0/35 Questions Answered

TIME REMAINING
1 hr 49 mins

Exam Statistics:

Min: 40/90
Median: 73/90
Max: 91/90

Mean: 70.21/90
StdDev: 17.37/90

Lowest scoring questions:
5.2 (24%)
8.8 (52%)
6.3 (60%)
7.1 (62%)
5.1 (62%)

Questions scoring >90%:
2.3
3.3
5.3
6.1, 6.2, 6.5, 6.7, 6.8
8.2, 8.5, 8.6

COS 217 Final Exam

Q1 Be our guest - put our exam to the test

0 Points

This exam consists of 7 substantive questions (Q2-Q8 -- Q1 is only exam information and Q10 is only the Honor pledge) totaling 90 points. Q9 is a 1-point lecture "attention to detail" extra credit. Most of the substantive questions are made up of multiple parts, with points allocated as indicated.

Unless you have confirmed ODS accommodations, you have 3 hours (180 minutes) to complete the exam from the time you begin. Unless you have received an alternate exam window from Dr. Moretti via your Dean or Director of Studies, you must complete the exam by the end of the day (11:59 PM Princeton time (US Eastern Time)) on December 12, irrespective of when you began.

In Gradescope, students' answers are autosaved as they enter them. We have observed a couple second latency, though, so we advise against changing answers right up to the deadline. There is a countdown timer (which can be hidden) in the top right corner of the screen.

This exam is "open-book" but "closed-communication":

- You are not allowed to communicate with any other person, whether inside or outside the class. You may not send the exam problems to anyone, nor receive them from anyone, nor communicate any information about the problems or their topics.
- You are allowed to consult any material from course lectures, precepts, readings, assignments, Ed, etc.

- You are allowed to use resources found on the web, so long as they do not violate the communication rule above (i.e., so long as they are not solicited by you). As an example, you can read an old Stack Overflow post, but you can't post a question to Stack Overflow.
- You may build and run any code on armlab (though be careful, as this can be a dramatic time sink!)

You may post (private!) posts to Ed to seek clarification from the course staff. We will monitor Ed regularly, however we cannot guarantee 24-hour availability throughout the exam period.

This examination is administered under the Princeton University Honor Code. All suspected violations of the Honor Code must be reported to the Committee on Discipline. You will attest to the standard pledge in Q10 after you finish your responses.

☑ I've read this. Okay!

Q2 Be Bashful
9 Points

Q2.1
5 Points

Select all of the following options that will produce an empty file (that is, a file with length 0) called `myfile` when executed in `bash` (assume that `myfile` does not already exist):

- [ ] `echo > myfile` — echo > myfile creates a file containing a newline character. (You have to provide echo with the argument -n to suppress the newline.)
- [x] `touch myfile` — touch myfile updates the modification timestamp of the file, creating it (with no contents) if necessary.
- [x] `:> myfile` — : is a bash built-in null command: it takes no input, produces no output, and returns success. It is very similar to the command true.
- [ ] `true || cp /dev/null myfile` — Speaking of true, this also simply returns success. The || operator short-circuits (just like in C and Java), and since true OR (anything) is true, the cp command is never executed. Without the true || this would have been a valid answer: that cp command does in fact create an empty file.
- [x] `>| myfile` — You may have seen >| along the way in this course to indicate that you want to "clobber" the file that standard output is redirected to, even if it exists. So this is used instead of just > with another variant of the null command. bash (but not all shells) has the same behavior as the null command when you omit the command entirely.

Q2.2
2 Points

You have two programs, `first` and `second`, both of which appear in your `PATH`. You would like to execute `first` and then execute `second` only if `first` succeeds. Write 1-2 lines of `bash` commands that will accomplish this:

Enter your answer here

There were several approaches that could work. Here are three common ones:

first && second

if first; then second; fi

first
 if [$? -eq 0]; then second; fi

Q2.3
2 Points

You have several `bash` commands that you'll be using over and over. Give two plausible ways to improve this workflow versus typing in the commands repeatedly:

Enter your answer here

Examples of reasonable answers include:
* use a bash script
* use non-file targets in a Makefile
* define short bash aliases for the commands
* use ctrl-r to search the bash history
* use !prefix to access the bash history
* use the up arrow to manually cycle through history

Q3 You're no Dumbo when you automate building and testing.

13 Points

Consider the following C code, which is executed only once in its program. You may assume that `x`, `y`, and `z` are initialized with `int` values from standard input before executing this code:

```
switch (x) {
   case 1: y *= foo(y);
           y++;
   case 2: y *= bar(y);
           y--;
   default:
           y *= baz(y);
}

if (z)
   y++;
else
   y--;
```

Statement testing ensures that every statement in the file is executed at least once.

Because there are no break; statements at the end of each case, you can make sure every statement within the switch is executed by starting in case 1.

At that point, you only need two inputs: one that enters the consequent of the if and another that enters the alternative of the if, so long as at least one of those two enters into case 1 of the switch.

Thus, the correct answer is 2.

Q3.1

2 Points

How many input data sets will be necessary in order to achieve complete statement testing of this code?

Enter your answer here

Q3.2

2 Points

How many input data sets will be necessary in order to achieve complete path testing of this code?

Enter your answer here

Path testing requires every possible logical path through the code to be tested.

For this program, there are 3 paths through the switch statement (entering in case 1, entering in case 2, and entering in default), and each of those has two ways through the if statement (the consequent and the alternative).

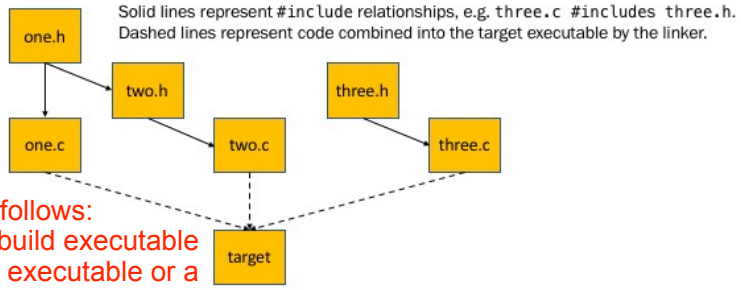
Thus, there are 6 possible paths through the code.

Q3.3

9 Points

Create a proper `Makefile` for the dependency graph below. Your

`Makefile` should minimize the number of recompilations necessary by supporting partial-builds based on the dependency graph. You are not required to have any non-file targets, however the final executable `target` must be created when `make` is invoked with no arguments.



The nine points were allocated as follows:
* make with no arguments should build executable (so either the first rule must be the executable or a non-file target with the executable as a dependency.)
* rule to make executable should have .o files as dependencies
* rule to make executable should *not* have .c or .h files as dependencies
* command to build executable should only use .o files and should name it target
* there should be rules for three .o files: one.o, two.o, and three.o
* each .o rule should have corresponding .c file as dependency
* each .o rule should have corresponding .h file as dependency
* the two.o rule should also have one.h as a dependency, since two.h #includes one.h
* each .o command should use only corresponding .c file, no .h files. Implicit rules are okay.

Note -- Gradescope will not let you enter a tab character in the response field, so instead you can use 8 spaces as the prefix to the command for any rule.

Enter your answer here

The correct answer is:

```
target: one.o two.o three.o
        gcc217 one.o two.o three.o -o target
one.o: one.c one.h
        gcc217 -c one.c
two.o: two.c two.h one.h
        gcc217 -c two.c
three.o: three.c three.h
        gcc217 -c three.c
```

Q4 Li Shang'll make a string out of you

12 Points

For each part of this question you are presented with three code snippets. Assume each snippet is in its own one-file program with all appropriate header files included. In each snippet, lines prefixed with `o:` are outside any function and lines prefixed with `f:` are within a function and appear contiguously in the order given.

You will indicate which of the snippets result in the specified memory section containing the specified data. It could be none of them, all of them, or anywhere in between.

For example, here is an example of a snippet that results in the RODATA section containing the 7 bytes of the string "RODATA":

```
f: puts("RODATA");
```

It is okay if the snippet also results in another section containing the specified data.

Q4.1
3 Points

Consider the following code snippets:

```
/* A */
f: char stack[] = {'S', 'T', 'A', 'C', 'K', '\0'};
```

```
/* B */
f: char *stack = "STACK";
```

```
/* C */
f: char stack[6];
f: strcpy(stack, "STACK");
```

Which of these snippets result in the STACK section containing the 6 bytes of the string "STACK"?

- [x] A
- [ ] B
- [x] C

A: declares stack as a local variable within a function, with array of characters as its type. Local variable arrays defined at compile-time are allocated in their function's stackframe. The initializer list sets its size (since it was not provided in the []) and contents.

B: declares stack as a local variable within a function, with pointer to a character as its type. This pointer is allocated in its function's stackframe, but it is initialized to the address of a string literal (which is stored in the RODATA section), not the bytes of the string itself.

