HOWTO: reduce dependencies of OSS packages

Contributed by Michael Felt

Reduce Dependencies? - Am I misleading you?

What I am really talking about is reducing the so-called "dependency-hell" when installing an OSS package. Somethimes they cannot avoided - but sometimes they can!

Shared Libraries - both a boon and a burden

When an OSS package uses another package this introduces a dependency. This means that to build, or install that OSS package the software it depends on needs to be installed first as well. When that also has a dependency the looping through packages to find (and install) grows.

A library such as libc.so (on Linix) or libc.a (on AIX) is extremely handy to have because, basically, if there is any part that depends on the C language it will need a "helper" function or procedure from libc.

But when I am building a package and it has a dependency on something obscure, or older than the "latest" it can become quite difficult to also build (and package) and install another OSS project that depends on a later version. In an ideal world the OSS package still working with an old version will update their sources so that they work with the new versions of the dependencies. But it doesn't always work that way.

And
sometimes, it is just a hassle to keep 20+ different "dependencies" with their dependencies all ordered and tested. The idea is that a dependency (say OpenSSL) can just be updated and all the OSS packages that depend on OpenSSL are updated without any additional work. Often, that works.

Bored with specifying a long list of dependencies

That is my other problem. I would like, ideally, to just have one file that has all it needs - within itself. Once upon a time - and I remember it fairly well (mid 1980's) the concept of shared libraries was discussed as a way to lower system resource requirements - both memory and disk. FYI: in those days a disk of 20 MByte was huge - and system memory > 128K byte was huge.

Yes - I know my applications might be smaller, and the total amount in system memory could also be smaller, but I am actually trying - as much as possible to use static linked libraries rather than shared libraries. And, from many of the comments I read, sometimes between the lines on the "ld" man pages - static-linked programs can (if not should) run faster than programs with MANY rtld (run-time loader) symbol resolutions.

Impact?

The packages I am rebuilding (e.g., httpd, php) are having fewer dependancies. I have already learned that the way apache apr and apr-util are designed - they do not work as static with apache httpd and shared modules. The modules (many many of them) need stuff from apr and/or apr-util that the httpd "main" did not need and these functions and procedures turn up missing. Building the modules with static links to the functions means that some of the symbols get loaded multiple times - and httpd core dumps. Sigh.

So, yes. There is an impact. But the impact I am looking for is software packages that are easier to maintain - for a "consumer" such as an AIX admin (like you?) even if it means more work in the evening getting all the pieces together.

Update: Seems some packages won't work with static libraries - because not everything they need is loaded - or more accurate - other packages need that "static" API as well, but there is overlap - and more. So while partX is fine, trying to use partY causes a core dump because the same routines get statically linked - and a bit more. Then the same routines are statically linked multiple times - and their static variables and routines are not the ones that get run 'all the time'.

My goal is not changed - but getting there is much harder than I had hoped. Sigh!

My goal

More stable packages and fewer (ideally none) for most OSS projects - rather than havig to download and install 4 or 5 other things before "what it is all about" can finally be installed.