CS107
Autumn 2022 December, 2022

CS107 Final Exam Solution

Solution 1: Linked Lists of Packed Character Nodes

The solution is insultingly compact.

```
char *array_to_list(char *strings[], size_t n) {
   char *head = NULL; // could be left as a void * as well
   for (ssize_t i = n - 1; i >= 0; i--) { // ssize_t can be negative!
      char *node = malloc(strlen(strings[i]) + 1 + sizeof(char *));
      assert(node != NULL); // not necessary for solution
      strcpy(node, strings[i]);
      *(char **)(node + strlen(strings[i]) + 1) = head;
      head = node;
   }
   return head;
}
```

Solution 2: Assembly Code Analysis

The assembly code presented on the upper right was generated by compiling a function called **`ella`** without optimization—i.e., using **`-Og`**. Here's is the original function below.

```
char *ella(char *aretha[], char *diana) {
    char *vocalist = diana + 4;
    if (strspn(aretha[0], diana) == 0)
        return strstr(vocalist, vocalist);
    if (diana[0] != '\0')
        return ella(aretha, vocalist);
    return vocalist;
}
```

```
0x116d <+4>:   push   %r12
0x116f <+6>:   push   %rbp
0x1170 <+7>:   push   %rbx
0x1171 <+8>:   mov    %rdi,%r12
0x1174 <+11>:  mov    %rsi,%rbx
0x1177 <+14>:  lea    0x4(%rsi),%rbp
0x117b <+18>:  mov    (%rdi),%rdi
0x117e <+21>:  callq  0x1060 <strspn@plt>
0x1183 <+26>:  test   %rax,%rax
0x1186 <+29>:  je     0x1195 <ella+44>
0x1188 <+31>:  cmpb   $0x0,(%rbx)
0x118b <+34>:  jne    0x11a5 <ella+60>
0x118d <+36>:  mov    %rbp,%rax
0x1190 <+39>:  pop    %rbx
0x1191 <+40>:  pop    %rbp
0x1192 <+41>:  pop    %r12
0x1194 <+43>:  retq
0x1195 <+44>:  mov    %rbp,%rsi
0x1198 <+47>:  mov    %rbp,%rdi
0x119b <+50>:  callq  0x1070 <strstr@plt>
0x11a0 <+55>:  mov    %rax,%rbp
0x11a3 <+58>:  jmp    0x118d <ella+36>
0x11a5 <+60>:  mov    %rbp,%rsi
0x11a8 <+63>:  mov    %r12,%rdi
0x11ab <+66>:  callq  0x1169 <ella>
0x11b0 <+71>:  mov    %rax,%rbp
0x11b3 <+74>:  jmp    0x118d <ella+36>
```

Note that one could invert the tests and correspondingly rearrange the return statements for an equivalent answer. Perhaps the second **`if`** test is **`diana[0] == '\0'`** and the last two return statements are swapped.

*The unoptimized version pushes three caller-owned registers to the stack, and the optimized version only pushes two. Why doesn't the optimized version need to push **%r12**?*

```
0x11b4 <+4>:   push   %rbp
0x11b5 <+5>:   mov    %rsi,%rbp
0x11b8 <+8>:   push   %rbx
0x11b9 <+9>:   sub    $0x8,%rsp
0x11bd <+13>:  mov    (%rdi),%rbx
0x11c0 <+16>:  jmp    0x11ce <ella+30>
0x11c2 <+18>:  nopw   0x0(%rax,%rax,1)
0x11c8 <+24>:  cmpb   $0x0,-0x4(%rbp)
0x11cc <+28>:  je     0x11f8 <ella+72>
0x11ce <+30>:  mov    %rbp,%rsi
0x11d1 <+33>:  mov    %rbx,%rdi
0x11d4 <+36>:  add    $0x4,%rbp
0x11d8 <+40>:  callq  0x1060 <strspn@plt>
0x11dd <+45>:  test   %rax,%rax
0x11e0 <+48>:  jne    0x11c8 <ella+24>
0x11e2 <+50>:  add    $0x8,%rsp
0x11e6 <+54>:  mov    %rbp,%rsi
0x11e9 <+57>:  mov    %rbp,%rdi
0x11ec <+60>:  pop    %rbx
0x11ed <+61>:  pop    %rbp
0x11ee <+62>:  jmpq   0x1070 <strstr@plt>
0x11f3 <+67>:  nopl   0x0(%rax,%rax,1)
0x11f8 <+72>:  add    $0x8,%rsp
0x11fc <+76>:  mov    %rbp,%rax
0x11ff <+79>:  pop    %rbx
0x1200 <+80>:  pop    %rbp
0x1201 <+81>:  retq
```

The most straightforward answer is that the computation doesn't use **%r12** so that its incoming value gets clobbered, so there's no reason to spill the contents of **%r12** to be stack.

The unoptimized version clearly makes a recursive call to **ella**, whereas the second version doesn't. What is the second version doing instead, and why can it do it?

Because the call to **ella**, when made, is tail recursive, the compiler can reframe the recursive call to execute iteratively and reuse the space set up for the original call to **ella**. After all, the original call doesn't need that space anymore.

The unoptimized version uses **callq** to invoke the **strstr** function whereas the optimized version uses **jmpq** instead. What does **callq** do that **jmpq** doesn't, and why can the optimized version use **jmpq** instead of **callq**?

