Debian – the kernel-neutral OS

Matthew Garrett

July 9, 2004

Abstract

The free Unix world has traditionally thought of userspace environments as being quite tightly tied to the kernel. The vast majority of Linux-based distributions utilise a GNU userland, while BSD environments (unsurprisingly) tend to have BSD-derived userlands [1] This results in issues when a user wishes to switch kernels. Rather than having to learn a small set of kernel-specific features, the user must instead come to terms with what is potentially an entirely different userland. This paper will investigate the practicality of porting the Debian userland environment to multiple kernels, allowing for easy transitions between kernels.

The aims of a port

A Debian port to a non-Linux kernel should provide two things. Firstly, it should provide whatever advantages that kernel may have. Secondly, it should be recognisable as Debian. These two aims are not entirely compatible – what should be considered as "Debian"? Consensus so far has been that low-level tools such as ifconfig and mount should keep their native semantics, while administrator tools such as ifup, dpkg, pppconfig and the like should behave similarly on all systems. Where appropriate, these tools should offer the same level of functionality on all platforms with extensions to support platform-specific features.

Choice of libc

Linux and Hurd both use glibc2 as their "native" C library, while FreeBSD and NetBSD each have their own libc derived from the original BSD. Any port must make a decision regarding which of these is more practical. Porting glibc is a moderately awkward task, but not an impossible one. Doing so offers the advantage of increased compatibility with existing Linux source code. However, the job of allowing glibc to take full advantage of the features of the kernel is somewhat harder. For example, a useful port must be able to take advantage of any kernel-provided threading mechanism analagous to NPTL on Linux. The difficulty of this task is likely to outweigh the advantage of the increased ease of porting of other code.

The Debian multiarch proposal[2]

The number of ABIs available to Linux users has been increasing steadily. If PPC64 is taken as an example, users may quite reasonably want to run:

I PPC64 binaries (for example, applications that need more than 4GB of memory)

II PPC32 binaries (applications that would not benefit from being 64 bit)

III x86 binaries (applications that are not available for PPC, run via QEmu)

The current FHS makes this difficult. All libraries must be placed in separate directories depending on their ABI, but the naming scheme of these directories is not well-defined. /lib is defined as the directory containing the libraries required to boot, with the FHS grudgingly allowing this to be a symlink to another directory. Each ABI's library

[1] With the notable exception of Mastodon Linux – `http://www.pell.portland.or.us/~orc/Mastodon/`

[2] See `http://raw.no/debian/amd64-multiarch-3` and `http://www.linuxbase.org/~taggart/multiarch.html`

/lib/i386-linux
/lib/i386-netbsd
/lib/amd64-linux
/lib/amd64-netbsd

Table 1: Directory structure on an AMD64 NetBSD multiarch system

directory must be of the form /lib(suffix), with the example cases being /lib32 and /lib64 for architectures that have a 32 and 64 bit ABI.

This presents two significant problems. Firstly, the suggested naming scheme is sufficiently coarse-grained that seamless multi-arch support is impractical. In the above example, the best case scenario would be that both PPC32 and x86 would expect their libraries to be in /lib32. This would, unsurprisingly, fail. To some extent this can be worked around by using different run-time dynamic linkers with different library search paths, but this will still fail with binaries built with the -rpath argument. The second problem is that a sufficiently fine-grained namespace would lead to a proliferation of long directory names in the root directory.

The basic concept of the Debian multiarch proposal is to provide a fine-grained library namespace without polluting the directory structure, while also allowing for integration with the package management system. A user should be able to install any binary capable of running on his system without having to be aware of the multiple ABIs involved.

To this end, libraries would instead be installed in a subdirectory of /lib defined by their ABI. Binary packages would depend on the appropriate library packages. Any binary package with an ABI supported by the OS would be installable, bringing with it the dependent libraries. Since the subdirectory names would be standardised, this would avoid any difficulties with -rpath.

On an AMD64 NetBSD system, perl would link against libraries in /lib/amd64-netbsd. A 64-bit version of Intel's C compiler would link against libraries in /lib/amd64-linux. Legacy NetBSD applications would link against libraries in /lib/i386-netbsd, and legacy Linux applications (such as Unreal Tournament) would link against libraries in /lib/i386-linux.

