

CSE 351 – Final Exam – Spring 2016

June 8, 2016

Name: _____

UWNetID: _____

Please do not turn the page until told to.

Instructions

- The exam is closed book, closed notes, no calculators, no mobile phones, no laptops.
- **You will have 1 hour and 50 minutes. Please stop promptly at 4:20 (or when asked).**
- There are **160** points total, divided unevenly among **8** problems (each with multiple parts).
- The exam is **printed double-sided**. If you separate any pages, be sure to print your name at the top of each separated page so we can match them up.
- Useful **reference material** can be found on the last few pages of the exam. Feel free to tear it off.

Advice

- Read questions carefully before starting. If you do not understand a question or are unsure of what is expected for an answer, *please ask!*
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- Questions are not necessarily in order of difficulty; some can be done relatively quickly if you know the answer, while others take more time. So **make sure you get to all the questions**.
- Relax. Take your time, think critically, and use the provided reference material to your advantage.

Problem	Points
1. Structs & Data Layout	30
2. Optimizing for Big Data	15
3. Address Translation	25
4. Operation: Cache Is King	20
5. Java of the C	10
6. Programs, processes, and processors (oh my!)	25
7. Bug Zapper	15
8. Miscellaneous	20

Name: _____

1. Structs & Data Layout (30 pts)

Use the following snippet of C code to answer the questions for this problem.

```
typedef struct Item {
    char * name;
    int quantity;
    double weight;
} Item;

Item * inventory[100];
size_t inventory_size = 0;

void add_to_inventory(Item * item) {
    inventory[inventory_size] = item;
    inventory_size++;
}

int main(int argc, char const *argv[]) {
    int num_groceries = 2;
    Item * groceries = (Item*)malloc(2*sizeof(Item));
    groceries[0].name = "lemonade";
    /* ... */

    Item * favorite = inventory + 1;
    /* ... */
}
```

A	Kernel memory
B	User stack
C	Library data
D	Heap
E	Read/write segment (.data, .bss)
F	Read-only segment (.init, .text, .rodata)
G	Unused

- (a) Each of the following C expressions computes an address. Fill in the table with the C type (some sort of pointer, for example, int*), and the memory region in which the address would be, using the letters from the table above (for example, if the address is in *kernel memory*, then you would answer "A").

C expression	C Type	Memory region (A-G)
&inventory_size		
&groceries[1]		
groceries[0].name		
&num_groceries		
add_to_inventory (<i>address of function</i>)	void (*)(Item*)	

- (b) Does Item have any internal or external fragmentation? If so, where is it?

Name: _____

(c) Is it possible to reduce the size of `Item` by reordering its members? If so, what reordering would minimize the size? If not, why not? Explain in about 1 sentence.

(d) Fill in the following table with the *value* of each C expression, using the given initial values. Write "?" if the value cannot be determined. *Note:* these addresses are fake, they will not help you solve part (a) above, sorry. It may help you to draw a picture of memory (and may help us give partial credit).

C expression	Value (hex or decimal)
"lemonade"	0x2000
inventory	0x6000
groceries	0x4000
sizeof(Item)	
&groceries[0].quantity	
&groceries[1].weight	
groceries[0].name + 3	
&favorite->name	

(e) Write a function called `add_avocado()`, with return type `void`, that adds avocados to the inventory (new item with name: "avocado", quantity: 4, weight: 0.24). Your implementation should use the provided `add_to_inventory` function. Don't worry about missing semicolons. *See reference sheet for C functions.*

```
void add_avocado() {
```

```
}
```

Name: _____

2. Optimizing for Big Data (15 pts)

Imagine you are working on a new deep learning system (a "big data" application) that must process a vast number of images in order to learn patterns (such as cute cat pics). However, it is not running as fast as you need it to, so you are trying to improve its performance. The following questions will have you try to optimize the application by changing different aspects of the system.

- (a) The application usually operates with batches of many large images at a time. When running a single instance, performance is acceptable, but when running two instances in different processes, memory accesses frequently take several thousand times longer than a typical DRAM (main memory) access should. What is most likely causing this slowdown? What could you change about the *hardware* to fix this?

- (b) The application allocates and frees lots of 32-byte blocks from the heap. What could you do to make the memory allocator work better for this use case? Say whether your optimization would improve *throughput* or *utilization* (or both). Answer does not need to be complete sentences.

- (c) When processing images, the application sequentially accesses each element in order with a stride of 1. What single change to the cache geometry would most improve (reduce) the miss rate? (*circle one*)
 - A. Increase cache block size.
 - B. Increase number of sets (and decrease associativity).
 - C. Increase associativity (and decrease number of sets).
 - D. Switch to write-through and write-no-allocate.

- (d) Registers and caches typically use the same technology (SRAM) to store data, whereas main memory is typically implemented out of a slower technology called DRAM.
 - (i) Why, then, are registers faster than L1 cache hits? *Give one (short) reason.*

 - (ii) If registers and L1 are so fast, why don't we build all of memory this way? *Give one (short) reason.*

Name: _____

3. Address Translation (25 pts)

Imagine we have a machine with 16-bit virtual addresses, 12-bit physical addresses, and:

- Page size of 256 bytes.
- Translation lookaside buffer (TLB) with 8 ways and 4 sets.
- One-level cache with capacity of 256 bytes, 16-byte cache block size, and 2-way associativity.

(a) For the virtual address below, label the bits used for each component, either by labelling boxes or with arrows to indicate ranges of bits. *Hint:* there may be more than one label for some bits.

- Virtual page offset ("VPO")
- Virtual page number ("VPN")
- TLB index ("index")
- TLB tag ("tag")

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

(b) How many total page-table-entries are there per process in this system?

(c) How many **sets** are there in the cache?

(d) Assume that the virtual address above has been translated to a physical address in memory.

Fill in the known bits of the physical address below, and **label the bits for each component** as you did in part (a) — again, some bits may have more than one label.

- Physical page number ("PPN")
- Physical page offset ("PPO")
- Cache index ("index")
- Cache tag ("tag")
- Cache offset ("offset")

11	10	9	8	7	6	5	4	3	2	1	0

Name: _____

(e) *Choose your own adventure.* For this problem, there are two different scenarios. Your job is to number the events according to the order in which they would occur for the given scenario. Leave blank any events that would not occur for that scenario. When you get to a "STOP HERE" in a scenario, then you'll know that adventure has come to an end.

(i) **Page fault (first column).** Memory access resulting in a page fault for an allocated page (stop when the page fault has been handled, but data has not been returned).

(ii) **Page hit (second column).** Memory access from before re-executes, resulting in a page hit now.

Order for page fault:	Order for page hit:	Events:
1	1	CPU issues virtual memory access.
		CPU translates virtual address to physical address.
		Page fault detected.
		MMU fetches page table entry from memory.
		MMU constructs physical address using page table entry.
		MMU checks TLB for page table entry and <i>finds it</i> .
		MMU checks TLB for page table entry and <i>does not find it</i> .
		Page fault handler chooses a page to evict and does so.
		Load requested page into memory and add page table entry to TLB.
		Load requested block into cache.
		Lookup physical address in <i>main memory</i> .
		Lookup physical address in <i>cache</i> , not found.
		Lookup physical address on <i>disk</i> .
		Return control to program <i>before</i> original memory access. STOP HERE.
		Return control to program <i>after</i> original memory access. STOP HERE.
		Return data to the program. STOP HERE.

Name: _____

4. Operation: Cache Is King (20 pts)

Your mission, should you choose to accept it (please do), is to simulate executing memory accesses on a system with a cache. Assume that the contents of physical memory (in hex) are as shown below:

	+0	+1	+2	+3	+4	+5	+6	+7
0x00	00	01	02	03	04	05	06	07
0x08	ca	fe	be	ef	fe	ed	de	ad
0x10	a0	b0	c0	d0	e0	f0	f1	f2
0x18	0a	0b	0c	0d	0e	0f	1f	2f
0x20	00	11	22	33	44	55	66	77
0x28	88	99	aa	bb	cc	dd	ee	ff

Now, simulate executing the following memory accesses on a 2-way set-associative cache, with 4-byte cache blocks and 4 sets and an LRU (least recently used) replacement policy; treat each access as a single-byte read (e.g. char read). Fill in the contents of the cache as you go, then fill in the value read and whether or not the access was a hit or a miss. Assume the cache starts empty. If the contents of a cache cell change, then cross out the old value (or erase) and write the new value. You may leave any unmodified cells empty.

	Access (1-byte):	Value	Hit (H) or Miss (M)?
(a)	0x08		
(b)	0x13		
(c)	0x19		
(d)	0x0b		
(e)	0x28		
(f)	0x29		

Set	Tag (binary)	Valid	Data (in hex, leave off 0x)				Tag (binary)	Valid	Data			
0												
1												
2												
3												

Name: _____

5. Java of the C (10 pts)

Imagine you prefer programming in Java, but your curmudgeonly boss (or instructor) insists that you use C. You miss Java's convenience and safety features, so to make your life easier, you start to implement some of them yourself in C. You start by implementing a function, `error`, to serve like an "exception" in Java: it takes a string message as an argument, prints the error message, then exits the program with an error code. Now you are trying to implement an array data structure that behaves similar to Java's, using the struct declared below. In your implementations, you need not worry about semicolons.

```
void error(char * message) {  
    printf("error: %s", message);  
    exit(1);  
}
```

```
typedef struct Array {  
    int length; // number of elements in the array  
    int * data; // pointer to storage for array data  
} Array;
```

- (a) **Array Constructor.** Create a Java-like constructor *in C* that creates a new array of integers (using the `Array` struct from above) and initializes it. *Hint:* Java arrays do not have a vtable, but think carefully about everything else that Java does when creating new objects. *See reference sheet for C library functions.*

```
Array* new_Array(int n) {
```

```
}
```

- (b) **Array Out Of Bounds.** Implement a safe array access function in C, using the `error` function to exit if there was an attempted out of bounds access.

```
int get(Array* a, int i) {
```

```
}
```

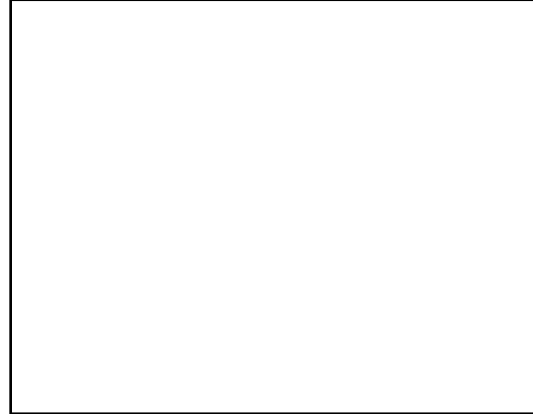

Name: _____

6. Programs, processes, and processors (oh my!) (25 pts)

- (a) Consider the following C code on the left (running on Linux), then give *one* possible output of running it. Assume that `printf` flushes its output immediately.

```
void oz() {
    char * name = "toto\n";
    printf("dorothy\n");
    if (fork() == 0) {
        name = "wizard\n";
        printf("scarecrow\n");
        fork();
        printf("tinman\n");
        exit(0);
        printf("witch\n");
    } else {
        printf("lion\n");
    }
    printf(name);
}
```

Possible output:



- (b) "*Pay no attention to the man behind the curtain.*" We have seen several different mechanisms used to create illusions or abstractions for running programs:

- A. Context switch
- B. Virtual memory
- C. Virtual method tables (vtables)
- D. Caches
- E. Timer interrupt
- F. Stack discipline
- G. None of the above, or impossible.

For each of the following, indicate which mechanism above (A-F) enables the behavior, or G if the behavior is impossible or untrue.

- (i) _____ Allows operating system kernel to run to make scheduling decisions.
- (ii) _____ Prevents buffer overflow exploits.
- (iii) _____ Allows multiple instances of the same program to run concurrently.
- (iv) _____ Lets programs use more memory than the machine has.
- (v) _____ Makes recently accessed memory faster.
- (vi) _____ Multiple processes appear to run concurrently on a single processor.
- (vii) _____ Enables programs to run different code depending on an object's type.
- (viii) _____ Allows an x86-64 machine to execute code for a different ISA.

