Developer’s guide

This guide is aimed for mod_perl 2.0 core and 3rd party modules developers.

Last modified Sun Feb 16 01:36:36 2014 GMT

Part V: Help

1 mod_perl 2.0 Source Code Explained

1.1 Description

This document explains how to navigate the mod_perl source code, modify and rebuild the existing code and most important: how to add new functionality.

1.2 Project's Filesystem Layout

In its pristine state the project is comprised of the following directories and files residing at the root directory of the project:

```
Apache-Test/      - test kit for mod_perl and Apache2::* modules
ModPerl-Registry/ - ModPerl::Registry sub-project
build/            - utilities used during project build
docs/             - documentation
lib/              - Perl modules
src/              - C code that builds libmodperl.so
t/                - mod_perl tests
todo/             - things to be done
util/             - useful utilities for developers
xs/               - source xs code and maps
Changes           - Changes file
LICENSE           - ASF LICENSE document
Makefile.PL       - generates all the needed Makefiles
```

After building the project, the following root directories and files get generated:

```
Makefile     - Makefile
WrapXS/      - autogenerated XS code
blib/        - ready to install version of the package
```

1.3 Directory src

1.3.1 Directory src/modules/perl/

The directory *src/modules/perl* includes the C source files needed to build the *libmodperl* library.

Notice that several files in this directory are autogenerated during the *perl Makefile* stage.

When adding new source files to this directory you should add their names to the `@c_src_names` variable in *lib/ModPerl/Code.pm*, so they will be picked up by the autogenerated *Makefile*.

1.4 Directory xs/

```
Apache2/                - Apache specific XS code
APR/                    - APR specific XS code
ModPerl/                - ModPerl specific XS code
maps/                   -
tables/                 -
Makefile.PL             -
```

```
modperl_xs_sv_convert.h  -
modperl_xs_typedefs.h    -
modperl_xs_util.h        -
typemap                  -
```

1.4.1 xs/Apache2, xs/APR and xs/ModPerl

The *xs/Apache2*, *xs/APR* and *xs/ModPerl* directories include *.h* files which have C and XS code in them. They all have the *.h* extension because they are always `#include-d`, never compiled into their own object file. and only the file that `#include-s` an *.h* file from these directories should be able to see what's in there. Anything else belongs in a *src/modules/perl/foo.c* public API.

1.4.2 xs/maps

The *xs/maps* directory includes mapping files which describe how Apache Perl API should be constructed and various XS typemapping.

These files get modified whenever:

- a new function is added or the API of the existing one is modified.
- a new struct is added or the existing one is modified
- a new C datatype or Perl typemap is added or an existing one is modified.

The execution of:

```
% make source_scan
```

or:

```
% perl build/source_scan.pl
```

converts these map files into their Perl table representation in the *xs/tables/current/* directory. This Perl representation is then used during `perl Makefile.PL` to generate the XS code in the *./WrapXS/* directory by the xs_generate() function. This XS code is combined of the Apache API Perl glue and mod_perl specific extensions.

If you need to skip certain unwanted C defines from being picked by the source scanning you can add them to the array `$Apache2::ParseSource::defines_unwanted` in *lib/Apache2/ParseSource.pm*.

Notice that *source_scan* target is normally not run during the project build process, since the source scanning is not stable yet, therefore everytime the map files change, `make source_scan` should be run manually and the updated files ending up in the *xs/tables/current/* directory should be committed to the svn repository.

lib/ModPerl/CScan.pm requires Data::Flow from CPAN which is used by *build/source_scan.pl*

There are three different types of map files in the *xs/maps/* directory:

- **Functions Mapping**

```
apache_functions.map
modperl_functions.map
apr_functions.map
```

- **Structures Mapping**

```
apache_structures.map
apr_structures.map
```

- **Types Mapping**

```
apache_types.map
apr_types.map
modperl_types.map
```

The following sections describe the syntax of the files in each group

1.4.2.1 Functions Mapping

The functions mapping file is comprised of groups of function definitions. Each group starts with a header similar to XS syntax:

```
MODULE=... PACKAGE=... PREFIX=... BOOT=... ISA=...
```

where:

- **`MODULE`**

  the module name where the functions should be put. e.g. `MODULE Apache2::Connection` will place the functions into *WrapXS/Apache2/Connection.{pm,xs}*.

- **`PACKAGE`**

  the package name functions belong to, defaults to `MODULE`. The value of *guess* indicates that package name should be guessed based on first argument found that maps to a Perl class. If the value is not defined and the function's name starts with *ap_* the `Apache2` package will be used, if it starts with *apr_* then the `APR` package is used.

- **`PREFIX`**

  prefix string to be stripped from the function name. If not specified it defaults to `PACKAGE`, converted to C name convention, e.g. `APR::Base64` makes the prefix: *apr_base64_*. If the converted prefix does not match, defaults to *ap_* or *apr_*.

- **BOOT**

  The BOOT directive tells the XS generator, whether to add the boot function to the autogenerated XS file or not. If the value of BOOT is not true or it's simply not declared, the boot function won't be added.

  If the value is true, a boot function will be added to the XS file. Note, that this function is not declared in the map file.

  The boot function name must be constructed from three parts:

  ```
  'mpxs_' . MODULE . '_BOOT'
  ```

  where MODULE is the one declared with MODULE= in the map file.

  For example if we want to have an XS boot function for a class APR::IO, we create this function in *xs/APR/IO/APR__IO.h*:

  ```
  static void mpxs_APR__IO_BOOT(pTHX)
  {
     /* boot code here */
  }
  ```

  and now we add the BOOT=1 declaration to the *xs/maps/modperl_functions.map* file:

  ```
  MODULE=APR::IO PACKAGE=APR::IO BOOT=1
  ```

  Notice that the PACKAGE= declaration is a must.

  When *make xs_generate* is run (after running *make source_scan*), it autogenerates *Wrap/APR/IO/IO.xs* and amongst other things will include:

  ```
  BOOT:
      mpxs_APR__IO_BOOT(aTHXo);
  ```

- **ISA**

  META: complete

Every function definition is declared on a separate line (use \ if the line is too long), using the following format:

```
C function name | Dispatch function name | Argspec | Perl alias
```

where:

- **C function name**

  The name of the real C function.

Function names that do not begin with `/^\w/` are skipped. For details see: `%ModPerl::MapUtil::disabled_map`.

The return type can be specified before the C function name. It defaults to *return_type* in `{Apache2,ModPerl}::FunctionTable`.

META: DEFINE nuances

- **Dispatch function name**

  Dispatch function name defaults to C function name. If the dispatch name is just a prefix (*mpxs_*, *MPXS_*) the C function name is appended to it.

  See the explanation about function naming and arguments passing.

- **Argspec**

  The argspec defaults to arguments in `{Apache2,ModPerl}::FunctionTable`. Argument types can be specified to override those in the `FunctionTable`. Default values can be specified, e.g. `arg=default_value`. Argspec of `...` indicates *passthru*, calling the function with `(aTHX_ I32 items, SP **sp, SV **MARK)`.

- **Perl alias**

  the Perl alias will be created in the current `PACKAGE`.

1.4.2.2 Structures Mapping

See %ModPerl::MapUtil::disabled_map in lib/ModPerl/MapUtil.pm

META: complete

1.4.2.3 Types Mapping

META: complete

1.4.2.4 Modifying Maps

As explained in the beginning of this section, whenever the map file is modified you need first to run:

```
% make source_scan
```

Next check that the conversion to Perl tables is properly done by verifying the resulting corresponding file in *xs/tables/current*. For example *xs/maps/modperl_functions.map* is converted into *xs/tables/current/ModPerl/FunctionTable.pm*.

If you want to do a visual check on how XS code will be generated, run:

```
% make xs_generate
```

and verify that the autogenerated XS code under the directory *./WrapXS* is correct. Notice that for functions, whose arguments or return types can't be resolved, the XS glue won't be generated and a warning will be printed. If that's the case add the missing type's typemap to the types map file as explained in Adding Typemaps for new C Data Types and run the XS generation stage again.

You can also build the project normally:

```
% perl Makefile.PL ...
```

which runs the XS generation stage.

1.4.3 XS generation process

As mentioned before XS code is generated in the *WrapXS* directory either during `perl Makefile.PL` via xs_generate() if `MP_GENERATE_XS=1` is used (which is the default) or explicitly via:

```
% make xs_generate
```

In addition it creates a number of files in the *xs/* directory:

```
modperl_xs_sv_convert.h
modperl_xs_typedefs.h
```

1.5 Gluing Existing APIs

If you have an API that you simply want to provide the Perl interface without writing any code...

META: complete

WrapXS allows you to adjust some arguments and supply default values for function arguments without writing any code

META: complete

`MPXS_` functions are final `XSUBs` and always accept:

```
aTHX_ I32 items, SP **sp, SV **MARK
```

as their arguments. Whereas `mpxs_` functions are either intermediate thin wrappers for the existing C functions or functions that do something by themselves. `MPXS_` functions also can be used for writing thin wrappers for C macros.

1.6 Adding Wrappers for existing APIs and Creating New APIs

In certain cases the existing APIs need to be adjusted. There are a few reasons for doing this.

First, is to make the given C API more Perlish. For example C functions cannot return more than one value, and the pass by reference technique is used. This is not Perlish. Perl has no problem returning a list of value, and passing by reference is used only when an array or a hash in addition to any other variables need to be passes or returned from the function. Therefore we may want to adjust the C API to return a list rather than passing a reference to a return value, which is not intuitive for Perl programmers.

Second, is to adjust the functionality, i.e. we still use the C API but may want to adjust its arguments before calling the original function, or do something with return values. And of course optionally adding some new code.

Third, is to create completely new APIs. It's quite possible that we need more functionality built on top of the existing API. In that case we simply create new APIs.

The following sections discuss various techniques for retrieving function arguments and returning values to the caller. They range from using usual C argument passing and returning to more complex Perl arguments' stack manipulation. Once you know how to retrieve the arguments in various situations and how to put the return values on the stack, the rest is usually normal C programming potentially involving using Perl APIs.

Let's look at various ways we can declare functions and what options various declarions provide to us:

1.6.1 Functions Returning a Single Value (or Nothing)

If its know deterministically what the function returns and there is only a single return value (or nothing is returned == *void*), we are on the C playground and we don't need to manipulate the returning stack. However if the function may return a single value or nothing at all, depending on the inputs and the code, we have to manually manipulate the stack and therefore this section doesn't apply.

Let's look at various requirements and implement these using simple examples. The following testing code exercises the interfaces we are about to develop, so refer to this code to see how the functions are invoked from Perl and what is returned:

```
file:t/response/TestApache2/coredemo.pm
----------------------------------------
package TestApache2::coredemo;

use strict;
use warnings FATAL => 'all';

use Apache2::Const -compile => 'OK';

use Apache::Test;
use Apache::TestUtil;
```

```
use Apache2::CoreDemo;

sub handler {
    my $r = shift;

    plan $r, tests => 7;

    my $a = 7;
    my $b = 3;
    my ($add, $subst);

    $add = Apache2::CoreDemo::print($a, $b);
    t_debug "print";
    ok !$add;

    $add = Apache2::CoreDemo::add($a, $b);
    ok t_cmp($a + $b, $add, "add");

    $add = Apache2::CoreDemo::add_sv($a, $b);
    ok t_cmp($a + $b, $add, "add: return sv");

    $add = Apache2::CoreDemo::add_sv_sv($a, $b);
    ok t_cmp($a + $b, $add, "add: pass/return svs");

    ($add, $subst) = @{ Apache2::CoreDemo::add_subst($a, $b) };
    ok t_cmp($a + $b, $add,   "add_subst: add");
    ok t_cmp($a - $b, $subst, "add_subst: subst");

    $subst = Apache2::CoreDemo::subst_sp($a, $b);
    ok t_cmp($a - $b, $subst, "subst via SP");

    Apache2::Const::OK;
}

1;
```

The first case is the simplest: pass two integer arguments, print these to the STDERR stream and return nothing:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
static MP_INLINE
void mpxs_Apache2__CoreDemo_print(int a, int b)
{
    fprintf(stderr, "%d, %d\n", a, b);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
 mpxs_Apache2__CoreDemo_print
```

Now let’s say that the *b* argument is optional and in case it wasn’t provided, we want to use a default value, e.g. 0. In that case we don’t need to change the code, but simply adjust the map file to be:

```
file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
 mpxs_Apache2__CoreDemo_print | | a, b=0
```

In the previous example, we didn't list the arguments in the map file since they were automatically retrieved from the source code. In this example we tell WrapXS to assign a value of `0` to the argument b, if it wasn't supplied by the caller. All the arguments must be listed and in the same order as they are defined in the function.

You may add an extra test that test teh default value assignment:

```
    $add = Apache2::CoreDemo::add($a);
    ok t_cmp($a + 0, $add, "add (b=0 default)");
```

The second case: pass two integer arguments and return their sum:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
static MP_INLINE
int mpxs_Apache2__CoreDemo_add(int a, int b)
{
    return a + b;
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
 mpxs_Apache2__CoreDemo_add
```

The third case is similar to the previous one, but we return the sum as as a Perl scalar. Though in C we say SV*, in the Perl space we will get a normal scalar:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
static MP_INLINE
SV *mpxs_Apache2__CoreDemo_add_sv(pTHX_ int a, int b)
{
    return newSViv(a + b);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
 mpxs_Apache2__CoreDemo_add_sv
```

In the second example the XSUB function was converting the returned *int* value to a Perl scalar behind the scenes. In this example we return the scalar ourselves. This is of course to demonstrate that you can return a Perl scalar, which can be a reference to a complex Perl datastructure, which we will see in the fifth example.

The forth case demonstrates that you can pass Perl variables to your functions without needing XSUB to do the conversion. In all previous examples XSUB was automatically converting Perl scalars in the argument list to the corresponding C variables, using the typemap definitions.

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
static MP_INLINE
SV *mpxs_Apache2__CoreDemo_add_sv_sv(pTHX_ SV *a_sv, SV *b_sv)
{
    int a = (int)SvIV(a_sv);
    int b = (int)SvIV(b_sv);

    return newSViv(a + b);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
 mpxs_Apache2__CoreDemo_add_sv_sv
```

So this example is the same simple case of addition, though we manually convert the Perl variables to C variables, perform the addition operation, convert the result to a Perl Scalar of kind *IV* (Integer Value) and return it directly to the caller.

In case where more than one value needs to be returned, we can still implement this without directly manipulating the stack before a function returns. The fifth case demonstrates a function that returns the result of addition and substruction operations on its arguments:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
static MP_INLINE
SV *mpxs_Apache2__CoreDemo_add_subst(pTHX_ int a, int b)
{
    AV *av = newAV();

    av_push(av, newSViv(a + b));
    av_push(av, newSViv(a - b));

    return newRV_noinc((SV*)av);
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
 mpxs_Apache2__CoreDemo_add_subst
```

If you look at the corresponding testing code:

```
    ($add, $subst) = @{ Apache2::CoreDemo::add_subst($a, $b) };
    ok t_cmp($a + $b, $add,   "add_subst: add");
    ok t_cmp($a - $b, $subst, "add_subst: subst");
```

you can see that this technique comes at a price of needing to dereference the return value to turn it into a list. The actual code is very similar to the `Apache2::CoreDemo::add_sv` function which was doing only the addition operation and returning a Perl scalar. Here we perform the addition and the substraction operation and push the two results into a previously created *AV** data structure, which represents an array. Since only the *SV* datastructures are allowed to be put on stack, we take a reference *RV* (which is of an *SV* kind) to the existing *AV* and return it.

The sixth case demonstrates a situation where the number of arguments or their types may vary and aren't known at compile time. Though notice that we still know that we are returning at compile time (zero or one arguments), *int* in this example:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
static MP_INLINE
int mpxs_Apache2__CoreDemo_subst_sp(pTHX_ I32 items, SV **MARK, SV **SP)
{
    int a, b;

    if (items != 2) {
        Perl_croak(aTHX_ "usage: ...");
    }

    a = mp_xs_sv2_int(*MARK);
    b = mp_xs_sv2_int(*(MARK+1));

    return a - b;
}

file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
 mpxs_Apache2__CoreDemo_subst_sp | | ...
```

In the map file we use a special token `...` which tells the XSUB constructor to pass `items`, `MARK` and `SP` arguments to the function. The macro `MARK` points to the first argument passed by the caller in the Perl namespace. For example to access the second argument to retrieve the value of `b` we use `*(MARK+1)`, which if you remember represented as an *SV* variable, which nees to be converted to the corresponding C type.

In this example we use the macro *mp_xs_sv2_int*, automatically generated based on the data from the *xs/typemap* and *xs/maps/*_types.map* files, and placed into the *xs/modperl_xs_sv_convert.h* file. In the case of *int* C type the macro is:

```
#define mp_xs_sv2_int(sv) (int)SvIV(sv)
```

which simply converts the *SV* variable on the stack and generates an *int* value.

While in this example you have an access to the stack, you cannot manipulate the return values, because the XSUB wrapper expects a single return value of type *int*, so even if you put something on the stack it will be ignored.

1.6.2 Functions Returning Variable Number of Values

We saw earlier that if we want to return an array one of the ways to go is to return a reference to an array as a single return value, which fits the C paradigm. So we simply declare the return value as SV*.

This section talks about cases where it's unknown at compile time how many return values will be or it's known that there will be more than one return value--something that C cannot handle via its return mechanism.

Let's rewrite the function mpxs_Apache2__CoreDemo_add_subst from the earlier section to return two results instead of a reference to a list:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
static XS(MPXS_Apache2__CoreDemo_add_subst_sp)
{
    dXSARGS;
    int a, b;

    if (items != 2) {
        Perl_croak(aTHX_ "usage: Apache2::CoreDemo::add_subst_sp($a, $b)");
    }
    a = mp_xs_sv2_int(ST(0));
    b = mp_xs_sv2_int(ST(1));

    SP -= items;

    if (GIMME == G_ARRAY) {
      EXTEND(sp, 2);
      PUSHs(sv_2mortal(newSViv(a + b)));
      PUSHs(sv_2mortal(newSViv(a - b)));
    }
    else {
        XPUSHs(sv_2mortal(newSViv(a + b)));
    }

    PUTBACK;
}
```

Before explaining the function here is the prototype we add to the map file:

```
file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
DEFINE_add_subst_sp | MPXS_Apache2__CoreDemo_add_subst_sp | ...
```

The mpxs_ functions declare in the third column the arguments that they expect to receive (and optionally the default values). The MPXS functions are the real XSUBs and therefore they always accept:

```
aTHX_ I32 items, SP **sp, SV **MARK
```

as their arguments. Therefore it doesn't matter what is placed in this column when the MPXS_ function is declared. Usually for documentation the Perl side arguments are listed. For example you can say:

```
DEFINE_add_subst_sp | MPXS_Apache2__CoreDemo_add_subst_sp | x, y
```

In this function we manually manipulate the stack to retrieve the arguments passed on the Perl side and put the results back onto the stack. Therefore the first thing we do is to initialize a few special variables using the dXSARGS macro defined in *XSUB.h*, which in fact calls a bunch of other macros. These variables help to manipulate the stack. dSP is one of these macros and it declares and initial- izes a local copy of the Perl stack pointer sp which . This local copy should always be accessed as SP.

We retrieve the original function arguments using the ST() macros. ST(0) and ST(1) point to the first and the second argument on the stack, respectively. But first we check that we have exactly two arguments on the stack, and if not we abort the function. The items variable is the function argument.

Once we have retrieved all the arguments from the stack we set the local stack pointer SP to point to the bottom of the stack (like there are no items on the stack):

```
SP -= items;
```

Now we can do whatever processing is needed and put the results back on the stack. In our example we return the results of addition and substraction operations if the function is called in the list context. In the scalar context the function returns only the result of the addition operation. We use the GIMME macro which tells us the context.

In the list context we make sure that we have two spare slots on the stack since we are going to push two items, and then we push them using the PUSHs macro:

```
EXTEND(sp, 2);
PUSHs(sv_2mortal(newSViv(a + b)));
PUSHs(sv_2mortal(newSViv(a - b)));
```

Alternatively we could use:

```
XPUSHs(sv_2mortal(newSViv(a + b)));
XPUSHs(sv_2mortal(newSViv(a - b)));
```

The XPUSHs macro e*X*tends the stack before pushing the item into it if needed. If we plan to push more than a single item onto the stack, it's more efficient to extend the stack in one call.

In the scalar context we push only one item, so here we use the XPUSHs macro:

```
XPUSHs(sv_2mortal(newSViv(a + b)));
```

The last command we call is:

```
PUTBACK;
```

which makes the local stack pointer global. This is a must call if the state of the stack was changed when the function is about to return. The stack changes if something was popped from or pushed to it, or both and changed the number of items on the stack.

In our example we don't need to call PUTBACK if the function is called in the list context. Because in this case we return two variables, the same as two function arguments, the count didn't change. Though in the scalar context we push onto the stack only one argument, so the function won't return what is expected. The simplest way to avoid errors here is to always call PUTBACK when the stack is changed.

For more information refer to the *perlcall* manpage which explains the stack manipulation process in great details.

Finally we test the function in the list and scalar contexts:

```
file:t/response/TestApache2/coredemo.pm
----------------------------------------
...
my $a = 7;
my $b = 3;
my ($add, $subst);

# list context
($add, $subst) = Apache2::CoreDemo::add_subst_sp($a, $b);
ok t_cmp($a + $b, $add,   "add_subst_sp list context: add");
ok t_cmp($a - $b, $subst, "add_subst_sp list context: subst");

# scalar context
$add = Apache2::CoreDemo::add_subst_sp($a, $b);
ok t_cmp($a + $b, $add,   "add_subs_spt scalar context: add");
...
```

1.6.3 Wrappers Functions for C Macros

Let's say you have a C macro which you want to provide a Perl interface for. For example let's take a simple macro which performs the power of function:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
#define mpxs_Apache2__CoreDemo_power(x, y) pow(x, y)
```

To create the XS glue code we use the following entry in the map file:

```
file:xs/maps/modperl_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
double:DEFINE_power | | double:x, double:y
```

This works very similar to the MPXS_Apache2__CoreDemo_add_subst_sp function presented earlier. But since this is a macro the XS wrapper needs to know the types of the arguments and the return type, so these are added. The return type is added just before the function name and separated from it by the colon (:), the argument types are specified in the third column. The type is always separated from the name of the variable by the colon (:).

And of course finally we need to test that the function works in Perl:

```
file:t/response/TestApache2/coredemo.pm
---------------------------------------
...
my $a = 7;
my $b = 3;
my $power = Apache2::CoreDemo::power($a, $b);
ok t_cmp($a ** $b, $power, "power macro");
...
```

1.6.4 Passing aTHX for DEFINE map entries

Let's say you have a function or a C macro which you want to provide a Perl interface for, and you don't need to write a wrapper since C arguments are the same as Perl arguments. For example:

```
char *foo(aTHX_ int bar);
```

The map entry will look like:

```
MODULE=Apache2::CoreDemo
char *:DEFINE_foo | | int:bar
```

But there is no way to pass `aTHX_` since this is a macro and it's an empty string with non-threaded Perls. Another macro comes to help:

```
file:xs/Apache2/CoreDemo/Apache2__CoreDemo.h
--------------------------------------------
#define mpxs_Apache2__CoreDemo_foo(x, y) foo(aTHX_ x, y)
```

1.7 Wrappers for modperl_, apr_ and ap_ APIs

If you already have a C function whose name starts from *modperl_*, *apr_* or *ap_* and you want to do something before calling the real C function, you can write a XS wrapper using the same method as in the MPXS_Apache2__CoreDemo_add_subst_sp . The only difference is that it'll be clearly seen in the map file that this is a wrapper for an existing C API.

Let's say that we have an existing C function apr_power(), this is how we declare its wrapper:

```
file:xs/maps/apr_functions.map
------------------------------
MODULE=APR::Foo
apr_power | MPXS_ | x, y
```

The first column specifies the existing function's name, the second tells that the XS wrapper will use the `MPXS_` prefix, which means that the wrapper must be called `MPXS_apr_power`. The third column specifies the argument names, but for `MPXS_` no matter what you specify there the `...` will be passed:

```
aTHX_ I32 items, SP **sp, SV **MARK
```

so you can leave that column empty, but here we use `x` and `y` to remind us that these two arguments are passed from Perl.

If the forth column is empty this function will be called `APR::Foo::power` in the Perl namespace. But you can use that column to give a different Perl name, e.g with:

```
apr_power | MPXS_ | x, y | pow
```

This function will be available from Perl as `APR::Foo::pow`.

Similarly you can write a `MPXS_modperl_power` wrapper for a `modperl_power()` function but here you have to explicitly give the Perl function's name in the forth column:

```
file:xs/maps/apr_functions.map
----------------------------------
MODULE=Apache2::CoreDemo
modperl_power | MPXS_ | x, y | mypower
```

and the Perl function will be called `Apache2::CoreDemo::mypower`.

The `MPXS_` wrapper's implementation is similar to MPXS_Apache2__CoreDemo_add_subst_sp .

1.8 MP_INLINE vs C Macros vs Normal Functions

To make the code maintainable and reusable functions and macros are used in when programming in C (and other languages :).

When function is marked as *inlined* it's merely a hint to the compiler to replace the call to a function with the code inside this function (i.e. inlined). Not every function can be inlined. Some typical reasons why inlining is sometimes not done include:

- the function calls itself, that is, is recursive
- the function contains loops such as `for(;;)` or `while()`
- the function size is too large

Most of the advantage of inline functions comes from avoiding the overhead of calling an actual function. Such overhead includes saving registers, setting up stack frames, etc. But with large functions the overhead becomes less important.

Use the `MP_INLINE` keyword in the declaration of the functions that are to be inlined. The functions should be inlined when:

- Only ever called once (the *wrappers* that are called from *.xs* files), no matter what the size of code is.

- Short bodies of code called in a *hot* code (like *modperl_env_hv_store*, which is called many times inside of a loop), where it is cleaner to see the code in function form rather than macro with lots of \'s. Remember that an inline function takes much more space than a normal functions if called from many places in the code.

Of course C macros are a bit faster then inlined functions, since there is not even *short jump* to be made, the code is literally copied into the place it's called from. However using macros comes at a price:

- Also unlike macros, in functions argument types are checked, and necessary conversions are performed correctly. With macros it's possible that weird things will happen if the caller has passed arguments of the wrong type when calling a macro.

- One should be careful to pass only absolute values as *"arguments"* to macros. Consider a macro that returns an absolute value of the passed argument:

  ```
  #define ABS(v) ( (v) >= 0 ? (v) : -(v) )
  ```

  In our example if you happen to pass a function it will be called twice:

  ```
  abs_val = ABS(f());
  ```

  Since it'll be extended as:

  ```
  abs_val = f() >= 0 ? f() : -f();
  ```

  You cannot do simple operation like increment--in our example it will be called twice:

  ```
  abs_val = ABS(i++);
  ```

  Because it becomes:

  ```
  abs_val = i++ >= 0 ? i++ : -i++;
  ```

- It's dangerous to use the if() condition without enclosing the code in `{}`, since the macro may be called from inside another if-else condition, which may cause the else part called if the if() part from the macro fails.

  But we always use `{}` for the code inside the if-else condition, so it's not a problem here.

- A multi-line macro can cause problems if someone uses the macro in a context that demands a single statement.

  ```
  while (foo) MYMACRO(bar);
  ```

  But again, we always enclose any code in conditional with `{}`, so it's not a problem for us.

- Inline functions present a problem for debuggers and profilers, because the function is expanded at the point of call and loses its identity. This makes the debugging process a nightmare.

A compiler will typically have some option available to disable inlining.

In all other cases use normal functions.

1.9 Adding New Interfaces

1.9.1 Adding Typemaps for new C Data Types

Sometimes when a new interface is added it may include C data types for which we don't have corresponding XS typemaps yet. In such a case, the first thing to do is to provide the required typemaps.

Let's add a prototype for the *typedef struct scoreboard* data type defined in *httpd-2.0/include/scoreboard.h*.

First we include the relevant header files in *src/modules/perl/modperl_apache_includes.h*:

```
#include "scoreboard.h"
```

If you want to specify your own type and don't have a header file for it (e.g. if you extend some existing datatype within mod_perl) you may add the *typedef* to *src/modules/perl/modperl_types.h*.

After deciding that `Apache::Scoreboard` is the Perl class will be used for manipulating C *scoreboard* data structures, we map the *scoreboard* data structure to the `Apache::Scoreboard` class. Therefore we add to *xs/maps/apache_types.map*:

```
struct scoreboard       | Apache::Scoreboard
```

Since we want the *scoreboard* data structure to be an opaque object on the perl side, we simply let mod_perl use the default `T_PTROBJ` typemap. After running `make xs_generate` you can check the assigned typemap in the autogenerated *WrapXS/typemap* file.

