University of Washington – Computer Science & Engineering

Summer 2018 Instructor: Justin Hsia 2018-08-17

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 10:50.

Instructions

- This exam contains 5 double-sided pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- Please write your **S**tudent **ID** number (7 digits) in the upper-right corner of every odd page.
- 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 one page (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 60 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	1	2	3	4	5	6	Total
Possible Points	20	22	16	20	16	6	**100**

Question 1: Arrays & Structs [20 pts]

Answer the questions below about the following C code which *uses a struct*.
`strcpy(char *dst, char *src)` copies the string from `src` to `dst`, including the null-terminator.

```
1   typedef struct {
2     long traffic;
3     char code[3];
4     char *name;
5     int  built;
6   } airport;

7   airport seatac;
8   seatac.traffic = 46934194;  // 2017 annual passenger traffic
9   strcpy(seatac.code, "SEA");
10  seatac.name = "Seattle-Tacoma International Airport";
11  seatac.built = 1944;
12  airport airports[10];
```

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

sizeof(airport)	Internal	External

(B) Harry the Husky points out that Line 9 causes buffer overflow because `strcpy()` copies the null terminator! What is the minimum length of string literal (*e.g.* `"SEA"` is length 3) that we can use in the 2nd argument to `strcpy()` that overwrites useful data? [4 pt]

(C) Complete the formula below for value returned by **`&(airports[3].code[0])`**. You may use the variable `A` to represent `sizeof(airport)` (*i.e.* the correct answer to Part A). [4 pt]

`(char *)&airports +`

(D) Give an alternate ordering of the fields (one has been given for you) that **reduces the size of the struct** AND **eliminates internal fragmentation**: [6 pt]

(1) traffic	(2)	(3)	(4)

SID: _____

Question 2: Caching [22 pts]

We have 256 KiB of RAM and a 512-byte data cache that is 2-way set associative with 64-byte blocks and LRU 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 `long`s. Assume that `i` is stored in a register. How many **memory accesses** occur in *each* iteration of the for-loop? [1 pt]

```
#define N 256
long data[N];        // &data = 0x10000 (physical addr)
data[0] = 0;         // warm up the cache and initialize sum
for (i = 1; i < N; i += 1)
    data[0] += data[i];
```

(C) For the code above with an initially *cold* cache, what is the **first/smallest value of `i`** that causes a block to be **evicted** from the cache? [6 pt]

`i =`

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

Use `int` instead ——— Double the associativity ———

Random replacement ——— Half the block size ———

(E) Harry the Husky claims that we could improve our code's *performance* (this is unrelated to Part C) by **storing our sum in a register** inside our loop and then storing the final value into `data[0]` after the loop finishes. [4 pt]

Circle one: Agree Disagree

Explanation:

Question 3: Processes [16 pts]

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

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

(1) _____

(2) _____

(3) _____

(B) In the code above, we will refer to the 3 processes as the "parent," "child," and "grandchild." Who cleans up the grandchild process? **Circle the true statement** below. [2 pt]

Reaped by `parent`	Reaped by `child`	Reaped by `init/systemd`	Must be manually killed

(C) In the following blanks, write "**Y**" for yes or "**N**" for no if the following need to be updated *during* a **context switch**. [4 pt]

Stack _____ `%rsp` _____ PTBR _____ `%rip` _____

(D) Harry the Husky wants to write a concurrent algorithm using `fork()` where all processes **write to different parts of the same array in the heap**. Will this work or not? Explain *briefly*. [4 pt]

Circle one: Yes No

Explanation:

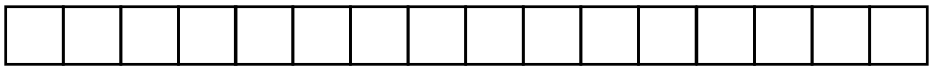
SID: _____

Question 4: Virtual Memory [20 pts]

Our system has the following setup:

- 16-bit virtual addresses and 12-bit physical addresses with 128-B pages
- A 4-entry TLB that is 2-way set associative with LRU replacement
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)

(A) The individual bits of a virtual address are shown below. For the following fields, indicate the *highest* and *lowest* bit (these could be the same) that constitute that field: [8 pt]



VPO: _____ to _____ TLBI: _____ to _____

VPN: _____ to _____ TLBT: _____ to _____

(B) Name **three separate benefits** of using virtual memory (instead of physical addressing): [6 pt]

(1)
(2)
(3)

(C) If the TLB is in the state shown (permission bits: 1 = allowed, 0 = disallowed), give example addresses that will fulfill the following scenarios: [6 pt]

TLBT	PPN	Valid	R	W	X
0x70	0x03	1	1	0	0
0x72	0x1D	0	1	0	0
0x04	0x14	1	1	1	1
0x71	0x02	1	1	1	0

A *write* address that causes a TLB Hit and segmentation fault: `0x`

A value in `%rip` that causes a TLB Hit and no exception: `0x`

Question 5: Memory Allocation [16 pts]

(A) Below is the current state of the heap after the following sequence of allocations and frees:

A allocated, **B** allocated, **C** allocated, **B** freed, **D** allocated

Start End

A	**A**	**A**			**C**	**D**	**D**		

Which allocation strategy was used? [4 pt]

(B) We are designing a dynamic memory allocator for a **64-bit computer** with **8-byte boundary tags** and **alignment size of 16 bytes** using an **explicit free list**. Assume a footer is always used. Answer the following questions: [6 pt]

Maximum tags we can fit into the header (not counting `size`): _____ tags

Minimum block size: _____ bytes

Minimal amount of *internal fragmentation* in block allocated by `malloc(7)` request: _____ bytes

(C) Consider the C code shown here. Assume that the `malloc` call succeeds and that `foo` and `str` 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;

int main(int argc, char *argv[]) {
    char *str = "cse351";
    int *foo = malloc(8);
    return 0;
}
```

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

SID: _____

Question 6: C and Java [6 pts]

All of the low-level concepts like data representation, memory management, and compilation that we illustrated in this class using C apply in one way or another to *all* higher-level languages because at the end of the day, it's just a CPU executing machine instructions! We briefly showed some *possible* Java implementation details.

Use the word bank below to write in the **351 concept that is *most similar*** to the following Java concepts discussed in lecture:

Java Concept	**Similar 351 Concept**
Reference	_____
Object	_____
Arrays store their own length	_____
Virtual method table (vtable)	_____

Word Bank:

boundary tag	buffer overflow	casting	exception
free list	jump table	malloc	pointer
sign extension	stack overflow	struct	typedef

This page purposely left blank

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

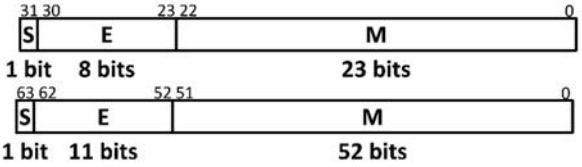
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

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
jb	"Below" (unsigned >)	d (op) s < 0U	b & a > 0U	b < a

Registers

Name	Convention	Name of "virtual" register		
		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