University of Washington – Computer Science & Engineering

Autumn 2016 Instructor: Justin Hsia 2016-12-13

CSE351 FINAL

Last Name:	**Perfect**
First Name:	**Perry**
Student ID Number:	1234567
Section you attend (circle):	Chris Yufang John Kevin Sachin Suraj Waylon Thomas Xi
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. (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	**M1a**	**M1b**	**M2**	**M3**	**M4**	**F5**	**F6**	**F7**	**F8**	**F9**	**Total**
Possible Points	3	4	8	12	8	10	9	10	9	5	**78**

Question M1a: Floating Point [3 pts]

(A) What is the decimal value of the `float` **`0xFF800000`**? [1 pt]

-∞

Sign bit is 1 and the Exponent field is all 1's and the Mantissa field is all 0's, so this is the special case where the value is $-\infty$.

(B) We are storing scientific data on the order of $\mathbf{2^{-10}}$ using 32-bit `float`s. What is the *minimum number* of these data points, when multiplied together (e.g. `a*b*c` is 3), that cause **underflow** numerical issues? [2 pt]

15

The smallest denormalized number is given by the encoding of all 0's with a 1 in the least significant bit (i.e. the Mantissa is 22 zeros followed by a one). This has the value of $2^{-23} \times 2^{-126} = 2^{-149}$. We then need to multiply 2^{-10} fifteen times in a row before we hit 2^{-150}.

Question M1b: Number Representation [4 pts]

DNA is comprised of four nucleotides (<u>A</u>, <u>C</u>, <u>G</u>, <u>T</u> – the building blocks of life!). We can convert data into DNA nucleotide representation using the encoding 00_2↔<u>A</u>, 01_2↔<u>C</u>, 10_2↔<u>G</u>, 11_2↔<u>T</u>. For example, 0x0 = 0000_2 = <u>AA</u>.

(C) What is the *unsigned* decimal value of the DNA encoding **<u>TAG</u>**? [2 pt]

50

<u>TAG</u> = $110010_2 = 2^5 + 2^4 + 2^1$ = 32+16+2 = 50.

(D) If we have 256 bytes of binary data that we want to store, how many *nucleotides* would it take to store that same data? [2 pt]

1024 nucleotides

256 bytes is 2048 bits. As we can see from the encoding, one nucleotide is equivalent to 2 bits, so we need 2048/2 = 1024 nucleotides (base 4).

Question M2: Pointers & Memory [8 pts]

For this problem we are using a 64-bit x86-64 machine (**little endian**). Below is the factorial function disassembly, *showing where the code is stored in memory.*

```
000000000040052d <fact>:
  40052d:  83 ff 00         cmpl   $0, %edi
  400530:  74 05            je     400537 <fact+0xa>
  400532:  83 ff 01         cmpl   $1, %edi
  400535:  75 07            jne    40053e <fact+0x11>
  400537:  b8 01 00 00 00   movl   $1, %eax
  40053c:  eb 0d            jmp    40054b <fact+0x1e>
  40053e:  57               pushq  %rdi
  40053f:  83 ef 01         subl   $1, %edi
  400542:  e8 e6 ff ff ff   call   40052d <fact>
  400547:  5f               popq   %rdi
  400548:  0f af c7         imull  %edi, %eax
  40054b:  f3 c3            rep ret
```

(A) 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. [4 pt]

```
leal (%rdi, %rsi), %eax
movb 3(%rdi,%rsi,2), %bl
```

Register	Value (hex)
%rdi	0x0000 0000 0040 052D
%rsi	0x0000 0000 0000 0003
%eax	**0x0040 0530**
%bl	**0x07**

The `leal` stores the *address* calculated by adding the contents of `%rdi` and `%rsi` together.

The address calculation for `movb` equates to 0x40052D + 9. Nine bytes past the start of `fact` is the byte 0x07.

(B) Complete the C code below to fulfill the behaviors described in the inline comments using pointer arithmetic. Let **`char* cp = 0x40052D`**. [4 pt]

```
char v1 = *(cp + __8__);                   // set v1 = 0x75

int* v2 = (int*)((long/double*)cp + 2);    // set v2 = 0x40053D
```

The only 0x75 byte in `fact` is found at address 0x400535, 8 bytes beyond `cp`.

The difference between `v2` and `cp` is 16 bytes. Since by pointer arithmetic we are moving 2 "things" away, cp must be cast to a data type of size 8 bytes.

