1   Apache2::SubProcess -- Executing SubProcesses under mod_perl

1.1 Synopsis

```
use Apache2::SubProcess ();

use Config;
use constant PERLIO_IS_ENABLED => $Config{useperlio};

# pass @ARGV / read from the process
$command = "/tmp/argv.pl";
@argv = qw(foo bar);
$out_fh = $r->spawn_proc_prog($command, \@argv);
$output = read_data($out_fh);

# pass environment / read from the process
$command = "/tmp/env.pl";
$r->subprocess_env->set(foo => "bar");
$out_fh = $r->spawn_proc_prog($command);
$output = read_data($out_fh);

# write to/read from the process
$command = "/tmp/in_out_err.pl";
($in_fh, $out_fh, $err_fh) = $r->spawn_proc_prog($command);
print $in_fh "hello\n";
$output = read_data($out_fh);
$error  = read_data($err_fh);

# helper function to work w/ and w/o perlio-enabled Perl
sub read_data {
    my ($fh) = @_;
    my $data;
    if (PERLIO_IS_ENABLED || IO::Select->new($fh)->can_read(10)) {
        $data = <$fh>;
    }
    return defined $data ? $data : '';
}

# pass @ARGV but don't ask for any communication channels
$command = "/tmp/argv.pl";
@argv = qw(foo bar);
$r->spawn_proc_prog($command, \@argv);
```

1.2 Description

Apache2::SubProcess provides the Perl API for running and communicating with processes spawned from mod_perl handlers.

At the moment it's possible to spawn only external program in a new process. It's possible to provide other interfaces, e.g. executing a sub-routine reference (via B::Deparse) and may be spawn a new program in a thread (since the APR api includes API for spawning threads, e.g. that's how it's running mod_cgi on win32).

1.3 API

1.3.1 *spawn_proc_prog*

Spawn a sub-process and return STD communication pipes:

```
                              $r->spawn_proc_prog($command);
                              $r->spawn_proc_prog($command, \@argv);
 $out_fh                    = $r->spawn_proc_prog($command);
 $out_fh                    = $r->spawn_proc_prog($command, \@argv);
 ($in_fh, $out_fh, $err_fh) = $r->spawn_proc_prog($command);
 ($in_fh, $out_fh, $err_fh) = $r->spawn_proc_prog($command, \@argv);
```

- **obj: $r (Apache2::RequestRec object)**
- **arg1: $command (string)**

  The command to be $exec()'ed.

- **opt arg2: \@argv (ARRAY ref)**

  A reference to an array of arguments to be passed to the process as the process' ARGV.

- **ret: ...**

  In VOID context returns no filehandles (all std streams to the spawned process are closed).

  In SCALAR context returns the output filehandle of the spawned process (the in and err std streams to the spawned process are closed).

  In LIST context returns the input, outpur and error filehandles of the spawned process.

- **since: 2.0.00**

It's possible to pass environment variables as well, by calling:

```
  $r->subprocess_env->set($key => $value);
```

before spawning the subprocess.

There is an issue with reading from the read filehandle ($in_fh)):

A pipe filehandle returned under perlio-disabled Perl needs to call select() if the other end is not fast enough to send the data, since the read is non-blocking.

A pipe filehandle returned under perlio-enabled Perl on the other hand does the select() internally, because it's really a filehandle opened via :APR layer, which internally uses APR to communicate with the pipe. The way APR is implemented Perl's select() cannot be used with it (mainly because select() wants fileno() and APR is a crossplatform implementation which hides the internal datastructure).

Therefore to write a portable code, you want to use select for perlio-disabled Perl and do nothing for perlio-enabled Perl, hence you can use something similar to the `read_data()` wrapper shown in the Synopsis section.

Several examples appear in the Synopsis section.

`spawn_proc_prog()` is similar to `fork()`, but provides you a better framework to communicate with that process and handles the cleanups for you. But that means that just like `fork()` it gives you a different process, so you don't use the current Perl interpreter in that new process. If you try to use that method or fork to run a high-performance parallel processing you should look elsewhere. You could try Perl threads, but they are **very** expensive to start if you have a lot of things loaded into memory (since `perl_clone()` dups almost everything in the perl land, but the opcode tree). In the mod_perl "paradigm" this is much more expensive than fork, since normally most of the time we have lots of perl things loaded into memory. Most likely the best solution here is to offload the job to PPerl or some other daemon, with the only added complexity of communication.

To spawn a completely independent process, which will be able to run after Apache has been shutdown and which won't prevent Apache from restarting (releasing the ports Apache is listening to) call spawn_proc_prog() in a void context and make the script detach and close/reopen its communication streams. For example, spawn a process as:

```
use Apache2::SubProcess ();
$r->spawn_proc_prog ('/path/to/detach_script.pl', $args);
```

and the */path/to/detach_script.pl* contents are:

```
# file:detach_script.pl
#!/usr/bin/perl -w
use strict;
use warnings;

use POSIX 'setsid';

chdir '/'                or die "Can't chdir to /: $!";
open STDIN, '/dev/null'  or die "Can't read /dev/null: $!";
open STDOUT, '+>>', '/path/to/apache/error_log'
    or die "Can't write to /dev/null: $!";
open STDERR, '>&STDOUT'  or die "Can't dup stdout: $!";
setsid or die "Can't start a new session: $!";

# run your code here or call exec to another program
```

reopening (or closing) the STD streams and called `setsid()` makes sure that the process is now fully detached from Apache and has a life of its own. `chdir()` ensures that no partition is tied, in case you need to remount it.

1.4 See Also

mod_perl 2.0 documentation.

1.5 Copyright

mod_perl 2.0 and its core modules are copyrighted under The Apache Software License, Version 2.0.

1.6 Authors

The mod_perl development team and numerous contributors.

Table of Contents: