Full Name:____

Roll Number:______________________________________

IT215: Systems Software, Winter 2013-14

First In-Sem Exam (2 hours)

February 6, 2014

Instructions:

- Make sure your exam is not missing any sheets, then write your name and roll number on the top of this page.
- Clearly write your answer in the space indicated. None of the questions need long answers.
- For rough work, do not use any additional sheets. Rough work will not be graded.
- Assume IA32 machine running Linux.
- The exam has a maximum score of 50 points. It is CLOSED BOOK. Notes are NOT allowed.
- Anyone who copies or allows someone to copy will receive F grade.

  Good luck!

Problem 1 (/24):
Problem 2 (/4):
Problem 3 (/4):
Problem 4 (/4):
Problem 5 (/4):
Problem 6 (/5):
Problem 7 (/5):
TOTAL (/50):

Problem 1. (24 points):

Circle the *single best* answer to each of the following questions. You will get -1 points for a wrong answer, so don't just guess wildly.

1. If `A` is the smallest byte address used by a 4-byte integer `0x12345678`, then the memory byte at address `A` is

   (a) `0x12`

   (b) `0x34`

   (c) `0x56`

   (d) `0x78`

   (e) Can't tell

2. If writing to `buf[16]` (where `buf` is a stack-allocated `char` array) overwrites the least-significant byte of the return address of the current function, then the base address of `buf` is

   (a) `%ebp-4`

   (b) `%ebp-8`

   (c) `%ebp-12`

   (d) `%ebp-16`

   (e) none of the above

3. Suppose you are debugging code that includes the declaration given below.

```
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```

   Assume values of type `short` are allocated 2 bytes and of type `float` are allocated 4 bytes. Also assume the Linux convention for data alignment discussed in class. Which of the following is true of the memory byte with address `a+38`?

   (a) It is beyond the end of array `a`.

   (b) It is part of field `i` for one of the structs in the array.

   (c) It is part of field `v` for one of the structs in the array.

   (d) It is part of field `j` for one of the structs in the array.

   (e) It is part of an unused gap between fields in one of the structs in the array.

4. Which of the following is true concerning dynamic memory allocation?

   (a) External fragmentation is caused by chunks which are marked as allocated but cannot actually be used.

   (b) Internal fragmentation is caused by padding for alignment purposes and by overhead to maintain the heap data structure (such as headers and footers).

   (c) Coalescing while traversing the list during calls to malloc is known as immediate coalescing.

   (d) Garbage collection, employed by calloc, refers to the practice of zeroing memory before use.

For the next 3 questions, consider the following code. The function `calloc(size_t n, size_t size)` allocates space on the heap for an array of `n` objects of the specified size, and returns a pointer to the beginning of the block (i.e. payload). The allocated memory is initialized to zero.

```
int main()
{
   int a, *b, c;
   char **p;

   p = (char **) calloc(4, sizeof(char)); /* calloc returns 0x1dce1000 */

   a = (int) (p + 0x100);
   b = (int *) (*p + 0x200);
   c = (int) (b + 0x300);

   printf("p=0x%x a=0x%x b=0x%x c=0x%x", p, a, b, c);
}
```

Assuming the call to `calloc` succeeds, and it returns `0x1dce1000`.

5. When `printf` is called, the hex value of variable `a` is

   (a) Can't tell

   (b) `0x1dce1100`

   (c) `0x1dce1400`

   (d) `0x1dce1800`

6. When `printf` is called, the hex value of variable `b` is

   (a) Can't tell

   (b) `0x1dce1200`

   (c) `0x1dce2000`

   (d) `0x200`

   (e) `0x800`

7. When `printf` is called, the hex value of variable `c` is

   (a) Can't tell

   (b) `0x1dce2a00`

   (c) `0x1dce4400`

   (d) `0xc00`

   (e) `0xe00`

8. Which of these equations does **not** have a solution, if the operations are performed using 32-bit `unsigned ints`?

   (a) $x + 1 < x$

   (b) $2 \cdot x = 1$

   (c) $13 \cdot x = 10$

   (d) $2 \cdot x = 10$

   (e) $2 \cdot x < x$

Problem 2. (4 points):

Consider the following program.

