

CSE 351 FINAL

Last Name:		
First Name:		
Student ID Number:		
Name of person to your Left Right:		
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE351 who haven't taken it yet. Violation of these terms could result in a failing grade and a report to CSSC. (please sign)		

Do not turn the page until 14:30.

Instructions

- This exam contains 10 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices). You are allowed two pages (US letter/A4, double-sided) of handwritten notes. Scientific calculators are allowed.
- Please silence and put away all cell phones and other mobile or noise-making devices.
- Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read all questions first and start where you feel the most confident.
- Relax. You are here to learn. You can do it! 🍀

Question M1: Number Representation

A) Suppose we have a variable x whose hexadecimal representation is $0xFF\ 00\ 00\ 00$
(Note that any leading zeros are omitted).

- 1) Assuming x was encoded as one of the following types, which type(s) would make x negative? CIRCLE ALL THAT APPLY:

unsigned int **int** **float** long unsigned long

- 2) Assuming x was encoded as one of the following types, which type(s) would make x the most positive? If multiple answers are equal, CIRCLE them all:

unsigned int int float **long** **unsigned long**

- 3) Assuming x was encoded as one of the following types, which type(s) would make x have the largest magnitude? If multiple answers are equal, CIRCLE them all:

unsigned int int **float** long unsigned long

B) Suppose we have:

```
signed int z = [unknown];      signed short s = (signed short) z;
```

If s has the hexadecimal representation $0x8000$, which value(s) could replace [unknown] so that z and s have the same decimal value? CIRCLE ALL THAT APPLY:

$0x00008000$ **$0xFFFF8000$** $0x88888000$ $0x11118000$ $0x10008000$

Question M2: Pointers and Memory

For this problem we are using a 64-bit x86-64 machine (little-endian). A partial view of the current state of memory (values in hex) is shown below:

	Word Addr	+0	+1	+2	+3	+4	+5	+6	+7
	0xC8	82	55	72	AB	CE	0A	B1	00
	0xD0	91	A2	B3	6E	74	00	00	00
<i>// num is shown in memory</i>	0xD8	5A	00	88	77	66	55	44	33
char* msg = "wow!";	0xE0	00	00	00	74	20	30	00	69
int num = 0x74;	0xE8	68	6F	6D	65	00	64	32	55
char* str = 0xE8;									

A) How many bytes are allocated by the declarations and initializations in the code above?

25 bytes

B) What is the result of `printf("%s", str)`, printing `str` as a string? A partial ASCII chart is provided for reference.

Hex	0x64	0x65	0x66	0x67	0x68	0x69	0x6A	0x6B	0x6C	0x6D	0x6E	0x6F	0x70
Char	d	e	f	g	h	i	j	k	l	m	n	o	p

home

C) For the following C expressions, compute their **C type** and (hexadecimal) **value** (no leading zeroes). Then, assuming `%rdi` holds `str` and `%rsi` holds `&num`, fill in the blanks for the corresponding x86-64 instructions.

Expression	C Type	Hex Value	X86-64 Instruction
<code>dl = *(str + 1)</code>	<code>char</code>	<code>0x6F</code>	<code>movb 1(%rdi), %dl</code>
<code>rcx = &num - 2</code>	<code>int*</code>	<code>0xCC</code>	<code>leaq -8(%rsi), %rcx</code>

Question M3: Design Questions

A) Which of these are **benefits** of having function frames (local variables, return address, etc.) on the Stack instead of in blocks in the Heap? SELECT ALL THAT APPLY:

- Allocating and deallocating memory is faster on the Stack than in the Heap
- Saving a register value by pushing it on the Stack is more memory efficient than storing it in a new block in the Heap
- Consecutively called functions have contiguous frames in memory, so functions can access additional arguments in their caller's frame
- LIFO structure allows for easy access to the return address when exiting a function
- Functions can allocate data that persist after they return without leaking memory

B) Describe **one benefit** of using the Heap for user input instead of in the Stack:

Describe (1-2 sentences):

- Avoids stack smashing by dynamically allocating memory as needed at runtime
- Potentially malicious user input is spatially separated from any program-necessary memory (i.e. return addresses, saved registers, instructions, etc.)
- Data in the Heap has longer life than data on the Stack, letting functions pass pointers to user input back to callers

C) Without changing anything else about x86 register conventions, could we change %rax to be *callee-saved* instead of *caller-saved*? Explain why or why not.