Name: _____

(c) Give an example of a *synchronous* exception, what could trigger it, and where the exception handler would return control to in the original program.

(d) In what way does address translation (virtual memory) help make *exec* fast? Explain in less than 2 sentences. *Hint*: it may help to write down what happens during *exec*.

(e) Which of the following *can* a running process determine, assuming it does *not* have access to a timer? (*check all that apply*)

- Its own process ID
- Size of physical memory
- Size of the virtual address space
- L1 cache associativity
- When context switches happen

(f) For each of the following, fill in what is responsible for making the decision: hardware ("HW"), operating system ("OS"), or program ("P").

- (i) _____ Which physical page a virtual page is mapped to.
- (ii) _____ Which cache line is evicted for a conflict in a set-associative cache.
- (iii) _____ Which page is evicted from physical memory during a page fault.
- (iv) _____ Translation from virtual address to physical address.
- (v) _____ Whether data is stored in the stack or the heap.
- (vi) _____ Data layout optimized for spatial locality

Name: _____

7. Bug Zapper (15 pts)

In this problem, it will be your job to hunt down the bug or performance problem in each snippet of x86 or C code. Indicate the line that causes the error by circling it, then give a brief explanation (less than a sentence); something like: "wrong size" or "this should only be called once". If there is no bug, then write "No problems."

(a)

```
void shell_exec(char* command, char* args[]) {
    if (fork() == 0) {
        execv(command, args);
        int status;
        wait(&status);
    }
}
```

(b)

```
void work() {
    char * buf = (char*)malloc(sizeof(char) * 8);
    gets(buf);
    do_something(buf);
    free(buf);
}
```

(c)

```
void iterate() {
    int * a = (int*)malloc(sizeof(int) * 4);
    for (int i=0; i < 4; i++) {
        *a = 0;
        a += 1;
    }
    free(a);
}
```

(d)

```
convert():
    movsbq (%rdi), %ax
    ret
```

(e) Indicate where the problem is in the **x86-64 translation** of this C function.

```
long sum(long * array, long n) {
    long total = 0;
    for (long i=0; i < n; i++) {
        total += array[i];
    }
    return total;
}

sum(long*, long):
    movl    $0, %edx
    movl    $0, %eax
.L3:
    cmpq   %rsi, %rdx
    jl    .L2
    addq   (%rdi,%rdx,8), %rax
    addq   $1, %rdx
    jmp   .L3
.L2:
    rep   ret
```

Name: _____

- (f) Indicate where the problem is in the **x86-64 translation** of this C function.

```
void foo(int * x) {
    return *x + 1;
}

foo(int*):
    leal (%rdi), %eax
    addl $1, %eax
    ret
```

- (g) *Performance debugging*. For this one, rewrite the row_sums function to improve performance.

```
int N = 8192;
int big_data[N][N];

void row_sums(int * sums) {
    for (int i=0; i < N; i++) {
        sums[i] = 0;
    }
    for (int i=0; i < N; i++) {
        for (int j=0; j < N; j++) {
            sums[j] += big_data[j][i];
        }
    }
}
```

Name: _____

8. Miscellaneous (20 pts)

(a) Which of the following are valid reasons why virtual memory pages should be larger than cache blocks? (check all that apply)

- Physical memory is always smaller than virtual memory.
- Otherwise page tables would not fit in memory.
- It takes much longer to access disk than memory.
- The TLB typically holds fewer entries than the cache.

(b) Which of the following can *only* happen for a mis-aligned memory access (such as a movq to an address that is not a multiple of 8)? (check all that apply)

- Load 2 cache lines for one memory access.
- Stack smashing.
- Misaligned address exception leading to an abort.
- A TLB hit and a page fault for a single access.

(c) Which of the following buffer overflow attacks are possible on x86-64 in Linux? (check all that apply)

- Forcing the program to execute an arbitrary function in the binary.
- Executing new code inserted on the stack.
- Preventing the operating system from running.
- Reading another process' memory.

