Assignment 1

CIS3110/Operating SystemsMarking Scheme/W25

updated: January 4, 2025

CODING ASSIGNMENT

1 Evaluating Memory Management Algorithms

As we will learn in this course, the way in which free parts of a divisible resource like memory is managed can have a significant effect on how efficiently it is used.

We will examine this using a memory management experiment. You are to write a C program to implement the memory management scheme described below, and submit both the code and an analysis of your code based on the testing as described in this assignment.

A starter program consisting of a `main()` function is provided for you. This is provided so that the command-line for all of the assignments is consistent, to assist in marking. Be sure to use this code as the starting point for in your work.

In addition there are a number of example programs available in the CIS*3110/Operating Systems course directory (`/home/courses/cis3110`). You may find the "`memdbg_map.c`" tool useful as described below.

As with the use of any third-party resource in your academic writing, you must note the use of this (or any other) code in the writeup of your assignment, in order to avoid plagiarism (representing the work of another as your own).

Citing program code is always somewhat more confusing than citing a book or paper, but the same principles apply: indicate the author, the date, the name of the entity being cited, and the source.

For the the `memdbg_map` code in the course directory, this would be:

- author: Hamilton-Wright, Andrew
- name: Memory map block inspection
- year: 2011
- source: Operating Systems Course Directory, School of Computer Science, University of Guleph

You can format these in a reference document or comment as you wish, however it is best to follow an established standard (e.g. APA format for bibliographies.)

2 Memory allocation in C

In general, memory allocation in C works by having access to a large block of memory (a set of consecutive bytes), and giving back a portion of that block, identified by a starting address and a size.

On a virtual memory system we generally have access to a dynamically growing heap while in embedded systems (IoT devices, cars, *etc*) there is a fixed total memory block available, and all allocations have to come from that block.

Each allocation is satisfied by "carving out" an unassigned portion of the fixed block. You can envision this as much like cutting slices off of a loaf (of bread, cheese, etc.) to satisfy various requests.

Memory is, of course, constrained in size – that is, the total available block of memory can be envisioned as a fixed size (we will talk about the finer points of this in class, but for now, assume that there is one fixed size block). This means that one cannot keep carving off new allocations forever, as memory would run out.

The good news is, that while current programming styles demand hundreds, or even thousands, of allocations, they generally do not require use of each allocation forever. Put another way, allocation are requested, and then a short time later, released, or given back. This implies that an allocation can be used again to satisfy another request after it has been released by the client program.

You are already familiar with the allocation and release functions in C – these are `malloc()` and `free()`. In C++ these are performed using the keywords `new` and `delete`. The Java language hides the release operation from the user, so the releases occur within the garbage collector, but they still occur.

The bad news is that programs usually don't give back the memory in anything like the same order it was allocated. If memory releases happened in the same (or reverse) order of allocation, we would simply need to maintain a queue (or stack) to figure out how to put the pieces back together. Instead, we need a data structure that can be edited in the middle, as well as at both ends – frequently, this is implemented as a linked list.

If one simply uses a list (linked or otherwise) to keep track of "allocated" and "free" chunks of the block, then the `free()` function would mean changing the state of one of the chunks from "allocated" to "free". Any "free" chunk of a large enough size could then be given out again later, and changed to "allocated".

The first tricky bit is that the overhead for your list should come out of the chunk also. If you have a "next" pointer as part of your chunk management, for instance, then the memory for the next pointer should come out of the chunk allocation, so you will have to add on the extra overhead for this management to the size requested.

Another tricky bit is the size. Inconveniently for us, programs also usually don't ask for exactly the same size pieces again, so the question arises: how does one satisfy a request, knowing that there will likely not be a chunk of exactly the right size? Determining what happens for different answers to this question is the intent of this assignment.

A final wrinkle is that if there are two "free" chunks that are side-by-side (i.e.; contiguous in memory), then we really want to consider this to be one "free" chunk the size of both of these, merged together. This is a tricky bit of programming, so this assignment is structured to let you pursue most of the marks before you even attempt adding merging to your code.

Assume that when you start that there is one chunk, and that it fills the whole available memory region – your code will then carve it up from there.

3 Allocation Chunk Discovery Strategies

We will explore what happens if we use the following strategies to determine which chunk to use for a given allocation:

- **first**: this strategy uses the first chunk allocated that is free, and has a size equal to or greater than the requested size. If it is larger than the requested size, it is split in two, and the remaining portion remains as a free chunk that may be allocated later
- **best**: this strategy finds the chunk that will produce the least waste (minimizing the number of additional bytes beyond those that are required). As in "first", any additional bytes are split off, and treated as unallocated
- **worst**: similar to best, but maximizing, not minimizing. additional bytes

We will discuss the theory behind these algorithms in class in a few weeks, but this assignment will let you gather some empirical data yourselves (putting the "Science" in Computer Science).

4 Programming Task

Copy the starter code from the course directory into the location where you plan to work:

```
cp -r /home/courses/cis3110/assignments/CIS3110-W25-A1-code .
```

For this assignment, you are to write a C program (using the `main()` supplied in `allocator_main.c`) that will take a series of allocations and releases provided in a file, and print out whether each allocation can be satisfied. At the

end of the run, a summary should be printed indicating how many bytes are allocated, how many bytes are free, and a list of each "chunk" and its state (free or allocated). The summary should be preceded by the word "SUMMARY" and should then have the format shown in the example (bytes allocated, bytes free, and then a list of all of the chunks).

