1.
Floating Point
Convert the 32-bit floating point number 0x44361000 to decimal.

(Source: http://sandbox.mc.edu/~bennet/cs110/flt/ftod.html)

Answer:
728.25

0x44361000 = 0_10001000_01101100001000000000000

S = 0
E = 136 - 127 = 9
$M = 1 + 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-11}$

$(-1)^0 * (1 + 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-11}) * 2^9$
$= 1 * (2^9 + 2^7 + 2^6 + 2^4 + 2^3 + 2^{-2})$
$= 728.25$

2.
Fill in the Blanks:

_________ linking can suffer from issues such as code duplication, whereas _________ linking may take longer during runtime. (static, dynamic)

x86-64 is a (RISC/CISC) architecture, and MIPS is a (RISC/CISC) architecture. (CISC, RISC)

A ________________ is an array of page table entries (PTEs) that maps virtual pages to physical pages. (page table)

3.
Consider the following union and struct:

```
struct Galor {
    int first;
    float second;
    char third;

    union {
        struct {
            int number;
            float frac;
        };
        char name[10];
    } ;
};
```

Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called Sword, defined as:

```
struct Galor Sword[2][2];
```

Using gdb we find the following information at a particular stage in the application:

```
(gdb) p &Sword
$1 = (struct Galor (*)[2][2]) 0x7fffffffdff0

(gdb) x/96xb 0x7fffffffdff0
0x7fffffffdff0: 0x6b    0x72    0x00    0x00    0xec    0x51    0x05    0x42
0x7fffffffdff8: 0x3f    0x00    0x00    0x00    0x5a    0x61    0x6d    0x61
0x7fffffffe000: 0x7a    0x65    0x6e    0x74    0x61    0x00    0x00    0x00
0x7fffffffe008: 0x15    0x16    0x05    0x00    0xf5    0x19    0xd2    0x42
0x7fffffffe010: 0x2f    0x00    0x00    0x00    0x57    0x6f    0x6f    0x6c
0x7fffffffe018: 0x6f    0x6f    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe020: 0xe7    0x66    0xff    0xff    0x5c    0x2a    0x09    0x50
0x7fffffffe028: 0x32    0x00    0x00    0x00    0x43    0x53    0x33    0x33
0x7fffffffe030: 0x00    0x00    0xc8    0x43    0x00    0x00    0x00    0x00
0x7fffffffe038: 0x35    0x00    0x00    0x00    0x56    0x03    0x56    0xc3
0x7fffffffe040: 0x61    0xe1    0xff    0xff    0x44    0x72    0x65    0x64
0x7fffffffe048: 0x6e    0x61    0x77    0x00    0x00    0x00    0x00    0x00
```

What is the value of

```
Sword[1][0].frac
Sword[1][0].name
```

At this particular stage of the application?

`Sword[1][0].frac` == **400** // cuz there are 400 students enrolled heheh
`Sword[1][0].name` == **CS33**

Because of alignment, each object of type “`Galor`” is 24 bytes.

- 4 bytes for `first`
- 4 bytes for `second`
- 1 byte for `third`, plus 3 bytes of padding
- The union
  - The struct is 4 + 4 = 8 bytes
  - The cstring `name` is 10 bytes
  - The union is thus 10 bytes long
- 2 bytes of padding to remain aligned
  - (due to alignment, the next `int` has to be on an address of multiple of 4)
- 4 + 4 + (1+3) + 10 + 2 = 24 bytes

Thus, Sword[1][0] is at address …..fe020 to …..fe037

- Sword[1][0].frac is at address ….fe030 to ….fe033
  - 0x43c80000 => 400.000
- Sword[1][0].name is at address ….fe02c to ….fe035
  - cstrings stop at 0x00 (the ‘\0’ byte)
  - { 0x43, 0x53, 0x33, 0x33, 0x00 } => “CS33”

4.

Translate the x86 instructions into MIPS and vice versa:

a.

```
lea 0x4(%rdi,%rsi),%rax
```

With matching $t0 to %rdi, $t1 to $rsi, $t2 to %rax

```
      add $t3, $t1, $t0
      addi $t2, $t3, 4
```

b.

```
mov %rdx,(%rsp,%rsi,8)
```

With matching $t0 to %rsi, $sp to $rsp, $t1 to %rdx

```
      add $t2, $t0, $t0
      add $t2, $t2, $t2
      add $t2, $t2, $t2
      add $t3, $t2, $sp
      sw $t1, 0($t3)
```

c.

```
add $t1, $t0, $t0
add $t1, $t1, $t1
add $t3, $t2, $t1
lw $t3, 128($t3)
add $t4, $t4, $t3
```

```
add 0x80(%rsi,%rdi,4), %rax
```

5.

Is there a problem with the following code?
If yes, what is it? How can we fix the problem if there is one?

```
double* input = (double*) malloc (sizeof(double)*dnum);
double sum = 0;
int i;
for(i=0;i<dnum;i++){
      input[i] = i+1;
}

#pragma omp parallel for schedule(static)
for(i=0;i<dnum;i++)
{
      double* tmpsum = input+i;
      sum += *tmpsum;
}
```

There are a few things we can do. There is a race condition for the line with sum.

1. We can add a reduction(+:sum). This is the probably the most straightforward solution.
2. Or we can add in a critical section for this line so that only one thread can execute it at a time, it applies to all operations of the line.

```
#pragma omp parallel for schedule(static)
for(i=0;i<dnum;i++)
{
      double* tmpsum = input+i;
#pragma omp critical
      sum += *tmpsum;
}
```

3. Atomic allows only one thread to apply read/write operations at a time. This is better than critical because it only applies to read/write operations vs all of them so it is less costly.

```
#pragma omp parallel for schedule(static)
for(i=0;i<dnum;i++)
{
      double* tmpsum = input+i;
#pragma omp atomic
      sum += *tmpsum;
}
```

FYI: `schedule(static)`:
https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp

6.

We have a function that we are interested in:

```
int Toronto(char* game) {
      int curr_game = atoi(game);

      return Raptors(curr_game, 0);
}
```

We only know that the function `Raptors` has the following declaration:
`int Raptors(int game, int wins)`

While debugging, we notice the following output:

```
[(gdb) disas Raptors
 Dump of assembler code for function Raptors:
    0x000000000040068d <+0>:     push   %rbp
    0x000000000040068e <+1>:     mov    %rsp,%rbp
    0x0000000000400691 <+4>:     sub    $0x10,%rsp
    0x0000000000400695 <+8>:     mov    %edi,-0x4(%rbp)
    0x0000000000400698 <+11>:    mov    %esi,-0x8(%rbp)
    0x000000000040069b <+14>:    mov    -0x4(%rbp),%eax
    0x000000000040069e <+17>:    sub    -0x8(%rbp),%eax
    0x00000000004006a1 <+20>:    test   %eax,%eax
    0x00000000004006a3 <+22>:    js     0x4006bc <Raptors+47>
    0x00000000004006a5 <+24>:    mov    -0x8(%rbp),%eax
    0x00000000004006a8 <+27>:    lea    0x1(%rax),%edx
    0x00000000004006ab <+30>:    mov    -0x4(%rbp),%eax
    0x00000000004006ae <+33>:    sub    $0x1,%eax
    0x00000000004006b1 <+36>:    mov    %edx,%esi
    0x00000000004006b3 <+38>:    mov    %eax,%edi
    0x00000000004006b5 <+40>:    callq  0x40068d <Raptors>
    0x00000000004006ba <+45>:    jmp    0x4006ce <Raptors+65>
    0x00000000004006bc <+47>:    cmpl   $0x4,-0x8(%rbp)
    0x00000000004006c0 <+51>:    jne    0x4006c9 <Raptors+60>
    0x00000000004006c2 <+53>:    mov    $0x1,%eax
    0x00000000004006c7 <+58>:    jmp    0x4006ce <Raptors+65>
    0x00000000004006c9 <+60>:    mov    $0x0,%eax
    0x00000000004006ce <+65>:    leaveq
    0x00000000004006cf <+66>:    retq
 End of assembler dump.
```

What should be the input into the function `Toronto`, in order to get **a return value of 1**?

6 or 7

The above x86 instructions were from the following code:

```
int Raptors(int game, int wins) {

      if (game-wins >= 0)
            return Raptors(game-1, wins+1);
      else if (wins == 4)
            return 1;

      return 0;
}
```

7.
Say there was a function called `Warriors` in the Attack Lab, with the following C representation:

```
int Warriors(float* game) {

      float fourth = *(game+3);
      if (fourth == 68.75)
            return 1;

      return 0;
}
```

The function is at memory location `0x40178a`.
You need to execute the code for `Warriors` so that the function returns 1.
What should your input string be?

Your string is inputted using the same `getbuf` function as the Attack Lab, with a **24 byte buffer**.
The buffer begins at memory address `0x400680`.

You can assume that the **stack positions are consistent** from one run to the next, and that the section of memory holding the stack **is executable**.

68.75 is `0x42898000` in hex.

Accounting for 24 bytes of buffer, the return address pointing to the stack, the pop instruction, the float array location, and the function location, the array of floats should start at 56 bytes after the beginning of the buffer.
`0x400680 + 0x38 = 0x4006b8`

The input into the function should thus be `0x4006b8`, and should be popped into `%rdi`.

Thus, we can construct the following string input:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b0 06 40 00 00 00 00 00    // location "5f c3"
b8 06 40 00 00 00 00 00    // location of floats
8a 17 40 00 00 00 00 00    // function location
5f c3 00 00 00 00 00 00
00 00 00 00 00 00 00 00    // floats starts at first byte of this line
```

00 00 00 00 00 80 89 42