An Updated Directory Structure for Unix

Michael Homer
michael@gobolinux.org

Hisham Muhammad
hisham@gobolinux.org

Jonas Karlsson
jonas@gobolinux.org

Abstract

While the standard Unix structure has survived for many years, much of it rests on assumptions that are no longer true or necessary. This paper will explore the historical reasoning that gave rise to the current structure, and how greater functional organisation of the Unix filesystem layout may be achieved that is more in line with the reality of contemporary computing. GoboLinux is a distribution using an alternative filesystem hierarchy intended to provide a more logical layout for programs and data, and to make the structure of installed applications explicit in the directory tree.

1 Introduction

While the standard Unix structure has survived for many years, much of it rests on assumptions that are no longer true or necessary. This paper will explore the historical reasoning that gave rise to the current structure, and how greater functional organisation of the Unix filesystem layout may be achieved that is more line with the reality of contemporary computing.

GoboLinux is a distribution using an alternative filesystem hierarchy intended to provide a more logical layout for programs and data, and to make the structure of installed applications explicit in the directory tree. This paper presents the directory structure employed in GoboLinux, including the rationale behind the changes and an assessment of the shortcomings it was proposed to fix.

2 Overview

In GoboLinux, each program is installed into its own separate, versioned directory, which also represents the package database. An automatically-maintained tree of symbolic links keeps the program contents accessible without overhead, while another set of fixed links maintains compatibility with the original Unix tree. Programs may be installed using the distribution's binary packages, its "Compile" tool and database of recipes, or manually, each giving equal results.

2.1 History

Many of the decisions made in creating this layout are somewhat heretical, and it bears to investigate the origins of the current structure and explain why we feel it is acceptable and desirable to break with it.

The current standard hierarchy developed from one suited to the historical needs of large Unix systems. The `/usr` hierarchy is distinguished from the root so that it may be located on a separate physical disk or accessed over the network from a workstation with limited storage. Only the basic tools would be installed on the root partition, sufficient to boot and to repair the system in single-user mode, and all other software would be installed on the remote partition.

The single-user rescue mode is no longer a valid reason for the distinction:

> When I need to rescue my system, I can use a fully-featured live CD that runs a complete Linux distribution with a graphical desktop, that allows me to browse the web and search for the solution to my problem, and use all of the features of a regular system to fix it." [1]

There was a rationale behind having a bare-bones rescue mode at one point, but there is a better solution available now.

Remote mounting remains a justification for the split, though it is an increasingly less common use case with modern desktop and server systems having large disks capable of installing all of their software locally. It may still be desirable to store software separately for administrative reasons, but complicating the entire system for just one use case is not sensible. In any

case, the solution has been obsoleted from the other direction as well - what if you have more than one application server? You can mount one on `/usr`, and one on `/opt`, and for any more you have to create new non-standard directories[1]. It is no longer worth catering to this scenario when it will be necessary to break the standard regardless.

As with the live CD, a technical advance helps solve this problem for us as well: *union mounts* allow us to overlay as many directories as necessary on top of one another. These are very flexible and even allow localised overriding. The GoboLinux `/Programs` tree is "the collection of all programs available on the system"[1], and they could be drawn from many application servers if necessary. It is possible even without an overlay filesystem to move an individual program or version of a program to an arbitrary location using only a single symlink in the `/Programs` tree.

The distinction between `bin` and `sbin` is similarly obsolete and unnecessary. Unix systems have a robust permissions system, and any command that must only be run by the superuser can be `chmod 700`, when that is genuinely necessary. A number of commands required to be placed in `sbin` have perfectly valid non-root uses, such as `ifconfig`. Both distinctions are historical anomalies, and anachronisms today.

Of course, some of these reasons are stronger than others – they are being laid out to show why we feel that it is acceptable to break with tradition. Why it is desirable to do so is discussed in the next section.

GoboLinux in fact maintains full compatibility with this structure, possibly to a greater degree than many distributions. There is a simple legacy tree of symbolic links from the FHS directory to the GoboLinux equivalent, so resolving `/usr/bin/perl` will still find the correct executable (as will `/bin/perl`, or even `/usr/X11R6/bin/perl`). These links are kept hidden by default using a small patch to the kernel VFS layer called GoboHide. The patch is entirely optional and has only cosmetic function. Its only role is to inhibit certain directory entries from appearing in a directory listing, according to a list provided at runtime. These paths are still accessible, but invisible in directory listings.

