I am not clueless

or

Myths and misconceptions about the design of GoboLinux

Hisham H. Muhammad

13th June 2004

"Those who do not understand Unix
are doomed to reinvent it, poorly."
– Henry Spencer, 1987

This week we had another release of GoboLinux, and again a number of people, even if indirectly, called me "clueless" for coming up with such a structure for a Linux distribution, for a number of reasons. None of those reasons was new; I heard all of them many times. This article is an attempt to sum them up, and explain why I chose the design decisions I made, hopefully clearing any pending misconceptions. I don't have illusion this will prevent them keep happening, but at least I'll have a text to point people to. This article ranges from common misconceptions from those who have never used GoboLinux, to well-intentioned but poorly-thought-out ideas that keep coming from time to time to the GoboLinux mailing list, often causing long debates. I'll be separating the points in sections and they are meant to be self-contained, so feel free to skip directly to the ones that interest you, if you don't feel like reading the whole thing.

"There is a reason why things are the way they are"

This is something I hear constantly, often followed by an explanation about the difference between `/`, `/usr` and `/usr/local`, and/or `/bin` and `/sbin`. I do understand the difference[1]. If I did away with this three-level distinction, is because I believe there are other ways to approach the problems this distinction tries to solve. In a GoboLinux system, the argument for having separate `/usr` and `/usr/local` trees in order to separate programs shipped by the distribution and compiled by the user clearly does not hold. Each program is naturally separated, and this was the prime intention of creating GoboLinux in the first place.

[1] For those who always wondered (and those who want to check if I really understand the difference), the three main trees divide, respectively, files that need to be available in the root partition for single-user rescue mode, program files managed by the distribution, and programs added separately by the site admin (this distinction varies on different Unices, but this is mainly how it works on Linux distributions, read on for more). Regular programs go in `/bin`, and programs intended for the superuser in `/sbin`.

The historical reason why Unix systems have some of its tree directly at the root partition (`/bin`, `/lib`, `/sbin`) as opposed to having it under `/usr`, is because this way you can boot in a bare-bones single-user rescue mode using those files only, in order to fix problems in the `/usr` tree. This is arcane. 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. I understand the rationale for having a bare-bones rescue mode decades ago, but we have a better solution in our hands now.

The distinction between `bin` and `sbin` makes no sense, in the present context. Historical evolution led to crazy arbitrary distinctions, like `ping` and `traceroute` lying in different directories (I fail to see how can they be of distinct "program classes", by any measure). Unix systems have a permissions system. If one wants only the superuser to be able to run a command, then `chmod 700` it. I suspect the separation could have been conceived to reduce the number of programs in the `$PATH` of regular users. In today's Linux systems, having 400 or 500 programs in your `$PATH`, does not make any difference.

There is one last argument, however, that is still valid for Linux systems to day: partitioning and remote mounting. Those two are really different shades of the same color, with remote mounting being, to my eyes, the most valid concern of those two. I've seen arguments about this among the lines of "hard drives today are cheap, and you'll most likely have all software installed locally anyway, for performance". I agree with this, but I also understand the ones who'd like to maintain things centralized for administrative purposes. But imposing additional complexity on the overall system because of one particular scenario is usually not a good thing, and even then, the traditional Unix solution is not general enough: what if you have three or four application servers? You'll mount one at `/usr`, one at `/opt`, and then what? There goes the traditional Unix tree. In fact, in most of the larger Unix networks I had contact with, particular needs of the site configuration led to non-standard directories added to the Unix tree.

Fortunately, like with the live CD, we have nowadays a technological advancement that serves as a real solution to the problem: *union mounts*, also known as *overlay filesystems*. The idea is that you can mount several partitions in the same directory. This way, the semantics of `/Programs` as "the collection of all programs available in the system" is retained, independently of the physical location of the actual data. File systems are all about abstraction (we don't refer to files based on their track, sector and cylinder address), this progresses a step further. Overlay filesystems are very flexible: the sysadmin, for example, can overlay site-specific settings for an application on top of the defaults exported over NFS. Unfortunately, it is not in widespread use, for reasons beyond by understanding. The Plan 9 operating system has it as one of its basic filesystem operations: the `bind` command (in Plan 9, for example, you don't need a `$PATH` variable, because all directories containing executables are "bound" in a single directory). There is an implementation of an overlay filesystem for Linux: `ovlfs`.

