CSE 351 Spring 2019– Final Exam (12 June 2019)

Please read through the entire examination first!

- You have 110 minutes for this exam. Don't spend too much time on any one problem!
- The last page is a reference sheet. Feel free to detach it from the rest of the exam.
- The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones).

There are 9 problems for a total of 90 points. The point value of each problem is indicated in the table below. Write your answer neatly in the spaces provided.

Please do not ask or provide anything to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

POINTS WILL BE DEDUCTED if you are writing/erasing after the final bell has rung!

Good Luck!

Your Name:____Sample Solution_____

UWNet ID:_____

Problem	Topic	Max Score
1	Caches	15
2	Processes	10
3	Virtual Memory	12
4	Memory Allocation	11
5	Java	9
6	Compilation & Structs	8
7	Representation	10
8	Pointers & Memory	9
9	Buffer Overflow	6
TOTAL		**90**

1. Caches (15 points total)

You are using a byte-addressed machine with 64 KiB of Physical address space. You have a 2-way associative L1 data cache of total size 256 bytes with a cache block size of 16 bytes. It uses LRU replacement and write-allocate and write-back policies.

a) [2 pt] Give the number of bits needed for each of these:

Cache Block Offset: ____**4**_____ Cache Tag: ____**9**_____

b) [1 pt] How many **sets** will the cache have? ___**8**_____

c) [4 pts] Assume **`i`** and **`j`** are stored in registers, and that the array **`x`** starts at address 0x0. Give the miss rate (as a fraction or a %) for the following two loops, assuming that the cache starts out empty.

```
#define LEAP 2
#define SIZE 128
int x[SIZE];
... // Assume x has been initialized to contain values.
... // Assume the cache starts empty at this point.
for (int i = 0; i < SIZE; i += LEAP) {        // Loop 1
  x[i] = x[i] + i * i;
}
for (int j = 1; j < SIZE; j += LEAP) {        // Loop 2
  x[j] = x[j] + j * 2;
}
```

Miss Rate for Loop 1: ___**25%**_____ **Miss** Rate for Loop 2: ___**25%** _____

d) [8 pts] For each of the changes proposed below, indicate how it would affect the **miss rate** of each loop above in part c) *assuming that all other factors remained the same* as they were in the original problem. Circle one of: "increase", "no change", or "decrease" for each loop.

Change associativity from 2-way to direct mapped:	Loop 1:	increase	/	**no change**	/	decrease
	Loop 2:	increase	/	**no change**	/	decrease
Change **`LEAP`** from 2 to 4:	Loop 1:	**increase**	/	no change	/	decrease
	Loop 2:	**increase**	/	no change	/	decrease
Change cache size from 256 bytes to 512 bytes:	Loop 1:	increase	/	**no change**	/	decrease
	Loop 2:	increase	/	no change	/	**decrease**
Change block size from 16 bytes to 32 bytes:	Loop 1:	increase	/	no change	/	**decrease**
	Loop 2:	increase	/	no change	/	**decrease**

2. Processes (10 points total)
The following function prints out numbers.

```
void sunny(void) {
  int x = 4;
  if (fork()) {
    x += 6;
  } else {
    x += 1;
  }
  printf("%d ", x);
  if (fork()) {
    x += 1;
  } else {
    x -= 2;
  }
  printf("%d ", x);
  fork();
  exit(0);
}
```

Several answers are possible here. In each valid output 10 must always come before 8 and 11, AND 5 must always come before 6 and 3. If you actually run this code you may get confusing results unless you insert a call to `fflush()` after each `printf`.

a. [3 pts] List 3 possible outputs of the code above:

(1) __10 5 8 11 6 3_____

(2) __10 5 11 8 6 3_____

(3) __5 3 6 10 11 8_____

b. [2 pts] What is the total number of processes created (including the original process that called **`sunny`**) by this function?

8

c. [1 pt] Is it possible for the numbers to appear in descending order (highest value to lowest value) in the output?

YES / **NO**

d. [2 pts] The function call **`fork()`** returns something. Describe, in general, what **`fork()`** returns?

`fork()` returns 0 to the child process, and the PID of the child to the parent process

e. [2 pts] When context-switching from a process A to a process B, which elements of process B's state must be restored before process B can begin executing:

- Contents of registers **YES** / NO
- Contents of L1 cache YES / **NO**
- Contents of PTBR **YES** / NO
- Contents of TLB YES / **NO**

3. Virtual Memory (12 points)

Assume we have a virtual memory detailed as follows:

- 8 KiB Virtual Address Space,
- 2 KiB Physical Address Space,
- a TLB with 16 entries that is 4-way set associative with LRU replacement
- 64 B page size

a) [5 pts] How many bits will be used for:

Page offset? ___**6**_____

Virtual Page Number (VPN)? __**7**_____ Physical Page Number (PPN)? ___**5**_____

TLB index? ____**2**_____ TLB tag? _____**5**_____

b) [1 pt] How many TOTAL entries are in this page table?
(It is fine to leave your answer as powers of 2).

____**2^7 or 128**_____

3. (cont.) The current contents of the TLB and (partial) Page Table are shown below:

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	07	00	1	06	-	0	1F	03	1
1	00	0B	1	0A	-	0	0C	03	1	01	0F	1
2	07	-	0	0C	02	1	0F	01	1	0B	-	0
3	01	1C	1	0C	01	1	04	01	0	1A	01	1

Page Table (only first 16 of the PTEs are shown)

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
00	03	1	04	-	0	08	07	1	0C	0F	1
01	0B	1	05	0F	1	09	-	0	0D	-	0
02	03	1	06	-	0	0A	01	1	0E	06	1
03	03	1	07	1C	1	0B	08	1	0F	0A	1

c) [6 pts] Determine the physical address, TLB miss or hit, and whether there is a page fault for the following virtual address accesses (write "Y" or "N" for yes or no, respectively, in the TLB Miss? And Page Fault? columns). If you can't determine the PPN and/or physical address and/or TLB miss and/or Page Fault, simply write ND (for non-determinable) in the appropriate entry in the table.

Virtual Address	VPN (give bits)	TLBT (give bits)	TLBI (give bits)	PPN (give bits)	Physical Address (give bits)	TLB Miss?	Page Fault?
0x1306	*1001100*	*10011*	*00*	*ND*	*ND*	*Y*	*ND*
0x0C62	*0110001*	*01100*	*01*	*00011*	*00011 100010*	*N*	*N*
0x02C3	*0001011*	*00010*	*11*	*01000*	*01000 000011*	*Y*	*N*

4. Memory Allocation (11 points total)

```
1     #include <stdlib.h>
2     float pi = 3.14;
3
4     int main(int argc, char *argv[]) {
5       int year = 2019;
6       int* happy = malloc(sizeof(int*));
7       happy++;
8       free(happy);
9       return 0;
10    }
```

a) [3 pts] Consider the C code shown above. Assume that the `malloc` call succeeds and `happy` and `year` are stored in memory (not in a register). Fill in the following blanks with "<" or ">" or "UNKNOWN" to compare the *values* returned by the following expressions just before `return 0`.

&year ___>_____ &main

happy ___<_____ &happy

&pi ___<_____ happy

b) [4 pts] The code above has two memory-related errors. Use the line numbers in the code to describe what the errors are and where they occur.

Error #1: **On line 6 we are requesting more memory than we need. We should be requesting size of `int` (4 bytes), not size of `int*` (8 bytes). Alternatively we could have meant to declare happy to be of type `int**` (a pointer to a pointer to an `int`) so that we would have needed 8 bytes to hold a pointer to an `int`.**

Error #2: **On line 8 we are calling free on a pointer that was not the one returned to us by malloc. In line 7 we are incrementing happy (a pointer to an `int` that was returned to us by malloc).**

c) [2 pts] (Not related to code at top of page) Give one advantage that next fit placement policy has over a first fit placement policy in an implicit free list implementation.

