University of Washington – Computer Science & Engineering

Autumn 2018 Instructor: Justin Hsia 2018-12-12

CSE351 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. (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	16	4	16	22	16	10	20	18	18	18	**158**

Question M1: Numbers [16 pts]

(A) *Briefly* explain why we know that there may be data loss **casting** from `int` to `float`, but there won't be casting from `int` to `double`. [4 pt]

Explanation:

(B) What value will be read after we try to store $\mathbf{-2^{127} - 2^{104}}$ in a `float`? (Circle one) [4 pt]

-2^{127} –NaN $-\infty$ $-2^{127}-2^{104}$

(C) Complete the following C function that returns whether or not a pointer `p` is aligned for data type size `K`. Hint: be careful with data types! [4 pt]

```
int aligned(void* p, int K) {
    return ________________________________;
}
```

(D) Take the 32-bit numeral **`0x50000000`**. Circle the number representation below that has the *most positive* value for this numeral. [4 pt]

Floating Point Two's Complement Unsigned Two's AND Unsigned

Question M2: Design Question [4 pts]

(A) If the Stack grew upwards (*e.g.* we switched the positions of the Stack and Heap), which assembly instructions would need their behaviors changed? Name two and briefly describe the changes.

Instruction 1: Change:
Instruction 2: Change:

SID: ______________

Question M3: Pointers & Memory [16 pts]

We are using a 64-bit x86-64 machine (**little endian**). Below is the `sum_r` function disassembly, *showing where the code is stored in memory.* Hint: read the questions before the assembly!

```
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) What are the values (in hex) stored in each register shown after the following x86 instructions are executed? Use the appropriate bit widths. [8 pt]

```
leal 9(%rdi),      %eax
andb (%rdi,%rsi), %sil
```

Register	Value (hex)
%rdi	0x 0000 0000 0040 0507
%rsi	0x 0000 0000 0000 0003
%eax	0x
%sil	0x

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let **short* shortP = 0x400513**. [8 pt]

```
short v1 = shortP[______];                          // set v1 = 0x00B8

void* v2 = (void*)((____________*)shortP + 5);   // set v2 = 0x400527
```

Question M4: Procedures & The Stack [22 pts]

The function `sum_r2` calculates the sum of the elements in a `long` array (similar to `sum_r` from the Midterm, but with extra, *useless* parameters). The function and its disassembly are shown below:

```
int sum_r2(long* p, long len, long a, long b, long c, long d) {
    if (len > 0)
        return *p + sum_r2(p+1, len-1, a, b, c, d);
    return 0;
}
```

```
0000000000400507 <sum_r2>:
  400507:    48 85 f6          test   %rsi,%rsi
  40050a:    7f 06             jg     400512 <sum_r2+0xb>
  40050c:    b8 00 00 00 00    mov    $0x0,%eax
  400511:    c3                retq
  400512:    57                push   %rdi
  400513:    48 83 ee 01       sub    $0x1,%rsi
  400517:    48 83 c7 08       add    $0x8,%rdi
  40051b:    e8 e7 ff ff ff    callq  400507 <sum_r2>
  400520:    5f                pop    %rdi
  400521:    03 07             add    (%rdi),%eax
  400523:    c3                retq
```

(A) **Machine code** is *first* generated by which of the following? [2 pt]

Compiler   Assembler   Linker   Loader

(B) Which of the following changes the value of the **program counter**? [2 pt]

Compiler   Assembler   Linker   Loader

(C) Let **long** `ar[] = {1,2,3,4}`. When **`sum_r2(ar,3,0,0,0,0)`** is called, fill in the following quantities: [6 pt]

Size of each *recursive* stack frame:	bytes
Number of `sum_r2` stack frames created:	
Value returned:	

(D) We now generate the function **`sum_r3`** by adding another useless parameter **`long e`** to the *end* of the parameter list and changing the recursive call to **`sum_r3(p+1,len-1,a,b,c,d,e)`**. *Briefly* describe how a stack frame of `sum_r3` will differ from a stack frame of `sum_r2` – what gets added or removed and where in the stack frame? [4 pt]

Description:

(E) Assume `main` calls **`sum_r3(ar,2,0,0,0,0,0xf)`** – the NEW function with the extra parameter – with **`long ar[] = {4, 2}`** and **`&ar = 0x7f`**…**`db00`**. Fill in the snapshot of the stack below (in hex) as this call returns to `main`. Although the assembly code will change, use the addresses shown in `sum_r2` where applicable. For unknown words, write "`0x unknown`". [8 pt]

0x7fffffffdae8	<ret addr to main>
0x7fffffffdae0	0x
0x7fffffffdad8	0x
0x7fffffffdad0	0x
0x7fffffffdac8	0x
0x7fffffffdac0	0x
0x7fffffffdab8	0x
0x7fffffffdab0	0x
0x7fffffffdaa8	0x

Question M5: C & Assembly [16 pts]

Answer the questions below about the following x86-64 assembly function, which *uses a struct* with two fields named **one** and **two**, declared in that order:

```
mystery:
        movl    $0, %eax        # Line 1
        jmp     .L3             # Line 2
.L2:    movq    8(%rdi), %rdi   # Line 3
        testq   %rdi, %rdi      # Line 4
        je      .L4             # Line 5
.L3:    movl    (%rdi), %edx    # Line 6
        cmpl    %eax, %edx      # Line 7
        jle     .L2             # Line 8
        movl    %edx, %eax      # Line 9
        jmp     .L2             # Line 10
.L4:    retq                    # Line 11
```

(A) `%rdi` contains a pointer to an instance of the struct. What **variable type** is field `one`? [2 pt]

____

(B) How many **parameters** does this function have? [1 pt]

______

(C) Give a **value of the first argument** to `mystery` that will cause a *segmentation fault*. [3 pt]

$\boxed{}$ causes a segfault on Line ______

(D) Convert lines 6, 7, 8, and 9 into C code. Use variable names that correspond to the register names (e.g. `al` for the value in `%al`). Remember that the struct fields are named `one` and `two`. [6 pt]

```
if ( ________________ ) { ________________; }
```

(E) Describe at a high level what you think this function *accomplishes* (not line-by-line). [4 pt]

SID: ______________

Question F6: Structs [10 pts]

For this question, assume a 64-bit machine and the following C struct definition.

```
typedef struct {
  short age;        // age, in years
  char  name[16];   // name, as a C string
  int   height;     // height, in cm
  long  weight;     // weight, in kg
} person;
```

(A) How much memory, in bytes, does an instance of `person` use? How many of those bytes are *internal* fragmentation and *external* fragmentation? [6 pt]

`sizeof(person)`	Internal	External

(B) Instead of storing name as a 16-character array, we could use a character pointer instead. *Briefly* describe an advantage and disadvantage to this new implementation. [4 pt]

Advantage:
Disadvantage:

Question F7: Caching [20 pts]

We have 1 MiB of RAM and a 256-byte L1 data cache that is direct-mapped with 16-byte blocks and random replacement, write-back, and write allocate policies.

(A) Calculate the TIO address breakdown: [3 pt]

Tag bits	Index bits	Offset bits

(B) The code snippet below accesses an array of `floats`. Assuming `i` and `temp` are stored in registers, the **Miss Rate is 100%** if the cache starts *cold*. What is the memory access pattern (read or write to which elements/addresses) and why do we see this miss rate? [6 pt]

```
#define N 128
float data[N];     // &data = 0x08000 (physical addr)
for (i = 0; i < N/2; i += 1) {
    temp = data[i];
    data[i] = data[i+N/2];
    data[i+N/2] = temp;
}
```

Per Iteration:	Access 1:	Access 2:	Access 3:	Access 4:
(circle) →	R / W to	R / W to	R / W to	R / W to
(fill in) →	data[______]	data[______]	data[______]	data[______]
Miss Rate:				

(C) For each of the proposed (independent) changes, draw ↑ for "increased", — for "no change", or ↓ for "decreased" to indicate the effect on the **miss rate from Part B** for the code above: [8 pt]

Use `double` instead	______	Double the cache size	______
Increase the associativity	______	No-write allocate	______

(D) Assume it takes 120 ns to get a block of data from main memory. If our L1 data cache has a hit time of 3 ns and a miss rate of 5%, circle which of the following improvements would result in the best average memory access time (AMAT). [3 pt]

2-ns hit time   4% miss rate   100-ns miss penalty

Question F8: Processes [18 pts]

(A) The following function prints out four numbers. In the following blanks, list three possible outcomes: [6 pt]

```
void concurrent(void) {
   int x = 2, status;
   if (fork()) {
      x *= 2;
   } else {
      x -= 1;
   }
   if (fork()) {
      x += 1;
      wait(&status);
   } else {
      x -= 2;
   }
   printf("%d",x);
   exit(0);
}
```

(1) ________________

(2) ________________

(3) ________________

(B) For each of the following types of synchronous exceptions, indicate whether they are intentional (`I`) or unintentional (`U`) AND whether they are recoverable (`R`), maybe recoverable (`M`) or not recoverable (`N`). [6 pt]

	I/U	**R/M/N**
Trap	______	______
Fault	______	______
Abort	______	______

(C) For the following scenarios, circle the outcome when the child process executes **`exit(0)`**. [6 pt]

Scenario:		**Outcome for child:**	
Parent is still executing.	Alive	Reaped	Zombie
Parent has called `wait()`.	Alive	Reaped	Zombie
Parent has terminated.	Alive	Reaped	Zombie

Question F9: Virtual Memory [18 pts]

Our system has the following setup:

- 18-bit virtual addresses and 32 KiB of RAM with 1-KiB pages
- A 4-entry direct-mapped TLB with LRU replacement
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)

(A) Compute the following values: [8 pt]

PPO width ________ # of bits per PTE ________

VPN width ________ # of TLB sets ________

(B) In a machine that uses virtual memory, what is the relative usage of the following that you would expect? *Briefly* defend your choice. [4 pt]

Accessed more: (circle) TLB Page Table

Explanation:

(C) Assume the TLB is in the state shown (permission bits: 1 = allowed, 0 = disallowed) when the following loop is executed and that p is stored in a register. Which "event" occurs first and on which value of p? [6 pt]

```
short *p = 0x1F400;
while (1) {
   *p = 0;
    p += 8;
}
```

TLBT	PPN	Valid	R	W	X
0x04	0x1D	1	1	0	1
0x1F	0x0C	1	1	1	0
0x1F	0x03	1	1	1	0
0x06	0x14	1	1	1	0

Circle one: Page Fault Protection Fault TLB Replacement

p = 0x__________

SID: ______________

Question F10: Memory Allocation [18 pts]

(A) In the following code, briefly identify the TWO memory errors. They can be fixed by changing ONE line of code. [6 pt]

```
int N = 64;
double *func(double A[][], double x[]) {
   double *z = (double *) malloc(N * sizeof(float));
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         z[i] = A[i][j] + z[i] * x[j];
      }
   }
   return z;
}
```

Error 1:
Error 2:
Line of code with fixes:

(B) We are using a dynamic memory allocator on a **64-bit machine** with an **explicit free list**, **4-byte boundary tags**, and **16-byte alignment**. Assume that a footer is always used. [6 pt]

Request	return value	block addr	block size	internal fragmentation in this block
`p = malloc(12);`	`0x610`	`0x`_______	_______ bytes	_______ bytes

(C) Consider the C code shown here. Assume that the `malloc` call succeeds and that all variables are stored in memory (not registers). In the following groups of expressions, **circle the one** whose returned *value* (assume just before `return 0`) is the **lowest/smallest**. [6 pt]

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

int main(int argc, char *argv[]) {
    int *foo = malloc(888);
    int bar = 351;
    free(foo);
    return 0;
}
```

Group 1:	`&bar`	`&foo`	`foo`
Group 2:	`&foo`	`main`	`str`
Group 3:	`bar`	`&str`	`&ZERO`

CSE 351 Reference Sheet (Final)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

Sizes

C type	Suffix	Size
`char`	`b`	`1`
`short`	`w`	`2`
`int`	`l`	`4`
`long`	`q`	`8`

IEEE 754 FLOATING-POINT STANDARD

Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

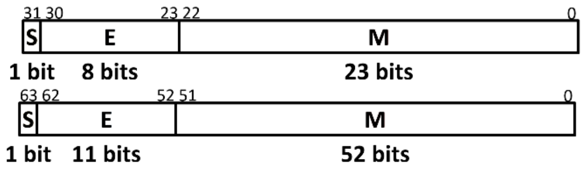
Bit fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

E	M	Meaning
all zeros	all zeros	± 0
all zeros	non-zero	± denorm num
1 to MAX-1	anything	± norm num
all ones	all zeros	± ∞
all ones	non-zero	NaN

IEEE Single Precision and Double Precision Formats:



Assembly Instructions

`mov a, b`	Copy from `a` to `b`.
`movs a, b`	Copy from `a` to `b` with sign extension. Needs *two* width specifiers.
`movz a, b`	Copy from `a` to `b` with zero extension. Needs *two* width specifiers.
`lea a, b`	Compute address and store in `b`. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
`push src`	Push `src` onto the stack and decrement stack pointer.
`pop dst`	Pop from the stack into `dst` and increment stack pointer.
`call <func>`	Push return address onto stack and jump to a procedure.
`ret`	Pop return address and jump there.
`add a, b`	Add from `a` to `b` and store in `b` (and sets flags).
`sub a, b`	Subtract `a` from `b` (compute `b-a`) and store in `b` (and sets flags).
`imul a, b`	Multiply `a` and `b` and store in `b` (and sets flags).
`and a, b`	Bitwise AND of `a` and `b`, store in `b` (and sets flags).
`sar a, b`	Shift value of `b` *right* (*arithmetic*) by `a` bits, store in `b` (and sets flags).
`shr a, b`	Shift value of `b` *right* (*logical*) by `a` bits, store in `b` (and sets flags).
`shl a, b`	Shift value of `b` *left* by `a` bits, store in `b` (and sets flags).
`cmp a, b`	Compare `b` with `a` (compute `b-a` and set condition codes based on result).
`test a, b`	Bitwise AND of `a` and `b` and set condition codes based on result.
`jmp <label>`	Unconditional jump to address.
`j* <label>`	Conditional jump based on condition codes (*more on next page*).
`set* a`	Set byte based on condition codes.

Conditionals

Instruction		`(op) s, d`	`test a, b`	`cmp a, b`
`je`	"Equal"	`d (op) s == 0`	`b & a == 0`	`b == a`
`jne`	"Not equal"	`d (op) s != 0`	`b & a != 0`	`b != a`
`js`	"Sign" (negative)	`d (op) s < 0`	`b & a < 0`	`b-a < 0`
`jns`	(non-negative)	`d (op) s >= 0`	`b & a >= 0`	`b-a >= 0`
`jg`	"Greater"	`d (op) s > 0`	`b & a > 0`	`b > a`
`jge`	"Greater or equal"	`d (op) s >= 0`	`b & a >= 0`	`b >= a`
`jl`	"Less"	`d (op) s < 0`	`b & a < 0`	`b < a`
`jle`	"Less or equal"	`d (op) s <= 0`	`b & a <= 0`	`b <= a`
`ja`	"Above" (unsigned >)	`d (op) s > 0U`	`b & a > 0U`	`b-a > 0U`
`jb`	"Below" (unsigned <)	`d (op) s < 0U`	`b & a < 0U`	`b-a < 0U`

Registers

		Name of "virtual" register		
Name	**Convention**	**Lowest 4 bytes**	**Lowest 2 bytes**	**Lowest byte**
`%rax`	Return value – Caller saved	`%eax`	`%ax`	`%al`
`%rbx`	Callee saved	`%ebx`	`%bx`	`%bl`
`%rcx`	Argument #4 – Caller saved	`%ecx`	`%cx`	`%cl`
`%rdx`	Argument #3 – Caller saved	`%edx`	`%dx`	`%dl`
`%rsi`	Argument #2 – Caller saved	`%esi`	`%si`	`%sil`
`%rdi`	Argument #1 – Caller saved	`%edi`	`%di`	`%dil`
`%rsp`	Stack Pointer	`%esp`	`%sp`	`%spl`
`%rbp`	Callee saved	`%ebp`	`%bp`	`%bpl`
`%r8`	Argument #5 – Caller saved	`%r8d`	`%r8w`	`%r8b`
`%r9`	Argument #6 – Caller saved	`%r9d`	`%r9w`	`%r9b`
`%r10`	Caller saved	`%r10d`	`%r10w`	`%r10b`
`%r11`	Caller saved	`%r11d`	`%r11w`	`%r11b`
`%r12`	Callee saved	`%r12d`	`%r12w`	`%r12b`
`%r13`	Callee saved	`%r13d`	`%r13w`	`%r13b`
`%r14`	Callee saved	`%r14d`	`%r14w`	`%r14b`
`%r15`	Callee saved	`%r15d`	`%r15w`	`%r15b`

C Functions

`void* malloc(size_t size):`
Allocate size bytes from the heap.

`void* calloc(size_t n, size_t size):`
Allocate n*size bytes and initialize to 0.

`void free(void* ptr):`
Free the memory space pointed to by ptr.

`size_t sizeof(type):`
Returns the size of a given type (in bytes).

`char* gets(char* s):`
Reads a line from stdin into the buffer.

`pid_t fork():`
Create a new child process (duplicates parent).

`pid_t wait(int* status):`
Blocks calling process until any child process exits.

`int execv(char* path, char* argv[]):`
Replace current process image with new image.

Virtual Memory Acronyms

MMU	Memory Management Unit	**VPO**	Virtual Page Offset	**TLBT**	TLB Tag
VA	Virtual Address	**PPO**	Physical Page Offset	**TLBI**	TLB Index
PA	Physical Address	**PT**	Page Table	**CT**	Cache Tag
VPN	Virtual Page Number	**PTE**	Page Table Entry	**CI**	Cache Index
PPN	Physical Page Number	**PTBR**	Page Table Base Register	**CO**	Cache Offset