The alleged user-friendliness of longer names

Many, many people, when they stumble upon GoboLinux, look at the long, descriptive directory names and say "Look! They changed the Linux directory names by making them longer and descriptive to make the system friendly!". There is some people who say this as if this were a good thing, and some people who say this as if this were a bad thing. Both are

wrong.

There is a number of reasons why the names in GoboLinux are the way they are, and none of them is "to attract new users who are scared by `/etc` and the like". The number one reason is: to not conflict with the Unix namespace. And when I say Unix namespace I actually mean the Linux namespace, which is not a very well settled thing. This is not like a set of reserved keywords from a programming language that says not to use `if`, `while`, `repeat`, etc. as variable names and the rest is okay. You never know what directories, files and programs will show up tomorrow, so the best I could do was to pick names that were very unlikely to be ever used. Others did that before me, and that worked, so I followed their example: NeXT and Mac OS X had to make their own directories coexist with Unix directories, so they capitalized the names, and while they were at it, they used full words instead of abbreviations. The abbreviations were a sign of the times from the origins of Unix. Dennis Ritchie once said that if he could go back in time and change only one thing in Unix, he'd rename the `creat` system call to `create`.

The one thing that reassures me that my decision was right is that, when we started with GoboLinux, back in the days of Linux 2.4.something, someone asked me "why didn't you pick `/sys` instead of `/System`? That would be easier to type". You can guess what would have happened, now that the kernel guys reserved `/sys` for their own use. In fact, the concerns on typing-friendliness always comes up in discussions about the GoboLinux tree. To that, I can only respond that, in a properly configured shell like the one that comes by default with GoboLinux, typing `/Programs` takes the exact same number of keystrokes as typing `/usr`: slash, lowercase `p`, Tab.

One could also ask: but why change the directories, for starters? Why not simply use the regular directory tree and make it behave like GoboLinux? Yes, I suspect it could be possible, but from an operating system design standpoint, I don't like the idea. I am not comfortable with the concept of a system where well-known directories have different semantics to those that most people expect. AtheOS, for example, has this problem. You see a `/usr` directory, but that is no `/usr`. In AtheOS, it behaves more like `/opt`, but unexplicably keeping the name that historically stood for "user" and then was turned into a backronym for Unix System Resources. Even if Kurt Skauen called it `/opt`, it would still be strange; those are not "optional packages".

The GoboLinux directories, too, have different semantics from the Unix directories. `/Programs` is the collection of all programs available in the system, where each subdirectory contains all files from a given program (the distinction of a program package is up to the developers of each project; the various tools from `CoreUtils` form a single program). Each subdirectory in /System/Links contains a *view* (in the database sense of the term) of each file class from the programs collection: libraries, executables, headers, and so on[2]. You see, these directories are not the Unix directories, they function differently, from an administrative point of view. I believe it is good design to make this explicit in the names.

For strict compatibility reasons, however, we have an extra set of symbolic links with the Unix names pointing to the closest GoboLinux equivalents (even making a few concessions in the GoboLinux side of the equation in order to preserve this compatibility). The fact that these are links, and we call them the *legacy* tree keeps this notion very clear. The work of Lucas Villa-Real and Felipe Damasio on GoboHide, the kernel patch for true hidden directories

[2]On Plan 9 that could be done with a `bind` command; since we don't have this in a vanilla Linux kernel, we do this with symbolic links. But I have to admit I like the notion that the views are stored in a persistent manner.

on Linux, further isolates the legacy tree as an isolated accessory.

Do you want to change the standard?

Of course not. For starters, we're not that naive to think that we could. But the actual reason why we don't want to change the standard is because we believe *there should be no standard*. I know this statement may sound even bolder than talking about changing a standard, but the reason I say that is because we believe it is the duty of each application to allow itself to be installed anywhere and to accept that other applications it needs to work with may be installed anywhere (more on this in the next section). Now, if there was a standard stating this, I'd even sign a petition to support it. In fact, there is: the GNU release standards, when they recommend the usage of GNU Autotools, supporting the `--prefix` family of switches, and probing for the location of applications with the configure script, do just that. But when a proposed standard like the FHS gives me an arbitrary list of binaries that should be, for unexplained reasons, in a separate directory, I laugh at that.

Different situations imply different needs, and so-called standards that attempt to fit every feet in the same shoe are doomed to failure. Standardize on flexibility instead. That's not we don't propose the GoboLinux tree as a standard to be followed by anyone else. In five, ten or twenty years, we may have completely different needs from the ones we have today. I don't want that the move away from the GoboLinux tree then to be as hard as the move away from the traditional Unix tree is today. Which leads us naturally to our next section...

