1   Notes on the design and goals of mod_perl-2.0

1.1 Description

Notes on the design and goals of mod_perl-2.0.

We try to keep this doc in sync with the development, so some items discussed here were already implemented, while others are only planned. If you find some inconsistencies in this document please let the list know.

1.2 Introduction

In version 2.0 of mod_perl, the basic concept of 1.0 still applies:

```
Provide complete access to the Apache C API
via the Perl programming language.
```

Rather than "porting" mod_perl-1.0 to Apache 2.0, mod_perl-2.0 is being implemented as a complete re-write from scratch.

For a more detailed introduction and functionality overview, see Overview.

1.3 Interpreter Management

In order to support mod_perl in a multi-threaded environment, mod_perl-2.0 will take advantage of Perl's *ithreads* feature, new to Perl version 5.6.0. This feature encapsulates the Perl runtime inside a thread-safe *PerlInterpreter* structure. Each thread which needs to serve a mod_perl request will need its own *PerlInterpreter* instance.

Rather than create a one-to-one mapping of *PerlInterpreter* per-thread, a configurable pool of interpreters is managed by mod_perl. This approach will cut down on memory usage simply by maintaining a minimal number of intepreters. It will also allow re-use of allocations made within each interpreter by recycling those which have already been used. This was not possible in the 1.3.x model, where each child has its own interpreter and no control over which child Apache dispatches the request to.

The interpreter pool is only enabled if Perl is built with -Dusethreads otherwise, mod_perl will behave just as 1.0, using a single interpreter, which is only useful when Apache is configured with the prefork mpm.

When the server is started, a Perl interpreter is constructed, compiling any code specified in the configuration, just as 1.0 does. This interpreter is referred to as the "parent" interpreter. Then, for the number of *PerlInterpStart* configured, a (thread-safe) clone of the parent interpreter is made (via perl_clone()) and added to the pool of interpreters. This clone copies any writeable data (e.g. the symbol table) and shares the compiled syntax tree. From my measurements of a *startup.pl* including a few random modules:

```
use CGI ();
use POSIX ();
use IO ();
use SelfLoader ();
use AutoLoader ();
use B::Deparse ();
use B::Terse ();
use B ();
use B::C ();
```

The parent adds 6M size to the process, each clone adds less than half that size, ~2.3M, thanks to the shared syntax tree.

NOTE: These measurements were made prior to finding memory leaks related to perl_clone() in 5.6.0 and the GvSHARED optimization.

At request time, If any Perl*Handlers are configured, an available interpreter is selected from the pool. As there is a *conn_rec* and *request_rec* per thread, a pointer is saved in either the conn_rec->pool or request_rec->pool, which will be used for the lifetime of that request. For handlers that are called when threads are not running (PerlChild{Init,Exit}Handler), the parent interpreter is used. Several configuration directives control the interpreter pool management:

- **PerlInterpStart**

  The number of intepreters to clone at startup time.

- **PerlInterpMax**

  If all running interpreters are in use, mod_perl will clone new interpreters to handle the request, up until this number of interpreters is reached. when PerlInterpMax is reached, mod_perl will block (via COND_WAIT()) until one becomes available (signaled via COND_SIGNAL())

- **PerlInterpMinSpare**

  The minimum number of available interpreters this parameter will clone interpreters up to PerlIn-terpMax, before a request comes in.

- **PerlInterpMaxSpare**

  mod_perl will throttle down the number of interpreters to this number as those in use become available

- **PerlInterpMaxRequests**

  The maximum number of requests an interpreter should serve, the interpreter is destroyed when the number is reached and replaced with a fresh one.

- **PerlInterpScope**

As mentioned, when a request in a threaded mpm is handled by mod_perl, an interpreter must be pulled from the interpreter pool. The interpreter is then only available to the thread that selected it, until it is released back into the interpreter pool. By default, an interpreter will be held for the lifetime of the request, equivalent to this configuration:

```
PerlInterpScope request
```

For example, if a `PerlAccessHandler` is configured, an interpreter will be selected before it is run and not released until after the logging phase.

Intepreters will be shared across subrequests by default, however, it is possible to configure the intepreter scope to be per-subrequest on a per-directory basis:

```
PerlInterpScope subrequest
```

With this configuration, an autoindex generated page for example would select an interpreter for each item in the listing that is configured with a Perl*Handler.

It is also possible to configure the scope to be per-handler:

```
PerlInterpScope handler
```

With this configuration, an interpreter will be selected before `PerlAccessHandlers` are run, and putback immediately afterwards, before Apache moves onto the authentication phase. If a `Perl-FixupHandler` is configured further down the chain, another interpreter will be selected and again putback afterwards, before `PerlResponseHandler` is run.

For protocol handlers, the interpreter is held for the lifetime of the connection. However, a C protocol module might hook into mod_perl (e.g. mod_ftp) and provide a `request_rec` record. In this case, the default scope is that of the request. Should a mod_perl handler want to maintain state for the lifetime of an ftp connection, it is possible to do so on a per-virtualhost basis:

```
PerlInterpScope connection
```

1.3.1 TIPool

The interpreter pool is implemented in terms of a "TIPool" (Thread Item Pool), a generic api which can be reused for other data such as database connections. A Perl interface will be provided for the *TIPool* mechanism, which, for example, will make it possible to share a pool of DBI connections.

1.3.2 Virtual Hosts

The interpreter management has been implemented in a way such that each `<VirtualHost>` can have its own parent Perl interpreter and/or MIP (Mod_perl Interpreter Pool). It is also possible to disable mod_perl for a given virtual host.

1.3.3 Further Enhancements

- The interpreter pool management could be moved into its own thread.

- A "garbage collector", which could also run in its own thread, examining the padlists of idle inter-
  preters and deciding to release and/or report large strings, array/hash sizes, etc., that Perl is keeping
  around as an optimization.

1.4 Hook Code and Callbacks

The code for hooking mod_perl in the various phases, including `Perl*Handler` directives is generated
by the `ModPerl::Code` module. Access to all hooks will be provided by mod_perl in both the tradi-
tional `Perl*Handler` configuration fashion and via dynamic registration methods (the ap_hook_* func-
tions).

When a mod_perl hook is called for a given phase, the glue code has an index into the array of handlers,
so it knows to return DECLINED right away if no handlers are configured, without entering the Perl
runtime as 1.0 did. The handlers are also now stored in an apr_array_header_t, which is much lighter and
faster than using a Perl AV, as 1.0 did. And more importantly, keeps us out of the Perl runtime until we're
sure we need to be there.

`Perl*Handlers` are now "compiled", that is, the various forms of:

```
PerlResponseHandler MyModule->handler
# defaults to MyModule::handler or MyModule->handler
PerlResponseHandler MyModule
PerlResponseHandler $MyObject->handler
PerlResponseHandler 'sub { print "foo\n"; return OK }'
```

are only parsed once, unlike 1.0 which parsed every time the handler was used. There will also be an
option to parse the handlers at startup time. Note: this feature is currently not enabled with threads, as each
clone needs its own copy of Perl structures.

A "method handler" is now specified using the 'method' sub attribute, e.g.

```
sub handler : method {};
```

instead of 1.0's

```
sub handler ($$) {}
```

1.5 Perl interface to the Apache API and Data Structures

In 1.0, the Perl interface back into the Apache API and data structures was done piecemeal. As functions
and structure members were found to be useful or new features were added to the Apache API, the xs code
was written for them here and there.

The goal for 2.0 is to generate the majority of xs code and provide thin wrappers where needed to make the API more Perlish. As part of this goal, nearly the entire APR and Apache API, along with their public data structures is covered from the get-go. Certain functions and structures which are considered "private" to Apache or otherwise un-useful to Perl don't get glued.

The Apache header tree is parsed into Perl data structures which live in the generated `Apache::FunctionTable` and `Apache::StructureTable` modules. For example, the following function prototype:

```
 AP_DECLARE(int) ap_meets_conditions(request_rec *r);
```

is parsed into the following Perl structure:

```
  {
    'name' => 'ap_meets_conditions'
    'return_type' => 'int',
    'args' => [
      {
        'name' => 'r',
        'type' => 'request_rec *'
      }
    ],
  },
```

and the following structure:

```
 typedef struct {
     uid_t uid;
     gid_t gid;
 } ap_unix_identity_t;
```

is parsed into:

```
  {
    'type' => 'ap_unix_identity_t'
    'elts' => [
      {
        'name' => 'uid',
        'type' => 'uid_t'
      },
      {
        'name' => 'gid',
        'type' => 'gid_t'
      }
    ],
  }
```

Similar is done for the mod_perl source tree, building `ModPerl::FunctionTable` and `ModPerl::StructureTable`.

Three files are used to drive these Perl structures into the generated xs code:

- **lib/ModPerl/function.map**

  Specifies which functions are made available to Perl, along with which modules and classes they reside in. Many functions will map directly to Perl, for example the following C code:

  ```
  static int handler (request_rec *r) {
      int rc = ap_meets_conditions(r);
      ...
  ```

  maps to Perl like so:

  ```
  sub handler {
      my $r = shift;
      my $rc = $r->meets_conditions;
  ...
  ```

  The function map is also used to dispatch Apache/APR functions to thin wrappers, rewrite arguments and rename functions which make the API more Perlish where applicable. For example, C code such as:

  ```
  char uuid_buf[APR_UUID_FORMATTED_LENGTH+1];
  apr_uuid_t uuid;
  apr_uuid_get(&uuid)
  apr_uuid_format(uuid_buf, &uuid);
  printf("uuid=%s\n", uuid_buf);
  ```

  is remapped to a more Perlish convention:

  ```
  printf "uuid=%s\n", APR::UUID->new->format;
  ```

- **lib/ModPerl/structure.map**

  Specifies which structures and members of each are made available to Perl, along with which modules and classes they reside in.

- **lib/ModPerl/type.map**

  This file defines how Apache/APR types are mapped to Perl types and vice-versa. For example:

  ```
  apr_int32_t => SvIV
  apr_int64_t => SvNV
  server_rec  => SvRV (Perl object blessed into the Apache::Server class)
  ```

1.5.1 Advantages to generating XS code

- Not tied tightly to xsubpp

- Easy adjustment to Apache 2.0 API/structure changes

- Easy adjustment to Perl changes (e.g., Perl 6)

- Ability to "discover" hookable third-party C modules.

- Cleanly take advantage of features in newer Perls

- Optimizations can happen across-the-board with one-shot

- Possible to AUTOLOAD XSUBs

- Documentation can be generated from code

- Code can be generated from documentation

1.5.2 Lvalue methods

A new feature to Perl 5.6.0 is *lvalue subroutines*, where the return value of a subroutine can be directly modified. For example, rather than the following code to modify the uri:

```
$r->uri($new_uri);
```

the same result can be accomplished with the following syntax:

```
$r->uri = $new_uri;
```

mod_perl-2.0 will support *lvalue subroutines* for all methods which access Apache and APR data structures.

1.6 Filter Hooks

mod_perl 2.0 provides two interfaces to filtering, a direct mapping to buckets and bucket brigades and a simpler, stream-oriented interface. This is discussed in the Chapter on filters.

1.7 Directive Handlers

mod_perl 1.0 provides a mechanism for Perl modules to implement first-class directive handlers, but requires an XS file to be generated and compiled. The 2.0 version provides the same functionality, but does not require the generated XS module (i.e. everything is implemented in pure Perl).

1.8 <Perl> Configuration Sections

The ability to write configuration in Perl carries over from 1.0, but but implemented much different internally. The mapping of a Perl symbol table fits cleanly into the new *ap_directive_t* API, unlike the hoop jumping required in mod_perl 1.0.

1.9 Protocol Module Support

Protocol module support is provided out-of-the-box, as the hooks and API are covered by the generated code blankets. Any functionality for assisting protocol modules should be folded back into Apache if possible.

1.10 mod_perl MPM

It will be possible to write an MPM (Multi-Processing Module) in Perl. mod_perl will provide a mod_perl_mpm.c framework which fits into the server/mpm standard convention. The rest of the functionality needed to write an MPM in Perl will be covered by the generated xs code blanket.

1.11 Build System

The biggest mess in 1.0 is mod_perl's Makefile.PL, the majority of logic has been broken down and moved to the `Apache::Build` module. The *Makefile.PL* will construct an `Apache::Build` object which will have all the info it needs to generate scripts and *Makefile*s that apache-2.0 needs. Regardless of what that scheme may be or change to, it will be easy to adapt to with build logic/variables/etc., divorced from the actual *Makefile*s and configure scripts. In fact, the new build will stay as far away from the Apache build system as possible. The module library (*libmodperl.so* or *libmodperl.a*) is built with as little help from Apache as possible, using only the `INCLUDEDIR` provided by *apxs*.

The new build system will also "discover" XS modules, rather than hard-coding the XS module names. This allows for switchabilty between static and dynamic builds, no matter where the xs modules live in the source tree. This also allows for third-party xs modules to be unpacked inside the mod_perl tree and built static without modification to the mod_perl Makefiles.

For platforms such as Win32, the build files are generated similar to how unix-flavor *Makefile*s are.

1.12 Test Framework

Similar to 1.0, mod_perl-2.0 provides a 'make test' target to exercise as many areas of the API and module features as possible.

The test framework in 1.0, like several other areas of mod_perl, was cobbled together over the years. mod_perl 2.0 provides a test framework that is usable not only for mod_perl, but for third-party `Apache::*` modules and Apache itself. See `Apache::Test`.

1.13 CGI Emulation

As a side-effect of embedding Perl inside Apache and caching compiled code, mod_perl has been popular as a CGI accelerator. In order to provide a CGI-like environment, mod_perl must manage areas of the runtime which have a longer lifetime than when running under mod_cgi. For example, the `%ENV` environment variable table, `END` blocks, `@INC` include paths, etc.

CGI emulation is supported in mod_perl 2.0, but done so in a way that it is encapsulated in its own handler. Rather than 1.0 which uses the same response handler, regardless if the module requires CGI emulation or not. With an *ithreads* enabled Perl, it's also possible to provide more robust namespace protection.

Notice that `ModPerl::Registry` is used instead of 1.0's `Apache::Registry`, and similar for other registry groups. `ModPerl::RegistryCooker` makes it easy to write your own customizable registry handler.

1.14 Apache::* Library

The majority of the standard `Apache::*` modules in 1.0 are supported in 2.0. The main goal being that the non-core CGI emulation components of these modules are broken into small, re-usable pieces to subclass Apache::Registry like behavior.

1.15 Perl Enhancements

Most of the following items were projected for inclusion in perl 5.8.0, but that didn't happen. While these enhancements do not preclude the design of mod_perl-2.0, they could make an impact if they were implemented/accepted into the Perl development track.

1.15.1 GvSHARED

(Note: This item wasn't implemented in Perl 5.8.0)

As mentioned, the perl_clone() API will create a thread-safe interpreter clone, which is a copy of all mutable data and a shared syntax tree. The copying includes subroutines, each of which take up around 255 bytes, including the symbol table entry. Multiply that number times, say 1200, is around 300K, times 10 interpreter clones, we have 3Mb, times 20 clones, 6Mb, and so on. Pure perl subroutines must be copied, as the structure includes the `PADLIST` of lexical variables used within that subroutine. However, for XSUBs, there is no PADLIST, which means that in the general case, perl_clone() will copy the subroutine, but the structure will never be written to at runtime. Other common global variables, such as `@EXPORT` and `%EXPORT_OK` are built at compile time and never modified during runtime.

Clearly it would be a big win if XSUBs and such global variables were not copied. However, we do not want to introduce locking of these structures for performance reasons. Perl already supports the concept of a read-only variable, a flag which is checked whenever a Perl variable will be written to. A patch has been submitted to the Perl development track to support a feature known as `GvSHARED`. This mechanism allows XSUBs and global variables to be marked as shared, so perl_clone() will not copy these structures, but rather point to them.

1.15.2 Shared SvPVX

The string slot of a Perl scalar is known as the `SvPVX`. As Perl typically manages the string a variable points to, it must make a copy of it. However, it is often the case that these strings are never written to. It would be possible to implement copy-on-write strings in the Perl core with little performance overhead.

1.15.3 Compile-time method lookups

A known disadvantage to Perl method calls is that they are slower than direct function calls. It is possible to resolve method calls at compile time, rather than runtime, making method calls just as fast as subroutine calls. However, there is certain information required for method look ups that are only known at runtime. To work around this, compile-time hints can be used, for example:

```
 my Apache::Request $r = shift;
```

Tells the Perl compiler to expect an object in the `Apache::Request` class to be assigned to `$r`. A patch has already been submitted to use this information so method calls can be resolved at compile time. However, the implementation does not take into account sub-classing of the typed object. Since the mod_perl API consists mainly of methods, it would be advantageous to re-visit the patch to find an acceptable solution.

1.15.4 Memory management hooks

Perl has its own memory management system, implemented in terms of *malloc* and *free*. As an optimization, Perl will hang onto allocations made for variables, for example, the string slot of a scalar variable. If a variable is assigned, for example, a 5k chunk of HTML, Perl will not release that memory unless the variable is explicitly *undef*ed. It would be possible to modify Perl in such a way that the management of these strings are pluggable, and Perl could be made to allocate from an APR memory pool. Such a feature would maintain the optimization Perl attempts (to avoid malloc/free), but would greatly reduce the process size as pool resources are able to be re-used elsewhere.

1.15.5 Opcode hooks

Perl already has internal hooks for optimizing opcode trees (syntax tree). It would be quite possible for extensions to add their own optimizations if these hooks were plugable, for example, optimizing calls to *print*, so they directly call the Apache *ap_rwrite* function, rather than proxy via a *tied filehandle*.

Another optimization that was implemented is "inlined" XSUB calls. Perl has a generic opcode for calling subroutines, one which does not know the number of arguments coming into and being passed out of a subroutine. As the majority of mod_perl API methods have known in/out argument lists, mod_perl implements a much faster version of the Perl *pp_entersub* routine.

1.16 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

Doug MacEachern <dougm (at) covalent.net>

1.17 Authors

- **Doug MacEachern <dougm (at) covalent.net>**

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: