CSC 412 – Operating Systems

Programming Assignment 04, Spring 2022

Thursday, March 10th, 2022

Due date: Friday, March 25, 11:55pm.

1 What This Assignment Is About

1.1 Objectives

The objectives of this assignment are for you to

- Get more practice with C pointers and `struct` data type;
- Implement a "modified" double-linked list (i.e., a list in which insertion and removal operations have side effects on other elements of the list);
- Implement a simplified version of a runtime memory manager;
- Use a bash script to generate a C++ program for testing purposes.

This is an individual assignment.

1.2 Memory manager

You are going to write new functions to "replace" the standard `malloc`, `calloc`, and `free` functions of C. Of course, you are not really going to replace these functions, as we will still need a call to `malloc` to create our "heap," but you will get to manage the list of allocated and free space inside that heap.

At the beginning of the execution of your program, you will allocate a large section of memory[1]. Subsequently, calls to your replacement `malloc` and `free` functions will reserve and free space within that memory section under your control.

At the beginning of your program you will use C preprocessor commands (macros) to switch at build time between the regular C runtime memory manager and your replacement manager.

1.3 Handout

There is no handout for this assignment.

[1]Of course, that memory section will be within the true heap of your process: We are just pretend-replacing the regular memory manager.

2 Part I: Runtime Memory Manager, Version 1

2.1 Organization of the source and header files

This week we need to start with this aspect of the project. Let's look at what needs to be implemented:

- Your "main" source file will need to be able to operate either with the regular memory management functions (and therefore to load the `stdlib.h` header file) or with your replacement functions.

- Your replacement functions cannot be enabled in a source file where `stdlib.h` has been loaded. They could still be implemented in the "main" source files and be disabled (through a preprocessor `#if` or `#ifdef` statement) when `stdlib.h` is loaded, but really, such low-level functions have no business being in the main source file. You will therefore implement them in a separate source file `replacementManager.c` with a companion header `replacementManager.h`. The main source file will load either that header or the `stdlib.h` header.

- When using your custom runtime memory manager, you still need to allocate, using `malloc`, a large chunk of memory at the beginning of execution. This *cannot* take place in the main source file since we would be in the case where `stdlib.h` has not been loaded, and for the same reason it cannot take place in the source file where the replacement `malloc` and `free` functions are implemented. We have no choice here: We will need a third source file with its companion header. But now we have a problem: The "pretend heap" (including lists of free and allocated space) is allocated in a function implemented in that source file, but the functions implemented in `replacementManager.c` need to have access to that information (and it cannot passed as arguments to the functions since we want to preserve the syntax and semantics of `malloc` and `free`). The only way to do this is to use application-level global variables, using the `extern` keyword[2].

2.2 Preprocessor statements (macros)

At the very beginning of your main source file, you should define a macro that will take the value 1 or 0 depending whether your replacement memory manager should be used. Then, throughout the source file, the value of that macro will be used to load `replacementManager.h` or `stdlib.h`, and to enable or disable the allocation of the "replacement heap."

```
#define USE_REPLACEMENT_MANAGER   1 // or
#if USE_REPLACEMENT_MANAGER
    // do one thing, e.g. load one header
#else
    // do something else
#endif
```

[2]Be careful not to develop the bad habit of using `extern` throughout your programs instead of passing parameters to functions. Yes, it is convenient, but it is something that you should only do if there is no reasonable alternative solution. The overuse of global variables is a sign of bad code design.

2.3 Data structure

Your runtime memory manager is going to maintain a list of all the space already allocated or free within the "heap." It will store that information in the form of a double-linked list of nodes. The head of this list is "private" to your memory manager and therefore should not be accessible directly from your main program. The information to store at a node will *a minima* include a start address, a size, and a flag indicating whether the node represents free space or space allocated (and of course the pointers necessary to implement the list, as we saw in lab this week).

Feel free to include any other information that you may find relevant. Discuss the design of your node struct in the report.

2.4 Memory allocation algorithm

Your memory manager should use a *first fit* strategy to allocate space when requested by the program.