An uphill battle to change all applications

This is not as hard as it seems. Before the first version of GoboLinux was fully built, I had already worked on and improved this model for about a whole year. When André Detsch and I got around to build, in two days, a system from scratch built around those concepts, I already knew that this was perfectly feasible.

I work in an university environment, and I have for many years. There, I am not the superuser, so I have to install every extra app I need in my `$HOME` directory. This is a perfectly common situation, it is expected that any decent application will allow this, and the vast majority of them do. In fact, one could argue that an app that doesn't has a broken build system. If you can install Gimp on `/usr`, or `/opt`, or `/home/hisham`, then you can install it on `/Programs/Gimp/2.0/`. Experience has shown that very few applications need to have their Makefiles dissected in order to cooperate. Even superuser-oriented software has (or should have) this flexibility: in a regular Unix system, the superuser should have the option to choose between, say, `/sbin` and `/usr/sbin`. There is no reason to have hardcoded paths in programs and installers[3].

A more delicate problem arises when a program, even though it allows itself to be installed under any directory, wrongly assumes that another programs it depends on is installed under the same directory. As you can guess, this is a major source of problems for GoboLinux, but I advocate this needs to be fixed for the benefit of the entire free software

[3] An installer per se is a rare concept in Linux; what I mean by installer here is usually the `install` target of a Makefile.

community. Let's return to the `$HOME` directory scenario. What if my favorite GNOME component was not installed by the system administrator, and I want to install it in my `$HOME`, while still using the rest of GNOME installed at `/usr`? Situations like this, especially in big multi-component software, is often problematic. There is a number of programs that solve this problem using a `$PATH`-like environment variable: `$GTKPATH`, `$PERL5LIB`, `$KDEDIRS`, `$PYTHON_PATH`, and so on. There is no reason to make a monolithic installation a requirement.

So, the battle GoboLinux is fighting with regard to installation paths is not specific to us; we are only exposing problems on the flexibility of installation of applications, that happen not only in our tree, but anywhere a user has a custom installation need. I see that the situation has improved greatly in the last few years, with more and more projects adopting GNU Autotools.

"It's easier to compile all programs relative to the same tree"

Sure it is. This is a point that comes up from time to time on the GoboLinux mailing list, when people suggest us to either model `/System/Links` after a regular Unix tree, with subdirectories such as `bin`, `lib`, etc., or just compile everything relative to `/usr` and let the legacy tree ensure that everything keeps working. People who suggest this are also implicitly suggesting one of two things: to compile relative to a tree and then install relative to another; or to compile relative to a tree and then use a redirection hack on installation. I don't like any of the two approaches. In the first one, you are expecting a certain flexibility from the build system that is not always there, but unlike the points I raised on the previous section, it is not justifiable that this flexibility *should* be in the application's build system in the first place[4]. As for the second approach, I don't like the idea of an operating system built around a hack that can be at any moment circumvented by a new system call or some unorthodox access method. Some might say that GoboHide, for example, also falls in this "low-level hack" territory. I point out, then, that GoboHide is not mandatory: GoboLinux is designed to work with a vanilla Linux kernel[5].

But instead of pointing flaws in the proposed alternatives, I'd prefer to constructively defend my original design decision. Our idea, with GoboLinux, is to exercise this new approach with self-contained directories and assess its impact on system management, and we have been collecting exciting results. If instead we just used every possible stratagem to make apps "easier to compile", I believe we would be detracting ourselves from this goal. When I run `ls -l /System/Links/Executables` and see *all* executables from my system, and the programs they belong to, I see a clean system design. I would hate to look at `/System/Links` (or whatever the directory would be called) and see within the Unix mess emulated, with `/bin`, `/sbin`, and (`$DEITY` forbid!) `/usr/X11R6`.

"You want to turn Linux into a Windows-like!"

If you read everything up to this point, I believe it should be clear enough that we're not. If we were doing this to attract the Windows users, a structural reorganization would be the

[4]For those interested in this approach, I point you to GNU Stow. I do not know, however, of any system built 100% with it, though.

[5]If it were not, we could also, for example, build the `/System/Links` views as union mounts using `ovlfs`.

last thing we would do. Instead, we would concentrate on making the user interface look like Windows, applying Windows-like themes, moving icons around, perhaps integrating Wine tightly into the distribution, and so on. And that's what Lindows, Lindash, Linspire, or whatever their name is today is doing, not us.

