University of Washington – Computer Science & Engineering

Summer 2018 Instructor: Justin Hsia 2018-08-17

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 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.

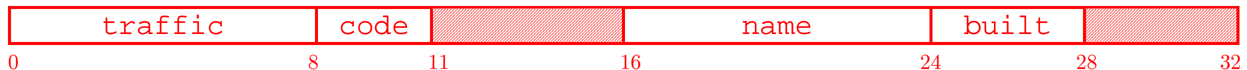
```
1   typedef struct {   // K:
2     long traffic;    // 8
3     char code[3];    // 1
4     char *name;      // 8
5     int  built;      // 4
6   } airport;         // Kmax = 8

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
32 bytes	**5 bytes**	**4 bytes**

Alignment requirements listed above in the code as red comments next to the struct fields. An `airport` instance will look as shown below:



Between `code` and `name` is internal fragmentation; external fragmentation is at the end.

(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]

This causes the null terminator to overwrite the first byte of `name`.

8

(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 + ` **`3*A + 8`**

(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) **name**	(3) **built**	(4) **code**

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
10	2	6

18 address bits. $\log_2 64 = 6$ offset bits. 512-B cache = 8 blocks. 2 lines/set → 4 sets.

(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];
```

3 (R `data[0]`, R `data[i]`, W `data[0]`)

(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 = ` **64**

`&data[0]` is the start of a block that goes in set 0. 64 bytes per block = 8 `long`s so every 8 values of `i` pulls a new block into the cache. We have to load 8 blocks (`i` from 0 to 63) to fill the cache (2 blocks per set into 4 sets). The next value of `i` then evicts a block from set 0.

(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	—

The answer to Part C is determined only by the number of array elements we can fit in the cache (data type and cache 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: Accessing registers is faster than the cache hit time, so reducing the number of memory accesses (regardless of hit or miss) will improve our execution time.

Question 3: Processes [16 pts]

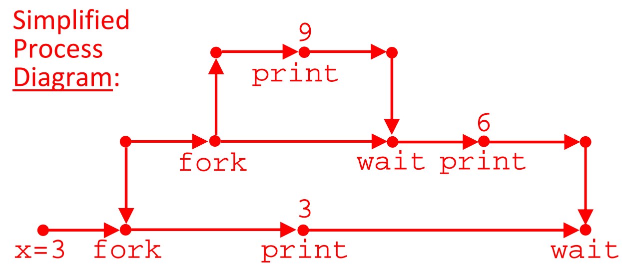
(A) The following function prints out *three numbers*. In the following blanks, **list three possible outcomes**: [6 pt]

```
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) _3 9 6 ________

(2) _9 3 6 ________

(3) _9 6 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 __N__ `%rsp` __Y__ PTBR __Y__ `%rip` __Y__

Each process has its own virtual address space, however, we do need to switch over to the new page table. Registers are part of the process state and must be updated for the new process.

(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: Each process gets its own independent virtual address space, so you can't share data between processes this way.

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]

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

VPO: __**6**__ to __**0**__ TLBI: __**7**__ to __**7**__

VPN: __**15**_ to __**7**__ TLBT: __**15**_ to __**8**__

VPO contains the $\log_2 128 = 7$ least significant bits, with the rest making up the VPN. The VPN is split into TLBT and TLBI: TLB has two sets so $\log_2 2 = 1$ TLBI bit.

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

Possible answers:
- Simplifies memory management for programmers (each process gets its own virtual address space).
- Enforces protection and sharing between processes.
- Adds disk to the memory hierarchy (treats physical memory as a cache for disk).
- Gives each process the illusion of more memory than physical memory.

(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]

Find the desired entry in the TLB, then use the set number (0 or 1) and stored TLBT to reconstruct the desired VPN. Any VPO within this page will access that TLB entry.

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:
Want TLB entry with `V=1`, `W=0` → Set `0`, TLBT `0x70`.

0x**7000**-0x**707F**

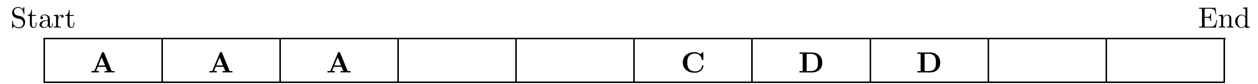
A value in `%rip` that causes a TLB Hit and no exception:
Want TLB entry with `V=1`, `X=1` → Set `1`, TLBT `0x04`.

0x**0480**-0x**04FF**

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



Which allocation strategy was used? [4 pt]

next-fit

D was allocated after C instead of after A, so can't be first-fit. Best-fit also would have put D after A to remove external fragmentation.

(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`): __**4**___ tags

Minimum block size: __**32**__ bytes

Minimal amount of *internal fragmentation* in block allocated by `malloc(7)` request: __**25**__ bytes

- With 16-byte alignment, lowest 4 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 8 bytes each and pointers are 8 bytes in a 64-bit architecture.
- Everything but the payload counts as internal fragmentation, so 32 – 7 = 25.

(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

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

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	**pointer**
Object	**struct**
Arrays store their own length	**boundary tag**
Virtual method table (vtable)	**jump table**

Word Bank:

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