This exam is based on the final exam given in Fall 2018. The class was taught by Cynthia Lee. This was a 3-hour paper exam.

**Problem 1: Floating Point Representation (10pts)**

A normalized IEEE 32-bit float is stored as the following bit pattern:

\[
\begin{array}{c}
N & \quad \text{EEE} & \quad \text{EEEEEEEE} & \quad \text{SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
Problem 2: Memory Diagram (10pts)

For this problem, you will draw a memory diagram of the state of memory as it would exist at the end of the execution of this code:

```c
struct wakanda *shuri = malloc(2 * sizeof(struct wakanda));
shuri[0].king = 5;
shuri[0].army = malloc(8);
shuri[1].army = &(shuri[0].king);
int *panther = shuri[0].army;
shuri[1].king = shuri[0].king + 1;
shuri[0].army = shuri[0].army + 1;
*(shuri[0].army) = 7;
```

Instructions:

- Place each item in the appropriate segment of memory (stack, heap).
- Please write array index labels (0, 1, 2, ...) next to each box of an array, in addition to any applicable variable name label. (With the array index labels, it doesn’t matter if you draw your array with increasing index going up, down, or sideways.)
- Draw structs as a series of boxes, one box per struct field. (You may assume for this problem that no padding is added to structs.)
- Take care to have pointers clearly pointing to the correct part of an array.
- Leave boxes of uninitialized memory blank.
- NULL pointer is drawn as a slash through the box, and null character is drawn as '\0'.

Stack

Heap
Problem 3: Pointers and Generics (12pts)
Recall our generic swap function from class (reproduced below). It is used to make two values trade places in memory, and is commonly used in sorting arrays. There’s a right way to call this swap function in normal circumstances, but we’re asking you to use it a bit “creatively” to achieve particular results. Here are some important constraints:

- As shown below, the third argument to swap is the return value of sizeof. Complete it with the name of a standard type.
- Do not move/change any memory outside the boxes shown in the diagram.
- Casting pointers is ok.

```c
void swap(void *a, void *b, size_t sz) {
    char tmp[sz];
    memcpy(tmp, a, sz);
    memcpy(a, b, sz);
    memcpy(b, tmp, sz);
}
```

(a) (5pts) Complete the mixup1 function to create this before and after result.

Before:

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>'F'</td>
</tr>
<tr>
<td>'B'</td>
<td>'G'</td>
</tr>
<tr>
<td>'C'</td>
<td>'H'</td>
</tr>
<tr>
<td>'D'</td>
<td>'I'</td>
</tr>
<tr>
<td>'E'</td>
<td>'J'</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>'F'</td>
</tr>
<tr>
<td>'B'</td>
<td>'G'</td>
</tr>
<tr>
<td>'C'</td>
<td>'H'</td>
</tr>
<tr>
<td>'D'</td>
<td>'I'</td>
</tr>
<tr>
<td>'E'</td>
<td>'J'</td>
</tr>
</tbody>
</table>

```c
void mixup1(char *ptr1, char *ptr2) {
    swap(
        ,
        ,
        sizeof( ));
}
```
Now consider the following before and after diagram, where ptr1 and ptr2 are int *s:

**Before:**

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

**After:**

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

To help you think about how to solve this one, let's first look at the hexadecimal versions of the two new numbers that appear in the “after.”

(b) (1pt) Write -256 in hexadecimal (32-bit):

(c) (1pt) Write 255 in hexadecimal (32-bit):