If you need to do some special handling while converting from C to Perl and back, you need to add the conversion functions to the *xs/typemap* file. For example the `Apache2::RequestRec` objects need special handling, so you can see the special `INPUT` and `OUTPUT` typemappings for the corresponding `T_APACHEOBJ` object type.

Now we run `make xs_generate` and find the following definitions in the autogenerated files:

```
file:xs/modperl_xs_typedefs.h
-----------------------------
typedef scoreboard * Apache__Scoreboard;

file:xs/modperl_xs_sv_convert.h
-------------------------------
#define mp_xs_sv2_Apache__Scoreboard(sv) \
((SvROK(sv) && (SvTYPE(SvRV(sv)) == SVt_PVMG)) \
|| (Perl_croak(aTHX_ "argument is not a blessed reference \
(expecting an Apache::Scoreboard derived object)"),0) ? \
```

```
(scoreboard *)SvIV((SV*)SvRV(sv)) : (scoreboard *)NULL)

#define mp_xs_Apache__Scoreboard_2obj(ptr) \
sv_setref_pv(sv_newmortal(), "Apache::Scoreboard", (void*)ptr)
```

The file *xs/modperl_xs_typedefs.h* declares the typemapping from C to Perl and equivalent to the `TYPEMAP` section of the XS's *typemap* file. The second file *xs/modperl_xs_sv_convert.h* generates two macros. The first macro is used to convert from Perl to C datatype and equivalent to the *typemap* file's `INPUT` section. The second macro is used to convert from C to Perl datatype and equivalent to the *typemap*'s `OUTPUT` section.

Now proceed on adding the glue code for the new interface.

1.9.2 Importing Constants and Enums into Perl API

To *import* httpd and APR constants and enums into Perl API, edit *lib/Apache2/ParseSource.pm*. To add a new type of `DEFINE` constants adjust the `%defines_wanted` variable, for `enums` modify `%enums_wanted`.

For example to import all `DEFINE`s starting with `APR_FLOCK_` add:

```
my %defines_wanted = (
    ...
    APR => {
        ...
        flock     => [qw{APR_FLOCK_}],
        ...
    },
);
```

When deciding which constants are to be exported, the regular expression will be used, so in our example all matches `/^APR_FLOCK_/` will be imported into the Perl API.

For example to import an *read_type_e* `enum` for APR, add:

```
my %enums_wanted = (
    APR => { map { $_, 1 } qw(apr_read_type) },
);
```

Notice that *_e* part at the end of the enum name has gone.

in case of Apache constants remove the leading `ap_` and terminating `/_(t|e)$/`. For example *ap_conn_keepalive_e* needs to be added as:

```
my %enums_wanted = (
    Apache2 => { map { $_, 1 } qw(conn_keepalive) },
);
```

After adding/modifying the datastructures make sure to run `make source_scan` or `perl build/source_scan.pl` and verify that the wanted constant or enum were picked by the source scanning process. Simply grep *xs/tables/current* for the wanted string. For example after adding

apr_read_type_e enum we can check:

```
% more xs/tables/current/Apache2/ConstantsTable.pm
...
  'read_type' => [
    'APR_BLOCK_READ',
    'APR_NONBLOCK_READ'
  ],
```

Of course the newly added constant or enum's typemap should be declared in the appropriate *xs/maps/*_types.map* files, so the XS conversion of arguments will be performed correctly. For example *apr_read_type* is an APR enum so it's declared in *xs/maps/apr_types.map*:

```
apr_read_type           | IV
```

`IV` is used as a typemap, Since enum is just an integer. In more complex cases the typemap can be different. (META: examples)

1.10 Maintainers

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

Stas Bekman [http://stason.org/]

1.11 Authors

- Stas Bekman [http://stason.org/]

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

2 mod_perl internals: Apache 2.0 Integration

2.1 Description

This document should help to understand the initialization, request processing and shutdown process of the mod_perl module. This knowledge is essential for a less-painful debugging experience. It should also help to know where a new code should be added when a new feature is added.

Internals of mod_perl-specific features are discussed in mod_perl internals: mod_perl-specific functionality flow.

Make sure to read also: Debugging mod_perl C Internals.

2.2 Startup

Apache starts itself and immediately restart itself. The following sections discuss what happens to mod_perl during this period.

2.2.1 The Link Between mod_perl and httpd

mod_perl.c includes a special data structure:

```
module AP_MODULE_DECLARE_DATA perl_module = {
    STANDARD20_MODULE_STUFF,
    modperl_config_dir_create, /* dir config creater */
    modperl_config_dir_merge,  /* dir merger --- default is to override */
    modperl_config_srv_create, /* server config */
    modperl_config_srv_merge,  /* merge server config */
    modperl_cmds,              /* table of config file commands       */
    modperl_register_hooks,    /* register hooks */
};
```

Apache uses this structure to hook mod_perl in, and it specifies six custom callbacks which Apache will call at various stages that will be explained later.

`STANDARD20_MODULE_STUFF` is a standard macro defined in *httpd-2.0/include/http_config.h*. Currently its main use is for attaching Apache version magic numbers, so the previously compiled module won't be attempted to be used with newer Apache versions, whose API may have changed.

`modperl_cmds` is a struct, that defines the mod_perl configuration directives and the callbacks to be invoked for each of these.

2.3 Configuration Tree Building

At the `ap_read_config` stage the configuration file is parsed and stored in a parsed configuration tree is created. Some sections are stored unmodified in the parsed configuration tree to be processed after the `pre_config` hooks were run. Other sections are processed right away (e.g., the `Include` directive includes extra configuration and has to include it as soon as it was seen) and they may or may not add a subtree to the configuration tree.

`ap_build_config` feeds the configuration file lines from to `ap_build_config_sub`, which tokenizes the input, and uses the first token as a potential directive (command). It then calls `ap_find_command_in_modules()` to find a module that has registered that command (remember mod_perl has registered the directives in the `modperl_cmds command_rec` array, which was passed to `ap_add_module` inside the `perl_module` struct?). If that command is found and it has the `EXEC_ON_READ` flag set in its *req_override* field, the callback for that command is invoked. Depending on the command, it may perform some action and return (e.g., `User foo`), or it may continue reading from the configuration file and recursively execute other nested commands till it's done (e.g., `<Location ...>`). If the command is found but the `EXEC_ON_READ` flag is not set or the command is not found, the current node gets added to the configuration tree and will be processed during the `ap_process_config_tree()` stage, after the `pre_config` stage will be over.

If the command needs to be executed at this stage as it was just explained, `execute_now()` invokes the corresponding callback with `invoke_cmd`.

Since `LoadModule` directive has the `EXEC_ON_READ` flag set, that directive is executed as soon as it's seen and the modules its supposed to load get loaded right away.

For mod_perl loaded as a DSO object, this is when mod_perl starts its game.

2.3.1 Enabling the mod_perl Module and Installing its Callbacks

mod_perl can be loaded as a DSO object at startup time, or be prelinked at compile time.

For statically linked mod_perl, Apache enables mod_perl by calling `ap_add_module()`, which happens during the `ap_setup_prelinked_modules()` stage. The latter is happening before the configuration file is parsed.

When mod_perl is loaded as DSO:

```
<IfModule !mod_perl.c>
    LoadModule perl_module "modules/mod_perl.so"
</IfModule>
```

mod_dso's `load_module` first loads the shared mod_perl object, and then immediately calls `ap_add_loaded_module()` which calls `ap_add_module()` to enable mod_perl.

`ap_add_module()` adds the `perl_module` structure to the top of chained module list and calls `ap_register_hooks()` which calls the `modperl_register_hooks()` callback. This is the very first mod_perl hook that's called by Apache.

`modperl_register_hooks()` registers all the hooks that it wants to be called by Apache when the appropriate time comes. That includes configuration hooks, filter, connection and http protocol hooks. From now on most of the relationship between httpd and mod_perl is done via these hooks. Remember that in addition to these hooks, there are four hooks that were registered with `ap_add_module()`, and there are: `modperl_config_srv_create`, `modperl_config_srv_merge`, `modperl_config_dir_create` and `modperl_config_dir_merge`.

Finally after the hooks were registered, `ap_single_module_configure()` (called from mod_dso's `load_module` in case of DSO) runs the configuration process for the module. First it calls the `modperl_config_srv_create` callback for the main server, followed by the `modperl_config_dir_create` callback to create a directory structure for the main server. Notice that it passes `NULL` for the directory path, since we at the very top level.

If you need to do something as early as possible at mod_perl's startup, the `modperl_register_hooks()` is the right place to do that. For example we add a `MODPERL2` define to the `ap_server_config_defines` here:

```
*(char **)apr_array_push(ap_server_config_defines) =
    apr_pstrdup(p, "MODPERL2");
```

so the following code will work under mod_perl 2.0 enabled Apache without explicitly passing `-DMODPERL2` at the server startup:

```
<IfDefine MODPERL2>
    # 2.0 configuration
    PerlSwitches -wT
</IfDefine>
```

This section, of course, will see the define only if inserted after the `LoadModule perl_module ...`, because that's when `modperl_register_hooks` is called.

One inconvenience with using that hook, is that the server object is not among its arguments, so if you need to access that object, the next earliest function is `modperl_config_srv_create()`. However remember that it'll be called once for the main server and one more time for each virtual host, that has something to do with mod_perl. So if you need to invoke it only for the main server, you can use a `s->is_virtual` conditional. For example we need to enable the debug tracing as early as possible, but we need the server object in order to do that, so we perform this setting in `modperl_config_srv_create()`:

```
if (!s->is_virtual) {
    modperl_trace_level_set(s, NULL);
}
```

2.4 The `pre_config` Phase

After Apache processes its command line arguments, creates various pools and reads the configuration file in, it runs the registered *pre_config* hooks by calling `ap_run_pre_config()`. That's when `modperl_hook_pre_config` is called. And it does nothing.

2.4.1 Configuration Tree Processing

`ap_process_config_tree` calls `ap_walk_config`, which scans through all directives in the parsed configuration tree, and executes each one by calling `ap_walk_config_sub`. This is a recursive process with many twists.

Similar to `ap_build_config_sub` for each command (directive) in the configuration tree, it calls `ap_find_command_in_modules` to find a module that registered that command. If the command is not found the server dies. Otherwise the callback for that command is invoked with `invoke_cmd`, after fetching the current directory configuration:

```
invoke_cmd(cmd, parms, dir_config, current->args);
```

The `invoke_cmd` command is the one that invokes mod_perl's directives callbacks, which reside in *modperl_cmd.c*. `invoke_cmd` knows how the arguments should be passed to the callbacks, based on the information in the `modperl_cmds` array that we have just mentioned.

Notice that before `invoke_cmd` is invoked, `ap_set_config_vectors()` is called which sets the current server and section configuration objects for the module in which the directive has been found. If these objects were't created yet, it calls the registered callbacks as `create_dir_config` and `create_server_config`, which are `modperl_config_dir_create` and `modperl_config_srv_create` for the mod_perl module. (If you write your custom module in Perl, these correspond to the `DIR_CREATE` and `SERVER_CREATE` Perl subroutines.)

The command callback won't be invoked if it has the `EXEC_ON_READ` flag set, because it was already invoked earlier when the configuration tree was parsed. `ap_set_config_vectors()` is called in any case, because it wasn't called during the `ap_build_config`.

So we have `modperl_config_srv_create` and `modperl_config_dir_create` both called once for the main server (at the end of processing the `LoadModule perl_module ...` directive), and one more time for each virtual host in which at least one mod_perl directive is encountered. In addition `modperl_config_dir_create` is called for every section and subsection that includes mod_perl directives (META: or inherits from such a section even though specifies no mod_perl directives in it?).

2.4.2 Virtual Hosts Fixup

After the configuration tree is processed, `ap_fixup_virtual_hosts()` is called. One of the responsibilities of this function is to merge the virtual hosts configuration objects with the base server's object. If there are virtual hosts, `merge_server_configs()` calls `modperl_config_srv_merge()` and `modperl_config_dir_merge()` for each virtual host, to perform this merge for mod_perl configuration objects.

META: is that's the place where everything restarts? it doesn't restart under debugger since we run with NODETACH I believe.

2.4.3 The open_logs Phase

After Apache processes the configuration it's time for the *open_logs* phase, executed by `ap_run_open_logs()`. mod_perl has registered the `modperl_hook_init()` hook to be called for this phase.

META: complete what happens at this stage in mod_perl

META: why is it called modperl_hook_init and not open_logs? is it because it can be called from other functions?

2.4.4 The post_config Phase

Immediately after *open_logs*, the *post_config* phase follows. Here `ap_run_post_config()` calls `modperl_hook_post_config()`

2.5 Request Processing

META: need to write

2.6 Shutdown

META: need to write

2.7 Maintainers

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

- Stas Bekman [http://stason.org/]

2.8 Authors

-

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

3 mod_perl internals: mod_perl-specific functionality flow

3.1 Description

This document attempts to help understand the code flow for certain features. This should help to debug problems and add new features.

This document auguments mod_perl internals: Apache 2.0 Integration and discusses the internals of the mod_perl-specific features.

Make sure to read also: Debugging mod_perl C Internals.

META: these notes are a bit out of sync with the latest svn, but will be updated once the innovation dust settles down.

3.2 Perl Interpreters

How and when Perl interpreters are created:

1. modperl_hook_init is invoked by one of two paths: Either normally, during the open_logs phase, or during the configuration parsing if a directive needs perl at the early stage (e.g. PerlLoadModule).

```
ap_hook_open_logs()            -> # normal mod_perl startup
load_module() -> modperl_run() -> # early startup caused by PerlLoadModule
```

2. modperl_hook_init() -> modperl_init():

```
o modperl_startup()
  - parent perl is created and started ("-e0"),
  - top level PerlRequire and PerlModule are run

o modperl_interp_init()
  - modperl_tipool_new() # create/init tipool
  - modperl_interp_new() # no new perls are created at this stage

o modperl_init_vhost() # vhosts are booted, for each vhost run:
  if +Parent
    - modperl_startup()  # vhost gets its own parent perl (not perl_clone()!)
  else
    - vhost's PerlModule/PerlRequire directives are run if any
  if +(Parent|Clone)
    - modperl_interp_init() (new tipool, no new perls created)
```

3. Next the post_config hook is run. It immediately returns for non-threaded mpms. Otherwise that's where all the first clones are created (and later their are created on demand when there aren't enough in the pool and more are needed).

```
o modperl_init_clones() creates pools of clones
  - modperl_tipool_init() (clones the PerlStartInterp number of perls)
    - interp_pool_grow()
      - modperl_interp_new()
        ~ this time perl_clone() is called
        ~ PL_ptr_table is scratched
          modperl_xs_dl_handles_clear
```

3.3 Filters

Apache filters work in the following way. First of all, a filter must be registered by its name, in addition providing a pointer to a function that should be executed when the filter is called and the type of resources it should be called on (e.g., only request's body, the headers, both and others). Once registered, the filter can be inserted into a chain of filters to be executed at run time.

For example in the pre_connection phase we can add connection phase filters, and using the ap_hook_insert_filter we can call functions that add the current request's filters. The filters are added using their registered name and a special context variable, which is typed to (void *) so modules can store anything they want there. You can add more than one filter with the same name to the same filter chain.

Here is how mod_perl uses this infrastructure:

There can be many filters inserted via mod_perl, but they all seen by Apache by four filter names:

```
MODPERL_REQUEST_OUTPUT
MODPERL_REQUEST_INPUT
MODPERL_CONNECTION_OUTPUT
MODPERL_CONNECTION_INPUT
```

XXX: which actually seems to be lowercased by Apache (saw it in gdb), (it handles these in the case insensitive manner?). how does then modperl_filter_add_request works, as it compares *fname with M.

These four filter names are registered in modperl_register_hooks():

```
ap_register_output_filter(MP_FILTER_REQUEST_OUTPUT_NAME,
                          MP_FILTER_HANDLER(modperl_output_filter_handler),
                          AP_FTYPE_RESOURCE);

ap_register_input_filter(MP_FILTER_REQUEST_INPUT_NAME,
                         MP_FILTER_HANDLER(modperl_input_filter_handler),
                         AP_FTYPE_RESOURCE);

ap_register_output_filter(MP_FILTER_CONNECTION_OUTPUT_NAME,
                          MP_FILTER_HANDLER(modperl_output_filter_handler),
                          AP_FTYPE_CONNECTION);

ap_register_input_filter(MP_FILTER_CONNECTION_INPUT_NAME,
                         MP_FILTER_HANDLER(modperl_input_filter_handler),
                         AP_FTYPE_CONNECTION);
```

At run time input filter handlers are always called by modperl_input_filter_handler() and output filter handler by modperl_output_filter_handler(). For example if there are three MODPERL_CONNECTION_INPUT filters in the filters chain, modperl_input_filter_handler() will be called three times.

The real Perl filter handler (callback) is stored in ctx->handler, which is retrieved by modperl_{output|input}_filter_handler and run as a normal Perl handler by modperl_run_filter() via modperl_callback():

```
                      retrieve ctx->handler
modperl_output_filter_handler -> modperl_run_filter -> modperl_callback
```

This trick allows to have more than one filter handler in the filters chain using the same Apache filter name (the real filter's name is stored in ctx->handler->name.

Now the only missing piece in the puzzle is how and when mod_perl filter handlers are inserted into the filter chain. It happens in three stages.

1. When the configuration file is parsed, every time a PerlInputFilterHandler or a PerlOutputFilterHandler directive is encountered, its argument (filter handler) is inserted into dcfg->handlers_per_dir[idx] by modperl_cmd_input_filter_handlers() and modperl_cmd_output_filter_handlers(). idx is either MP_INPUT_FILTER_HANDLER or MP_OUTPUT_FILTER_HANDLER. Since they are stored in the dcfg struct, normal merging of parent and child directories applies.

2. Next, modperl_hook_post_config calls modperl_mgv_hash_handlers which works through dcfg->handlers_per_dir[idx] and resolves the handlers (via modperl_mgv_resolve), so they are resolved by the time filter handlers are added to the chain in the next step (e.g. the attributes are set if any).

3. Now all is left is to add the filters to the appropriate chains at the appropriate time.

   modperl_register_hooks() adds a pre_connection hook modperl_hook_pre_connection() which inserts connection filters via:

   ```
   modperl_input_filter_add_connection();
   modperl_output_filter_add_connection();
   ```

   modperl_hook_pre_connection() is called during the pre_connection phase.

   modperl_register_hooks() directly registers the request filters via ap_hook_insert_filter():

   ```
   modperl_output_filter_add_request
   modperl_input_filter_add_request
   ```

   functions registered with ap_hook_insert_filter(), will be called when the request record is created and they are supposed to insert request filters if any.

   All four functions perform a similar thing: loop through dcfg->handlers_per_dir[idx], where idx is per filter type: MP_{INPUT|OUTPUT}_FILTER_HANDLER, pick the filters of the appropriate type and insert them to filter chain using one of the two Apache functions that add filters. Since we have connection and request filters there are four different combinations:

```
ap_add_input_filter( name, (void*)ctx, NULL, c);
ap_add_output_filter(name, (void*)ctx, NULL, c);
ap_add_input_filter( name, (void*)ctx, r,    r->connection);
ap_add_output_filter(name, (void*)ctx, r,    r->connection);
```

Here the name is one of:

```
MODPERL_REQUEST_OUTPUT
MODPERL_REQUEST_INPUT
MODPERL_CONNECTION_OUTPUT
MODPERL_CONNECTION_INPUT
```

ctx, storing three things:

```
SV *data;
modperl_handler_t *handler;
PerlInterpreter *perl;
```

we have mentioned ctx->handler already, that's where the real Perl filter handler is stored. ctx->perl stores the current perl interpreter (used only in the threaded environment).

the last two arguments are the request and connection records.

notice that dcfg->handlers_per_dir[idx] stores connection and request filters in the same array, so we have only two arrays, one for input and one for output filters. We know to distinguish between connection and request filters by looking at ctx->handler->attrs record, which is derived from the handler subroutine's attributes. Remember that we can have:

```
sub Foo : FilterRequestHandler {}
```

and:

```
sub Bar : FilterConnectionHandler {}
```

For example we can figure out what kind of handler is that via:

```
if (ctx->handler->attrs & MP_FILTER_CONNECTION_HANDLER)) {
    /* Connection handler */
}
else if (ctx->handler->attrs & MP_FILTER_REQUEST_HANDLER)) {
    /* Request handler */
}
else {
    /* Unknown */
}
```

3.4 Maintainers

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

Stas Bekman [http://stason.org/]

3.5 Authors

- Stas Bekman [http://stason.org/]

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

4 MPMs - Multi-Processing Model Modules

4.1 Description

Discover what are the available MPMs and how they work with mod_perl.

META: This doc is under construction. Owners are wanted. -- pgollucci volunteering

4.2 MPMs Overview

4.3 The Worker MPM

META: incomplete

You can test whether running under threaded env via: ?

```
#ifdef USE_ITHREADS
/* whatever */
#endif
```

When the server is running under the threaded mpm `scfg->threaded_mpm` is set to true.

Caveats:

All per-server data is shared between threads, regardless of locking, changing the value of something like ap_document_root changes it for all threads. Not just the current process/request, the way it was in 1.3. So we can't really support modification of things like ap_document_root at request time, unless the mpm is prefork. we could support modification of modperl per-server data by using r->request_config in the same way push_handlers et al is implemented. But it is not possible to use this approach for anything outside of modperl (ap_document_root for example).

4.4 The Prefork MPM

META: incomplete

4.5 The Event MPM

4.6 Maintainers

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

- Philip M. Gollucci <pgollucci (at) p6m7g8.com>

4.7 Authors

- Stas Bekman [http://stason.org/] Philip M. Gollucci <pgollucci (at) p6m7g8.com>

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

5 mod_perl Coding Style Guide

5.1 Description

This document explains the coding style used in the core mod_perl development and which should be followed by all core developers.

5.2 Coding Style Guide

We try hard to code mod_perl using an identical style. Because everyone in the team should be able to read and understand the code as quickly and easily as possible. Some will have to adjust their habits for the benefit of all.

- **C code**

  mod_perl's C code follows the Apache style guide: http://dev.apache.org/styleguide.html

- **XS code**

  C code inside XS modules also follows the Apache style guide.

- **Perl code**

  mod_perl's Perl code also follows the Apache style guide, in terms of indentation, braces, etc. Style issues not covered by Apache style of guide should be looked up in the *perlstyle* manpage.

Here are the rough guidelines with more stress on the Perl coding style.

- **Indentation and Tabs**

  Do use 4 characters indentation.

  Do NOT use tabs.

  Here is how to setup your editor to do the right thing:

  - **x?emacs: cperl-mode**

    ```
    .xemacs/custom.el:
    ------------------
    (custom-set-variables
       '(cperl-indent-level 4)
       '(cperl-continued-statement-offset 4)
       '(cperl-tab-always-indent t)
       '(indent-tabs-mode nil)
    )
    ```

  - **vim**

```
.vimrc:
-------
set expandtab " replaces any tab keypress with the appropriate number of spaces
set tabstop=4 " sets tabs to 4 spaces
```

- **Block Braces**

  Do use a style similar to K&R style, not the same. The following example is the best guide:

  Do:

```
sub foo {
    my ($self, $cond, $baz, $taz) = @_;

    if ($cond) {
        bar();
    }
    else {
        $self->foo("one", 2, "...");
    }

    return $self;
}
```

  Don’t:

```
sub foo
{
    my ($self,$bar,$baz,$taz)=@_;
    if( $cond )
    {
        &bar();
    } else { $self->foo ("one",2,"..."); }
    return $self;
}
```

- **Lists and Arrays**

  Whenever you create a list or an array, always add a comma after the last item. The reason for doing this is that it’s highly probable that new items will be appended to the end of the list in the future. If the comma is missing and this isn’t noticed, there will be an error.

  Do:

```
my @list = (
    "item1",
    "item2",
    "item3",
);
```

  Don’t:

```
my @list = (
    "item1",
    "item2",
    "item3"
);
```

- **Last Statement in the Block**

  The same goes for `;` in the last statement of the block. Almost always add it even if it's not required, so when you add a new statement you don't have to remember to add `;` on a previous line.

  Do:

```
sub foo {
    statement1;
    statement2;
    statement3;
}
```

  Don't:

```
sub foo {
    statement1;
    statement2;
    statement3
}
```

5.3 Function and Variable Prefixes Convention

- **modperl_**

  The prefix for mod_perl C API functions.

- **MP_**

  The prefix for mod_perl C macros.

- **mpxs_**

  The prefix for mod_perl XS utility functions.

- **mp_xs_**

  The prefix for mod_perl *generated* XS utility functions.

- **MPXS_**

  The prefix for mod_perl XSUBs with an XS() prototype.

5.4 Coding Guidelines

The following are the Perl coding guidelines:

5.4.1 Global Variables

- **avoid globals in general**
- **avoid $&, $', $`**

  See `Devel::SawAmpersand`'s *README* that explains the evilness. Under mod_perl everybody suffers when one is seen anywhere since the interpreter is never shutdown.

5.4.2 Modules

- **Exporting/Importing**

  Avoid too much exporting/importing (glob aliases eat up memory)

  When you do wish to import from a module try to use an explicit list or tag whenever possible, e.g.:

  ```
  use POSIX qw(strftime);
  ```

  When you do not wish to import from a module, always use an empty list to avoid any import, e.g.:

  ```
  use IO::File ();
  ```

  (explain how to use Apache2::Status to find imported/exported functions)

5.4.3 Methods

- **indirect vs direct method calls**

  Avoid indirect method calls, e.g.

  Do:

  ```
  CGI::Cookie->new
  ```

  Don't:

  ```
  new CGI::Cookie
  ```

5.4.4 Inheritance

- **Avoid inheriting from certain modules**

Exporter. To avoid inheriting **AutoLoader::AUTOLOAD**

Do:

```
*import = \&Exporter::import;
```

Don't:

```
@MyClass::ISA = qw(Exporter);
```

5.4.5 Symbol tables

- **%main::**

  stay away from `main::` to avoid namespace clashes

5.4.6 Use of $_ in loops

Avoid using `$_` in loops unless it's a short loop and you don't call any subs from within the loop. If the loop started as short and then started to grow make sure to remove the use of `$_`:

Do:

```
for my $idx (1..100) {
    ....more than few lines...
    foo($idx);
    ....
}
```

Don't:

```
for (1..100) {
    ....more than a few statements...
    foo();
    ....
}
```

Because foo() might change `$_` if foo()'s author didn't localize `$_`.

This is OK:

```
for (1..100) {
    .... a few statements with no subs called
    # do something with $_
    ....
}
```

5.5 Maintainers

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

Stas Bekman [http://stason.org/]

5.6 Authors

- Doug MacEachern<dougm (at) covalent.net>
- Stas Bekman [http://stason.org/]

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

6 Porting Apache:: XS Modules from mod_perl 1.0 to 2.0

6 Porting Apache:: XS Modules from mod_perl 1.0 to 2.0

6.1 Description

This document talks mainly about porting modules using XS code. It's also helpful to those who start developing mod_perl 2.0 packages.

Also make sure to first read about porting Apache:: Perl modules.

6.2 Porting Makefile.PL

It's only an issue if it was using `Apache::src`. A new configuration system is in works. So watch this space for updates on this issue.

ModPerl::MM is the new replacement of Apache::src.

6.3 Porting XS Code

If your module's XS code relies on the Apache and mod_perl C APIs, it's very likely that you will have to adjust the XS code to the Apache 2.0 and mod_perl 2.0 C API.

The C API has changed a lot, so chances are that you are much better off not to mix the two APIs in the same XS file. However if you do want to mix the two you will have to use something like the following:

```
#include ap_mmn.h
/* ... */
#if AP_MODULE_MAGIC_AT_LEAST(20020903,4)
    /* 2.0 code */
#else
    /* 1.0 code */
#endif
```

The `20020903,4` is the value of the magic version number matching Apache 2.0.47, the earliest Apache version supported by mod_perl 2.0.

6.4 Thread Safety

META: to be written

```
#ifdef MP_THREADED
    /* threads specific code goes here */
#endif
```

For now see: http://httpd.apache.org/docs-2.0/developer/thread_safety.html

6.5 PerlIO

PerlIO layer has become usable only in perl 5.8.0, so if you plan on working with PerlIO, you can use the `PERLIO_LAYERS` constant. e.g.:

```
#ifdef PERLIO_LAYERS
#include "perliol.h"
#else
#include "iperlsys.h"
#endif
```

6.6 'make test' Suite

The `Apache::Test` testing framework that comes together with mod_perl 2.0 works with 1.0 and 2.0 mod_perl versions. Therefore you should consider porting your test suite to use the Apache::Test Framework.

6.7 Apache C Code Specific Notes

Most of the documentation covering migration to Apache 2.0 can be found at: http://httpd.apache.org/docs-2.0/developer/

The Apache 2.0 API documentation now resides in the C header files, which can be conveniently browsed via http://docx.webperf.org/.