2.2 Layout

For this example of the layout we will use Bash 4.0 as our example program when necessary. Any other program could be used in its place with exactly the same treatment.

Bash will be installed in /Programs/Bash/4.0. The bash executable will be at /Programs/Bash/4.0/bin/bash and any hypothetical libraries will be in /Programs/Bash/4.0/lib/. If Bash 3.2 is installed as well, its files will be in /Programs/Bash/3.2. There is a tree of symbolic links pointing to all of the executables, libraries, man pages, and headers in the system: /System/Index/bin/bash points to the bash executable.

The goal of this layout is to make the structure of installed applications explicit in the directory tree. All of the files relating to Bash 4.0 are together in one place, as are all of the files relating to all installed versions of Bash. The view into the system provided by the links tree is separate as it has different concerns and should not affect the physical layout.

The advantages of this kind of structure are recognised by the FHS itself in the form of the /opt tree: it is reserved for "add-on application software packages"[2, 12], which must be installed as either /opt/<package> – much like the /Programs tree – or under /opt/<provider>, where <provider> is a LANANA-registered provider name[2, 12-13] (LSB extends this to allow an FQDN[3]). /usr/X11R6 is an FHS-mandated project-specific directory as well. Given this recognition it is entirely reasonable to extend it to cover the entire system, and historical inertia is all that holds us back.

2.2.1 Tree

As well as the /Programs tree, there are other directories in the system for the symlinks and other required components. Previous versions of GoboLinux have used the /System/Links structure below, while the next will use the /System/Index tree. With /System/Index software is built against the common prefix and then installed into the /Programs tree, in order to work better with some awkward software. The switch was motivated particularly by the increasing popularity of CMake, which has defaults that do not allow overriding its automatically-detected paths, and other systems that make strong assumptions about the filesystem layout.

This table shows the correspondence of FHS directories to both trees. In general, the FHS path will also be a hidden symlink to the GoboLinux location.

FHS	/System/Links	/System/Index
/bin	/System/Links/Executables	/System/Index/bin
/lib	/System/Links/Libraries	/System/Index/lib
/sbin	/System/Links/Executables	/System/Index/sbin*
/etc	/System/Settings	
/usr	/	/System/Index
/usr/local	/usr	
/usr/X11R6	/usr	
/mnt	/Mount	
/dev	/System/Kernel/Devices	
/sys	/System/Kernel/Objects	
/proc	/System/Kernel/Status	

* Note that bin and sbin remain merged; sbin is a symlink to bin.

The capitalised names were chosen to avoid possible conflicts with future paths reserved by the kernel or other components (as happened with `/sys`). They were not regarded as problematic to type because the shell can tab-complete case-insensitively, and in any case typing the full paths should be relatively rare [4][1]. Both the FHS and GoboLinux paths are available to users and applications, so programs with fixed paths will continue to work.

3 Advantages and Implications

3.1 Manipulability

This set-up has three major implications: firstly, that the origin of any file in the system can be found simply by reading the link to discover its location. `which mkpasswd` will give `/Programs/Whois/4.7.33/bin/mkpasswd`, revealing it to be a part of Whois and the version thereof. In fact, any package management operation can be performed using only the standard POSIX tools.

Finding which programs are installed, or which versions of a program, can be done using the `ls` command. Software can be removed using `rm`. Listing the files belonging to a program is simple with `ls` or `find`. It is not necessary or even recommended to do most package manipulation tasks this way within GoboLinux, and instead suggested to use the built-in tools `SymlinkProgram`,

`RemoveProgram`, or `DisableProgram`, but it is possible if desired or necessary. Read operations are very suitable for the POSIX commands and this is frequently convenient for finding information at a glance.

3.2 Parallelism

Another implication is that multiple versions of a program may inherently be maintained in parallel. Where the files contained have distinct names they may be fully active, as is the practical case for most libraries. Programs that link against specific versioned sonames can continue to function, while those that use generic names can use the current version. Where the files do not have distinct names – or at least some of them do not, as with many programs providing executables – it is possible to switch between the versions with a single command or to call the executable of another version by providing its full path. This is something of an unsung feature that is frequently very convenient. Some distributions provide a version of this selection functionality for a limited subset of programs like GCC, but here it is fully generic and available for every program with no additional effort.

3.3 Unpackaged software

It is also possible to install programs that are not, or not yet, contained within the distribution packaging system manually while still benefitting from all the advantages of the package management system. Unlike installing into other systems this does not run the risk of interfering with the default package manager or of creating "orphan" files strewn across the system - the program tree can easily be removed later in the usual fashion, avoiding the complications of maintaining extra programs in `/usr/local`.

If the program were later to be included in the packaging system the upgrade path would be clear and uncomplicated, with the existing installation being a first-class citizen of the system. If it is never included, because it is site-specific, proprietary, or licensing restrictions forbid distribution, the user can still use the tools of the packaging system to manipulate it. It is not necessary to consider that it is from outside the system except when installing, when it requires no additional effort above building or extracting it manually.

3.4 In combination

Those three advantages between them make for a very powerful and flexible system that is able to adjust to the needs of the user or local administrator, rather than bending them to its.

This is also a boon for system repair. While a corrupted package database or files, or an accidentally-removed system component can be fatal in other distributions it is usually fairly simple to repair here. Reverting to a previous version is trivial, and it is possible to disable a program by removing links to it to check whether it is really in use or not before deleting the package entirely. Even severe problems like removing the running libc are repairable, because the packaging system is manually manipulable. Experienced users report that they have never encountered an issue that was not resolvable or that required a reinstallation.

The slogan to go with the layout is "the filesystem is the package manager", though that has sometimes led to misapprehensions. It is possible to do virtually any task manually in the filesystem, but still encouraged to use the regular package-management tools included in the system wherever possible. These tools perform the same filesystem tasks automatically and sometimes more cleverly. It would be more accurate to say that "the filesystem is the package management database". The distribution provides both binary packages and "recipes" for building from source using the automated `Compile` tool, both covering a wide range of software. In order to dispel (hopefully) some of those misapprehensions preëmptively a brief overview of some of the tools follows.

3.5 Tools

3.5.1 DisableProgram and RemoveProgram

`DisableProgram` removes the links to a given program, and `RemoveProgram` both disables and deletes the program. Removing a program does not leave broken links, and it is not necessary to curate them manually.

The ability to disable a program without removing it from the system is another unsung feature of the structure. It is possible to test the removal of software without actually deleting it, and to restore it immediately if its removal was problematic. Disabled software can still be accessed using its full path but will not be found by other software looking for it, which can

also be useful in some cases. Clearing out old software to reclaim disk space can be done as a two-stage process, first disabling and only removing later when it is clear that no problems arose.

3.5.2 SymlinkProgram

`SymlinkProgram` creates the links to a program, called automatically as part of the installation process or used later on to reactivate a disabled program or change the active version. It will not by default overwrite links from a different program, so which executables and libraries are in use does not depend on which program was installed or upgraded most recently. Switching between versions and reactivating disabled software are the most common uses of this tool manually.

3.5.3 Compile

The `Compile` tool builds software from source using instructions from a "recipe". These are simple declarative files describing the software. The simplest possible recipe looks like this:

```
url=http://example.com/foo-1.3.tar.bz2
recipe_type=configure
```

All that is required in most cases is a declaration of where to find the source code and what type of build system it uses. For autoconf-based systems there is usually no more required information at all, while other build systems sometimes require specifying which Makefile variable to override or similar configuration.

While it is possible to make much more complicated recipes including further configuration, customised build steps, or optional behaviours, the simple basic structure has enabled the system to build up recipes for thousands of distinct programs from a relatively small user base. Creating a new recipe for software that isn't included is usually a simple task. As far as possible the complicated work of path configuration and sequencing is offloaded to the tool, which understands all of the commonly-encountered build systems. In most cases there is no particular adaptation required to fit well-designed software into the new directory structure: Compile will pass in the correct paths for the program to its configure script automatically.

