Guide to porting from Solaris to Linux on x86

Migrate your projects from Solaris on 32- or 64-bit SPARC to Linux on x86

Skill Level: Intermediate

Ajay Sood (r1sood@in.ibm.com)
Staff Software Engineer
IBM Global Services, Bangalore, India

29 Apr 2005

Solaris is considered one of the closest flavors of UNIX® to Linux™, but for migration purposes, there can be differences between the two in the areas of memory mapping, threading, or natural language support (to name just a few). This porting guide gives you advice on planning for the port to Linux/x86, and helps you understand the differences in the development environment and architecture.

Among the flavors of UNIX, Solaris is considered to be the closest to Linux, so before starting a port of large Unix-based application to Linux, the operating system-dependent code is generally picked up from Solaris. Even so, differences can arise in the areas that depend on the architecture, memory maps, threading, or some specific areas like system administration or natural language support.

This article discusses these differences and gives you comparisons to guide you through your migration from Solaris running on 32-/64-bit SPARC architectures to Linux running on x-86 architecture. For Solaris, the discussion is based on Version 8 and later. For Linux, the discussion focuses on the distributions available on x-86 processor-based servers: SUSE LINUX Enterprise Server 9 and Red Hat Enterprise Linux AS V3 or V4.

This article covers:

- Planning for the port

- Development environment (compilers, make utilities, and so on)
- Architecture- or system-dependent differences

Planning for the port

These six steps provide a complete roadmap for successful migration from Solaris on SPARC to Linux on x86. If you've ever ported applications from one operating system to another, these steps probably sound familiar:

1. Preparation
2. Environment and makefile changes
3. Compiler fixes
4. Testing and debugging
5. Performance tuning
6. Packaging and distribution

Step 1. Prepare

The key to proper preparation is to study the differences in such areas as:

- System calls
- Filesystem support
- Machine-dependent code
- Threading
- Memory maps
- System calls
- Endianness

While porting the application, be sure to ensure that the relevant third-party packages are available on the target platform. For 32-bit applications, consider if it is necessary to migrate to a 64-bit version. Also, decide which compiler to use on the target platform. On x86-based Linux platforms, gcc is used as the compiler.

Step 2. Environment and makefile changes

In this step, you'll set up the development environment, which includes deciding on environment variables, making changes to makefiles, and performing any other alterations necessary to the environment. At the end of this stage, you should be ready to start building your application.

This step may require several iterations before you're ready to move to the next step.

Step 3. Compile and build

At this stage, you will fix compiler errors, linker errors, and the like. This step may require several iterations before a clean build is produced.

Step 4. Test and debug

After the application is successfully built, test it for runtime errors. Some areas to look out for while testing are client/server communication, data-exchange formats, data-code conversions like conversion from single bytecode pages to multi bytecode pages, and persistent storage.

Step 5. Performance tuning

Now the ported code is running on the target platform. Monitor performance to ensure the ported code performs as expected; if not, performance tuning is needed.

Two good tools for performance analysis are Performance Inspector and OProfile. Performance Inspector provides a set of tools to identify performance problems in the application on Linux. From kernel 2.6 on, OProfile is the suggested tool for profiling the code. OProfile is also available for RHEL4. (See Resources for more information on these tools.)

Step 6. Packaging and distribution

Will you have to distribute the resulting code to others? If so, decide on your packaging method.

Linux provides several ways to package your application, such as a tarball, self-installing shell script, or RPM. RPM is the package-management system widely used on Linux. Solaris uses pkgadmin as its package manager.

The format of the package-specification template files used by pkgadmin in Solaris is different from the spec file used by RPM -- translating packaging information from template file into spec files requires a substantial effort. Using a software package like the InstallShield for Multiplatforms (ISMP) could deliver common packaging software across both operating systems and reduce the porting effort. By using ISMP, application developers can use a common spec file across platforms.