The APR API documentation can be found here http://apr.apache.org/.

The new Apache and APR APIs include many new functions. Though certain functions have been preserved, either as is or with a changed prototype (for example to work with pools), others have been renamed. So if you are porting your code and the function that you've used doesn't seem to exist in Apache 2.0, first refer to the "compat" header files, such as: *include/ap_compat.h*, *srclib/apr/include/apr_compat.h*, and *srclib/apr-util/include/apu_compat.h*, which list functions whose names have changed but which are otherwise the same. If this fails, proceed to look in other headers files in the following directories:

- *ap_* functions in *include/*
- *apr_* functions in *srclib/apr/include/* and *srclib/apr-util/include/*

6.7.1 ap_soft_timeout(), ap_reset_timeout(), ap_hard_timeout() and ap_kill_timeout()

If the C part of the module in 1.0 includes `ap_soft_timeout()`, `ap_reset_timeout()`, `ap_hard_timeout()` and `ap_kill_timeout()` functions simply remove these in 2.0. There is no replacement for these functions because Apache 2.0 uses non-blocking I/O. As a side-effect of this change, Apache 2.0 no longer uses `SIGALRM`, which has caused conflicts in mod_perl 1.0.

6.8 Maintainers

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

Stas Bekman [http://stason.org/]

6.9 Authors

- Stas Bekman [http://stason.org/]
- Doug MacEachern <dougm (at) covalent.net>

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

7 Measure sizeof() of Perl’s C Structures

7.1 Description

This document describes the *sizeof* various structures, as determined by *util/sizeof.pl*. These measurements are mainly for research purposes into making Perl things smaller, or rather, how to use less Perl things.

7.2 Perl Structures

Structures diagrams are courtesy gdb (print pretty) and a bit of hand crafting.

- **CV - 229 minimum, 254 minimum w/ symbol table entry**

```
cv = {
   sv_any = {        // XPVCV *
       xpv_pv = 0x0, // char *
       xpv_cur = 0,  // STRLEN
       xpv_len = 0,  // STRLEN
       xof_off = 0 , // IV
       xnv_nv = 0,   // NV
       xmg_magic = 0x0, // MAGIC *
       xmg_stash = 0x0, // HV *
       xcv_stash = 0x0, // HV *
       xcv_start = 0x0, // OP *
       xcv_root = 0x0,  // OP *
       xcv_xsub = 0x0,  // void (*)(register PerlInterpreter *, CV *)
       xcv_xsubany = {  // ANY
           any_ptr = 0x0,
           any_i32 = 0,
           any_iv = 0,
           any_long = 0,
           any_dptr = 0,
           any_dxptr = 0
       },
       xcv_gv = { // GV *
           sv_any = { // void *
               xpv_pv = 0x0, // char *
               xpv_cur = 0,  // STRLEN
               xpv_len = 0,  // STRLEN
               xiv_iv = 0,   // IV
               xnv_nv = 0,   // NV
               xmg_magic =  { // MAGIC *
                   mg_moremagic = 0x0, // MAGIC *
                   mg_virtual = 0x0,   // MGVTBL *
                   mg_private = 0,     // U16
                   mg_type = 0,        // char
                   mg_flags = 0,       // U8
                   mg_obj = 0x0,       // SV *
                   mg_ptr = 0x0,       // char *
                   mg_len = 0,         // I32
               },
               xmg_stash = 0x0, // HV *
               xgv_gp = { // GP *
```

```
                    gp_sv = { // SV *
                        sv_any = 0x0,  // void *
                        sv_refcnt = 0, // U32
                        sv_flags = 0   // U32
                    },
                    gp_refcnt = 0, // U32
                    gp_io = 0x0,   // struct io *
                    gp_form = 0x0, // CV *
                    gp_av = 0x0,   // AV *
                    gp_hv = 0x0,   // HV *
                    gp_egv = 0x0,  // GV *
                    gp_cv = 0x0,   // CV *
                    gp_cvgen = 0,  // U32
                    gp_flags = 0,  // U32
                    gp_line = 0,   // line_t
                    gp_file = 0x0, // char *
                },
                xgv_name = 0x0,  // char *
                xgv_namelen = 0, // STRLEN
                xgv_stash = 0x0, // void *
                xgv_flags = 0,   // U8
            },
            sv_refcnt = 0, // U32
            sv_flags = 0,  // U32
        },
        xcv_file = 0x0, // char *
        xcv_depth = 0, // long
        xcv_padlist = 0x0, // AV *
        xcv_outside = 0x0, // CV *
        xcv_flags = 0, // cv_flags_t
    }
    sv_refcnt = 0, // U32
    sv_flags = 0,  // U32
};
```

In addition to the minimum bytes:

- **name of the subroutine: GvNAMELEN(CvGV(cv))+1**
- **symbol table entry: HvENTRY (25 + GvNAMELEN(CvGV(cv))+1)**
- **minimum sizeof(AV) * 3: xcv_padlist if !CvXSUB(cv)**
- **CvROOT(cv) optree**

- **HV - 60 minmum**

```
hv = {
    sv_any = { // SV *
        xhv_array = 0x0,  // char *
        xhv_fill = 0,     // STRLEN
        xhv_max = 0,      // STRLEN
        xhv_keys = 0,     // IV
        xnv_nv = 0,       // NV
        xmg_magic = 0x0,  // MAGIC *
        xmg_stash = 0x0,  // HV *
        xhv_riter = 0,    // I32
        xhv_eiter = 0x0,  // HE *
        xhv_pmroot = 0x0, // PMOP *
```

```
        xhv_name = 0x0    // char *
    },
    sv_refcnt = 0, // U32
    sv_flags = 0,  // U32
};
```

Each entry adds `sizeof(HvENTRY)`, minimum of 7 (initial `xhv_max`). Note that keys of the same value share `sizeof(HEK)`, across all hashes.

- **HvENTRY - 25 + HeKLEN+1**

```
sizeof(HE *) + sizeof(HE) + sizeof(HEK)
```

- **HE - 12**

```
he = {
   hent_next = 0x0, // HE *
   hent_hek = 0x0,  // HEK *
   hent_val = 0x0   // SV *
};
```

- **HEK - 9 + hek_len**

```
hek = {
   hek_hash = 0, // U32
   hek_len = 0,  // I32
   hek_key = 0,  // char
};
```

- **AV - 53**

```
av = {
   sv_any =  { // SV *
       xav_array = 0x0,  // char *
       xav_fill = 0,     // size_t
       xav_max = 0,      // size_t
       xof_off = 0,      // IV
       xnv_nv = 0,       // NV
       xmg_magic = 0x0,  // MAGIC *
       xmg_stash = 0x0,  // HV *
       xav_alloc = 0x0,  // SV **
       xav_arylen = 0x0, // SV *
       xav_flags = 0,    // U8
   },
   sv_refcnt = 0, // U32
   sv_flags = 0   // U32
};
```

In addition to the minimum bytes:

  - **AvFILL(av) * sizeof(SV *)**

7.3 SEE ALSO

perlguts(3), B::Size(3),

http://gisle.aas.no/perl/illguts/

7.4 Maintainers

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

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

7.5 Authors

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

8 Which Coding Technique is Faster

8.1 Description

This document tries to show more efficient coding styles by benchmarking various styles.

WARNING: This doc is under construction

META: for now these are just unprocessed snippets from the mailing list. Please help me to make these into useful essays.

8.2 backticks vs XS

META: unprocessed yet.

compare the difference of calling an xsub that does _nothing_ vs. a backticked program that does _nothing_.

```
/* file:test.c */
int main(int argc, char **argv, char **env)
{
    return 1;
}

/* file:TickTest.xs */
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = TickTest             PACKAGE = TickTest

void
foo()

CODE:

# file:test.pl
use blib;
use TickTest ();

use Benchmark;

timethese(100_000, {
    backtick => sub { ‘./test‘ },
    xs => sub { TickTest::foo() },
});
```

Results:

```
Benchmark: timing 100000 iterations of backtick, xs...
  backtick: 292 wallclock secs (18.68 usr 43.93 sys + 142.43 cusr 84.00 csys = 289.04 CPU) @ 1597.19/s (n=100000)
        xs: -1 wallclock secs ( 0.25 usr +  0.00 sys =  0.25 CPU) @ 400000.00/s (n=100000)
            (warning: too few iterations for a reliable count)
```

8.3 sv_catpvn vs. fprintf

META: unprocessed yet.

and what i'm trying to say is that if both the xs code and external program are doing the same thing, xs will be heaps faster than backticking a program. your xsub and external program are not doing the same thing.

i'm guessing part of the difference in your code is due to fprintf having a pre-allocated buffer, whereas the SV's SvPVX has not been pre-allocated and gets realloc-ed each time you call sv_catpv. have a look at the code below, fprintf is faster than sv_catpvn, but if the SvPVX is preallocated, sv_catpvn becomes faster than fprintf:

```
timethese(1_000, {
    fprintf   => sub { TickTest::fprintf() },
    svcat     => sub { TickTest::svcat() },
    svcat_pre => sub { TickTest::svcat_pre() },
});

Benchmark: timing 1000 iterations of fprintf, svcat, svcat_pre...
   fprintf:  9 wallclock secs ( 8.72 usr +  0.00 sys =  8.72 CPU) @ 114.68/s (n=1000)
     svcat: 13 wallclock secs (12.82 usr +  0.00 sys = 12.82 CPU) @ 78.00/s (n=1000)
 svcat_pre:  2 wallclock secs ( 2.75 usr +  0.00 sys =  2.75 CPU) @ 363.64/s (n=1000)

#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

static FILE *devnull;

MODULE = TickTest             PACKAGE = TickTest

BOOT:
devnull = fopen("/dev/null", "w");

void
fprintf()

    CODE:
    {
        int i;
        char buffer[8292];

        for (i=0; i<sizeof(buffer); i++) {
            fprintf(devnull, "a");
        }
    }

void
svcat()

    CODE:
    {
```

```
        int i;
        char buffer[8292];
        SV *sv = newSV(0);

        for (i=0; i<sizeof(buffer); i++) {
            sv_catpvn(sv, "a", 1);
        }

        SvREFCNT_dec(sv);
    }

void
svcat_pre()

    CODE:
    {
        int i;
        char buffer[8292];
        SV *sv = newSV(sizeof(buffer)+1);

        for (i=0; i<sizeof(buffer); i++) {
            sv_catpvn(sv, "a", 1);
        }

        SvREFCNT_dec(sv);
    }
```

8.4 Maintainers

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

Stas Bekman [http://stason.org/]

8.5 Authors

- Stas Bekman [http://stason.org/]
- Doug MacEachern <dougm (at) covalent.net>

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

9 Porting Apache:: XS Modules from mod_perl 1.0 to 2.0

9.1 Description

This document talks mainly about porting modules using XS code. It's also helpful to those who start developing mod_perl 2.0 packages.

Also make sure to first read about porting Apache:: Perl modules.

9.2 Porting Makefile.PL

It's only an issue if it was using `Apache::src`. A new configuration system is in works. So watch this space for updates on this issue.

ModPerl::MM is the new replacement of Apache::src.

9.3 Porting XS Code

If your module's XS code relies on the Apache and mod_perl C APIs, it's very likely that you will have to adjust the XS code to the Apache 2.0 and mod_perl 2.0 C API.

The C API has changed a lot, so chances are that you are much better off not to mix the two APIs in the same XS file. However if you do want to mix the two you will have to use something like the following:

```
#include ap_mmn.h
/* ... */
#if AP_MODULE_MAGIC_AT_LEAST(20020903,4)
    /* 2.0 code */
#else
    /* 1.0 code */
#endif
```

The `20020903,4` is the value of the magic version number matching Apache 2.0.47, the earliest Apache version supported by mod_perl 2.0.

9.4 Thread Safety

META: to be written

```
#ifdef MP_THREADED
    /* threads specific code goes here */
#endif
```

For now see: http://httpd.apache.org/docs-2.0/developer/thread_safety.html

9.5 PerlIO

PerlIO layer has become usable only in perl 5.8.0, so if you plan on working with PerlIO, you can use the `PERLIO_LAYERS` constant. e.g.:

```
#ifdef PERLIO_LAYERS
#include "perliol.h"
#else
#include "iperlsys.h"
#endif
```

9.6 'make test' Suite

The `Apache::Test` testing framework that comes together with mod_perl 2.0 works with 1.0 and 2.0 mod_perl versions. Therefore you should consider porting your test suite to use the Apache::Test Framework.

9.7 Apache C Code Specific Notes

Most of the documentation covering migration to Apache 2.0 can be found at: http://httpd.apache.org/docs-2.0/developer/

The Apache 2.0 API documentation now resides in the C header files, which can be conveniently browsed via http://docx.webperf.org/.

The APR API documentation can be found here http://apr.apache.org/.

The new Apache and APR APIs include many new functions. Though certain functions have been preserved, either as is or with a changed prototype (for example to work with pools), others have been renamed. So if you are porting your code and the function that you've used doesn't seem to exist in Apache 2.0, first refer to the "compat" header files, such as: *include/ap_compat.h*, *srclib/apr/include/apr_compat.h*, and *srclib/apr-util/include/apu_compat.h*, which list functions whose names have changed but which are otherwise the same. If this fails, proceed to look in other headers files in the following directories:

- *ap_* functions in *include/*
- *apr_* functions in *srclib/apr/include/* and *srclib/apr-util/include/*

9.7.1 ap_soft_timeout(), ap_reset_timeout(), ap_hard_timeout() and ap_kill_timeout()

If the C part of the module in 1.0 includes `ap_soft_timeout()`, `ap_reset_timeout()`, `ap_hard_timeout()` and `ap_kill_timeout()` functions simply remove these in 2.0. There is no replacement for these functions because Apache 2.0 uses non-blocking I/O. As a side-effect of this change, Apache 2.0 no longer uses `SIGALRM`, which has caused conflicts in mod_perl 1.0.

9.8 Maintainers

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

Stas Bekman [http://stason.org/]

9.9 Authors

- Stas Bekman [http://stason.org/]
- Doug MacEachern <dougm (at) covalent.net>

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

10 Debugging mod_perl Perl Internals

10.1 Description

This document explains how to debug Perl code under mod_perl.