It may sound extremely paradoxical, but we strive to keep the Linux identity on the system. To be more precise, we strive for each project to keep its own identity. Whenever possible, we ship every application with unmodified sources. If you ever took the time to look inside the `.src.rpm` files of any major distribution, you know what I'm talking about: the vast majority of packages have patches to apply little modifications here and there to modify this and that behavior; be it to change the default state of a checkbox, or even to remove the "About" box of an application! We don't do that. Our K menu shows the KDE logo, and so does the KDE splash screen. We do ship a theme with a custom wallpaper, but that is presented as an option in the installer.

We go through great distances to ensure that our packages do not have GoboLinux-specific bugs. The worst thing as a Linux user is to discover that a given software works on distro X and doesn't on distro Y, and not know if that is because distro Y introduced a custom patch that caused the bug, or if it's because distro X introduced a patch that fixed the bug. Speaking now as a developer, this is also a major headache. Alexandre Julliard from Wine once said that the constant changes on Linux distributions slow down the project more than the changes from the Win32 API.

The fact that on GoboLinux all Unix library directories present all libraries from the system, all header directories present all headers, and so on, neutralizes many common compatibility problems between distributions, causing us to be, ironically, one of the most compatible distros, despite the unorthodox directory layout.

The scope of a distribution

Some people, perhaps excited by the fact that we made such a "big change" in the structure of the operating system, occasionaly come to us through the mailing list with this great idea about doing some other big system-wide change that would improve GoboLinux considerably. Sometimes this great idea is applicable, and we do apply them, like when Carlo Calica integrated a daemon managing tool, `Runit`, into GoboLinux[6]. But most of the time the idea is something that would require all applications to be greatly modified, if not rewritten. That is, obviously, something we can't and are not willing to do. If we were talking about a limited number of programs, some of them might even be feasible, but people need to keep in mind that the universe of programs to be used with GoboLinux is potentially infinite, as new Linux apps are written every day.

To list just a few of the unfeasible ideas we were suggested, I could mention:

- make all programs relocatable – I would love to see that, but that would either mean: rewrite every app in the world to use `libprefix`, store every dependency inside each program directory and have some libraries on disk one hundred times (and deal with the clashes that arise in a system that is not prepared for this), compile everything relative to the same tree (I devoted an entire section above for this).

[6]The current trend on GoboLinux development is to loosen this integration, making `Runit` an optional component, but still providing the required framework to make it work.

- make all programs use a unified configuration file format – Of course, the rewrite effort that this would require is even greater than the one from the previous item. Every idea that starts with "make all programs..." tends to have the same problems: the effort is potentially infinite; you can't make every project in the world agree to use your solution; even if you could, there are many legacy systems out there that just can't change even if they wanted, so incompatibility issues would happen no matter what. Another idea of this class would be to make all programs use the same graphic toolkit. Yeah, that would look great and unify the Linux desktop, but it's just not going to happen.

- turn entries on `/Programs` into AppDirs, because they look so much like RiscOS – They may look like, but they are pretty different. Yes, `/Programs/Emacs` could be made into an AppDir. `/Programs/LyX` too. Hey, a lot of them could. But what about `/Programs/FindUtils`? Or `/Programs/KDE`, what would happen when you click on that? Another key issue is that AppDirs are relocatable by nature and GoboLinux packages are not, a problem which I already covered above.

Internationalization

An often raised point is that "changing" names from things like `lib` to things like `Libraries` is too English-centric ("at least the old names were equally meaningless for everyone") and that we should do an effort to make the directory tree translatable. I could dismiss this point raising a number of technical issues that make this impractical, unless we are talking about hacks involving symbolic links and the GoboHide kernel patch. But I won't do that. I will, instead, assume that a clean and elegant way to translate all GoboLinux directories existed, and ask "then what?".

If people are willing to translate the directory tree in order to make the system more friendly to those who don't speak English, I'm sorry, but that won't help. A user that is defeated by the fact that `/System/Settings` is not called `/Sistema/Configurações` won't go very farther, once they reach this directory and need to edit `httpd.conf`. The point I am trying to make is that the kind of users that need internationalization won't be helped by a translated directory tree. Efforts for translation should instead be directed towards documentation and the user interface of programs. If the user can read a manual in his/her language that tells him/her to go to `/System/Settings` and do such and such change in `httpd.conf`, this is much more useful than having the name of the directory changed. If the user has a friendly GUI for configuring Apache like the one provided by Mac OS X, he/she will probably like it much more.

