Porting Linux applications to 64-bit systems

Tips and techniques for a smooth transition

Harsha S. Adiga

April 12, 2006

With the pervasiveness of 64-bit architectures, it's more important than ever that your Linux® software be 64-bit ready. Learn how to avoid portability pitfalls when making declarations and assignments, bit shifting, typing, formatting strings, and more.

Linux was one of the first cross-platform operating systems to use 64-bit processors, and now 64-bit systems are becoming commonplace in servers and desktops. Many developers are now facing the need to port applications from 32-bit to 64-bit environments. With the introduction of Intel® Itanium® and other 64-bit processors, making software 64-bit-ready has become increasingly important.

As with UNIX® and other UNIX-like operating systems, Linux uses the LP64 standard, where pointers and long integers are 64 bits but regular integers remain 32-bit entities. Although some high-level languages are not affected by the size differences, others such as the C language may be.

The effort to port an application from 32 bits to 64 bits might range from trivial to very difficult, depending on how these applications were written and maintained. Many subtle issues can cause problems even in a well-written, highly portable application, so this article outlines these issues and suggests ways to deal with them.

Advantages of 64 bits

32-bit platforms have a number of limitations that are increasingly frustrating to developers of large applications such as databases, especially those developers who wish to take advantage of advances in computer hardware. While scientific calculations normally rely on floating-point mathematics, a few applications such as financial calculations need a narrower numeric range but higher precision than floating point offers. 64-bit math provides this higher precision fixed-point math, with an adequate range. There is much discussion today in the computer industry about the barrier presented by 32-bit addresses. 32-bit pointers can address only 4GB of virtual address space. You can overcome this limitation, but application development becomes more complicated, and performance is significantly reduced.
As far as language implementation is concerned, the current C language standard allows the "long long" data type to be at least 64 bits. However, an implementation may define it as a larger size.

Another area that requires improvement is dates. In Linux, dates are expressed as signed 32-bit integers representing the number of seconds since January 1, 1970. This turns negative in 2038. But in 64-bit systems, dates are expressed as signed 64-bit integers, which extends the usable range.

In summary, the 64-bit architecture has the following advantages:

- A 64-bit application can directly access 4 exabytes of virtual memory, and the Intel Itanium processor provides a contiguous linear address space.
- 64-bit Linux allows for file sizes up to 4 exabytes (2 to the power of 63), a very significant advantage to servers accessing large databases.

The Linux 64-bit architecture

Unfortunately, the C programming language does not provide a mechanism for adding new fundamental data types. Thus, providing 64-bit addressing and integer arithmetic capabilities involves changing the bindings or mappings of the existing data types, or adding new data types to the language.

Table 1. 32-bit and 64-bit data models

<table>
<thead>
<tr>
<th></th>
<th>ILP32</th>
<th>LP64</th>
<th>LLP64</th>
<th>ILP64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>64</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>pointer</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

The difference among the three 64-bit models (LP64, LLP64, and ILP64) lies in the non-pointer data types. When the width of one or more of the C data types changes from one model to another, applications may be affected in various ways. These effects fall into two main categories:

- **Size of data objects.** The compilers align data types on a natural boundary; in other words, 32-bit data types are aligned on a 32-bit boundary on 64-bit systems, and 64-bit data types are aligned on a 64-bit boundary on 64-bit systems. This means that the size of data objects such as a structure or a union will be different on 32-bit and 64-bit systems.

- **Size of fundamental data types.** Common assumptions about the relationships between the fundamental data types may no longer be valid in a 64-bit data model. Applications that depend on those relationships will fail when compiled on a 64-bit platform. For example, the assumption `sizeof (int) = sizeof (long) = sizeof (pointer)` is valid for the ILP32 data model, but not valid for others.
In summary, the compilers align data types on a natural boundary, which means that "padding" will be inserted by the compiler to enforce this alignment, as in a C structure or union. The members of the structure or union are aligned based on their widest member. Listing 1 illustrates this structure.

**Listing 1. C structure**

```c
struct test {
    int i1;
    double d;
    int i2;
    long l;
}
```

Table 2 shows the size of each member of the structure and the structure size itself on 32-bit and 64-bit systems.