(d) How large are x86-64 instructions (in bytes)?

(e) When would *write-non-allocate* be a good choice for a memory access? (circle one)

- A. When initializing large data structures.
- B. When data fits in the L1 cache.
- C. When writing a location after reading it.
- D. When writing a location without reading it.
- E. Never.

(f) What does the translation lookaside buffer hold? (circle one)

- A. Physical page offsets.
- B. Physical to virtual page mappings.
- C. Virtual to physical page mappings.
- D. Recently accessed cache lines.
- E. None of the above.

Name: _____

(g) Java arrays can hold objects of different sizes. How do they accomplish this? Answer in one sentence.

(h) In a first-fit, explicit free list allocator, coalescing is important because it: *(check all that are true)*

- Reduces fragmentation.
- Increases memory utilization.
- Reduces search time when allocating.
- Prevents double-freeing.

(i) Garbage collectors work by: *(circle one)*

- A. Compiler checking that all allocations are matched with a corresponding free.
- B. Cleaning up objects that have not been accessed in a given amount of time.
- C. Freeing variables implicitly when the function returns.
- D. Iterating over all objects and freeing any that are no longer reachable.

(j) Which of the following is dictated by the x86-64 instruction set architecture?

- Size of registers.
- Size of physical addresses.
- Which register to use for return values.
- Which register holds the address of the instruction about to be executed.
- How many levels of cache there are.

(k) The JVM and x86 are both forms of "instruction set architectures". For each of the following, label if it is true for **JVM**, **x86**, or **Both**:

- _____ Values are *typed* (e.g. you can tell if it is signed or unsigned)
- _____ Instructions are encoded in a compact binary representation.
- _____ Source language must be compiled down to lower-level instructions.
- _____ Has a fixed number of named registers.
- _____ Execution involves repeatedly interpreting and running the next instruction.
- _____ Programs are portable among multiple different implementations of the ISA.
- _____ Has special instructions for getting fields of objects or structs.

References

Powers of 2

$2^0 = 1$	$2^{-1} = 0.5$
$2^2 = 4$	$2^{-2} = 0.25$
$2^3 = 8$	$2^{-3} = 0.125$
$2^4 = 16$	$2^{-4} = 0.0625$
$2^6 = 64$	$2^{-5} = 0.03125$
$2^8 = 256$	
$2^{10} = 1024$	

Hex Conversions

Hex	Binary	Decimal
0xA	1010	10
0xA		
0xF	1111	15
0xF		
0x10	0001 0000	16
0x20	0010 0000	32

Sizes

C type	x86-64 suffix	Size (bytes)
char	b	1
short	w	2
int	l	4
long	q	8

Assembly Instructions

mov a,b	Copy from a to b
movs a,b	Copy from a to b with sign extension.
movz a,b	Copy from a to b with zero extension.
lea a,b	Compute address and store in b. <i>Note: the scaling parameter of memory operands can only be 1, 2, 4, or 8.</i>
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 a to b and store in b (and sets flags)
imul a,b	Multiply a by 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 <i>right (arithmetic)</i> by a bits, store in b (and sets flags)
shr a,b	Shift value of b <i>right (logical)</i> by a bits, store in b (and sets flags)
shl a,b	Shift value of b <i>left</i> 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 a and b and set condition codes based on result.
jmp <label>	Jump to address
j_ <label>	Conditional jump based on condition codes (<i>more on next page</i>)
set_ a	Set byte based on condition codes.

Conditionals

	cmp b,a	test a,b
je	"Equal"	a == b a & b == 0
jne	"Not equal"	a != b a & b != 0
js	"Sign" (negative)	a & b < 0
jns	(non-negative)	a & b >= 0
jpg	"Greater"	a > b a & b > 0
jge	"Greater or equal"	a >= b a & b >= 0
jle	"Less"	a < b a & b < 0
jle	"Less or equal"	a <= b a & b <= 0
ja	"Above" (unsigned >)	a > b
jb	"Below" (unsigned <)	a < b

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 process by duplicating calling process.

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.

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