Dependencies are specified in a separate file, packaged up with the recipe, listing out required programs and versions. These are checked against the state of the /Programs tree itself, so software that has been added or removed manually will be picked up correctly. To build the Foo program here, all that would be necessary is `Compile foo` and the recipe would be fetched, its dependencies checked and installed, and the software would be built.

3.5.4 InstallPackage

The distribution also provides binary packages built by the developers for most software. It is not a source-based distribution, though the ease of submitting and using recipes has attracted people who are interested in such a system, and makes the recipe repository frequently advanced of the binary package repository. `InstallPackage bash` will fetch and install the newest Bash package from the repository without compiling.

3.5.5 DetachProgram and AttachProgram

These are less related to misapprehensions, but provide a unique feature of the structure that was mentioned briefly earlier. `DetachProgram` can take an installed program, move it to another location, and create symlinks in the `/Programs` tree to maintain its functionality. That allows distributing programs over multiple partitions without using a union filesystem, and seamlessly as far as the tools and user experience is concerned.

`AttachProgram` takes a disconnected program tree like that created by `DetachProgram` and ties it into the `/Programs` tree to be linked. It could be used to connect multiple systems to the same program on a network server, a more flexible solution than using `/usr` and `/opt` as mountpoints for application servers directly. Depending on the circumstances a union filesystem may be superior, but having the option here is valuable. An interesting trait that may be a plus or a minus is that when not on that network, the symlinks will not resolve and the program silently drops out of the system.

4 Alternative approaches

There have been a few other attempts to deal with the same issues that are at least superficially similar. Most of these in fact have different goals but have arrived at a structure resembling GoboLinux in some ways.

One such system is Stow[5]. Stow is intended for use within a standard FHS system, and maintains a separate tree `/usr/local/stow` containing named package directories. It creates symlinks to the files in those in in `/usr/local/{bin,lib,...}`. It also uses the filesystem structure as the authoritative source of data on installed packages. Stow cannot maintain multiple versions and can only be used for application software. Encap[6] is similar to Stow, and both are patterned after the Depot[7] system developed at CMU[8][9]. These are the most similar to GoboLinux of the approaches described here.

NixOS[10] also installs programs in separate directories and supports multiple parallel versions. Its goal is to be a system configuration management system and it uses cryptographic hashes in its paths to ensure consistency and atomic operations. As such it is not possible to install software from outside the packaging system or to manipulate packages manually. Instead it focuses on building the entire system state from a type of functional expression, which can be changed or rolled back to revert to a previous configuration. It serves a very different purpose despite being superficially comparable to GoboLinux.

Mac OS X combines a Unix core with the Mac OS GUI and structure familiar to its users and incorporates both a Unix and Mac OS tree in its filesystem, hiding the Unix tree in most cases in a similar way to GoboHide[4]. Software installation happens in the usual Mac OS fashion and the physical filesystem structure is mostly irrelevant. It is not possible (or at least advisable) to manipulate it directly. The structure is similar in appearance but not in functionality.

5 Other features

5.1 /System/Aliens

The Aliens system was developed to integrate third-party packaging systems into the distribution's package management system [11]. These systems, such as CPAN, LuaRocks, and RubyGems, have become increasingly common recently. Within some communities they are the only obvious way of publishing software, or even the only way at all. The Aliens system aims to embrace rather than extinguish these systems.

Because of their popularity, users frequently do want to use these systems or to use a library that is available through them. Historically it has been usual

to deprecate their use and advocate using the distribution's package management system as the sole source of software, which requires the maintainers to wrap all of the sometimes thousands of packages available in each system into the local packaging format or have missing software. This is impractical and so users have frequently installed the domain-specific system themselves into either `/usr/local` or – worse – over the top of the distribution-managed tree. The GoboLinux structure is able to support this better than most, but it is still ungainly and error-prone. With parallel trees it is sometimes not obvious which library will be loaded and there can be conflicts even when the files themselves do not overlap.

Many of these third-party systems in essence treat themselves as a single program overall, including all of the software available through them. The Aliens system acknowledges and accepts that behaviour, but keeps them segregated from the main body of programs with more usual behaviour. It also allows components of the systems to be used directly: a program in the distribution packaging system is able to depend on a library available through the alien system and have that dependency honoured.

