Sp17 Midterm Q1

1. Integers and Floats (7 points)

a. In the card game Schnapsen, 5 cards are used (Ace, Ten, King, Queen, and Jack) from 4 suits, so 20 cards in total. What are the minimum number of bits needed to represent a single card in a Schnapsen deck?

5

We need 2 bits to represent 4 suits, and 3 bits to represent 5 ranks. So 5 bits in total.

b. How many negative numbers can we represent if given 7 bits and using two's complement?

$$\mathbf{2^6}$$

Using 7 bits, the MSB has to be 1 for negative numbers. So there are 2^6 negative numbers in total.

Consider the following pseudocode (we've written out the bits instead of listing hex digits):

```
int a = 0b0100 0000 0000 0000 0000 0011 1100 0000
int b = (int)(float)a
int m = 0b0100 0000 0000 0000 0000 0011 0000 0000
int n = (int)(float)m
```

c. Circle one: True or [✓ False]:

`a == b`

The right-most 1 will be truncated (cannot fit in Mantissa)

d. Circle one: [✓ True] or False:

`m == n`

No precision will be lost

e. How many IEEE single precision floating point numbers are in the range [4, 6) (That is, how many floating point numbers are there where 4 <= x < 6?)

$$\mathbf{2^{22}}$$

4 in binary is $1.0 \cdot 2^2$.

6 in binary is $1.1 \cdot 2^2$.

So in Mantissa the right-most 22 bits can be either 0 or 1. Therefore, there are 2^{22} bits in range $[4, 6)$

Au17 Final M3

Question M3: Pointers & Memory [8 pts]

For this problem we are using a 64-bit x86-64 machine (**little endian**). Below is the `count_nz` function disassembly, *showing where the code is stored in memory.*

```
0000000000400536 <count_nz>:
  400536:  85 f6            testl  %esi,%esi
  400538:  7e 1b            jle    400555 <count_nz+0x1f>
  40053a:  53               pushq  %rbx
  40053b:  8b 1f            movl   (%rdi),%ebx
  40053d:  83 ee 01         subl   $0x1,%esi
  400540:  48 83 c7 04      addq   $0x4,%rdi
  400544:  e8 ed ff ff ff   callq  400536 <count_nz>
  400549:  85 db            testl  %ebx,%ebx
  40054b:  0f 95 c2         setne  %dl
    ... some instructions omitted here ...
```

(A) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Use the appropriate bit widths. Hint: what is the *value* stored in `%rsi`? [4 pt]

```
leal 2(%rdi, %rsi), %eax
movw (%rdi,%rsi,4), %bx
```

Register	Value (hex)
%rdi	0x 0000 0000 0040 0544
%rsi	0x FFFF FFFF FFFF FFFF
%eax	0x **0040 0545**
%bx	0x **8348**

`leal` instruction calculates the address `0x400544` + (-1) + 2 = `0x400545`.

`movw` instruction pulls two bytes starting at memory address `0x400544`+4*(-1) = `0x400540`, which is `0x48` and `0x83`. Remember little-endian!

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let **`char* charP = 0x400544`**. [4 pt]

```
char v1 = *(charP + __6__);                    // set v1 = 0xDB

int* v2 = (int*)((___double___*)charP - 2);    // set v2 = 0x400534
```

The only `0xDB` byte in `count_nz` is found at address `0x40054a`, 6 bytes beyond `charP`.

The difference between `v2` and `charP` is 16 bytes. Since by pointer arithmetic we are moving 2 "things" away, `charP` must be cast to a pointer to a data type of size 8 bytes. Long or any pointer (except void*) also accepted.

Au18 Midterm Q5

Question 5: Procedures & The Stack [24 pts]

The recursive function `sum_r()` calculates the sum of the elements of an `int` array and its x86-64 disassembly is shown below:

```
int sum_r(int *ar, unsigned int len) {
    if (!len) {
        return 0;
    else
        return *ar + sum_r(ar+1,len-1);
}
```

```
0000000000400507 <sum_r>:
  400507:  41 53           pushq  %r12
  400509:  85 f6           testl  %esi,%esi
  40050b:  75 07           jne    400514 <sum_r+0xd>
  40050d:  b8 00 00 00 00  movl   $0x0,%eax
  400512:  eb 12           jmp    400526 <sum_r+0x1f>
  400514:  44 8b 1f        movl   (%rdi),%r12d
  400517:  83 ee 01        subl   $0x1,%esi
  40051a:  48 83 c7 04     addq   $0x4,%rdi
  40051e:  e8 e4 ff ff ff  callq  400507 <sum_r>
  400523:  44 01 d8        addl   %r12d,%eax
  400526:  41 5b           popq   %r12
  400528:  c3              retq
```

(A) The addresses shown in the disassembly are all part of which section of memory? [2 pt]

Text or `.text` also accepted.

Instructions/Code

(B) *Disassembly* (as shown here) is different from *assembly* (as would be found in an assembly file). Name two major differences: [4 pt]

Differences: Some possible answers include:

- No machine code (middle column) would be shown in the assembly (*i.e.* the code hasn't been assembled yet).
- Finalized addresses would not be found in the assembly (left column).
- All labels would still be symbolic/named in the assembly instructions (*e.g.* `jne`, `jmp`, `callq`).

(C) What is the return address to `sum_r` that gets stored on the stack? Answer in hex. [2 pt]

The address of the instruction *after* `call`.

0x **400523**

(D) What value is saved across each recursive call? Answer using a *C expression*. [2 pt]

The instruction at address `0x400514` dereferences `%rdi` and stores the value in `%r12d`.

***ar**

(E) Assume `main` calls `sum_r(ar,3)` with `int ar[] = {3,5,1}`. Fill in the snapshot of memory below the top of the stack **in hex** as this call to `sum_r` returns to `main`. For unknown words, write "`0x unknown`". [6 pt]

0x7fffffffde20	<ret addr to main>	sum_r(ar,3)
0x7fffffffde18	<original r12>	
0x7fffffffde10	0x **400523 <ret addr>**	sum_r(ar+1,2)
0x7fffffffde08	0x **3 <*ar>**	
0x7fffffffde00	0x **400523 <ret addr>**	sum_r(ar+2,1)
0x7fffffffddf8	0x **5 <*ar>**	
0x7fffffffddf0	0x **400523 <ret addr>**	sum_r(ar+3,0)
0x7fffffffdde8	0x **1 <*ar>**	

The base case DOES still push `%r12` onto the stack.

(F) Assembly code sometimes uses *relative addressing*. The last 4 bytes of the `callq` instruction encode an integer (in *little endian*). This value represents the difference between which two addresses? Hint: both addresses are important to this `callq`. [4 pt]

`0xffffffe4 = -(0x1b + 1) = -28` value (decimal): **-28**

This corresponds to the address we jump to. address 1: 0x **400507**

This corresponds to the return address. address 2: 0x **400523**

(G) What could we change in the assembly code of this function to **reduce the amount of Stack memory used** while keeping it *recursive* and *functioning properly*? [4 pt]

The issue with recursive functions is that no matter what kind of register you use to save a value (caller-saved or callee-saved), the recursive call will overwrite that value because it's an identical function! So we actually *can't* avoid pushing something to the stack without making the function iterative. So any potential saving of Stack space will come from the base case. Keep reading for two possible solution types:

Callee-saved: `%r12` is a *callee*-saved register. This means that its old value just needs to be saved before we overwrite its value; it does not need to be saved at the very top of `sum_r`.

1) Move the `pushq` instruction into the recursive case (below the `jmp` instruction).

2) Either make the `jmp` go to address `0x400528` instead OR
move the `movl $0,%eax` above the `jne` and change the `jne` to `je 0x400528`.

Caller-saved: The value we really care about saving across the recursive call (`ar` or `*ar`), already starts in a caller-saved register in `%rdi`! This value must then be saved before we make a recursive call to `sum_r` and restored once it returns:

1) Convert the `pushq %r12` to `pushq %rdi` and move it down to *replace* the `movl (%rdi),%r12d` instruction.

