CS107 Final Exam

Question Booklet

CS107 Spring 2019 – Instructor: Nick Troccoli

You may not use any internet devices. You will be graded on functionality – but good style saves time and helps graders understand what you were attempting. You have 180 minutes. We hope this exam is an exciting journey.

Note: DO NOT WRITE in this booklet. Only work in the answer booklet will be graded.

1) Floats **25 Points/180 Total**

Note: you may need to scroll to fully view blocks of code.

While generally useful for high-precision computation, in areas such as machine learning where precision is less important than runtime, 32-bit floating point numbers are unnecessarily costly due to their size. For this reason, there is active research into smaller floating point representations that still provide sufficient precision.

One such example, `FP16`, is structured the same as an IEEE float, but with 1 sign bit, 5 exponent bits, and 10 mantissa bits, and an exponent bias of 15 (meaning an exponent of -14 for denormalized floats). Use this format to answer the following questions below.

A) What is the corresponding (simplified) decimal number for the FP16 float `0 10011 100000000` ?

B) What is the corresponding number in scientific notation (e.g. x * 2^y) for the FP16 float `0 00000 1100000000` ?

C) One of your colleagues whips up a short program with regular 32-bit IEEE floats. It is meant as a long-running program that crunches data - their intention is to leave it running for an extremely long period of time (e.g. days), and Ctl-C when they want to stop the program. Here is the code they wrote:

```
int main(int argc, char *argv[]) {
    float counter = 0;
    while (true) {
        do_calculation();
        counter++;
        printf("%f calculations completed\n", counter);
    }

    return 0;
}
```

Initially the program print statement for the counter seems to work properly. However, after leaving the program running for an extremely long time, the counter seems to get stuck at an extremely large number (that is not infinity) and no longer increment! How could this be? In 3 sentences or less, explain to your colleague why the program output works correctly at the start but stops working correctly after a long period of time.

D) Regardless of bitwidth, floats exhibit strange behaviors when performing arithmetic. What is the output of the following program, which uses regular IEEE 32-bit floats?

```
void print_equality(float x, float y) {
    printf("%s\n", x == y ? "true" : "false");
}

int main(int argc, char **argv) {
    float a = 5.5;
    float b = 1.5;
    float c = 1 << 31;
    float d = -c;
    float inf = +INFINITY;  // floating point +infinity
    print_equality(a + b, 7);
    print_equality(c - b, c + b);
    print_equality(a + (c + d), a + c + d);
    return 0;
}
```

2) Generics **45 Points/180 Total**

Note: you may need to scroll to fully view blocks of code.

While writing a program that manipulates arrays of different types of data, you decide it would be useful to implement a few generic helper functions to make working with resizable arrays a bit easier. Implement the following helper functions for generic arrays. **When using `memcpy` / `memmove` , you should be sure to choose between them appropriately.**

A) Implement the generic array `remove_at` function. The function should remove the element at the specified index from the provided array and copy it to the location specified by the caller. For example, if the function is passed an array of ints `[5, 10, 7, 4]` and the caller wishes to remove the element at index 2, the resulting array should be `[5, 10, 4]` and the function would copy 7 to the destination specified by the caller. A generic `remove_at` function would therefore have the following signature:

```
void remove_at(void *base, size_t nelems, size_t elem_size_bytes,
               size_t remove_index, void *dest);
```

It takes in the following parameters:

- `base` : a pointer to the first element in the array
- `nelems` : the number of elements in the provided array
- `elem_size_bytes` : the size of a single provided array element, in bytes
- `remove_index` : the index of the element to remove from the array. You can assume this index is within the bounds of the array.
- `dest` : a pointer to the location where the removed element should be copied to.

B) Implement the generic array `set_at` function. The function should set the element at the specified index of the specified array to the value pointed to by the `elem` parameter. If the index is outside the bounds of the array, you should resize the array just enough to fit an element at that index, and then set the element at the index to the value pointed to by the `elem` parameter. A generic `set_at` function would therefore have the following signature:

```
void set_at(void **base, size_t *p_nelems, size_t elem_size_bytes,
            void *elem, size_t index);
```

It takes in the following parameters:

- `base` : a pointer **to the address of** the first element in the array. If the location of the array in memory changes, you should update the value at the address stored in `base` to be the new location of the array.
- `p_nelems` : a pointer to the number of elements in the array. This number should be updated if the array length is changed.
- `elem_size_bytes` : the size of a single provided array element, in bytes
- `elem` : a pointer to the element value we should set the specified index to.
- `index` : the index to insert at in the provided array. If the index is outside the bounds of the array, the array should be resized just enough to fit the element at that index.

Here are some examples of using the function:

- In `[1, 3, 5, 7, 9]` , setting the value at index 2 to be 10 should modify the provided array to be `[1, 3, 10, 7, 9]` .
- In `[1, 3, 5, 7, 9]` , setting the value at index 5 to be 10 should modify the provided array to be `[1, 3, 5, 7, 9, 10]` .
- In `[1, 3]` , setting the value at index 5 to be 10 should modify the provided array to be `[1, 3, ?, ?, ?, 10]` (it does not matter what is stored at indexes 2-4).

C) Implement the generic array `order_pairs` function. The function should sort the elements in each pair in the specified array according to the provided comparison function. For example, if the function is passed an array of ints `[15, 10, 7, 4]` along with an int comparison function that orders ints in ascending order, the resulting array should be `[10, 15, 4, 7]`. Specifically, notice how the first two elements are now ordered in ascending order, as well as the second two elements. A generic `order_pairs` function would therefore have the following signature:

```
void order_pairs(void *base, size_t nelems, size_t elem_size_bytes,
                 int (*cmp_fn)(void *, void *));
```

It takes in the following parameters:

- `base` : a pointer to the first element in the array
- `nelems` : the number of elements in the provided array
- `elem_size_bytes` : the size of a single provided array element, in bytes
- `cmp_fn` : a function pointer to a comparison function in the format required by `qsort` and other generics functions. It accepts two parameters, both pointers to elements in the array, and returns a negative number if the first element parameter should come before the second, `0` if the element parameters should be at the same position, or a positive number if the first element parameter should come after the second.

Here are some examples of using the function:

- Ordering pairs in `[3, 1, 9, 5, 7]` with a comparison function that orders numbers in ascending order should modify the provided array to contain `[1, 3, 5, 9, 7]`.
- Ordering pairs in `[9]`, regardless of the comparison function, should do nothing.
- Ordering pairs in `[9, 9, 12, 15, 16, 5]` with a comparison function that orders numbers in descending order should modify the provided array to contain `[9, 9, 15, 12, 16, 5]`.

D) Implement the `compare_ints` function that could be used as a comparison function for `order_pairs` (or other generics functions like `qsort`) with an array of `int`s. It should order elements in **descending** order.

E) Given an existing int array `arr` which has length `length`, fill in the blanks for the parameters to `set_at` below to append `elem_to_append` onto `arr`.

```
int *arr = ...;
size_t length = ...;
int elem_to_append = ...;
set_at(___1___, ___2___, ___3___, ___4___, ___5___);
```

F) Given an existing int array `arr` which has length `length`, fill in the blanks for the parameters to `order_pairs` below to order the elements of each pair in `arr` in descending order. You may use your previously-defined comparison function.

```
int *arr = ...;
size_t length = ...;
order_pairs(___1___, ___2___, ___3___, ___4___);
```

3) Assembly **40 Points/180 Total**

Note: you may need to scroll to fully view blocks of code.

Consider the following code, generated by gcc with the usual -0g and other settings for this class, and use it to answer the following questions:

```
0000000000400546 <func2>:
  400546:   8d 04 7f                lea    (%rdi,%rdi,2),%eax
  400549:   8d 14 00                lea    (%rax,%rax,1),%edx
  40054c:   8d 42 03                lea    0x3(%rdx),%eax
  40054f:   85 d2                   test   %edx,%edx
  400551:   0f 49 c2                cmovns %edx,%eax
  400554:   c1 f8 02                sar    $0x2,%eax
  400557:   89 f1                   mov    %esi,%ecx
  400559:   d3 e0                   shl    %cl,%eax
  40055b:   c3                      retq

000000000040055c <func1>:
  40055c:   41 55                   push   %r13
  40055e:   41 54                   push   %r12
  400560:   55                      push   %rbp
  400561:   53                      push   %rbx
  400562:   48 83 ec 08             sub    $0x8,%rsp
  400566:   49 89 fc                mov    %rdi,%r12
  400569:   41 89 f5                mov    %esi,%r13d
  40056c:   bb 00 00 00 00          mov    $0x0,%ebx
  400571:   bd 04 00 00 00          mov    $0x4,%ebp
  400576:   eb 17                   jmp    40058f <func1+0x33>
  400578:   8d 53 01                lea    0x1(%rbx),%edx
  40057b:   89 d8                   mov    %ebx,%eax
  40057d:   41 8b 34 94             mov    (%r12,%rdx,4),%esi
  400581:   41 8b 3c 84             mov    (%r12,%rax,4),%edi
  400585:   e8 bc ff ff ff          callq  400546 <func2>
  40058a:   01 c5                   add    %eax,%ebp
  40058c:   83 c3 02                add    $0x2,%ebx
  40058f:   41 8d 45 ff             lea    -0x1(%r13),%eax
  400593:   39 c3                   cmp    %eax,%ebx
```

```
  400595:   72 e1                  jb     400578 <func1+0x1c>
  400597:   89 ea                  mov    %ebp,%edx
  400599:   be 44 06 40 00         mov    $0x400644,%esi
  40059e:   bf 01 00 00 00         mov    $0x1,%edi
  4005a3:   b8 00 00 00 00         mov    $0x0,%eax
  4005a8:   e8 83 fe ff ff         callq  400430 <__printf_chk@plt>
  4005ad:   48 83 c4 08            add    $0x8,%rsp
  4005b1:   5b                     pop    %rbx
  4005b2:   5d                     pop    %rbp
  4005b3:   41 5c                  pop    %r12
  4005b5:   41 5d                  pop    %r13
  4005b7:   c3                     retq
```

A) Fill in the C code below (also included in the answer area) so that it is consistent with the above x86-64 assembly. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this (this may mean slightly adjusting the syntax or style of your initial decoding guess to an equivalent version that fits). All constants of type signed/unsigned int must be written in decimal. Your C code should not include any casting.

```
int func2(int x, int y) {
    int z = _________;
    return _________ << _________;
}

void func1(int *arr, int count) {
    int z = _________;
    for (int i = _________; i < _________; i += _________) {
        _________ += func2(_________, _________);
    }

    printf("%d\n", _________);
}
```

B) The assembly code includes several “push” and “pop” instructions. What kind of registers are being pushed (and popped), and why is gcc required to push and pop them, given how those registers are used in the body of the function?

C) Cite a single line from the provided assembly above and explain how we can tell from that line that `arr` is a pointer to `ints` (as opposed to a pointer to another size type).

D) Explain briefly how, when `func1` calls `func2`, it is able to resume execution at the correct instruction in `func1` when `func2` finishes.

4) ATM Part 2 **25 Points/180 Total**

Note: you may need to scroll to fully view blocks of code.

The author of the `atm.c` program from assignment 6 has implemented login functionality in their banking system. However, there have been cases where, even without knowing the secret password, users are able to log in! Hearing about your success helping with their faulty ATM system, they're eager for your assistance.

They've narrowed down the issue to a single function, `login`, for which they have provided you with the source code. You can run the code on your local computer, but you don't have access to the `read_real_password` function, so you make that function up yourself for your testing. The `login` function is supposed to take in as a parameter the password the user entered to log in, and return true if the user should be logged in, or false otherwise. Their criteria for a valid login is the entered password must match the secret password, or if there is no secret password set, the user should always be logged in. The erroneous logins they have seen have all taken place with a non-empty password less than 8 characters long. How is this possible, they say?!

```
// Assumed to work properly
void read_real_password(char *buf) {
    strcpy(buf, "fakepw");
}

// Erroneous function
bool login(const char *user_password) {
    char real_password[8];
    read_real_password(real_password);

    char buffer[8];
    strcpy(buffer, user_password);

    return strcmp(real_password, buffer) == 0 || strlen(real_password
) == 0;
}
```

You start investigating with GDB, where you uncover the following information about the program state:

```
$ gdb login
Reading symbols from login...done.
(gdb) b 16
Breakpoint 1 at 0x40066c: file login.c, line 16.
(gdb) r "guess"
...
Breakpoint 1, login (user_password=0x7fffffffecf4 "guess") at login.c
:16
16      read_real_password(real_password);
(gdb) p &real_password
$1 = (char (*)[8]) 0x7fffffffe96c
(gdb) p &buffer
$2 = (char (*)[8]) 0x7fffffffe960
(gdb)
...
```

After combining this information with the code, you find the issue! Answer the following questions about this vulnerable program below.

A) Write a note to the developer with an example for how to be logged in to the system even with valid non-empty secret password. Explain why your example works.

B) Provide the developer with a 1-line code change to `login` to fix the issue. Explain why your fix resolves the issue.

5) Heap Allocators **45 Points/180 Total**

Note: you may need to scroll to fully view blocks of code.

After seeing the promising results of your heap allocator implementations on assignment 7, you decide to add a few additional features to make your allocator even more capable. In particular, you are eager for more functionality to improve utilization, and it would be nice to have a way to detect memory leaks in programs, similar to Valgrind.

The design you have been using for your heap allocator is an implicit free list allocator, and looks like the following:

- All requests are rounded up to a multiple of 8 bytes and all returned pointers are aligned to 16-byte boundaries. The minimum payload size is 8 bytes.
- Each block has an 8-byte header storing the payload size and whether it is allocated or free. Because block sizes must be multiples of 16 bytes, the rightmost (least significant) bit of the size is unused to store the size and is thus used to store whether a block is free (0) or allocated (1).

Below are the allocator's global variables, constants and type definitions:

```
#define FREE 0
#define ALLOCATED 1

static void *heap_start;
static size_t heap_size;
```

Each of the parts below has you implement functionality for this allocator design. For full credit, you can (and should) call other functions in this problem as needed rather than re-implementing logic.

A) Implement `is_used` . Given a pointer to a block's header, it should return true if the block is allocated, or false if the block is free.

```
bool is_used(size_t *header);
```

B) Implement `get_size`. Given a pointer to a block's header, it should return the size of that block, in bytes.

```
size_t get_size(size_t *header);
```

C) Implement `set_size`. Given a pointer to a block's header, and a new size in bytes, it should update the header to store the updated size. Its status (allocated or free) should not change.

```
void set_size(size_t *header, size_t new_size);
```

D) Implement `get_next_header`. Given a pointer to a block's header, it should return a pointer to the header of the right-adjacent block. If the specified header is the last header on the heap, the function should return NULL.

```
size_t *get_next_header(size_t *header);
```

Now that you've implemented several useful helper functions, you are eager to improve the utilization of your allocator. Up to now, you implemented coalesce-on-free, where a newly-freed block is coalesced if possible with its immediate right neighbor. You wish to add a `coalesce_all` function that could be called periodically, such as after every 50 allocations, that goes through the entire heap and coalesces all free blocks together as much as possible.

E) As a helper function for `coalesce_all`, first implement `coalesce_starting_at`. Given a pointer to a free block's header, it should coalesce as much as possible with its immediate right free neighbors.

```
void coalesce_starting_at(size_t *header);
```

F) Implement `coalesce_all`. It should iterate through the heap from start to finish, coalescing all free blocks.

```
void coalesce_all();
```

The second main feature you wish to implement is leak detection. Similar to Valgrind, it would be nice if you had a function that you could call at the end of your program's execution for debugging purposes which returned an array of the addresses of all payloads that were leaked (not yet freed). To implement this, a coworker of yours proposes a new design addition to your allocator; similar to how an explicit allocator builds a linked list of free blocks using pointers stored in free block payloads, they propose building a doubly-linked list of allocated blocks using pointers stored in block headers to efficiently traverse them. To do this, they enlarge the space for each header to contain, immediately following the size_t, a pointer to the payload of the previous allocated block in the list, and that immediately followed by a pointer to the payload of the next allocated block in the list.

G) One common operation for this functionality is, given a pointer to an allocated block's payload, to get a pointer to the next allocated block's payload in the linked list.

```
void *ptr = alloc_list_start_payload;
void *next_payload = ____________________________;
```

Fill in the blank above with an expression, using `ptr`, that gets a pointer to the payload of the next allocated block in the allocated list.

H) To the dismay of your style instincts, your coworker changes their mind and insists that instead you implement this operation using a single `memcpy` call instead. Fill in the blanks below using ptr where needed, that results in `next_payload` storing the address of the payload of the next allocated block in the allocated list.

```
void *ptr = alloc_list_start_payload;
void *next_payload;
memcpy(__________, __________, __________);
```