CS107 Final Examination Solution

Problem 1: Serializing Strings [10 points]

```
char *serialize(struct string strings[], size_t len) {
   char *serialization = strdup("");
   size_t length = 0;
   for (size_t i = 0; i < len; i++) {
      serialization = realloc(serialization,
                              length + strings[i].length + 1);

      if (strings[i].length <= 15) {
         strcpy(serialization + length, strings[i].chars);
      } else {
         strncpy(serialization + length, strings[i].chars, 8);
         strcpy(serialization + length + 8,
                *(char **) &strings[i].chars[8]);
      }

      length += strings[i].length;
   }

   return serialization;
}
```

Problem 2: x86-64 and gcc Optimizations [20 points]

a) First, fill in the blanks below so that **`allspice`** is programmatically consistent with the unoptimized assembly you see on the right. Note that the C code is nonsense and should just be a faithful reverse engineering of the assembly. You may not typecast anything.

```
size_t allspice(char *mustard, char *cardamon[]) {
    size_t cinnamon = cardamon[0] - mustard;
    for (size_t i = cinnamon; i < 2 * cinnamon; i += cinnamon) {
        printf(mustard, &cardamon, &cinnamon);
        if ((~cinnamon & 0x7) == 0) break;
        cinnamon = allspice(cardamon[1], cardamon);
    }
    return cinnamon;
}
```

b) Note that both the unoptimized and optimized versions some caller-owned registers to the stack but then proceeds to modify **`%rsp`** without having pushed its value to the stack. However, you also know that **`%rsp`** is caller-owned as well, even though it's not being pushed to the stack. Why does, say, **`%rbp`** needs to be pushed to the stack while **`%rsp`** doesn't need to be?

The callee's only responsibility it to ensure that a caller-owned register is restored to its original value prior to exit should the callee change it during its own execution. As you see from the assembly, **`%rsp`** is demoted by several bytes to make space for local variables, but prior to any call to **`retq`**, it's promoted to its original value. There's no specific obligation to use **`pushq`** and **`popq`**.

c) It should be evident that the number of instructions emitted when **`-O2`** is used is much higher than the number emitted with **`-Og`**. Explain why the number of instructions executed by a typical call to **`allspice`** will, in practice, still be smaller.

The real win here is that the number of instructions within the loop has gone down dramatically. Compilers typically assume a loop will iterate many, many times, so if static instruction goes up but dynamic instruction count goes down, then that's still a substantial optimization.

```
0x1170 <+0>:   push   %r13
0x1172 <+2>:   push   %r12
0x1174 <+4>:   push   %rbp
0x1175 <+5>:   push   %rbx
0x1176 <+6>:   sub    $0x28,%rsp
0x117a <+10>:  mov    (%rsi),%rbx
0x117d <+13>:  mov    %rsi,0x8(%rsp)
0x1182 <+18>:  sub    %rdi,%rbx
0x1185 <+21>:  lea    (%rbx,%rbx,1),%rax
0x1189 <+25>:  mov    %rbx,0x18(%rsp)
0x118e <+30>:  cmp    %rax,%rbx
0x1191 <+33>:  jae    0x11f0 <allspice+128>
0x1193 <+35>:  mov    %rdi,%rbp
0x1196 <+38>:  lea    0x18(%rsp),%r13
0x119b <+43>:  lea    0x8(%rsp),%r12
0x11a0 <+48>:  mov    %r12,%rdx
0x11a3 <+51>:  xor    %eax,%eax
0x11a5 <+53>:  mov    %r13,%rcx
0x11a8 <+56>:  mov    %rbp,%rsi
0x11ab <+59>:  mov    $0x1,%edi
0x11b0 <+64>:  callq  0x1050 <printf@plt>
0x11b5 <+69>:  mov    0x18(%rsp),%rax
0x11ba <+74>:  mov    %rax,%rdx
0x11bd <+77>:  not    %rdx
0x11c0 <+80>:  and    $0x7,%edx
0x11c3 <+83>:  je     0x11e4 <allspice+116>
0x11c5 <+85>:  mov    0x8(%rsp),%rsi
0x11ca <+90>:  mov    0x8(%rsi),%rdi
0x11ce <+94>:  callq  0x1170 <allspice>
0x11d3 <+99>:  add    %rax,%rbx
0x11d6 <+102>: lea    (%rax,%rax,1),%rdx
0x11da <+106>: mov    %rax,0x18(%rsp)
0x11df <+111>: cmp    %rbx,%rdx
0x11e2 <+114>: ja     0x11a0 <allspice+48>
0x11e4 <+116>: add    $0x28,%rsp
0x11e8 <+120>: pop    %rbx
0x11e9 <+121>: pop    %rbp
0x11ea <+122>: pop    %r12
0x11ec <+124>: pop    %r13
0x11ee <+126>: retq
0x11ef <+127>: nop
0x11f0 <+128>: add    $0x28,%rsp
0x11f4 <+132>: mov    %rbx,%rax
0x11f7 <+135>: pop    %rbx
0x11f8 <+136>: pop    %rbp
0x11f9 <+137>: pop    %r12
0x11fb <+139>: pop    %r13
0x11fd <+141>: retq
```