Integration with other distros

This is another point that is raised from time to time, in different shapes, sizes and colors. The one reason I see people leaning towards this idea is because of the huge libraries of ready-to-use software provided by the other distributions. At first sight, the idea of combining all the innovations of GoboLinux with the enormous package base of distro X seems amazing. Looking closer, we'll see it's not.

First of all, there is the issue of the dependency systems. GoboLinux has a very loose dependency system, designed to be resilient to user customizations[7]. If you take advantage of these GoboLinux features, you won't be able to auto-update system of distro X, and vice-versa. This way, you would have to choose between using GoboLinux as if you were using distro X, giving up much of the GoboLinux flexibility, or ignoring the cool auto-update features from distro X. Either way, you would give up on one of the reasons you started this integration project in the first place.

Then, there are all of the little peculiarities of both distributions, which you would have to be constantly dealing with: different boot scripts, possible library incompatibilities, the "value-added" package customizations of distro X... Not to mention the inability to properly use Compile or the GoboLinux binaries repository, due to, for example, different naming conventions of packages.

In short, even if you convert a whole system to use distro X's packages, what you'll end up with is not a "turbo" GoboLinux, but a quirky distro X. It is trivial to take, for example, all RPM's that compose a RedHat system, unpack them, and symlink them to look like a GoboLinux system. The resulting system, in the end, would pretty much be still RedHat. Different people have done this, with different goals (some to build a full distro, others just to convert a binary package or two), with RedHat, Slackware, Debian and more recently Gentoo. The general lesson I learned, from watching them do it, is that it is not worth it.

The `root` user

Of course, I saved the best for last. The decision of naming user zero something other than `root` is among the ones we are most criticized for. The origins of this predate GoboLinux. On my experimental system, my regular user was named `hisham`, and the superuser `lode`. I never liked the Unix notion of "an arbitrary `root` versus regular users" and wanted to see how well a Linux system would behave without a `root` user. After a few adaptations here and there, it worked very well. It was nice to know that every time someone would try to log as `root` in my machine, they would always fail.

When we made the hackathon that resulted in the first version of GoboLinux, André and I decided to keep doing it. We chose `gobo`, an inside joke. The intention of course, was to have a system that could support a non-`root` superuser cleanly, but the users (a handful of people back then) never changed the default and `gobo` somehow got stuck. It is still possible to change the default without much effort, though. For a short while I administered a set of machines at the university, and, to have them blend with the NIS environment more easily (the network was basically composed of RedHat boxes), I changed the superuser from `gobo` back to `root`. Now that GoboLinux has a graphical installer, we are considering putting the superuser name as an installation-time option.

Now that I'm through with the historical explanation, one thing I would like to point out that it is a well-known fact that the existence of a single god-like entity is one of the weaknesses of the Linux security model, and that is what bothered me with the notion of an arbitrary `root` versus the rest of users; it is akin to a single point of failure in a distributed system. The first thing every project aiming to improve the security of Linux does is to in-

[7] It's beyond the scope of this article to describe the GoboLinux dependency system, but for example, if you install app A and it depends on B, it won't complain if B was not installed with a vanilla package, and if B is missing, it will report, but not refuse to install.

crease the granularity of the security model, do dilute the power of `root`: ACLs, capabilities, SELinux... It may be argued that some of those add excessive complexity to the model, but I won't dive into this discussion here. The one thing that is clear is that the `root` model is overly simplistic for today's complex systems, and that the "setuid" kludge is the source of most security issues. Plan 9, for example, doesn't have a superuser at all; it offers a virtualized view of the file system to each process. The `gobo` experiment was an interesting assessment on how ingrained in the Linux world is the expectation on having a `root` user; fortunately, not much (it does not measure how attached the security model is to the user #0, of course). One future direction I would like GoboLinux to take (and in fact Linux in general) is to adopt some of the technologies listed above as a way to improve the control over the system security and administration; to detach ourselves from `root` was the first step in this direction.

Conclusions

Well, I believe I covered a lot of ground in this article. I'm sure I forgot many issues, but I think the most important ones are all here. But the main idea I hope I passed here is that GoboLinux is not just a cosmetic change in the filesystem. We are pretty much aware of what we are doing, and what are the implications of the things we are doing.

It is no secret that when I came up with the first versions of this directory layout, I did not expect it to turn into a Linux distribution used by people all around the world (even though it was shaped as a distribution project as early as when Guilherme Bedin joined). The one thing I'm most happy about is that the original goal, from way back when it was not a proper distro, remains: a clean design.

I wholeheartedly agree with the quote in the beginning of the article, and I definitely believe this is not the case.