```
// mystery.c (headers omitted)
int main()
{
   int *b = malloc(sizeof(int));
   int a = 0;
   if ((&a) < b) {
       printf("Trick!\n");
   } else {
       printf("Treat!\n");
   }
   return 0;
}
```

What does this program print out and why? (Assume that the `malloc()` call succeeds.)

Problem 3. (4 points):

Assume that a progam under execution ends up in the following state with the given stack diagram and associated register values:

```
            +--------------+
0xffff1018  |     0x00     |
            +--------------+
0xffff1014  |  0x08048301  |
            +--------------+
0xffff1010  |  0xffff1040  |
            +--------------+
0xffff100c  |    0xdead    |
            +--------------+
0xffff1008  |    0xbeef    |
            +--------------+
0xffff1004  |     0x00     |
            +--------------+
0xffff1000  |     0x00     |
            +--------------+
    %esp:    0xffff1008
    %ebp:    0xffff1010
    %eip:    0x080483c2
```

The following state represents the result of executing *exactly one* additional instruciton from the above state. Fill in the mystery instruction. If the instruction requires operands, supply them. For your convenience, fields that differ between the original state and the new state are labeled with an asterisk: (*)

```
      Original state                  New State
             +-------------+        +-------------+
0xffff1018   |     0x00    |        |    0x00     |
             +-------------+        +-------------+
0xffff1014   |  0x08048301 |        | 0x08048301  |
             +-------------+        +-------------+
0xffff1010   |  0xffff1040 |        | 0xffff1040  |
             +-------------+  ----> +-------------+
0xffff100c   |    0xdead   |        |  0xdead     |
             +-------------+        +-------------+
0xffff1008   |    0xbeef   |        |  0xbeef     |
             +-------------+        +-------------+
0xffff1004   |     0x00    |        | 0x080483c7  |  (*)
             +-------------+        +-------------+
0xffff1000   |     0x00    |        |   0x00      |
             +-------------+        +-------------+
    %esp:     0xffff1008        %esp:  0xffff1004 (*)
    %ebp:     0xffff1010        %ebp:  0xffff1010
    %eip:     0x080483c2        %eip:  0x08048c94 (*)
```

MYSTERY INSTRUCTION: ___________________

Problem 4. (4 points):

The assembly code for a C function is given below.

```
foo3:
    pushl %ebp
    movl  %esp, %ebp
    movl 12(%ebp), %ecx
    movl 16(%ebp), %edx
    leal (%edx,%ecx,), %eax
    imull %ecx, %edx
    imull 8(%ebp), %eax
    popl %ebp
    addl %edx, %eax
    ret
```

Write the corresponding C function.

Problem 5. (4 points):

Consider the following program, which consists of two modules:

```
/* Module foo6.c */
void p2(void);

int main() {
   p2();
   return 0;
}
```

```
/* Module bar6.c */
#include <stdio.h>

char main;

void p2()
{
   printf("0x%x\n", main);
}
```

When this program is compiled and executed, it prints the string `"0x55\n"` and terminates normally, even though `p2` never initializes variable `main`. Can you explain this?

Problem 6. (5 points):

Consider a memory allocator that uses an implicit free list. The layout of each allocated and free memory block is as follows:

```
             31                            2 1 0
             ______________________________________
    Header   |          Block Size (bytes)    |     |
             |________________________________|_____|
             |                                      |
             |                                      |
             |______________________________________|
    Footer   |          Block Size (bytes)    |     |
             |________________________________|_____|
```

Each memory block, either allocated or free, has a size that is a multiple of 8 bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer. The usage of the remaining 3 lower order bits is as follows:

- bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- bit 2 is unused and is always set to be 0.

Note: The header and footer will always be present regardless of whether the block is allocated or not. Given the contents of the heap shown on the left side, show the new contents of the heap (in the right table) after a call to `free(0x400b010)` is executed. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing. Also assume that any blocks not shown are allocated.