2) Convert the `popq %r12` to `popq %rdi` and move it right after/below the `callq`.

3) Convert the `addl %r12d,%eax` to `addl (%rdi),%eax`.

Wi17 Final Q1

1. C and Assembly (15 points)

Consider the following (partially blank) x86-64 assembly, (partially blank) C code, and memory listing. Addresses and values are 64-bit, and the machine is little-endian. All the values in memory are in hex, and the address of each cell is the sum of the row and column headers: for example, address 0x1019 contains the value 0x18.

Assembly code:

```
foo:
  movl $0, %eax

L1:
  cmpq 0x0, %rdi
  je L2
  cmp 0x18, 0x1(%rdi)
  je L3
  mov 0x8(%rdi), %rdi
  jmp L1

L2:
  ret

L3:
  mov (%rdi), %eax
  jmp L2
```

C code:

```
typedef struct person {
  char height;
  char age;
  struct person* next_person;
} person;

int foo(person* p) {
    int answer = 0;
    while (p != NULL) {
        if (p->age == 24){
          answer = p->height;
          break;
        }
        p = p->next_person;
    }
    return answer;
}
```

Memory Listing
Bits not shown are 0.

	0x00	0x01	...	0x05	0x06	0x07
0x1000	80	1B	...	00	00	00
0x1008	80	1B	...	00	00	00
0x1010	3F	18	...	00	00	00
0x1018	3F	18	...	00	00	00
0x1020	00	00	...	00	00	00
0x1028	18	10	...	00	00	00
0x1030	18	10	...	00	00	00
0x1038	40	40	...	00	00	00
0x1040	40	40	...	00	00	00
0x1048	00	00	...	00	00	00

(a) Given the code provided, fill in the blanks in the C and assembly code.

(b) Trace the execution of the call to `foo((person*) 0x1028)` in the table to the right. Show which instruction is executed in each step until `foo` returns. In each space, place **the assembly instruction** and the values of the appropriate registers **after that instruction executes**. *You may leave those spots blank when the value does not change.* You might not need all steps listed on the table.

Instruction	`%rdi` (hex)	`%eax` (decimal)
`movl`	`0x1028`	0
`cmpq`		
`je`		
`cmp`		
`je`		
`mov`	`0x1018`	
`jmp`		
`cmpq`		
`je`		
`cmp`		
`je`		
`mov`		63
`jmp`		
`ret`		

(c) Briefly describe the value that `foo` returns and how it is computed. Use only variable names from the C version in your answer.

foo traverses a linked list of person structs, and returns the height of the first person whose age == 24.

Au16 Final F5

Question F5: Caching [10 pts]

We have 16 KiB of RAM and two options for our cache. Both are two-way set associative with 256 B blocks, LRU replacement, and write-back policies. **Cache A** is size 1 KiB and **Cache B** is size 2 KiB.

(A) Calculate the TIO address breakdown for **Cache B**: [1.5 pt]

Tag bits	Index bits	Offset bits
4	**2**	**8**

14 address bits. $\log_2 256 = 8$ offset bits. 2 KiB cache = 8 blocks. 2 blocks/set → 4 sets.

(B) The code snippet below accesses an integer array. Calculate the **Miss Rate** for **Cache A** if it starts *cold*. [3 pt]

```
#define LEAP 4
#define ARRAY_SIZE 512
int nums[ARRAY_SIZE];            // &nums = 0x0100 (physical addr)
for (i = 0; i < ARRAY_SIZE; i+=LEAP)
    nums[i] = i*i;
```

1/16

Access pattern is a single write to `nums[i]`. Stride = LEAP = 4 ints = 16 bytes. 256/16 = 16 strides per block. First access is a compulsory miss and the next 15 are hits. Since we never revisit indices, this pattern continues for all cache blocks. You can also verify that the offset of &nums is 0x00, so we start at the beginning of a cache block.

(C) For each of the proposed (independent) changes, write **MM** for "higher miss rate", **NC** for "no change", or **MH** for "higher hit rate" to indicate the effect on **Cache A** for the code above:[3.5 pt]

Direct-mapped _NC_ Increase block size _MH_

Double LEAP _MM_ Write-through policy _NC_

Since we never revisit blocks, associativity doesn't matter. Larger block size means more strides/block. Doubling LEAP means fewer strides/block. Write hit policy has no effect.

(D) Assume it takes 200 ns to get a block of data from main memory. Assume **Cache A** has a hit time of 4 ns and a miss rate of 4% while **Cache B**, being larger, has a hit time of 6 ns. What is the worst miss rate Cache B can have in order to perform as well as Cache A? [2 pt]

0.03 or 3%

$\text{AMAT}_A = \text{HT}_A + \text{MR}_A \times \text{MP} = 4 + 0.04*200 = 12$ ns.

$\text{AMAT}_B = \text{HT}_B + \text{MR}_B \times \text{MP} \leq 12 \rightarrow 200\,\text{MR}_B \leq 6 \rightarrow \text{MR}_B \leq 0.03$

Question F7: Processes [9 pts]

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [3 pt]

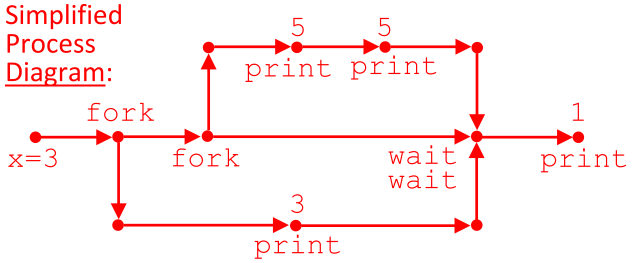
```
void concurrent(void) {
   int x = 3, status;
   if (fork()) {
      if (fork() == 0) {
         x += 2;
         printf("%d",x);
      } else {
         wait(&status);
         wait(&status);
         x -= 2;
      }
   }
   printf("%d",x);
   exit(0);
}
```

(1) _3, 5, 5, 1______

(2) _5, 3, 5, 1______

(3) _5, 5, 3, 1______

Simplified Process Diagram:



(B) For the following examples of exception causes, write "**N**" for intentional or "**U**" for unintentional from the perspective of the user process. [2 pt]

System call __N__ Hardware failure __U__

Segmentation fault __U__ Mouse clicked __U__

Syscalls are part of code you are executing. The others are external to the process.

(C) Briefly define a **zombie** process. Name a process that can *reap* a zombie process. [2 pt]

Zombie process: A process that has ended/exited but is still consuming system resources.

Reaping process: The parent process or `init/systemd` (PID 1).

(D) In the following blanks, write "**Y**" for yes or "**N**" for no if the following need to be updated when **execv** is run on a process. [2 pt]

Page table __Y__ PTBR __N__ Stack __Y__ Code __Y__

The process already has its own page table, so while we will need to invalidate PTEs from the old process image, we don't need to create another page table, so the PTBR can remain the same. We replace/update the old process image's virtual address space, including Stack and Code.

Sp17 Final Q3

3. Virtual Memory (9 points)

Assume we have a virtual memory detailed as follows:

- 256 MiB Physical Address Space
- 4 GiB Virtual Address Space
- 1 KiB page size
- A TLB with 4 sets that is 8-way associative with LRU replacement

For the following questions it is fine to leave your answers as powers of 2.

a) How many bits will be used for:

Page offset? _____**10**______

Virtual Page Number (VPN)? ____**22**_____ Physical Page Number (PPN)? ___**18**______

TLB index? _______**2**_________ TLB tag? _______**20**__________

b) How many entries in this page table?

$$2^{22}$$

c) We run the following code with an empty TLB. Calculate the TLB miss rate for data (ignore instruction fetches). Assume **i** and **sum** are stored in registers and **cool** is page-aligned.

```
#define LEAP 8
int cool[512];
... // Some code that assigns values into the array cool
... // Now flush the TLB. Start counting TLB miss rate from here.
int sum;
for (int i = 0; i < 512; i += LEAP) {
  sum += cool[i];
}
```

TLB Miss Rate: (fine to leave you answer as a fraction) ____ $\frac{1}{32}$ __________

Au16 Final Q7

Question F7: Virtual Memory [10 pts]

Our system has the following setup:

- 24-bit virtual addresses and 512 KiB of RAM with 4 KiB pages
- A 4-entry TLB that is fully associative with LRU replacement
- A page table entry contains a valid bit and protection bits for read (R), write (W), execute (X)

(A) Compute the following values: [2 pt]

Page offset width __**12**__ PPN width __**7**__
Entries in a page table __**2^{12}**__ TLBT width __**12**__

Because TLB is fully associative, TLBT width matches VPN. There are $2^{\text{VPN width}}$ entries in PT.

(B) Briefly explain why we make the page size so much larger than a cache block size. [2 pt]

Take advantage of spatial locality and try to avoid page faults as much as possible. Disk access is also super slow, so we want to pull a lot of data when we do access it.

(C) Fill in the following blanks with "**A**" for always, "**S**" for sometimes, and "**N**" for never if the following get updated during a **page fault**. [2 pt]

Page table __**A**__ Swap space __**S**__ TLB _**A/N**_ Cache __**S**__

When the page is place in physical memory, the new PPN is written into the **page table** entry.

Swap space will get updated if a dirty page is kicked out of physical memory.

For this class, we say that the page fault handler updates the **TLB** because it is more efficient.

In reality not all do (OS does not have access to hardware-only TLB; instr gets restarted).

To update a PTE (in physical mem), you check the **cache**, so it gets updated on a cache miss.

(D) The TLB is in the state shown when the following code is executed. Which iteration (value of `i`) will cause the **protection fault (segfault)**? Assume `sum` is stored in a register.
Recall: the hex representations for TLBT/PPN are padded as necessary. [4 pt]

```
long *p = 0x7F0000, sum = 0;
for (int i = 0; 1; i++) {
   if (i%2)
      *p = 0;
   else
      sum += *p;
   p++;
}
```

TLBT	PPN	Valid	R	W	X
0x7F0	0x31	1	1	1	0
0x7F2	0x15	1	1	0	0
0x004	0x1D	1	1	0	1
0x7F1	0x2D	1	1	0	0

`i` = **513**

Only the current page (VPN = TLBT = 0x7F0) has write access. Once we hit the next page (TLBT = 0x7F1), we will encounter a segfault once we try to *write* to the page. We are using pointer arithmetic to increment our pointer by 8 bytes at a time. One page holds $2^{12}/2^3 = 512$ `long`s, so we first access TLBT 0x7F1 when `i` = 512. However, the code is set up so that we only write on *odd* values of `i`, so the answer is `i` = 513.

Au16 Final Q8

Question F8: Memory Allocation [9 pts]

(A) Briefly describe one drawback and one benefit to using an *implicit* free list over an *explicit* free list. [4 pt]

Implicit drawback:	Implicit benefit:
• Slower – have to check both allocated and free blocks • Must use both boundary tags in every block – less room for payload	• Simpler code; easier to manage • Smaller minimum block size (less internal fragmentation for free blocks)

(B) The table shown to the right shows the *value of the header* for the block returned by the request: **`(int*)malloc(N*sizeof(int))`** What is the alignment size for this dynamic memory allocator? [2 pt]

N	header value
6	33
8	49
10	49
12	65

16 bytes

The alignment size is given by the difference in size once we cross an alignment boundary. Remembering to mask out the allocated tag, we see that 6 `ints` = 24 bytes gets rounded up to 32 and 8 `ints` = 32 bytes gets rounded up to 48 (remember extra space for internal fragmentation – at least the header, possibly other things).