d) Notice that 5 of the final 6 instructions appear earlier, in the same order, at offsets **+116** through **+126**, and that the **mov %rbx, %rax** at offset **+132** is the only one not replicated. Knowing that you could, in theory, implement **allspice** directly in x86-64 yourself, by hand, explain how you might use most of what's presented to the right (which is a copy of what's on the previous page) while reordering a few instructions and updating one or more jump offsets so that only one copy of these five instructions is needed instead of two.

The mov instruction at +132 could be replicated to appear in between what's currently at +114 and +116. Then update the **jae** at +33 to jump to this replicated **mov** instruction and the update the **je** at +83 to jump to the instruction after this replicated move. You can then get rid of the **nop** orginally at +147 and everything beyond it.

e) At offset +51, you'll see an **xor** instruction. What line in the unoptimized version does that correspond to? In what sense is the **xor** alternative considered an optimization?

Any bit pattern exclusive-or'ed with itself generates a zero, so that **xor** instruction has the effect of zeroing out **%eax**. The unoptimized code achieves the same thing using a **mov $0, %eax** instruction. So, what's the optimization? That **xor** instruction is encoded in just two bytes, and the **mov** instruction is encoded as a five-byte instruction.

Problem 3: Runtime Stack [10 points]

a) The key function to examine is below:

```
#define MAX_ATTEMPTS 3
bool login() {
    size_t canary1 = rand_size_t(); // generates random size_t
    char actual[16];
    size_t canary2 = canary1;
    char supplied[16];

    strcpy(actual, retrieve_password()); // assume no issues
    size_t attempts = 0;
    while (attempts < MAX_ATTEMPTS) {
        printf("Enter password: ");
        gets(supplied);
        if (canary1 != canary2) {
            printf("Hacking attempt! Aborting login!\n");
            return false;
        }
        if (strncmp(supplied, actual, 16) == 0) return true;
        printf("Supplied password failed. Try again!\n");
        attempts++;
    }
    printf("Max attempts exceeded.\n");
    return false;
}
```

- You have the idea of entering 47 `'0'`s as the first password and then 23 `'0'`s as the second password, and you try. How far do you get? Leveraging your understanding of how the stack organizes **`login`**'s local variables as per the diagram on the prior page, explain why this won't work.

  You don't get very far at all because the last (highest-address) byte of **`canary1`** would be a true zero-byte—that is, a `'\0'`—whereas the last byte of **`canary2`** would store the ASCII code of the digit character `'0'`. That two canaries would be immediately flagged as unequal, and the function would return **`false`** after printing an angry message.

- Now explain why a single supplied password of 48 `'a'`'s in a row might allow **`login`** to stage a **`true`** as a return value, only to crash or cause other problems as it attempts to pass control back to the caller. But also explain why entering 48 `'a'`'s might actually work without a problem, and what must be true for that to happen.

