Porting applications to Linux for System z

Technical hints and tools to help simplify porting applications to System z

Wolfgang Gellerich
June 08, 2012
(First published May 28, 2008)

Server consolidation based on Linux® for IBM System z™ offers advantages, but moving existing applications requires some specialized knowledge. In this article, get general advice on how to organize your porting project, including technical details on mainframe virtualization, byte-ordering, and address calculation specific to System z. This article also covers how development tools (compiler, linker, debugger) are supported on System z, and introduces IBM's free-of-charge Migration Kit for Solaris OS to Linux.

08 Jun 2012 - Per author request, removed sentence that no longer applies in second paragraph of 31-bit addresses.

Consolidating servers in an environment based on Linux for System z has many advantages, but as you know, there is no free lunch—not even for penguins. Moving existing applications and servers to a Linux for System z environment requires specific knowledge and some effort.

This article gives you an overview of application porting rather than a detailed guide. References to more detailed information are included for those who wish to go deeper.

The Migration Kit for Solaris OS to Linux contains several interactive tools to help you port applications, as well as some technical documentation (the Guide to Application Porting from Solaris OS to Linux and the IBM Redbook® Solaris to Linux Migration: A Guide for System Administrators).

Organizing your porting project

Before you actually modify anything, look for information about the differences between Linux for System z and your current operating system and hardware platform. All issues mentioned in this article might be relevant (byte-ordering, the options passed to the compiler, and so forth). In addition to the references in this article, there are some excellent sources of information providing lots of details about the differences of particular operating systems and Linux. The book UNIX® to
Linux Porting covers general issues of taking application programs from proprietary UNIX variants such as AIX®, Solaris, and HP-UX. It also gives a detailed introduction to Linux development tools. (See Related topics for links to this and other helpful documents.)

When you begin to modify your application, be sure not to make too many changes in each step. Before changing the source code itself, modify the build environment. Check whether the development tools needed under Linux are available on your current platform (the compiler collection GCC and GNU make, to name the most important ones). While any search engine will help you, the best single sources of information are probably the GNU and the GCC homepages.

Next, build your application on your current platform, but use the new build tools and run a full test cycle. Only then should you move the source code to Linux and make modifications, as appropriate. The remaining steps are the same as for any other software project:

- Do extensive testing.
- Fix any bugs you find.

Several Linux debugging tools support this step and help with performance tuning.

Linux for IBM System z

Fortunately, the conceptual issues to consider when porting application programs to Linux running on an IBM System z mainframe are relatively few. The official interfaces of the Linux kernel are designed to be platform-independent.

One important difference, though, concerns virtualization. Personal computers usually run native operating systems controlling the entire computer system. In contrast, IBM System z computers support the concept of virtualization. Every operating system executes in a virtual environment, and there might be (and normally are) several instances of different operating systems running at the same time. This virtual environment also produces some administration concerns.

Overall structure of a System z mainframe

On an IBM System z mainframe, Linux is always executed in a virtual environment. The mainframe is divided into logical partitions (LPARs). Each LPAR can either directly execute an operating system, such as the Linux kernel, or execute a number of Virtual Machines (VM) images as shown in Figure 1. Using VM is an alternate means you can create a virtual environment for Linux execution.
Figure 1. Overall structure of a System z computer running several LPARs and VMs

This strategy has several advantages, ranging from dynamic load balancing, capacity planning, and security issues to high-speed virtual networking between Linux images. Virtual networking allows the communicating parties to behave as if they were sending and receiving data via a real network, but the data transfer is much faster and actually performed by moving data within the mainframe's main memory.

However, the fact that Linux for System z runs in a virtual environment does not affect the actual task of porting an application; therefore, I won't go into more detail here. The book *Linux on the Mainframe* provides an excellent overview and introduction. In addition, IBM Redbooks provide a good deal of valuable and up-to-date information; these are available online. (See Related topics for links.)

**Naming conventions**

Over the years, the IBM System z models changed names several times. Linux for System z was first released when the system's name was System/390®. The platform is still called S/390® as far as Linux is concerned.

If code needs to be platform-dependent, use the following predefined symbols in preprocessor expressions:

- `__s390__` when compiling for System z in 31-bit or 64-bit mode or when compiling for S/390
- `__s390x__` when compiling for System z in 64-bit mode

Probably the most important difference concerns address ranges. Today's System z models are 64-bit architectures, while older models provided 31-bit addresses only. Whenever it is important to distinguish between these two address ranges, the convention is that `s390` denotes a 31-bit platform, while `s390x` denotes a 64-bit platform. The addressing mode has its own set of implications as discussed here. Note that one of the bits in the 32-bit architecture defines the architecture as 31-bit addressing mode; hence our use of the term 31-bit mode.

**31-bit addresses**

Note that 31-bit addresses actually do not have a usable 32nd bit; therefore, any address such as:
$0x80000000 + n$

maps onto:

$0x00000000 + n$

Therefore, applications casting integer values to pointer types might need attention. Also, absolute addresses are most likely platform-dependent, and any code actually affected by this situation probably needs changes as well.

Of course, pointer values created using the C language's "official" features will always be correct. Therefore, the compiler might not clear the leftmost bit, causing the following expression to yield false:

$\texttt{(void*)0x80000000 == (void*)0x00000000}$

A related problem is caused by applications that try to load code at an absolute address by calling `mmap` with the flag `MAP_FIXED`. Such usage of absolute addresses is not portable and might need changes.

**Numbering conventions for bits**

In Linux it is common to denote individual bits of a byte or word by the exponent they would have when being interpreted as a binary number. The bits of a 32-bit word, for example, would be denoted as:

```
| 31 | 30 | 29 | ... | 2 | 1 | 0 |
```

However, documentation related to IBM System z counts bits from left to right:

```
| 0 | 1 | 2 | ... | 29 | 30 | 31 |
```

In this example, the right-most bit with binary value $2^0$ is called bit 31. On a 64-bit system, this bit is called bit 63.

**Standard data type sizes**

Table 1 shows the standard C data types in bytesizes:

<table>
<thead>
<tr>
<th>Table 1. Standard C data types in bytesizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>pointer</td>
</tr>
</tbody>
</table>
The alignment is always the same as the type size; that is, a variable of type `int` is four bytes long and will be stored at a 4-byte boundary. Table 2 defines the standard data types of the Linux kernel.

### Table 2. Standard data types of the Linux kernel

<table>
<thead>
<tr>
<th>Mode</th>
<th>gid_t</th>
<th>mode_t</th>
<th>pid_t</th>
<th>uid_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-bit mode</td>
<td>unsigned int</td>
<td>unsigned int</td>
<td>int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>64-bit mode</td>
<td>unsigned int</td>
<td>unsigned int</td>
<td>int</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>

The **ELF Application Binary Interface Supplement** document details data layout.

## Endianness

**Endianness** refers to the order in which the bytes of a multi-byte word are stored in memory. There are several ways of storing a 32-bit binary value such as `4A3B2C1D`, including:

- The big-endian scheme stores the most significant byte (MSB) first, yielding `0x4A 0x3B 0x2C 0x1D`.
- The little-endian scheme stores the least significant byte (LSB) first, yielding `0x1D 0x2C 0x3B 0x4A`.

Big endian architectures include Sun SPARC and IBM System z. All processors of the Intel x86 family are little endian.

Unfortunately, many programs rely on the way data is organized in memory. This incompatibility between platforms is particularly dangerous as it might cause programs to return wrong results rather than causing a compile-time error. This is a serious matter. Code patterns that might be affected by different endianness include:

- Byte-oriented processing data.
- Data structures accessed by assembler code. You must have an in-depth understanding of the data layout when rewriting those parts of the source code that are implemented in assembler.
- Data received via a network.
- Type conversion using "dirty hacks" with pointers. Note that even seemingly simple code like the following example depends on the machine's byte order:
```c
int x = 1;
if (*(char *)&x == 1)
    printf("little endian \n");
else
    printf("big endian \n");
```

- Passing actual arguments of a too small size to a function. This situation is similar to the previous pointer example.
- Type conversion (mis-)using union data types.

The Endianness Checking Tool provided as part of the Migration Kit for Solaris OS to Linux helps locate those parts of your code that depend on endianness. The input for this tool is the application's source code and the resulting binaries compiled with certain compiler options enabled. The tool's operation guide provides detailed instructions. A typical found instance looks like this:

```
/test/src/init.c - Line 199: E30001
Variable/parameter size mismatch arg 2 size 4 in call to mystrncpy.
(Defined in /test/src/init.c at line 190 size 1)
```

The Endianness Checking Tool is not specific to Solaris; you can apply it to any C source code.

If you need to support multiple platforms with different endianness, you can include the Linux kernel header file asm/byteorder.h, which defines one of the two symbols depending on the machine's endianness: __BIG_ENDIAN and __LITTLE_ENDIAN. The header files found in /usr/src/linux/include/linux/byteorder/ provide macros for converting between big and little endian representations. If the order is the same, these macros evaluate to a no-op. Otherwise, they return the converted value.

Similar functions exist to handle byte-order issues related to data transferred via network. These functions are declared in /usr/include/netinet/in.h and convert values between network- and host-byte order. Note that networks usually use big-endian byte order, so an application running on System z does not need conversion here. In any case, it is always a good programming practice to insert appropriate conversion routines.

