Notes

- Project 7 due today
  – questions?
- Project 5 secret tests posted, visible on submit server
- Practice final exam posted, with answers soon
- Read Sections 2.2 – 2.4 of Bryant and O'Hallaron on data representation
- Please do course evaluation, at www.CourseEvalUM.umd.edu

Creating a shared library

- To create a shared library:
  – use the special gcc flags
    - nostdlib -shared -fPIC -Wl,-soname,libraryname.so.1
    - nostdlib means that no standard C library is needed
    - shared says to generate a shared library
    - fPIC says to generate position-independent code
    - Wl,-soname,libraryname.so.1 says to name the shared object
       libraryname.so.1 (for whatever libraryname is)
  – example Makefile rules that do this, supposing we want to create a
    shared library libavl.so from two object files avl.o and node.o:

```
LIBFLAGS = -nostdlib -shared -fPIC -Wl,-soname,libavl.so.1
libavl.so: avl.o node.o
  $(CC) $(LIBFLAGS) avl.o node.o -o libavl.so.1
  ln -s -f libavl.so.1 libavl.so
```
Using a shared library

- To compile a program that uses a shared library:
  - Assume the library file `libavl.so.1` is in the current directory, and the symbolic link `libavl.so` points to it, and the program in `main.c` wants to use functions from the library `libavl.so` (the functions from `avl.o` and `node.o`) created above:
    
    ```
    gcc -o main main.o -L. -lavl
    ```
  - the option `-L.` tells the compiler to search the current directory during compilation for libraries (although not during runtime)
  - the option `-lavl` tells the compiler to look for a library file `libavl.so` (which in this case is a symlink to the actual library)

Using a shared library, con't.

- To run a program that uses a shared library:
  - setting the environment variable `LD_LIBRARY_PATH`, as in `setenv LD_LIBRARY_PATH .`
    
    tells the program loader to look in a nonstandard location (the current directory) for shared libraries
  - then just run `main` and the library is loaded when `main` begins to run (when it's first loaded into memory). Notice that the code in `avl.o` and `node.o` was never linked with the code in `main.o`, but it calls the functions in them via the shared library

Dynamically loading a library

- C functions that support this:
  - `void *dlopen(const char *pathname, int mode);`
    
    - `pathname` is the name of a shared library
    - `mode` controls the function's operation
      - `RTLD_NOW`: when this shared library is loaded, indicate if there is anything that is not included which is needed immediately
      - `RTLD_LAZY`: wait and look for things only when they're actually needed from the library
    - returns a pointer or `handle` referring to the library, which can be used for subsequent calls to look up functions in the library

Dynamically loading a library, con't.

- `void *dlsym(void *handle, const char *name);`
  - looks up a function by name in the passed shared library
  - returns a pointer to that function (or `NULL` if not found)

- `int dlclose(void *handle);`
  - returns 0 on success

- `const char *dlerror(void);`
  - returns a pointer to a string describing the error from the last call to any of the other functions, or `NULL` if no errors have occurred since initialization, or since it was last called

- To use these functions, `#include <dlfcn.h>`
Dynamically loading a library, con't.

• To compile a program that uses the above functions to dynamically load a library:
  – add the options \texttt{-rdynamic} and \texttt{-ldl}
  – for example, assume the program in \texttt{main.c} was modified to use the \texttt{dlfcn} functions above, and wants to dynamically load functions from the library \texttt{libavl.so.1} in the current directory:

\begin{verbatim}
gcc -rdynamic -ldl -o main main.c
\end{verbatim}

Dynamically loading a library, con't.

• To run a program that dynamically loads a library:
  – we again need to tell the program loader to look in the current directory for libraries, using \texttt{setenv LD_LIBRARY_PATH}.
  – then just run \texttt{main}
    • the library is opened when the program calls \texttt{dlopen()}
    • functions in it are loaded when it calls \texttt{dlsym()}, and can be executed via the returned function pointer.
  – Notice that the code in \texttt{avl.o} and \texttt{node.o} was never linked with the code in \texttt{main.o}, and \texttt{main.c} doesn't even contain regular calls to the functions, just their names in calls to \texttt{dlsym()}

Representing characters

• We need:
  – to be able to represent common characters
  – to have standards so computers can interoperate
• Common formats
  – ASCII
    • is the most commonly used character code
    • uses 7 bits for characters (stored in 8 bits normally)
  – EBCDIC
    • an 8-bit code, used now only by some IBM mainframes
  – UNICODE
    • a family of encodings - 8, 16, and 32 bits per character
    • allows a greater variety of characters
    • is able to represent virtually any character in use today in any language, and some no longer in use
ASCII