  To write 48 `'a'`'s is really to write 49, the 49th being a `'\0'` that gets written in the first byte of the return address. The canaries and passwords match, so **`true`** will be

staged in **%eax**, but **retq** might pop a malformed address! If the **'\0'** overwrites a nonzero byte, the address might be invalid, and an attempt to execute that instruction might cause a crash. If that **'\0'**overwrites another **'\0'**, execution proceeds as normal.

b) Here's a reminder of the function of interest:

```
void backtrace() {
    struct func_info info;
    void **start = &info;
    while (true) {
        if (get_function_by_address(*start, &info)) {
            printf("%s\n", info.name);
            if (strcmp(info.name, "main") == 0) return;
        }
        start++;
    }
}
```

- Briefly explain how and why the **backtrace** function works. Specifically, explain what types of values on the stack prompt the **get_function_by_address** to return **true**, and also explain why **start** is initialized to **&info**.

  &info is the lowest meaningful stack address at the time **info** is declared. The stack frames of all functions are, well, stacked at higher addresses, and each stack frame is separated from the one below by a return address. The loop in **backtrace** walks the stack as an array of 64-bit values, and each time one is an address within some function, the function name gets printed. Everything stops when that function name is **"main"**.

- You're convinced the general implementation is pretty good, but then you realize that **function pointers** can be passed as parameters and stored in local variables, and you don't want the names of the functions they address to be printed just because they're stored on the stack. Describe how you might change the implementation of **get_function_by_address** to only print the names of functions that are currently executing without printing the names of functions simply function pointers are passed as parameters or stored as local variables.

  Modify test to check if the return value of **get_function_by_address** is true and that ***start** is strictly greater than **info.start**. If so, then the name of the function can be printed as it has been. If ***start == info.start**, then ***start** is a function pointer and should be skipped.

Problem 4: `stop-and-copy` Heap Compaction [20 points]

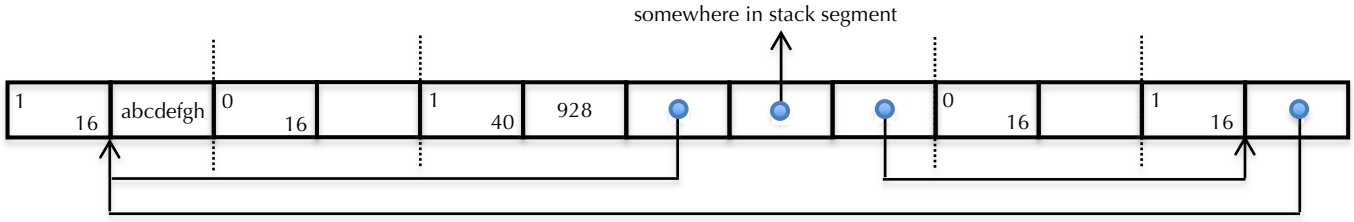
Some custom allocators optimize to solve the heap compaction problem by maintaining **two** heaps instead of one. Only one heap is active at any one moment, but when heap compaction is invoked, the ordered concatenation of all allocated blocks is written to the inactive heap, all internal pointers aliasing active heap memory are rewritten to reference the inactive heap clone, and then the active heap is rendered inactive and vice versa.

All pointers—whether stored in the heap, on the stack, or in global variables—would need to be updated, but here we'll assume all pointers into the active heap are stored within the active heap itself and nowhere else and that all such pointers always address the first byte of an allocated node's payload. Throughout the problem, we'll assume we're working with a 64-bit system where all **`size_t`**'s and pointers are eight bytes long. The active heap is **`sizeof(size_t)`**-aligned, is subdivided into nodes that are always a multiple of eight bytes, and nodes always have enough space for at least eight bytes of payload. The header is an eight-byte **`size_t`**, where the most significant bit is 1 for allocated nodes and 0 for free nodes, and the other 63 bits encode the node size, in bytes—that is, the header size plus the payload size, which is understood to be a multiple of 8.

