AN ALTERNATIVE FOR THE UNIX DIRECTORY STRUCTURE*

Hisham Muhammad André Detsch

Programa de Pós-Graduação em Computação Aplicada - PIPCA
Centro de Ciências Exatas e Tecnológicas
UNISINOS - Universidade do Vale do Rio dos Sinos
São Leopoldo - RS - Brasil
{hisham,detsch}@exatas.unisinos.br

Abstract. This work presents the directory hierarchy used in the GoboLinux distribution, a new model of directory tree for UNIX-based operating systems. This alternative approach provides a greater functional organization, designed to improve the management of software installed from source code compilation. It does that by making the structure of installed applications explicit in the directory tree.

1 Introduction

The UNIX operating system was first used on environments where users accessed a central application server through terminal stations with low (or no) storage capacity. A number of characteristics of this system, most notably the data organization structured as a directory tree, reflect this history. The storage model based on trees proves to be adequate to this day; however, the logic behind the UNIX directory hierarchy is based on assumptions that no longer correspond to the reality of most of the existing Linux installations. The Linux operating system is getting widely used on personal stations which execute and store the user's applications. In this context, there is no "application server", and still, conventions such as the existence of different repositories for libraries persist (`/lib`, `/usr/lib` and `/usr/local/lib`).

Because of the fast pace of development in the Free Software community (*"release early, release often"* [3]), the process of installing and removing programs became common and frequent. This is very different from the scenario on which the criteria for the UNIX directory hierarchy (still used by the Linux distributions) were based upon. As these criteria do not take into account the needs of today's reality, it becomes interesting to reevaluate them and seek an alternative.

This paper presents a directory tree that was conceived from the needs and usage patterns of modern Linux systems and, still, manages to retain compatibility with the UNIX legacy. Initially, aspects of the classic hierarchy are discussed (Section 2). Next, an overview of the approaches employed in directory trees of other systems is presented (Section 3). Section 4 describes the devised hierarchy, while Section 5 reports experiences related to the use of this model. Finally, Section 6 concludes the paper.

2 Characteristics of the current hierarchy

In the UNIX tree, directories serve two purposes: differentiate categories of files and differentiate their location in the network. Files of the "executable" category from all applications are stored in six directories: `/bin`, `/usr/bin`, `/usr/local/bin`, `/sbin`, `/usr/sbin` and `/usr/local/sbin`[1], where the criteria of choice to determine which of those six directories is used for a given file is basically its physical location (local or remote).

*This is a translated and slightly revised version of the paper presented at the III International Forum on Free Software, Porto Alegre, Brazil, 2002.

[1]files from the X Window System are, historically, an exception to this rule, possessing an entire UNIX-like hierarchy under `/usr/X11R6`

Still, some programs are installed in other locations than those dictated by the above rules. For instance, the File System Hierarchy standard has an arbitrary list of which executable files should be stored at the /bin directory.

Some programs assume that certain files are stored in specific locations (for example, /lib/cpp, /usr/bin/python). This is a source of incompatibilities, even between different Linux distributions that follow the traditional model of directories. But the biggest problem caused by this approach is the difficulty in the removal of programs, since files of different applications are mixed in the same directories and different files from a single application are spread throughout a number of directories.

The solution used by companies and organizations that develop Linux distributions in order to maintain a correspondence between individual files and applications is *"package managing"*, that is, installing and removing software using a program that maintains a database that relates existing files in the system to the applications from which they were originated. The main limitation of this method is the fact that installation of applications from source code generates inconsistencies in the database.

A common practice is to keep the /usr hierarchy maintained by the package manager and target the installation of programs compiled locally to /usr/local. This keeps the database consistent, but does not solve the issue on how to remove programs installed from source.

The UNIX hierarchy standard defined an extra directory, /opt, to allow groups of applications to be installed separately from the rest. This can be considered an acknowledgment of the existence of the problems enumerated above. Further, this causes a conflict of criteria in the standard itself.

3 Alternative approaches

Practically all operating systems developed after the creation of UNIX use the model of directory trees. The different organizations applied to the directory hierarchy of those systems reflect the changes both in the way how computers are used and in their storage capacity. Below, we describe the directory trees existing in Mac OS X and AtheOS, desktop operating systems that possess a certain degree of UNIX heritage.

The adoption of a kernel and tools based, respectively, on Mach 3.0 and FreeBSD brought to Apple Computer the challenge to combine the UNIX hierarchy with a look and feel familiar to Mac OS users. Mac OS X ([1]) uses an uncommon strategy to achieve this feat. In its graphical interface, a Macintosh directory tree is presented, containing directories such as /System/Library and /Network/*Remote_Station*. Actually, these directories are a subset of the real directory tree. Also, the interface displays some directories in locations other than their physical storage points. For instance, /Mac OS X is a link to the root directory, and some directories, such as /Applications, appear in the interface as /Mac OS X/Applications. Accessing the file system through a text-based shell, "hidden" UNIX directories such as /usr and /etc become accessible. Note that this approach is only possible in a proprietary environment, where the entire user interface of the system is developed by a single company. In a system like Linux this would be impossible, given the heterogeneity of graphical user interfaces available.

The directory hierarchy of the AtheOS operating system ([5]) is partially based on the UNIX tree. In AtheOS, for example, the /usr directory is used for the intents that, in UNIX, /opt is used. This is made feasible through the ^ directory convention. The AtheOS libraries recognize this as the "directory where the currently running executable file resides", analogously as ~, the UNIX indicator to the user's home directory. Unfortunately, Linux being a UNIX clone, it cannot employ solutions as this one, since this causes considerable portability problems (a text-based Linux application can be easily ported to AtheOS, but the opposite is not true).

There are programs that try to present alternatives to provide, to some degree, a reorganization of the directory hierarchy. Two of the most used programs of this kind are *GNU Stow* ([4]) and *Encap* ([2]). Both follow the basic idea presented by the *Depot* software ([7]), developed at Carnegie Mellon. The principle is to maintain two directory trees: a "real" one, where the files are sorted by application; and another one, sorted in the traditional way, containing links to the files located at the first tree[2]. GNU Stow intends to be a simplified alternative to Depot, since, contrary to Depot, it does not maintain a database. When using Stow, the application should be compiled with paths relative to /usr/local and installed with paths relative to /usr/local/stow/*application*. Encap uses

[2]The AtheOS operating system uses a similar technique to maintain compatibility with UNIX applications

```
/                             Root directory
|--Programs                   Programs
|--Mount                      Mounting point for additional local or remote filesystems
|--Users                      Personal areas for users
|--System
|    |--Boot                  Files necessary for booting (kernel and bootloader)
|    |--Links
|    |    |--Executables      Links to files from the programs' bin and sbin directories
|    |    |--Headers          Links to files from the programs' include directories
|    |    |--Libraries        Links to files from the programs' lib directories
|    |    \--Manuals
|    |         |--info        Links to files from the programs' info directories
|    |         \--man{1-9}    Links to files from the programs' man/man{1-9} directories
|    |--Settings              Configuration files and links to files from Settings directories
|    \--Variable              Variable data
|         \--Temp             Temporary files
|--proc                       Kernel status files (managed by the proc file system)
\--dev                        Device files (managed by the dev file system)
```

Figure 1: The GoboLinux directory hierarchy

similar systematics, with a rudimentary support for version control (the package manager software, *epkg*, tries to detect versions through the name of the application created under `/usr/local/encap`, for example, `sed-2.0` and `sed-3.0.2`).

4 The GoboLinux hierarchy

The basic idea behind the GoboLinux hierarchy is to combine ideas from the previously presented operating systems and the link system introduced by Depot, creating a new hierarchy that maintains total compatibility with the UNIX tree. Like in Depot, each program (*Gimp*, *Fileutils*, *Glibc*, *Qt*, etc.) is installed in its entirety inside a separate directory. Inside this directory, standard UNIX directories are typically created, containing files that in the traditional UNIX directory tree would be copied to `/usr/bin` (or `/bin`), `/usr/sbin` (or `/sbin`), `/usr/lib` (or `/lib`), and so on.

