The following is a report of the binary incompatability issues that have arisen between Solaris for x86 and USL's SVR4. Each separate problem has been described in its own numerical section along with affected applications, proposed plans of action, and our opinion of whether USL has broken the ABI in this case.

1. Run-time Loader Name

Statement of difference

USL's SVR4 offering used /usr/lib/libc.so.1 as the run-time loader (aka the "interpreter"). Solaris uses /usr/lib/ld.so.

Applications affected by difference

All USL (standardly) built binaries are affected. Some USL binaries *could* be built with a different interpreter specified, but that takes special effort and none were actually found.

Is USL ABI compliant?

Since the ABI never specifies what the name of the interpreter *must* be, USL is techinally compliant, however it would appear that Solaris is more "typical" in this respect.

Intended plan of action

We (SunSoft) have added code to /usr/lib/libc.so.1 to cause control to be transfered to /usr/lib/ld.so.1 in the event that /usr/lib/libc.so.1 is invoked as the interpreter. Thus USL binaries that specify /usr/lib/libc.so.1 as their interpreter no longer pose a problem when run on Solaris x86.

Technical discussion

The ABI states:

(Page 5-13)

"When building an executable file that uses dynamic linking, the link editor adds a program header element of the type PT_INTERP to an executable file, telling the system to invoke the dynamic linker as the program interpreter. Typically, the dynamic linker requested is the system library, libsys, described in Chapter 6. Exec(BA_OS) and the dynamic linker cooperate to create the process image for the program, which entails the following actions..."

Note: the ABI uses the word "typically" which implies that other interpreters may be used, and as a matter of fact the ABI goes on to say: (Page 5-12)

"An interpreter may be either a shared object or an executable file..."

The ABI also states: (Page 6-2 Figure 6-1 Shared Library Names)

LibraryReference Name libc/usr/lib/libc.so.1 libsys/usr/lib/ld.so.1

2. R_386_COPY Relocations

Statement of difference

USL's SVR4 fails to generate R_386_COPY relocations for two specific symbols.

Applications affected by difference

Virtually all applications that utilize stdio (standard I/O) are affected. Further all programs that use the is*() (checks for if a chacter lies within a chacter class) are similarly affected. Unpredictable behaviour can result.

Is USL ABI compliant?

It is my belief that USL is non-ABI complaint with respect to this issue.

Intended plan of action

Solaris for x86's dynamic linker now has code to check to see if the dynamic linker that was initially invoked is *not* /usr/lib/ld.so.1 and if that is true it assumes that this is a USL type executable file which may need this special treatment and in that case emulates the USL dynamic linker. Assuming that USL continues to utilize an interpreter other than /usr/lib/ld.so.1 and continues to special case those two symbols, Solaris x86 will handle this situation.

Technical discussion

The first symbol is __ctype and the second is __iob. What is done is that /usr/lib/libc.so is in reality an archive type file rather than an actual shared object. In that archive __ctype and __iob are declared to be the appropriate number of bytes in the .bss. Thus the linker resolves those two symbols by statically linking them into the user's executable file. Unfortunately, those two symbols are *not* supposed to be initialized to zeros and the copy in /usr/lib/libc.so.1 has the proper initializations. The iABI states that: (Page 4-6)

"R_386_COPYThe link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist in both the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset."

With no copy relocation generated (because of the symbol being resolved by the act of static linkage) the run-time loader is under no obligation to cause the properly initialized version in /usr/lib/libc.so.1 to be copied into the user's executable image. USL's dynamic linker has explicit code to search for those two symbols and if they exist in both the executable and in /usr/lib/libc.so.1 to copy the data.

3. hrtsys() Statically Bound

Statement of difference

USL's SVR4 uses a system call by the name of hrtsys() which implements various high-resolution timer functions

Applications affected by difference:

Any application that relies on gettimeofday(), setitimer() or getitimer() (and perhaps other functions) working correctly will not function properly. Oracle is the most visible application at this time.

Is USL ABI compliant?

It would appear that the USL code is non-ABI compliant with respect to this issue

.

Intended plan of action

SunSoft is investigating the effort it would take to add the hrtsys() system call to the Solaris operation system. This would appear to be a non-minor undertaking.

.

Technical discussion

This functionality is not specified in the ABI, the iABI, POSIX 1003.1, nor the SVID issue 3. In short, no obvious place can be found where the need nor the semantics for this system call is specified. It appears from a review of the USL code that the gettimeofday(), setitimer() and getitimer() calls are implemented using hrtsys(). Each of the preceeding funcions *are* supported by Solaris and in a binary interface manner as suggested by the ABI. USL's code implements these functions as a statically linked functions which are extracted from /usr/lib/libc.so.

The ABI states: (Page 6-2):

"The ABI required conforming applications to use the dynamic linking mechanism described in Chapter 5 to access the services provided in the System Library, libsys, and in the Networking Services Library, libnsl. Use of the other shared libraries documented here is options for applications; the services they contain may be obtained through the use of equivalent archive library routines."

Thus the use of a statically linked libc routines is in itself not prohibited, but the ABI further states: (Page 6-1):

"Shared libraries contribute to the application execution environment and thus appear in the ABI. Functions that reside directly in application files are not specified. For example, mathematical routines, such as sin(BA_LIB), do not appear below. They would be available in a System V development environment, but an application's executable file would contain the associated code. Assuming the implementations of the functions themselves are ABI-conforming, their presence does not affect the conformance of the application. Moverover, the absence of shared library version of particular services does not imply a deprecation of those services."

So it would appear that since gettimeofday(), setitimer() and getitimer() do *not* use the dynamic linking mechanism to access services provided in libsys and/or libnsl (or any other provided shared object) it is non-conforming.

4. bcopy() in libX11

Statement of difference

USL's SVR4 implementation of the libX11 appears to include the routine bcopy(). Solaris for x86 does not.

Applications affected by difference

Any USL binary built with bcopy() that linked against libX in order to resolve the function. Most notably Frame suffers from this problem.

Is USL ABI compliant?

It would appear that USL is non-ABI compliant.

Intended plan of action

SunSoft will provide bcopy in its version of libX so as to allow libX to resolve the reference to bcopy.

Technical discussion

The ABI states: (Page 10-3)

"libXThe X library, libX, generates the X11 protocol and buffers traffic between each client application and the server. A full specification of the libX library and its contents can be found in Xlib - C Language Interface, X Window System, X Version 11, Release 3, (Massachusetts Institue of Technology, 1988). The ABI will track upward-compatible future release of the X library."

The document referenced above does not state that bcopy() is part of the contents of libX.