Most of the mod_perl 1.0 debug documentation applies to mod_perl 2.0:

10.1.1 Detecting Hanging Processes

See Hanging Processes: Detection and Diagnostics for the explanation, but under mp2 to use signals to detect where the process is spinning, you can't use `$SIG{USR2}`, you have to use POSIX signals. i.e. the code becomes:

```
use Carp ();
use POSIX qw(SIGUSR2);
my $mask      = POSIX::SigSet->new( SIGUSR2 );
my $action    = POSIX::SigAction->new(\&tell_where_spinning, $mask);
my $oldaction = POSIX::SigAction->new();
POSIX::sigaction(SIGUSR2, $action, $oldaction );

sub tell_where_spinning {
    Carp::confess("caught SIGUSR2!");
};
```

and then:

```
% kill USR2 <pid_of_the_spinning_process>
```

and watch for the trace in *error_log*.

10.2 Maintainers

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

- Stas Bekman [http://stason.org/]

10.3 Authors

- Stas Bekman [http://stason.org/]

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

11 Debugging mod_perl C Internals

11.1 Description

This document explains how to debug C code under mod_perl, including mod_perl core itself.

For certain debugging purposes you may find useful to read first the following notes on mod_perl internals: Apache 2.0 Integration and mod_perl-specific functionality flow.

11.2 Debug notes

META: needs more organization (if you grok any of the following, patches are welcome)

META: there is a new compile-time option in perl-5.8.8+: -DDEBUG_LEAKING_SCALARS, which prints out the addresses of leaked SVs and new_SV() can be used to discover where those SVs were allocated. (see perlhack.pod for more info)

META: httpd has quite a lot of useful debug info: http://httpd.apache.org/dev/debugging.html (need to add this link to mp1 docs as well)

META: profiling: need a new entry of profiling. + running mod_perl under gprof: Defining GPROF when compiling uses the moncontrol() function to disable gprof profiling in the parent, and enable it only for request processing in children (or in one_process mode).

META: Jeff Trawick wrote a few useful debug modules, for httpd-2.1: mod_backtrace (similar to bt in gdb, but doesn't require the core file) and mod_whatkilledus (gives the info about the request that caused the segfault). http://httpd.apache.org/~trawick/exception_hook.html

11.2.1 Entering Single Server Mode

Most of the time, when debugging Apache or mod_perl, one needs to start Apache in a single server mode and not allow it to detach itself from the initial process. This is accomplished with:

```
% httpd -DONE_PROCESS -DNO_DETACH
```

11.2.2 Setting gdb Breakpoints with mod_perl Built as DSO

If mod_perl is built as a DSO module, you cannot set the breakpoint in the mod_perl source files when the *httpd* program gets loaded into the debugger. The reason is simple: At this moment *httpd* has no idea about mod_perl module yet. After the configuration file is processed and the mod_perl DSO module is loaded then the breakpoints in the source of mod_perl itself can be set.

The trick is to break at *apr_dso_load*, let it load *libmodperl.so*, then you can set breakpoints anywhere in the modperl code:

```
% gdb httpd
(gdb) b apr_dso_load
(gdb) run -DONE_PROCESS
[New Thread 1024 (LWP 1600)]
```

```
[Switching to Thread 1024 (LWP 1600)]

Breakpoint 1, apr_dso_load (res_handle=0xbfffb48c, path=0x811adcc
  "/home/stas/apache.org/modperl-perlmodule/src/modules/perl/libmodperl.so",
  pool=0x80e1a3c) at dso.c:138
141         void *os_handle = dlopen(path, RTLD_NOW | RTLD_GLOBAL);
(gdb) finish
...
Value returned is $1 = 0
(gdb) b modperl_hook_init
(gdb) continue
```

This example shows how to set a breakpoint at *modperl_hook_init*.

To automate things you can put those in the *.gdb-jump-to-init* file:

```
b apr_dso_load
run -DONE_PROCESS -d `pwd`/t -f `pwd`/t/conf/httpd.conf
finish
b modperl_hook_init
continue
```

and then start the debugger with:

```
% gdb /home/stas/httpd-2.0/bin/httpd -command \
`pwd`/t/.gdb-jump-to-init
```

11.2.3 Starting the Server Fast under gdb

When the server is started under gdb, it first loads the symbol tables of the dynamic libraries that it sees going to be used. Some versions of gdb may take ages to complete this task, which makes the debugging very irritating if you have to restart the server all the time and it doesn't happen immediately.

The trick is to set the `auto-solib-add` flag to 0:

```
set auto-solib-add 0
```

as early as possible in *~/.gdbinit* file.

With this setting in effect, you can load only the needed dynamic libraries with *sharedlibrary* gdb command. Remember that in order to set a breakpoint and step through the code inside a certain dynamic library you have to load it first. For example consider this gdb commands file:

```
.gdb-commands
------------
file ~/httpd/prefork/bin/httpd
handle SIGPIPE pass
handle SIGPIPE nostop
set auto-solib-add 0
b ap_run_pre_config
run -d `pwd`/t -f `pwd`/t/conf/httpd.conf \
-DONE_PROCESS -DAPACHE2 -DPERL_USEITHREADS
sharedlibrary mod_perl
```

```
b modperl_hook_init
# start: modperl_hook_init
continue
# restart: ap_run_pre_config
continue
# restart: modperl_hook_init
continue
b apr_poll
continue

# load APR/PerlIO/PerlIO.so
sharedlibrary PerlIO
b PerlIOAPR_open
```

which can be used as:

```
% gdb -command=.gdb-commands
```

This script stops in *modperl_hook_init()*, so you can step through the mod_perl startup. We had to use the *ap_run_pre_config* so we can load the *libmodperl.so* library as explained earlier. Since httpd restarts on the start, we have to *continue* until we hit *modperl_hook_init* second time, where we can set the breakpoint at *apr_poll*, the very point where httpd polls for new request and run again *continue* so it'll stop at *apr_poll*. This particular script passes over modperl_hook_init(), since we run the `continue` command a few times to reach the *apr_poll* breakpoint. See the Precooked gdb Startup Scripts section for standalone script examples.

When gdb stops at the function *apr_poll* it's a time to start the client, that will issue a request that will exercise the server execution path we want to debug. For example to debug the implementation of `APR::Pool` we may run:

```
% t/TEST -run apr/pool
```

which will trigger the run of a handler in *t/response/TestAPR/pool.pm* which in turn tests the `APR::Pool` code.

But before that if we want to debug the server response we need to set breakpoints in the libraries we want to debug. For example if we want to debug the function `PerlIOAPR_open` which resides in *APR/PerlIO/PerlIO.so* we first load it and then we can set a breakpoint in it. Notice that gdb may not be able to load a library if it wasn't referenced by any of the code. In this case we have to load this library at the server startup. In our example we load:

```
PerlModule APR::PerlIO
```

in *httpd.conf*. To check which libraries' symbol tables can be loaded in gdb, run (when the server has been started):

```
gdb> info sharedlibrary
```

which also shows which libraries are loaded already.

Also notice that you don't have to type the full path of the library when trying to load them, even a partial name will suffice. In our commands file example we have used `sharedlibrary mod_perl` instead of saying `sharedlibrary mod_perl.so`.

If you want to set breakpoints and step through the code in the Perl and APR core libraries you should load their appropriate libraries:

```
gdb> sharedlibrary libperl
gdb> sharedlibrary libapr
gdb> sharedlibrary libaprutil
```

Setting *auto-solib-add* to 0 makes the debugging process unusual, since originally gdb was loading the dynamic libraries automatically, whereas now it doesn't. This is the price one has to pay to get the debugger starting the program very fast. Hopefully the future versions of gdb will improve.

Just remember that if you try to *step-in* and debugger doesn't do anything, that means that the library the function is located in wasn't loaded. The solution is to create a commands file as explained in the beginning of this section and craft the startup script the way you need to avoid extra typing and mistakes when repeating the same debugging process again and again.

Under threaded mpms (e.g. worker), it's possible that you won't be able to debug unless you tell gdb to load the symbols from the threads library. So for example if on your OS that library is called *libpthread.so* make sure to run:

```
sharedlibrary libpthread
```

somewhere after the program has started. See the Precooked gdb Startup Scripts section for examples.

Another important thing is that whenever you want to be able to see the source code for the code you are stepping through, the library or the executable you are in must have the debug symbols present. That means that the code has to be compiled with *-g* option for the gcc compiler. For example if I want to set a breakpoint in /lib/libc.so, I can do that by loading:

```
gdb> sharedlibrary /lib/libc.so
```

But most likely that this library has the debug symbols stripped off, so while gdb will be able to break at the breakpoint set inside this library, you won't be able to step through the code. In order to do so, recompile the library to add the debug symbols.

If debug code in response handler you usually start the client after the server was started, when doing this a lot you may find it annoying to need to wait before the client can be started. Therefore you can use a few tricks to do it in one command. If the server starts fast you can use sleep():

```
% ddd -command=.debug-modperl-init & ; \
sleep 2 ; t/TEST -verbose -run apr/pool
```

or the `Apache::Test` framework's `-ping=block` option:

```
% ddd -command=.debug-modperl-init & ; \
t/TEST -verbose -run -ping=block apr/pool
```

which will block till the server starts responding, and only then will try to run the test.

11.2.4 Precooked gdb Startup Scripts

Here are a few startup scripts you can use with gdb to accomplish one of the common debugging tasks. To execute the startup script, simply run:

```
% gdb -command=.debug-script-filename
```

They can be run under gdb and any of the gdb front-ends. For example to run the scripts under `ddd` substitute `gdb` with `ddd`:

```
% ddd -command=.debug-script-filename
```

- **Debugging mod_perl Initialization**

  The *code/.debug-modperl-init*:

  ```
  # This gdb startup script breaks at the modperl_hook_init() function,
  # which is useful for debug things at the modperl init phase.
  #
  # Invoke as:
  # gdb -command=.debug-modperl-init
  #
  # see ADJUST notes for things that may need to be adjusted

  # ADJUST: the path to the httpd executable if needed
  file ~/httpd/worker/bin/httpd
  handle SIGPIPE nostop
  handle SIGPIPE pass
  set auto-solib-add 0

  define myrun
      tbreak main
      break ap_run_pre_config
      #  ADJUST: the httpd.conf file's path if needed
      #  ADJUST: add -DPERL_USEITHREADS to debug threaded mpms
      run -d `pwd`/t -f `pwd`/t/conf/httpd.conf -DONE_PROCESS -DAPACHE2
      continue
  end

  define modperl_init
      sharedlibrary mod_perl
      b modperl_hook_init
      continue
  end

  define sharedap
      # ADJUST: uncomment next line to debug threaded mpms
      #sharedlibrary libpthread
      sharedlibrary apr
  ```

```
    sharedlibrary aprutil
    #sharedlibrary mod_ssl.so
    continue
end

define sharedperl
    sharedlibrary libperl
end

# start the server and run till modperl_hook_init on start
myrun
modperl_init

# ADJUST: uncomment to reach modperl_hook_init on restart
#continue
#continue

# ADJUST: uncomment if you need to step through the code in apr libs
#sharedap

# ADJUST: uncomment if you need to step through the code in perlib
#sharedperl
```

startup script breaks at the `modperl_hook_init()` function, which is useful for debugging code at the modperl's initialization phase.

- **Debugging mod_perl's Hooks Registeration With httpd**

  Similar to the previous startup script, the *code/.debug-modperl-register*:

```
# This gdb startup script allows to break at the very first invocation
# of mod_perl initialization, just after it was loaded. When the
# perl_module is loaded, and its pointer struct is added via
# ap_add_module(), the first hook that will be called is
# modperl_register_hooks().
#
# Invoke as:
# gdb -command=.debug-modperl-register
#
# see ADJUST notes for things that may need to be adjusted

define sharedap
    sharedlibrary apr
    sharedlibrary aprutil
    #sharedlibrary mod_ssl.so
end

define sharedperl
    sharedlibrary libperl
end

### Run ###

# ADJUST: the path to the httpd executable if needed
file ~/httpd/prefork/bin/httpd
handle SIGPIPE nostop
```

```
handle SIGPIPE pass
set auto-solib-add 0

tbreak main

# assuming that mod_dso is compiled in
b load_module


# ADJUST: the httpd.conf file's path if needed
# ADJUST: add -DPERL_USEITHREADS to debug threaded mpms
run -d `pwd`/t -f `pwd`/t/conf/httpd.conf \
-DONE_PROCESS -DNO_DETACH -DAPACHE2

# skip over 'tbreak main'
continue

# In order to set the breakpoint in mod_perl.so, we need to get to
# the point where it's loaded.
#
# With static mod_perl, the bp can be set right away
#

# With DSO mod_perl, mod_dso's load_module() loads the mod_perl.so
# object and it immediately calls ap_add_module(), which calls
# modperl_register_hooks(). So if we want to bp at the latter, we need
# to stop at load_module(), set the 'bp modperl_register_hooks' and
# then continue.

# Assuming that 'LoadModule perl_module' is the first LoadModule
# directive in httpd.conf, you need just one 'continue' after
# 'ap_add_module'. If it's not the first one, you need to add as many
# 'continue' commands as the number of 'LoadModule foo' before
# perl_module, but before setting the 'ap_add_module' bp.
#
# If mod_perl is compiled statically, everything is already preloaded,
# so you can set modperl_* the breakpoints right away

b ap_add_module
continue

sharedlibrary mod_perl
b modperl_register_hooks
continue

#b modperl_hook_init
#b modperl_config_srv_create
#b modperl_startup
#b modperl_init_vhost
#b modperl_dir_config
#b modperl_cmd_load_module
#modperl_config_apply_PerlModule

# ADJUST: uncomment next line to debug threaded mpms
#sharedlibrary libpthread

# ADJUST: uncomment if you need to step through the code in apr libs
```

```
#sharedap

# ADJUST: uncomment if you need to step through the code in perlib
#sharedperl
```

startup script breaks at the `modperl_register_hooks()`, which is the very first hook called in the mod_perl land. Therefore use this one if you need to start debugging at an even earlier entry point into mod_perl.

Refer to the notes inside the script to adjust it for a specific *httpd.conf* file.