For this purpose, a `/Programs` directory was estabilished, containing a subdirectory for each installed program. Each of those subdirectories has, on its turn, a subdirectory for each version of the specific program, and a link labeled `Current` pointing to the currently used version. Each program also has a `Settings` directory storing the program's configuration files (which would be normally stored at `/etc`). Notice that this directory is unique for all versions of the program. This design choice eases the version control, since the personal configurations are preserved in the event of an upgrade or downgrade.

Installation of programs from source is made through scripts. In the case of programs with configuration files generated using *GNU autoconf* (which is the majority of free software packages) the script uses the `--prefix` parameter to define the destination of the files being installed. For example, when installing the *Qt* library, version 2.3.2, the command `configure --prefix=/Programs/Qt/2.3.3` is automatically executed, and the required directories are created. For programs that do not provide this parameter in their configuration files, the *configure* script (or the *Makefiles* themselves) will be adjusted, automatically, through scripts that attempt to replace the paths contained in the file; or, in a few cases, manually.

Once a new version of a program is installed, links are created from each file in the application at directories that centralize those links according to file types. For example, `/System/Links/Executables` stores links for executable files of all programs (contained in the `bin` and `sbin` directories). This way, all executables can be called or accessed from a single directory (same happening to th libraries, headers and manuals / info files). This approach is different from the ones used by the symbolic link management systems discussed on Section 3, as the generated links structure does not reflect the UNIX hierarchy, but a functional categorization of the links.

Figure 1 describes in greater detail the general directory hierarchy as proposed. An important treat of this structure is the nonexistence of a global `share` directory (programs can have their own `share` directories). This decision is explained by the fact that, even though the `share` directory permits sharing of data between different applications, in practice this directory serves a repository for application-specific files that has no place on the UNIX hierarchy (such as icons and fonts). This way, we opted not to have a `/System/Links/Shared` directory, because the different `share` directories of each application have no relation between them.

Compatibility with the UNIX legacy is obtained through creation of extra links not present in the above diagram, such as `/etc -> /System/Settings`, `/bin -> /System/Links/Executables` and `/lib -> /System/Links/Libraries`, mirroring the GoboLinux tree into the UNIX tree. Unlike previous proposals that attempted to organize the directory tree maintaining historical compatibility, in GoboLinux there is a single point for installation of programs, without a legacy tree in parallel.

5 Experience

Practical experience with the directory structure presented in this paper can be divided in two parts. Firstly, the project was started off with a package based distribution which was gradually converted to the GoboLinux tree as the programs were updated to new versions. In this stage, structural ideas evolved and their validity and viability were verified.

In a second stage, we opted to compile the entire system following the GoboLinux hierarchy, intending to have total control of all files installed in the machine. We used as a basis the documentation created by the "Linux from Scratch" project ([6]).

6 Concluding remarks

As presented in Section 3, the search for alternatives to reorganize the UNIX directory tree is a subject of several projects. This paper presented a new proposition for the UNIX directory structure, differing from previously existing projects mainly because there is no legacy tree coexisting with the main tree, allowing for a greater self-consistency and elegance. Practical experience showed the total viability and compatibility of the presented ideas, as well as making evident the benefits brought by this new structure.

Further information about GoboLinux can be obtained at `http://www.gobolinux.org`.

References

[1] Apple Computer Inc., *"Inside Mac OS X - System Overview"*, ISBN: 1400524806, 292p., Fev. 2001.

[2] *"Encap Archive"*, Computing and Communications Services Office at the University of Illinois at Urbana-Champaign, `http://www.encap.org`, 2002.

[3] Eric Raymond, *"The Cathedral & The Bazaar"*, O'Reilly and Associates, hardback edition, Jan. 2001.

[4] Guillaume Morin, Bob Glickstein, *"GNU Stow"*, `http://www.gnu.org/software/stow/stow.html`, 2001.

[5] Kurt Skauen, *"AtheOS"*, `http://www.atheos.cx`, 2002.

[6] Gerard Beekmans, *"Linux From Scratch"*, `http://www.linuxfromscratch.org/`, acessado em 17/03/2002

[7] Wallace Colyer, Walter Wong, *"Depot: A Tool for Managing Software Environments"*, Usenix LISA VI Conference, 1992.