Dynamic Shared Object (DSO) Support

The Apache HTTP Server is a modular program where the administrator can choose the functionality to include in the server by selecting a set of modules. Modules will be compiled as Dynamic Shared Objects (DSOs) that exist separately from the main `httpd` binary file. DSO modules may be compiled at the time the server is built, or they may be compiled and added at a later time using the Apache Extension Tool (`apxs`).

Alternatively, the modules can be statically compiled into the `httpd` binary when the server is built.

This document describes how to use DSO modules as well as the theory behind their use.

Implementation

Related Modules	Related Directives
mod_so	LoadModule

The DSO support for loading individual Apache httpd modules is based on a module named `mod_so` which must be statically compiled into the Apache httpd core. It is the only module besides `core` which cannot be put into a DSO itself. Practically all other distributed Apache httpd modules will then be placed into a DSO. After a module is compiled into a DSO named `mod_foo.so` you can use `mod_so`'s `LoadModule` directive in your `httpd.conf` file to load this module at server startup or restart.

The DSO builds for individual modules can be disabled via `configure`'s `--enable-mods-static` option as discussed in the install documentation.

To simplify this creation of DSO files for Apache httpd modules (especially for third-party modules) a support program named `apxs` (*APache eXtenSion*) is available. It can be used to build DSO based modules *outside of* the Apache httpd source tree. The idea is simple: When installing Apache HTTP Server the `configure`'s `make install` procedure installs the Apache httpd C header files and puts the platform-dependent compiler and linker flags for building DSO files into the `apxs` program. This way the user can use `apxs` to compile his Apache httpd module sources without the Apache httpd distribution source tree and without having to fiddle with the platform-dependent compiler and linker flags for DSO support.

Usage Summary

To give you an overview of the DSO features of Apache HTTP Server 2.x, here is a short and concise summary:

1. Build and install a *distributed* Apache httpd module, say `mod_foo.c`, into its own DSO `mod_foo.so`:

```
$ ./configure --prefix=/path/to/install --enable-foo
$ make install
```

2. Configure Apache HTTP Server with all modules enabled. Only a basic set will be loaded during server startup. You can change the set of loaded modules by activating or deactivating the `LoadModule` directives in `httpd.conf`.

```
$ ./configure --enable-mods-shared=all
$ make install
```

3. Some modules are only useful for developers and will not be build. when using the module set *all*. To build all available modules including developer modules use *reallyall*. In addition the `LoadModule` directives for all built modules can be activated via the configure option `--enable-load-all-modules`.

```
$ ./configure --enable-mods-shared=reallyall --
enable-load-all-modules
$ make install
```

4. Build and install a *third-party* Apache httpd module, say `mod_foo.c`, into its own DSO `mod_foo.so` *outside of* the Apache httpd source tree using `apxs`:

```
$ cd /path/to/3rdparty
$ apxs -cia mod_foo.c
```

In all cases, once the shared module is compiled, you must use a `LoadModule` directive in `httpd.conf` to tell Apache httpd to activate the module.

See the apxs documentation for more details.

Background

On modern Unix derivatives there exists a mechanism called dynamic linking/loading of *Dynamic Shared Objects* (DSO) which provides a way to build a piece of program code in a special format for loading it at run-time into the address space of an executable program.

This loading can usually be done in two ways: automatically by a system program called `ld.so` when an executable program is started or manually from within the executing program via a programmatic system interface to the Unix loader through the system calls `dlopen()/dlsym()`.

In the first way the DSO's are usually called *shared libraries* or *DSO libraries* and named `libfoo.so` or `libfoo.so.1.2`. They reside in a system directory (usually `/usr/lib`) and the link to the executable program is established at build-time by specifying `-lfoo` to the linker command. This hard-codes library references into the executable program file so that at start-time the Unix loader is able to locate `libfoo.so` in `/usr/lib`, in paths hard-coded via linker-options like `-R` or in paths configured via the environment variable `LD_LIBRARY_PATH`. It then resolves any (yet unresolved) symbols in the executable program which are available in the DSO.