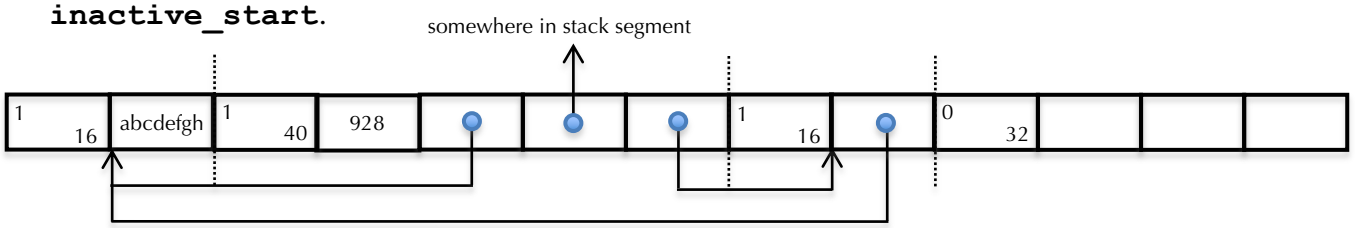
The free list is always implicit, so there are no next or previous pointers to worry about. For simplicity, we'll assume each of the two heaps are the same size, and that the bases of the two heaps are stored in global variables, as with:

```
static size_t *active_start;
static size_t *inactive_start;
#define HEADER_SIZE sizeof(size_t);
#define HEAP_SIZE (1L << 28) // 256 megabytes
```

The goal of this problem is to take a picture like so, the base address of which is stored **`active_start`**:



and replicate the in-use nodes to look like this, the base address of which is stored in **`inactive_start`**.



The whole process looks like a loop of **memcpy** calls, but with the added complexity that comes because pointers in the payloads of active nodes that appear to be active heap addresses need to be recalculated when dropped in the inactive-but-soon-to-be-made-active heap.

For this problem, you'll implement this heap compaction algorithm, affectionately known as the **stop-and-copy** algorithm. You'll be led through a series of steps—less efficient than it could be, but easier to do if carefully managed across several steps.

a) Here's the implementation of **node_is_allocated** :

```
bool node_is_allocated(size_t *header) {
    return (*header >> 63) == 1;
} // or, return (*header & (1L << 63)) != 0;
```

b) Here's the implementation of **node_get_size**:

```
size_t node_get_size(size_t *header) {
    return *header & ~(1L << 63);
}
```

c) Implement the **replicate_active_heap** function, which verbatim replicates all of the active heap's allocated nodes—and just the allocated nodes—by copying them, in order, to the inactive heap, and then taking care to properly mark the last node of the inactive heap as unallocated.

```
void replicate_active_heap() {
    size_t *active_curr = active_start;
    size_t *inactive_curr = inactive_start;
    size_t inactive_used = 0;
    size_t *active_end = active_start + HEAP_SIZE/sizeof(size_t);
    while (active_curr < active_end) {
        size_t size = node_get_size(active_curr);
        if (node_is_allocated(active_curr)) {
            memcpy(inactive_curr, active_curr, size);
            *(active_curr + 1) = inactive_used + sizeof(size_t);
            inactive_curr += size/sizeof(size_t);
            inactive_used += size;
        }
        active_curr += size/sizeof(size_t);
    }

    size_t inactive_unused = HEAP_SIZE - inactive_used;
    if (inactive_unused == 0) return;
    *inactive_curr = inactive_unused;
}
```

d) Finally, implement the **rewrite_addresses** function, which walks through all the payload words in the inactive heap, and rewrites any pointer addressing a word in the active heap with the corresponding address in the inactive heap. If the eight-byte figure doesn't look like a pointer within the active heap—that is, it's outside the regional between **active_start** and some **active_end** you'll need to compute—then you can ignore it and simply move onto the next eight bytes of payload. Note that if a figure looks to be an address within the active heap, then it is guaranteed to be the base address of an allocated node's payload.

```
bool within_active(size_t *address) {
    size_t *active_end = active_start + HEAP_SIZE/sizeof(size_t);
    return address >= active_start && address < active_end;
} // can make active_end and inactive_end globals if you want
```

```
void rewrite_addresses() {
    size_t *inactive_curr = inactive_start;
    size_t *inactive_end = inactive_start + HEAP_SIZE/sizeof(size_t);
    while (inactive_curr < inactive_end &&
           node_is_allocated(inactive_curr)) {
        size_t *payload_curr = inactive_curr + 1;
        size_t num_words = node_get_size(inactive_curr)/sizeof(size_t);
        size_t *payload_end = inactive_curr + num_words;
        while (payload_curr < payload_end) {
            if (within_active((size_t *)(*payload_curr))) {
                size_t offset = *(size_t *)(*payload_curr);
                *payload_curr =
                    (size_t)(inactive_start + offset/sizeof(size_t));
            }
            payload_curr++;
        }
        inactive_curr = payload_end;
    }
}
```