**Table 2. Size of structure and structure members**

<table>
<thead>
<tr>
<th>Structure member</th>
<th>Size on 32-bit system</th>
<th>Size on 64-bit system</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct test {</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int i1;</td>
<td>32-bits</td>
<td>32-bits</td>
</tr>
<tr>
<td></td>
<td>32-bits filler</td>
<td>32-bits filler</td>
</tr>
<tr>
<td>double d;</td>
<td>64-bits</td>
<td>64-bits</td>
</tr>
<tr>
<td>int i2;</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td></td>
<td>32-bits filler</td>
<td>32-bits filler</td>
</tr>
<tr>
<td>long l;</td>
<td>32 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>}</td>
<td>Structure size 20 bytes</td>
<td>Structure size 32 bytes</td>
</tr>
</tbody>
</table>

Note here that on a 32-bit system, the compiler may not align the variable `d`, even though it is a 64-bit object, because the hardware treats it as two 32-bit objects. However, a 64-bit system aligns both `d` and `l` causing two 4-byte fillers to be added.

**Porting from 32-bit to 64-bit systems**

This section shows you how to correct common trouble spots:

- Declarations
- Expressions
- Assignments
- Numeric constants
- Endianism
- Type definitions
- Bit shifting
- Formatting strings
- Function parameters

**Declarations**

To enable your code to work on both 32-bit and 64-bit systems, note the following regarding declarations:
• Declare integer constants using "L" or "U", as appropriate.
• Ensure that an unsigned int is used where appropriate to prevent sign extension.
• If you have specific variables that need to be 32-bits on both platforms, define the type to be int.
• If the variable should be 32-bits on 32-bit systems and 64-bits on 64-bit systems, define them to be long.
• Declare numeric variables as int or long for alignment and performance. Don’t try to save bytes using char or short.
• Declare character pointers and character bytes as unsigned to avoid sign extension problems with 8-bit characters.

Expressions
In C/C++, expressions are based upon associativity, precedence of operators and a set of arithmetic promotion rules. To enable your expression to work correctly on both 32-bit and 64-bit systems, note the following rules:

• Addition of two signed ints results in a signed int.
• Addition of an int and a long results in a long.
• If one of the operands is unsigned and the other is a signed int, the expression becomes an unsigned.
• Addition of an int and a double results in a double. Here, the int is converted to a double before addition.

Assignments
Since pointer, int, and long are no longer the same size on 64-bit systems, problems may arise depending on how the variables are assigned and used within an application. A few tips in this regard:

• Do not use int and long interchangeably because of the possible truncation of significant digits. For example, don’t do this:
  ```
  int i;
  long l;
  i = l;
  ```
• Do not use an int to store a pointer. The following example works on a 32-bit system but fails on a 64-bit system, because a 32-bit integer cannot hold a 64-bit pointer. For example, don’t do this:
  ```
  unsigned int i, *ptr;
  i = (unsigned) ptr;
  ```
• Do not use a pointer to store an int. For example, don’t do this:
  ```
  int *ptr;
  int i;
  ptr = (int *) i;
  ```
• In cases where unsigned and signed 32-bit integers are mixed in an expression and assigned to a signed long, cast one of the operands to its 64-bit type. This will cause the other operands to be promoted to 64-bits and no further conversion is needed when the expression is assigned. Another solution is to cast the entire expression such that sign extension occurs on assignment. For example, consider the problem caused by the following:
long n;
int i = -2;
unsigned k = 1;
n = i + k;

Arithmetically, the result should be -1 in the expression shown in bold above. But since the expression is unsigned, no sign extension occurs. The solution is to cast one of the operands to its 64-bit type (as in the first line below) or cast the entire expression (as in the second line below):

\[
\begin{align*}
n &= (\text{long}) i + k; \\
n &= (\text{int}) (i + k);
\end{align*}
\]

Numeric constants

Hexadecimal constants are commonly used as masks or specific bit values. Hexadecimal constants without a suffix are defined as an unsigned int if it will fit into 32-bits and if the high order bit is turned on.

For example, the constant 0xFFFFFFFFL is a signed long. On a 32-bit system, this sets all the bits, but on a 64-bit system, only the lower order 32-bits are set, resulting in the value 0x00000000FFFFFFFF.

If you want to turn all the bits on, a portable way to do this is to define a signed long constant with a value of -1. This turns all the bits on since twos-compliment arithmetic is used:

\[
\begin{align*}
\text{long } x &= -1L;
\end{align*}
\]

Another problem that might arise is the setting of the most significant bit. On a 32-bit system, the constant 0x80000000 is used. But a more portable way of doing this is to use a shift expression:

\[
\begin{align*}
1L &\ll ((\text{sizeof(long)} \times 8) - 1);
\end{align*}
\]

Endianism

Endianism refers to the way in which data is stored, and defines how bytes are addressed in integral and floating point data types.

Little-endian means that the least significant byte is stored at the lowest memory address and the most significant byte is stored at the highest memory address.

Big-endian means that the most significant byte is stored at the lowest memory address and the least significant byte is stored at the highest memory address.

Table 3 shows a sample layout of a 64-bit long integer.

Table 3. Layout of a 64-bit long int

<table>
<thead>
<tr>
<th>Low address</th>
<th>High address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little endian</td>
<td></td>
</tr>
<tr>
<td>Byte 0</td>
<td>Byte 7</td>
</tr>
<tr>
<td>Byte 1</td>
<td></td>
</tr>
<tr>
<td>Byte 2</td>
<td></td>
</tr>
<tr>
<td>Byte 3</td>
<td></td>
</tr>
<tr>
<td>Byte 4</td>
<td></td>
</tr>
<tr>
<td>Byte 5</td>
<td></td>
</tr>
<tr>
<td>Byte 6</td>
<td></td>
</tr>
</tbody>
</table>
For example, the 32-bit word 0x12345678 will be laid out on a big endian machine as follows:

Table 4. 0x12345678 on a big-endian system

<table>
<thead>
<tr>
<th>Memory offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory content</td>
<td>0x12</td>
<td>0x34</td>
<td>0x56</td>
<td>0x78</td>
</tr>
</tbody>
</table>

If we view 0x12345678 as two half words, 0x1234 and 0x5678, we would see the following in a big endian machine:

Table 5. 0x12345678 as two half words on a big-endian system

<table>
<thead>
<tr>
<th>Memory offset</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory content</td>
<td>0x1234</td>
<td>0x5678</td>
</tr>
</tbody>
</table>

However, on a little endian machine, the word 0x12345678 will be laid out as follows:

Table 6. 0x12345678 on a little-endian system

<table>
<thead>
<tr>
<th>Memory offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory content</td>
<td>0x78</td>
<td>0x56</td>
<td>0x34</td>
<td>0x12</td>
</tr>
</tbody>
</table>

Similarly, the two half-words 0x1234 and 0x5678 would look like the following:

Table 7. 0x12345678 as two half words on a little-endian system

<table>
<thead>
<tr>
<th>Memory offset</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory content</td>
<td>0x5678</td>
<td>0x1234</td>
</tr>
</tbody>
</table>

The following example illustrates the difference in byte order between big endian and little endian machines.

The C program below will print out "Big endian" when compiled and run on a big endian machine, and "Little endian" when compiled and run on a little endian machine.

Listing 2. Big endian vs. little endian

```c
#include <stdio.h>
main () {
  int i = 0x12345678;
  if (*(char *)&i == 0x12)
    printf("Big endian\n");
  else if (*(char *)&i == 0x78)
    printf("Little endian\n");
  }
```

Endianness is important when:

- Bit masks are used
• Indirect pointers address portions of an object

We have bit fields in C and C++ that help to deal with endian issues. I recommend the use of bit fields rather than mask fields or hexadecimal constants. There are several functions that are used to convert 16-bit and 32-bit from "host-byte-order" to "net-byte-order." For example, htonl (3), ntohl (3) are used to convert 32-bit integers. Similarly, htons (3), ntohs (3) are used for 16-bit integers. However, there is no standard set of functions for 64-bit. But Linux provides the following macros on both big and little endian systems:

- bswap_16
- bswap_32
- bswap_64

**Type definitions**

I recommend that you do not code your applications with the native C/C++ data types that change size on a 64-bit operating system, but rather use type definitions or macros that explicitly call out the size and type of data contained in a variable. Some type definitions help make the code more portable.

- ptrdiff_t:
  A signed integer type that results from subtracting two pointers.
- size_t:
  An unsigned integer and the result of the sizeof operator. This is used when passing parameters to functions such as malloc (3), and returned from several functions such as fred (2).
- int32_t, uint32_t etc.:
  Define integer types of a predefined width.
- intptr_t and uintptr_t:
  Define integer types to which any valid pointer to void can be converted.