Development environments

Now let's look at some of the differences in the development environments of Solaris and Linux, including the following:

- Makefiles
- Compiler and linker options
- Considerations in moving from 32 to 64 bits

GNU Make versus Solaris Make

If you use Solaris Make in the source platform, you need to modify your makefile in order to use GNU Make on Linux. Chapter 6 of the *AIX 5L Porting Guide* (see Resources for a link) gives detailed information on the differences between the two.

Compiler and linker options

As mentioned earlier, a compiler available for Linux on x86 is GNU GCC. Following is a list of widely used compiler options for the SUN Studio C/C++ compiler and the equivalent options for the GNU GCC.

Table 1. Solaris-to-Linux equivalent compiler options

SUN Studio	GNU GCC	Description
-#	-v	Instructs the compiler to report information on the progress of the compilation.
-###	-###	Traces the compilation without invoking anything.
-Bstatic	-static	Causes the link editor to look only for files named libx.a.
-Bdynamic	(Default)	Instructs the link editor to look for files named libx.so and then libx.a when given the lx option.
-G	-shared	Produces a shared object rather than a dynamically

		linked executable.
-xmemalign	-malign-natural,	Specifies the maximum assumed memory alignment and the behavior of misaligned data accesses.
-xO1, -xO2, -xO3, -xO4, -xO5	-O, -O2, -O3	Optimizes code at a choice of levels during compilation. Something about O2.
-KPIC	-fPIC	Generates position-independent code for use in shared libraries (large model).
-Kpic	-fpic	Generates position-independent code for use in shared libraries (small model).
-mt	-pthread	Compiles and links for multithreaded code.
-R dirlist	-Wl, -rpath, dirlist	Builds dynamic library search path into the executable file.
-xarch	-mcpu, -march	Specifies the target architecture instruction set.

Table 2 explains the difference in levels for the C compilers for the different flavors of Linux.

Table 2. Differences in levels of C compilers

Operating system	Kernel level	GCC level	Glibc level	Gnu binutils level	JDK level
SLES9	2.6	3.3.3	2.3.3	2.15.90	1.4.2*
RHEL3	2.4	3.2.3	2.3.2	2.14.90	1.4.2
RHEL4	2.6	3.4.3	2.3.4-2	2.15.92.0.2-10.EL4	1.4.2*
Solaris 10	Solaris 10	3.3.2	2.2.3		1.4.2

Notes: *LinuxThreads is not supported with the 64-bit version of the IBM JDK 1.4.2 for Linux on x86. Only the NPTL 64-bit threading libraries are supported for Java.

Also, if you build your C++ applications on SLES9 and deploy them on RHEL4, you will need to use the compat library packaged as compat-libstdc++-33-3.2.3-47.3.i386.rpm. This compat is required because of the different level of ABI compatibility between SLES9 and RHEL4. While SLES9 follows the LSB 1.3 standards, RHEL4 follows the LSB 2.0 standards.

Considerations for moving from 32 bits to 64 bits

To the application from a 32-bit environment to a 64-bit environment, you have two alternatives:

- Making the application 64-bit tolerant
- Exploiting the 64-bit capabilities of the target platform

First, enable the application for a 64-bit environment and then try to exploit the 64-bit capabilities of the target platform.

Even though the 32-bit applications normally run fine on a 64-bit target, enabling the application for 64-bit environment is important because some of the relevant third-party products will start operating in 64-bit modes but do not ship the 32-bit versions of their library objects.

While migrating to a 64-bit environment, you need to decide if support for the 32-bit version of the product should be continued. If both of the versions have to be supported, allow appropriate time for packaging and testing.

Exploiting the 64-bit capabilities should be the next step after becoming 64-bit tolerant. You should carefully weigh the advantages of exploiting the 64-bit capabilities before starting the exercise.

The topic of migrating applications from 32- to 64-bit environments is covered extensively in other publications (such as Chapter 3 of the *AIX 5L Porting Guide* listed in Resources).

