NAME

IPC::Cmd - finding and running system commands made easy

SYNOPSIS

```
use IPC::Cmd qw[can_run run run_forked];

my $full_path = can_run('wget') or warn 'wget is not installed!';

### commands can be arrayrefs or strings ###
my $cmd = "$full_path -b theregister.co.uk";
my $cmd = [$full_path, '-b', 'theregister.co.uk'];

### in scalar context ###
my $buffer;
if( scalar run( command => $cmd,
                verbose => 0,
                buffer  => \$buffer,
                timeout => 20 )
) {
    print "fetched webpage successfully: $buffer\n";
}

### in list context ###
my( $success, $error_code, $full_buf, $stdout_buf, $stderr_buf ) =
        run( command => $cmd, verbose => 0 );

if( $success ) {
    print "this is what the command printed:\n";
    print join "", @$full_buf;
}

### check for features
print "IPC::Open3 available: "  . IPC::Cmd->can_use_ipc_open3;
print "IPC::Run available: "    . IPC::Cmd->can_use_ipc_run;
print "Can capture buffer: "    . IPC::Cmd->can_capture_buffer;

### don't have IPC::Cmd be verbose, ie don't print to stdout or
### stderr when running commands -- default is '0'
$IPC::Cmd::VERBOSE = 0;
```

DESCRIPTION

IPC::Cmd allows you to run commands, interactively if desired, platform independent but have them still work.

The `can_run` function can tell you if a certain binary is installed and if so where, whereas the `run` function can actually execute any of the commands you give it and give you a clear return value, as well as adhere to your verbosity settings.

CLASS METHODS

$ipc_run_version = IPC::Cmd->can_use_ipc_run([VERBOSE])

Utility function that tells you if `IPC::Run` is available. If the verbose flag is passed, it will print diagnostic messages if `IPC::Run` can not be found or loaded.

$ipc_open3_version = IPC::Cmd->can_use_ipc_open3([VERBOSE])

Utility function that tells you if `IPC::Open3` is available. If the verbose flag is passed, it will print diagnostic messages if `IPC::Open3` can not be found or loaded.

$bool = IPC::Cmd->can_capture_buffer

Utility function that tells you if `IPC::Cmd` is capable of capturing buffers in it's current configuration.

$bool = IPC::Cmd->can_use_run_forked

Utility function that tells you if `IPC::Cmd` is capable of providing `run_forked` on the current platform.

FUNCTIONS

$path = can_run(PROGRAM);

`can_run` takes but a single argument: the name of a binary you wish to locate. `can_run` works much like the unix binary `which` or the bash command `type`, which scans through your path, looking for the requested binary .

Unlike `which` and `type`, this function is platform independent and will also work on, for example, Win32.

It will return the full path to the binary you asked for if it was found, or `undef` if it was not.

$ok | ($ok, $err, $full_buf, $stdout_buff, $stderr_buff) = run(command => COMMAND, [verbose => BOOL, buffer => \$SCALAR, timeout => DIGIT]);

`run` takes 4 arguments:

command

This is the command to execute. It may be either a string or an array reference. This is a required argument.

See *CAVEATS* for remarks on how commands are parsed and their limitations.

verbose

This controls whether all output of a command should also be printed to STDOUT/STDERR or should only be trapped in buffers (NOTE: buffers require `IPC::Run` to be installed or your system able to work with `IPC::Open3`).

It will default to the global setting of `$IPC::Cmd::VERBOSE`, which by default is 0.

buffer

This will hold all the output of a command. It needs to be a reference to a scalar. Note that this will hold both the STDOUT and STDERR messages, and you have no way of telling which is which. If you require this distinction, run the `run` command in list context and inspect the individual buffers.

Of course, this requires that the underlying call supports buffers. See the note on buffers right above.

timeout

Sets the maximum time the command is allowed to run before aborting, using the built-in `alarm()` call. If the timeout is triggered, the `errorcode` in the return value will be set to an object of the `IPC::Cmd::TimeOut` class. See the `errorcode` section below for details.

Defaults to `0`, meaning no timeout is set.

