Chapter 1

iBCS2 under Linux

1.1 Introduction

Early in the development of Linux there were discussions about what the future had in store, and people speculated as to the degree of success that they each thought Linux would ultimately achieve. A common point which was raised again and again was that many of the commercial applications that are used on other platforms either had no "free" equivalents, or had equivalents that were not polished to a commercial quality. Some said that until these packages were available for Linux that Linux itself would just be more of a curiosity of the Internet than a viable platform for the masses. This led to the obvious question as to exactly what would be required in order to run such applications directly under Linux. Most of the commercial applications that are available on the market today come for one of two different platforms; one is Microsoft Windows, and the other is commercial variants of Unix such as SCO Unix or SVr4. Being able to run Windows binaries under Linux is of course the heart of the Wine project, and is discussed elsewhere in this book. In this chapter we will be discussing the work that was done to make it possible to run both SCO and SVr4 binaries under Linux. While the primary focus has been on SCO and SVr4 binaries, it is also possible to run binaries from vanilla SVr3, ISC, Wyse V/386 and Xenix V/386 systems as well - all of these are in effect SVr3 types of systems with different extended capabilities beyond a base SVr3 system.

1.2 Standards

Commercial variants of Unix have over the years have undergone standardization and the result is that there are a number of sets of specifications which define the interfaces and data structures that are used. The version of Unix that is known as SCO is basically a version of SVr3 with a number of SCO specific extensions. SVr4 maintains backwards compatibility with SVr3, but defines new standards which must be adhered to. For our purposes the standards document that we are interested in is the one for which SVr3 is the reference implementation, and this standard is known as the "Intel Binary Compatibility Specification 2", or known as "iBCS2" for short[**?**]. The iBCS2 standard itself is based upon a merging of the older "System V Interface Definition, Issue 2" (SVID)[1], and POSIX.

The iBCS2 standard itself was somewhat imperfect from the start, and with time the cracks have grown to the point where they are now much more obvious. It was originally intended to provide complete binary compatibility between different versions of Unix, but it failed to specify the formats of some of the system files. It did not contain specifications for symbolic links, and it specified only 16 bit inode and device numbers. Also, it specified a scheme of fixed address libraries whereby each entry point was assigned an absolute address in virtual memory. The `libc_s` shared library was designed with a jump table so that it was possible to make changes and fix bugs without breaking old binaries, but in the libraries for `libnsl` and `libX11` there were no provisions made for jump tables.

Different vendors took different approaches to extending iBCS2 to suit their needs. SCO used one set of numbers for syscalls related to symbolic links, but when AT&T released SVr4 they used a different set. SCO uses a socket based networking whereas SVr3 supported networking based upon Transport Layer Interface (TLI) and streams, but SVr4 supports *both* TLI and socket based interfaces. Finally, early versions of shared libraries under SVr3/SCO were considered to be buggy, so many software vendors selling applications chose to distribute binaries that are staticly linked instead of linked to shared libraries.

The iBCS2 standard specifies the interface between the application and the kernel to achieve binary compatibility. In other words, it specifies a standard set of syscalls and the arguments that they take. With SVr4, things are different - here the specification interface is the interface between the application and the shared library. The interface between the shared library and the kernel is unspecified, and while it works well to also use the iBCS2 specifications at the library/kernel interface, it is possible however

[1]SVID Issue 3 describes SVr4.

to use a completely different interface at this level[2]. In ELF/SVr4 libraries each entry point has an address that is assigned at runtime instead of having fixed virtual a address, and a dynamic linker is supplied which performs the final linking between the applications and the libraries. This greatly simplified the procedures for generating shared libraries, and made it much easier to fix bugs without breaking the binaries themselves, and makes ELF libraries very popular with programmers and vendors.

1.3 Kernel Internals

The iBCS2 emulator itself is just a portion of the kernel, and it consists of several portions. The first are the binary loaders, and there are currently three of them to load ELF, COFF and x.out(386) format binaries. The loaders have the responsibility of calling `mmap` as required to map the binary into the user's address space, and in addition they must set up the arguments on the stack in a manner compatible with the respective binaries. In the case of ELF binaries, we look for a program interpreter (i.e. the dynamic linker), and act accordingly if we find reference to it. Finally the binary loaders try and figure out what host system the binary is designed for. The reason is that there are "personality" differences must be taken into account as the binary runs in order for the binary to run properly. The binary personality is used to allow for extensions beyond the iBCS2 specification for:

- Different syscall numbers for some syscall functions.
- Different numerical values used for errors (and returned in `errno`).
- Different numbers assigned to some signals.
- Different numbers assigned to the symbolic constants used in the socket() syscall.
- Emulate different behaviors of some syscalls. For example, some flavors of Unix do not modify the timeout parameter in the select() syscall, and this must be emulated for some applications to work correctly.

Once an iBCS2 binary has begun to execute, the second part of the emulator is used to emulate the actual syscalls. Under linux, syscalls are made through a software interrupt with the "`int 0x80`" instruction where

[2]The earliest implementation of SVr4 binary support under Linux had the dynamic linker simply use functions directly from the linux `libc`.

the arguments are left in machine registers. The iBCS2 standard requires that you instead use the "`lcall 7,0`" instruction[3] where the arguments are left on the stack. Linux itself can trap this instruction, of course, and call the iBCS2 emulator.

The emulator itself looks up the number of the syscall that the application is attempting to call, and then uses the "personality" of the process to decide what function we really need to call. In many cases, we can directly call an existing Linux kernel function, but in other cases the iBCS2 code needs to do something special, either to provide a translation of some of the arguments, or to provide a new functionality. Once the function returns, we check the error code and set the machine registers in the manner specified in the iBCS2 standard for returning error conditions after using the personality dependent translation tables for the value of `errno`.

There are also some subsystems that are also emulated. This includes System V IPC, the `/dev/socksys` socket interface which is used by the Lachman streams based networking (and appears in SCO Unix), Wyse V/386 socket interface and a `/dev/spx` streams device that can be used to make connections to the local X server. Finally, some Linux-specific extensions are available, such as allowing all iBCS2 binaries to use the socket() based networking that the Wyse variant of SVr3 implements.

1.4 Shared libraries

In order to provide a complete runtime environment, it is also necessary to provide shared libraries for those applications that require them. The linkage and loading of shared libraries for SCO/SVr3 and SVr4 is completely different, so different tools are required to create the libraries, but many of the functions in the shared libraries are essentially the same, so it is possible to use the same source tree for shared libraries for both SVr3 and SVr4 binaries. In many cases, it was possible to simply copy functions from the regular linux `libc` and recompile them using the header files for iBCS2. In other cases, such as stdio, existing implementations were not close enough to the iBCS2 standard to be of any use, and in these cases brand new code needed to be written.[4].

[3]This instruction is in a sense nothing more than a "far" call as used with DOS, except that we are in 32 bit protected mode and the segment registers cannot be arbitrarily set. A segment selector value of 7 is set up to be a "call gate" which allows a task running at the lowest level of privilege to call a section of code at a higher level of privilege such as the kernel.[?]

[4]The source tree for the shared libraries can be found on tsx-11.mit.edu in /pub/linux/ALPHA/ibcs2, and all sites that mirror this directory.

1.4.1 SCO/SVr3 shared libraries

The linux version of the `libc_s` shared library for SCO/SVr3 is nearly done as of this writing. Nearly all of the source code is written, and the tools are available to generate the actual libc shared library. The main difficulty at the moment is that there are undocumented internal functions which need to be added to stdio, and these are now being added. Until this version is ready you can simply copy the shared library from an SCO box if your license allows this.

The iBCS2 standard also defines an interface for `libnsl`, which contains the network services library. This library uses a streams based through what is known as the TLI. Linux does not currently support a streams based approach, so for the time being all binaries that require TLI and/or streams will not run. There is interest in eventually adding support for streams/TLI to the linux networking code, and once this happens, so direct support of TLI by the iBCS2 emulator will be possible. The lack of TLI/streams turns out to not be a serious limitation because SCO Unix to use a socket based approach to networking that is accessed through a few special devices in `/dev`. Since linux directly supports sockets, SCO binaries that require access to the network should run without any difficulty.