```
Address       Content                          Address       Content
            +------------+                                 +-----------+
0x400b028   | 0x00000012 |                     0x400b028   |           |
            +------------+                                 +-----------+
0x400b024   | 0x400b611c |                     0x400b024   |           |
            +------------+                                 +-----------+
0x400b020   | 0x400b512c |                     0x400b020   |           |
            +------------+                                 +-----------+
0x400b01c   | 0x00000012 |                     0x400b01c   |           |
            +------------+                                 +-----------+
0x400b018   | 0x00000013 |                     0x400b018   |           |
            +------------+                                 +-----------+
0x400b014   | 0x400b511c |                     0x400b014   |           |
            +------------+                                 +-----------+
0x400b010   | 0x400b601c |                     0x400b010   |           |
            +------------+                                 +-----------+
0x400b00c   | 0x00000013 |                     0x400b00c   |           |
            +------------+                                 +-----------+
0x400b008   | 0x00000013 |                     0x400b008   |           |
            +------------+                                 +-----------+
0x400b004   | 0x400b601c |                     0x400b004   |           |
            +------------+                                 +-----------+
0x400b000   | 0x400b511c |                     0x400b000   |           |
            +------------+                                 +-----------+
0x400affc   | 0x00000013 |                     0x400affc   |           |
            +------------+                                 +-----------+
```

The following problem concerns the following, low-quality code:

```
void foo(int x)
{
  int a[3];
  char buf[4];
  a[0] = 0xF0F1F2F3;
  a[1] = x;
  gets(buf);
  printf("a[0] = 0x%x, a[1] = 0x%x, buf = %s\n", a[0], a[1], buf);
}
```

In a program containing this code, procedure `foo` has the following disassembled form on an IA32 machine:

```
080485d0 <foo>:
80485d0: 55                 pushl  %ebp
80485d1: 89 e5              movl   %esp,%ebp
80485d3: 83 ec 10           subl   $0x10,%esp
80485d6: 53                 pushl  %ebx
80485d7: 8b 45 08           movl   0x8(%ebp),%eax
80485da: c7 45 f4 f3 f2     movl   $0xf0f1f2f3,0xfffffff4(%ebp)
80485df: f1 f0
80485e1: 89 45 f8           movl   %eax,0xfffffff8(%ebp)
80485e4: 8d 5d f0           leal   0xfffffff0(%ebp),%ebx
80485e7: 53                 pushl  %ebx
80485e8: e8 b7 fe ff ff     call   80484a4 <_init+0x54>  # gets
80485ed: 53                 pushl  %ebx
80485ee: 8b 45 f8           movl   0xfffffff8(%ebp),%eax
80485f1: 50                 pushl  %eax
80485f2: 8b 45 f4           movl   0xfffffff4(%ebp),%eax
80485f5: 50                 pushl  %eax
80485f6: 68 ec 90 04 08     pushl  $0x80490ec
80485fb: e8 94 fe ff ff     call   8048494 <_init+0x44>  # printf
8048600: 8b 5d ec           movl   0xffffffec(%ebp),%ebx
8048603: 89 ec              movl   %ebp,%esp
8048605: 5d                 popl   %ebp
8048606: c3                 ret
8048607: 90                 nop
```

For the following questions, recall that:

- `gets` is a standard C library routine.
- C strings are null-terminated (i.e., terminated by a character with value 0x00).
- Characters ‘`0`’ through ‘`9`’ have ASCII codes `0x30` through `0x39`.

Problem 7. (5 points):

Consider the case where procedure `foo` is called with argument `x` equal to `0xE3E2E1E0`, and we type in the string "`123456789`" in response to `gets`.

A. Fill in the following table indicating which program values are/are not corrupted by the response from `gets`, i.e., their values were altered by some action within the call to `gets`.

Program Value	Corrupted? (Y/N)
`a[0]`	
`a[1]`	
`a[2]`	
`x`	
Saved value of register `%ebp`	
Saved value of register `%ebx`	

B. What will the `printf` function print for the following:

- `a[0]` (hexadecimal): ______________________________

- `a[1]` (hexadecimal): ______________________________

(Blank page for rough work.)