At the time that **strstr** is called, **%rsp** contains the address of the instruction immediately following the call to **ella**. Because **strstr**'s return value is **ella**'s return value, execution can simply jump to the code for **strstr**, and when execution within hits some **retq** instruction, it can bypass the code for **ella** and return directly to the instruction immediately following the **callq** to **ella**, wherever that was.

Solution 3: Ellipses and `printf`

Here's the partial implementation of **`myprintf`**. You're to work through the code I provide you and complete the implementation. You can assume that **`args`** addresses a properly assembled array of manually packed bytes as described above. If there were no additional arguments, you can assume that **`args`** is **`NULL`**. You can also assume that every **`'%'`** in the control string will be following by either a **`'d'`** or an **`'s'`**.

```
void myprintf(const char *control, const void *args) {
   while (true) {
      const char *placeholder = strchr(control, '%');
      if (placeholder == NULL) placeholder = control + strlen(control);
      char buffer[placeholder - control + 1];
      strncpy(buffer, control, placeholder - control);
      buffer[placeholder - control] = '\0';
      print_string(buffer);
      control = placeholder;
      if (control[0] == '\0') break;

      // here's my own solution
      if (placeholder[1] == 'd') {
         print_int(*(int *)args);
         args = (char *) args + sizeof(int);
      } else {
         print_string(*(char **)args);
         args = (char *) args + sizeof(char *);
      }
      control += 2; // hop over placeholder and continue afresh
   }
```

*Describe what would be printed by each of the following calls to **`printf`** if it just relies on the **`myprintf`** you've implemented above. If the call generates a segmentation fault, then say so.*

- `printf("%s", 0, 0);`

  This would crash, because those two 0's would collectively be interpreted as an eight-byte **`NULL`** pointer, which would be passed to **`print_string`**, which would presumably deference the pointer and generate a segmentation fault. (If you explicitly write that **`print_string`** would print **`(nil)`**, we'll accept that as well).

- `printf("%d", "107");`

  This would print four bytes of the eight-byte address as an integer. What eight-byte address? The address of the **`'1'`** at the beginning of that **`"107"`** string.

- `printf("%d %d", 555);`

  This would print 555 followed by whatever random four-byte integer happens to come after it in stack memory. Note that this would not crash, because the memory incorrectly accessed because of that second **`"%d"`** will still be memory accessible to the program—i.e., it's still part of the stack frame of **`printf`**.

- `printf("lots of smoke and mirrors", "lots", "of", "them");`

  This just prints **`"lots of smoke and mirrors"`**. The three additional **`char *`**s reachable through the **`args`** parameter would just go ignored.

Solution 4: Implicit Allocators with Headers and Footers

Assume the following **`#define`** constants and global variables have already been set up:

```
#define HEAD_SIZE sizeof(size_t)
#define FOOT_SIZE HEAD_SIZE

// flags used to isolate free and left-free bits from payload size
#define FREE (1L << 63)
#define LEFT (1L << 62)
#define SIZE ______________

static size_t *heap_start; // base address of entire heap segment
static size_t heap_size;   // number of bytes in the entire heap segment
```

a) First off, note that the **`#define`** value for **`SIZE`** is blank! What expression—which you must frame in terms of **`FREE`** and **`LEFT`**—should be used so that **`SIZE`** is a mask of 2 0's followed by 62 1's? (The **`SIZE`** mask can then be used to isolate the payload-size portion of a header or footer.)

```
#define SIZE (~(FREE | LEFT))
// outer parentheses not needed for full credit, though needed in practice
```

b) You wonder whether it make sense to **`#define FREE`**, **`LEFT`**, and **`SIZE`** to be **`0x8000000000000000`**, **`0x4000000000000000`**, and **`0x3FFFFFFFFFFFFFFF`**, respectively, so that repeated reevaluation of **`1 << 63`**, **`1 << 62`**, and your expression for **`SIZE`** doesn't impact allocator throughput. After using **`callgrind`** to profile the number of instructions executed on test scripts, you note that it doesn't seem to make a difference, even when your allocator is compiled at **`-Og`**? Give a reasonable explanation why that might be.

   The compiler would still evaluate **`1L << 63`** and **`1L << 62`** at compile time and insert their evaluations—**`0x8000000000000000`**, **`0x4000000000000000`**, and **`0x3FFFFFFFFFFFFFFF`**—that directly into the assembly. Some optimizations are so obvious that they're not even considered to be optimizations.

c) Complete the implementation of the **`count_available_bytes`** function, which scans the heap from front to back and returns the total number of available payload bytes. Your implementation will need to examine all nodes—both free and allocated—to compute the answer, since the allocator is an implicit one.

```
size_t count_available_bytes() {
   size_t count = 0;
   size_t *curr = heap_start;
   size_t *end = curr + heap_size/sizeof(size_t);
   while (curr != end) {
      size_t node_size = *curr & SIZE;
      if (*curr & FREE) count += node_size - HEAD_SIZE;
      curr += node_size/sizeof(size_t);
   }

   return count;
}
```

d) Complete the implementation of **`coalesce_left`**, which accepts the address of a free node header and, if the node to its left is also free, merges the two into one larger node. If the node to the left isn't free, then **`coalesce_left`** should simply return without doing anything.

```
void coalesce_left(size_t *header) {
   if (!(*header & LEFT)) return;
   size_t right_node_size = *header & SIZE;
   size_t *footer = header - 1;
   size_t left_node_size = *footer & SIZE;
   header -= left_node_size/sizeof(size_t);
   *header += right_node_size;
   footer += right_node_size/sizeof(size_t);
   *footer = *header;
}
```