Symbols in the executable program are usually not referenced by the DSO (because it's a reusable library of general code) and hence no further resolving has to be done. The executable program has no need to do anything on its own to use the symbols from the DSO because the complete resolving is done by the Unix loader. (In fact, the code to invoke `ld.so` is part of the run-time startup code which is linked into every executable program which has been bound non-static). The advantage of dynamic loading of common library code is obvious: the library code needs to be stored only once, in a system library like `libc.so`, saving disk space for every program.

In the second way the DSO's are usually called *shared objects* or *DSO files* and can be named with an arbitrary extension (although the canonical name is `foo.so`). These files usually stay inside a program-specific directory and there is no automatically established link to the executable program where they are used. Instead the executable program manually loads the DSO at run-time into its

address space via `dlopen()`. At this time no resolving of symbols from the DSO for the executable program is done. But instead the Unix loader automatically resolves any (yet unresolved) symbols in the DSO from the set of symbols exported by the executable program and its already loaded DSO libraries (especially all symbols from the ubiquitous `libc.so`). This way the DSO gets knowledge of the executable program's symbol set as if it had been statically linked with it in the first place.

Finally, to take advantage of the DSO's API the executable program has to resolve particular symbols from the DSO via `dlsym()` for later use inside dispatch tables *etc.* In other words: The executable program has to manually resolve every symbol it needs to be able to use it. The advantage of such a mechanism is that optional program parts need not be loaded (and thus do not spend memory) until they are needed by the program in question. When required, these program parts can be loaded dynamically to extend the base program's functionality.

Although this DSO mechanism sounds straightforward there is at least one difficult step here: The resolving of symbols from the executable program for the DSO when using a DSO to extend a program (the second way). Why? Because "reverse resolving" DSO symbols from the executable program's symbol set is against the library design (where the library has no knowledge about the programs it is used by) and is neither available under all platforms nor standardized. In practice the executable program's global symbols are often not re-exported and thus not available for use in a DSO. Finding a way to force the linker to export all global symbols is the main problem one has to solve when using DSO for extending a program at run-time.

The shared library approach is the typical one, because it is what the DSO mechanism was designed for, hence it is used for nearly all types of libraries the operating system provides.

Advantages and Disadvantages

The above DSO based features have the following advantages:

- The server package is more flexible at run-time because the server process can be assembled at run-time via `LoadModule` `httpd.conf` configuration directives instead of `configure` options at build-time. For instance, this way one is able to run different server instances (standard & SSL version, minimalistic & dynamic version [mod_perl, mod_php], *etc.*) with only one Apache httpd installation.
- The server package can be easily extended with third-party modules even after installation. This is a great benefit for vendor package maintainers, who can create an Apache httpd core package and additional packages containing extensions like PHP, mod_perl, mod_security, *etc.*
- Easier Apache httpd module prototyping, because with the DSO/`apxs` pair you can both work outside the Apache httpd source tree and only need an `apxs -i` command followed by an `apachectl restart` to bring a new version of your currently developed module into the running Apache HTTP Server.

DSO has the following disadvantages:

- The server is approximately 20% slower at startup time because of the symbol resolving overhead the Unix loader now has to do.
- The server is approximately 5% slower at execution time under some platforms, because position independent code (PIC) sometimes needs complicated assembler tricks for relative addressing, which are not necessarily as fast as absolute addressing.
- Because DSO modules cannot be linked against other DSO-based libraries (`ld -lfoo`) on all platforms (for instance a.out-based platforms usually don't provide this functionality while ELF-based platforms do) you cannot

use the DSO mechanism for all types of modules. Or in other words, modules compiled as DSO files are restricted to only use symbols from the Apache httpd core, from the C library (`libc`) and all other dynamic or static libraries used by the Apache httpd core, or from static library archives (`libfoo.a`) containing position independent code. The only chances to use other code is to either make sure the httpd core itself already contains a reference to it or loading the code yourself via `dlopen()`.