

Advanced Filesystem Layout, AFSL Version 1

Magnus Achim Deininger

March 9, 2009

Contents

Preface	3
1 The Advanced Filesystem Layout	4
1.1 General Notes	4
1.2 Hierarchical Elements	4
1.2.1 <i>/(os)?(architecture)?(subarchitecture)*/</i> : Architecture Classification	4
1.2.2 <i>/medium/mediumname[0-9]*/</i> : Storage Media	4
1.2.3 <i>/home/username/</i> : Home Directories	4
1.2.4 <i>/service/servicename/</i> : Services	5
1.2.5 <i>/dev, /proc, /sys, /net, (...)</i> : Special Kernel Filesystems	5
1.3 Final Path Elements	5
1.3.1 <i>./bin</i> : Binaries	5
1.3.2 <i>./lib</i> : Libraries	5
1.3.3 <i>./configuration</i> or <i>./etc</i> : Configuration Files	5
1.3.4 <i>./documentation</i> or <i>./doc</i> : Documentation	5
1.3.5 <i>./temporary</i> or <i>./tmp</i> : Temporary Storage	5
1.3.6 <i>./work</i> : Intermediary Storage	5
1.4 Notable Differences to the FHS	5
1.4.1 <i>./sbin</i> vs <i>./bin</i>	5
1.4.2 <i>/var/tmp</i> vs <i>/work</i>	5
1.4.3 <i>/media</i> vs <i>/medium</i>	5
1.4.4 <i>/var, /opt</i> and <i>/usr</i>	5
2 Usage	6
2.1 Environment Variables	6
3 Issues with Legacy Systems	7
3.1 POSIX Compatibility Considerations	7
3.2 Booting	7
A Path Elements	8
A.1 <i>os</i> : Operating System Codes	8
A.2 <i>architecture</i> : CPU Architecture Codes	8
A.3 <i>subarchitecture</i> : CPU Sub-Architecture Codes	8
A.4 <i>mediumname</i> : Storage Medium Codes	8
A.5 <i>servicename</i> : Service Codes	8
References	9

Preface

Current filesystem layouts employed by contemporary operating systems suffer from a number of shortcomings. Among the more common of these are general inconsistencies and issues with contemporary multi-architecture systems. Minor, similar issues occur when trying to develop for architectures other than that of the host system. In general, it is becoming increasingly common that binaries (executables and libraries) of distinct systems occur together on a single system. Examples for these would include contemporary desktop systems with “64-bit” capabilities mixing binaries compiled for x86-64 and x86-32 on the same filesystem with poor separation, or average Linux installations containing binaries intended for Microsoft Windows for use with WINE, or FreeBSD installations including binaries intended for Linux.

This has led to what may be perceived as inconsistencies in order to fix some of these issues, such as the lib/lib32/lib64 dilemma that would require sane software build scripts to figure out in which of these directories to put created libraries. This alone is already inconsistent with requirements that expressly forbid this -32/-64 extension on bin/ directories¹, and the idea itself seems rather hackish.

Additionally, the original reasoning behind a split of the root filesystem and the /usr hierarchy (and also somewhat the /opt and /usr/local hierarchies) seems to have been to be able to separate machine-local binaries and configuration from site-local binaries and shared data and read-only from read-write data. Originally a physical separation between the hierarchies was necessary, but due to the ability of contemporary operating systems to unify multiple filesystem hierarchies, sometimes called “union mounts”, this division seems quite obsolete².

This paper will try to explore an alternative filesystem hierarchy scheme that should provide consistency for years to come. Keep in mind that this is a draft, not yet a final document.

¹See the FHS document for details.

²Operating Systems such as Plan 9 seem to agree with this.

1 The Advanced Filesystem Layout

The general idea behind this filesystem layout is fairly simple: "traditional" directories are at the end of an arbitrarily long list of host specifications. This leads to a general layout that looks something like this: `os/architecture/subarchitecture/(bin|lib)`. The idea is to get more and more specific along the path, with the operating system being considered less specific than the hardware architecture. This latter choice is fairly random, but it seems to be easier or more common to run binaries from a different architecture on the same operating system than it is to run binaries from a different operating system even when on the same architecture (although there are legitimate examples of both).

1.1 General Notes

All of the elements that vary by target system are written *likethis*. These are always assumed to be lowercase, without spaces or underscores. Use dashes ('-') as a delimiter when needed. For example, for *architecture* use 'x86-64' instead of 'x86_64' or 'power-macintosh' instead of 'power macintosh'. Also, use values that are consistent and generic across all architectures and operating systems. That means, instead of 'power-macintosh', what you really should be using is 'powerpc-32' or 'powerpc-64', and instead of 'amd64' or 'em64t' you should be using 'x86-64'.

1.2 Hierarchical Elements

Paths discussed in this section are prefixes for other parts. Also refer to the appendix for values to be used for common variables in this section.

1.2.1 `/(os/)?(architecture/)?(subarchitecture)*/:` Architecture Classification

os refers to a short code, hopefully a single word that would describe the intended target operating systems. Examples for this would be 'linux' or 'plan9'. *architecture* is the general architecture class of the target host, for example 'x86-32', 'x86-64', 'powerpc-32', etc. Unless there is only one conceivable implementation to refer to, this should always include the GPR or address space length of the architecture. For both *os* and *architecture*, the value 'generic' may be used if it doesn't matter which operating system or cpu architecture some of the content is, but it would still be desirable to keep the parent subdirectory count low and this could be achieved by means of a placeholder such as 'generic'.

subarchitecture is anything that could be used to further specify requirements that the binaries may have. This may be anything as specific as 'sse3' or 'altivec', or something more generic such as 'i686' or just 'little-endian'. System vendor specifications may also be used here, i.e. 'sun', 'intel', 'apple', etc.

As should be evident by the title above, more than one *subarchitecture* specification may be used. They must still be specified in reverse order of importance, that is, more generic specifications must come first, whereas more specific details should be last. This is to facilitate finding applicable files in the hierarchy.

With all of these, redundant specifications may be omitted. For example, if the target system will always be a linux system, then it's allowed to drop the `/linux` prefix.

Now, when actually locating applicable files or hierarchies, it should also be noted that more specific, applicable subhierarchies always take precedence over less specific but nevertheless applicable hierarchies. That is to say, files in `/linux/x86-32/sse2/bin` take precedence over `/linux/x86-32/bin` or even just `/bin`. This makes optimisations a lot more effective and easy, by being able to put binaries that are more heavily optimised for a certain cpu model in the filesystem without interfering with the other binaries on the system.

1.2.2 `/medium/mediumname[0-9]*/:` Storage Media

Mountpoints for storage media belong under this

1.2.3 `/home/username/:` Home Directories

You may use the architecture classification as discussed above in subdirectories below here if necessary.

1.2.4 `/service/servicename/`: Services

You may use the architecture classification as discussed above in subdirectories below here if necessary.

1.2.5 `/dev, /proc, /sys, /net, (...)`: Special Kernel Filesystems

These obviously have operating-system specific meaning and layout. They usually need to be kept in place (for example `sysfs` on Linux is specifically documented as being required to be mounted on `/sys`). Obviously only nodes that are actually going to be mounted need to be created on a filesystem, but having more than necessary doesn't seem to hurt, so it's unspecified whether or not unavailable special filesystems still have their respective directories created or not.

1.3 Final Path Elements

Paths in this section are the ending parts of a sequence of path elements.

1.3.1 `./bin`: Binaries

All binaries ('executables') go into this directory. No exceptions, not even for "system binaries", which would traditionally end up in `/sbin`. If you need finer-grained control or systematic clustering of related binaries, use a subdirectory in `/service` instead.

1.3.2 `./lib`: Libraries

1.3.3 `./configuration` or `./etc`: Configuration Files

1.3.4 `./documentation` or `./doc`: Documentation

1.3.5 `./temporary` or `./tmp`: Temporary Storage

1.3.6 `./work`: Intermediary Storage

1.4 Notable Differences to the FHS

1.4.1 `./sbin` vs `./bin`

`/sbin` is pointless: system binaries may sound like a nice thing to hide from users, but deciding which of the binaries are system binaries and which are regular binaries is fairly arbitrary. For example the 'ifconfig' programme is often put in `/sbin` instead of `/bin`, although there are legitimate uses of ifconfig for regular users. The programme's query functions immediately come to mind. Since this functionality works fine for regular users, they can just access the programme by entering the complete path, so the "hiding" was both point- and fruitless.

Additionally the split is nearly impossible to perform systematically on operating systems that have finer-grained access control than a simple `root/user` divide. Operating systems like Plan9 or Linux³ come to mind. With these operating systems, half the users would probably end up with `/sbin` in their path anyway since they'd have legitimate use for some system programme or another.

For these reasons, there is no `/sbin` with this layout. If you really want to hide system binaries, use filesystem namespaces and union mounts instead, but remember that hiding them really is pointless.

1.4.2 `/var/tmp` vs `/work`

1.4.3 `/media` vs `/medium`

`/media` was quite an odd inconsistency. Virtually everything else used words or abbreviations in the singular form, this is about the only place where a plural was used. The idea to split it up was good though, so there we go.

1.4.4 `/var, /opt` and `/usr`

³This may not seem obvious at first, but internally the Linux kernel is documented to use so-called 'capabilities' that provide fine-grained access control to privileged functions

2 Usage

2.1 Environment Variables

3 Issues with Legacy Systems

There are some obvious incompatibilities with legacy filesystem layouts, at least when employing the more descriptive paths.

3.1 POSIX Compatibility Considerations

The most notable problem to be encountered with live multi-host setups is the likely lack of an appropriate `/bin/sh` binary. It should be possible to circumvent this issue with either automatically created symlinks on boot or with strategic application of namespaces.

It should be noted that this problem is easily underestimated. Current BSD and Linux systems, for example, employ a large amount of shell scripting for common functionality, including things such as the “init scripts” that are responsible for bringing up system services. In these script files, the first line starts with a shabang, followed by the path to a script interpreter. Since POSIX guarantees that `/bin/sh` is a valid shell script interpreter, scripts ‘out in the wild’ will rely on this and reference this location. The same applies to the `system()` function in the standard C library.

3.2 Booting

During bootup the kernel needs to find and execute any binaries on the root filesystem that are required to boot into userspace. With most other filesystem layouts, this is easily done, as the binaries were usually always in the same, so it’s just a matter of hardcoding the location, however with this new layout it’s not quite that easy anymore, as the location of the binary might depend on the host the computer is run on. (‘might depend’, because `/bin/init` would still be a legitimate path with this filesystem layout.)

There would be a number of ways to go about this issue. One of them would be by means of a small bootstrap partition or `initramfs` or similar with appropriate binaries in a fixed location. This has the advantage of not requiring any modifications to the kernel itself and it should always work. However, it’d be quite a pain in the neck to maintain this separate, boot-only filesystem.

An alternative would be making the bootloader pass a parameter to the kernel with the location of an appropriate init programme. This should be possible by default with a good deal of popular kernels and bootloaders, such as FreeBSD or Linux with GRUB or LILO, and is frequently done in practice with systems employing “alternative init systems”. Since the bootloader needs to know the appropriate kernel to execute on the host system anyway, it is very likely that it would be possible to specify a binary that would at least execute properly on the target machine.

The kernel itself could also be modified to search for a suitable binary to execute. This method is fairly invasive, as it requires direct modification of kernel source code, but it would nevertheless be possible and provide a clean and final solution to the issue.

A Path Elements

The values listed here are to be considered mandatory to follow. If you don't like some of them, you can still omit them most of the time, or suggest better alternatives.

A.1 *os*: **Operating System Codes**

A.2 *architecture*: **CPU Architecture Codes**

A.3 *subarchitecture*: **CPU Sub-Architecture Codes**

A.4 *mediumname*: **Storage Medium Codes**

A.5 *servicename*: **Service Codes**

References