University of Washington – Computer Science & Engineering

Autumn 2017 Instructor: Justin Hsia 2017-12-13

CSE351 FINAL

Last Name:	**Perfect**	
First Name:	**Perry**	
Student ID Number:	1234567	
Name of person to your Left | Right	Samantha Student	Larry Learner
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. **(please sign)**		

Do not turn the page until 12:30.

Instructions

- This exam contains 14 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, or calculators). You are allowed two pages (US letter, double-sided) of *handwritten* notes.
- 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.

Question	**M1**	**M2**	**M3**	**M4**	**M5**	**F6**	**F7**	**F8**	**F9**	**F10**	**Total**
Possible Points	8	2	8	10	8	10	9	10	9	5	**79**

Question M1: Number Representation [8 pts]

(A) Take the 32-bit numeral **`0xC0800000`**. Circle the number representation below that has the *negative* value for this numeral. [2 pt]

Floating Point [✓ Sign & Magnitude] Two's Complement Unsigned

Unsigned: Can only represent positive numbers.

Floating Point: S = 1 and E = 10000001_2 → Exp = 2, so a small negative number.

Sign & Mag: Negative number with magnitude 100 0000 $10...0_2$.

Two's: Negative number with magnitude 011 1111 $10...0_2$ (flip bits + 1).

(B) Let `float f` hold the value $\mathbf{2^{20}}$. What is the *largest power of 2* that gets rounded off when added to `f`? Answer in exponential form, not just the exponent. [2 pt]

23 bits in M, so need 24th power less than 2^{20} to get rounded off.

$\boxed{\mathbf{2^{-4}}}$

Traffic lights display three basic colors: red (R), yellow (Y), and green (G), so we can use them to encode base 3! We decide to use the encoding 0↔R, 1↔Y, 2↔G. For example, $5 = 1\times3^1+2\times3^0$ would be encoded as **YG**. Assume each traffic light can only display one color at a time.

(C) What is the *unsigned* decimal value of the traffic lights displaying **RGYY**? [2 pt]

$0\times3^3+2\times3^2+1\times3^1+1\times3^0 = 18+3+1 = 22.$

$\boxed{\mathbf{22}}$

(D) If we have **9 bits** of binary data that we want to store, how many *traffic lights* would it take to store that same data? [2 pt]

9 bits represents 512 things. Powers of 3: 1, 3, 9, 27, 81, 243, 729.

6 traffic lights

Question M2: Design Question [2 pts]

(A) The machine code for x86-64 instructions are variable length. Name one advantage and one disadvantage of this design decision. [2 pt]

Advantage:	Machine code/Code section of memory is more compact (don't need to pad). No limit on number of instructions in ISA.
Disadvantage:	Harder to tell/find where to read next instruction. Need more complex hardware to fetch and/or decode instructions.

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.

Question M4: Procedures & The Stack [10 pts]

The function `count_sp` counts the number of *spaces* in a `char` array (this is the recursive version of the `mystery` function from the Midterm). The function and its *disassembly* are shown below:

```
int count_sp(char* str) {
     if (*str)
          return (*str == ' ') + count_sp(str+1);
     return 0;
}
```

```
0000000000400536 <count_sp>:
  400536:  0f b6 07        movzbl (%rdi),%eax
  400539:  84 c0           testb  %al,%al
  40053b:  74 16           je     400553 <count_sp+0x1d>
  40053d:  53              pushq  %rbx
  40053e:  3c 20           cmpb   $0x20,%al
  400540:  0f 94 c3        sete   %bl
  400543:  0f b6 db        movzbl %bl,%ebx
  400546:  48 83 c7 01     addq   $0x1,%rdi
  40054a:  e8 e7 ff ff ff  callq  400536 <count_sp>
  40054f:  01 d8           addl   %ebx,%eax
  400551:  eb 06           jmp    400559 <count_sp+0x23>
  400553:  b8 00 00 00 00  movl   $0x0,%eax
  400558:  c3              retq
  400559:  5b              popq   %rbx
  40055a:  c3              retq
```

(A) The information found in the *right-most* column/portion of the disassembly is first generated as the output of which of the following? Circle one. [1 pt]

[✓ Compiler] Assembler Linker Loader

(B) The *left-most* column of the disassembly was generated by which of the following? [1 pt]

Compiler Assembler [✓ Linker] Loader

(C) Why is `%rbx` being pushed onto the stack? What is `%rbx` being used for in this function? [2 pt]

Why push: Because (1) `%rbx` is a callee-saved register and (2) `count_sp` chooses to use/change this register.

Usage: `%rbx` is being used to store the value of `*str == ' '` (is this char a space?).

