CS107
Autumn 2023 December 9, 2023

CS107 Mock Final Exam

This is a closed book, closed note, closed computer exam, though you're permitted to refer to the reference sheet I've provided and the short cheat sheet you've prepared ahead of time.

You have 180 minutes to complete all problems. You don't need to **`#include`** any libraries, and you needn't use **`assert`** to guard against any errors. Understand that the majority of points are awarded for concepts taught in CS107, and not prior classes. You don't get many points for **`for`**-loop syntax, but you certainly get points for proper use of **`&`**, **`*`**, and the low-level C functions introduced in the course. Understand that the mock final in no way obligates me to imitate its structure while writing your official final exam for this coming Monday.

Good luck!

SUNet ID (@stanford.edu): ______________________________

Last Name: ______________________________

First Name: ______________________________

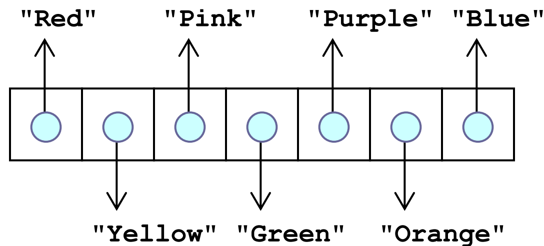
I accept the letter and spirit of Stanford's Honor Code.

[signed] ______________________________

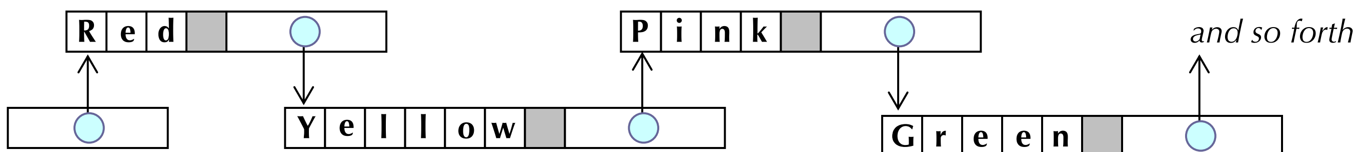
		Score	Grader
1. Linked Lists of Character Nodes	[10]	______	______
2. Assembly Code Analysis	[20]	______	______
3. Ellipses and **`printf`**	[15]	______	______
4. Implicit Allocators, Headers, Footers	[15]	______	______
Total	**[60]**	______	______

Problem 1: Linked Lists of Packed Character Nodes [10 points]

Write a function **`arrayToList`** to convert an array of null-terminated strings…



into a linked list of **perfectly sized** character nodes, where each node wedges all of the characters of a string and the address of the next node into one contiguous block:



Notice that each node of the list stores the individual characters of the string, followed by the `'\0'` (represented by a shaded box), followed by an eight-byte address identifying the location of the **next** node of the list (and if these eight bytes are all zeroes, then you've reached the end of the list.)

`array_to_list` takes a standard array of **`char *`**s along with its length and constructs the corresponding list of perfectly sized nodes. Your implementation:

- shouldn't worry about alignment restrictions. In practice, padding would sometimes be inserted between the `'\0'`s and the pointers, but don't worry about that here.
- should return the address of the first character in the first node of the list, or **`NULL`** if the array is empty.
- should make use of **`malloc`**, **`strlen`**, and **`strcpy`** as appropriate.
- can be implemented iteratively or recursively.

Use the space on the next page to present your implementation of **`array_to_list`**.

```
char *array_to_list(char *strings[], size_t n) {
```

Problem 2: Assembly Code Analysis [20 points]

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

```
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>
```

a. [14 points] First, fill in the blanks below so that **`ella`** 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.

```
char *ella(char *aretha[], char *diana) {

   char *vocalist = ________________________________;

   if (________________________________)

      return ________________________________;

   if (________________________________)

      return ________________________________;

   return ________________________________;
}
```

Now, study the more aggressively optimized version of **ella** presented on the right, and answer the questions below. These short answer questions don't really require an understanding of the **gcc** optimizations discussed in lecture, but rather a clear understanding of what the compiler can get away with to make the code run more quickly.

```
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
```

b. [2 points] 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**?

c. [2 points] 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?

d. [2 points] 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**?

Problem 3: Ellipses and `printf` [15 points]

The C standard allows for a variable number of arguments to be passed to a function, provided the prototype makes use of the **ellipses** to clarify where the required arguments end and the optional arguments begin. The prototype for **`printf`**, for instance, is:

```
int printf(const char *control, ...); // we'll ignore the return value
```

The prototype mandates that a C string be passed in as the first argument. But after that, the caller can provide zero, one, two, three, or even more arguments beyond that.

For our purposes, assume the implementation of **`printf`** allocates enough **stack memory** to store all of the additional arguments, side by side, as a packed array of bytes. For instance, a call to **`printf("%d %s %s\n", 153, "frisbee", "sunshine")`** would prompt **`printf`** to allocate stack space for exactly 20 bytes—or rather, **`sizeof(int) + 2 * sizeof(char *)`** bytes—and populate that space with the four-byte data representation of 153, followed by the eight-byte data representation of the first **`char *`** value, followed by the eight-byte data representation of the second **`char *`** value. Assuming the **`'f'`** of **`"frisbee"`** and the **`'s'`** of **`"sunshine"`** reside at addresses **`0x453200`** and **`0x453220`**, respectively, the memory for this stack array would be laid out like this:

153	0x453200	0x453220

The example above generalizes to all reasonable calls to **`printf`**. For example:

- A call to **`printf("%s %s %s %d %d", "abc", "def", "wxyz", 45, 55)`** would allocate space for 32 bytes and pack together, in order, the three addresses and the two integers. **`3 * sizeof(char *) + 2 * sizeof(int)`** equal 32.
- A call to **`printf("%d %d %s %d %d", 1, 2, "3", 4,  5)`** would allocate space for 24 bytes: **`int`**, **`int`**, **`char *`**, **`int`**, **`int`**.

In all cases, **`printf`** essentially constructs a miniature stack frame and populates it with copies of any additional arguments in the order they were supplied. Once **`printf`** assembles and populates the array, it then calls a helper function—a function called **`myprintf`**—and passes one the original **`control`** parameter verbatim and the base address of the packed array of bytes as arguments one and two.

```
void myprintf(const char *control, const void *args);
```

By doing so, **`myprintf`** has access to all of the material—the control string as a template of what to print and the additional values needed to fill in any placeholders—to publish the string to the console.

For simplicity, we'll assume that the only placeholders ever present in **`control`** are **`%d`** and **`%s`**. You can further assume the following core helper operations figure out how to print a C string and an integer to standard output:

```
void print_string(const char *str); // '%' will never be present in str
void print_int(int num);
```

The space below includes a partial implementation of **myprintf**, and you're to complete it. You'll do so by manually crawling down the array of memory addressed by **args**, using the control string to decide whether the next figure to be consumed from that memory is an **int** or a **char ***. Once implemented, your **myprintf** would contribute to the following:

```
int main(int argc, const char *argv[]) {
   printf("My favorite numbers are %d, %d, and %d.\n", 28, 496, 8128);
   printf("You remind me of %s%d%s%s.\n", "R", 2, "D", "2");
   return 0;
}
```

and generate the following output:

```
My favorite numbers are 28, 496, and 8128.
You remind me of R2D2.
```

a) [7 points] 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;
      // place the rest of the implementation in the space below
```

b) [8 points] 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);`**

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

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

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

Problem 4: Implicit Allocators with Headers and Footers [15 points]

You are implementing a custom allocator that relies on an eight-byte header and an eight-byte ***footer***—a replica of the same node's header—that overlays the last eight bytes of a free node's payload. The footer is used to implement **left coalescing**, which can be implemented by **`free`** to reduce fragmentation.

The most significant bit of the header (and footer) records whether the node is free (1 free, 0 in-use). The second most significant bit records whether the node to its immediate left is free (1 free, 0 otherwise). The remaining 62 bits encode the size of the entire node—header plus payload—in bytes. All request sizes are rounded up to the nearest multiple of eight bytes, so all payload addresses returned/accepted by **`malloc`**/**`free`** are aligned accordingly.

0x2000	2008	2010	2018	2020	2028	2030	2038	2040	2048	2050	2058	2060	2068	2070
size 40 left 0 free 1				size 40 left 0 free 1	size 32 left 1 free 0				size 48 left 0 free 1					size 48 left 0 free 1

In the above diagram, the 32-byte payload with base address 0x2008 is free, the 24-byte payload with base address 0x2030 is in use, and the 40-byte payload at address 0x2050 is free. Note this allocator is technically an implicit one, as there's no mention of linked lists anywhere.

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) [2 points] 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.)

b) [2 points] You wonder whether it make sense to `#define FREE`, `LEFT`, and `SIZE` to be `0x8000000000000000`, `0x4000000000000000`, and `0x3FFFFFFFFFFFFFFF`, respectively, so that repeated reevaluation of `1L << 63`, `1L << 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.

c) [5 points] Complete the implementation of the `count_available_payload_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_payload_bytes() {
```

d) [6 points] 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) {
```