In the Aliens system each of the third-party systems is given control of its own directory tree under `/System/Aliens` and the relevant language or tool is configured to use that location for libraries along with the default path. The user can use the standard tool that comes with the alien system to install libraries they want to use into that tree and they will be automatically available across the system. A program within the distribution is also able to depend on a part of an alien system: a dependency of the form "`LuaRocks:json >= 1.0`" will cause the "json" LuaRock to be installed if it is not already available. This means there is no overhead of repackaging or of packages being out-of-date while the full array of software is available to users.

In some ways this is a step back from the explicitly-structured tree that is at GoboLinux's heart, but the benefits it provides in terms of software availability and user satisfaction are worthwhile. The systems at hand do treat themselves as a single program and still have all of their files located together, but with the trees separated to make their unusual behaviour clear. Removing the possibility of conflict with user trees and embracing the domain-specific knowledge that the maintainers of the alien systems have is an advantage over the old arrangement.

5.2 Use flags

Similar to those in Gentoo, these allow user input into build configuration of software built from source. They are fairly simple in comparison to Gentoo's and designed to fit in with the rest of the GoboLinux philosophy. Inside a recipe their use is principally declarative, though it can be more complicated in the same way as other recipe functions when it is needed. Importantly, they are for the most part used only when the option introduces a new dependency for the program. The flag can then be named after that program, based on the name used within the tree.

This has an important implication: it is possible to enable flags automatically based on the installed software. These "automatic flags" are turned on before the configuration file is parsed, so they can be overridden later, but enable a very flexible usage. In line with GoboLinux allowing software to be installed manually within the package management system, the flags remain out of the way and do not require any extra bookkeeping until something unusual is required. The presumption is that installing a program means that it should be used and supported elsewhere, until that is contradicted. These flags are read from the `/Programs` tree directly and so include any manually-installed software as well.

There are also "generic flags" in the system. These flags collect multiple other flags in a particular order and will enable their components when available. The "gui" generic flag, if enabled, will enable whichever GUI library flag the program in question supports. If more than one library is supported, it will enable the one that is specified as most preferred by the user in the generic flag.

Between these two features the Compile system is able to give the "sensible" behaviour by default, without giving up any flexibility for the user. Automatic flags, in particular, help the system to conform to user expectations in the common case. Binary packages are built using a set of default flags corresponding to the packages shipped on the CD. If the user does not change the flags themselves the system will behave in the same way it did before the flags existed.

6 Conclusion

GoboLinux uses a different filesystem hierarchy with the aim of making the role of programs in the system explicit. The hierarchy places each program in

its own directory and uses a tree of symbolic links to maintain compatibility with the existing layout. It is designed to allow manual manipulation of programs and to use the filesystem as the live package management database, allowing multiple versions of the same program to be installed at once as well as the installation of unpackaged software into the system. While it breaks with the existing structure, which has become outdated and overtaken by developments, it remains fully backwards-compatible with it.

References

[1] Hisham H. Muhammad. I am not clueless -or- Myths and misconceptions about the design of GoboLinux. `http://gobolinux.org/index.php?page=doc/articles/clueless`.

[2] Rusty Russell, Daniel Quinlan, and Christopher Yeoh. Filesystem Hierarchy Standard 2.3. Technical report, Filesystem Hierarchy Standard Group, 2004.

[3] LANANA LSB Provider Name Registry. `http://www.lanana.org/lsbreg/providers/`.

[4] Hisham Muhammad and André Detsch. An alternative for the UNIX directory structure. In *Proceedings of the III WSL - Workshop em Software Livre, Porto Alegre*, 2002.

[5] GNU Stow. `http://www.gnu.org/software/stow/stow.html`.

[6] Encap Package Management System. `http://www.encap.org/`.

[7] Wallace Collyer and Walter Wong. Depot: A tool for managing software environments. In *Usenix LISA VI Conference*, 1992.

[8] Encap FAQ. `http://www.encap.org/faq.html`.

[9] GNU Stow manual. `http://www.gnu.org/software/stow/manual.html`.

[10] About NixOS. `http://nixos.org/nixos/`.

[11] Michael Homer. Aliens: Integrating domain-specific package managers into distribution package management systems. In *Distro Summit at linux.conf.au*, 2010.