NAME

"IO::Async::Process" – start and manage a child process

SYNOPSIS

```
use IO::Async::Process;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

my $process = IO::Async::Process->new(
   command => [ "tr", "a-z", "n-za-m" ],
   stdin => {
      from => "hello world\n",
   },
   stdout => {
      on_read => sub {
         my ( $stream, $buffref ) = @_;
         while( $$buffref =~ s/^(.*)\n// ) {
            print "Rot13 of 'hello world' is '$1'\n";
         }

         return 0;
      },
   },

   on_finish => sub {
      $loop->stop;
   },
);

$loop->add( $process );

$loop->run;
```

Also accessible via the ‘‘open_process’’ in IO::Async::Loop method:

```
$loop->open_process(
   command => [ "/bin/ping", "-c4", "some.host" ],

   stdout => {
      on_read => sub {
         my ( $stream, $buffref, $eof ) = @_;
         while( $$buffref =~ s/^(.*)\n// ) {
            print "PING wrote: $1\n";
         }
         return 0;
      },
   },

   on_finish => sub {
      my $process = shift;
      my ( $exitcode ) = @_;
      my $status = ( $exitcode >> 8 );
      ...
   },
);
```

DESCRIPTION

This subclass of IO::Async::Notifier starts a child process, and invokes a callback when it exits. The child process can either execute a given block of code (via `fork(2)`), or a command.

EVENTS

The following events are invoked, either using subclass methods or CODE references in parameters:

on_finish `$exitcode`

Invoked after the process has exited by normal means (i.e. an `exit(2)` syscall from a process, or `return`ing from the code block), and has closed all its file descriptors.

on_exception `$exception`**,** `$errno`**,** `$exitcode`

Invoked when the process exits by an exception from `code`, or by failing to `exec(2)` the given command. `$errno` will be a dualvar, containing both number and string values. After a successful `exec()` call, this condition can no longer happen.

Note that this has a different name and a different argument order from `Loop->open_process`'s `on_error`.

If this is not provided and the process exits with an exception, then `on_finish` is invoked instead, being passed just the exit code.

Since this is just the results of the underlying `$loop->spawn_child` `on_exit` handler in a different order it is possible that the `$exception` field will be an empty string. It will however always be defined. This can be used to distinguish the two cases:

```
on_exception => sub {
   my $self = shift;
   my ( $exception, $errno, $exitcode ) = @_;

   if( length $exception ) {
      print STDERR "The process died with the exception $exception " .
         "(errno was $errno)\n";
   }
   elsif( ( my $status = W_EXITSTATUS($exitcode) ) == 255 ) {
      print STDERR "The process failed to exec() - $errno\n";
   }
   else {
      print STDERR "The process exited with exit status $status\n";
   }
}
```

CONSTRUCTOR

new

```
$process = IO::Async::Process->new( %args )
```

Constructs a new `IO::Async::Process` object and returns it.

Once constructed, the `Process` will need to be added to the `Loop` before the child process is started.

PARAMETERS

The following named parameters may be passed to `new` or `configure`:

on_finish => CODE

on_exception => CODE

CODE reference for the event handlers.

Once the `on_finish` continuation has been invoked, the `IO::Async::Process` object is removed from the containing IO::Async::Loop object.

The following parameters may be passed to `new`, or to `configure` before the process has been started (i.e. before it has been added to the `Loop`). Once the process is running these cannot be changed.

command => ARRAY or STRING
Either a reference to an array containing the command and its arguments, or a plain string containing the command. This value is passed into perl's `exec(2)` function.

code => CODE
A block of code to execute in the child process. It will be called in scalar context inside an `eval` block.

setup => ARRAY
Optional reference to an array to pass to the underlying `Loop` `spawn_child` method.

fd*n* => HASH
A hash describing how to set up file descriptor *n*. The hash may contain the following keys:

via => STRING
Configures how this file descriptor will be configured for the child process. Must be given one of the following mode names:

pipe_read
The child will be given the writing end of a `pipe(2)`; the parent may read from the other.