Can change to callee-saved? (CIRCLE ONE):

Yes

No

Explain (1-3 sentences):

Callee-saved values need to be restored to their original values when the callee function returns. However, %rax is the return register, so it needs to be changed to the callee function's return value when the callee function returns. Thus, there is no way for %rax to be both callee-saved and the return register.

Question M4: C & Assembly

Consider the following x86-64 assembly for a function `mystery`:

```
mystery:
    pushq   %rbx                # Line 1
    movq    %rdi, %rax          # Line 2
    movq    %rax, %rbx         # Line 3
    jmp     .L2                 # Line 4
.L3:
    movb    (%rax), %dl         # Line 5
    xor     $0x20, %dl          # Line 6
    movb    %dl, (%rax)        # Line 7
    addq    $1, %rax           # Line 8
.L2:
    movzbl  (%rax), %edx        # Line 9
    testb   %dl, %dl           # Line 10
    jne     .L3                 # Line 11
    movq    %rbx, %rax         # Line 12
    popq    %rbx               # Line 13
    ret                                # Line 14
```

A) Fill in the missing C code that is equivalent to the x86-64 assembly above:

```
char* mystery ( char* param) {
    char* var = param;
    while ( *var != 0 ) {
        *var ^= 0x20;
        var += 1;
    }
    return param;
}
```

B) If we called `mystery` with "Abacadaba" as a parameter, what would be returned from the function call? Answer in the format " "

Hint: in ASCII, 'a' is 0x61, 'b' is 0x62, 'c' 0x63, ..., 'A' is 0x41, 'B' 0x42, 'C' 0x43...

`aBACADABA`

For the following instructions, write an equivalent line(s) of assembly using the replacement instruction(s) suggested.

C) `popq %rbx`

`movq \(%rsp\), %rbx`

`addq \$8, %rsp`

D) `retq`

`movq \(%rsp\), %r10`

`addq \$8, %rsp`

`jmp *%r10 // hint: a register may go in this blank _(0o0)_/`

Question M5: Procedures & The Stack

Consider the following C code, which defines a struct for a linked list node and finds the sum of a linked list of nodes:

```
// struct definition
struct ll_node {
    int data;
    struct ll_node* next;
};

// find the sum of a linked list
int sum_ll(struct ll_node* start) {
    if (start->next == 0) {
        return start->data;
    }
    return start->data +
        sum_ll(start->next);
}

// create a linked list and sum it
int main() {
    // 0 -> 1 -> 2 -> 3 -> 4
    struct ll_node four = {4, 0};
    struct ll_node three = {3, &four};
    struct ll_node two = {2, &three};
    struct ll_node one = {1, &two};
    struct ll_node root = {0, &one};

    int sum = sum_ll(&root);
    printf("%d\n", sum);
}
```

Also consider the disassembly for sum_ll:

```
#: 0000000000401126 <sum_ll>:
1   401126: 48 8b 47 08          mov     0x8(%rdi),%rax
2   40112a: 48 85 c0             test   %rax,%rax
3   40112d: 74 0f              je     40113e <sum_ll+0x18>
4   40112f: 53                push   %rbx
5   401130: 8b 1f             mov    (%rdi),%ebx
6   401132: 48 89 c7          mov    %rax,%rdi
7   401135: e8 ec ff ff ff    call  401126 <sum_ll>
8   40113a: 01 d8            add   %ebx,%eax
9   40113c: 5b              pop   %rbx
10  40113d: c3              ret
11  40113e: 8b 07          mov   (%rdi),%eax
12  401140: c3              ret
```

A) What is the return address to sum_ll that is saved on the stack for the recursive calls of sum_ll?

