COS 217 Fall

Fall 2007

Please write your answers clearly in the space provided. For partial credit, show all work. State all assumptions. You have exactly 3 hours for this exam. This final is open book, open notes. Put your name on every page. Write out and sign the Honor Code pledge just before turning in the test. "*I pledge my honor that I have not violated the Honor Code during this examination.*"

Question	Score
1	/10
2	/10
3	/20
4	/20
5	/20
6	/20
Total	/100

Name:

Honor Code:

1 Floating Point

1. What is the IEEE 754 Wimpy Precision number 10101010 in decimal, assuming 4 exponent bits and 3 mantissa bits?

2. What is the IEEE 754 Wimpy Precision number 10000010 in decimal, assuming 4 exponent bits and 3 mantissa bits?

3. Write the formula to compute the value of a number in **denormalized** IEEE 754 format, use the S, E, and M variables to represent the integer values encoded in the binary fields. (Assume S is 1 bit, E is 11 bits, M is 52 bits.)

4. Write the formula to compute the value of a number in **normalized** IEEE 754 format, use the S, E, and M variables to represent the integer values encoded in the binary fields. (Assume S is 1 bit, E is 11 bits, M is 52 bits.)

2 Recursion

Consider the following program compiled and executed on a machine with 32-bit integers:

```
int fib(int n) {
  int temp = 0;
  if ((n == 0) || (n == 1))     /*  base cases */
     return 1;
  else  {                       /* recursive case */
     temp = fib(n - 1);
     return temp + fib(n - 2);
  }
}
```

1. What is the value of `fib(3)`?

2. While executing `fib(3)`, show the state of the stack (at the C-level is fine) at its largest point (that is with the most active activation records). Show the value of n and temp in each record, and label each activation record with function and argument value that initiated it.

3 Virtual Memory

1. Consider a laptop with 32-bit virtual addresses (per process), 32-bit physical addresses, and a 4KB page size. Assume each page table entry is 32 bits long and that the page table for all of virtual memory exists in physical memory. To map the full virtual address space, how much memory will be used by the page table for a single process? Clearly show your calculations.

2. Is that practical on a laptop with 1GB of physical memory installed? Why or why not?

3. Consider a typical server with 64-bit virtual addresses (per process), 48-bit physical addresses, and a 4KB page size. Assume each page table entry is 48 bits long and that the page table for all of virtual memory exists in physical memory. To map the full virtual address space, how much memory will be used by the page table for a single process? Clearly show your calculations.

4. Is that practical on a server with 8GB of physical memory installed? Why or why not?

5. Modern operating systems use multi-level page tables to address the problem identified above. In such a system, the first level page table refers to a second level of page tables. In some cases, a second level page table will refer to a third level of page tables. While the upper level page tables refer to lower level page tables, the lowest level page table refers to the actual process page. When combined with the ability to have a NULL pointer, indicating that a lower level page table or process page is not yet allocated, the above mentioned problem is solved. Describe how this addresses the above mentioned problem using a diagram of such a page table system. Use an example process with 12K of memory allocated as one 4K page of text, one 4K page of stack at the top of virtual memory, and one 4K page of heap at the bottom of virtual memory. Show how the virtual address is used to index each level page table.

4 DFA

You have been asked to write a program that inspects ASCII strings (input via stdin) and accepts those with both "Humpty" and "Dumpty" anywhere in the string. For example, the inputs "Humpty Dumpty sat on wall" and "Dumpty Humpty had a great fall" have both strings at least once (both accepted). But the string "Alice in Wonderland" does not (rejected). Neither do the strings "Humpty is fat" and "Dumpty is not" (both rejected). Don't write the program, but design it by drawing a deterministic finite automaton that recognizes the substrings "Humpty" and "Dumpty", showing and labeling all transitions (using the word "other" to label the transitions for any characters not otherwise specified, and "all" to label transitions taken for all characters) and clearly indicating the initial state with S (start) and accepting state(s) as F (finish). Draw neatly. For 2 points of extra credit try to minimize the number of intersections of transitions in your drawing.

5 Bug Hunt!

The assembly code on the next two pages reads in a number between 0 and 255 and prints it in binary. A typical instance of the I/O of this program should be:

```
INPUT:              Enter a number between 0 and 255: 58
OUTPUT:             00111010
```

(Note that the strings INPUT: and OUTPUT: are not actually printed out.) As you can see, it must print the bits in the "intuitive" way we are accustomed to.

1. The shown code compiles and links without any error, but upon executing the program, a segmentation fault occurs immediately. Locate and fix the bug. It is somewhere in the first 10 lines.

2. Assume now that bug 1 is fixed. When the user inputs 128, the program outputs the string "00000001", and then a segmentation fault occurs. Locate and then describe, in English, the flaw in the program's logic that is causing the wrong output. Don't worry about the segmentation fault just yet.

3. Locate and then describe, in English, the bug that is causing the segmentation fault.

4. Fix the bug(s) you identified above so that the program matches the specification.

5. From the way it is used, infer what the movsx instruction does.

6. Extend this program to accept a 32-bit integer from the user and print it out in binary. Make a minimum of changes.

```
 1  .section        ".roadata"
 2  cFormat1:
 3          .asciz  "%d"
 4          .align 4
 5  cFormat2:
 6          .string "Enter a number between 0 and 255: "
 7
 8  .section        ".text"
 9  ##Formal parameter offset
10  .equ    BYTE, 8
11
12  ##Local variable offset
13  .equ    COUNT, -4
14
15  ##Constants
16  .equ    INITIAL_COUNT, 8
17  .equ    MASK, 1
18  .globl convert_byte
19          .type   convert_byte, @function
20  convert_byte:
21          pushl   %ebp
22          movl    %esp, %ebp
23          pushl   %ebx
24          pushl   %esi
25          movl    $INITIAL_COUNT, %esi
26          movl    BYTE(%ebp), %eax
27          movl    %eax, %ebx
28          jmp     .L2
29
30  .L3:
31          movsx   %bl, %eax
32          andl    $MASK, %eax
33          movl    %eax, 4(%esp)
34          movl    $cFormat1, (%esp)
35          call    printf
36          movsx   %bl, %eax
37          sarl    %eax
38          movl    %eax, %ebx
39
40  .L2:
41          subl    $1, %esi
42          cmpl    $-1, %esi
43          jne     .L3
44          movl    $10, (%esp)
45          call    putchar
46          popl    %esi
47          popl    %ebx
48          movl    %ebp, %esp
49          popl    %ebp
50          ret
51
52  ##Local variable offset
53  .equ    MAINBYTE, -5
54  .equ    NUM, -8
55
56  ##Constants
57  .equ    INITIAL_COUNT, 8
58
59  .globl main
60          .type   main, @function
```

```
61  main:
62          pushl   %ebp
63          movl    %esp, %ebp
64          subl    $8, %esp
65          movl    $cFormat2, (%esp)
66          call    printf
67          leal    NUM(%ebp), %eax
68          movl    %eax, 4(%esp)
69          movl    $cFormat1, (%esp)
70          call    scanf
71          movl    NUM(%ebp), %eax
72          movb    %al, MAINBYTE(%ebp)
73          movsx   MAINBYTE(%ebp),%eax
74          movl    %eax, (%esp)
75          call    convert_byte
76          movl    $0, %eax
77          addl    $8, %esp
78          ret
```

6 Portability

1. Identify any portion of the following C program that is not portable. Show how to make those portions portable on multiple hardware and software platforms.

```
 1   #include <stdio.h>
 2   #include <stdlib.h>
 3   #include <string.h>
 4   #include <assert.h>
 5
 6   void invertCase(char *buf, int *numChar)
 7   {
 8      int i = 0;
 9      while(i<1024) {
10         if (buf[i] > 96 && buf[i] < 123) {
11           buf[i++] -= 32;
12       }
13       else if (buf[i] > 64 && buf[i] < 91) {
14          buf[i++] += 32;
15       }
16       else {
17          i++;
18       }
19     }
20     *numChar = i;
21  }
22
23  void getData(char *buf)
24  {
25     gets(buf);
26  }
27
28  void writeData(char *buf)
29  {
30     int len = strlen(buf);
31     fprintf(stdout, "%s\n", buf);
32  }
33
34  int main(int argc, char *argv[])
35  {
36     char *buf;
37     int *numChar;
38
39     buf = calloc(1, 1024);
40     numChar = calloc(1, 4);
41
42     getData(buf);
43     invertCase(buf, numChar);
44     writeData(buf);
45
46     return 0;
47  }
```

2. After making this program portable, please identify the portions of the resulting code that are not robust, and show how to make them robust. You can label the portions you have identified, and rewrite them here.