pipe_write
The child will be given the reading end of a `pipe(2)`; the parent may write to the other. Since an EOF condition of this kind of handle cannot reliably be detected, `on_finish` will not wait for this type of pipe to be closed.

pipe_rdwr
Only valid on the `stdio` filehandle. The child will be given the reading end of one `pipe(2)` on STDIN and the writing end of another on STDOUT. A single Stream object will be created in the parent configured for both filehandles.

socketpair
The child will be given one end of a `socketpair(2)`; the parent will be given the other. The family of this socket may be given by the extra key called `family`; defaulting to `unix`. The socktype of this socket may be given by the extra key called `socktype`; defaulting to `stream`. If the type is not `SOCK_STREAM` then a IO::Async::Socket object will be constructed for the parent side of the handle, rather than IO::Async::Stream.

Once the filehandle is set up, the `fd` method (or its shortcuts of `stdin`, `stdout` or `stderr`) may be used to access the IO::Async::Handle−subclassed object wrapped around it.

The value of this argument is implied by any of the following alternatives.

on_read => CODE
The child will be given the writing end of a pipe. The reading end will be wrapped by an IO::Async::Stream using this `on_read` callback function.

into => SCALAR
The child will be given the writing end of a pipe. The referenced scalar will be filled by data read from the child process. This data may not be available until the pipe has been closed by the child.

from => STRING
The child will be given the reading end of a pipe. The string given by the `from` parameter will be written to the child. When all of the data has been written the pipe will be closed.

prefork => CODE
Only valid for handles with a `via` of `socketpair`. The code block runs after the `socketpair(2)` is created, but before the child is forked. This is handy for when you adjust both ends of the created socket (for example, to use `setsockopt(3)`) from the controlling parent, before the child code runs. The arguments passed in are the IO::Socket objects for the parent and child ends of the socket.

```
$prefork->( $localfd, $childfd )
```

stdin => ...
stdout => ...
stderr => ...
Shortcuts for `fd0`, `fd1` and `fd2` respectively.

stdio => ...
Special filehandle to affect STDIN and STDOUT at the same time. This filehandle supports being configured for both reading and writing at the same time.

METHODS

pid

```
$pid = $process->pid
```

Returns the process ID of the process, if it has been started, or `undef` if not. Its value is preserved after the process exits, so it may be inspected during the `on_finish` or `on_exception` events.

kill

```
$process->kill( $signal )
```

Sends a signal to the process

is_running

```
$running = $process->is_running
```

Returns true if the Process has been started, and has not yet finished.

is_exited

```
$exited = $process->is_exited
```

Returns true if the Process has finished running, and finished due to normal `exit(2)`.

exitstatus

```
$status = $process->exitstatus
```

If the process exited due to normal `exit(2)`, returns the value that was passed to `exit(2)`. Otherwise, returns `undef`.

exception

```
$exception = $process->exception
```

If the process exited due to an exception, returns the exception that was thrown. Otherwise, returns `undef`.

errno

```
$errno = $process->errno
```

If the process exited due to an exception, returns the numerical value of `$!` at the time the exception was thrown. Otherwise, returns `undef`.

errstr

```
$errstr = $process->errstr
```

If the process exited due to an exception, returns the string value of `$!` at the time the exception was thrown. Otherwise, returns `undef`.

fd

```
$stream = $process->fd( $fd )
```

Returns the IO::Async::Stream or IO::Async::Socket associated with the given FD number. This must have been set up by a `configure` argument prior to adding the `Process` object to the `Loop`.

The returned object have its read or write handle set to the other end of a pipe or socket connected to that FD number in the child process. Typically, this will be used to call the `write` method on, to write more data into the child, or to set an `on_read` handler to read data out of the child.

The `on_closed` event for these streams must not be changed, or it will break the close detection used by the `Process` object and the `on_finish` event will not be invoked.

stdin
stdout
stderr
stdio

```
$stream = $process->stdin

$stream = $process->stdout

$stream = $process->stderr

$stream = $process->stdio
```

Shortcuts for calling `fd` with 0, 1, 2 or `io` respectively, to obtain the IO::Async::Stream representing the standard input, output, error, or combined input/output streams of the child process.