0x40113a

B) How many bytes does sum_ll take up in its executable?

27 bytes

C) How many **total stack frames** are created by the above program?

7 frames

D) Fill out the stack snapshot below at the point where the stack is largest. The number of boxes is arbitrary. Use ??? for unknown words.

	Value	Description
...	<...>	<main's stack frame>
0x7fffffffdf8	???	<ret addr to main>
0x7fffffffdf0	???	main's %rbx value
0x7fffffffdf8	0x40113a	ret addr to sum_ll(root)
0x7fffffffdf0	0	sum_ll(root)'s %rbx
0x7fffffffdf8	0x40113a	ret addr to sum_ll(one)
0x7fffffffdf0	1	sum_ll(one)'s %rbx
0x7fffffffdfc8	0x40113a	ret addr to sum_ll(two)
0x7fffffffdfc0	2	sum_ll(two)'s %rbx
0x7fffffffdfb8	0x40113a	ret addr to sum_ll(three)
0x7fffffffdfb0		
0x7fffffffdfa8		
0x7fffffffdfa0		

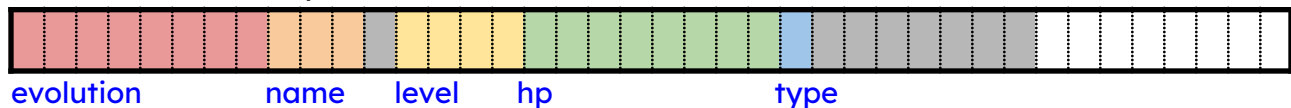
Question F6: Structs

Consider the following definition for a struct pokemon:

```
typedef struct {
    short* evolution;
    char name[3];
    int level;
    long hp;
    char type;
} pokemon;
```

- A) Use the box below to draw out the bytes in an instance of the pokemon struct. Clearly **label each field and its size (in bytes)**. Clearly **specify any internal and external fragmentation**. **Draw a clear line indicating the end of the struct**; some boxes may be unnecessary. Then, state the amount of internal and external fragmentation, as well as the total size of the struct (including any fragmentation).

pokemon (1 box = 1 byte):



Internal Fragmentation:

1 bytes

External Fragmentation:

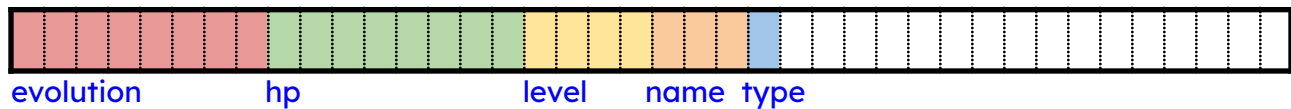
7 bytes

Total Size:

32 bytes

- B) Can we reorder the fields in pokemon to reduce the size of an instance? If so, draw out the diagram using the same rules as above. If not, provide a justification of why. You only need to draw out the diagram or provide a justification, **not both**.

Smaller pokemon struct layout (**if possible**) (1 box = 1 byte):



Justification (**if not possible**) (1-2 sentences):

It's possible!

C) Suppose you have an array that contains 3 instances of pokemon defined as follows:

```
pokemon* pokemons[] = {pokemon1, pokemon2, pokemon3};
```

What is an expression of C code that would set the evolution field of pokemon3 to be the same as the evolution field of pokemon1?

Your expression **CANNOT contain** the variables pokemon1, pokemon2, or pokemon3.

C Expression:

```
pokemons[2]->evolution = (*pokemons[0]).evolution;
```

Question F7: Caching

A **next-line prefetcher** is a component that **fetches the next cache block** early. **On a cache miss**, the cache will pull in a requested block from memory. At the same time, the next-line prefetcher will **also pull in the block that immediately follows in memory**.

