CS24 Assignment 3

The files for this assignment are provided in the archive **`cs24hw3.tgz`** on the course website. The archive contains a top-level directory **`cs24hw3`**, in which all other files are contained. Download and unpack the archive, then complete your work in this **`cs24hw3`** directory, being careful to put your answers in the locations specified in the assignment.

When you have finished the assignment, you can re-archive this directory into a tarball:

> *(From the directory containing **`cs24hw3`**; replace **`username`** with your username.)*
> **`tar -czvf cs24hw3-username.tgz cs24hw3`**

Then, submit the resulting file through the course website.

As mentioned before, failure to follow the specified filenames, or submitting an otherwise wonky tarball, will result in point deductions. Always make things easy for your grader. ☺

Important Collaboration Note!

In this assignment you will have the opportunity to write a simple heap allocator. **You may not consult any existing heap allocator implementations in the creation of your work.** For example, both the CS:APP book and the K&R C book contain example implementations of a heap allocator, but you may not refer to them at all in the implementation of your own allocator.

This should not be a problem for you. You are not required to write a very sophisticated allocator in this assignment, and the fundamental concepts are not very complicated. Most of the difficulty lies in eliminating the bugs, anyway. As long as you follow the steps in the assignment, and you methodically test and debug your work as you go, you should have no problems completing the required functionality without referencing existing implementations.

Arrays, Pointer Arithmetic, and Structs (10 points)

Pointer casting and pointer arithmetic can be bewildering at first, but it is a feature of C that is heavily utilized in low-level code, such as the memory allocator you will be implementing this week. This section will give you some basic practice with arrays, pointer arithmetic, and structs. Assume all code is compiled and run on a typical 64-bit Linux platform, where **`int`**s are 4 bytes, **`char`**s are 1 byte, and pointers are 8 bytes.

Put your answers in the file **`ptrmath/answers.txt`**. *Scoring: part 1 is 4 points, parts 2 and 3 are 3 points each.*

1. Here is some C code:

```
char *p = malloc(16);
bzero(p, 16);          /* Sets all bytes to 0. */

*((int *) (p + 3)) = 0xDEADBEEF;

*(((int *) p) + 3) = 0xCAFEBABE;
```

   Without compiling or running this code, specify the value of each byte of the 16-byte region

pointed to by **p**. (Don't forget that Intel x86 processors are little-endian when it comes to manipulating multi-byte values.)

Here is a C struct declaration:

```
typedef struct entry {
    int id;
    char code[5];
    struct entry* next;
} entry;
```

Feel free to use the compiler (including its assembly-code output) to understand the details of the above struct, in answering the following two questions.

2. Specify the offset and size of each member in the struct, based on the rule that **gcc** follows as described in class. What is the total size of the struct in memory? How many bytes are unused?

3. Here is some C code using the previous struct declaration:

```
entry *arr = malloc(10 * sizeof(entry));

char *p = (char *) arr;  /* arr and p hold the same address */

arr[4].code[3] = 'A';

p[offset] = 'A';
```

What value of **offset** will cause the last two lines of code to write the same byte? Explain your answer.

Ben Bitdiddle Strikes Again (30 points)

Run-length encoding (RLE) is a very simple compression mechanism in which runs of the same value are compressed into a pair of values: the repeated value, and the number of times that the value appeared.

For example, given the following data: "AAAAAIIIIEEEEEEE!!!"
This would be compressed into the sequence: 5,A,4,I,7,E,3,! (Note the counts come first.)

While this is a very simplistic compression scheme, it works extremely well on simple data such as icons and monochrome images; in fact, fax machines use a variant of RLE to compress documents that are being faxed.

In the **rle** directory you will find the sources to build two different utility programs, **rlenc** and **rldec**. The **rlenc** program takes an input file and creates a run-length-encoded version of the input file, and the **rldec** program does the opposite of this. These two programs use a very simple version of RLE, where the raw input is an array of unsigned char values, and the encoded output is an array of [*count*][*value*] pairs, where both the *count* and *value* are also unsigned char values. This means that if we had, for example, a string of 377 "A" characters, this would be encoded as 255 "A" characters, then 122 "A" characters, since the value 377 doesn't fit in an 8-bit unsigned char. The output would contain [0xFF 0x41 0x7A 0x41]. ("A" has the ASCII value 0x41, or 65 in base-10.)

