NAME

Lintian::Command – Utilities to execute other commands from lintian code

SYNOPSIS

```
use Lintian::Command qw(spawn);

# simplest possible call
my $success = spawn({}, ['command']);

# catch output
my $opts = {};
$success = spawn($opts, ['command']);
if ($success) {
    print "STDOUT: $opts->{out}\n";
    print "STDERR: $opts->{err}\n";
}

# from file to file
$opts = { in => 'infile.txt', out => 'outfile.txt' };
$success = spawn($opts, ['command']);

# piping
$success = spawn({}, ['command'], "|", ['othercommand']);
```

DESCRIPTION

Lintian::Command is a thin wrapper around IPC::Run, that catches exception and implements a useful default behaviour for input and output redirection.

Lintian::Command provides a function **spawn()** which is a wrapper around **IPC::Run::run()** resp. **IPC::Run::start()** (depending on whether a pipe is requested). To wait for finished child processes, it also provides the **reap()** function as a wrapper around **IPC::Run::finish()**.

`spawn($opts, @cmds)`

The `@cmds` array is given to **IPC::Run::run()** (or ::**start()**) unaltered, but should only be used for commands and piping symbols (i.e. all of the elements should be either an array reference, a code reference, '|', or '&'). I/O redirection is handled via the `$opts` hash reference. If you need more fine grained control than that, you should just use IPC::Run directly.

`$opts` is a hash reference which can be used to set options and to retrieve the status and output of the command executed.

The following hash keys can be set to alter the behaviour of **spawn()**:

in STDIN for the first forked child. Defaults to `\undef`.

CAVEAT: Due to #301774, passing a SCALAR ref as STDIN for the child leaks memory. The leak is plugged for the `\undef` case in spawn, but other scalar refs may still be leaked.

pipe_in

Use a pipe for STDIN and start the process in the background. You will need to close the pipe after use and call `$opts->{harness}->finish` in order for the started process to end properly.

out STDOUT of the last forked child. Will be set to a newly created scalar reference by default which can be used to retrieve the output after the call.

Can be '&N' (e.g. &2) to redirect it to (numeric) file descriptor.

out_append

STDOUT of all forked children, cannot be used with out and should only be used with files. Unlike out, this appends the output to the file instead of truncating the file.

pipe_out
: Use a pipe for STDOUT and start the process in the background. You will need to call `$opts->{harness}->finish` in order for the started process to end properly.

err
: STDERR of all forked children. Defaults to STDERR of the parent.

  Can be '&N' (e.g. &1) to redirect it to (numeric) file descriptor.

err_append
: STDERR of all forked children, cannot be used with err and should only be used with files. Unlike err, this appends the output to the file instead of truncating the file.

pipe_err
: Use a pipe for STDERR and start the process in the background. You will need to call `$opts->{harness}->finish` in order for the started process to end properly.

fail
: Configures the behaviour in case of errors. The default is 'exception', which will cause **spawn()** to die in case of exceptions thrown by IPC::Run. If set to 'error' instead, it will also die if the command exits with a non-zero error code. If exceptions should be handled by the caller, setting it to 'never' will cause it to store the exception in the `exception` key instead.

child_before_exec
: Run the given subroutine in each of the children before they run ``exec''.

  This is passed to ``harness'' in IPC::Run as the *init* keyword.

The following additional keys will be set during the execution of **spawn()**:

harness
: Will contain the IPC::Run object used for the call which can be used to query the exit values of the forked programs (E.g. with **results()** and **full_results()**) and to wait for processes started in the background.

exception
: If an exception is raised during the execution of the commands, and if `fail` is set to 'never', the exception will be caught and stored under this key.

success
: Will contain the return value of **spawn()**.

`reap($opts[, $opts[,...]])`
: If you used one of the `pipe_*` options to **spawn()** or used the shell-style ``&'' operator to send the process to the background, you will need to wait for your child processes to finish. For this you can use the **reap()** function, which you can call with the `$opts` hash reference you gave to **spawn()** and which will do the right thing. Multiple `$opts` can be passed.

  Note however that this function will not close any of the pipes for you, so you probably want to do that first before calling this function.

  The following keys of the `$opts` hash have roughly the same function as for **spawn()**:

  harness
  fail
  success
  exception

  All other keys are probably just ignored.

`kill($opts[, $opts[, ...]])`
: This is a simple wrapper around the kill_kill function. It doesn't allow any customisation, but takes an `$opts` hash ref and SIGKILLs the process two seconds after SIGTERM is sent. If multiple hash refs are passed it executes kill_kill on each of them. The return status is the ORed value of all the executions of kill_kill.

`done($opts)`
Check if a process and its children are done. This is useful when one wants to know whether **reap()** can be called without blocking waiting for the process. It takes a single hash reference as returned by spawn.

`safe_qx([$opts,] @cmds)`
Variant of spawn that emulates the `qx()` operator by returning the captured output.

It takes the same arguments as `spawn` and they have the same basic semantics with the following exceptions:

The initial `$opts` is optional.
If only a single command is to be run, the surrounding list reference can be omitted (see the examples below).

If `$opts` is given, caller must ensure that the output is captured as a scalar reference in `$opts->{out}` (possibly by omitting the "out" and "out_append" keys).

Furthermore, the commands should not be backgrounded, so they cannot use '&' nor (e.g. `$opts->{pipe_in}`).

If needed `$?` will be set after the call like for `qx()`.

Examples:

```
# Capture the output of a simple command
# - Both are eqv.
safe_qx('grep', 'some-pattern', 'path/to/file');
safe_qx(['grep', 'some-pattern', 'path/to/file']);

# Capture the output of some pipeline
safe_qx(['grep', 'some-pattern', 'path/to/file'], '|',
        ['head', '-n1'])

# Call nproc and capture stdout and stderr interleaved
safe_qx({ 'err' => '&1'}, 'nproc')

#  WRONG: Runs grep with 5 arguments including a literal "|" and
# "-n1", which will generally fail with bad arguments.
safe_qx('grep', 'some-pattern', 'path/to/file', '|',
        'head', '-n1')
```

Possible known issue: It might not possible to discard stdout and capture stderr instead.

EXPORTS

Lintian::Command exports nothing by default, but you can export the **spawn()** and **reap()** functions.

AUTHOR

Originally written by Frank Lichtenheld <djpig@debian.org> for Lintian.

SEE ALSO

lintian (1), IPC::Run