SVr3/SCO binaries that require either the `libnsl` or the `libX11` shared library will probably never run, and the reason for this is that the entry points of these shared libraries were apparently chosen by simply linking the libraries and extracting the addresses of the individual functions (in other words, it appears as if someone forgot to add the jump table). This makes it virtually impossible to even provide simple bug fixes to these libraries as the slightest change could move the virtual address of the entry points. As a result of this, many vendors who supply X-Windows applications simply choose to use static linking, which should work fine under Linux with the iBCS2 emulator.

1.4.2 SVr4 shared libraries

The shared libraries for SVr4 contain a `libc.so.1`, a `ld.so.1` (a bare dynamic linker for Solaris applications), and skeletal versions of `libnsl.so` and `libsocket.so`. As was the case with SVr3 binaries, there is no kernel support yet for TLI networking, and until this is the case some networking binaries will fail to run. The skeletal version of `libsocket.so` contains definitions of functions used with sockets, and these call the kernel directly (using the Wyse TCP/IP entry points), thus bypassing the need for any TLI code.

As was alluded to earlier, the linkage between the application and the shared library with SVr4/ELF libraries is handled at runtime. Initially the program loader passes control to the dynamic linker, and the dynamic linker goes through the list of relocations that the linker generated, and in effect performs a final linking so that the binary can run. The advantage of this approach is that you do not need to pre-register address spaces for libraries, and the shared libraries themselves can be loaded anywhere in virtual memory[5]. This does mean that there is a startup latency whenever you run a binary, however it is possible to minimize this by means of "lazy symbol binding" where we skip the relocations to the jump table and then pass control to the application. The jump table entries themselves are initialized by the linker in such a way that they all point to the dynamic linker - thus when you attempt to call these functions the proper address can be looked up and inserted back into the jump table.

Not all of the functions that appear in the SVr4 version of `libc.so.1` are present in the linux iBCS2 version. If the application calls a function that is not present, the dynamic linker will print an error message with the name of the missing function and then exit. It is possible to get a complete list of the unimplemented functions called by a given application by using the "`ldd -r`" command. The dynamic linker in the linux iBCS2 version of `libc.so.1` supports the same environment variables to control it's behavior that the SVr4 version of `libc.so.1` does:

- LD_BIND_NOW - Disable "lazy symbol binding", and bind all symbols before the application receives control.
- LD_LIBRARY_PATH - A list of the paths which we should search for the libraries.
- LD_TRACE_LOADED_OBJECTS - Used by "`ldd`", if you set this you get a list of the shared libraries that are required by an application.
- LD_WARN - Used by "`ldd`", if you set this, you get a list of all of the symbols that are required by the application that are not defined in any of the shared libraries. Note that the LD_BIND_NOW environment variable influences the results of this.

It should also be mentioned that wherever possible, the linux iBCS2 `libc.so.1` shared library is compatible with the SVr4 shared library to a degree that some binaries will actually run under SVr4 using the linux version. This compatibility is not present in the `libnsl.so` and `libsocket.so`

[5]This also means that they must be compiled as position independent. On some architectures there is a slight performance penalty for position independence. It is hard to quantify, but for the i386 architecture it is probably around 5%.

shared libraries because these libraries make special syscalls to open and use sockets which are not available under SVr4.

Applications that use X-Windows will require access to one or more shared libraries that contain X-Windows code. Fortunately it is possible to obtain pre compiled shared libraries for SVr4 from binary distributions of XFree86. A list of the ftp sites which carry these distributions should be available in the same location as the `libc_s` library sources.

1.5 Native compilation of iBCS2 programs

1.5.1 SCO/SVr3 native compilation

It is possible to compile iBCS2 compliant COFF binaries under linux because it is possible to configure GCC, GAS and the GNU linker for COFF. Pre-compiled binaries for many of these tools may be available in the same locations as the `libc_s` source tree. This capability has not been thoroughly tested, but all the utilities that you should need should be present. The `crt0.o` files can be generated from sources in the `libc_s` source distribution.