Architecture- and system-specific differences

Now, let's look at the architecture- and system-specific differences between Solaris and Linux on x86, including base data types, settings for kernel parameters, architecture-dependent differences, endianness, system calls, signals, data types, and threading libraries.

Resources offers a practical checklist of things to consider while doing a large porting exercise.

Base data type and alignment

There are two different classes of data types available on a system: base data types and derived data types.

Base data types are all data types defined by the C and C++ language specification. Table 3 compares base data types for Linux on x86 and Solaris on SPARC.

Table 3. Comparing Linux and Solaris base data types

	Linux (x86)		Solaris (SPARC)	
Base type	ILP32 (sizeof) (in bytes)	LP64 (for IA64-based systems) (sizeof) (in bytes)	ILP32 (sizeof) (in bytes)	LP64 (sizeof) (in bytes)
char	1	1	1	1
short	2	2	2	2
Int	4	4	4	4
float	4	4	4	4
long	4	8	4	8
pointer	4	8	4	8
long long	8	8	8	8
double	8	8	8	8
long double	12	16	16	16

When porting applications between platforms or between 32-bit and 64-bit modes, you need to take into account the differences between alignment settings available in the different environments to avoid possible degradation in performance and data corruption.

Table 4 shows the alignment values in bytes for Linux on x86.

Table 4. Alignment values (in bytes) for Linux on x86

Data type	Linux IA-32 (ILP32)	Linux IA-64 (LP64)
Bool	1	1
Char	1	1
wchar_t	4	4

int	4	4
Short	2	2
long	4	8
long long	8	8
Float	4	4
Double	4	8
long double	4*	16

Note: Alignment depends the compiler switches being used. These switches control the size of the "long double" type. The i386 application binary interface specifies the size to be 96 bits, so `-m96bit-long-double` is the default in 32-bit mode. Modern architectures (Pentium and newer) would prefer "long double" to be aligned to an 8- or 16-byte boundary. In arrays or structures conforming to the ABI, this would not be possible. So specifying a `-m128bit-long-double` will align "long double" to a 16-byte boundary by padding the "long double" with an additional 32-bit zero.

System-derived data types

A derived data type is a derivative or structure of existing base types or other derived types. *System-derived data types* can have different byte sizes depending on the employed data model (32-bit or 64-bit) and the hardware platforms.

Table 5 shows some of the derived data types on Linux that are different from those on Solaris.

Table 5. Derived data types on Solaris and Linux

OS	gid_t	mode_t	pid_t	uid_t	wint_t
Solaris ILP32 l	long	unsigned long	long	long	long
Solaris LP64	int	unsigned int	int	int	int
Linux ILP32	unsigned int	unsigned int	int	unsigned int	unsigned int
Linux LP64	unsigned int	unsigned int	int	unsigned int	unsigned int

Examples of machine-dependent code

Machine dependent-code is needed in these cases:

- Getting the stack pointer

- Atomic locking

Getting the stack pointer on Linux-x86 can be implemented as:

Listing 1. Getting the stack pointer on Linux-x86

```
int get_stack(void **StackPtr)
{
  *StackPtr = 0;

  __asm__ __volatile__ ("movl %%esp, %0": "=m" (StackPtr) );

 return(0);
}
```

On Solaris, this sample code allows you to get the stack pointer:

Listing 2. Getting the stack pointer on Solaris

```
        .section       ".text"
        .align  8
        .global my_stack
        .type   my_stack,2
my_stack:
        ! Save stack pointer through 1st parameter
        st      %sp,[%o0]
        ! Compute size of frame in return result reg
        retl
        sub     %fp,%sp,%o0
        .size  my_stack,(.-my_stack)
```

On Linux x86, you can use a compare and swap operation to implement atomic locking. For example, a sample implementation for compare and swap on Linux x86 can be the following:

Listing 3. Compare and swap on Linux x86

```
bool_t My_CompareAndSwap(IN int *ptr, IN int old, IN int new)
{

        unsigned char ret;

        /* Note that sete sets a 'byte' not the word */

        __asm__ __volatile__ (
                "  lock\n"
                "  cmpxchgl %2,%1\n"
                "  sete %0\n"
                : "=q" (ret), "=m" (*ptr)
                : "r" (new), "m" (*ptr), "a" (old)
                : "memory");

        return ret;

}
```

On Solaris SPARC, atomic operation for locking can be implemented as the following:

Listing 4. Atomic locking on Solaris

```
        .section        ".text"
        .align  8
        .global         My_Ldstub
        .type           My_Ldstub,2
My_Ldstub:
        ldstub          [%o0],%o0           ! Atomic load + set
        sll             %o0,24,%o0          ! Zero fill ...
        retl                                ! ... result register
        srl             %o0,24,%o0          ! ... and retrn
        .size           My_Ldstub,(.-My_Ldstub)
```

Byte ordering (endianness)

Since SPARC is big-endian and x86 is little-endian, you need to consider endianness issues.

Endianness issues become important when porting the pieces of the application that relate to client communication over a heterogeneous network, persistent data storage on the disk, product tracing (it is important that trace generated on SPARC gets formatted correctly on x86), and other related areas. The IA-64 Linux kernel uses little-endian by default, but allows for the possibility of using big-endian byte order.

System calls

Solaris system calls that use a different name or signature, or are not available on Linux, are listed in the "Guide to porting from Solaris to Linux on POWER". The following system calls are now available on RHEL4:

- `Waitid`
- `Putmsg`
- `putpmsg`
- `Getmsg`
- `getpmsg`

Curses library

Linux library functions for the curses library are closer to AIX than Solaris. For example, functions like `addwch` are not supported in Linux. Some functions calls like `mvchgat` are not available in Solaris. Much of the code could require a rewrite when you port the curses-related code from Solaris to Linux.

Terminal IO

The termio structure in Solaris is different from that in Linux. The termio structure has some extra fields in Linux with which you can mention the input and output speeds.

The definition of termio structure for Linux is in /usr/include/bits/termios.h, and for Solaris it is in /usr/include/sys/termios.h.

IOCTL

Options available for performing ioctl are different in Solaris and Linux. For example, to get the resource usage, you can pass the option PIOCUSAGE to the ioctl system call:

```
Return_code = ioctl("/proc/<pid>", PIOCUSAGE, &buff) ;
```

In Linux, you must read the relevant files from the /proc/<pid> directory to get the relevant details, and the option `PIOCUSAGE` is not available. In both cases, `pid` is the process id of the current process executing the command.

Another example is when you want to get the process heap info. To get the process heap info in Solaris, you can use the following:

```
Return_code = Ioctl("/proc/<pid>",PIOCPSINFO,&psinfo)
```

In this bit of code, `psinfo` is of type `struct prpsinfo`; `Prpsinfo.pr_size` gives the process heap size.

In Linux, the number of pages in use is available from /proc/<pid>/mem and /proc/<pid>/stat. The pagesize is available by the system call `getpagesize`. Multiplying the two values -- number of pages and the pagesize -- gives the current size of the heap.

Getopt

In Linux, the `getopt` call follows the POSIX standards only if the environment variable `POSIXLY_CORRECT` is set.

If the environment variable `POSIXLY_CORRECT` is set, parsing stops as soon as the first non-option parameter (a parameter that does not start with a "-") is found that is not an option argument. The remaining parameters are all interpreted as non-option parameters.

Differences while invoking shell scripts

If the shell script uses the `su` command, a child shell will be spawned. This behavior is different from Solaris, where the `su` command does not spawn a new shell.

If you issue the command `su - root` on Linux, the output of `ps` command will look like this:

```
30931 pts/4    00:00:00 su
31004 pts/4    00:00:00 ksh
```

In this code, 30931 is the parent process for the process 31004. You might have to modify some scripts that could be affected by this relationship.

Device handling on reboot

From RHEL4 onwards, Linux employs the concept of hotplug-subsystem udev. It provides the configurable dynamic device-naming support. The device configuration is built up dynamically on every reboot.

For example, if you make a new directory /dev/dsk, the system might be unaware of its existence, and the directory will disappear on reboot. To avoid the disappearance of the /dev/dsk directory, make a directory /etc/udev/devices/dsk, and the system will be able to retain the information on reboot, and /dev/dsk will be available even after a reboot.

Kernel parameters

In Solaris, the kernel parameters can be set in the file /etc/system. In Linux, the kernel parameters can be changed using `sysctl` system call. The parameters that can be changed are available in /proc/sys/kernel, and `procfs` support is required for `sysctl`to work.

Signals

Differences in signaling include the fact that Solaris has 38 signals and Linux uses 31, and the `sigaction` structures are different. For example, in Linux, the `sigaction` structure looks like this:

Listing 5. sigaction structure in Linux

```
struct sigaction
  {
    /* Signal handler.  */
#ifdef __USE_POSIX199309
    union
      {
        /* Used if SA_SIGINFO is not set.  */
        __sighandler_t sa_handler;
        /* Used if SA_SIGINFO is set.  */
        void (*sa_sigaction) (int, siginfo_t *, void *);
      }
    __sigaction_handler;
# define sa_handler     __sigaction_handler.sa_handler
# define sa_sigaction   __sigaction_handler.sa_sigaction
#else
```

```
    __sighandler_t sa_handler;
#endif

    /* Additional set of signals to be blocked.  */
    __sigset_t sa_mask;

    /* Special flags.  */
    int sa_flags;

    /* Restore handler.  */
    void (*sa_restorer) (void);
  };
```

In Solaris, the sigaction structure looks like this:

Listing 6. sigaction structure in Solaris

```

struct sigaction {
        int sa_flags;
        union {
#ifdef  __cplusplus
                void (*_handler)(int);
#else
                void (*_handler)();
#endif
#if defined(__EXTENSIONS__) || ((__STDC__ - 0 == 0) && \
        !defined(_POSIX_C_SOURCE) && !defined(_XOPEN_SOURCE)) || \
        (_POSIX_C_SOURCE > 2) || defined(_XPG4_2)
                void (*_sigaction)(int, siginfo_t *, void *);
#endif
        }       _funcptr;
        sigset_t sa_mask;
#ifndef _LP64
        int sa_resv[2];
#endif
};
```

`siginfo_t` structures are different because the number of supported signals are different. They can be found in /usr/include/bits/signal.h for Linux and in /usr/include/sys/siginfo.h for Solaris.

Threading support and IPC

The publication on porting from Linux on POWER and Solaris covers the differences between Solaris threads and POSIX threads on Linux (provided by the NPTL library).

My experience has been that if the application is already using POSIX threads on Solaris, the migration to Linux using NPTL-based threading is relatively straightforward. Features like process-shared mutexes are also available on Linux, and thus the Solaris code related to IPC mechanisms using pthread mutex locks and condition variable can be used on Linux.

Natural language support

The code pages on natural language support in Solaris may be different from those in Linux, or they could just be named differently. Most of the locales and code pages in Solaris have an equivalent in Linux, though.

Conclusion

The porting effort from Solaris to Linux on x86 in most cases involves just a recompile or minor changes in compiler/linker switches. Some platform-specific changes in the areas of locking, memory maps, threading, and so on may also be required. You should study the differences and plan the port to reduce the time to port.

About the author

Ajay Sood

Ajay Sood is an Senior Software Engineer at IBM in Bangalore, India. He has been with IBM for more than 13 years, and has been a developer on such product development team efforts as DB2 DataLinks, and TXSeries-CICS. His experiences include work in transaction processing, middleware, and system programming on UNIX platforms.