Assume a cache with a next-line prefetcher starts cold and has the following parameters:

- Associativity: 1
- Block size: 8 B
- Capacity: 512 B
- Write-back, write-allocate
- LRU replacement policy

Consider the following C code snippets:

```
// Snippet 1
int sum = 0;
int arr[1024]; // &arr[0] = 0x10
for (int i = 0; i < 1024; i++) {
    sum += arr[i];
}
```

```
// Snippet 2
#define LEAP = 4
int sum = 0;
int arr[1024]; // &arr[0] = 0x10
for (int i = 0; i < 1024; i += LEAP) {
    sum += arr[i];
}
```

A) Assuming `sum` and `i` are stored in registers, calculate the miss rate of each snippet on the above cache as a percentage:

Snippet 1:

25%

Snippet 2:

100%

B) Compared to an identical cache with no prefetching, does the presence of the next-line prefetcher improve the performance of Snippet 1? Why or why not?

Performance (CIRCLE ONE):

Improves

Does NOT improve

Explain (1-3 sentences):

Snippet 1 will perform better with a next-line prefetcher because accesses to `arr` are sequential and since arrays are contiguous in memory, the next-line prefetcher will always fetch something useful.

C) Compared to an identical cache with no prefetching, does the presence of the next-line prefetcher improve the performance of Snippet 2? Why or why not?

Performance (CIRCLE ONE):	Improves	Does NOT improve
<u>Explain (2-3 sentences):</u> Snippet 2 will perform the same with or without a next-line prefetcher because the stride (4 * 4 bytes per int = 16 bytes) is the size of 2 cache blocks. This means the block we actually need will not be fetched into the cache early, and will still cause a cache miss.		

D) Which cache set does the block associated with `arr[7]` map to?

Set 5

E) Which of the following array elements does **NOT** map to the same set as `arr[7]`? (Circle one):

`arr[6]` `arr[134]` **`arr[136]`** `arr[262]`

F) For each of the following proposed changes, CIRCLE ONE option that describes its effect on the **miss rate** of the code snippets above. Assess each change **independently**: assume all other factors remain the same as they were in the original cache.

1) Change the associativity from 1-way to 2-way: Increase **No Change** Decrease

2) Change LEAP in Snippet 2 from 4 to 3: Increase No Change **Decrease**

3) Change the cache size from 512 B to 1 KiB: Increase **No Change** Decrease

4) Change the cache replacement policy to Least **frequently** used: Increase **No Change** Decrease

Question F8: Memory Allocation

A) Consider the following C code snippet:

```
#:
1  #include <stdlib.h>
2
3  int NUM_STUDENTS = 30;
4  void avg_exam_score(float* midterm, float* final) {
4     char* course = "CSE351";
5     float* grades = (float*) malloc(NUM_STUDENTS * sizeof(float));
6     float* start = grades;
7
8     for (int i = 0; i < NUM_STUDENTS; i++) {
9         *grades = (midterm[i] + final[i]) / 2;
11        grades++;
12    }
13    free(grades);
14 }
```

- 1) This snippet has a memory-related bug. Identify the line number of the bug and write one line of C code to replace the buggy line that fixes the bug.

Line number: 13

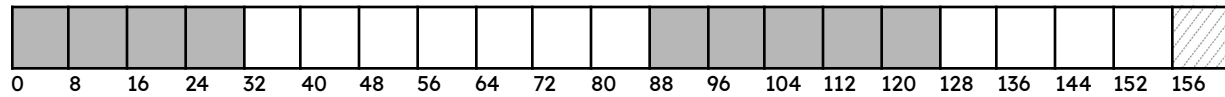
Fix (1 line of C code): `free(start)`

- 2) For the groups of expressions below, CIRCLE the one in each group whose returned **value** is **largest/highest**. Assume the malloc call succeeds and all variables are stored in memory (not registers).

Group 1:	&NUM_STUDENTS	*course	start
Group 2:	course	&start	NUM_STUDENTS
Group 3:	&course	grades	&avg_exam_score

B) For the following questions, consider a dynamic allocator on a 64-bit machine with 8-byte alignment, 8-byte boundary tags, and an explicit free list. Allocated blocks (shaded) have only a header, and free blocks have a header and a footer.

