CS230 System Programming
Midterm Exam
Wednesday, 22 Oct 2014

Name:

Student Number:

Problems	Points
1 (13pts)	
2 (12pts)	
3 (10pts)	
4 (10pts)	
5 (18pts)	
6 (6pts)	
7 (4pts)	
8 (7pts)	
9 (5pts)	
10 (10pts)	
11 (15pts)	
Total (110pts)	

IMPORTANT: Explain your answer briefly. Do not just write a short answer or fill the assembly code.

1. [13pts][integer data representation]
   M[A] : the memory content of address A

   A. [2pts] x86 is a little endian machine. Suppose %eax = 0x12345678 and %edx = 0x1000.

   After an instruction "movl %eax, 0(%edx)" is executed, what are the following memory contents ? (one byte for each line)

   M[0x1000] = _____
   M[0x1001] = _____
   M[0x1002] = _____
   M[0x1003] = _____

   B. [5pts] answer whether the following conditions are true or false. (Describe why they are true or show counter examples.)

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;

B-1)  x >> 3 == x/8
B-2)  x & (x-1) != 0
B-3)  (x | -x) >> 31 == -1
B-4) x+y == uy + ux
B-5) x*~y + uy*ux == -x
```

   C. [3pts] Explain why the following function is buggy, and show how it can be fixed. The function is supposed to return 0.0 when the length is 0.

```
float sum_elements(float a[], unsigned length) {
      int i;
      float result = 0;

      for (i=0; i<=length; i++)
          result += a[i];
      return result;
}
```

D. [3pts] After executing the following code, which of the variables (a-e) are equal to 0? (Assume 64-bit architecture)

(a) unsigned int a = 0xffffffff;
(b) unsigned int b = 1;
(c) unsigned int c = a + b;
(d) unsigned long d = a + b;
(e) unsigned long e = (unsigned long)a + b;

2. [12pts][floating point data representation]

A. [5pts] Assume variables x, f, and d are of type `int`, `float`, and `double`, respectively. (Neither f nor d equals +infinity, -infinity, or NaN). For each of the following expressions, either argue that it is always true or give a counterexample if it is not.

```
B-1) x == (int) (double) x
B-2) f == -(-f)
B-3) 1.0/2 == 1/2.0
B-4) d*d >= 0.0
B-5) (f+d) – f == d
```

B. [4pts] Write the rounded binary numbers for the following values. They should be rounded to nearest 1/4 (2 bits fright of binary point, and must use "round-to-even" rule.

Explain the advantage of such "round-to-even" rule, compared to round-down or round-up.

10.00011 => _____
10.00110 => _____
10.11100 => _____
10.10100 => _____

C. [3pts] Explain how a floating point compare instruction (fcmp) can be implemented for the IEEE fp format. How will it be different from the integer compare instruction (cmp)?

3. [10pts] Consider the following C function and its corresponding x86-64 assembly code:

```
int foo(int x, int i)
{
  switch(i)
  {
    case 1:
      x -= 10;
    case 2:
      x *= 8;
      break;
    case 3:
      x += 5;
    case 5:
      x /= 2;
      break;
    case 0:
      x &= 1;
    default:
      x += i;
  }
  return x;
}
```

```
00000000004004a8 <foo>:
  4004a8:       mov    %edi,%edx
  4004aa:       cmp    $0x5,%esi
  4004ad:       ja     4004d4 <foo+0x2c>
  4004af:       mov    %esi,%eax
  4004b1:       jmpq   *0x400690(,%rax,8)
  4004b8:       sub    $0xa,%edx
  4004bb:       shl    $0x3,%edx
  4004be:       jmp    4004d6 <foo+0x2e>
  4004c0:       add    $0x5,%edx
  4004c3:       mov    %edx,%eax
  4004c5:       shr    $0x1f,%eax
  4004c8:       lea    (%rdx,%rax,1),%eax
  4004cb:       mov    %eax,%edx
  4004cd:       sar    %edx
  4004cf:       jmp    4004d6 <foo+0x2e>
  4004d1:       and    $0x1,%edx
  4004d4:       add    %esi,%edx
  4004d6:       mov    %edx,%eax
  4004d8:       retq
```

Recall that the gdb command x/g $rsp will examine an 8-byte word starting at address in $rsp. Please fill in the switch jump table as printed out via the following gdb command:

>(gdb) x/6g 0x400690

0x400690: 0x_____ 0x_____

0x4006a0: 0x_____ 0x_____

0x4006b0: 0x_____ 0x_____

4. [10pts] Consider the following C code:

```
struct triple
{
    int x;
    char c;
    int y;
};

int mystery1(int x);
int mystery2(int x);
int mystery3(struct triple* t);

int main()
{
    struct triple t = {35, 'q', 10};

    int result1 = mystery1(42);
    int result2 = mystery2(19);
    int result3 = mystery3(&t);

    printf("result1 = %d\n", result1);
    printf("result2 = %d\n", result2);
    printf("result3 = %d\n", result3);

    return 0;
}
```

Using the assembly code for mystery1, mystery2, and mystery3 on the next page, fill in the proper values in this program's output:

result1 = _____
result2 = _____
result3 = _____

```
080483d0 <mystery1>:
 80483d0:       55                      push   %ebp
 80483d1:       89 e5                   mov    %esp,%ebp
 80483d3:       53                      push   %ebx
 80483d4:       8b 45 08                mov    0x8(%ebp),%eax
 80483d7:       89 c3                   mov    %eax,%ebx
 80483d9:       83 e3 01                and    $0x1,%ebx
 80483dc:       85 c0                   test   %eax,%eax
 80483de:       74 0b                   je     80483eb <mystery1+0x1b>
 80483e0:       c1 f8 01                sar    $0x1,%eax
 80483e3:       50                      push   %eax
 80483e4:       e8 e7 ff ff ff          call   80483d0 <mystery1>
 80483e9:       01 c3                   add    %eax,%ebx
 80483eb:       89 d8                   mov    %ebx,%eax
 80483ed:       8b 5d fc                mov    0xfffffffc(%ebp),%ebx
 80483f0:       c9                      leave
 80483f1:       c3                      ret

080483f4 <mystery2>:
 80483f4:       55                      push   %ebp
 80483f5:       89 e5                   mov    %esp,%ebp
 80483f7:       8b 55 08                mov    0x8(%ebp),%edx
 80483fa:       31 c0                   xor    %eax,%eax
 80483fc:       85 d2                   test   %edx,%edx
 80483fe:       7e 06                   jle    8048406 <mystery2+0x12>
 8048400:       40                      inc    %eax
 8048401:       4a                      dec    %edx
 8048402:       85 d2                   test   %edx,%edx
 8048404:       7f fa                   jg     8048400 <mystery2+0xc>
 8048406:       c9                      leave
 8048407:       c3                      ret

08048408 <mystery3>:
 8048408:       55                      push   %ebp
 8048409:       89 e5                   mov    %esp,%ebp
 804840b:       8b 45 08                mov    0x8(%ebp),%eax
 804840e:       8b 10                   mov    (%eax),%edx
 8048410:       0f af 50 08             imul   0x8(%eax),%edx
 8048414:       89 d0                   mov    %edx,%eax
 8048416:       c9                      leave
 8048417:       c3                      ret
```

5. [18pts] Throughout this question, remember that it might help you to draw a picture. It helps us see what you're thinking when we grade you, and you'll be more likely to get partial credit if your answers are wrong. Consider the following C code:

```
void foo(int a, int b, int c, int d) {
  int buf[16];
  buf[0] = a;
  buf[1] = b;
  buf[2] = c;
  buf[3] = d;
  return;
}

void bar() {
  foo(0x15213, 0x18243, 0xdeadbeef, 0xcafebabe)
}
```

When compiled with default options (32-bit), it gives the following assembly:

```
00000000 <foo>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 40                sub    $0x40,%esp

   6:   8b 45 08                mov    _____(%ebp),%eax //temp = a;
   9:   89 45 c0                mov    %eax,-0x40(%ebp) //buf[0] = temp;

   c:   8b 45 0c                mov    _____(%ebp),%eax //temp = b;
   f:   89 45 c4                mov    %eax,-0x3c(%ebp) //buf[1] = temp;

  12:   8b 45 10                mov    _____(%ebp),%eax //temp = c;
  15:   89 45 c8                mov    %eax,-0x38(%ebp) //buf[2] = temp;

  18:   8b 45 14                mov    _____(%ebp),%eax //temp = d;
  1b:   89 45 cc                mov    %eax,-0x34(%ebp) //buf[3] = temp;
  1e:   c9                      leave
  1f:   c3                      ret

