Lab 8: Memory Sections and Memory Dump
Due: December 4th, 2015

This lab consists of 3 main parts.

Learning Objectives:
Part1: Checking the Endianness of the RPI and Purdue's 'data' and 'lore' servers
Part2: Identifying memory sections
Part 3: Memory Dump

Part 1: Machine Endianness

Data which occupies more than one byte in a byte-addressable memory can be
stored in two different ways: Little-Endian or Big-Endian.

- In Little-Endian, the least significant byte of the integer is stored in the
  lowest address in memory.
- In Big-Endian, the most significant byte is stored in the lowest address in
  memory.

1.1 Goal
- Understand machine endianness
- Tell the difference between little-endian and big-endian

1.2 What to do
- Download endian.c code from:
  http://courses.cs.purdue.edu/_media/cs25000:fall2015:labs:lab08materials.zip
- Write the “isLittleEndian” function to determine the endianness of the
  machine the program is compiled and run on. Hint: In order to write the
  isLittleEndian function, you can take advantage of different types of pointers.
  For example, dereferencing integer pointers will return 4 bytes of data, and
dereferencing character pointers will return 1 byte of data.
- Run your code on your pi and try the same program in lore.cs.purdue.edu
  and in data.cs.purdue.edu. Type “uname -a” to know what processor each
  machine uses.
- Fill up the following table

<table>
<thead>
<tr>
<th>Host Name</th>
<th>Architecture(x86,ARM,SPARC)</th>
<th>Endianness</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lore.cs.purdue.edu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data.cs.purdue.edu</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part 2: Memory Sections

The memory of the program is divided in the following memory sections:

<table>
<thead>
<tr>
<th>Memory Section Name</th>
<th>Description</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>text (or code segment)</td>
<td>This is the area of memory that contains the machine instructions that correspond to the compiled program. This area is shared by multiple instances of a running program. The text segment also contains constants such as string literals and variables defined using the const keyword.</td>
<td>Read, Execute</td>
</tr>
<tr>
<td>data</td>
<td>This area in the memory image of a running program contains storage for initialized global variables, and static variables that are explicitly initialized to a non-zero value. This area is separate for each running instance of a program.</td>
<td>Read, Write</td>
</tr>
<tr>
<td>bss</td>
<td>This is the memory area that contains storage for uninitialized global variables and static variables that are not explicitly initialized or initialized to zero. It is also separate for each running instance of a program.</td>
<td>Read, Write</td>
</tr>
<tr>
<td>stack</td>
<td>This region of the memory image of a running program contains storage for the automatic (non-static local) variables of the program. It also stores context-specific information before a function call, e.g. the value of the Instruction Pointer (Program Counter) register before a function call is made. On most architecture the stack grows from higher memory to lower memory addresses. A running instance of a program can have multiple stacks (as in a multi-threaded program)</td>
<td>Read, Write</td>
</tr>
<tr>
<td>heap</td>
<td>This memory region is reserved for dynamically allocating memory for variables, at run time. Dynamic memory allocation is done by using the malloc or calloc functions and new operator.</td>
<td>Read, Write</td>
</tr>
<tr>
<td>shared libraries</td>
<td>This region contains the executable image of shared libraries being used by the program.</td>
<td>Read, Execute</td>
</tr>
</tbody>
</table>

2.1 Goal

- Understand the memory layout and structure of a program on Linux-based system.

2.2 What to do
Get the sections.c program from:
http://courses.cs.purdue.edu/_media/cs25000:fall2015:labs:lab08materials.zip

Compile the code and run it like this:
```
gcc -o sections sections.c
./sections
```

In a piece of paper, draw an approximate map of the memory of the program indicating the text, data, bss, stack, heap of the program. Also draw where each variable is located as well as the address as indicated by the program.

Part 3: Memory Dump

3.1 Goal
- Read and parse lower level data representation
- Understand basic mechanism of memory allocation and management on Linux-based system

3.2 What to do
- Get the memdump.c program from:
  http://courses.cs.purdue.edu/_media/cs25000:fall2015:labs:lab08materials.zip
- Complete the function memdump(char *p, int len) in the memdump.c program. The memdump( ) function stands for memory dump. It will print a chunk of memory to the terminal starting at the location pointed by 'p', and stopping after 'len' bytes.

Output format:
The output will have three components per line.

- The first component on the line is the full address in hexadecimal format, followed by a colon and a space. The Raspberry Pi is a 32-bit machine, meaning pointers are 32 bits long. The address you print should also be 32 bits long, or 8 hexadecimal digits since each hex digit represents 4 bits. The address should be 16-byte aligned. If the starting address is not 16 byte aligned, round down to the nearest 16th byte, and print periods until your reach the starting address (See example output for reference) To easily print addresses using printf(), use the %p format string (see the provided code for example). Keep in mind you still need to print 'len' number of bytes after the original starting address. Here is an example of what the address should look like: 0xbeab36e0:

- The second component on the line is 16 consecutively-located bytes of memory contents starting at the address printed previously. The contents of memory should be printed in hexadecimal format with spaces between each
byte. One byte is represented by two hex digits. To easily print two hex digits using printf, use the '%02hhx' format string. For example, printf("%02hhx", *pointer) will print two hex digits (1 byte) at the memory location pointed by the variable 'pointer'.

- For the third component on the line, interpret the same 16 bytes of memory content as ASCII characters and print the result (which sounds very wrong, I know) as follows. "Printable" ASCII characters are all letters, symbols, and punctuation, or decimal equivalent values between 32 \(\leq c \leq 126\) (or 20 \(\leq c \leq 7E\) in hex). "Non-printable" characters are the control characters with decimal values below 32 (\(\leq 0x20\)) or equal to 127 (0x7F). Attempting to print control characters yields non-visible results in some cases and badly formatted output in other cases. So, you should print the "printable" characters and print a period '.' as a placeholder for the non-printable characters. The reason many memdump programs print out the ASCII representation of memory is to expose characters and character strings in memory.

Here is a partial example of the output. Make sure the format matches the one below:

```
3.2 What to do Next

- Run your version of memdump that prints the output above. The output may be different than the output above because the addresses will be different, and uninitialized memory is random.
- Print the output and then with a pen as clear as you can indicate (underline, circle, highlight) where the following items are located:
  o Part 3.1 after variable assignment
    i. str
```
ii. a
iii. b
iv. y
v. x.a
vi. x.b
vii. x.p
viii. int_array[n] for n equals to 0 through 3
  o Part 3.1 after buffer overflow
    i. int_array[n] for n equals to 4 through 7
  o Part 3.2 after variable assignment
    i. root
    ii. left
    iii. right
    iv. root->str
    v. left->str
    vi. right->str

• Also, in paper compute binary value of “int b” (two’s complement of -5) and verify value of “double y” (the sign, mantissa, and exponent). Verify that the values stored in memory are correct by showing your work.

What to submit for lab 8

1. Put your completed endian.c and memdump.c programs in the folder lab8_yourPUID, and turn in the folder using the command:
   $ turnin -c cs250 -p lab8 lab8_yourPUID
   $ turnin -c cs250 -p lab8 -v

2. The completed table in Part 1
3. The map of the sections in memory in Part 2
4. The output of your memdump program in Part 3 indicating where each item is found in the output.
5. Your work showing the verification of the “int b” and “double y” values stored in memory