(D) What is the return address to `count_sp` that gets stored on the stack? Answer in hex. [1 pt]

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

0x **40054f**

(E) Provide a call to `count_sp` that is *guaranteed* to cause a **segmentation fault**. [1 pt]

`count_sp( ____NULL____ );`

Zero also accepted, with and without casting.

(F) We call `count_sp(" ! ")`. Fill in the incomplete snapshot of the stack below (in hex) once this call to `count_sp` returns to `main`. For unknown words, write "garbage". [4 pt]

0x7fffffffdb68	<ret addr to main>
0x7fffffffdb60	<original rbx>
0x7fffffffdb58	**0x40054f**
0x7fffffffdb50	**0x1**
0x7fffffffdb48	**0x40054f**
0x7fffffffdb40	**0x0**
0x7fffffffdb38	**0x40054f**
0x7fffffffdb30	**garbage**
0x7fffffffdb28	**garbage**
0x7fffffffdb20	**garbage**

4 total stack frames of `count_sp` created: `count_sp(" ! ")` → `count_sp("! ")` → `count_sp(" ")` → `count_sp("")`. Stack data in these frames alternates between return addresses (full credit given for your answer to part D) and pushed `%rbx` values (credit given for your answer to part C). The last stack frame hits the base condition and doesn't push `%rbx` onto the stack. The data below this point is considered garbage.

Question M5: C & Assembly [8 pts]

Answer the questions below about the following x86-64 assembly function, which *uses a struct*:

```
mystery:
.L3:    testq   %rdi, %rdi     # Line 1
        je      .L4            # Line 2
        cmpw    %si, 0(%rdi)   # Line 3
        je      .L5            # Line 4
        movq    8(%rdi), %rdi  # Line 5
        jmp     .L3            # Line 6
.L4:    movl    $0, %eax       # Line 7
        retq                   # Line 8
.L5:    movl    $1, %eax       # Line 9
        retq                   # Line 10
```

(A) What **C variable type** would `%rsi` be in the corresponding C program? [1 pt]

In Line 3, `%si` is used in a cmp**w** instruction.

___**short**__ rsi

(B) `%rdi` is a pointer to a struct that contains 2 fields. What is the width of the second field? [1 pt]

Used in a **`movq`** instruction in Line 5. Also at offset of 8 bytes, matching its alignment requirement.

__**8**__ bytes

(C) Based on Line 5, give an intuitive name for the second field in the struct. [1 pt]

Other variants accepted.

next or **ptr**

(D) Convert lines 1, 2, 7, and 8 into C code. Use variable names that correspond to the register names (e.g. `al` for the value in `%al`). [3 pt]

`if ( _rdi == NULL_ ) ___return 0___;`

(E) Describe at a high level what you think this function *accomplishes* (not line-by-line). [2 pt]

Returns 1 if linked list (singly-linked, linear) contains a specified value (in `%si`).

Question F6: Caching [10 pts]

We have 64 KiB of RAM and a 2-KiB L1 data cache that is 4-way set associative with 32-byte blocks and random replacement, write-back, and write allocate policies.

(A) Calculate the TIO address breakdown: [1.5 pt]

Tag bits	Index bits	Offset bits
7	4	5

16 address bits. $\log_2 32 = 5$ offset bits. 2-KiB cache = 64 blocks. 4 lines/set → 16 sets.

(B) How many management bits (bits *other* than the block data) are there in every line in the cache? [1 pt]

Tag bits + Valid bit + Dirty bit (write-back) __**9**__ bits

(C) The code snippet below accesses an array of `double`s. Assume `i` is stored in a register. Calculate the **Miss Rate** if the cache starts *cold*. [2.5 pt]

```
#define ARRAY_SIZE 256
double data[ARRAY_SIZE];        // &data = 0x1000 (physical addr)
for (i = 0; i < ARRAY_SIZE; i += 1)
    data[i] /= 100;
```

1/8 = 12.5%

Access pattern is read then write to `data[i]`. Stride = 1 double = 8 bytes. 32/8 = 4 strides per block. The offset of `&data` is 0b00000, so we start at the beginning of a cache block. First access (read) is a compulsory miss and the next 7 (over 4 different addresses) are hits. Since we never revisit indices, this pattern continues for all cache blocks.

(D) For each of the proposed (independent) changes, write **IN** for "increased", **NC** for "no change", or **DE** for "decreased" to indicate the effect on the **Miss Rate** for the code above: [4 pt]

Use `float` instead _**DE**_ Half the cache size _**NC**_

Split the loop body into: _**DE**__ No-write allocate _**NC**_

```
data[i] /= 10;
data[i] /= 10;
```

Using `float`s means more strides/block. We never revisit blocks, so cache size doesn't matter. Since the entire array fits in the cache, running it through a 2^{nd} loop results in all hits. No-write allocate has no effect because all of our misses are on *reads*.