The **rldec** program relies on an x86-64 assembly function called **rl_decode**, which conforms to the System V AMD64 ABI calling convention, and has a C signature like this:

```
unsigned char * rl_decode(unsigned char *input_data,
                          int input_length,
                          int *output_length);
```

The function takes a buffer of RLE-compressed data and the length of the buffer (the first two arguments). The function uses **malloc()** to allocate a buffer to hold the decoded data, and the size of the decoded result is returned to the caller via the *output parameter* **decoded_length**. The buffer of decoded data itself is returned to the caller. You can see how this function is used in the **rldec.c** source file.

The only problem is that the **rl_decode()** function was quickly coded up by your highly unreliable colleague Ben Bitdiddle[1] – and then he vanished on a long vacation! Now it has fallen to you to get the **rl_decode()** function working properly before the next big demo.

For this problem you will receive points for properly identifying and fixing the bugs in rl_decode(); therefore, you must record what bugs you found and fixed in the file **rle/bugs.txt**. You must describe every defect you found in the code, along with a brief description of why it is a defect, and how you went about resolving it. In your description, don't just repeat what the code says; explain the conceptual errors that Ben Bitdiddle made in implementing the various operations.

To simplify your bug-hunt, there is also a program **test_rldec** which will help you in your quest to find all bugs in **rl_decode()**. (Alyssa P. Hacker saw this train-wreck coming, and coded it up for you.) When you are done you should also be able to RLE-encode and then decode the **.bmp** files included in the archive, and get the same results as the input. (See **README.txt** for details.) Note that you may not have fixed all bugs until you can successfully complete this step as well!

Hints:

There are four bugs in the **rl_decode()** function, with varying characteristics. Some are definitely subtler than others, but all four must be fixed before the entire test suite will pass.

You can use whatever tools or techniques you want to identify the issues; if you can simply read through the code and find them, good for you! However, most people will benefit from using **gdb**, and this is the approach that we will encourage you to pursue. Hence, all source files are built with the **-g** option to include debug symbols.

Note that when you fix some of these bugs, you may cause other issues to manifest in more dramatic ways. If you fix a bug and then the tests start to crash, don't assume that you necessarily did something wrong! You should carefully check your work, but you may in fact be exposing another defect in the function's implementation.

Finally, not all of the bugs require a single-line fix…

[1] https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs - Characters

Explicit Heap Allocator (50 points)

The files for this problem are located in the **`myalloc`** directory of the archive. Copy the allocation and deallocation routines in **`unacceptable_myalloc.c`** to **`myalloc.c`** and revise them to implement a simple heap memory allocator. Here are additional requirements and guidelines:

- Your allocator must support deallocation, and it must also coalesce adjacent free blocks when a memory block is deallocated. You could use boundary tags, or keep block pointers in sorted order, or some other strategy of your own devising, to implement this.

- Your allocator should follow some placement strategy (e.g. first-fit, best-fit) to limit memory fragmentation. In your comments, document the strategy you choose, and the kinds of scenarios where this strategy will be good or bad.

- If your allocator requires statically declared, fixed-size state (e.g. a global variable specifying where the last free block was found, in order to implement a next-fit allocation strategy), this is fine. However, any dynamically allocated/managed state needed by your allocator must be located and maintained within the memory pool. In other words, you can't use **`malloc()`**, etc. to implement your allocator!

- You should make reasonable effort to make your allocator efficient. Operations that are linear-time in the number of blocks are acceptable, but more credit will be given for faster implementations, e.g. constant-time deallocation. Similarly, points will be deducted for very slow implementations.

- Your allocator does not have to consider data alignment issues at all.

- **In your comments, explain the time complexity of each operation (e.g. allocation, deallocation, coalescing adjacent free blocks, etc.) in your allocator.**

You may assume that the program using your allocator is well-behaved: the program always calls **`myfree()`** with an address returned by **`myalloc()`**, and **`myfree()`** is called at most once on an address returned by **`myalloc()`**.