`run` will return a simple `true` or `false` when called in scalar context. In list context, you will be returned a list of the following items:

success

A simple boolean indicating if the command executed without errors or not.

error message

If the first element of the return value (success) was 0, then some error occurred. This second element is the error message the command you requested exited with, if available. This is generally a pretty printed value of $? or $@. See `perldoc perlvar` for details on what they can contain. If the error was a timeout, the `error message` will be prefixed with the string `IPC::Cmd::TimeOut`, the timeout class.

full_buffer

This is an arrayreference containing all the output the command generated. Note that buffers are only available if you have `IPC::Run` installed, or if your system is able to work with `IPC::Open3` -- See below). This element will be `undef` if this is not the case.

out_buffer

This is an arrayreference containing all the output sent to STDOUT the command generated. Note that buffers are only available if you have `IPC::Run` installed, or if your system is able to work with `IPC::Open3` -- See below). This element will be `undef` if this is not the case.

error_buffer

This is an arrayreference containing all the output sent to STDERR the command generated. Note that buffers are only available if you have `IPC::Run` installed, or if your system is able to work with `IPC::Open3` -- See below). This element will be `undef` if this is not the case.

See the `HOW IT WORKS` Section below to see how `IPC::Cmd` decides what modules or function calls to use when issuing a command.

$hashref = run_forked(command => COMMAND, { child_stdin => SCALAR, timeout => DIGIT, stdout_handler => CODEREF, stderr_handler => CODEREF});

`run_forked` is used to execute some program, optionally feed it with some input, get its return code and output (both stdout and stderr into seperate buffers). In addition it allows to terminate the program which take too long to finish.

The important and distinguishing feature of run_forked is execution timeout which at first seems to be quite a simple task but if you think that the program which you're spawning might spawn some children itself (which in their turn could do the same and so on) it turns out to be not a simple issue.

`run_forked` is designed to survive and successfully terminate almost any long running task, even a fork bomb in case your system has the resources to survive during given timeout.

This is achieved by creating separate watchdog process which spawns the specified program in a separate process session and supervises it: optionally feeds it with input, stores its exit code, stdout and stderr, terminates it in case it runs longer than specified.

Invocation requires the command to be executed and optionally a hashref of options:

`timeout`

Specify in seconds how long the command may run for before it is killed with with SIG_KILL (9) which effectively terminates it and all of its children (direct or indirect).

`child_stdin`

Specify some text that will be passed into `STDIN` of the executed program.

`stdout_handler`

You may provide a coderef of a subroutine that will be called a portion of data is received on stdout from the executing program.

`stderr_handler`

You may provide a coderef of a subroutine that will be called a portion of data is received on

stderr from the executing program.

`run_forked` will return a HASHREF with the following keys:

`exit_code`

The exit code of the executed program.

`timeout`

The number of seconds the program ran for before being terminated, or 0 if no timeout occurred.

`stdout`

Holds the standard output of the executed command (or empty string if there were no stdout output; it's always defined!)

`stderr`

Holds the standard error of the executed command (or empty string if there were no stderr output; it's always defined!)

`merged`

Holds the standard output and error of the executed command merged into one stream (or empty string if there were no output at all; it's always defined!)

`err_msg`

Holds some explanation in the case of an error.

$q = QUOTE

Returns the character used for quoting strings on this platform. This is usually a `'` (single quote) on most systems, but some systems use different quotes. For example, `Win32` uses `"` (double quote).

You can use it as follows:

```
use IPC::Cmd qw[run QUOTE];
my $cmd = q[echo ] . QUOTE . q[foo bar] . QUOTE;
```

This makes sure that `foo bar` is treated as a string, rather than two seperate arguments to the `echo` function.

__END__

HOW IT WORKS

`run` will try to execute your command using the following logic:

- If you have `IPC::Run` installed, and the variable `$IPC::Cmd::USE_IPC_RUN` is set to true (See the `GLOBAL VARIABLES` Section) use that to execute the command. You will have the full output available in buffers, interactive commands are sure to work and you are guaranteed to have your verbosity settings honored cleanly.
- Otherwise, if the variable `$IPC::Cmd::USE_IPC_OPEN3` is set to true (See the `GLOBAL VARIABLES` Section), try to execute the command using `IPC::Open3`. Buffers will be available on all platforms except `Win32`, interactive commands will still execute cleanly, and also your verbosity settings will be adhered to nicely;
- Otherwise, if you have the verbose argument set to true, we fall back to a simple system() call. We cannot capture any buffers, but interactive commands will still work.
- Otherwise we will try and temporarily redirect STDERR and STDOUT, do a system() call with your command and then re-open STDERR and STDOUT. This is the method of last resort and

will still allow you to execute your commands cleanly. However, no buffers will be available.

Global Variables

The behaviour of IPC::Cmd can be altered by changing the following global variables:

$IPC::Cmd::VERBOSE

This controls whether IPC::Cmd will print any output from the commands to the screen or not. The default is 0;

$IPC::Cmd::USE_IPC_RUN

This variable controls whether IPC::Cmd will try to use *IPC::Run* when available and suitable. Defaults to true if you are on `Win32`.

$IPC::Cmd::USE_IPC_OPEN3

This variable controls whether IPC::Cmd will try to use *IPC::Open3* when available and suitable. Defaults to true.

$IPC::Cmd::WARN

This variable controls whether run time warnings should be issued, like the failure to load an `IPC::*` module you explicitly requested.

Defaults to true. Turn this off at your own risk.

Caveats

Whitespace and IPC::Open3 / system()

When using `IPC::Open3` or `system`, if you provide a string as the `command` argument, it is assumed to be appropriately escaped. You can use the `QUOTE` constant to use as a portable quote character (see above). However, if you provide and `Array Reference`, special rules apply:

If your command contains `Special Characters` (< > | &), it will be internally stringified before executing the command, to avoid that these special characters are escaped and passed as arguments instead of retaining their special meaning.

However, if the command contained arguments that contained whitespace, stringifying the command would loose the significance of the whitespace. Therefor, `IPC::Cmd` will quote any arguments containing whitespace in your command if the command is passed as an arrayref and contains special characters.

Whitespace and IPC::Run

When using `IPC::Run`, if you provide a string as the `command` argument, the string will be split on whitespace to determine the individual elements of your command. Although this will usually just Do What You Mean, it may break if you have files or commands with whitespace in them.

If you do not wish this to happen, you should provide an array reference, where all parts of your command are already separated out. Note however, if there's extra or spurious whitespace in these parts, the parser or underlying code may not interpret it correctly, and cause an error.

Example: The following code

```
gzip -cdf foo.tar.gz | tar -xf -
```

should either be passed as

```
"gzip -cdf foo.tar.gz | tar -xf -"
```

or as

```
['gzip', '-cdf', 'foo.tar.gz', '|', 'tar', '-xf', '-']
```

But take care not to pass it as, for example

```
['gzip -cdf foo.tar.gz', '|', 'tar -xf -']
```

Since this will lead to issues as described above.

IO Redirect

Currently it is too complicated to parse your command for IO Redirections. For capturing STDOUT or STDERR there is a work around however, since you can just inspect your buffers for the contents.

Interleaving STDOUT/STDERR

Neither IPC::Run nor IPC::Open3 can interleave STDOUT and STDERR. For short bursts of output from a program, ie this sample:

```
for ( 1..4 ) {
    $_ % 2 ? print STDOUT $_ : print STDERR $_;
}
```

IPC::[Run|Open3] will first read all of STDOUT, then all of STDERR, meaning the output looks like 1 line on each, namely '13' on STDOUT and '24' on STDERR.

It should have been 1, 2, 3, 4.

This has been recorded in *rt.cpan.org* as bug #37532: Unable to interleave STDOUT and STDERR

See Also

`IPC::Run`, `IPC::Open3`

ACKNOWLEDGEMENTS

Thanks to James Mastros and Martijn van der Streek for their help in getting IPC::Open3 to behave nicely.

Thanks to Petya Kohts for the `run_forked` code.

BUG REPORTS

Please report bugs or other issues to <bug-ipc-cmd@rt.cpan.org>.

AUTHOR

This module by Jos Boumans <kane@cpan.org>.

COPYRIGHT