• Represents normal characters on US keyboards
  – A-Z (the characters numbered 65-90)
  – a-z (97-122)
  – 0-9 (48-57)
  – space (32)
  – control characters (0-31, 127)
    • the first 26 (after 0) of the 33 ASCII control characters have names
      Ctrl-A - Ctrl-Z
    • for example, ASCII character 13, Ctrl-M, is CR (carriage return)
      \r in C; ASCII char. 9, Ctrl-I, is HT (horizontal tab) \t in C
  – punctuation: !@#$%^&*()_+-=\[\]{}|;:"'<>,./ (the remaining characters)
• The UNIX command "man ascii" shows the ASCII character set

UNICODE

• Different representations
• UTF-32: a 32-bit representation of all characters
  – all characters are the same size
  – uses lots of space (twice as much as UTF-16 for most things, four times as much as ASCII for many things)
• UTF-16: a 16-bit representation of characters
  – some characters are stored in two-character forms
  – is popular since most things can be represented in 16 bits
• UTF-8: an 8-bit representation of characters
  – provides backwards compatibility with ASCII
    • the low 7 bits are exactly ASCII
  – if the high bit is on it indicates part of UNICODE extensions
  – popular for web and other applications

Representing unsigned integers

• All data is stored in binary
  – all digits are 0 or 1
• In an unsigned number every bit position $i$ represents the value $2^i$, where $i$ is 0 for the rightmost bit. The value of a number is the sum of the values of the bit positions containing a 1.
• Example bit position values for an 8-bit number:

\[
\begin{array}{c|c|c|c|c|c|c|c}
\hline
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\hline
\end{array}
\]

• The range of values that can be stored is 0 to $2^n - 1$, where $n$ is the number of bits used
  – for example, using 16 bits, the numbers 0 to 65,535 can be represented

Representing signed integers

• Signed integers are usually stored using two's complement
  – the leftmost bit indicates if a number is positive (0) or negative (1)
• To get the two's complement representation of a negative value, flip all the bits of the positive value and add 1
• Two's complement allows easy addition of positive and negative numbers (just ignore overflow)
• The range of values that can be stored is $-2^{n-1}$ to $2^{n-1}-1$ for $n$ bits
  – for example, using 16 bits, the numbers -32,768 to 32,767 can be represented
• Two's complement isn't the same thing as an unsigned number with a sign bit
  – -5 is \(~(5) + 1 = 11111010 + 1 = 11111011\), not 10000101
Floating point representation

- Computers normally use a radix of 2 – binary (people often prefer a radix of 10 – decimal)
- Examples of floating point numbers
  10.5_{10} = 1010.1_{2} = 1.0101 \times 2^{3}
  7.4375_{10} = 111.0111_{2} = 1.110111 \times 2^{2}
- Decimal/binary points:

<table>
<thead>
<tr>
<th>10^3</th>
<th>10^2</th>
<th>10^1</th>
<th>10^0</th>
<th>10^{-1}</th>
<th>10^{-2}</th>
<th>10^{-3}</th>
<th>10^{-4}</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
<td>2^0</td>
<td>2^{-1}</td>
<td>2^{-2}</td>
<td>2^{-3}</td>
<td>2^{-4}</td>
</tr>
</tbody>
</table>

How are floats/doubles represented internally?

- Each number has three parts:
  - its sign (s), which is 0 for positive numbers, and 1 for negative numbers
  - a mantissa (m), which represents a number between 0 and 1
    - it's represented as a binary number, i.e., \( \frac{1}{2} = 0.1 \)
    - it's normalized into \([1,2)\) (the exponent is adjusted as needed)
  - an exponent (e), which designates the position of the binary point
- A number is \((-1)^{s} \times m \times r^{e}\), where \(r\) is the radix
  - the number 6132.789_{10} = 1 \times 6.132789 \times 10^{3} (the radix is 10 for this example)
  - the number 0.05_{10} = 1 \times 5.0 \times 10^{-2} (radix is also 10)
  - the number -1001.1110_{2} = -1 \times 1.001110 \times 2^{3} (here the radix is 2)
- This is much like scientific notation, with the addition of the sign as a factor, and the ability to use a base other than 10