2.5 Handle failure

When your memory manager cannot find a hole large enough for the amount required, your `malloc` replacement function should return `NULL`. When the program calls `free` for a pointer for which you didn't record an allocation, your `free` replacement function should terminate execution with an error message.

2.6 Heap map

Implement a function that prints out a "heap map" giving the list of allocated and free space on your heap.

Add a "verbose" state variable to your memory manager so that when in "verbose" mode the memory manager prints the heap map following each memory allocating or freeing operation.

2.7 Start and end of execution of the program

This version of the program doesn't take any arguments and is a simple testbed for your memory manager, will all calls to the memory manager hard-coded in the main function.

When in `USE_REPLACEMENT_MANAGER` mode, your program should make an initialization call to the memory manager to allocate "heap space." After that, regardless of the mode, it should make a sequence of calls to the appropriate `malloc` and `free` functions.

At the end of the program, when in `USE_REPLACEMENT_MANAGER` mode, the main function should make a "shut down" call to the memory manager to free the "heap" that it had allocated. At this point, the memory manager should verify that all space allocated has been freed, and print a warning if some space was not freed.

2.8 Extra credit

2.8.1 Extra credit 1 (3 points)

Implement a replacement function for `calloc`. If you do so, don't forget to include calls to this function in your hard-coded test sequence in the main function.

2.8.2 Extra credit 2 (up to 8 points)

Implement the *best fit* and *worst fit* algorithms. To select the algorithm to run, define new macros `FIRST_FIT`, `BEST_FIT`, and `WORST_FIT`, and assign one of these three values to a macro `ALGORITHM_CHOICE`. Based on the value selected, the appropriate implementation of `malloc` should be called.

There are 3 points of extra credit for each algorithm implementation and 2 more points of extra credit for a good implementation of the preprocessor commands that handle the algorithm selection.

2.8.3 Extra credit 2 (8 points)

Combine your hole fitting algorithm with a "smart search" strategy that maintains separate lists of common hole sizes (small, medium, large, very large).

3 Part II: The `bash` Script and Test Code

3.1 Changes to the C++ code

You shouldn't have to change a thing to your memory manager. The script will generate the source file of a new C++ program, and then compile and execute this program.

3.2 What the script does

Your script, named `script04.sh`, will take as arguments Like in the last assignment, the script will build the executable of the program, and will launch it with the following arguments: , the size (in bytes) of the "heap" to preallocate (this information would be ignored if your program is built to use the regular `malloc` and `free` functions) followed by the list of memory management commands to execute. The memory management commands will have the following syntax:

- the path to the directory containing the memory manager code;
- the optional string `-v` if the memory manager should be operating in "verbose" mode;
- the size (in bytes) of the "heap" to preallocate (this information would be ignored if your program is built to use the regular `malloc` and `free` functions);
- a list of memory commands that the generated C++ program should perform:
  - `m <size in bytes>`: simply means: allocate a block of the required size.

  - `c <size in bytes>`: if you implemented the replacement `calloc`, clear (set to 0) and allocate a block of the required size.
  - `f <index of block>`: means free the block indicated

An example of valid list of arguments for the script would be

```
  ../Version1 -v 12000 m 3000 m 1000 m 2400 m 500 f 1 m 500 m 2000
m 1500 f 2 f 0 f 6 m 1000 f 7 f 4 f 5 f 3
```

for the above list of arguments, the source code produce would include the following fragment:

```
void* p0 = malloc(3000);
void* p1 = malloc(1000);
void* p2 = malloc(2400);
void* p3 = malloc(500);
free(p1);
void* p4 = malloc(500);
void* p5 = malloc(2000);
void* p6 = malloc(1500);
free(p2);
free(p0);
 free(p6);
void* p7 = malloc(1000);
free(p7);
free(p4);
free(p5);
free(p3);
```

and of course all the code needed around this fragment to produce a complete C++ program.

4 Part III: Runtime Memory Manager, Version 2

4.1 Handles

4.1.1 Motivation

One problem with our small memory manager is that after a while the heap can become severely fragmented. We will see in the second half of the semester how modern memory management addresses this problem. The old Mac OS (pre OS X) did so by relying massively on *handles* (in relatively recent years, Window's `.Net` has introduce handles for some memory management operations, which shows that old clever solutions may always come back in a new setup).