**Example 1:**

The 64-bit return value from sizeof in the following statement is truncated to 32-bits when assigned to bufferSize.

```c
int bufferSize = (int) sizeof (something);
```

The solution is to cast the return value using size_t and assign it to bufferSize declared as size_t as shown below:

```c
size_t bufferSize = (size_t) sizeof (something);
```

**Example 2:**

On a 32-bit system, int and long are of the same size. Due to this, some developers use them interchangeably. This can cause pointers to be assigned to int and vice-versa. But on a 64-bit system, assigning a pointer to an int causes the truncation of the high-order 32-bits.
The solution is to store pointers as pointer types or the special types defined for this purpose, such as `intptr_t` and `uintptr_t`.

**Bit shifting**

Untyped integral constants are of type (unsigned) int. This might lead to unexpected truncation while shifting.

For example, in the following code snippet, the maximum value for `a` can be 31. This is because the type of `1 << a` is int.

```c
long t = 1 << a;
```

To get the shift done on a 64-bit system, `1L` should be used as shown below:

```c
long t = 1L << a;
```

**Formatting strings**

The function `printf(3)` and related functions can be a major source of problems. For example, on 32-bit platforms, using `%d` to print either an int or a long will usually work, but on 64-bit platforms, this would truncate a long to its least significant 32-bits. The proper specification for a long is `%ld`.

Similarly, when a small integer (char, short, int) is passed into `printf(3)`, it will be widened to 64-bits and the sign will be extended if appropriate. In the example below, the `printf(3)` assumes that a pointer is 32-bits.

```c
char *ptr = &something;
printf("%x\n", ptr);
```

The above code snippet will fail on 64-bit systems and will display only the lower 4 bytes.

The solution for this is to use the `%p` specification as shown below, which will work fine on both 32-bit and 64-bit systems.

```c
char *ptr = &something;
printf("%p\n", ptr);
```

**Function parameters**

There are a few things that you need to remember while passing parameters to functions:

- In the case where the data type of the parameter is defined by a function prototype, the parameter is converted to that type according the standard rules.
- When the type of the parameter is not specified, the parameter is promoted to the larger type.
- On a 64-bit system, integral types are converted to 64-bit integral types, and single precision floating point types are promoted to double precision.
• If a return value is not otherwise specified, the default return value for a function is int.

The problem arises when passing the sum of signed and unsigned ints as long. Consider the following case:

Listing 3. Passing the sum of signed and unsigned ints as long

```c
long function (long l);

int main () {
    int i = -2;
    unsigned k = 1U;
    long n = function (i + k);
}
```

The above code snippet will fail on 64-bit systems, because the expression (i + k) is an unsigned 32-bit expression, and when promoted to a long, the sign doesn’t extend. The solution is to cast one of the operands to its 64-bit type.

There is another problem on register-based systems where registers are used to pass parameters to functions rather than the stack. Consider the following example:

```c
float f = 1.25;
printf ("The hex value of %f is %x", f, f);
```

On a stack-based system, the appropriate hexadecimal value is printed. But on a register-based system, the hexadecimal value is read from an integer register, not the floating point register.

The solution is to cast the address of the floating point variable to a pointer to an int, which is then de-referenced as shown below:

```c
printf ("The hex value of %f is %x", f, *(int *)&f);
```

**Conclusion**

Major hardware vendors have recently expanded their 64-bit offerings because of the performance, value, and scalability that 64-bit platforms can provide. The constraints of 32-bit systems, particularly the 4GB virtual memory ceiling, have spurred companies to consider migrating to 64-bit platforms. Knowing how to port applications to comply with a 64-bit architecture can help you write portable and efficient code.
Related topics

- **64-Bit Programming Models: Why LP64?** provides more detail on the various 64-bit programming models and argues for LP64.
- Read about the **Year 2038 problem** that 32-bit systems have in Wikipedia.
- "**Porting Intel applications to 64 bit Linux PowerPC**" gives advice on some of issues to consider when porting Linux from IA32 to PowerPC.
- With **IBM trial software**, available for download directly from developerWorks, build your next development project on Linux.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)