(E) Assume it takes 100 ns to get a block of data from main memory. If our L1 data cache has a hit time of 2 ns and a miss rate of 3%, what is the average memory access time (AMAT)? [1 pt]

AMAT = HT + MR × MP = 2 + 0.03 × 100 = 5 __**5**__ ns

Question F7: Processes [9 pts]

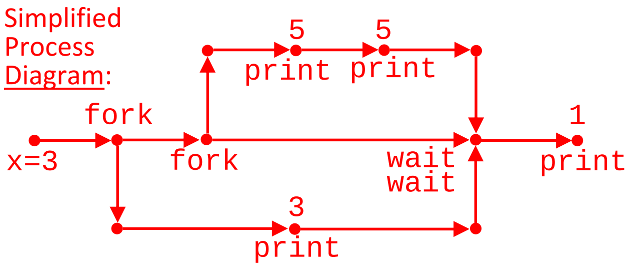
(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [3 pt]

```
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**______



(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.

Question F8: Virtual Memory [10 pts]

Our system has the following setup:

- 20-bit virtual addresses and 64 KiB of RAM with 256-B pages
- A 4-entry TLB that is fully associative with LRU replacement
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)

(A) Compute the following values: [4 pt]

Page offset width **_8 bits_**   # of physical pages **_2^8=256**

# of virtual pages **2^{12}=4096**   TLBI width **_0 bits_**

Page offset is $\log_2 256 = 8$ bits wide. VA space is 2^{20} bytes, so $2^{20}/2^8$ virtual pages and $2^{16}/2^8$ physical pages. TLB is fully associative, so just one set and $\log_2 1 = 0$ TLBI bits.

(B) Briefly explain why we make physical memory **write-back** and **fully-associative**. [2 pt]

Write-back: Avoid writing to disk as much as possible, or only when we absolutely need to.

Fully-associative: Don't waste space in RAM; we want to use it fully utilize the limited space we have.

(C) The TLB is in the state shown when the following code is executed. The code eventually causes a **protection fault**. What are the values of the variables when the fault occurs? [4 pt]

```
long *p = 0x7F080;
for (int i = 0; 1; i++) {
   *p += 1;
    p += 4;
}
```

TLBT	PPN	Valid	R	W	X
0x7F0	0xC3	1	1	1	0
0x7F2	0x3D	1	1	0	0
0x004	0xF4	1	1	0	1
0x7F1	0x42	1	1	1	0

p = ___**0x7F200**___

i = __**12**__

The loop reads and then writes to the address in pointer `p` and then strides by 4 `long`s = 32 bytes = 0x20 addresses. `p` starts at address `0x7F080`, which is at page offset `0x80`, the middle of the page. We see in the TLB that this page (TLBT `0x7F0`) is valid with read and write privileges, as is the following page (TLBT `0x7F1`). However, the following page lacks write privileges, so our first *write* to that page will cause a protection fault. This occurs at the very first byte of the page: `0x7F200`, which is reached when `i = 12`.

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.

Question F10: C and Java [5 pts]

For this question, use the following Java object definition and C struct definition. Assume addresses are all 64-bits.

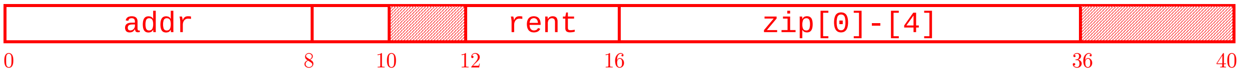
```
public class RentalJ {
   String addr;
   short  rooms;
   float  rent;
   int[]  zip;

   public void info() {
      System.out.println("Rental at "+addr);
   }
}
public class Apt extends RentalJ {
   int roommates;
   public int occupants() {
      return roommates+1;
   }
}
```

```
struct RentalC {  K:
   char* addr;    8
   short rooms;   2
   float rent;    4
   int   zip[5];  4
};          Kmax = 8
```

(A) How much memory, in bytes, does an instance of `struct RentalC` use? How many of those bytes are *internal* fragmentation and *external* fragmentation? [3 pt]

sizeof(struct RentalC)	Internal	External
40 bytes	2 bytes	4 bytes

Alignment requirements listed above in red, next to the struct fields. A `struct RentalC` instance will look as shown below:



The 2 bytes between `rooms` and `rent` count as internal fragmentation.
The 4 bytes at the end count as external fragmentation.

(B) How much *longer*, in bytes, are the following for `Apt` than for `RentalJ`? Assume the Java instance fields are aligned to 4 bytes. [2 pt]

Instance:	4 bytes
Virtual method table (vtable):	8 bytes

`Apt` extends `RentalJ` by adding a field and a method, so the length of that field (4 bytes for an `int`) is added to the object instance length (no padding needed due to 4-byte alignment) and the length of a reference is added to the vtable.

This page purposely left blank