NAME

IPC::System::Simple − Run commands simply, with detailed diagnostics

SYNOPSIS

```
use IPC::System::Simple qw(system systemx capture capturex);

system("some_command");        # Command succeeds or dies!

system("some_command",@args);  # Succeeds or dies, avoids shell if @args

systemx("some_command",@args); # Succeeds or dies, NEVER uses the shell


# Capture the output of a command (just like backticks). Dies on error.
my $output = capture("some_command");

# Just like backticks in list context.  Dies on error.
my @output = capture("some_command");

# As above, but avoids the shell if @args is non−empty
my $output = capture("some_command", @args);

# As above, but NEVER invokes the shell.
my $output = capturex("some_command", @args);
my @output = capturex("some_command", @args);
```

DESCRIPTION

Calling Perl's in-built `system()` function is easy, determining if it was successful is *hard*. Let's face it, `$?` isn't the nicest variable in the world to play with, and even if you *do* check it, producing a well-formatted error string takes a lot of work.

`IPC::System::Simple` takes the hard work out of calling external commands. In fact, if you want to be really lazy, you can just write:

```
use IPC::System::Simple qw(system);
```

and all of your `system` commands will either succeed (run to completion and return a zero exit value), or die with rich diagnostic messages.

The `IPC::System::Simple` module also provides a simple replacement to Perl's backticks operator. Simply write:

```
use IPC::System::Simple qw(capture);
```

and then use the "**capture()**" command just like you'd use backticks. If there's an error, it will die with a detailed description of what went wrong. Better still, you can even use `capturex()` to run the equivalent of backticks, but without the shell:

```
use IPC::System::Simple qw(capturex);

my $result = capturex($command, @args);
```

If you want more power than the basic interface, including the ability to specify which exit values are acceptable, trap errors, or process diagnostics, then read on!

ADVANCED SYNOPSIS

```
use IPC::System::Simple qw(
  capture capturex system systemx run runx $EXITVAL EXIT_ANY
);

# Run a command, throwing exception on failure
```

```
  run("some_command");

  runx("some_command",@args);  # Run a command, avoiding the shell

  # Do the same thing, but with the drop-in system replacement.

  system("some_command");

  systemx("some_command", @args);

  # Run a command which must return 0..5, avoid the shell, and get the
  # exit value (we could also look at $EXITVAL)

  my $exit_value = runx([0..5], "some_command", @args);

  # The same, but any exit value will do.

  my $exit_value = runx(EXIT_ANY, "some_command", @args);

  # Capture output into $result and throw exception on failure

  my $result = capture("some_command");

  # Check exit value from captured command

  print "some_command exited with status $EXITVAL\n";

  # Captures into @lines, splitting on $/
  my @lines = capture("some_command");

  # Run a command which must return 0..5, capture the output into
  # @lines, and avoid the shell.

  my @lines  = capturex([0..5], "some_command", @args);
```

ADVANCED USAGE

run() and system()

`IPC::System::Simple` provides a subroutine called `run`, that executes a command using the same semantics as Perl's built-in `system`:

```
  use IPC::System::Simple qw(run);

  run("cat *.txt");           # Execute command via the shell
  run("cat","/etc/motd");     # Execute command without shell
```

The primary difference between Perl's in-built system and the `run` command is that `run` will throw an exception on failure, and allows a list of acceptable exit values to be set. See ‘‘Exit values’’ for further information.

In fact, you can even have `IPC::System::Simple` replace the default `system` function for your package so it has the same behaviour:

```
  use IPC::System::Simple qw(system);

  system("cat *.txt");  # system now succeeds or dies!
```

`system` and `run` are aliases to each other.

See also "**runx()**, **systemx()** and **capturex()**" for variants of `system()` and `run()` that never invoke the shell, even with a single argument.

capture()

A second subroutine, named `capture` executes a command with the same semantics as Perl's built-in backticks (and `qx()`):

```
use IPC::System::Simple qw(capture);

# Capture text while invoking the shell.
my $file  = capture("cat /etc/motd");
my @lines = capture("cat /etc/passwd");
```

However unlike regular backticks, which always use the shell, `capture` will bypass the shell when called with multiple arguments:

```
# Capture text while avoiding the shell.
my $file  = capture("cat", "/etc/motd");
my @lines = capture("cat", "/etc/passwd");
```

See also "**runx()**, **systemx()** and **capturex()**" for a variant of `capture()` that never invokes the shell, even with a single argument.

runx(), systemx() and capturex()

The `runx()`, `systemx()` and `capturex()` commands are identical to the multi-argument forms of `run()`, `system()` and `capture()` respectively, but *never* invoke the shell, even when called with a single argument. These forms are particularly useful when a command's argument list *might* be empty, for example:

```
systemx($cmd, @args);
```

The use of `systemx()` here guarantees that the shell will *never* be invoked, even if `@args` is empty.

Exception handling

In the case where the command returns an unexpected status, both `run` and `capture` will throw an exception, which if not caught will terminate your program with an error.

Capturing the exception is easy:

```
eval {
    run("cat *.txt");
};

if ($@) {
    print "Something went wrong − $@\n";
}
```

See the diagnostics section below for more details.

Exception cases

`IPC::System::Simple` considers the following to be unexpected, and worthy of exception:

- Failing to start entirely (eg, command not found, permission denied).
- Returning an exit value other than zero (but see below).
- Being killed by a signal.
- Being passed tainted data (in taint mode).

Exit values

Traditionally, system commands return a zero status for success and a non-zero status for failure. `IPC::System::Simple` will default to throwing an exception if a non-zero exit value is returned.

You may specify a range of values which are considered acceptable exit values by passing an *array* *reference* as the first argument. The special constant`EXIT_ANY` can be used to allo w *any* exit value to be returned.

```
use IPC::System::Simple qw(run system capture EXIT_ANY);

run( [0..5], "cat *.txt");             # Exit values 0−5 are OK

system( [0..5], "cat *.txt");          # This works the same way

my @lines = capture( EXIT_ANY, "cat *.txt"); # Any exit is fine.
```

The `run` and replacement `system` subroutines returns the exit value of the process:

```
my $exit_value = run( [0..5], "cat *.txt");

# OR:

my $exit_value = system( [0..5] "cat *.txt");

print "Program exited with value $exit_value\n";
```

$EXITVAL

The exit value of any command executed by `IPC::System::Simple` can always be retrieved from the `$IPC::System::Simple::EXITVAL` variable:

This is particularly useful when inspecting results from `capture`, which returns the captured text from the command.

```
use IPC::System::Simple qw(capture $EXITVAL EXIT_ANY);

my @enemies_defeated = capture(EXIT_ANY, "defeat_evil", "/dev/mordor");

print "Program exited with value $EXITVAL\n";
```

`$EXITVAL` will be set to `−1` if the command did not exit normally (eg, being terminated by a signal) or did not start. In this situation an exception will also be thrown.

WINDOWS-SPECIFIC NOTES

The `run` subroutine make available the full 32−bit exit value on Win32 systems. This has been true since `IPC::System::Simple` v0.06 when called with multiple arguments, and since v1.25 when called with a single argument. This is different from the previous versions of `IPC::System::Simple` and from Perl's in-build `system()` function, which can only handle 8−bit return values.

The `capture` subroutine always returns the 32−bit exit value under Windows. The`capture` subroutine also never uses the shell, even when passed a single argument.

The `run` subroutine always uses a shell when passed a single argument. On NT systems, it uses `cmd.exe` in the system root, and on non-NT systems it uses `command.com` in the system root.

As of `IPC::System::Simple` v1.25, the `runx` and `capturex` subroutines, as well as multiple-argument calls to the `run` and `capture` subroutines, have their arguments properly quoted, so that arugments with spaces and the like work properly. Unfortunately, this breaks any attempt to invoke the shell itself. If you really need to execute `cmd.exe` or `command.com`, use the single-argument form. For single-argument calls to `run` and `capture`, the argument must be properly shell-quoted in advance of the call.

Versions of `IPC::System::Simple` before v0.09 would not search the `PATH` environment variable when the multi-argument form of `run()` was called. Versions from v0.09 onwards correctly search the path provided the command is provided including the extension (eg, `notepad.exe` rather than just `notepad`, or `gvim.bat` rather than just `gvim`). If no extension is provided, `.exe` is assumed.

Signals are not supported on Windows systems. Sending a signal to a Windows process will usually cause it to exit with the signal number used.

DIAGNOSTICS

''%s'' failed to start: ''%s''
: The command specified did not even start. It may not exist, or you may not have permission to use it. The reason it could not start (as determined from `$!`) will be provided.

''%s'' unexpectedly returned exit value `%d`
: The command ran successfully, but returned an exit value we did not expect. The value returned is reported.

''%s'' died to signal ''%s'' (%d) `%s`
: The command was killed by a signal. The name of the signal will be reported, or `UNKNOWN` if it cannot be determined. The signal number is always reported. If we detected that the process dumped core, then the string `and dumped core` is appended.

IPC::System::Simple::%s called with no arguments
: You attempted to call `run` or `capture` but did not provide any arguments at all. At the very lease you need to supply a command to run.

IPC::System::Simple::%s called with no command
: You called `run` or `capture` with a list of acceptable exit values, but no actual command.

IPC::System::Simple::%s called with tainted argument ''%s''
: You called `run` or `capture` with tainted (untrusted) arguments, which is almost certainly a bad idea. To untaint your arguments you'll need to pass your data through a regular expression and use the resulting match variables. See ''Laundering and Detecting Tainted Data'' in perlsec for more information.

IPC::System::Simple::%s called with tainted environment `$ENV`{%s}
: You called `run` or `capture` but part of your environment was tainted (untrusted). You should either delete the named environment variable before calling `run`, or set it to an untainted value (usually one set inside your program). See ''Cleaning Up Your Path'' in perlsec for more information.

Error in IPC::System::Simple plumbing: ''%s'' − ''%s''
: Implementing the `capture` command involves dark and terrible magicks involving pipes, and one of them has sprung a leak. This could be due to a lack of file descriptors, although there are other possibilities.

  If you are able to reproduce this error, you are encouraged to submit a bug report according to the ''Reporting bugs'' section below.

Internal error in IPC::System::Simple: ''%s''
: You've found a bug in `IPC::System::Simple`. Please check to see if an updated version of `IPC::System::Simple` is available. If not, please file a bug report according to the ''Reporting bugs'' section below.

IPC::System::Simple::%s called with undefined command
: You've passed the undefined value as a command to be executed. While this is a very Zen-like action, it's not supported by Perl's current implementation.

DEPENDENCIES

This module depends upon Win32::Process when used on Win32 system. `Win32::Process` is b undled as a core module in ActivePerl 5.6 and above.

There are no non-core dependencies on non−Win32 systems.

COMPARISON TO OTHER APIs

Perl provides a range of in-built functions for handling external commands, and CPAN provides even more. The `IPC::System::Simple` differentiates itself from other options by providing:

Extremely detailed diagnostics

The diagnostics produced by `IPC::System::Simple` are designed to provide as much information as possible. Rather than requiring the developer to inspect `$?`, `IPC::System::Simple` does the hard work for you.

If an odd exit status is provided, you're informed of what it is. If a signal kills your process, you are informed of both its name and number. If tainted data or environment prevents your command from running, you are informed of exactly which data or environmental variable is tainted.

Exceptions on failure

`IPC::System::Simple` takes an aggressive approach to error handling. Rather than allow commands to fail silently, exceptions are thrown when unexpected results are seen. This allows for easy development using a try/catch style, and avoids the possibility of accidentally continuing after a failed command.

Easy access to exit status

The `run`, `system` and `capture` commands all set `$EXITVAL`, making it easy to determine the exit status of a command. Additionally, the `system` and `run` interfaces return the exit status.

Consistent interfaces

When called with multiple arguments, the `run`, `system` and `capture` interfaces *never* invoke the shell. This differs from the in-built Perl `system` command which may invoke the shell under Windows when called with multiple arguments. It differs from the in-built Perl backticks operator which always invokes the shell.

BUGS

When `system` is exported, the exotic form `system { $cmd } @args` is not supported. Attemping to use the exotic form is a syntax error. This affects the calling package *only*. Use`CORE::system` if you need it, or consider using the autodie module to replace `system` with lexical scope.

Core dumps are only checked for when a process dies due to a signal. It is not believed there are any systems where processes can dump core without dying to a signal.

`WIFSTOPPED` status is not checked, as perl never spawns processes with the `WUNTRACED` option.

Signals are not supported under Win32 systems, since they don't work at all like Unix signals. Win32 signals cause commands to exit with a given exit value, which this modules *does* capture.

Reporting bugs

Before reporting a bug, please check to ensure you are using the most recent version of `IPC::System::Simple`. Your problem may have already been fixed in a new release.

You can find the `IPC::System::Simple` bug-tracker at <http://rt.cpan.org/Public/Dist/Display.html?Name=IPC−System−Simple> . Please check to see if your bug has already been reported; if in doubt, report yours anyway.

Submitting a patch and/or failing test case will greatly expedite the fixing of bugs.

FEEDBACK

If you find this module useful, please consider rating it on the CPAN Ratings service at <http://cpanratings.perl.org/rate/?distribution=IPC−System−Simple> .

The module author loves to hear how `IPC::System::Simple` has made your life better (or worse). Feedback can be sent to <pjf@perltraining.com.au>.

SEE ALSO

autodie uses `IPC::System::Simple` to provide succeed-or-die replacements to `system` (and other built-ins) with lexical scope.

POSIX, IPC::Run::Simple, perlipc, perlport, IPC::Run, IPC::Run3, Win32::Process

AUTHOR

Paul Fenwick <pjf@cpan.org>

COPYRIGHT AND LICENSE