Multiarch – transition plan or long-term strategy?

Multiarch allows for the porting process to be greatly simplified. Rather than requiring every single package in the Debian archive to be ported, a usable system can be generated purely by porting the base distribution and toolchain. In most cases, a user will not be aware that they are running a binary from a different kernel – a request to install a package will simply find the closest match to the native architecture and pull in any library dependencies.

However, no matter how good the emulation, this is unlikely to be the preferred mechanism for performance-critical applications. Linux binaries running on NetBSD 2.0 would be unable to take advantage of the new kernel threading support, for example. It will be necessary to determine what level of porting effort is appropriate to maximise the functionality payoff.

The process of porting

Regrettably, Debian is not designed with the aim of being easy to port. As a consequence, cyclical build dependencies are not uncommon. The net result is that much of the early porting work must be done by hand. An assumption will be made here that a GNU toolchain already exists for the platform concerned.

- Firstly, dpkg must be ported. This is generally not difficult, with the primary task being to add an entry to the archtable[3]
- Next, some number of the base packages must be built. At this point several of their build-dependencies will be unavailable, and so this will still tend to be by hand.
- Once a sizable proportion of the base packages are built, they can be rebuilt in the form of .deb files rather than simply being installed directly.
- The base system must then be rebuilt in order to ensure that it has been built entirely against a Debian configuration.

[3]The archtable contains a mapping of architecture triplets to Debian architecture names

- Around this point, it becomes possible to run an automated build daemon. A large proportion of packages will then be built automatically, and some may even be correct.

The vast majority of packages will build without modification. However, those packages which are problematic often turn out to be awkward infrastructural packages, and their absence will hold up building of a large number of other packages. Much "by hand" intervention will be necessary at this stage. Gradually the required patches will be merged into the standard Debian source, allowing for autobuilding to take place without manual intervention. Modifications to increase portability to one platform tend to help porting to others, as troublespots are highlighted.

Porting code

All kernels currently being worked on are broadly POSIX-compliant and have a GNU toolchain available. As a result, the vast majority of code in Debian builds without modification. There are three main classes of failure:

- Platform dependent code. This is probably best exemplified by programs such as mount which are tightly tied to the kernel. The easiest solution to this problem is likely to be to simply use the native code.
- Code requiring features not present in the platform's C library. This is now fairly rare, since there is an incentive for OS authors to ensure that as many third party applications as possible run. At the same time, there is an incentive for application authors to ensure that their code runs on as many platforms as possible. In the small number of cases where it is an issue, patches to fix the code are generally trivial.
- Code making assumptions about userland based on the kernel. Code may behave differently depending on the platform it believes itself to be running on. This may result in it making assumptions about filesystem layout or command behaviour. Convincing it that it should assume a GNU userland is generally trivial.

However, there are generally still one or two underlying awkwardnesses. For example, the BSDs have an approach to password databases that is markedly different to Linux. Passwords and user information are stored in a binary database, and the traditional plain text files are generated from this. This could be dealt with in two ways. Firstly, all applications that attempt to manipulate the password database could be ported to the BSD mechanism. Since the vast majority of password manipulation and checking is now done via PAM, this would require relatively little modification. Alternatively, an implementation of the Linux-style password management could be written and used. This would merely require that all relevant applications be modified to link against this new implementation – alternatively, the implementation could be merged into the C library.

Social issues

While free software allows for the production of derived projects without the consent of the upstream developers, this is obviously undesirable. Upstream benefits from a greater potential userbase, while a good working relationship with upstream developers makes it easier to resolve portability issues. Discussion must take place with the awareness that, fundamentally, a port to a kernel needs the support of the kernel's upstream far more than the kernel's upstream need the support of the port developers.

An especially tricky area is that of trademarks and branding. There is generally a desire that a derived product not be confused with the "real thing", and trademark law potentially allows this to become a legal issue if not resolved to the satisfaction of all concerned. Finding an appropriate name that clearly describes the derivation of the port without interfering with upstream's desires for lack of brand confusion is a job that should not be underestimated.

Conclusion

Porting Debian to further kernels allows for more widespread use, both on hardware not well supported by the existing ports and in areas where a specific kernel is deemed desirable. The work in-

volved is difficult but not insurmountable, and the potential for a larger userbase is likely to make this worthwhile.