(C) Consider the C code shown here. Assume that the `malloc` call succeeds and `foo` is stored in memory (not just in a register). Fill in the following blanks with ">" or "<" to compare the *values* returned by the following expressions just before `return 0`. [3 pt]

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";

int main(int argc, char *argv[]) {
    int *foo = malloc(8);
    free(foo);
    return 0;
}
```

`ZERO` __<__ `&ZERO`

`foo` __<__ `&foo`

`foo` __>__ `&str`

`ZERO` and `str` are global variables, so their *addresses* are in the Static Data section of memory.
`str`'s *value* is the address of a string literal, which sits at the bottom portion of Static Data.
`foo` is a local variable, so its *address* is in the Stack, but its *value* is the address of a block in the Heap.
The virtual address space is arranged such that 0 < Instructions < Static Data < Heap < Stack.

Name:____

Wi16 Final Q10

10. *C vs. Java* (**11** points) Consider this Java code (left) and somewhat similar C code (right) running on x86-64:

```
public class Foo {
  private int[] x;
  private int y;
  private int z;
  private Bar b;
  public Foo() {
     x = null;
     b = null;
  }
}
```

```
struct Foo {
  int x[6];
  int y;
  int z;
  struct Bar * b;
};

struct Foo * make_foo() {
  struct Foo * f = (struct Foo *)malloc(sizeof(struct Foo));
  f->x = NULL;
  f->b = NULL;
  return f;
}
```

(a) In Java, `new Foo()` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `Foo`'s fields? (Do *not* include space for any header information, vtable pointers, or allocator data.)

(b) In C, `malloc(sizeof(struct Foo))` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `struct Foo`'s fields? (Do *not* include space for any header information or allocator data.)

(c) The function `make_foo` attempts to be a C variant of the `Foo` constructor in Java. One line fails to compile. Which one and why?

(d) What, if anything, do we know about the values of the `y` and `z` fields after Java creates an instance of `Foo`?

(e) What, if anything, do we know about the values of the `y` and `z` fields in the object returned by `make_foo`?

Solution:

(a) 24

(b) 40

(c) `f->x = NULL` does not compile. In C, the field declaration `int x[6]` creates an inline array, not a pointer, so it does not make any sense to "assign NULL to the array" — the struct itself has slots for six array elements.

(d) We know both fields hold 0.

(e) We know nothing. (We know something abou their size, but not their contents – it could be any bit-pattern.)

Au17 Final Q9

Question F9: Memory Allocation [9 pts]

(A) In a free list, what is a **footer** used for? Be specific. Why did we not need to use one in allocated blocks in Lab 5? [2 pt]

Footer: The footer is used to get information about the previous neighboring block.
The footer is used for traversing the blocks in the heap *backwards*.
The footer is used for bidirectional coalescing.

Lab 5: In Lab 5, we used a `TAG_PRECEDING_USED` tag in block headers instead, which was sufficient because we don't coalesce with allocated blocks.

(B) We are designing a dynamic memory allocator for a **64-bit computer** with **4-byte boundary tags** and **alignment size of 4 bytes**. Assume a footer is always used. Answer the following questions: [4 pt]

Maximum tags we can fit into the header (ignoring size): __**2**__ tags

Minimum block size if we implement an *explicit* free list: __**24**__ bytes

Maximum block size (leave as expression in powers of 2): __$2^{32}-2^{2}$__ bytes

With 4-byte alignment, lowest 2 bits are guaranteed to be zeros.
Explicit free list has minimum size that includes header, two pointers, and footer. We are told boundary tags (header, footer) are 4 bytes each and pointers are 8 bytes in a 64-bit machine.
Max block size is when the size field is all 1's (with two 0's at the bottom for alignment).

(C) Consider the C code shown here. Assume that the `malloc` call succeeds and `foo` is stored in memory (not just in a register). Fill in the following blanks with ">" or "<" to compare the *values* returned by the following expressions just before `return 0`. [3 pt]

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";

int main(int argc, char *argv[]) {
    int *foo = malloc(8);
    free(foo);
    return 0;
}
```

`&foo` __**>**__ `&ZERO`

`&str` __**>**__ `ZERO`

`&main` __**<**__ `str`

`ZERO` and `str` are global variables, so their *addresses* are in the Static Data section of memory.
`str`'s *value* is the address of a string literal, which sits at the bottom portion of Static Data.
`foo` is a local variable, so its *address* is in the Stack, but its *value* is an address in the Heap.
`main` is a label in Code/Instructions.
The virtual address space is arranged such that 0 < Instructions < Static Data < Heap < Stack.