1.5.2 SVr4 native compilation

The situation with ELF is somewhat different. The latest public releases of the GNU assembler and GNU linker do not support position independent code (PIC) or ELF shared libraries. This means that all ELF binaries (and even the linux versions of the ELF shared libraries) must be compiled and linked on a SVr4 machine. This is only a temporary situation as patches for binutils currently exist which provide most of the required functionality. These patches are being added to the development source tree maintained at cygnus.com, and should appear in the next public release.

There has been serious discussion about changing the native binary format for Linux from a.out to ELF. There are several advantages of this - first of all, the current linux shared libraries can be a bit cumbersome to generate, which places an additional burden upon the library developer. ELF shared libraries can be generated quite easily, almost as easily as linking a normal application. Also, other machine architectures have requirements that are much more easily met by ELF than by a.out. The major obstacle that has prevented such a switch are the missing functionalities in GAS and the GNU linker described above. In the event of a switch in native binary format, it is likely that native linux binaries will continue to use the `int 0x80` syscall interface, and that native linux binaries would not use the iBCS2 emulator. It will thus be necessary to distinguish the ELF shared libraries used by the iBCS2 emulator from the ELF shared libraries used for native application, and this will probably be done by moving the iBCS2 compatible libraries from `/usr/lib` to a separate directory. At this writing, such details have not been decided.

1.6 Installation

Installing the iBCS2 emulator on a linux system is relatively straightforward matter. To begin with you should start from either a 1.0 or 1.1 kernel source tree. If you are using anything older than this you may have a difficult time getting things to work properly[6]. Next you should obtain the source tree for the iBCS2 emulator from either tsx-11.mit.edu or any mirror sites of tsx-11, and can be found in the directory `pub/linux/ALPHA/ibcs2`. This distribution should be unpacked into the kernel source tree so that the files go into `/usr/src/linux/ibcs` (assuming that your kernel is in `/usr/src/linux`).

Depending upon what version of the kernel you are running you may need to apply some patches to your kernel source tree. Examine the directory `/usr/src/linux/ibcs/Patches` to see what specific patches may be required. In general, there will be one patch-kit that will be required if you are using a 1.0 kernel, and there may or may not be patches that are required for 1.1 kernels (1.1 kernel versions at this writing do not require patches).

For our purposes here, we will assume that you wish to use the iBCS2 emulator as a loadable kernel module instead of having it linked directly into the kernel. The advantage of using the emulator as a loadable module is that if you are not using the iBCS2 emulator you can unload it and free some memory. In addition, if you find that you need to make changes to the emulator source code, you can unload the old emulator and then load the new version without rebooting the system. If you wish, you can have `/etc/rc` load the module at boot time so that the module will always be present for you.

If you are using a 1.0 kernel source tree, you will then need to do a "`make config`", followed by a "`make dep`". The iBCS support question in the configuration script is essentially asking you whether you want the iBCS2 emulator linked directly into the kernel, and if you want a loadable module you should answer "`n`". Once you have done this, you can type "`make`" to rebuild both the kernel and the iBCS2 emulator (the emulator

[6] If you are using the 1.1 source tree, you should try and obtain all of the patches that are available to maximize your chances of success.

itself will appear in the file `/usr/src/linux/ibcs/iBCS`). Once this is done you should reboot.

If you are running a 1.1 kernel source tree you can simply go to the /usr/src/linux/ibcs directory and type "`make`" to build the module "iBCS".

Before you actually use the iBCS2 emulator for the first time, you will need to create a few special device files such as `/dev/socksys`, `/dev/spx`, and `/dev/XOR`. These are used for TCP/IP in SCO applications and for the local X interface. See the `README` file in the iBCS2 emulator source tree for more specific instructions on how to do this.

Now that you have a kernel compatible with the iBCS2 emulator and you have the emulator itself, you can use the `insmod` utility which is supplied with the `modutils` package[7], and this will actually load the emulator into the kernel's address space and perform all required initialization. When you load the iBCS2 module it is normal for a few messages to be printed to the screen., a few messages will be printed. You can check the file `/proc/modules` to see what loadable modules are currently loaded, and a `rmmod` utility is supplied with the modutils package which can be used to unload the iBCS2 emulator.

Once the emulator is loaded, you can simply run any SCO/SVr3/SVr4 applications in the same way you would run any other application. If you are running a SVr3/SCO binary that requires a shared library, you will have to obtain the library image and install it in the directory `/shlib` on your system. If you are using a SVr4 type of binary that requires a shared library, you have some flexibility because you can use the LD_LIBRARY_PATH environment variable to specify the location of the shared libraries to be used. The default location for SVr4 libraries is `/usr/lib`.[8].

1.7 Tracing

The iBCS2 emulator has a built in trace capability, somewhat like the `truss` facility under SVr4. This is an invaluable tool that can be used to help troubleshoot bugs in the emulator, and to help determine why a given application is not running correctly. There is a small program which you can run that will turn on and off the assorted tracing features of the emulator, and this program can be found in the iBCS2 emulator source tree in the `Tools` subdirectory.

Run "`trace`" with no arguments to get a list of capabilities. Full tracing is enabled using "`trace api`". This is extremely verbose - you probably

[7]This can be found in the directory /pub/linux/sources/sbin/ on tsx-11.mit.edu.

[8]The default location for iBCS2 ELF shared libraries may change if the default native binary format for Linux is changed to ELF.

want to kill syslogd and use "`tail -f`" to dump `/proc/kmsg` directly to a file as quickly as possible if you enable this.

If you have no intention of ever using the trace facilities (and will never complain that something doesn't work) you can remove the IBCS_TRACE option and recompile the emulator without the tracing facilities. This makes the emulator about a third smaller but ensures that there is no way for you to find out if program failures are directly due to faults in the emulator.

1.8 Applications known to work

The list that is presented here is probably incomplete as this information had to be compiled far in advance of press time, and thus you should consider this just a sampling of what is possible. For an uptodate list of the applications that will work, see the file `COMPAT` that comes with the iBCS2 emulator source tree.

- WordPerfect 5.1 - The first "big name" application that worked under the iBCS2 emulator. This works both in character and X modes, and is staticly linked so that you do not need to obtain any shared libraries to use it. The WordPerfect Corporation has a demonstration version that is available via anonymous ftp from ftp.wordperfect.com in `ftp/unix/demos/sco/sco.z`, and you can use this to try it out for free and see how you like it.

- Oracle - this also works, at least to some degree. Here the `/shlib/libnsl_s` shared library is required, which indicates that some of the network capabilities of Oracle will not work until TLI/streams support is present.

- Informix - various programs also have been reported to work. Details are sketchy at this point.

- CorelDraw - Corel has an SCO version of CorelDraw available, and to a large degree it works. So far the only version that has been tested is a demo version that comes on a CDROM, and for some reason this demo version does not come with the fonts required to use text labels. It is unclear whether these fonts actually come with the production version or not - if they do, it is likely that Corel will work completely. Note that Corel uses a license server, and you need to have the `portmap` daemon up and running before you will be able to run either the demo or a real version.

1.9 Other issues - troubleshooting

- The keyboard mapping will not be quite for some SCO applications (especially for function keys). Linux is quite configurable in this regard, so there is an alternate keyboard mapping file that comes with the iBCS2 emulator that can be loaded and used so as to correctly map the function keys. You will need the loadkeys utility to make use of this.

- If you are trying to install WordPerfect, there are some oddities in the installation scripts which you should be aware of. Please examine the file PROD.patches/WP in the iBCS2 emulator source tree for more information.

- Some SCO binaries (notably programs built with Informix 4GL) are sensitive to problems in `/etc/passwd`. For Informix to work, you will need to make sure that each entry in `/etc/passwd` does not have a blank specification for the home directory.

- Some versions of X libraries seem to require 'localhost' to be allowed to connect to the X server even if we aren't using a TCP/IP connection. The X libraries used for VSI*FAX require this, the X libraries used for WordPerfect don't. To solve this problem simply allow 'localhost' to connect to the X server using "`xhost localhost`".

- Some Unix installation disks claim to be Unix tar format, but they appear to be blank. This problem arises because Unix provides two floppy devices, the normal floppy devices that we all know and love (and which are listed as the Xenix compatible devices in the man page) and a second set which skips the first track of the disk. For some reason a few vendors seem to use this second set when preparing distribution disks. WordPerfect seem to do this. Linux currently only supports the normal floppy devices. The work-around is to skip the first track by hand and read each disk individually. Try "`dd if=/dev/fd0 bs=18k skip=1 | tar xfv -`" for a 3.5 high density disk. Change the 18k to 15k for a 5.25" high density disk.

- Sometimes scripts bomb with an unexpected EOF looking for `'`. This only happens on Linux. This is caused by a simple bug in the script that is illustrated by the following example:

  ```
  count=`ls | wc | awk '{ printf "%05d", $1 }`
  ```

  Note the missing `'` at the end of the awk statement. The /bin/sh supplied with SCO will assume (in this case correctly) that the `'` should have occurred immediately before the closing ` and the expression will succeed. The `/bin/sh` used with Linux (normally `bash`) does not make this assumption and gives an error message.

- Sometimes `test` will complain that a numeric argument is required before -eq, -le etc. This comes about because the GNU shellutils `test` and the `test` built in to `bash` (which are the versions of `test` used under Linux) do not accept a null argument as equivalent to 0 so "`test "" -le 5`" will give an error. Under SCO a null argument is taken as equivalent to 0 so the statement would be evaluated as "`test 0 -le 5`".

- Some X fonts that are supplied appear to be corrupt. This is probably because they are snf fonts. The XFree86 X server used with Linux appears to fail to load some snf fonts silently and displays garbage. Pcf fonts work ok and should be used where possible. If you only have the snf fonts all you can do is to try asking the vendor for pcf versions or the bdf definitions (If you have the bdf definitions (WordPerfect ships them) then you can build a pcf set using the `bdftopcf` utility).

- Sometimes line drawing characters do not come out right. This is because the application is assuming that you have the IBM-PC character set and is using 8-bit codes instead of escape sequences. The work-around is to have Linux switch to the PC character set mode with the escape sequence `ESC-(-U`. Arrange to have this sequence sent either before the application is started or as part of the initialization that the application does. You can restore the ISO character set afterwards with `ESC-(-K`.

- Some SVr4 binaries use terminfo instead of termcap. Terminfo compilers (`tic`) are available in a number of places. The ncurses package contains one version.

- SVr3/SCO binaries that use the stat() syscall to obtain the inode number of a given file may get the incorrect number. This is because the st_ino field in the stat structure is only 16 bits wide, while under linux this field is 32 bits wide. The top 16 bits are truncated when this value is assigned to the 16 bit field.

- Any application that uses TLI networking will probably fail.

- At this writing there are a few problems with ipc. These are being worked on, and may be resolved shortly.

- COFF binaries that are linked with `-N` will currently not load. If this gets to be a problem, support will be added.
- If an application dies for any reason, the core dump will be in the native linux format rather than a format that corresponds to the image. Ultimately support will be added for ELF so that a core dump from an ELF image will also be in the ELF format.
- There is currently no support for the `mprotect` syscall under linux, so this means that currently all ELF code pages must be writable for the dynamic linker to work. Patches exist for mprotect, and once these are integrated into the kernel the ELF binary loader and dynamic linker will be modified to make use of this function.

1.10 Copyright Issues

One of the most important things from the outset was to make sure that when writing the iBCS2 emulator and the shared libraries that no copyrights be violated. The fact that very few people actually have access to the actual source code for SVr3, SCO Unix or SVr4 is actually an advantage here, as any such access would probably disqualify someone from working on the iBCS2 project. As far as I know, all of the development of the ibcs2 emulator and libraries was done using published standards which describe many of the things which we needed to know (see Bibliography), man pages, header files, and by observing how iBCS2 compliant test programs operate under linux.

1.11 Acknowledgements

The following people contributed to the development of the iBCS2 emulator and shared libraries. Brandon S. Allbery, Graham Adams, Tor Arnsen, Philip Balister, Alfred Longyear, Mike Jagdis, Joseph L. Portman III, Drew Sullivan and Eric Youngdale.