C: declares stack as a local variable within a function, with array of characters as its type and a length of 6. Local variable arrays defined at compile-time are allocated in their function's stackframe. The address of that variable and the address (in the RODATA section) of the string literal are passed to strcpy, which copies the bytes of the string from the RODATA section into the array on the stack.

Q4.2
3 Points

Consider the following code snippets:

```
/* D */
f: char *heap = malloc(strlen("HEAP")+1);
f: heap = "heap";
```

```
/* E */
f: char *heap = calloc(5, sizeof(char));
f: char *pile = "HEAP";
f: for (i = 0; i < 4; i++) /* assume i has been declared as a size
f:    heap[i] = pile[i];
```

```
/* F */
f: char **heap = malloc(sizeof(char*));
f: heap[0] = strcpy(malloc(5), "HEAP");
```

D: declares heap as a local variable within a function, with pointer to a character as its type. It is initialized to an address in the heap returned by the call to malloc, which is the beginning of an allocation sufficiently large to store the required string. However no bytes are copied into this space on the heap, instead, the variable is pointed to the location of a string literal in RODATA.

It doesn't impact the answer, but perhaps worth noting:
1 this creates a memory leak: there is no way to access the malloc'ed memory
2 the string in RODATA isn't even the correct string, being lower-case.

Assuming memory allocation always succeeds, which of these snippets result in the HEAP section containing the 5 bytes of the string "HEAP"?

- [ ] D
- [x] E
- [x] F

E: declares heap as a local variable within a function, with pointer to a character as its type. It is initialized to an address in the heap returned by calloc, which is the beginning of an allocation sufficiently large to store the required string that has had all its bytes set to '\0'. The local variable pile (also a char *) is declared and initialized to point to the address of the string literal in the RODATA section. Within the loop, each of the first four characters of string in RODATA are accessed by dereferencing pile (using array indexing syntax) and copied into the heap by dereferencing the heap pointer in the same manner. Because the allocation in the heap was initialized to all null-bytes, not copying the trailing null-byte from RODATA is not a problem.

F: The strcpy alone is sufficient here. heap is declared as a local variable within a function, with pointer to a character pointer as its type. It is initialized to the address returned by malloc, which is the first byte of an allocation in the heap large enough to store a single pointer to a character. strcpy takes as its destination argument another address returned by malloc, which is the first byte of a second allocation in the heap, this one 5 bytes long. strcpy's source argument is the address in the RODATA section of the string literal. strcpy copies each of the bytes of the string from the RODATA section into the second heap allocation (satisfying the requirement), and returns the address of the allocation to be referenced by the pointer from the first heap allocation.

Q4.3
3 Points

Consider the following code snippets:

```
/* G */
o: char data[5] = "DATA";
```

```
/* H */
o: char *data;
```

```
f: data = "DATA";
```

```
/* I */
o: char *data = "217!";
f: data = "DATA";
```

Which of these snippets result in the DATA section containing the 5 bytes of the string "DATA"?

- [x] G
- [ ] H
- [ ] I

G: declares data as a file-scope variable with array of characters as its type and a length of 5. Variables defined outside any function and given an explicit initialization in a declaration are put in the DATA section. Its initialization uses the same string syntactic sugar for an initializer list as from this precept handout on stack-resident strings: https://www.cs.princeton.edu/courses/archive/fall20/cos217/precepts/07arraysstrings/strings.pdf

H: declares data as a file-scope variable with pointer to a character as its type. Variables defined outside any function and not given an explicit initialization in a declaration are put in the BSS section.

I: declares data as a file-scope variable with pointer to a character as its type. Variables defined outside any function and given an explicit initialization in a declaration are put in the DATA section. It is initialized to point at the address of the first byte of the string literal "217!", in RODATA. The assignment within the function points data at a different string (also in the RODATA section) instead of copying the bytes of the string into the DATA section.

Q4.4
3 Points

Consider the following code snippets:

```
/* J */
o: char *bss;
f: bss = "BSS";
```

```
/* K */
o: char bss[4];
f: strcpy(bss, "BSS");
```

```
/* L */
o: char *bss = NULL;
f: strcpy(bss, "BSS");
```

Which of these snippets result in the BSS section containing the 4 bytes of the string "BSS"?

- [ ] J

J: declares bss as a file-scope variable with pointer to a character as its type. Variables defined outside any function and not given an explicit initialization in a declaration are put in the BSS section. The assignment within the function gives bss a value that is an address in the RODATA section (the address of the first byte of the string literal) instead of copying the bytes of the string into the BSS section.

- [x] K

K: declares bss as a file-scope variable with array of characters as its type and a length of 4. Variables defined outside any function and not given an explicit initialization in a declaration are put in the BSS section. Later, strcpy takes the address of bss in the BSS section and the address of a string literal in the RODATA section and copies each byte of the latter into the former, so the required bytes do end up in BSS.

- [ ] L

L: declares bss as a file-scope variable with pointer to a character as its type. Variables defined outside any function and given an explicit initialization in a declaration are put in the DATA section. Also, though it does not affect the answer, there is another problem here: passing a NULL pointer as strcpy's destination will crash.

Q5 Soul -> Onward -> Frozen II -> ... -> Snow White

21 Points

Consider the following buggy implementation of a `List` construct, and assume that all necessary interface files have been included:

```
/* a Node_T is a member of the List with a string as contents */
typedef struct node* Node_T;

/* building block of the List */
struct node {
   /* contents of node */
   char* payload;
   /* next node in List */
   Node_T next;
};

/* head of the List */
static Node_T first = NULL;

/* if payload is not already in the List,
   inserts a new node at front of the List having contents payloac
   returns 1 if insertion is successful, 0 if unsuccessful. */
int List_insert(const char* payload) {
   Node_T curr = first;
   assert(payload != NULL);
   while(curr != NULL)
      if(!strcmp(curr->payload, payload))
         return 0;
   curr = malloc(sizeof(struct node));
   if(curr == NULL)
      return 0;
   curr->next = first;
   curr->payload = malloc(strlen(payload)+1);
   if(curr->payload == NULL)
```

```
      return 0;
   strcpy(curr->payload, payload);
   return 1;
}

/* removes all nodes from the List */
void List_free() {
   Node_T current;
   for(current = first; current != NULL; current = current->next)
      free(current);
}
```

Q5.1
1 Point

`List`, as defined and used here, is a(n):

- ○ Stateless Module
- ○ ADT
- ✓ ○ AO
- ○ None of these

List is an abstract object:

* it has a file-scope field defining its state and a set of functions using that state, such that the only way for a client to interact with the state is through those functions.

* the functions interact with only the single instance of it, unlike in an ADT where a pointer to the instance is passed in as a parameter to the functions.

Q5.2
1 Point

`Node_T`, as defined and used here, is a(n):

- ○ Stateless Module
- ○ ADT
- ○ AO
- ✓ ○ None of these

A Node_T, as a pointer to an underlying structure, does represent an object with state (unlike, e.g., the string module from A2).

A Node_T's state may be directly manipulated by List functions, so it is not abstract.

A Node_T can have an arbitrary number of instances instantiated, so it is not an AO.

So "None" is the best answer.

Q5.3
1 Point

Making a defensive copy of the payload string in `List_insert` is

unnecessarily cautious, because the `payload` parameter to `List_insert` is declared `const`.

○ True

✓ False

The const keyword stops the module implementation from changing the value, but it doesn't stop the client (who owns the data) from changing it unbeknownst to the module. The premise is identical to our rationale for the defensive copy of the key in the SymTable assignment.

Q5.4

9 Points

Identify three bugs in the `List_insert` function and how each could be fixed.

A bug for this problem is something that causes a warning or error from gcc217, a runtime crash, behavior that violates the function's contract, or a dynamic memory management issue observable by MemInfo or Valgrind.

Bug 1:

Enter your answer here

The "contains check" while loop doesn't advance curr (which results in an infinite loop when the list isn't empty and you're inserting a payload that doesn't already exist at the head of the list). The fix is to add an else clause to the if statement within the while loop that updates curr, e.g. curr = curr->next; (or make the while's body a compound statement with the update after the if statement)

Bug 2:

Enter your answer here

There is a memory leak if the second malloc call fails, because the space allocated by the first malloc call is not freed. The fix is to free(curr); before returning 0 from the curr->payload malloc check.

Bug 3:

Enter your answer here

The new node's next pointer is set to point at first, so that it will come before all existing items in the list, but first is not updated to point to the new node, so the list effectively does not change. Further, this means that the new node and its payload are inaccessible, and thus a memory leak. The fix is to update first = curr; before returning 1.

Q5.5

9 Points

Identify three bugs in the `List_free` function and how each could be fixed.

A bug for this problem is something that causes a warning or error from gcc217, a runtime crash, behavior that violates the function's contract, or a dynamic memory management issue observable by MemInfo or Valgrind.

Bug 1:

Enter your answer here

There is a memory leak, because only the struct is freed, but the defensive copy of the payload needs to be freed too. The fix is to add free(current->payload); before the existing free(current); within the while loop's body.

Bug 2:

Enter your answer here

The update step of the for loop dereferences a dangling pointer, since current was just freed in the previous iteration of the loop body. The easiest fix is to add another variable declaration at the top of the function, e.g. Node_T next;, set next = current->next; within the body of the loop before freeing current, and change the update step to be current = next.

Bug 3:

Enter your answer here

first is left as a dangling pointer instead of being set to NULL, so future list accesses will attempt to traverse through nodes that have already been freed. The fix is to add first = NULL; after the loop.

Q6 Ursula implores you to "Go ahead! Make your choice!"

10 Points

Here are the C definitions for a slightly different list from the one in the previous Question:

```
/* a Node_T is a member of a collection holding unsigned long valu
typedef struct node* Node_T;

struct node {
   /* contents of node */
   unsigned long payload;
   /* next node in the list */
   Node_T next;
```

```
};

struct list {
   /* head of the list */
   Node_T first;
   /* number of nodes in the list */
   unsigned long length;
};

/* a List_T is a collection of unsigned longs */
typedef struct list* List_T;
```

And here is an AARCH64 assembly language function correctly implementing some operation for such a list:

```
1       .global List_mystery
2  List_mystery:
3       sub     sp, sp, 32
4       str     x0, [sp,8]
5       ldr     x1, [x0,8]
6       cmp     x1, xzr
7       bne     .L2
8       mov     x0, 0
9       b       .L3
10 .L2:
11      ldr     x1, [x0]
12      ldr     x2, [x1]
13      str     x2, [sp,16]
14      str     x1, [sp,24]
15      ldr     x2, [x1,8]
16      b       .L4
17 .L6:
18      ldr     x3, [x2]
19      ldr     x0, [sp,16]
20      cmp     x3, x0
21      bls     .L5
22      str     x3, [sp,16]
23      str     x2, [sp,24]
24 .L5:
25      ldr     x2, [x2,8]
26 .L4:
27      cmp     x2, xzr
28      bne     .L6
29      ldr     x0, [sp,24]
30 .L3:
31      add     sp, sp, 32
32      ret
```

Answer each part of this question based on the code above.

Q6.1
1 Point

How many parameters does the function `List_mystery` take?

- ○ 0
- ☑ 1
- ○ 2
- ○ 3 or more

Line 4 stores the initial value of x0, which is the first argument. No registers among x1-x7 are referenced before being loaded into. So there is a single parameter.

Q6.2
1 Point

Does the function `List_mystery` appear to return a value?

- ☑ Yes
- ○ No

Line 8 puts the value 0 in x0 before branching to the epilogue and return. Line 29 loads the value of a local variable into x0 before continuing on sequentially to the epilogue and return. Since x0 is the register used for the return value, this behavior suggests that the function does return a value.

Q6.3
1 Point

What is the purpose of lines 13-14:

```
str x2, [sp, 16]
str x1, [sp, 24]
```

- ○ Save the value of some parameters
- ○ Save the value of some callee-saved registers
- ○ Allocate space for some local variables
- ☑ Initialize some local variables

The stack *can* be used for each of storing function arguments, saving initial values of callee-saved registers to be restored before returning, and storing local variables.

But in this case, [sp, 8] is used for the parameter (based on line 4), and the function never uses any callee-saved registers, so [sp, 16] and [sp, 24] are the locations in the stackframe for local variables. The str instructions on lines 13 and 14 set their values. (The space was allocated by the sub on line 3 in the function prologue.)

Q6.4
2 Points

Label `.L6` on line 17 begins the body of a loop. Which is the last instruction corresponding to the loop's body in the C code? Note: a C loop's body does not include the `for(...)` or `while(...)` portion.

- ○ Line 23: `str x2, [sp, 24]`
- ○ Line 25: `ldr x2, [x2, 8]`
- ✓ Either Line 23 or Line 25, depending on whether the C loop is a `for` or `while` loop.
- ○ Line 28: `bne .L6`
- ○ None of the above.

The last instruction that corresponds to the loop's purpose, as opposed to loop control, is Line 23 (which saves curr as maxNode).

If the corresponding C code is structured like a standard while loop, with the update step as the very last thing in the body, then Line 25 would also be inside the body, as that's the line that updates curr to be curr->next. If the corresponding C code is a for loop, however, that line is the third piece of the for(... ; ... ; ...) structure, and Line 23 is the final one within the body.

In either case, the bne instruction comes after the conditional, and thus Line 28 is part of the for or while structure, not the body.

So the best answer was the one that says it depends on the corresponding C code's details, however either 23 or 25 could be reasonable answers and thus earned partial credit.

Q6.5
1 Point

Lines 20 and 21 are comparing values corresponding to which C variable type?

- ○ `struct node`
- ✓ `unsigned long`
- ○ `Node_T`
- ○ `struct list`
- ○ `List_T`

On line 11, [x0] dereferences the function parameter -- so the parameter must not be an unsigned long, instead we should think whether it's a Node_T or a List_T (or another type of pointer, e.g., an unsigned long*).

On line 12, [x1] dereferences the value that was gotten from line 11. That wouldn't be possible if the function parameter were an unsigned long* or pointer to another primitive. If the function parameter were a Node_T, then the first dereference would give back its first field: an unsigned long, and thus it wouldn't be possible to dereference again, so that's also out. But if the function parameter were a List_T, then the first dereference would give back its first field: a Node_T, and the second dereference would give back the Node_T's first field: an unsigned long.

On line 13, the value loaded on line 12, which we now know is an unsigned long, gets stored into [sp, 16]. Finally, on line 19, that value is loaded into x0 to be used in the comparison and conditional branch that the question asks about.

(One could do the same logic with the other operand of cmp as well.)

Q6.6
1 Point

What is the purpose of Line 29 (`ldr x0, [sp,24]`)?

- [x] Copy the return value
- [ ] Clean up a local variable
- [ ] Restore a caller-saved register
- [ ] Restore a callee-saved register

Loading from the stack doesn't change the value on the stack, so there is no cleanup happening.

x0 is a caller saved register, but the value that is being loaded in is not necessarily the value that it originally held (which was stored at [sp, 8]), so this is not necessarily restoring its original value.

x0 is not a callee-saved register.

Line 29 loads the value of a local variable into x0 before continuing on sequentially to the epilogue and return. Since x0 is the register used for the return value, this behavior is copying the value to be returned into the appropriate register to return it.

Q6.7
1 Point

Nothing is stored at [SP, 0], thus a reasonable space optimization would be to move the data stored at stack offsets 8, 16, and 24 to 0, 8, and 16, instead, allowing us to change the first and penultimate instructions of the function to `sub sp, sp, 24` and `add sp, sp, 24`, respectively.

- [ ] True
- [x] False

AARCH64 requires that the stack pointer SP must be a multiple of 16, so it is not possible to decrement it by 24 in the prologue.

Q6.8
2 Points

The function `List_mystery` doesn't save or restore the value of `x30`. Choose the **best** answer for why this is okay:

- [ ] The function doesn't use the stack
- [ ] The function doesn't use callee-saved registers
- [ ] The function doesn't return a value
- [ ] The function isn't recursive
- [x] The function doesn't call any other functions

Partial credit: recursive functions, by definition, make function calls and thus must save and restore x30. But the more general answer is the better answer.

x30 is set by the bl instruction with the address of the next instruction after the bl. x30 is referenced by the ret instruction to do an unconditional jump to the address it contains.

Thus, if function f calls function g and g calls function h: when h returns, x30 will be an address in g -- and so the ret from g would not go back to f unless it is able to be restored to the value it had when g first began!

But if a function doesn't make any other function calls (and doesn't explicitly overwrite x30), then x30 will retain the correct address to return to and does not need to be saved to the stack.

Q7 Like Yen Sid from Fantasia.
16 Points

This question will again be dealing with the code for `List_mystery` from the previous Question, repeated here for convenience:

```
/* a Node_T is a member of a collection with an unsigned long as c
typedef struct node* Node_T;

struct node {
   /* contents of node */
   unsigned long payload;
   /* next node in the list */
   Node_T next;
};

struct list {
   /* head of the list */
   Node_T first;
   /* number of nodes in the list */
   unsigned long length;
};

/* a List_T is a collection of unsigned longs */
typedef struct list* List_T;
```

```
1       .global List_mystery
2  List_mystery:
3       sub     sp, sp, 32
4       str     x0, [sp,8]
5       ldr     x1, [x0,8]
6       cmp     x1, xzr
7       bne     .L2
8       mov     x0, 0
9       b       .L3
10 .L2:
11      ldr     x1, [x0]
12      ldr     x2, [x1]
13      str     x2, [sp,16]
14      str     x1, [sp,24]
15      ldr     x2, [x1,8]
16      b       .L4
17 .L6:
18      ldr     x3, [x2]
19      ldr     x0, [sp,16]
20      cmp     x3, x0
21      bls     .L5
22      str     x3, [sp,16]
23      str     x2, [sp,24]
24 .L5:
25      ldr     x2, [x2,8]
```

Here's the function in flattened C:

```
/* return pointer to the (first) node in l
   with the largest payload value,
   or NULL if l is empty */

Node_T List_mystery(List_T l) {
   unsigned long max; /* stored at [sp, 16] */
   Node_T maxNode;  /* stored at [sp, 24] */
   Node_T curr;         /* held in x2 */

   if(l->length != 0)
     goto L2;
   maxNode = NULL;
   goto L3;

L2:
   max = (l->first)->payload;
   maxNode = l->first;

   curr = l->first->next;
   goto L4;
L6:
   if(curr->payload <= max)
     goto L5;
   max = curr->payload;
   maxNode = curr;
L5:
   curr = curr->next;
L4:
   if(curr != NULL)
     goto L6;
L3:
   return maxNode;
}
```

```
26 .L4:
27      cmp     x2, xzr
28      bne     .L6
29      ldr     x0, [sp,24]
30 .L3:
31      add     sp, sp, 32
32      ret
```

Point allocation:
1: function return type `Node_T`.
2: function argument type `List_T` and syntax.
3: at least the three local variables
`max`, `maxNode`, and `curr`.
OK if extra variables for
`l->first`, `l->first->payload`,
or `curr->payload`.
4: `if` conditional + return setup.
OK if conditional isn't inverted.
OK if it does `return NULL;`
directly instead of assigning
and jumping to unified return.
5: `max` initialization.
6: `maxNode` initialization.
7: `curr` initialization: must start at l->first->next not l->first!
8: loop control labels and gotos (L4 and L6)
9: while loop condition
10: in-loop conditional + goto L5
11: update `max` and `maxNode`
12: L5 + update `curr`
13: `return maxNode`
14: comment mentions
argument by name
15: comment mentions
return + normal behavior
(no need to specify tie case)
16: comment mentions
return NULL for empty list

Q7.1
16 Points

Translate the `List_mystery` function into "Flattened C", using the same labels as the given assembly language code. Include a function comment for `List_mystery` that meets the requirements from your programming assignments in this course.

Enter your answer here

Q8 Avengers Disassemble!
9 Points

Consider the following `objdump` output:

```
armlab02$ objdump --disassemble --reloc simple.o

simple.o:     file format elf64-littleaarch64


Disassembly of section .text:

0000000000000000 <main>:
   0:   a9be7bfd        stp     x29, x30, [sp,#-32]!
   4:   910003fd        mov     x29, sp
   8:   90000000        adrp    x0, 0 <main>
                        8: R_AARCH64_ADR_PREL_PG_HI21   .rodata
   c:   91000000        add     x0, x0, #0x0
                        c: R_AARCH64_ADD_ABS_LO12_NC    .rodata
  10:   f9000fa0        str     x0, [x29,#24]
  14:   f9400fa0        ldr     x0, [x29,#24]
  18:   94000000        bl      0 <strlen>
                        18: R_AARCH64_CALL26    strlen
```

```
1c:   aa0003e1        mov     x1, x0
20:   90000000        adrp    x0, 0 <main>
                      20: R_AARCH64_ADR_PREL_PG_HI21   .rodata+0x
24:   91000000        add     x0, x0, #0x0
                      24: R_AARCH64_ADD_ABS_LO12_NC    .rodata+0x
28:   94000000        bl      0 <printf>
                      28: R_AARCH64_CALL26    printf
2c:   52800000        mov     w0, #0x0                        //
30:   a8c27bfd        ldp     x29, x30, [sp],#32
34:   d65f03c0        ret
```

Q8.1
1 Point

simple.o also contains additional information, for example strings defined in this file that are stored in the RODATA section.

A slightly different objdump command would show them:

Does this output represent the full contents of `simple.o`?

- ○ Yes
- ✓ No

armlab02$ objdump -s simple.o | head -n 12

simple.o: file format elf64-littleaarch64

Contents of section .text:
0000 fd7bbea9 fd030091 00000090 00000091 .{..............
0010 a00f00f9 a00f40f9 00000094 e10300aa ......@.........
0020 00000090 00000091 00000094 00008052 ....R
0030 fd7bc2a8 c0035fd6 .{...._.
Contents of section .rodata:
0000 5a6f6f6d 20556e69 76657273 69747920 Zoom University
0010 2d20434f 53203231 370a0000 00000000 - COS 217.....
0020 4c656e67 74683a20 256c750a 00 Length: %lu..

Q8.2
1 Point

What is `0:` on the line beginning with that same prefix?

- ○ Assembly language code
- ○ Machine language code
- ○ Address
- ✓ Offset
- ○ Relocation record
- ○ Data value

These are offsets. .o files are generated by the assembler, but final addresses cannot be generated until the link stage.

Q8.3
1 Point

What is `910003fd` on the line prefixed with `4:`?

- ○ Assembly language code
- ☑ Machine language code
- ○ Address
- ○ Offset
- ○ Relocation record
- ○ Data value

This is machine language code:
the 8 hexits are shorthand for the 32 bits
of the machine language instruction as
seen in the final lecture.

Q8.4
1 Point

What is `R_AARCH64_ADR_PREL_PG_HI21 .rodata` on the (indented) second line prefixed with `8:`?

- ○ Assembly language code
- ○ Machine language code
- ○ Address
- ○ Offset
- ☑ Relocation record
- ○ Data value

Q8.5
1 Point

Which software produced the `R_AARCH64_ADR_PREL_PG_HI21 .rodata` on the (indented) second line prefixed with `8:`?

- ○ Preprocessor
- ○ Compiler
- ✓ Assembler
- ○ Linker
- ○ Standard Library

Relocation records are produced by the assembler, because it cannot yet know the final address that extern data and functions will inhabit in the final program's virtual memory.

Q8.6
1 Point

Which software is the intended consumer of the `R_AARCH64_ADR_PREL_PG_HI21 .rodata` on the (indented) second line prefixed with `8:`?

- ○ Preprocessor
- ○ Compiler
- ○ Assembler
- ✓ Linker
- ○ Standard Library

The linker is responsible for handling relocation records from the assembler and correcting instructions that require the final (relative) addresses in a program.

Q8.7
2 Points

Resolving a `R_AARCH64_CALL26` construct could change which byte(s) in the corresponding `bl` instruction? For this question, bytes are numbered 0 (least significant) through 3 (most significant).

- [x] All four (bytes 0-3)
- [ ] The three least significant (bytes 0-2)
- [ ] The two least significant (bytes 0-1)
- [ ] The three most significant (bytes 1-3)
- [ ] The two most significant (bytes 2-3)
- [ ] Only a single byte
- [ ] No change will be made by processing `R_AARCH64_CALL26`

The three least significant bytes are changed in their entirety (except for any bits of the relative address that happen to be 0).

But the most significant byte is also changed in its lower order bits: the relative address is 26 bits, which requires an additional 2 bits beyond the 3 entire least significant bytes, and thus 2 bits of the most significant byte.

Q8.8
1 Point

While resolving a `R_AARCH64_CALL26` construct, from where will the machine code defining `strlen` or `printf` be sourced?

- [ ] `simple.h`
- [ ] `simple.c`
- [ ] `simple.s`
- [ ] `simple.o`
- [x] `libc.a` or `libc.so`
- [ ] `string.h` or `stdio.h`, respectively
- [ ] `strlen.o` or `printf.o`, respectively

The functions strlen and printf are from the C standard library. The standard library's object files, which contain (among other data) the machine language instructions that these files define in the TEXT section, are found in a well-defined location known to the linker.

In the case of static linking, which is all we've talked about in this class, the object files are bundled together into an archive file: libc.a. On armlab, that file is found at /usr/lib64/libc.a. (libc.so is a similar file used in dynamic linking, in which the full set of object code is not stored in the final executable, but instead referenced from the executable to its place in the "shared object" file.)

Q9 Dr. Moretti, not Rapunzel, has hidden it -- somewhere you'll never find it.
1 Point

Note any easter egg (sub-second flash image with a pop culture or historical reference) from any COS 217 lecture video. As an alternative, list any similarly tortured stretch of a pop culture or historical reference given in a lecture's narration, even if it did not make an appearance on the slides. (Feel free to list as many as you

noticed and recall -- for better or worse! -- but a single one will do for the bonus point.)

Enter your answer here

There were dozens, not even counting the theme of these exam question names. Popular responses included:
* Tributes to Chadwick Boseman, Alex Trebek, and Eddie Van Halen.
* Lyrics referenced by their singers, e.g. Elsa from Frozen (at least twice) and Uma from Descendants 2.
* Memes, e.g. Gru's plan, Spiderman pointing at Spiderman, and Roll Safe.
* Movie/Show image references, e.g. LotR, Monty Python, Lady and the Tramp, Beethoven, Winnie-the-Pooh, Thomas the Tank Engine, and Downton Abbey.

Q10 The wonderful thing about Honor Codes is Honor Codes are wonderful things

0 Points

Copy the Honor pledge in the field below:

I pledge my honor that I have not violated the Honor Code during this examination.

Enter your answer here

Enter your name in the field below, attesting to the Honor pledge you have copied above:

Enter your answer here

Submit & View Submission ›