University of Washington – Computer Science & Engineering

Autumn 2018 Instructor: Justin Hsia 2018-12-12

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	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:
> An `int` contains 32 bits of information, which always fits into the 52-bit mantissa of a `double`, but not always into the 23-bit mantissa of a `float`.

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

-2^{127} –NaN $-\infty$ **(circled)** $-2^{127}-2^{104}$

$-2^{127} - 2^{104} = -2^{127} \times 1.00000000000000000000001_2$.

Exp = 127 is a representable normalized exponent (E = 0b11111110)

All of the bits following the binary point in the Mantissa fit into the M field (23 bits).

(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 !((long)p%K);   // other variants accepted, e.g.
}                          // (long)p%K == 0
                           // (long)p == (long)p/K*K
                           // !((long)p & (K-1))
```

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

(circled) Floating Point   Two's Complement   Unsigned   Two's AND Unsigned

`float`: S = 0, E = 0b1010 0000, M = 0, so $+1.0_2 \times 2^{33}$. You can recognize that this is larger than TMax and UMax.

`int/unsigned int`: Positive encodings are the same for both representations. Value is $5 \times 16^7 = 2^{30}+2^{28}$.

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.

> There are 4 instructions that would need to be changed: `push`, `pop`, `call`, `ret`.
>
> `push` and `call` would now need to *increase* `%rsp`.
>
> `pop` and `ret` would now need to *decrease* `%rsp`.

SID: __1234567___

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 **0040 0510**
%sil	0x **02**

`leal` instruction calculates the address `0x400507 + 9 = 0x400510`.

`andb` instruction pulls the byte at memory address `0x400507+3 = 0x40050a`, which is `0xf6`.
and-ing this with the lowest byte of `%rsi` yields `0xf6 & 0x03 = 0x02`.

(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[__-3__];                     // set v1 = 0x00B8
void* v2 = (void*)((_int/float_*)shortP + 5);  // set v2 = 0x400527
```

`0xb8` byte in `sum_r` is at address `0x40050d`, 6 bytes = 3 `shorts` behind `shortP`.

The difference between `v2` and `shortP` is 20 bytes. Since by pointer arithmetic we are moving 5 "things" away, `shortP` must be cast to a pointer to a data type of size 4 bytes.

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)**

The loader actually starts the process running, so it's the only one that affects registers.

(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:	**16** bytes
Number of `sum_r2` stack frames created:	**4**
Value returned:	**6**

In the recursive case, the return address starts the stack frame and then we also push `%rdi`, so that takes up 16 bytes.

`sum_r2` is called with `len` = 3, so there are stack frames for `len` = 3, 2, 1, 0 (base case).

This call to `sum_r2` sums the first 3 elements, so 1+2+3 = 6.

SID: __**1234567**___

(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:
The 7th argument (e) spills onto the Stack in the argument build portion at the "end" (lowest addresses) of the stack frame during a recursive call. The value of e will be found directly above the return address that starts the next frame.

(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]

Address	Value	Frame
0x7fffffffdae8	<ret addr to main>	
0x7fffffffdae0	0x **7f…db00**	sum_r3(ar,2,…,0xf)
0x7fffffffdad8	0x **f**	
0x7fffffffdad0	0x **400520**	
0x7fffffffdac8	0x **7f…db08**	sum_r3(ar,1,…,0xf)
0x7fffffffdac0	0x **f**	
0x7fffffffdab8	0x **400520**	sum_r3(ar,0,…,0xf)
0x7fffffffdab0	0x **unknown**	
0x7fffffffdaa8	0x **unknown**	

3 total stack frames of sum_r3 created: sum_r3(ar,2,…) → sum_r3(ar+1,1,…) → sum_r3(ar+2,0,…). In the recursive case, first %rdi (holding the address of the current ar element) is pushed onto the stack, and *then* the 7th argument is pushed onto the stack (the argument build comes right above the return address). The last stack frame hits the base condition and doesn't push %rdi or the 7th argument onto the stack. The data below this point is considered unknown/garbage.

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]

In Line 6, `0(%rdi)` is used in a `movl` instruction. ____**int**____

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

`%rsi` is not used and `%edx`/`%rdx` used as a temporary variable. ____**1**____

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

Any low address (below Code) accepted. [0/null] causes a segfault on Line __**6**__

(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 ( _rdi->one > eax_ ) { _eax = rdi->one_; }
```

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

Returns the maximum positive value in a singly-linked list.

Question F6: Structs [10 pts]

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

```
typedef struct {   K:
  short age;       2   // age, in years
  char  name[16];  1   // name, as a C string
  int   height;    4   // height, in cm
  long  weight;    8   // weight, in kg
} person;   Kmax = 8
```

(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**
32 bytes	**2 bytes**	**0 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 `name` and `height` count as internal fragmentation.
There is no external fragmentation.

(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]

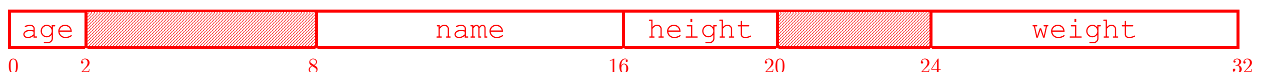
Advantage:

- More flexibility: don't have to use 16 bytes for every name (wasted space for short names) and no upper limit of 16 bytes (really 18 bytes).
- Can "reuse" the same name in memory for different people.

Disadvantage:

- Harder to manage: need to malloc separate space for the name.
- Extra overhead of 8 bytes for pointer that isn't used to store the name.
- Extra memory access to read name (have to read pointer value first).

Note that you don't actually save space in a struct instance due to alignment issues:



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
12	**4**	**4**

20 address bits. $\log_2 16 = 4$ offset bits. 256-B cache = 16 blocks. 1 line/set → 16 sets.

(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[___**i**___]	data[_**i+N/2**_]	data[___**i**___]	data[_**i+N/2**_]

Miss Rate: The distance between `data[i]` and `data[i+N/2]` is (128/2)*4 = 256 bytes, which is the size of our data cache. Therefore `data[i]` and `data[i+N/2]` map to the same set (*e.g.* `&data[0]=0x8000`, `&data[64]=0x8100`). The alternating nature of our access pattern causes continuous conflict misses because it is direct-mapped.

(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 __↓__

Using `doubles` doubles our jump between `data[i]` and `data[i+N/2]`, but they still conflict. Doubling cache size means they map into different sets while increasing the associativity allows both blocks to coexist in the same set, so both remove the conflict misses. No-write allocate means we skip the cache on access 3, so that access 4 in each loop iteration then becomes a Hit.

(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] AMAT = HT + MR×MP

2-ns hit time	(4% miss rate)	100-ns miss penalty
8 ns	7.8 ns	8 ns

SID: __**1234567**___

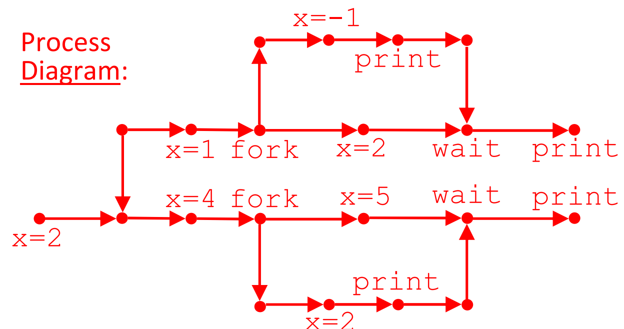
Question F8: Processes [18 pts]

(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);
}
```

The 5 possible outcomes:

```
1)  2, -1,  2,  5
2) -1,  2,  2,  5
3) -1,  2,  5,  2
4)  2, -1,  5,  2
5)  2,  5, -1,  2
```



(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] By definition; see lecture slides.

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

(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

`wait()` will cause the parent to reap the child, while `init/systemd` will reap the child if the parent has terminated.

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 **10 bits** # of bits per PTE **10 bits**

VPN width **8 bits** # of TLB sets **4 sets**

Page offset is $\log_2 1024 = 10$ bits wide. VPN is n-p = 8 bits wide. PTE holds PPN + management bits = 5+5 = 10 bits. TLB is direct-mapped, so one set per entry.

(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** (circled) Page Table

Explanation: TLB is accessed on *every* memory access. The Page Table is *only* accessed on a TLB miss (≤ 100% of the time).

(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** (circled) Protection Fault **TLB Replacement** (circled) ← Sorry! This part of the question was poorly posed. ☹

p = 0x__**1FC00**__

Our offset field is 10 bits wide and TLBI is 2 bits wide (for 4 sets). The loop has 1 memory (a write to *p) and increments our pointer by 8 shorts = 16 bytes = 0x10.

p starts at 0b01 1111 / 01/00 0000 0000, which will hit in set 1 (valid, tag 0x1F, write permissible). After running through this page, the next page we access will be in set 2 (valid, tag 0x1F, and write permissible). The next page (set 3, tag 0x1F) will cause a TLB miss – this will *definitely* cause a replacement, but *might* cause a page fault beforehand (the page table is not shown). In our increments of 16 bytes, the first address we access on this page is **0x1FC00**.

SID: __1234567___

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: Wrong allocation size / buffer overflow – `N*sizeof(float)`
Error 2: Using uninitialized values – `z[i] = A[i][j] + z[i] * x[j]`
Line of code with fixes: `double *z = (double *) calloc(N, sizeof(double));`

(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_60C_`	_32_ bytes	_20_ bytes

Block starts a header size before the payload (returned addr). Minimum block size in explicit free list is set by header+footer+2 pointers = 24 bytes, then aligned to 16-bytes: **32 bytes**. Internal fragmentation is size of block – payload size = 32 – 12 = 20 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` (circled)
Group 2:	`&foo`	`main` (circled)	`str`
Group 3:	`bar` (circled)	`&str`	`&ZERO`

7) `&foo/&bar` (Stack)
6) `foo` (Heap)
5) `&ZERO/&str` (Static Data)
4) `str` (Literals)
3) `main` (Code)
2) `bar` (351)
1) `ZERO` (0)