(d) (5pts) With those values in mind, now complete the mixup2 function to create this before and after result. (Note: you do not need to be concerned with “endian-ness”, or byte order – any endianness will be accepted).

```c
void mixup2(int *ptr1, int *ptr2) {
    swap(__________________________,
         ____________________________);
    sizeof(______________________));
}
```
Problem 4: Assembly (23pts)
Consider the following code, generated by gcc with the usual -0g and other settings for this class:

```
0000000000400612 <vp>:  
  400612:    push  %rbp  
  400613:    push  %rbx  
  400614:    sub   $0x8,%rsp  
  400618:    mov   %edi,%ebp  
  40061a:    mov   %rsi,%rbx  
  40061d:    mov   %rsi,%rdi  
  400620:    callq  400490 <strlen@plt>  
  400625:    lea    (%rax,%rax,2),%edi  
  400628:    cmp    $0xffffffffef,%ebp  
  40062b:    jbe    400632 <vp+0x20>  
  40062d:    jmp    400652 <vp+0x40>  
  40062f:    shr    $0x2,%edi  
  400632:    cmp    $0x6,%edi  
  400635:    ja     40062f <vp+0x1d>  
  400637:    cmp    $0xff,%ebp  
  40063d:    jbe    400646 <vp+0x34>  
  40063f:    callq  4005f3 <pass_final_level>  
  400644:    jmp    40065c <vp+0x4a>  
  400646:    mov    $0x0,%eax  
  40064b:    callq  4005d6 <explode_bomb>  
  400650:    jmp    40065c <vp+0x4a>  
  400652:    mov    $0x0,%eax  
  400657:    callq  4005d6 <explode_bomb>  
  40065c:    lea    0x3(%rbx),%rax  
  400660:    add   $0x8,%rsp  
  400664:    pop    %rbx  
  400665:    pop    %rbp  
  400666:    retq
```
(a) (17pts) Fill in the C code below so that it is consistent with the above x86-64 code. 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. The signatures for two functions called in the code are provided for your reference.

```c
void explode_bomb();
void pass_final_level(unsigned int x);

char *vp(unsigned int aaron, char *burr) {
    // see part (c)
    unsigned int leslie = strlen(burr) * [197];
    if ([167]) {
        while (leslie > [184]) {
            // see part (c)
            [212] /= [229];
        }
    }
    if (aaron >= 256) {
    }
} else {
} else {
    explode_bomb();
}
return [283];
}
```
(b) (2pts) 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) (2pts) The C code includes a multiply and a divide (marked with comments “see part (c)”), but there is no multiply (imul) or divide instruction in the assembly code. Which instructions are used instead (name both), and why would gcc generate the assembly code this way?

(d) (2pts) Is it possible to provide a value for aaron that would allow this function to complete without calling explode_bomb? (Check box for one.)  □ YES  □ NO

If yes, give such a value for aaron. If no, explain the constraints that prevent it.
Problem 5: Heap Allocator (24pts)
You are writing code for an allocator that uses a block header and maintains an explicit free list. Implementation details of this allocator include:

- All requests are rounded up to a multiple of 16-bytes and all returned pointers are aligned to 16-byte boundaries. The minimum payload size is 16 bytes.
- The header is 16 bytes as:
  - a size_t (8 bytes) storing the payload size, expressed as a count of 8-byte words
  - an unsigned long (8 bytes), 1 if block is in-use, 0 if free
- The allocator maintains an explicit free list as a doubly-linked list stored in the payload. A global variable points to the payload of a free block (or NULL if there are no free blocks).
- The first 8 payload bytes of each free block store a “next” pointer to the payload of another free block. The last free block on the list stores NULL as its “next.”
- The next 8 payload bytes of each free block store a “previous” pointer to the payload of another free block. The first free block on the list stores NULL as its “previous.”

Here is an example heap after a few requests have been serviced (box sizes not to scale):

<table>
<thead>
<tr>
<th>0x20</th>
<th>0x28</th>
<th>0x30</th>
<th>0x50</th>
<th>0x58</th>
<th>0x60</th>
<th>0x68</th>
<th>0x70</th>
<th>0x78</th>
<th>0x80</th>
<th>0x90</th>
<th>0x98</th>
<th>0x100</th>
<th>0x108</th>
</tr>
</thead>
<tbody>
<tr>
<td>mwords</td>
<td>used</td>
<td></td>
<td>mwords</td>
<td>used</td>
<td></td>
<td>mwords</td>
<td>used</td>
<td></td>
<td>mwords</td>
<td>used</td>
<td></td>
<td>mwords</td>
<td>used</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td>2</td>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td>6</td>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This segment starts at address 0x20 and ends at 0x138 (0x137 is the last byte of the last block). Two blocks are in-use, two are free. The in-use payloads are shown shaded in gray. The free_list points to the payload at 0x100, and the payload at 0x100 stores a “next” pointer to the payload at 0x60, and the payload at 0x60 stores a “next” pointer of NULL. The “previous” pointers of the two free blocks are also set accordingly.