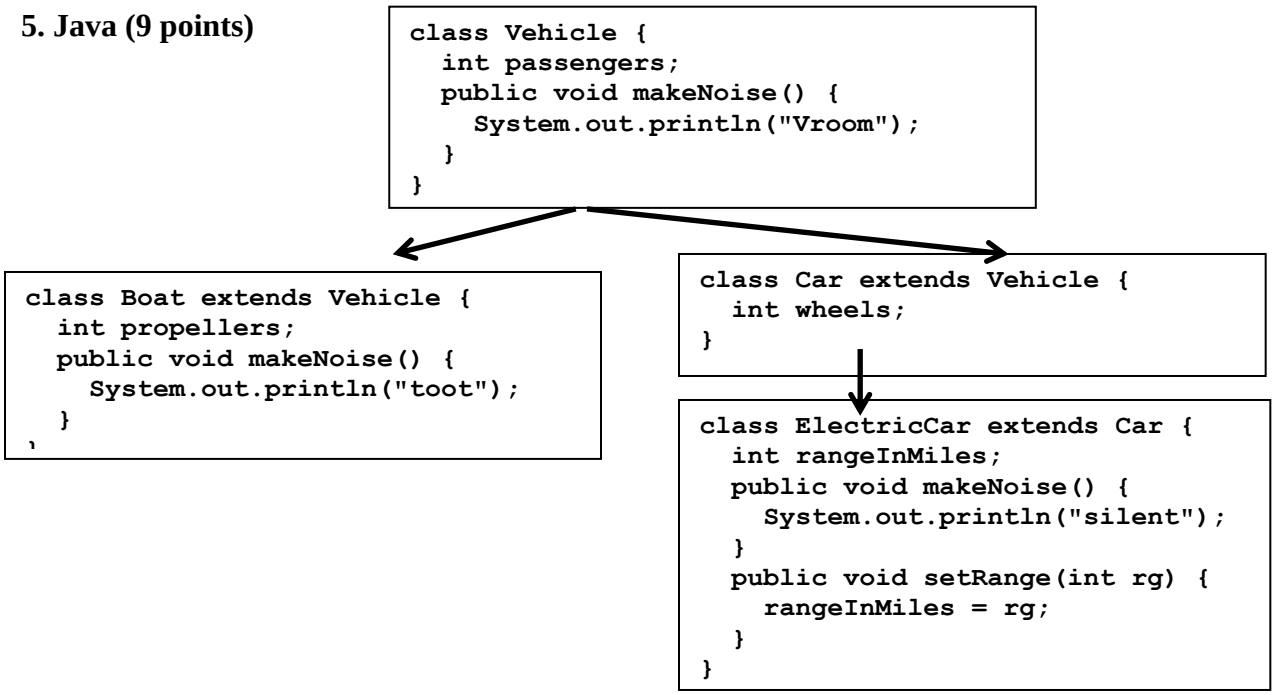
Next fit searches the list starting where the previous search finished. This should often be faster than first fit because it avoids re-scanning unhelpful blocks. First fit always starts searching at the beginning of the list. In an implicit free list this is particularly bad because the "free" list actually contains all allocated blocks as well as free blocks. So starting from the beginning of the list is likely to traverse many allocated blocks each time.

d) [2 pts] List two reasons why it would be hard to write a garbage collector for the C programming language.

Reason #1: **Pointers in C can point to a location other than the beginning of a block of memory on the heap.**

Reason #2**: In C you can "hide" pointers e.g. by casting them to longs.**

5. Java (9 points)



a) Given our discussion in class, circle whether you would expect the following to be True or False:

i. **TRUE** / FALSE: An instance of the `Car` class will be the same size as an instance of the `Boat` class.

ii. TRUE / **FALSE**: An instance of the `ElectricCar` class will be the same size as an instance of the `Boat` class.

iii. **TRUE** / FALSE: The vtable for a `Car` will be the same size as the vtable for a `Boat`.

iv. TRUE / **FALSE**: The vtable for a `ElectricCar` will be the same size as the vtable for a `Car`.

v. TRUE / **FALSE**: Each instance of a class will have a separate copy of the vtable for that class.

vi. **TRUE** / FALSE: Each instance of the `ElectricCar` class will initially contain the value 0 for `rangeInMiles` until `setRange()` is called

b) More Java….

vii. TRUE / **FALSE**: The Java Virtual Machine converts Java instructions into bytecodes.

viii. TRUE / **FALSE**: The Java compiler can always detect if an array reference is out of bounds at compile time.

ix. TRUE / **FALSE**: The programmer determines if Java objects are allocated on the stack or the heap.

6. Compilation and Structs (8 points)

a) [2 pts] Assume that we compile a C source file into an object file. Which part of the object file keeps track of the symbols and labels needed later by the code in that file?

Relocation table

b) [2 pts] The tool used to combine one or more .o files into an executable is called the: _____ (Hint: the answer is not "`gcc`", we want the name of tool that does this particular step.)

Linker

c) [4 pts] For this question, assume a 64 bit machine and the following C struct definition.

```
typedef struct {
    short year;
    char *title;
    char artist[16];
    float rating;
} song;
```

- [1 pt] What does **`sizeof(song)`** return?_____**40**_____

- [1 pt] Is there any internal fragmentation? If so, how many bytes and where?

  Yes, 6 bytes between `year` and `title`

- [1 pt] Is there any external fragmentation? If so, how many bytes and where?

  Yes, 4 bytes at the end after `rating`

- [1 pt] Is there an ordering of the fields that reduces the amount of fragmentation in the struct? If yes, provide the order. If not, explain why not.

  Yes. The order: `title, rating, year, artist` will result in 32 bytes total and no internal fragmentation, just two bytes of external fragmentation at the end. Other orders also get to this size.