Your completed allocator should be contained in `myalloc.c`. As stated earlier, comments in your code should explain your strategy for minimizing fragmentation and should identify the time complexity of each operation.

Allocator Testing

We provide the program **`testmyalloc.c`**, which tests your allocation routine and reports the memory utilization achieved as a percentage value between 0.0 and 1.0 (0% to 100%). Note that allocators with poor memory utilization will also be penalized. You shouldn't have much difficulty achieving a memory utilization of at least 60% (0.6 as reported by the tester). If your utilization is unusually low, your coalesce code may not be working properly.

Note that the memory tester will likely perform operations too complex to help you to debug your allocator. Because of that, we also provide the program **`simpletest.c`** that you can use to help debug your allocator in simple allocation scenarios. This file will not be graded; change it as you see fit. Set breakpoints in your **`myalloc()`** / **`myfree()`** code to verify they work properly.

Implementation Suggestions

You should feel free to implement this allocator in any way you wish, but a very simple approach would be to give each block a 32-bit **`int`** header that stores the size of the block. A negative value would indicate that the block is allocated, and a positive value would indicate that the block is free. This allows you to maintain an implicit free-list, but several operations are quite slow in this approach. According to the scoring guidelines below, a correct and properly documented version of this implementation would be worth 40/50 points.

Other features can be added to this implementation relatively easily to increase your score. See the scoring guidelines below for suggestions. (For example, you could implement a best-fit strategy, constant-time coalescing, and fill out the HW3 feedback survey, and that's 51/50 points.)

If you need help with the pointer arithmetic for this problem, see the appendix at the end of this assignment.

Do not write the entire allocator and then start trying to debug it! As indicated in the Allocator Testing section, the tester is very complex, and debugging will be extremely difficult if you try to write the entire allocator before starting to debug it. Rather, you should use a step-by-step approach, like the following:

1) Choose a representation for blocks within the memory pool (e.g. header, or header + footer, etc.), and clearly document this representation so that it's easy to refer to. Next, implement your **`init_myalloc()`** function to set up the memory pool in the way you have documented. Inside of **`myalloc()`**, don't worry about splitting the block – just mark it as allocated. See if this works correctly. (Don't even think about freeing yet; just leave it as a no-op.)

2) After getting step 1 working, add block-splitting to your implementation of **`myalloc()`**. (In other words, if the free block is much larger than the allocation size, split it into two blocks.) See if this works correctly.

   This might also be a good point to implement some verification code to ensure your heap is managed correctly. You can traverse all blocks in your heap, computing the sum of allocated and free space; this sum should exactly match the pool size. Additionally, the last block should end exactly where the memory pool ends. These are simple checks that can be implemented in a sanity-check function. Then, call the sanity-check function before and after major operations on the heap, so you can identify exactly when and where your code mangles the heap.

3) After step 2 works, add a placement strategy to your allocator: traverse the sequence of blocks, and choose a good block to satisfy the allocation request. You will need to make sure your block-splitting code still works, after you make these changes. Again, a sanity-check function would help you immensely in identifying bugs.

Note that for each step, you are adding a limited amount of code, then you are testing and debugging it. Do not go on to the next step until you are sure the code for each step is working properly. This way, each time you add more functionality, any bugs will likely be in a very limited region of your code.

Once you have completed the previous three steps, you should be pretty solid on allocation. The tester will likely report a terrible memory utilization value, but at least it should work. Now you

can start thinking about deallocation:

4) When **`myfree()`** is called, simply mark the block as available; don't coalesce with adjacent regions at this point. See if your program works. (You may uncover subtle bugs in your **`myalloc()`** code once you add this.) At this point, your allocator will begin having better memory utilization results.

5) After step 4 works, add code to coalesce with free blocks *after* the just-freed block. (This is usually the easier case to handle.) Again, make sure your code still works.

6) Finally, add in coalescing with free blocks that come *before* the just-freed block.

(Alternatively, you could implement all coalescing as a single pass through all blocks in the memory pool, instead of having two cases to worry about. This tends to be the easiest approach.)

If you follow this kind of step-by-step approach, it should make writing and debugging the allocator much easier. Also, **GDB is your friend!** GDB will tell you exactly where your program is seg-faulting, and it will allow you to examine your program's state as it executes.

Scoring Guidelines

For this problem, 20 points are devoted to correctness and memory utilization, as measured by the provided testing program.

The other 30 points of this problem are devoted to the implementation details and documentation of your allocator. A simple approach like the first one outlined in the "Suggested Approaches" section will receive 20 of these points, as long as it is cleanly and correctly written, follows good coding style, and includes the explanatory comments specified above.

Additional points can be earned by making your allocator more advanced, up to a maximum of 50 points on this problem:

- +3 points for a best-fit strategy instead of a first-fit or next-fit strategy
- +5 points for constant-time deallocation
- +5 points for an explicit free-list instead of an implicit free-list
- Similar extra credit will be granted for other advanced techniques used in your allocator.

Note that your implementation must also be clean and well documented to earn the points specified above.

Explicit-Allocator Robustness (10 points)

Answer the following questions in a file **`myalloc/errors.txt`**

(a) Describe what your allocator does in each of the following error cases. What happens if a programmer calls **`myfree()`** on:

- An address that did not originate from **`myalloc()`**?
- An address that is contained within a memory block returned by **`myalloc()`**?

For example: `a = myalloc(10); myfree(a + 5);`

- An address that has already been freed?

  For example: `a = myalloc(10); myfree(a); myfree(a);`

(b) Describe how to implement an allocator that would be more robust to the kinds of abuses suggested in the previous question. By "more robust," we mean that the program should flag the invalid use and then terminate, identifying the offending `myfree()` call.

You don't have to describe "every minute implementation detail," but your discussion should be detailed enough that you could give your allocator code and the descriptions to a programmer and they could implement the changes you describe without any problems. Ambiguity will be penalized.

Note: If your current allocator already properly handles one of the specified cases, simply refer back to your explanation in part (a).

Appendix: C Pointer Arithmetic

If your allocator follows the traditional approach of including a header and/or footer on memory blocks, you will have to solve the problem of taking a header-pointer and creating a payload-pointer from it, and also the inverse: taking a payload-pointer and creating a header-pointer from it.

As discussed in class, if you have a typed pointer, you can perform *pointer arithmetic* with it to move the pointer forward or backward in memory, but the type controls how far you move. For example, with an **`int *p`** (a pointer to an **`int`** value), **`p + 1`** moves the pointer forward one **`int`** (4 bytes), not one byte. Similarly, if you have a struct for your memory-block headers, e.g.

```
struct header {
    /* Negative size means the block is allocated, */
    /* positive size means the block is available. */
    int size;
};
```

If you have a **`header *h`** variable, the expression **`h + 1`** will move the pointer forward by the number of bytes in a **`header`** struct, or **`sizeof(header)`** bytes. The one exception is **`void *`** pointers, also known as *untyped* pointers: if you have a **`void *p`** variable, **`p + 1`** moves forward one byte, because the pointer has no type. *(Note: This is not generally true, but it is true for gcc on x86.)*

How much C adjusts a pointer always depends on the pointer's type, so if we need to change how this works, we simply cast the pointer to a different type. For example, your **`myalloc()`** function might start out with a **`header *h`** pointing to the first memory block. To move to the next block, you need to skip over the first header, plus the actual memory block's payload. Thus, you could do something like this: **`h = (header *) ((void *) h + h->size);`** (This of course assumes that the **`size`** value also includes the header size; if you need to add in the header size separately, you could add a **`sizeof(header)`** term to this expression.)

Similarly, if you have a **`void *p`** value that points to the start of the payload, and you need to get back to the start of the header for that block, you could simply do this:

```
header *h = (header *) p - 1;
```

(This code casts **`p`** to be of type header *, then subtracts 1 header from the pointer.)

Or this (leave **`p`** as a **`void *`**, subtract the size of a **`header`** from it, and then cast it to **`header *`**):

```
header *h = (header *) (p - sizeof(header));
```