4.1 Input Data Format

Parsing code has been supplied for you to read files of allocations and deallocations.

4.1.1 Allocations

Allocation lines will begin with the letter A, and then have an *ID*, a *size* and an optional "paint character" indicated separated by colons (:). Sizes can be given in hexadecimal or in decimal. Spacing is ignored. Examples:

- A:11:1231
- A:12:0x0200:+

In the first case, when the paint character is not given, a default value of "~" is used.

It is an error to re-use the same ID for more than one allocations. You can assume that our testing files will not do this.

4.1.2 Deallocations

Deallocation lines will begin with the letter F for "free" or (R, for "release"), and then have an *ID*. No size is required, as the *ID* should let you locate the allocation. Spacing is again ignored. Examples:

- F:1
- R:2

It is an error (of course) to attempt to deallocate the same ID more than once.

You must however handle a request to deallocate an ID that is not there – this may simply refer to an allocation request that could not be honoured because memory ran out. If your code encounters this is is fine to report it at the time, but do not stop processing.

4.2 Sample Run

A sample run of the program might look like this:

```
$ ./allocator -m 4096 -s first smalltest.txt
alloc 1024 bytes : SUCCESS - return location 0
alloc 2048 bytes : SUCCESS - return location 1024
alloc 2048 bytes : FAIL
alloc 512 bytes : SUCCESS - return location 3072
free location 1024
free location 0
alloc 4096 bytes : FAIL
alloc 3000 bytes : SUCCESS - return location 0
SUMMARY:
3512 bytes allocated
584 bytes free
4 allocation chunks:
chunk  0 location    0:3000 bytes - allocated
chunk  1 location 3000:72 bytes - free
chunk  2 location 3072:512 bytes - allocated
chunk  3 location 3584:512 bytes - free
```

An astute reader will note that the memory taken up by any indexing required to track the chunks (i.e.; the linked list or other structure) has not been included in this. Again, this makes things trickier, and so assignments are accepted without adding this additional code (see marking scheme below).

4.3 Debugging Tool

A "memory dump" utility has been provided for you in the course directory. This is called "`memdbg_map.c`" and it prints out a specified region of memory in the same format that `hexdump(1)` uses when run with the `-C` option.. (That is, it prints the requested block of memory, 16 bytes per output line – first printing the values in hexadecimal, and then printing the same values as ASCII characters if printable, and as a '`.`' if not printable.)

If you want to print 192 bytes to standard output starting from the address stored in the variable `my_allocation`, and indent each line by three spaces (to set it off from the rest of your program output) then you could call it like this:

```
memdbg_dump_map(stdout, NULL, my_allocation, 192, 3);
```

Passing NULL here will provide the absolute memory addresses along the left margin. If you instead want to see the offsets relative to the `my_allocation` pointer, then simply use this form instead:

```
memdbg_dump_map(stdout, NULL, my_allocation, 192, 3);
```

4.3.1 Note on Memory Access

Note that all the regular rules apply about memory access – in particular, if you provide a large number as the `nBytesToPrint` value and this causes the dump tool to run past the end of allocated memory, then you will get a SEGV program crash. (There is nothing special that would allow this code to access memory that would otherwise be invalid.)

5 Deliverables

You will hand this assignment in electronically, in a single `.zip` or `.tar` file, using CourseLink. This file should contain:

1) all `.c` and `.h` files required to build your assignment
2) a `makefile` that will build your assignment when the command "`make`" is issued
3) an `OUTCOMES.md` file indicating which parts of the assignment is successfully completed, and what experiments were performed, in order to answer the following questions (based on your implementation and findings).

Include the "SUMMARY" information for at least two runs based on allocations of your own devising, and include these allocation files in your submission.

Use your code to answer these questions:

- which algorithm manages memory in the least overall number of chunks?
- which algorithm manages memory with the greatest number of successful allocations?

This file should be in either ASCII or UTF-8 text "markdown" format.

6 Marking Scheme

The following outlines the mark breakdown:

- Basic coding of a program that does allocations and reports them as described (35%)
- Merging adjacent free allocations upon release (15%)

- Managing indexing within the allocation block (20%)
- Creating experiments to explore different allocation patterns (15%)
- Final report, explaining your code and your findings (15%)

7 Suggested Approach

A good first step would be to get the basic program working, without trying to manage the the indexing within the allocation blocks, and without merging.

At that point, you can run experiments. If you then wish to tackle merging, or internal management, first **save your code** as is, and then work on the additional functionality. You can update your experimentation based on your new code, or you can (with a clear explanation) submit the report on an earlier version of your code, and include a more full featured version.

If you like, you can even **hand in an early working version** and then later update that in CourseLink. We will simply mark whatever is the last version submitted before the assignment deadline.

8 Submitting your Assignment

Use `tar(1)` to bundle up your code, makefile OUTCOMES.md file and your test run files. See the `make tar` rule in the supplied makefile for this – please modify if it does not capture all of your files.

Before submitting, **UNPACK** your archive in a new directory to make sure that you are in fact submitting everything you want to submit, and that everything works. If your code doesn't build based on what you submit, there will not be a chance to resubmit for this foolish error.