IT215 Lab 1 – Week 2 (11.1.16)

Goal: In this lab you will learn about endianness, gdb and stack frames.

To get checked off on this lab, the following exercises must be completed and shown to your lab TA by the end of the lab. If anything is not clear, please ask the TA for help.

Exercise 1

Consider a C function that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. Using casting, the procedure is this:

- Set an integer to 1
- Cast a pointer to the integer as a char *
- If the dereferenced pointer is 1, the machine is little-endian
- If the dereferenced pointer is 0, the machine is big-endian

The code for this function is as follows:

```
/* Returns 1 if the machine is little-endian, 0 if this machine is big-endian */

int endianness(void)
{
    int num;
    char *ptr;

    num = 1;
    ptr = (char *) &num;
    return (*ptr);    /* Returns the byte at the lowest address */
}
```

Let us consider a different way of solving this problem. A union is like a struct, except that all of the members are allocated starting at the same location in memory. This allows you to access the same data with different variable types. The syntax is almost identical to a struct.

Write a program lab1-ex1.c that uses a union to determine whether it is running on a little-endian machine or a big-endian machine.

Exercise 2

One area where endianness is of major concern to applications programmers is a situation where a binary file is written by one computer system and read by another. Before introducing binary file operations, let's very briefly review text file operations. Consider a C program using fprintf to write an int to a text file. Suppose that the character set in use is ASCII, that fp is of type FILE* and has been opened for output, and that i is an int. Consider this code fragment:

```
i = 12345;
fprintf(fp, "%d\n", 12345);
```

The fprintf function converts the computer's internal representation of 12345 to a sequence of bytes: the ASCII code for '1', the ASCII code for '2', and so on. These five bytes followed by the code for '\n' are written to the file. So the effect of the function call is to put this sequence of bytes in the file:

```
00110001
00110010
00110011
00110100
00110101
00001010
```

(Actually, the above is what you would get on a Linux system. On Microsoft Windows you would get

```
00110001
00110010
00110011
00110100
00110101
00001101
00001010
```

because Windows uses a two-byte sequence to terminate lines in text files. But this exercise is mostly about binary files, not text files.) Unlike a text file, a binary file is not necessarily a sequence of character codes. The C view of a binary file is a sequence of bytes that might be character codes, pieces of instructions, pieces of integers or pieces of floating point numbers, or that might have some other meaning. The key C library functions for binary file input and output are fread and fwrite. The following code demonstrates writing the bit pattern for an int to a binary file. Again suppose that fp is of type FILE* and has been opened for output:

```
i = 12345;
fwrite((void *) &i, sizeof(int), 1, fp);
```

The arguments are: the address of the first byte to be copied to the file; the size, in bytes, of the data items to be copied to the file; the number of data items to be written to the file; fp, which specifies the file. The 32-bit representation of 12345 is

0000_0000_0000_0000_0011_0000_0011_1001

fwrite simply copies a sequence of bytes from memory to a file. So on a 32-bit big-endian machine, the call to fwrite will put the following sequence of bytes in the file:

00000000
00000000
00110000
00111001

But on a little-endian machine the bytes would be written in this order:

00111001
00110000
00000000
00000000

Note that in both cases the bytes are totally different from what is produced by the call to fprintf. The function to read from a binary file is fread. It should be clear that endianness can cause problems if a binary file is written using fwrite on a machine with one endianness and read using fread on a machine with the opposite endianness.

There are two C programs in the lab folder. The program binwrite.c generates a small binary file with the following format: the first four bytes are an unsigned int giving the number of array elements stored in the file, and the remaining bytes are elements from an array of unsigned ints, four bytes per element. The program binread.c reads a file in the same format and displays the number of elements and array contents.

Read the two programs carefully to get an idea of how they work.

There are two data files in the directory: x86_binwrite_out.dat is an output file produced by binwrite.c on a (little-endian) x86-based Linux system; ppc_binwrite_out.dat is an output file produced by binwrite.c on a (big-endian) Apple iBook (with a big-endian PowerPC processor) running Mac OS X. On Linux, the hexdump tool can be useful for examining the contents of binary files. It takes various options for displaying file contents in various ways; with the -C option, each line of output shows 16 bytes from the input file as two-digit hexadecimal numbers, along with ASCII interpretations where possible.

Enter the command

```
hexdump -C ppc_binwrite_out.dat
```

and then the command

hexdump -C x86_binwrite_out.dat

What you see should help you understand the effect of endianness on the sequences of bytes in the two files.

Build an executable from binread.c. Run it using x86_binwrite_out.dat as the input file; then run it with ppc_binwrite_out.dat as the input file. The difference in behaviour will be obvious. Make a copy of binread.c and modify it so that it can correctly read the data in ppc_binwrite_out.dat. Do not modify the calls to fread; instead use operations such as shifts (C << and >> operators) and bitwise and's and or's (C & and | operators) to rearrange the bytes within variables.

Exercise 3

Learn about gdb, you may use the following link: http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html

1. Download the file foo.c, and compile it with the –g option. This embeds the line numbers in the executable, so that we can set breakpoints. Type *gcc –g –o foo foo.c*
2. Run this program in the debugger: *gdb ./foo*
3. Set a breakpoint at line 9 of the program: <*break foo.c: 9* > This is the name of a source file, and a line number in that source file, where the breakpoint will be set. We could also have set a breakpoint at the point where the function *foo* is entered, by typing break *foo*.
4. Run the program until it hits the breakpoint: *run*
5. List the function that we have stopped in: *type l (the letter l, and return).*
6. List the disassembly of the assembly code that we have stopped in: *disassemble*
7. Print the contents of the local variables: *print acc, print buf* . We can also list the arguments by info args. Try it.
8. Print the contents of the *$esp* register: *print $esp*. Print the contents of all the processor registers.
9. Look at the contents of the stack. The stack is in memory starting at the address in *esp*. We want to print out the contents of memory at *esp* and the bytes at immediately higher addresses. This is done using the "*x*" (examine memory) command. The "*x*" command allows you to print the contents of memory locations starting from a specified address.
10. Look at the words in the stack frame and interpret the meaning of most of them. There is a simple way to interpret the entries: *info frame*. You need to be able to interpret the stack yourself, though, without the use of this command.
11. Set a breakpoint at the return statement. Run the program until it hits the breakpoint. Print the contents of the *$esp* register. Look at the stack again and interpret the words in the stack frame.

Draw a picture of the stack for each of the following situations:

A. Immediately after the function *foo* is called, but before executing any statement from this function.
B. Just before returning from *foo* (the *return* statement is not executed)
C. After *foo* returns to *main*, but before *printf* is called in *main.*

Exercise 4

Compile and run the following program. When you try to run it, you should get the message “Segmentation fault”. This and “Bus Error” are the two most common run-time errors. Note that no indication is given as to where the program failed. You can imagine that in a large program, the message is pretty useless in trying to locate the problem.

```
#include <stdio.h>

int main()
{
      int a[10];
      int *a1;

      a1 = a;
      a1 = 0;

      a1[0] = 1;

      return 0;
}
```

You can use gdb to find the location of a segmentation fault. Run the program in gdb. Save the results of your gdb session in a text file lab1-ex4.txt. Answer the following questions, in the text file:

- Which line of this program resulted in a segmentation fault?
- Why does the segfault occur?

Exercise 5

Given the C function

```
1   int proc(void)
2   {
3       int x, y;
4       scanf(“%x %x”, &y, &x);
5       return x – y;
6   }
```

Assume GCC generates the following assembly code:

```
1   proc:
2      pushl %ebp
3      movl %esp, %ebp
4      subl $24, %esp
5      addl $-4, %esp
6      leal  -4(%ebp), %eax
7      pushl %eax
8      leal -8(%ebp), %eax
9      pushl %eax
10     pushl $.LC0              Pointer to string "%x %x"
11     call scanf
       Diagram stack frame at this point
12     movl -8(%ebp), %eax
13     movl -4(%ebp), %edx
14     subl %eax, %edx
15     movl %edx, %eax
16     movl %ebp, %esp
17     popl %ebp
18     ret
```

Assume that procedure proc starts executing with the following register values:

Register	Value
%esp	0x800040
%ebp	0x800060

Suppose proc calls scanf (line 11), and that scanf reads values 0x46 and 0x53 from the standard input. Assume that the string "%x %x" is stored at memory location 0x300070. Create a text file lab1-ex5.txt containing the answers to the following questions. Be sure to write the reasoning behind your answers.

A. What value does %ebp get set to on line 3?
B. At what addresses are local variables x and y stored?
C. What is the value of %esp after line 10?