EXAMPLES

Capturing the STDOUT stream of a process

By configuring the `stdout` filehandle of the process using the `into` key, data written by the process can be captured.

```
my $stdout;
my $process = IO::Async::Process->new(
   command => [ "writing-program", "arguments" ],
   stdout => { into => \$stdout },
   on_finish => sub {
      my $process = shift;
      my ( $exitcode ) = @_;
      print "Process has exited with code $exitcode, and wrote:\n";
      print $stdout;
   }
);

$loop->add( $process );
```

Note that until `on_finish` is invoked, no guarantees are made about how much of the data actually written by the process is yet in the `$stdout` scalar.

See also the `run_child` method of IO::Async::Loop.

To handle data more interactively as it arrives, the `on_read` key can instead be used, to provide a callback function to invoke whenever more data is available from the process.

```
my $process = IO::Async::Process->new(
   command => [ "writing-program", "arguments" ],
   stdout => {
      on_read => sub {
         my ( $stream, $buffref ) = @_;
         while( $$buffref =~ s/^(.*)\n// ) {
            print "The process wrote a line: $1\n";
         }

         return 0;
      },
   },
   on_finish => sub {
      print "The process has finished\n";
   }
);
```

```
$loop->add( $process );
```

If the code to handle data read from the process isn't available yet when the object is constructed, it can be supplied later by using the `configure` method on the `stdout` filestream at some point before it gets added to the Loop. In this case, `stdin` should be configured using `pipe_read` in the `via` key.

```
my $process = IO::Async::Process->new(
   command => [ "writing-program", "arguments" ],
   stdout => { via => "pipe_read" },
   on_finish => sub {
      print "The process has finished\n";
   }
);

$process->stdout->configure(
   on_read => sub {
      my ( $stream, $buffref ) = @_;
      while( $$buffref =~ s/^(.*)\n// ) {
         print "The process wrote a line: $1\n";
      }

      return 0;
   },
);

$loop->add( $process );
```

Sending data to STDIN of a process

By configuring the `stdin` filehandle of the process using the `from` key, data can be written into the `STDIN` stream of the process.

```
my $process = IO::Async::Process->new(
   command => [ "reading-program", "arguments" ],
   stdin => { from => "Here is the data to send\n" },
   on_finish => sub {
      print "The process has finished\n";
   }
);

$loop->add( $process );
```

The data in this scalar will be written until it is all consumed, then the handle will be closed. This may be useful if the program waits for EOF on `STDIN` before it exits.

To have the ability to write more data into the process once it has started. the `write` method on the `stdin` stream can be used, when it is configured using the `pipe_write` value for `via`:

```
my $process = IO::Async::Process->new(
   command => [ "reading-program", "arguments" ],
   stdin => { via => "pipe_write" },
   on_finish => sub {
      print "The process has finished\n";
   }
);

$loop->add( $process );

$process->stdin->write( "Here is some more data\n" );
```

Setting socket options

By using the `prefork` code block you can change the socket receive buffer size at both ends of the socket before the child is forked (at which point it would be too late for the parent to be able to change the child end of the socket).

```
use Socket qw( SOL_SOCKET SO_RCVBUF );

my $process = IO::Async::Process->new(
   command => [ "command-to-read-from-and-write-to", "arguments" ],
   stdio => {
      via => "socketpair",
      prefork => sub {
         my ( $parentfd, $childfd ) = @_;

         # Set parent end of socket receive buffer to 3 MB
         $parentfd->setsockopt(SOL_SOCKET, SO_RCVBUF, 3 * 1024 * 1024);
         # Set child end of socket receive buffer to 3 MB
         $childfd ->setsockopt(SOL_SOCKET, SO_RCVBUF, 3 * 1024 * 1024);
      },
   },
);

$loop->add( $process );
```

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>