The IEEE 754 floating point standard

- The IEEE 754 floating point standard has different sizes for values:
  - 32 bit floating point (C float):
    - 1 sign bit, 8 bits exponent, 23 bits mantissa
    - the range of representable values is approximately \(2^{-126} \ldots 2^{127}\), which is approximately \(1.2 \times 10^{-38} \ldots 3.4 \times 10^{38}\)
  - 64 bit floating point (C double):
    - 1 sign bit, 11 bits exponent, 52 bits mantissa
    - this is the precision most commonly used for real applications
    - the range of representable values is approximately \(2^{-1022} \ldots 2^{1023}\), which is approximately \(2.2 \times 10^{-308} \ldots 1.8 \times 10^{308}\)
  - 128 bit floating point (quad):
    - 1 sign bit, 15 bits exponent, 112 bits of mantissa
    - this is not commonly used

More about IEEE 754 floating-point numbers

- The leading 1 of the mantissa isn't stored:
  - the binary point (like a decimal point) is moved just to the right of the leftmost nonzero digit
  - but in binary, the leftmost nonzero digit must be a 1, so there's no need to actually store it, giving one more bit of precision in the mantissa for free
- The exponent:
  - uses a bias, rather than two's complement, for storing negative as well as positive exponents. The bias is added to the exponent's value.
  - the bias is 127 for single-precision IEEE numbers (C float's), and 1023 for double-precision numbers (C double's)
- The use of a bias allows the representation of the number zero to be all zeros; in fact, an exponent of all 1s or all 0s represents a special number:
  - 0, infinities, NaN, denormalized numbers
Example IEEE floating-point number

- Here's how the example number -25.625 is represented in IEEE floating point (single precision):
  - The sign bit (one bit) is 1, since the number is negative; we compute the absolute value of the number below
  - To compute the mantissa (23 bits):
    - write the number in binary, with a binary point:
      \[
      25_{10} = 11001_2
      \]
      \[
      .625_{10} = 1/2 + 1/8, \text{ which is } .101_2
      \]
      so 25.625_{10} is 11001.101_2
    - move the binary point right after the first nonzero digit, giving 1.1001101 (moved 4 places to the left)
    - drop the leading 1 (and the binary point), giving 1001101
    - add zeros to the right to get 23 bits (here 16 zeros are needed)
    - so the mantissa is \(10011010000000000000000\)

Example IEEE floating-point number, cont.

- Recall the example number is -25.625
  - To determine the exponent (8 bits):
    - in the previous step, we moved the binary point 4 places to the left to place it to the right of the first nonzero digit, so the exponent value is 4
    - to bias the exponent, we add 127; 127 + 4 = 131, so the value of the exponent field is 131
    - 131 in binary is 10000011
  - Putting it all together, the number is represented as \((-1)^1 \times 1.1001101 \times 2^4 = -1.6015625 \times 16 = -25.625\)
  - And the number is stored in memory as

Imprecision with real numbers

- The real numbers are dense (unlike the integers), but anything in computer memory has to be stored in a finite bit representation; this causes imprecision
- First consider an analogy with decimal numbers:
  - There are some numbers that can't be represented exactly in a finite number of digits - they require an infinite number of repeating digits
  - Example: 1/3 = .333333333333...
  - Suppose we have only a fixed number of decimal digits in which to express 1/3, say for example 8 digits. The closest we can get is .33333333. But notice this is .00000000333333... away from the actual number 1/3
  - The next representable number (if we only have 8 digits) is .33333334, and any number between these two can only be approximated as one or the other of these two values- there are no values between them

Imprecision with real numbers, cont.

- In binary there are also real numbers (not necessarily the same ones as in decimal) that can't be represented in a finite number of (binary) digits
- Example: \((1/3)_{10} = .01010101010101010101...
- Another example: \((1/5)_{10} = .00110011001100110011...
- If we have only four binary digits, the closest we can come to representing \((1/5)_{10}\) is .0011 (1/8 + 1/16 = .1875)
- If we have eight binary digits, we can come closer to representing \((1/5)_{10}\) : .00110011 (1/8 + 1/16 + 1/128 + 1/256 = .19921875). The more digits we have, the closer we can come to representing it
- But we'll never get exactly to 0.2_{10}, if we only have a fixed number of binary digits in which to represent the number
Imprecision with real numbers, cont.

- The IEEE representation of 1/5, with a 23-digit mantissa, is 00111110010011001100110011001101, which works out to 0.20000000298023223876953125
- The next smaller bit pattern (only one bit different) is 00111110010011001100110011001100, which works out to 0.199999988079071044921875000
- **There is no (single-precision) IEEE 754 float between these two values** because, with a fixed 23 digits of mantissa, there is no bit pattern between them
- If you try to compute or store values between these, such as 0.19999998825, 0.19999998850, 0.19999998875, etc., they'll all be represented as 00111110010011001100110011001100, which is 0.199999988079071044921875000