**ASCII and EBCDIC issues**

For historical reasons, mainframe operating systems like z/OS® and z/VM® use the EBCDIC character set. Linux for System z is completely ASCII-based. This means that porting data or applications from Linux for other platforms will not create problems. On the other hand, interfacing Linux for System z with data or programs running under control of one of the EBCDIC-based operating systems requires a conversion.

The Linux tool named "recode" supports the conversion of files between different character sets. The current version recognizes approximately 280 character sets, including 19 EBCDIC variants with national character support. If a program needs to perform such conversions, use the `iconv` function declared in /usr/include/iconv.h
Variable argument lists

Normally, C application programs do not need to know how variable argument lists are implemented. Some code might, however, use non-portable assignments, which will fail on Linux for System z.

Linux for System z implements the `va_list` type as a `struct`. Therefore, a normal assignment like this:

```c
va_list va1, va2;
va_start (va1);
va2 = va1;       /* This assignment will not work. */
```

from one variable of this type to a second will not work as expected. Rather, use the `__va_copy()` macro for copying, as in `__va_copy (va2, va1);`.

Platform-specific libraries and calls

All operating systems provide some interface to application programs and usually have a large number of associated libraries. Some of them might not be available under Linux or might have a different name. Information about such differences is, for example, is provided in *Unix to Linux Porting*.

When porting applications from Solaris OS to Linux, the interactive Source Checking Tool (provided in IBM's Migration Kit for Solaris OS to Linux) can help. The tool recognizes approximately 3,800 different calls to the Solaris OS API, including files specific to Solaris OS, as well as the pragmas (or directives) of the Sun compiler.

Figure 2 shows input to the tool as some C- or C++-written Solaris OS application source code that can be selected interactively.
The tool then detects all calls that need to be changed and highlights them (Figure 3).

**Figure 3. The Source Checking Tool highlights any library calls that need to be modified**

Linux alternatives and further technical information are available interactively or can be included as comments to the source files so developers can use the information without having to run the tool (Figure 4).
Fortunately, there are also some standards and general conventions. For example, while developing the Source Checking Tool, the team noted that 46 percent of all calls were identical. One example where no differences exist are the mathematical functions found in math.h.

**Compiling, linking, and debugging**

Porting an application program includes using the Linux standard compiler GCC and some associated tools. They all provide some support for System z. A review of the performance improvements achieved between 1999 and 2005 suggests that it is advantageous to exploit the System z-specific optimization options and the options enabling optimization for a specific processor model (see Figure 5).

**Figure 5. Performance increase due to improved compiler optimization**
All data provided in Figure 5 came from the latest System z model available in the particular year and is normalized. Overlapping measurements have been used to scale when the measurements were taken on a new System z model. The *IBM Systems Journal* article "Contributions to the GNU Compiler Collection" details how the measurement testing was conducted.

### Compiler options

GCC and the binary utilities provide some options supporting Linux for System z. A detailed description of all GCC options is in *Using the GNU Compiler Collection (GCC)*. Because this manual includes a section devoted to System z-specific options, we need only provide an overview here.

Also, it is likely that additional compiler options will be provided for new System z models, and improvements might be offered for existing models. Check the most recent version of the manual for details. The System z-specific code optimization options include:

- `-march=cpu-type`: Exploit all instructions provided by `cpu-type`, including those not available on older models. In general, code compiled with this option will not execute on older CPUs. See the GCC site for a list of supported CPU types.

- `-mtune=cpu-type`: Schedule instructions to best exploit the internal structure of `cpu-type`. This option will not introduce incompatibility with older CPUs, but code tuned for a different CPU might execute more slowly.

- `-mzarch` and `-mesa`: Generate code exploiting the instruction set of the ESA/390 or the z/Architecture®, respectively. See the GCC site for default values and interaction with other options.

- `-m64` and `-m31`: Controls whether the generated code is compliant with the Linux for S/390 ABI (`-m31`) or the Linux for System z ABI (`-m64`). For zSeries, refer to *z/Architecture Principles of Operation* and *Linux for zSeries ELF Application Binary Supplement*, or for S/390 refer to *ESA/390 Principles of Operation* and *LINUX for S/390 ELF Application Binary Interface Supplement* (all in *Related topics*) for details about the application binary interfaces (ABIs).

- `-msmall-exec` and `-mno-small-exec`: A switch to optimize jump instructions in executables with a total size not exceeding 64KB.

Some options control and optimize stack growth:

- `-mwarn-framesize=framesize` and `-mwarn-dynamicstack`: These options cause a compile-time check whether a function exceeds a given stack frame size or uses dynamically sized stack frames. These options help in cases when stack space is rare, as is the case in the Linux kernel, or where application programs fail because of a stack overflow.

- `-mstack-guard=stack-guard` and `-mstack-size=stack-size`: These options also help debugging stack size problems. The checks are, however, performed during runtime by executing some extra code inserted into the binary.