Before I explain what a handle is and how it helps address (no pun intended) the memory fragmentation problem, let me explain what that problem is. Let's say that you allocate an array of 1000 `int` values: `int* a = (int*) calloc(1000, sizeof(int));`
Later on, the memory manager determines that if only it could move that block of 4000 bytes at some other location on the heap, it could regroup together a big chunk of free space. The only

problem is that it cannot do that because the value stored in `a` would be invalid, and the memory manager has no way to know in which of your variables that address value is stored, so it cannot go modify the value stored in that variable to be that of the relocated block of 4000 bytes.

4.1.2 The handle solution

Here come in *handles*. Handles are nothing more than pointers to pointers. The variables of your program storing pointers to memory locations would not be allowed anymore to store "master pointers" but pointers to master pointers, that is, handles. In our example, we would need a new function to produce handles (and a new `Handle` data type. So, for example we would write

```
Handle a = makeHandle(1000, sizeof(int));
```

If we wanted to access the data, we could to dereference the pointer twice:

```
int* av = (int*)(*a);
av[120] = 2019;
```

There are two major issues with this scheme. The first is that it is cumbersome; the second is that it is prone to leading to data type errors. Both objections are valid, but before we look at fixes or workarounds, let's see why the handles solution was/is considered worth the trouble. Let's consider the state of our heap after the process has been executing for a while, dynamically allocating and freeing memory. Things could look as pictured in Figure 1.

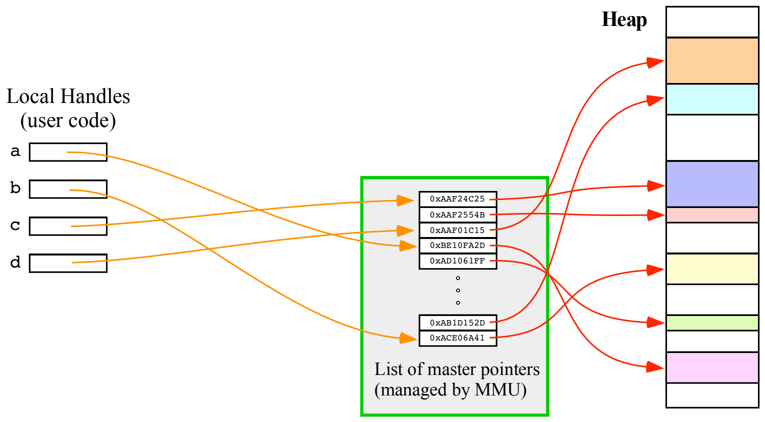

Figure 1: Example of state of the heap, master pointers, and local handles after a process has been running for a while.

The handles are local variables in the various functions currently on the stack. They point to master pointers managed by the memory manager. These master pointers point to the actual data stored on the heap. Now, the memory manager can rearrange the blocks of memory to defragment the heap, and it can do so because it can change the addresses stored in the master pointers, while the values stored in the user-controlled handles remain unchanged: They still point to the same master pointers, resulting in the heap map shown in Figure 2.

To summarize what just happened: The runtime memory manager can now perform warehousing operations in the background without the user process even being "aware" of such operations.

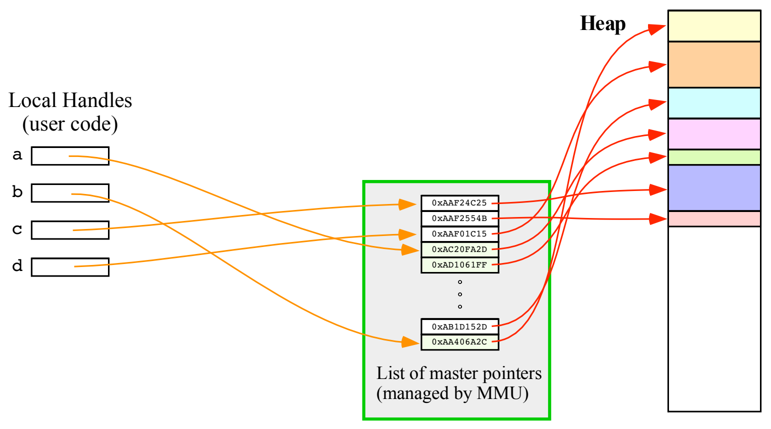

Figure 2: State of the heap, master pointers, and local handles after the memory manager has defragmented the heap.

There is still a big problem with the implementation of the scheme. We will address this problem later, and implement a solution, but I hope you appreciate the cleverness of the solution and see why it is worth a good amount of trouble: It gives the possibility to eliminate the problem of heap external fragmentation in a way that it almost transparent to the developer.

4.1.3 How is the `Handle` type defined?

How do we differentiate a handle used to access `double` data from one used to access `int` or `struct Node` data? If there exists a single `Handle` data type, then we cannot. Since all pointers have the same size, an `int**` handle or a `struct Node**` both will occupy 8 bytes (on a 64-bit OS. So, simplest, most C-ish way to define the type is:
`typedef void* Handle;`
Note the single `*` in the definition. Now, before dereferencing the handle to access the data, it must be cast to the proper type:

```
Handle a = makeHandle(1000, sizeof(int));
...
int* av = *((int**)a);
av[520] = 2019;
```

If you find this weird, you had better get used to it, because this is a very common way to implement type abstraction in C, and we will encounter it over and over when we start looking at process and thread creation.

4.1.4 Practical issues on the developer's side

As mentioned earlier, there are two immediate issues with the handle solution: It is slightly cumbersome, adding one more step (dereferencing the handle to get access to the master pointer), and it is prone to to leading to data type errors. The first issue is casually dismissed as being a negligible cost to pay for the benefits of the solution. The second issue is more problematic.

Let's look again at that code fragment:

```
Handle a = makeHandle(1000, sizeof(int));
...
int* av = *((int**)a);
av[120] = 2019;
```

There is nothing in the name `Handle` that informs us about the type of the data it gives access to. The declaration of the handle and its dereferencing may be distant from each other, which could lead to a mistake when casting. I could have mistakenly written

```
Handle a = makeHandle(1000, sizeof(int));
...
double* av = *((double**)a);
av[520] = 2019;
```

I wouldn't get any compiler warning, and the program may even seem to run fine for a while, but the address that `av[520]` refers to is outside of the memory area allocated for the array. Again, this is a common problem in C, and the common wisdom is that because it is so easy to make this kind of mistakes in C, one just has to be extra careful.

One way of being "extra careful" is by creating new types for specific types of handles, in order to get as much of type enforcement help as we can from the compiler. So, for example, we could define

```
typedef int* IntPointer;
typedef double* DoublePointer;
typedef int** IntHandle;
typedef double** DoubleHandle;
```

and now we could dereference our handles without an explicit typecast:

```
IntHandle a = (IntHandle) makeHandle(1000, sizeof(int));
...
IntPointer av = *a;
av[520] = 20.19;
```

We now would get a compiler error, or at least a warning if we mis-cast:

```
IntHandle a = (IntHandle) makeHandle(1000, sizeof(int));
...
DoublePointer av = *a; // implicit cast *int --> *double
av[520] = 20.19;
```

The drawback of this approach is that it forces us to define specific handle and pointer types for *all* the data types that we want to use. This is the way that the old Mac OS operated. There were handle types for every single data type encountered in the API: buttons, movies, text, windows, etc.

Another way to "be extra careful" is to adopt a convention for variable identifiers that informs about the type of the data. The most famous such convention is the "Hungarian notation." In this convention, a variable's identifier is prefixed with one or more letters that indicate the type of the data. For example:

- `strName` would indicate that the variable stores a string;
- `u8Label` or `ucLabel` indicates that the label is of type `unsigned char`;
- `iCounter` indicates that the counter is of type `int`;
- `afHeight` indicates that the height is stored in an array of `float`;
- etc.

I am not the biggest fan of the Hungarian notation, would it be only because I think that it's ugly, but I am not opposed to people using it (as long as I don't pay for the development). You can consult the Wikipedia page if you want to further investigate this naming convention.