Question M3: The Stack [12 pts]

The recursive Fibonacci sequence function `fib()` and its x86-64 disassembly are shown below:

```
int fib (int n) {
   if (n<2)
      return 1;
   else
      return fib(n-2) + fib(n-1);
}
```

```
000000000040055d <fib>:
  40055d:  55                push   %rbp
  40055e:  53                push   %rbx
  40055f:  89 fb             mov    %edi,%ebx
  400561:  83 ff 01          cmp    $0x1,%edi
  400564:  7e 16             jle    40057c <fib+0x1f>
  400566:  8d 7f fe          lea    -0x2(%rdi),%edi
  400569:  e8 ef ff ff ff    callq  40055d <fib>
  40056e:  89 c5             mov    %eax,%ebp
  400570:  8d 7b ff          lea    -0x1(%rbx),%edi
  400573:  e8 e5 ff ff ff    callq  40055d <fib>
  400578:  01 e8             add    %ebp,%eax
  40057a:  eb 05             jmp    400581 <fib+0x24>
  40057c:  b8 01 00 00 00    mov    $0x1,%eax
  400581:  5b                pop    %rbx
  400582:  5d                pop    %rbp
  400583:  c3                retq
```

(A) In no more than a sentence, explain what the instruction at address `0x40055f` does (in terms of the function – don't be too literal) and why it is necessary. [2 pt]

It is saving the current value of *n* into a callee-saved register (`%rbx`) so that it doesn't get overwritten by the first recursive call.

«Problem continued on next page»

(B) How much space (**in bytes**) does this function take up in our final executable? [1 pt]

Count all bytes (middle columns) or subtract 0x40055d from the address of the next instruction after `fib` (0x400584).

39 B

(C) Calling `fib(4)`: How many **total** `fib` *stack frames* are created? [2 pt]

9

```
fib(4)   → fib(2)    → fib(0)
                     → fib(1)
         → fib(3)    → fib(1)
                     → fib(2)    → fib(0)
                                 → fib(1)
```

(D) Calling `fib(4)`: What is the *maximum* amount of memory on the stack (**in bytes**) used for `fib` stack frames at any given time? [3 pt]

88 or 96 bytes

The maximum depth is 4 stack frames. From the assembly code, we know that `%rbp` and `%rbx` get pushed onto the stack *every* time `fib` is called. The return address to `fib` is also pushed whenever we make a recursive call, so that makes 3 words for the first 3 levels and only 2 words for the 4th level – a total of 11 words = 88 bytes. (96 bytes if counting return address as part of Callee's stack frame – which is valid according to the x86-64 Application Binary Interface)

(E) Below is an incomplete snapshot of the stack during the call to `fib(4)`. Fill in the values of the four missing intermediate words in hex: [4 pt]

Remember that we push the *old* values of `%rbp` and `%rbx` onto the stack. `%rbx` holds the old value of n (i.e. the node above this one in the "tree"). `%rbp` is used to hold the return value of the first recursive call to `fib`.

The lowest part of the shown stack is part of the stack frame for the `fib(2)` call at "depth 3". As long as you know that you're in one of the recursive calls of `fib(3)` – given by `%rbx` at `0x7fffc39b72a8` – then the stack frame above is for `fib(3)`.

0x7fffc39b72e8	<ret addr to main>
0x7fffc39b72e0	<original rbp>
0x7fffc39b72d8	<original rbx>
0x7fffc39b72d0	**0x400578**
0x7fffc39b72c8	**0x2**
0x7fffc39b72c0	**0x4**
0x7fffc39b72b8	**0x400578**
0x7fffc39b72b0	0x1
0x7fffc39b72a8	0x3

In `fib(3)`'s stack frame, it will have stored 0x2 (the return value from the first recursive call of `fib(4)`) and 0x4 (n of the function that called `fib(3)`).

`fib(2)` is the second recursive call of `fib(3)` and `fib(3)` is the second recursive call of `fib(4)`, so the return address 0x400578 (not 0x40056e) is pushed onto the stack in both cases.

Question M4: C & Assembly [8 pts]

We are writing the *recursive* function `search`, which takes a `char` pointer and returns the *address* of the first instance in the string of a specified `char c`, or the null pointer if not found.
Example: `char* p = “TeST oNe”`, then `search(p,‘N’)` will return the address `p+6`.

```
char *search (char *p, char c) {
   if (!*p)
      return 0;
   else if (*p==c)
      return p;
   return search(p+1,c);
}
```

Fill in the blanks in the x86-64 code below with the correct instructions and operands. *Remember to use the proper size suffixes and correctly-sized register names!*

```
    search(char*, char):

 1      movzbl  (%rdi), %eax      # get *p

 2      testb   %al,    %al       # conditional

 3      je      .NotFound         # conditional jump

 4      cmpb    %sil,   %al       # conditional

 5      je      .Found            # conditional jump

 6      addq    $1, %rdi          # argument setup

 7      call    search            # recurse
 8      ret
    .NotFound:

 9      movl    $0, %eax          # return value
10      ret
    .Found:

11      movq    %rdi,   %rax    # return value
12      ret
```

Grading Notes:

Line 2: `cmpb $0, %al` also accepted.

Line 6: Given that argument `p` is a pointer, needed to use the full add**q** and `%rdi`.

Line 7: `callq` also accepted

Question F5: Caching [10 pts]

We have 16 KiB of RAM and two options for our cache. Both are two-way set associative with 256 B blocks, LRU replacement, and write-back policies. **Cache A** is size 1 KiB and **Cache B** is size 2 KiB.

(A) Calculate the TIO address breakdown for **Cache B**: [1.5 pt]

Tag bits	Index bits	Offset bits
4	2	8

14 address bits. $\log_2 256 = 8$ offset bits. 2 KiB cache = 8 blocks. 2 blocks/set → 4 sets.

(B) The code snippet below accesses an integer array. Calculate the **Miss Rate** for **Cache A** if it starts *cold*. [3 pt]

```
#define LEAP 4
#define ARRAY_SIZE 512
int nums[ARRAY_SIZE];           // &nums = 0x0100 (physical addr)
for (i = 0; i < ARRAY_SIZE; i+=LEAP)
    nums[i] = i*i;
```

1/16

Access pattern is a single write to `nums[i]`. Stride = LEAP = 4 ints = 16 bytes. 256/16 = 16 strides per block. First access is a compulsory miss and the next 15 are hits. Since we never revisit indices, this pattern continues for all cache blocks. You can also verify that the offset of &nums is 0x00, so we start at the beginning of a cache block.

(C) For each of the proposed (independent) changes, write **MM** for "higher miss rate", **NC** for "no change", or **MH** for "higher hit rate" to indicate the effect on **Cache A** for the code above:[3.5 pt]

Direct-mapped _NC_ Increase block size _MH_

Double LEAP _MM_ Write-through policy _NC_

Since we never revisit blocks, associativity doesn't matter. Larger block size means more strides/block. Doubling LEAP means fewer strides/block. Write hit policy has no effect.

(D) Assume it takes 200 ns to get a block of data from main memory. Assume **Cache A** has a hit time of 4 ns and a miss rate of 4% while **Cache B**, being larger, has a hit time of 6 ns. What is the worst miss rate Cache B can have in order to perform as well as Cache A? [2 pt]

0.03 or 3%

$\text{AMAT}_\text{A} = \text{HT}_\text{A} + \text{MR}_\text{A} \times \text{MP} = 4 + 0.04*200 = 12$ ns.

$\text{AMAT}_\text{B} = \text{HT}_\text{B} + \text{MR}_\text{B} \times \text{MP} \leq 12 \;\rightarrow\; 200\ \text{MR}_\text{B} \leq 6 \;\rightarrow\; \text{MR}_\text{B} \leq 0.03$

Question F6: Processes [9 pts]

(A) In keeping with the explosive theme of this class, please complete the function below to create a **fork bomb**, which continually creates new processes. [2 pt]

```
void forkbomb(void) {

   while(1) {
      fork();
   }

}
```

← Write within the text box

(B) Why is a fork bomb bad? Briefly explain what will happen to your system when it goes off. [2 pt]

Resource starvation from new processes that results in eventual grinding to a halt and/or a system crash. We were looking for any answer along the lines of: (1) eats into memory because of page tables and/or duplicated virtual address space or (2) eats up CPU time with context switching.

(C) Name the three possible *control flow outcomes* (i.e. what happens next?) of an exception. [3 pt]

1) Abort

2) Restart the current instruction

3) Continue at the next instruction

(D) In the following blanks, write "**Y**" for yes or "**N**" for no if the following need to be updated *during* a **context switch**. [2 pt]

Page table __**N**__ PTBR __**Y**__ TLB __**Y**__ Cache __**N**__

All of the page tables live in physical memory and continue to do so during a context switch.

The page table base register points to the current processes' page table, so gets updated.

The TLB uses stores the VPN→PPN mappings for the old process, and thus needs to be flushed.

The cache is accessed using physical addresses, which aren't altered during a context switch.

Question F7: Virtual Memory [10 pts]

Our system has the following setup:

- 24-bit virtual addresses and 512 KiB of RAM with 4 KiB pages
- A 4-entry TLB that is fully associative with LRU replacement
- A page table entry contains a valid bit and protection bits for read (R), write (W), execute (X)

(A) Compute the following values: [2 pt]

Page offset width __**12**__ PPN width __**7**__
Entries in a page table __$\mathbf{2^{12}}$__ TLBT width __**12**__

Because TLB is fully associative, TLBT width matches VPN. There are $2^{\text{VPN width}}$ entries in PT.

(B) Briefly explain why we make the page size so much larger than a cache block size. [2 pt]

Take advantage of spatial locality and try to avoid page faults as much as possible. Disk access is also super slow, so we want to pull a lot of data when we do access it.