- `-mpacked-stack` and `-mno-packed-stack`: These options control whether the stack frame uses a space-optimized, dense-packing scheme for register save slots. Refer to the GCC manual for details concerning the compatibility with other options and for call-compatibility with binary code generated by pre-3.0 versions of GCC.
-mbackchain and -mno-backchain: These options control whether the address of the caller's frame is stored as so-called "backchain" pointer into the callee's stack frame. Refer to the GCC manual for details concerning the compatibility with other options and for details related to debugging.

On Linux for System z, there is an important difference compared to other systems concerning shared libraries: -fpic and -fPIC. Both options cause the compiler to generate position-independent code (PIC) for use in shared, re-entrant libraries. On System z, -fpic causes the global offset table to be rather small; therefore, you should use -fPIC for large shared libraries.

If the linker reports the error message "relocation overflow," check whether the project uses the -fpic option. Changing it to -fPIC might fix it. Note that you must use the same variant of this option for all compiler runs related to the same project.

The options -mfused-madd, -mno-fused-madd, -mhard-float, and -msoft-float control the usage of floating point operations. See the z/Architecture or ESA/390 Principles of Operation for a complete description of the System z instruction set and architecture. Several technical papers (see Related topics) describe the internals of how GCC generates code for System z:

- "The GNU 64-bit PL8 compiler: Towards an open standard environment for Firmware development."
- "Porting GCC to the IBM S/390 Platform."
- "Contributions to the GNU Compiler Collection."

Debugging

Linux provides a rich collection of debugging tools. The GNU debugger (GDB) is quite powerful; the Data Display Debugger is a graphical front end for GDB and even supports difficult tasks like the visualization of pointer-linked data structures.

A number of debugging tools focus on analyzing bugs related to memory references. Electric Fence is a useful tool and available on many platforms. Its capability is, however, limited to dynamically allocated memory.

Another extremely powerful tool is VALGRIND. The outstanding property of this program is that you can debug any binaries even if they have not been compiled with debug options enabled or even if the source code is not available. This is achieved by an approach called dynamic code instrumentation.

An alternative is to use the MUDFLAP option that the GCC provides. It causes the compiler to insert checks whenever a critical memory reference takes place. These checks are then executed during program runtime. Because the compiler had access to the full program source code, the MUDFLAP diagnostics are very valuable.

A powerful feature is available for Linux running under VM. VM's TRACE command offers a convenient way to debug the entire Linux system.
For an introduction to debugging, refer to *Linux on the Mainframe*; it includes a detailed description of debugging under Linux in the file `/usr/src/linux/Documentation/s390/debugging390.txt.

For debugging more difficult problems, information about register usage, stack frame layout, and other conventions is found in the *ELF Application Binary Interface Supplement*. Additionally, *Porting GCC to the IBM S/390 Platform* describes conventions used when compiling for System z.

I wish you success with your Linux porting projects.
Related topics

- Need a **UNIX porting guide**? This site has them all, including Red Hat's "Solaris to Linux Porting Guide" (both the [PDF](#) and HTML versions).
- You can download software and knowledge about the **GNU operating system here**. You'll also find the
  - GNU Compiler Collection (GCC).
  - GCC manual of online docs.
  - GNU Project Debugger (gdb).
  - Data Display Debugger.
- **Electric Fence (efence)** stops your program on the exact instruction that overruns (or underruns) a `malloc()` memory buffer.
- **Valgrind** is an instrumentation framework for building dynamic analysis tools that can automatically detect memory management and threading bugs and profile your programs in detail.
- **Linux on the Mainframe** (Prentice-Hall, 2003) is a comprehensive guide to the fastest growing trend in IT -- a Linux mainframe -- with in-depth coverage of virtualization, deployment, data management, debugging, security, systems management, application porting, and more.
- **IBM Redbooks** include such gems as "**Solaris to Linux Migration: A Guide for System Administrators**" (2006) with a goal of providing a technical reference for IT systems administrators in organizations that are considering a migration from Solaris to Linux-based systems.
- **Linux Kernel Development** (SAMS Publishing, 2003) details the design and implementation of the Linux kernel, presenting the content in a manner that is beneficial to those who wish to write and develop kernel code.
- "**Contributions to the GNU Compiler Collection**" (*IBM Systems Journal*, 2005) is a review several of IBM's contributions to the compiler, including a code generator for the IBM zSeries and many optimizations like the interblock instruction scheduler, software pipeliner, and vectorizer.
- See all **Linux tips** on developerWorks.
- In the developerWorks **Linux zone**, find more resources for Linux developers, including **Linux tutorials**, as well as our readers' most recent favorite **Linux articles and tutorials**.
- Get more information on **IBM System z**.
- With **IBM trial software**, available for download directly from developerWorks, build your next development project on Linux.