University of Washington – Computer Science & Engineering
Autumn 2016        Instructor: Justin Hsia        2016-12-13

CSE351 FINAL

Last Name: 
First Name: 
Student ID Number: 
Section you attend (circle):
Chris  Yufang  John  Kevin  Sachin  Suraj  Waylon  Thomas  Xi

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. (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.

<table>
<thead>
<tr>
<th>Question</th>
<th>M1a</th>
<th>M1b</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possible Points</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>5</td>
<td>78</td>
</tr>
</tbody>
</table>
Question M1a: Floating Point  [3 pts]

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

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

Question M1b: Number Representation  [4 pts]

DNA is comprised of four nucleotides (A, C, G, T – the building blocks of life!). We can convert data into DNA nucleotide representation using the encoding 00$2^0$ ↔ A, 01$2^1$ ↔ C, 10$2^2$ ↔ G, 11$2^3$ ↔ T. For example, 0x0 = 0000$2^0$ = AA.

(C) What is the unsigned decimal value of the DNA encoding TAG? [2 pt]

(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]
**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.*

(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]

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x0000 0000 0040 052D</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x0000 0000 0000 0003</td>
</tr>
</tbody>
</table>

leal (%rdi, %rsi), %eax  
\%

movb 3(%rdi,%rsi,2), %bl

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

```c
char v1 = *(cp + _____);     // set v1 = 0x75
int* v2 = (int*)((__________*)cp + 2); // set v2 = 0x40053D
```
Question M3: The Stack  [12pts]

The recursive Fibonacci sequence function \texttt{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
```

\[\text{(A)}\] 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]
(B) How much space (in bytes) does this function take up in our final executable? [1 pt]

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

(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]

(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]

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffc39b72e8</td>
<td>&lt;ret addr to main&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72e0</td>
<td>&lt;original rbp&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72d8</td>
<td>&lt;original rbx&gt;</td>
</tr>
<tr>
<td>0x7fffc39b72d0</td>
<td></td>
</tr>
<tr>
<td>0x7fffc39b72c8</td>
<td></td>
</tr>
<tr>
<td>0x7fffc39b72c0</td>
<td></td>
</tr>
<tr>
<td>0x7fffc39b72b8</td>
<td></td>
</tr>
<tr>
<td>0x7fffc39b72b0</td>
<td>0x1</td>
</tr>
<tr>
<td>0x7fffc39b72a8</td>
<td>0x3</td>
</tr>
</tbody>
</table>
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`.

```c
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!

```assembly
search(char*, char):
    1    movzbl _____, %eax     # get *p
    2    _____ _____,  %al      # conditional
    3    _____ .NotFound      # conditional jump
    4    _____ _____,  %al      # conditional
    5    _____ _______        # conditional jump
    6    _____ $1, ______      # argument setup
    7    _____ _______        # recurse
    8    ret
     .NotFound:
    9    _____ $0, %eax        # return value
   10    ret
     .Found:
   11    movq _____, _____     # return value
   12    ret
```
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]

<table>
<thead>
<tr>
<th>Tag bits</th>
<th>Index bits</th>
<th>Offset bits</th>
</tr>
</thead>
</table>

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

```c
#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;
```

(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 ____
- Increase block size ____
- Double LEAP ____
- Write-through policy ____

(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]
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]

```c
void forkbomb(void) {
  ← 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]

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

1)  
2)  
3)  

(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  _____  
PTBR  _____  
TLB  _____  
Cache  _____  

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]

<table>
<thead>
<tr>
<th>Page offset width</th>
<th>PPN width</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entries in a page table   TLBT width
_______                ______

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

(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]

<table>
<thead>
<tr>
<th>Page table</th>
<th>Swap space</th>
<th>TLB</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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]

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

```
<table>
<thead>
<tr>
<th>TLBT</th>
<th>PPN</th>
<th>Valid</th>
<th>R</th>
<th>W</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7F0</td>
<td>0x31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x7F2</td>
<td>0x15</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x00A</td>
<td>0x1D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x7F1</td>
<td>0x2D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

{ }
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]

<table>
<thead>
<tr>
<th>Implicit drawback:</th>
<th>Implicit benefit:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>N</th>
<th>header value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>49</td>
</tr>
<tr>
<td>10</td>
<td>49</td>
</tr>
<tr>
<td>12</td>
<td>65</td>
</tr>
</tbody>
</table>

(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;
}
```
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.

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

```c
struct School {
    long students;
    char* name;
    char abbrev[5];
    float tuition;
};
```

(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]

<table>
<thead>
<tr>
<th>sizeof(struct School)</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Instance: 

vtable:
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.
  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).
- `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.

### IEEE 754 Floating-Point Standard

Value: \( \pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}} \)

Bit Fields: \((-1)^{E} \times 1.M \times 2^{(E-bias)}\)
where Single Precision Bias = -127,
Double Precision Bias = -1023.

### IEEE 754 Symbols

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Fraction</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>±0</td>
</tr>
<tr>
<td>0</td>
<td>≠0</td>
<td>±Denorm</td>
</tr>
<tr>
<td>1 to MAX - 1</td>
<td>anything ± Fl. Pt. Num.</td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>0</td>
<td>±∞</td>
</tr>
<tr>
<td>MAX</td>
<td>≠0</td>
<td>NaN</td>
</tr>
</tbody>
</table>

### IEEE Single Precision and Double Precision Formats:

- 1 bit
- 8 bits
- 23 bits
- 52 bits
### Conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>cmp b, a</th>
<th>test a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>a == b</td>
<td>a &amp; b == 0</td>
</tr>
<tr>
<td>jne</td>
<td>a != b</td>
<td>a &amp; b != 0</td>
</tr>
<tr>
<td>js</td>
<td>a &amp; b &lt; 0</td>
<td></td>
</tr>
<tr>
<td>jns</td>
<td>a &amp; b &gt;= 0</td>
<td></td>
</tr>
<tr>
<td>jg</td>
<td>a &gt; b</td>
<td>a &amp; b &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>a &gt;= b</td>
<td>a &amp; b &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>a &lt; b</td>
<td>a &amp; b &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>a &lt;= b</td>
<td>a &amp; b &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>a &gt; b</td>
<td></td>
</tr>
<tr>
<td>jb</td>
<td>a &lt; b</td>
<td></td>
</tr>
</tbody>
</table>

### Sizes

<table>
<thead>
<tr>
<th>C type</th>
<th>x86-64 suffix</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>q</td>
<td>8</td>
</tr>
</tbody>
</table>

### Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Convention</th>
<th>Name of “virtual” register</th>
<th>Lowest 4 bytes</th>
<th>Lowest 2 bytes</th>
<th>Lowest byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value – Caller saved</td>
<td>%eax %ax %al</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
<td>%ebx %bx %bl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4 – Caller saved</td>
<td>%ecx %cx %cl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3 – Caller saved</td>
<td>%edx %dx %dl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2 – Caller saved</td>
<td>%esi %si %sil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1 – Caller saved</td>
<td>%edi %di %dil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack Pointer</td>
<td>%esp %sp %spl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
<td>%ebp %bp %bpl</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5 – Caller saved</td>
<td>%r8d %r8w %r8b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6 – Caller saved</td>
<td>%r9d %r9w %r9b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
<td>%r10d %r10w %r10b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>Caller saved</td>
<td>%r11d %r11w %r11b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
<td>%r12d %r12w %r12b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
<td>%r13d %r13w %r13b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
<td>%r14d %r14w %r14b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
<td>%r15d %r15w %r15b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C Functions

```c
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.
```