- **Debugging mod_perl XS Extensions**

  The *code/.debug-modperl-xs*:

```
# This gdb startup script breaks at the mpxs_Apache__Filter_print()
# function from the XS code, as an example how you can debug the code
# in XS extensions.
#
# Invoke as:
# gdb -command=.debug-modperl-xs
# and then run:
# t/TEST -v -run -ping=block filter/api
#
# see ADJUST notes for things that may need to be adjusted

# ADJUST: the path to the httpd executable if needed
file /home/stas/httpd/worker/bin/httpd
handle SIGPIPE nostop
handle SIGPIPE pass
set auto-solib-add 0

define myrun
    tbreak main
    break ap_run_pre_config
    # ADJUST: the httpd.conf file's path if needed
    # ADJUST: add -DPERL_USEITHREADS to debug threaded mpms
    run -d `pwd`/t -f `pwd`/t/conf/httpd.conf \
    -DONE_PROCESS -DNO_DETACH -DAPACHE2
    continue
end

define sharedap
    # ADJUST: uncomment next line to debug threaded mpms
    #sharedlibrary libpthread
    sharedlibrary apr
    sharedlibrary aprutil
    #sharedlibrary mod_ssl.so
    continue
end

define sharedperl
```

```
    sharedlibrary libperl
end

define gopoll
    b apr_poll
    continue
    continue
end

define mybp
    # load Apache/Filter.so
    sharedlibrary Filter
    b mpxs_Apache__Filter_print
    # no longer needed and they just make debugging harder under threads
    disable 2
    disable 3
    continue
end

myrun
gopoll
mybp

# ADJUST: uncomment if you need to step through the code in apr libs
#sharedap

# ADJUST: uncomment if you need to step through the code in perlib
#sharedperl
```

startup script breaks at the `mpxs_Apache2__Filter_print()` function implemented in *xs/Apache2/Filter/Apache2__Filter.h*. This is an example of debugging code in XS Extensions. For this particular example the complete test case is:

```
% ddd -command=.debug-modperl-xs & \
t/TEST -v -run -ping=block filter/api
```

When *filter/api* test is running it calls mpxs_Apache2__Filter_print() which is when the breakpoint is reached.

- **Debugging code in shared objects created by `Inline.pm`**

  This is not strictly related to mod_perl, but sometimes when trying to reproduce a problem (e.g. for a p5p bug-report) outside mod_perl, the code has to be written in C. And in certain cases, Inline can be just the right tool to do it quickly. However if you want to interactively debug the library that it creates, it might get tricky. So similar to the previous sections, here is a gdb *code/.debug-inline*:

```
# save this file as .debug and execute this as:
# gdb -command=.debug
# or if you prefer gui
# ddd -command=.debug
#
# NOTE: Adjust the path to the perl executable
# also this perl should be built with debug enabled
file /usr/bin/perl
```

```
# If you need to debug with gdb a live script and not a library, you
# are going to have a hard time to set any breakpoint in the C code.
# the workaround is force Inline to compile and load .so, by putting
# all the code in the BEGIN {} block and call Inline->init from there.
#
# you also need to prevent from Inline deleting autogenerated .xs so
# you can step through the C source code, and of course you need to
# add '-g' so .so won't be stripped of debug info
#
# here is a sample perl script that can be used with this gdb script
#
# test.pl
# #-----#
# use strict;
# use warnings;
#
# BEGIN {
#     use Inline Config =>
#         #FORCE_BUILD => 1,
#         CLEAN_AFTER_BUILD => 0;
#
#     use Inline C => Config =>
#         OPTIMIZE => '-g';
#
#     use Inline C => <init;
#
# }
#
# my_bp();

tb main
# NOTE: adjust the name of the script that you run
run test.pl

# when Perl_runops_debug breakpoint is hit Inline will already load
# the autogenerated .so, so we can set the bp in it (that's only if
# you have run 'Inline->init' inside the BEGIN {} block

b S_run_body
continue
b Perl_runops_debug
continue

# here you set your breakpoints
b my_bp
continue
```

startup script that will save you a lot of time. All the details and a sample perl script are inside the gdb script.

11.3 Analyzing Dumped Core Files

When your application dies with the *Segmentation fault* error (which is generated by the `SIGSEGV` signal) and optionally generates a *core* file you can use `gdb` or a similar debugger to find out what caused the *Segmentation fault* (or a *segfault* as we often call it).

11.3.1 Getting Ready to Debug

In order to debug the *core* file you may need to recompile Perl and mod_perl with debugging symbols. Usually you have to recompile only mod_perl, but if the *core* dump happens in the *libperl.so* library and you want to see the whole backtrace, you need to recompile Perl as well. It may also occur inside httpd or 3rd party module, in which case you will need to recompile those. The following notes should help to accomplish the right thing:

- **mod_perl**

  rebuild mod_perl with `MP_DEBUG=1`.

  ```
  % perl Makefile.PL MP_DEBUG=1 ...
  % make && make test && make install
  ```

  Building mod_perl with `PERL_DEBUG=1` will:

  1. add `-g` to `EXTRA_CFLAGS`
  2. turn on `MP_TRACE` (tracing)
  3. Set `PERL_DESTRUCT_LEVEL=2`
  4. Link against *libperld.so* if *$Config{archlibexp}/CORE/libperld$Config{lib_ext}* exists.

- **httpd**

  If the segfault happens inside *ap_* or *apr_* calls, rebuild httpd with `--enable-maintainer-mode`:

  ```
  % CFLAGS="-g" ./configure --enable-maintainer-mode ...
  % make && make install
  ```

- **perl**

  If the segfault happens inside *Perl_* calls, rebuild perl with `-Doptimize='-g'`:

  ```
  % ./Configure -Doptimize='-g' ...
  % make && make test && make install
  ```

  Remember to recompile mod_perl if you've recompiled perl.

- **3rd party perl modules**

  if the trace happens in one of the 3rd party perl modules, make sure to rebuild them, now that you've perl re-built with debugging flags. They will automatically pick the right compile flags from perl.

Now the software is ready for a proper debug.

11.3.2 Causing a SegFault

Most likely you already have the segfault situation, but sometimes you want to create one. For example sometimes you need to make sure that your system is configured to dump core files.

For that purpose you can use `Debug::DumpCore` available from CPAN:
http://search.cpan.org/dist/Debug-FaultAutoBT/

```
% perl -MDebug::DumpCore -eDebug::DumpCore::segv
Segmentation fault (core dumped)
```

Notice that you could use Perl's `CORE::dump` to achieve the same goal:

```
% perl -le 'dump'
Abort (core dumped)
```

but the generated in that case backtrace is not very useful for learning purposes. If all you want to test is whether your system is configured to dump core files then Perl's `CORE::dump` will do just fine.

11.3.3 Getting the core File Dumped

Now let's get the *core* file dumped from within the mod_perl server. Sometimes the program aborts abnormally via the SIGSEGV signal (*Segmentation Fault*), but no *core* file is dumped. And without the *core* file it's hard to find the cause of the problem, unless you run the program inside `gdb` or another debugger in first place. In order to get the *core* file, the application has to:

1. have the effective UID the same as real UID (the same goes for GID). Which is the case of mod_perl unless you modify these settings in the program.

2. be running from a directory which at the moment of the *Segmentation fault* is writable by the process that received this signal. Notice that the program might change its current directory during its run, so it's possible that the *core* file will need to be dumped in a different directory from the one the program was originally started from.

   Under Apache `ServerRoot` is used as the default directory. Since that directory is sually not writable by the user running Apache, it's possible to use the directive `CoreDumpDirectory` (available since Apache 2.0.45) to tell Apache to dump the core file elsewhere.

3. be started from a shell process with sufficient resource allocations for the *core* file to be dumped. You can override the default setting from within a shell script if the process is not started manually. In addition you can use `BSD::Resource` to manipulate the setting from within the code as well.

You can use `ulimit` for `bash` and `limit` for `csh` to check and adjust the resource allocation. For example inside `bash`, you may set the core file size to unlimited:

```
panic% ulimit -c unlimited
```

or for `csh`:

```
panic% limit coredumpsize unlimited
```

For example you can set an upper limit on the *core* file size to 8MB with:

```
panic% ulimit -c 8388608
```

So if the core file is bigger than 8MB it will be not created.

4. Of course you have to make sure that you have enough disk space to create a big core file (mod_perl *core* files tend to be of a few MB in size).

Note that when you are running the program under a debugger like `gdb`, which traps the `SIGSEGV` signal, the *core* file will not be dumped. Instead it allows you to examine the program stack and other things without having the *core* file.

So let's write a simple script that uses `Debug::DumpCore`:

```
core_dump.pl
------------
use strict;
use warnings FATAL => 'all';

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Debug::DumpCore ();
use Cwd;

my $r = shift;
$r->content_type('text/plain');

my $dir = getcwd();
$r->print("The core should be found at $dir/core.$$\n");
$r->rflush;

Debug::DumpCore::segv();
```

In this script we load the `Apache2::RequestRec`, `Apache2::RequestIO`, `Debug::DumpCore` and `Cwd` modules, then we acquire the Apache request object and set the HTTP response header. Now we come to the real part -- we get the current working directory, print out the location of the *core* file that we are about to dump and finally we call `Debug::DumpCore::segv()` which dumps the *core* file.

Before we run the script we make sure that the shell sets the *core* file size to be unlimited, start the server in single server mode as a non-root user and generate a request to the script:

```
panic% cd /home/httpd/bin
panic% limit coredumpsize unlimited
panic% ./httpd -DONE_PROCESS -DNO_DETACH
    # issue a request here
Segmentation fault (core dumped)
```

Our browser prints out:

```
The core should be found at /home/httpd/bin/core.12345
```

And indeed the core file appears where we were told it will be:

```
panic% ls -l /home/httpd/bin/core.12345
-rw-------  1 stas stas 13758464 Nov 23 18:33 /home/httpd/bin/core.12345
```

As you can see it's about 14MB *core* file. Notice that mod_perl was started as user *stas*, which had write permission for directory */home/httpd/bin*.

Notice that on certain platforms you get no PID digits appended to the core file name, so sometimes, it'll be just *core*.

11.3.4 Analyzing the core File

First we start gdb:

```
panic% gdb /home/httpd/bin/httpd /home/httpd/bin/core.12345
```

with the location of the mod_perl executable and the core file as the arguments.

To see the backtrace you run the *where* or the *bt* command:

```
(gdb) bt
#0  0x407ab26c in crash_now_for_real (
    suicide_message=0x407ad300 "Cannot stand this life anymore")
    at DumpCore.xs:10
#1  0x407ab293 in crash_now (
    suicide_message=0x407ad300 "Cannot stand this life anymore",
    attempt_num=42) at DumpCore.xs:17
#2  0x407ab39b in XS_Debug__DumpCore_segv (my_perl=0x86a9298, cv=0x8d36750)
    at DumpCore.xs:26
#3  0x40540649 in Perl_pp_entersub () from .../libperl.so
...
#7  0x404530cc in modperl_callback () from .../mod_perl.so
```

Well, you can see the last commands, but our perl and mod_perl are probably without the debug symbols. This is not the kind of trace you should send as a part of your bug report, because a lot of important information that should aid resolve the reported problem is missing.

Therefore the next step is to recompile Perl and mod_perl (and may be Apache) with debug symbols as explained earlier in this chapter.

Now when we repeat the process of starting the server, issuing a request and getting the core file, after which we run `gdb` again against the executable and the dumped *core.6789* file.

```
panic% gdb /home/httpd/bin/httpd /home/httpd/bin/core.6789
```

Now we can see the whole backtrace:

```
(gdb) bt
#0  0x407ab26c in crash_now_for_real (
    suicide_message=0x407ad300 "Cannot stand this life anymore")
    at DumpCore.xs:10
#1  0x407ab293 in crash_now (
    suicide_message=0x407ad300 "Cannot stand this life anymore",
    attempt_num=42) at DumpCore.xs:17
#2  0x407ab39b in XS_Debug__DumpCore_segv (my_perl=0x86a9298, cv=0x8d36750)
    at DumpCore.xs:26
#3  0x40540649 in Perl_pp_entersub (my_perl=0x86a9298) at pp_hot.c:2890
#4  0x4051ca4d in Perl_runops_debug (my_perl=0x86a9298) at dump.c:1449
#5  0x404c1ea3 in S_call_body (my_perl=0x86a9298, myop=0xbfffed90, is_eval=0)
    at perl.c:2298
#6  0x404c19cf in Perl_call_sv (my_perl=0x86a9298, sv=0x8cd0914, flags=4)
    at perl.c:2216
#7  0x404530cc in modperl_callback (my_perl=0x86a9298, handler=0x81ba6d8,
    p=0x8d16828, r=0x8d16860, s=0x813d238, args=0x8d018d8)
    at modperl_callback.c:102
#8  0x404539ce in modperl_callback_run_handlers (idx=6, type=4, r=0x8d16860,
    c=0x0, s=0x813d238, pconf=0x0, plog=0x0, ptemp=0x0,
    run_mode=MP_HOOK_RUN_FIRST) at modperl_callback.c:263
#9  0x40453c2d in modperl_callback_per_dir (idx=6, r=0x8d16860,
    run_mode=MP_HOOK_RUN_FIRST) at modperl_callback.c:351
#10 0x4044c728 in modperl_response_handler_run (r=0x8d16860, finish=0)
    at mod_perl.c:911
#11 0x4044cadb in modperl_response_handler_cgi (r=0x8d16860) at mod_perl.c:1006
#12 0x080db2bc in ap_run_handler (r=0x8d16860) at config.c:151
#13 0x080dba19 in ap_invoke_handler (r=0x8d16860) at config.c:363
#14 0x080a9953 in ap_process_request (r=0x8d16860) at http_request.c:246
#15 0x080a3ef8 in ap_process_http_connection (c=0x8d10920) at http_core.c:250
#16 0x080e7efc in ap_run_process_connection (c=0x8d10920) at connection.c:42
#17 0x080e82f8 in ap_process_connection (c=0x8d10920, csd=0x8d10848)
    at connection.c:175
#18 0x080d9b6d in child_main (child_num_arg=0) at prefork.c:609
#19 0x080d9c44 in make_child (s=0x813d238, slot=0) at prefork.c:649
#20 0x080d9d6a in startup_children (number_to_start=2) at prefork.c:721
#21 0x080da177 in ap_mpm_run (_pconf=0x81360a8, plog=0x817e1c8, s=0x813d238)
    at prefork.c:940
#22 0x080e0de8 in main (argc=11, argv=0xbffff284) at main.c:619
```

That’s the perfect back trace to send as a part of the bug report.

Reading the trace from bottom to top, we can see that it starts with Apache calls, followed by mod_perl calls which end up in `modperl_callback()` which calls the Perl program via `Perl_call_sv`.