Below are the allocator’s global variables, constants, and type definitions (you may assume that the structs’ memory layouts are as shown and described above).

```c
struct Header { size_t nwords; unsigned long used; };  
struct Node   { struct Node *next; struct Node *prev; };  
#define HDRSIZE sizeof(struct Header)  
#define NODSIZE sizeof(struct Node)
static void *segment_start; // base address of heap segment
static void *segment_end;   // end address of heap segment
static void *free_list;     // pointer to payload of first free block
                          // (NULL if no free blocks)
```
(a) (4pts) Implement `get_neighbor`. Given a pointer to a block’s **header**, it returns a pointer to the **header** of the neighbor to the right, i.e., at the next higher address in the heap. If `hdr` has no right neighbor (i.e., it is the rightmost block in the heap segment), the function returns `NULL`.

```c
struct Header *get_neighbor(struct Header *hdr)
{
}
```

(b) (2pts) You consider implementing a corresponding `get_left_neighbor` (given a pointer to a block’s header, it would return a pointer to the header of the neighbor to the left), but soon realize that it while it would be possible, it would be much slower than getting the right neighbor. Briefly describe how is it possible (what actions would the function have to take), and why it would be slower in terms of Big-O cost.
(c) (3pts) As you think about your heap allocator design, you realize that in many places in your code, you will have a pointer to a block's payload, and you’ll need to get a pointer to the header of the same block. Implement a helper function to do this conversion.

```c
struct Header *pay_to_hdr(struct Node *payload)
{
}
```

Parts (d)-(f) concern helper functions that you plan to use in your `validate_heap` function. Specifically, you’d like to loop over every block in the entire heap and count how many free blocks you find, and then compare that count to the number of free blocks you encounter while traversing your explicit free list. In parts (d)-(f), you’ll write and analyze functions to do these two counts.

(d) (4pts) Write a function `count_free_inorder` that starts at the leftmost block of the heap and proceeds to the right neighbor, then its right neighbor, and so on to the end of the heap, counting how many free blocks it encounters. For full credit, you should use other functions in this problem as helper functions if/when appropriate. In particular, while you won’t be writing it yourself, you may assume there exists the following already-implemented function:

```c
struct Node *hdr_to_pay(struct Header *header);
```

```c
size_t count_free_inorder()
{
    size_t nfree = 0;
    for (struct Header *curr = ;
```
(e) (3pts) Now write a function that traverses the explicit free list and counts the number of free blocks it finds. Again, for full credit, you should use other functions in this problem as helper functions if/when appropriate.

```c
size_t count_free_list()
{
    size_t nfree = 0;
    for (struct Node *curr = ; ; ) {
        if ( == 0) // check if free - see part (f)
            nfree++;
    }
    return nfree;
}
```

(f) (2pts) Your coworker looks at your count_free_list and count_free_inorder functions above, and says that only one of them needs the “if” test (see the line marked “see part (f)” in each function; note that they both do need the line “nfree++;”). In other words, you could just cross out one of the lines marked “see part (f)” and the functionality would not change.

From which function could you safely remove the “if” test? (Check box for one.)

- [ ] count_free_inorder
- [ ] count_free_list

Explain why that function does NOT need the “if” test.


(g) (2pts) Your heap allocator does coalescing, that is, merging a pair of adjacent free blocks to create one big free block. The merge consumes the entire right block, including its header, adding it to the left block’s payload. This involves a few steps, but one of them is to update the header of the left block of the pair to reflect the new size. Write a helper function update_header that takes a pointer to the header of the left block of a pair of free blocks (assume you’ve already checked they’re both free) and updates its nwords field. Again, for full credit, you should use other functions in this problem as helper functions if/when appropriate.

```c
void update_header(struct Header *left)
{
    left->nwords +=
};
```

(h) (4pts) In operations such as malloc and coalesce, you need to remove a block from the free_list and repair the doubly-linked list structure so it reconnects around the removed link node. Write a helper function that performs this task. It takes a pointer to the payload of the block to be removed from the free_list. We have divided the work into four cases. You may not need to write code for each case. If nothing needs to be done for a particular case, just write a semicolon in the box for that case.

```c
void remove_node(struct Node *remove)
{
    if (remove->prev == NULL) {
    } else {
    }
    if (remove->next == NULL) {
    } else {
    }
}
```