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.

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]

(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]

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↔A, 01_2↔C, 10_2↔G, 11_2↔T. For example, 0x0 = 0000_2 = 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.*

```
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	
%bl	

(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 + _____);                      // set v1 = 0x75

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

Question M3: The Stack [12 pts]

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

```c
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]

«Problem continued on next page»

(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]

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

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  ______, %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]

Tag bits	Index bits	Offset bits

(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;
```

(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 ______

SID: ___________

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 ______ PPN width ______

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]

Page table ______ Swap space ______ TLB ______ Cache ______

(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` =

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:

(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

(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`

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 {
   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]

sizeof(struct School)	Internal	External

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

Instance:	
vtable:	

This page purposely left blank

CSE 351 Reference Sheet (Final)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

IEEE 754 FLOATING-POINT STANDARD

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

Bit Fields: $(-1)^S \times 1.M \times 2^{(E+\text{bias})}$

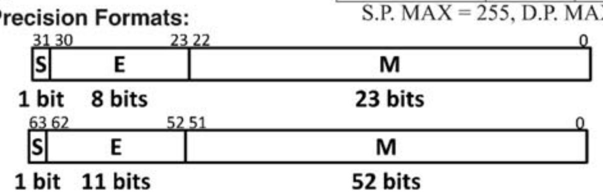
where Single Precision Bias = -127,
Double Precision Bias = -1023.

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:



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.

Conditionals

Instruction		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
jg	“Greater”	a > b	a & b > 0
jge	“Greater or equal”	a >= b	a & b >= 0
jl	“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	

Sizes

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

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

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 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.