Answer the following questions given the current state of the heap below. Each box represents one word; the final box (address 156) is the end-of-heap block.



- 1) What is the largest payload that can be allocated without requesting more space on the heap?

malloc (48)

- 2) Assuming a best-fit allocation strategy, what pointer would be returned by a call to `malloc(2 * sizeof(int))`? Format your answer in decimal.

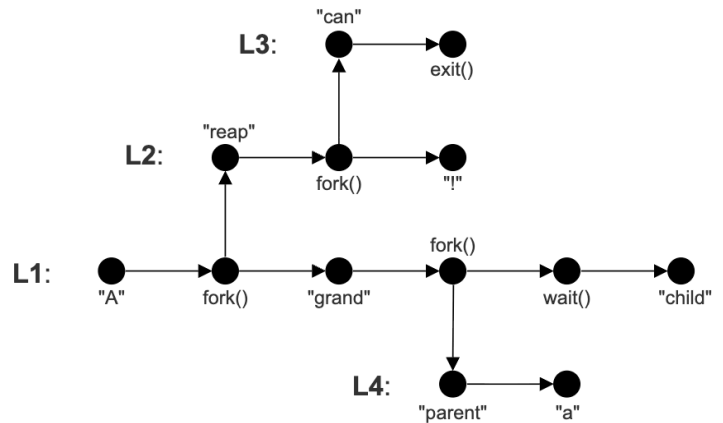
136

Question F9: Processes

A) Fill in the blanks in the C code to match the behavior of the process diagram. The labels L1-L4 number the four processes created.

```
printf("A");
if (fork()) {
    printf("grand");
    if (fork()) {
        wait();
        printf("child");
    } else {
        printf("parent");

        printf("a");
    }
} else {
    printf("reap");
    if (fork()) {
        printf("!");
    } else {
        printf("can");
        exit(0);
    }
}
```



B) For each of the following outputs, CIRCLE whether the output is possible or not possible. If an output is not possible, **CIRCLE the first invalid word** in the output.

"A grand reap ! can parent a child"	Possible	Not possible
"A reap grand a parent can ! child"	Possible	Not possible
" child A can ! reap grand parent a"	Possible	Not possible
"A grand parent can reap a child !"	Possible	Not possible

⚠ Did you remember to **CIRCLE the first invalid word** on all impossible outputs? ⚠

C) Which process is L3's parent, if any? CIRCLE ONE option:

L1 **L2** L3 L4 None

D) Which process is L3's grandparent, if any? CIRCLE ONE option:

L1 L2 L3 L4 None

E) When process L3 terminates, who will reap it? CIRCLE ONE option:

init/systemd **L3's parent** L3's grandparent Elba

Question F10: Virtual Memory

A) Circle ONE option to indicate whether these events occur always, sometimes, or never in a single address translation:

1) TLB hit → segmentation fault	Always	<u>Sometimes</u>	Never
2) Page table hit → page fault	Always	Sometimes	<u>Never</u>
3) TLB miss → check page table	<u>Always</u>	Sometimes	Never
4) TLB hit → cache hit	Always	<u>Sometimes</u>	Never

B) Circle ONE option to indicate whether the following statements about virtual memory are true or false.

1) Main memory acts as a “cache” for swap space on disk.	<u>True</u>	False
2) As part of a context switch, we must invalidate the TLB.	<u>True</u>	False
3) Every virtual page is mapped to a location either in physical memory or on disk.	True	<u>False</u>
4) The widths of the virtual page offset and the physical page offset fields are always the same.	<u>True</u>	False
5) Multiple processes can share a single page table.	True	<u>False</u>

C) For a system with the following properties:

- 32-bit virtual addresses,
- 20-bit physical addresses,
- 4 KiB virtual pages,
- An 8-entry, 4-way associative TLB,

Calculate the following:

1 TLB Index bits 19 TLB Tag bits 2²⁰ Page table entries

This page is purposely left blank and can be used for scratch work.