00000020 <bar>:
  20:   55                      push   %ebp
  21:   89 e5                   mov    %esp,%ebp
  23:   83 ec 10                sub    $0x10,%esp
  26:   c7 44 24 0c be ba fe ca movl   $0xcafebabe,0xc(%esp)
  2e:   c7 44 24 08 ef be ad de movl   $0xdeadbeef,0x8(%esp)
  36:   c7 44 24 04 43 82 01 00 movl   $0x18243,0x4(%esp)
  3e:   c7 04 24 13 52 01 00    movl   $0x15213,(%esp)
  45:   e8 fc ff ff ff          call   foo
  4a:   c9                      leave
  4b:   c3                      ret
```

A. Very briefly explain what purpose is served by the first three lines of the disassembly of foo (just repeating the code in words is not sufficient). No more than two sentences should be necessary here.

B. Note that in foo (C version), each of the four arguments are accessed in turn. The assembly dump of foo is commented to show where this is done. Recall that the current %ebp value points to where the pushed old base pointer resides, and immediately above that is the return address from the function call. Write into the gaps in the disassembly of foo the offsets from %ebp needed to access each of the four arguments a, b, c, and d. (Hint: Look at how they are arranged in bar before the call.)

C. GCC has a compile option called -fomit-frame-pointer. When given this flag in addition to the previous flags, the function foo is compiled like this:

```
00000000 <foo>
83 ec 40       sub    $0x40,%esp

8b 44 24 44    mov    ____(%esp),%eax //temp = a;
89 04 24       mov    %eax,(%esp)     //buf[0] = temp;

8b 44 24 48    mov    ____(%esp),%eax //temp = b;
89 44 24 04    mov    %eax,0x4(%esp)  //buf[1] = temp;

8b 44 24 4c    mov    ____(%esp),%eax //temp = c;
89 44 24 08    mov    %eax,0x8(%esp)  //buf[2] = temp;

8b 44 24 50    mov    ____(%esp),%eax //temp = d;
89 44 24 0c    mov    %eax,0xc(%esp)  //buf[3] = temp;
83 c4 40       add    $0x40,%esp
c3             ret
```

What is the difference between the first few lines of foo in the first compilation and in this compilation? What does this mean about what the stack frame looks like? (Consider drawing a before/after picture.)

D. Note what has changed in how the arguments a, b, c, d and the stack-allocated buffer are accessed: they are now accessed relative to %esp instead of %ebp. Considering that the arguments are in the same place when foo starts as last time, and recalling what has changed about the stack this time around (note: the pushed return address is still there!), fill in the blanks on the previous page to correctly access the function's arguments.

E. Consider what the compiler has done: foo is now using its stack frame without dealing with the base pointer at all... and, in fact, all functions in the program compiled with -fomit-frame-pointer also do this. What is a benefit of doing this? (0-point bonus question: What is a drawback?)

6. [6pts] Conditions

   A. [3pts] In x86, there are four condition codes: CF (carry), ZF (zero), SF (sign), OF (overflow) "setl" (set less than) instruction checks whether (SF^OF) is true. Explain why just using "SF" is not enough for the "less than" condition.

   B. [3pts] Explain why using conditional move for the following cases may be risky or incorrect.

      B-1 val = p ? *p : 0;

      B-2 val = x > 0 ? x*=7 : x+=3;

7. [4pts] Explain why the compiler and architecture define and follow the convention for caller/callee save registers .

8. [7pts] Consider the following C code

```
#define N 16
typedef int fix_matrix[N][N];

void fix_set_diag(fix_matrix A, int val) {
  int i;
  for (i=0; I < N; i++)
       A[i][j] = val;
}
```

GCC generates the following assembly code. The fill the gap in the code, and explain your answers.

```
    movl  8(%ebp), %ecx
    movl  12(%ebp), %edx
    movl  $0, %eax
.L14:
    movl  %edx, _____________    // hint: memory access
    addl  _____ , %eax           // hint: constant
    cmpl  _____ , %eax           // hint: constant
    jne   .L14
```

9. [5pts] In the following code, argue whether b and c always have the same value or not.

```
typedef union {
  float f;
  unsigned u;
} bit_float_t

float bit2float (unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}

unsigned  a = random();
float b = bit2float(a);
float c = (float) a;
```