(C) Fill in the following blanks with "**A**" for always, "**S**" for sometimes, and "**N**" for never if the following get updated during a **page fault**. [2 pt]

Page table __**A**__ Swap space __**S**__ TLB _**A/N**_ Cache __**S**__

When the page is place in physical memory, the new PPN is written into the **page table** entry.

Swap space will get updated if a dirty page is kicked out of physical memory.

For this class, we say that the page fault handler updates the **TLB** because it is more efficient.

In reality not all do (OS does not have access to hardware-only TLB; instr gets restarted).

To update a PTE (in physical mem), you check the **cache**, so it gets updated on a cache miss.

(D) The TLB is in the state shown when the following code is executed. Which iteration (value of `i`) will cause the **protection fault (segfault)**? Assume `sum` is stored in a register.
Recall: the hex representations for TLBT/PPN are padded as necessary. [4 pt]

```
long *p = 0x7F0000, sum = 0;
for (int i = 0; 1; i++) {
   if (i%2)
      *p = 0;
   else
      sum += *p;
   p++;
}
```

TLBT	PPN	Valid	R	W	X
0x7F0	0x31	1	1	1	0
0x7F2	0x15	1	1	0	0
0x004	0x1D	1	1	0	1
0x7F1	0x2D	1	1	0	0

`i` = **513**

Only the current page (VPN = TLBT = 0x7F0) has write access. Once we hit the next page (TLBT = 0x7F1), we will encounter a segfault once we try to *write* to the page. We are using pointer arithmetic to increment our pointer by 8 bytes at a time. One page holds $2^{12}/2^{3} = 512$ `long`s, so we first access TLBT 0x7F1 when `i` = 512. However, the code is set up so that we only write on *odd* values of `i`, so the answer is `i` = 513.

Question F8: Memory Allocation [9 pts]

(A) Briefly describe one drawback and one benefit to using an *implicit* free list over an *explicit* free list. [4 pt]

Implicit drawback:	Implicit benefit:
• Slower – have to check both allocated and free blocks • Must use both boundary tags in every block – less room for payload	• Simpler code; easier to manage • Smaller minimum block size (less internal fragmentation for free blocks)

(B) The table shown to the right shows the *value of the header* for the block returned by the request: **`(int*)malloc(N*sizeof(int))`** What is the alignment size for this dynamic memory allocator? [2 pt]

N	**header value**
6	33
8	49
10	49
12	65

16 bytes

The alignment size is given by the difference in size once we cross an alignment boundary. Remembering to mask out the allocated tag, we see that 6 `int`s = 24 bytes gets rounded up to 32 and 8 `int`s = 32 bytes gets rounded up to 48 (remember extra space for internal fragmentation – at least the header, possibly other things).

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

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";

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

```
ZERO    __<__    &ZERO

foo     __<__    &foo

foo     __>__    &str
```

`ZERO` and `str` are global variables, so their *addresses* are in the Static Data section of memory. `str`'s *value* is the address of a string literal, which sits at the bottom portion of Static Data. `foo` is a local variable, so its *address* is in the Stack, but its *value* is the address of a block in the Heap.
The virtual address space is arranged such that 0 < Instructions < Static Data < Heap < Stack.

Question F9: C and Java [5 pts]

For this question, use the following Java object definition and C struct definition. Assume addresses are all 64-bits.

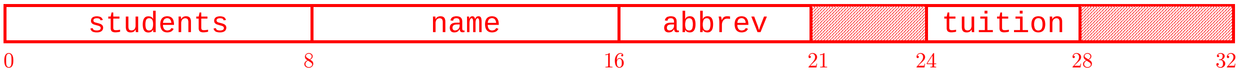
```
public class School {
   long students;
   String name;
   String abbrev;
   float tuition;

   public void cheer() {
      System.out.println("Go "+name);
   }
}
public class Univ extends School {
   String[] majors;
   public void cheer() {
      System.out.println("Go "+abbrev);
   }
}
```

```
struct School {       K:
   long students;     8
   char* name;        8
   char abbrev[5];    1
   float tuition;     4
};              Kmax = 8
```

(A) How much memory, in bytes, does an instance of `struct School` use? How many of those bytes are *internal* fragmentation and *external* fragmentation? [3 pt]

sizeof(struct School)	Internal	External
32 bytes	3	4

Alignment requirements listed above in red, next to the struct fields. A `struct School` instance will look as shown below:



The 3 bytes between abbrev and tuition count as internal fragmentation.
The 4 bytes at the end count as external fragmentation.

(B) How much *longer*, in bytes, are the following for `Univ` than for `School`? [2 pt]

Instance:	8 bytes
vtable:	0 bytes

`Univ` extends `School` by adding a field and overriding a method, so the length of that field (8 bytes for a reference) is added to the object instance length, but the vtable remains the same length.

This page purposely left blank