4.1.5 The real problem with handles, and its solution

Let's look once more at that code fragment, with some minor additions:

```
IntHandle a = (IntHandle) makeHandle(1000, sizeof(int));
...
IntPointer av = *a;
for (int k=0; k<1000; k++) {
    // do things with av[i]
}
```

To see what the potential problem is, you have to start thinking in a multi-processed way: The memory manager is running concurrently with your process. What would happen if, while your process is going through the above loop, in the middle of that "do things with av[i]" block, the memory manager decided that now is a good time to defragment the heap, and therefore maybe relocate the memory block storing the array's data? The `av` pointer would not point to the right location anymore, and therefore the program would end up reading the wrong data and/or overwriting the data of another data structure.

How can we solve this problem: By adding two functions to lock and unlock respectively the location of the memory block addressed by a handle. When a handle is "locked," the memory manager is not allowed to relocate the memory block it refers to (but may still move the blocks of other unlocked handles). Our new code would become

```
IntHandle a = (IntHandle) makeHandle(1000, sizeof(int));
...
lock(a);
IntPointer av = *a;
for (int k=0; k<1000; k++) {
    // do things with av[i]
}
unlock(a);
```

Yes, that code is more complicated than what we started with, but now our heap can be safely defragmented while the process is executing, and the gains are well worth the price of a few extra lines of coding.

Note that we don't need to pass a pointer to the handle, because the handle is not what is being modified: The handle gives us access to the table where the master pointer is stored, and the locked/unlicked information needs to be added to that table in addition to the value of the master pointer.

At this point, some may object that our `av` pointer still points to the location in memory were the data was last seen. It would be silly to add a pointer to `unlock` just so that the function would clear the pointer (thus guaranteeing a crash if it is dereferenced. Furthermore, this goes somewhat against the C philosophy, which considers that it is not really worth it to try to make APIs "idiot-proof." This is even almost viewed as a way to challenge/encourage Nature into producing a more efficient idiot.

A better solution would be to limit the scope of `av` to a small block of code:

```
IntHandle a = (IntHandle) makeHandle(1000, sizeof(int));
...
{
    lock(a);
    IntPointer av = *a;
    for (int k=0; k<1000; k++) {
        // do things with av[i]
    }
    unlock(a);
}
// av not defined anymore
```

4.2 What to implement

You are going to define the `Handle` data type and add the following functions to your runtime memory manager:

- `void lock(Handle h),`
- `void unlock(Handle h).`
- a heap defragmentation function that gets called when the new command `defrag` is encountered in the list of arguments.

In addition, at the end of your program's execution, the memory manager should print out a warning if some handles are still locked.

Next, you are going to revise your main program to work with the new handle-based API and add support for the "defragment" command in your interpretation of the arguments. Your revised data files (lists of memory management commands) should include some "defragment" commands.

4.3 Extra credit (8 points)

As I was `typedef`-ying my `IntHandle` and `DoubleHandle` types, many of you were probably thinking "What about templates?" (or, at least, I hope so). Indeed, C++ templates seem well suited to take care of this kind of problem: Instead of doing C-style polymorphism through the use of `void*` pointers, use C++-style compile time polymorphism (templates). This assignment was originally to be completed in C, and `void*` was the only option. Now that we program in C++, this opens up the option of a template-based solution, for extra credit.

Design and implement a template-based revision of the `Handle` data type

5 What to submit

5.1 The pieces

All your work should be organized inside a folder named `Prog04`. Inside this folder you should place:

- Your report;
- A folder containing the html documentation produced by Doxygen (you remembered that you were supposed to write Doxygen-style comments for your code, didn't you?);
- A folder named `Version1` containing all the source and header files required to build the program;
- A folder named `Version2` containing all the source and header files required to build the program, the bash script, and a couple of examples of input files containing a list of commands for the runtime memory manager.
- Your script `script04.sh`

5.2 Grading

- Quality of C++ code: 15%
- Quality of `bash` script: 5%
- Execution: 55%
  - Version 1: 20%
  - Version 2: 20%
  - bash script: 15%
- Folder organization: 10%
- Report: 15%