7. Representation (10 points)

a) [4 pts] Consider the **`signed char x = 0b 1000 0110`**

i. What is the value of **`x`**? You may answer as the sum of powers of 2.

$-2^7 + 2^2 + 2^1 = -122$

ii. Evaluate each of the following expressions:

`x & (x >> 4)`	`~x`	`x ^ 0xC2`
`(arithmetic shift, since x is signed)`		
`0b 1000 0000 __`	`0b 0111 1001__`	`0b 0100 0100__`

b) [3 pts] What 32-bit bit pattern would be used in IEEE 754 floating point to represent the decimal value -1 (e.g. in a C float)?

`__1____`	`___01111111_____`	`____00000000000000000000000_____`
S (1 bit)	E (8 bits)	M (23 bits)

c) [3 pts] On a 64-bit word machine, you are given the following array declaration in C:

```
double x[8][2]
```

If `x` starts at address 0, what will the expression `&(x[2][4])` evaluate to? If "unknown" or "cannot be guaranteed", state that. Otherwise give your answer as a single number in **decimal**.

0 + 2 * 2 * 8 + 4 * 8 = 32 + 32 = 64

8. Pointers & Memory (9 points)
We are using a 64-bit x86-64 machine (**little endian**). Below is the `husky` function disassembly, *showing where the code is stored in memory*. Hint: read the questions before reading the assembly!

```
0000000000400507 <husky>:
  400507:       48 83 fe 02             cmp    $0x2,%rsi
  40050b:       7f 05                   jg     400512 <husky+0xb>
  40050d:       48 8d 04 7f             lea    (%rdi,%rdi,2),%rax
  400511:       c3                      retq
  400512:       48 83 ec 08             sub    $0x8,%rsp
  400516:       48 83 ee 01             sub    $0x1,%rsi
  40051a:       e8 e8 ff ff ff          callq  400507 <husky>
  40051f:       48 83 c4 08             add    $0x8,%rsp
  400523:       c3                      retq
```

a) [4 pts] What are the values (in hex) stored in each register shown after the following x86 instructions are executed? *Remember to use the appropriate bit widths.*

```
movswl 4(%rsi,%rax), %ecx

leaw (%rsi,%rsi,2), %di
```

Register	Value (in hex)
%rax	0x0000 0000 0040 050d
%rsi	0x0000 0000 0000 0010
%rcx	0x0000 0000 0000 08c4
%di	0x0030

b) [4 pts] Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let **`short* shortP = 0x400514`**

```
short v1 = shortP[__-3____];                    // set v1 = 0x048d

long* v2 = (long*) ((__int_____*)shortP + 3); // set v2 = 0x400520
```

c) [1 pt] **`husky`** is a recursive function. What address is put on the stack when **`husky`** calls itself. Give the exact address:

0x40051f

9. Buffer Overflow (6 points)
The following piece of C code is vulnerable to buffer overflow:

```
void foo() {
    char buf[8];
    gets(buf);
    printf("You typed %s!\n", buf);
}

int main() {
    foo();
    return 0;
}
```

a) [2 pts] What line of this C code is vulnerable, and why?

The call to `gets(buf)` in foo is vulnerable because `gets` doesn't check array bounds and can overwrite the return address.

The x86-64 assembly below corresponds to the C code above:

```
.LC0:
        .string "You typed %s!\n"
foo:
        subq    $24, %rsp
        movq    %rsp, %rdi
        call    gets
        movq    %rsp, %rsi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        addq    $24, %rsp
        ret
main:
        subq    $8, %rsp
        call    foo()
        movl    $0, %eax
        addq    $8, %rsp
        ret
```

b) [2 pts] How many bytes do you need to enter to overwrite the return address to `main` with a stack address?

30 bytes. 24 bytes to fill `buf`, and 6 more bytes to overwrite the lower 6 bytes of the original return address. Stack addresses start with `0x00007fff...` and the original return address will be low down in memory, so no need to overwrite the most significant bytes (and x86 is little-endian, so the most significant bytes are at the highest addresses).

c) [2 pts] Suppose you know that there is a function at memory address `0x40806c` that you want to execute. What bytes can you give as input such that the vulnerable program will call your function? (Note: we are looking for bytes, not ASCII characters). If you need to enter the same byte multiple times, you may write "<byte> * <number of times>"

`ffffffffffffffffffffffffffffffffffffffffffffffff6c80400000000000`
or <ff>*<24> `6c80400000000000`
(the leading 24 bytes of 'f' can be any character)