike the K&R implementation, the baseline code uses a **non-circular** list, and your code should implement a **non-circular, doubly-linked** list.

If designed properly, heapmgr1.c will be reasonably efficient in most cases. However, heapmgr1.c is subject to poor worst-case behavior. Your second implementation, heapmgr2.c, should enhance heapmgr1.c so the worst-case behavior is not poor. To do that it should use multiple doubly-linked lists, alias *bins* (as described above, in lectures, and in precepts).

Your `HeapMgr` implementations should not call the standard `malloc`, `free`, `calloc`, or `realloc` functions.

Your `HeapMgr` implementations should thoroughly validate function parameters by calling the standard `assert` macro.

Your `HeapMgr` implementations should check invariants by:

- Defining a thorough `CheckHeapValidity` function, and
- Calling `assert(CheckHeapValidity())` at the leading and trailing edges of the `HeapMgr_malloc` and `HeapMgr_free` functions.

Logistics

Develop on lab machines, using `emacs` to create source code and `gdb` to debug.

You can download following files using `wget` command on lab machines:

```
wget
http://www.ndsl.kaist.edu/~swhang/ee209/assignment/heapman/src/filename
```

For example,

```
wget
http://www.ndsl.kaist.edu/~swhang/ee209/assignment/heapman/src/heapmgr.h
```

should downlaod heapmgr.h to your current directory("."). Note that `testheap` and `testheapimp` are used as executables, so you need to enable the file execution bit like "`chmod u+x testheap`" before using them.

- `heapmgr.h`, `heapmgrgnu.c`, `heapmgrkr.c`, and `heapmgrbase.c`: as described above.

- chunkbase.h and chunkbase.c: a Chunk module used by heapmgrbase.c.
- chunk.h and chunk.c: a Chunk module that you may use in both implementations of your HeapMgr module.
- testheapmgr.c: a client program that tests the HeapMgr module, and reports timing and memory usage statistics.
- testheap and testheapimp: bash shell scripts that automate testing. The testheap script assumes the existence of executable files named testheapmgrgnu, testheapmgrkr, testheapmgrbase, testheapmgr1, and testheapmgr2 (as described below).

The testheapmgr program requires three command-line arguments. The first should be any one of seven strings, as shown in the following table, indicating which of seven tests the program should run:

Argument	**Test Performed**
LifoFixed	LIFO with fixed size chunks
FifoFixed	FIFO with fixed size chunks
LifoRandom	LIFO with random size chunks
FifoRandom	FIFO with random size chunks
RandomFixed	Random order with fixed size chunks
RandomRandom	Random order with random size chunks
Worst	Worst case order for a heap manager implemented using a single linked list

The second command-line argument is the number of calls of HeapMgr_malloc and HeapMgr_free that the program should execute. The third command-line argument is the (maximum) size, in bytes, of each memory chunk that the program should allocate and free.

Immediately before termination testheapmgr prints to stdout an indication of how much CPU time and heap memory it consumed. See the `testheapmgr.c` file for more details.

To test your `HeapMgr` implementations, you should build two programs using these `gcc209` commands:

```
gcc209 -std=gnu99 testheapmgr.c heapmgr1.c chunk.c -o testheapmgr1
gcc209 -std=gnu99 testheapmgr.c heapmgr2.c chunk.c -o testheapmgr2
```

To collect timing statistics, you should build five programs using these `gcc209` commands:

```
gcc209 -O3 -D NDEBUG -std=gnu99 testheapmgr.c heapmgrgnu.c -o
testheapmgrgnu
gcc209 -O3 -D NDEBUG -std=gnu99 testheapmgr.c heapmgrkr.c -o
testheapmgrkr
gcc209 -O3 -D NDEBUG -std=gnu99 testheapmgr.c heapmgrbase.c chunkbase.c
-o testheapmgrbase
gcc209 -O3 -D NDEBUG -std=gnu99 testheapmgr.c heapmgr1.c chunk.c -o
testheapmgr1
gcc209 -O3 -D NDEBUG -std=gnu99 testheapmgr.c heapmgr2.c chunk.c -o
testheapmgr2
```

The `-O3` (that's uppercase "oh", followed by the number "3") argument commands gcc to optimize the machine language code that it produces. When given the `-O3` argument, `gcc` spends more time compiling your code so, subsequently, the computer spends less time executing your code. The `-D NDEBUG` argument commands gcc to define the `NDEBUG` macro, just as if the preprocessor directive `#define NDEBUG` appeared in the specified .c file(s). Defining the `NDEBUG` macro disables the calls of the `assert` macro within the `HeapMgr` implementations. Doing so also disables code within `testheapmgr.c` that performs (very time consuming) checks of memory contents.

Create additional test programs as you deem necessary. You need not submit your additional test programs.

Create a `makefile`. The first dependency rule of the `makefile` should build five executable
files: `testheapmgrgnu`, `testheapmgrkr`, `testheapmgrbase`, `testheapmgr1`,
and `testheapmgr2`. That is, the first dependency rule of
your `makefile` should be:

```
all: testheapmgrgnu testheapmgrkr testheapmgrbase testheapmgr1
testheapmgr2
```

The `makefile` that you submit should:

- Maintain object (.o) files to allow for partial builds of all five executable binary files.
- Encode the dependencies among the files that comprise your program.
- Build using the `-O3`, `-D NDEBUG`, and `-std=gnu99` options.

We recommend that you **create your `makefile` early** in your development process. Doing so will allow you to use and test your `makefile` during development.

Create a `readme` text file that contains:

- Your name and student ID.
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course "Policy" web page.

- The **CPU times and heap memory consumed** by `testheapmgr` using `heapmgrgnu.c`, `heapmgrkr.c`, `heapmgrbase.c`, `heapmgr1.c`, and `heapmgr2.c`, with tests `RandomRandom` and `Worst`, with call count 100000, and with maximum chunk sizes 1000 and 10000. Note that if the CPU time consumed is more than **5 minutes**, `testheapmgr` will **abort execution**. To report the time and memory consumption, it is sufficient to paste the output of the `testheap` script into your `readme` file.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Prepare the Ethics document. Sign on it, save it as a PDF file, and submit the PDF file.

Submit your work electronically on KLMS via following commands:

```
mkdir 20161234_assign5
mv heapmgr1.c heapmgr2.c readme makefile EthicsOath.pdf
20161234_assign5
tar zcf 20161234_assign5.tar.gz 20161234_assign5
```

Grading

We will grade your work on quality from the user's point of view and from the programmer's point of view. From the user's point of view, your module has quality if it behaves as it should. The correct behavior of the `HeapMgr` module is defined by the previous sections of this assignment specification. From the programmer's point of view, your module has quality if it is well styled and thereby simple to maintain. See the specifications of previous assignments for guidelines concerning style. Specifically, function modularity will be a prominent part of your grade. To encourage good coding practices, we will deduct points if `gcc209` generates warning messages.