Notice that in our example we knew what script has caused the Segmentation fault. In a real world the chances are that you will find the *core* file without any clue to which of handler or script has triggered it. The special *curinfo* `gdb` macro comes to help:

For perl enabled with threads that's:

```
define curinfo
   printf "%d:%s\n", my_perl->Tcurcop->cop_line, \
       my_perl->Tcurcop->cop_file
end
```

For a non-threaded version that's:

```
define curinfo
   printf "%d:%s\n", PL_curcop->cop_line, \
   ((XPV*)(*(XPVGV*)PL_curcop->cop_filegv->sv_any)\
   ->xgv_gp->gp_sv->sv_any)->xpv_pv
end
```

Simply past the correct version at the gdb prompt (in this example the perl is threaded):

```
(gdb) define curinfo
Type commands for definition of "curinfo".
End with a line saying just "end".
>   printf "%d:%s\n", my_perl->Tcurcop->cop_line, \
       my_perl->Tcurcop->cop_file
>end
```

and now we can call it:

```
(gdb) curinfo
No symbol "my_perl" in current context.
```

Oops, the function where the segfault has happened doesn't have the perl context, so we need to look at the backtrace and find the first function which accepts the `my_perl` argument (this is because we use a threaded perl). In this example this is the second frame:

```
#2  0x407ab39b in XS_Debug__DumpCore_segv (my_perl=0x86a9298, cv=0x8d36750)
    at DumpCore.xs:26
```

therefore we need to go two frames up:

```
(gdb) up 2
#2  0x407ab39b in XS_Debug__DumpCore_segv (my_perl=0x86a9298, cv=0x8d36750)
    at DumpCore.xs:26
26      in DumpCore.xs
```

and now we call `curinfo` again:

```
gdb) curinfo
14:/home/httpd/cgi-bin/core_dump.pl
```

Et voilà, we can see that the segfault was triggered on line 14 of *core_dump.pl*, which has the line:

```
Debug::DumpCore::segv();
```

And we are done.

These are the bits of information that are important to extract and include in your bug report in order for us to be able to reproduce and resolve a problem. In this example it was the full backtrace, the filename and line where the faulty function was called (the faulty function is `Debug::DumpCore::segv()`) and the actual line where the Segmentation fault occured (`crash_now_for_real` at `DumpCore.xs:10`).

11.3.5 Analyzing the core File Automatically

If the core file(s) are found in the mod_perl source directory, when running *t/REPORT* the core file back-traces will be automatically extracted and added to the report if the perl module `Devel::GDB` is installed.

See the function `dump_core_file()` in *Apache-Test/lib/Apache/TestReport.pm* if you want to see how it is invoked or refer to the `Devel::GDB` manpage.

11.3.6 Obtaining core Files under Solaris

There are two ways to get core files under Solaris. The first is by configuring the system to allow core dumps, the second is by stopping the process when it receives the SIGSEGV signal and "manually" obtaining the core file.

11.3.6.1 Configuring Solaris to Allow core Dumps

By default, Solaris 8 won’t allow a setuid process to write a core file to the file system. Since apache starts as root and spawns children as ’nobody’, core dumps won’t produce core files unless you modify the system settings.

To see the current settings, run the coreadm command with no parameters and you’ll see:

```
% coreadm
     global core file pattern:
       init core file pattern: core
            global core dumps: disabled
       per-process core dumps: enabled
      global setid core dumps: disabled
per-process setid core dumps: disabled
     global core dump logging: disabled
```

These settings are stored in the */etc/coreadm.conf* file, but you should set them with the coreadm utility. As super-user, you can run coreadm with -g to set the pattern and path for core files (you can use a few variables here) and -e to enable some of the disabled items. After setting a new pattern, enabling global, global-setid, and log, and rebooting the system (reboot is required), the new settings look like:

```
% coreadm
     global core file pattern: /usr/local/apache/cores/core.%f.%p
       init core file pattern: core
            global core dumps: enabled
       per-process core dumps: enabled
      global setid core dumps: enabled
per-process setid core dumps: disabled
     global core dump logging: enabled
```

Now you'll start to see core files in the designated cores directory and they will look like *core.httpd.2222* where httpd is the name of the executable and the 2222 is the process id. The new core files will be read/write for root only to maintain some security, and you should probably do this on development systems only.

11.3.6.2 Manually Obtaining core Dumps

On Solaris the following method can be used to generate a core file.

1. Use truss(1) as *root* to stop a process on a segfault:

   ```
   panic% truss -f -l -t \!all -s \!SIGALRM -S SIGSEGV -p <pid>
   ```

   or, to monitor all httpd processes (from bash):

   ```
   panic% for pid in ‘ps -eaf -o pid,comm | fgrep httpd | cut -d’/’ -f1‘;
   do truss -f -l -t \!all -s \!SIGALRM -S SIGSEGV -p $pid 2>&1 &
   done
   ```

   The used truss(1) options are:

   - `-f` - follow forks.
   - `-l` - (that's an el) includes the thread-id and the pid (the pid is what we want).
   - `-t` - specifies the syscalls to trace,
   - !all - turns off the tracing of syscalls specified by `-t`
   - `-s` - specifies signals to trace and the `!SIGALRM` turns off the numerous alarms Apache creates.
   - `-S` - specifies signals that stop the process.
   - `-p` - is used to specify the pid.

   Instead of attaching to the process, you can start it under truss(1):

   ```
   panic% truss -f -l -t \!all -s \!SIGALRM -S SIGSEGV \
          /usr/local/bin/httpd -f httpd.conf 2>&1 &
   ```

2. Watch the *error_log* file for reaped processes, as when they get SISSEGV signals. When the process is reaped it's stopped but not killed.

3. Use gcore(1) to get a *core* of stopped process or attach to it with gdb(1). For example if the process id is 662:

   ```
   panic% gcore 662
   gcore: core.662 dumped
   ```

   Now you can load this *core* file in gdb(1).

4. `kill -9` the stopped process. Kill the truss(1) processes as well, if you don't need to trap other segfaults.

Obviously, this isn't great to be doing on a production system since truss(1) stops the process after it dumps core and prevents Apache from reaping it. So, you could hit the clients/threads limit if you segfault a lot.

11.4 Debugging Threaded MPMs

11.4.1 Useful Information from gdb Manual

Debugging programs with multiple threads: http://sources.redhat.com/gdb/current/online-docs/gdb_5.html#SEC25

Stopping and starting multi-thread programs: http://sources.redhat.com/gdb/current/online-docs/gdb_6.html#SEC40

11.4.2 libpthread

when using:

```
set auto-solib-add 0
```

make sure to:

```
sharedlibrary libpthread
```

(or whatever the shared library is used on your OS) without which you may have problems to debug the threaded mpm mod_perl.

11.5 Defining and Using Custom gdb Macros

GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files. See: http://sources.redhat.com/gdb/current/onlinedocs/gdb_21.html

Apache 2.0 source comes with a nice pack of macros and can be found in *httpd-2.0/.gdbinit*. To use it issue:

```
gdb> source /wherever/httpd-2.0/.gdbinit
```

Now if for example you want to dump the contents of the bucket brigade, you can do:

```
gdb> dump_brigade my_brigade
```

where `my_brigade` is the pointer to the bucket brigade that you want to debug.

mod_perl 1.0 has a similar file (*modperl/.gdbinit*) mainly including handy macros for dumping Perl datastructures, however it works only with non-threaded Perls. But otherwise it's useful in debugging mod_perl 2.0 as well.

11.6 Expanding C Macros

Perl, mod_perl and httpd C code makes an extensive use of C macros, which sometimes use many other macros in their definitions, so it becomes quite a task to figure out how to figure out what a certain macro expands to, especially when the macro expands to different values in differnt environments. Luckily there are ways to automate the expansion process.

11.6.1 Expanding C Macros with `make`

The mod_perl *Makefile*'s include a rule for macro expansions which you can find by looking for the `c.i.` rule. To expand all macros in a certain C file, you should run `make filename.i`, which will create *filename.i* with all macros expanded in it. For example to create *apr_perlio.i* with all macros used in *apr_perlio.c*:

```
% cd modperl-2.0/xs/APR/PerlIO
% make apr_perlio.i
```

the *apr_perlio.i* file now lists all the macros:

```
% less apr_perlio.i
# 1 "apr_perlio.c"
# 1 "<built-in>"
#define __VERSION__ "3.1.1 (Mandrake Linux 8.3 3.1.1-0.4mdk)"
...
```

11.6.2 Expanding C Macros with `gdb`

With gcc-3.1 or higher and gdb-5.2-dev or higher you can expand macros in gdb, when you step through the code. e.g.:

```
(gdb) macro expand pTHX_
expands to:  PerlInterpreter *my_perl __attribute__((unused)),
(gdb) macro expand PL_dirty
expands to: (*Perl_Tdirty_ptr(my_perl))
```

For each library that you want to use this feature with you have to compile it with:

```
CFLAGS="-gdwarf-2 -g3"
```

or whatever is appropriate for your system, refer to the gcc manpage for more info.

To compile perl with this debug feature, pass `-Doptimize='-gdwarf-2 -g3'` to `./Configure`. For Apache run:

```
CFLAGS="-gdwarf-2 -g3" ./configure [...]
```

for mod_perl you don't have to do anything, as it'll pick the `$Config{optimize}` Perl flags automatically, if Perl is compiled with `-DDEBUGGING` (which is implied on most systems, if you use `-Doptimize='-g'` or similar.)

Notice that this will make your libraries **huge**! e.g. on Linux 2.4 Perl 5.8.0's normal *libperl.so* is about 0.8MB on linux, compiled with `-Doptimize='-g'` about 2.7MB and with `-Doptimize='-gdwarf-2 -g3'` 12.5MB. `httpd` is also becomes about 10 times bigger with this feature enabled. *mod_perl.so* instead of 0.2k becomes 11MB. You get the idea. Of course since you may want this only during the development/debugging, that shouldn't be a problem.

The complete details are at: http://sources.redhat.com/gdb/current/onlinedocs/gdb_10.html#SEC69

11.7 Maintainers

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

Stas Bekman [http://stason.org/]

11.8 Authors

- Stas Bekman [http://stason.org/]

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

12 Getting Help with mod_perl 2.0 Core Development

12.1 Description

This document covers the resources available to the mod_perl 2.0 core developer. Please notice that you probably want to read the user's help documentation if you have problems using mod_perl 2.0.

The following mailing lists and resources can be of a major interest to the mod_perl 2.0 developers.

12.2 mod_perl

12.2.1 Submitting Patches

If you submit patches the *Porting/patching.pod* manpage can be very useful. You can find it *perl-5.7.0/Porting/patching.pod* or similar or read it online at http://sunsite.ualberta.ca/Documentation/Misc/perl-5.6.1/Porting/patching.html.

Note that we prefer the patches inlined into an email. This makes easier to comment on them. If your email client mangles the spacing and wraps lines, then send them as MIME attachments.

12.2.2 mod_perl 2.0 Core Development Discussion List

This list is used by the mod_perl 2.0 core developers to discuss design issues, solve problems, munch on patches and exchange ideas.

- mailing list subscription: mailto:dev-subscribe@perl.apache.org
- archive: http://marc.theaimsgroup.com/?l=apache-modperl-dev&r=1&w=2#apache-modperl-dev

When reporting problems, be sure to include the output of:

```
% perl build/config.pl
```

which generates the output from:

- **perl -V**
- **httpd -V**
- **Makefile.PL options**

Please use the output generated by *t/REPORT* utility.

If you get segmentation faults please send the stack backtrace to the modperl developers list.

12.2.3 mod_perl 2.0 Core Development SVN Commits List

This list's traffic is comprised of solely svn commits, so this is the place to be if you want to see mod_perl 2.0 evolve before your eyes.

- mailing list subscription: mailto:modperl-cvs-subscribe@perl.apache.org
- archive: http://marc.theaimsgroup.com/?l=apache-modperl-cvs&r=1&w=2#apache-modperl-cvs

12.2.4 Apache-Test

The `Apache-Test` project, originally developed as a part of mod_perl 2.0, is now a part of the Apache `httpd-test` project. You get this repository automatically when checking out the mod_perl-2.0 svn repository or you could check it out explicitly via:

```
% svn co http://svn.apache.org/repos/asf/perl/Apache-Test/trunk/ Apache-Test
```

The module is discussed on the mod_perl development mailing list. Commit changes go to the mod_perl svn commit mailing list. Both lists are covered in the preceding sections.

12.3 Apache

12.3.1 httpd 2.0

- **discussion/problems report:**

  mailing list subscription: mailto:dev-subscribe@httpd.apache.org

  archive: http://marc.theaimsgroup.com/?l=apache-new-httpd&r=1&w=

- **svn commits**

  mailing list subscription: mailto:httpd-2.0-cvs-subscribe@perl.apache.org

  archive: http://marc.theaimsgroup.com/?l=apache-cvs&r=1&w=2

- Apache source code cross-reference (LXR): http://lxr.webperf.org/
- Apache source code through Doxygen documentation system:

  http://docx.webperf.org/

-

12.3.2 Apache Portable Runtime (APR)

The Apache Portable Run-time libraries have been designed to provide a common interface to low level routines across any platform. mod_perl comes with a partial Perl APR API.

- **discussion/problems report:**

mailing list subscription: mailto:apr-dev-subscribe@perl.apache.org

archive: http://marc.theaimsgroup.com/?l=apr-dev&r=1&w=2

- **svn commits**

  mailing list subscription: mailto:apr-cvs-subscribe@perl.apache.org

  archive: http://marc.theaimsgroup.com/?l=apr-cvs&r=1&w=2

12.3.3 Perl 5

Currently mod_perl 2.0 requires perl 5.6.1 and higher.

If you think you have found a bug in perl 5 report it to the perl5-porters mailing list. Otherwise please choose the appropriate list from the extensive perl related lists: http://lists.perl.org/.

- **discussion/problems reports:**

  mailing list subscription: mailto:perl5-porters-subscribe@perl.org

  archive: http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/ and http://archive.develooper.com/perl5-porters@perl.org/

  news gateway: news://news.perl.com/perl.porters-gw/

- **Perl Dev Resources**

  http://dev.perl.org/

- **perforce**

  Perl uses `perforce` for its source revision control, see *Porting/repository.pod* manpage coming with Perl for more information.

  the perforce repository: http://public.activestate.com/gsar/APC/ or ftp://ftp.linux.activestate.com/pub/staff/gsar/APC/

  the Perl Repository Browser: http://public.activestate.com/cgi-bin/perlbrowse

  the Perl cross-reference: http://pxr.perl.org/source/

  mailing list subscription: perl5-changes-subscribe@perl.org

  archive: http://archive.develooper.com/perl5-changes@perl.org/

12.4 More Help

There is a parallel help document in the user documentation set which covers mod_perl user's issues.

12.5 Maintainers

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

Stas Bekman [http://stason.org/]

12.6 Authors

- Stas Bekman [http://stason.org/]
-

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

Table of Contents: