# Package 'processx'

May 8, 2019

**Title** Execute and Control System Processes

**Version** 3.3.1

**Description** Tools to run system processes in the background.
It can check if a background process is running; wait on a background process to finish; get the exit status of finished processes; kill background processes. It can read the standard output and error of the processes, using non-blocking connections. 'processx' can poll a process for standard output or error, with a timeout. It can also poll several processes at once.

**License** MIT + file LICENSE

**LazyData** true

**URL** https://github.com/r-lib/processx#readme

**BugReports** https://github.com/r-lib/processx/issues

**RoxygenNote** 6.1.1

**Imports** ps (>= 1.2.0), R6, utils

**Suggests** callr, covr, crayon, curl, debugme, parallel, testthat, withr

**Encoding** UTF-8

**NeedsCompilation** yes

**Author** Gábor Csárdi [aut, cre, cph] (<https://orcid.org/0000-0001-7098-9676>),
Winston Chang [aut],
RStudio [cph, fnd],
Mango Solutions [cph, fnd]

**Maintainer** Gábor Csárdi <csardi.gabor@gmail.com>

**Repository** CRAN

**Date/Publication** 2019-05-08 05:20:02 UTC

## R topics documented:

---

`base64_decode` *Base64 Encoding and Decoding*

---

## Description

Base64 Encoding and Decoding

## Usage

```
base64_decode(x)

base64_encode(x)
```

## Arguments

| | |
|---|---|
| `x` | Raw vector to encode / decode. |

## Value

Raw vector, result of the encoding / decoding.

---

`conn_create_fd` *Processx connections*

---

## Description

These functions are currently experimental and will change in the future. Note that processx connections are *not* compatible with R's built-in connection system.

## Usage

```
conn_create_fd(fd, encoding = "", close = TRUE)

conn_create_pipepair(encoding = "")

conn_read_chars(con, n = -1)

## S3 method for class 'processx_connection'
conn_read_chars(con, n = -1)
```

```
processx_conn_read_chars(con, n = -1)

conn_read_lines(con, n = -1)

## S3 method for class 'processx_connection'
conn_read_lines(con, n = -1)

processx_conn_read_lines(con, n = -1)

conn_is_incomplete(con)

## S3 method for class 'processx_connection'
conn_is_incomplete(con)

processx_conn_is_incomplete(con)

conn_write(con, str, sep = "\n", encoding = "")

## S3 method for class 'processx_connection'
conn_write(con, str, sep = "\n",
  encoding = "")

processx_conn_write(con, str, sep = "\n", encoding = "")

conn_create_file(filename, read = NULL, write = NULL)

conn_set_stdout(con, drop = TRUE)

conn_set_stderr(con, drop = TRUE)

conn_get_fileno(con)

conn_disable_inheritance()

## S3 method for class 'processx_connection'
close(con, ...)

processx_conn_close(con, ...)
```

## Arguments

| | |
|---|---|
| fd | Integer scalar, a Unix file descriptor. |
| encoding | Encoding of the readable connection when reading. |
| close | Whether to close the OS file descriptor when closing the connection. Sometimes you want to leave it open, and use it again in a conn_create_fd call. Encoding to re-encode str into when writing. |
| con | Processx connection object. |
| n | Number of characters or lines to read. -1 means all available characters or lines. |

| | |
|---|---|
| str | Character or raw vector to write. |
| sep | Separator to use if str is a character vector. Ignored if str is a raw vector. |
| filename | File name. |
| read | Whether the connection is readable. |
| write | Whethe the connection is writeable. |
| drop | Whether to close the original stdout/stderr, or keep it open and return a connection to it. |
| ... | Extra arguments, for compatibility with the close() generic, currently ignored by processx. |

## Details

conn_create_fd() creates a connection from a file descriptor.

conn_create_pipepair() creates a pair of connected connections, the first one is writeable, the second one is readable.

conn_read_chars() reads UTF-8 characters from the connections. If the connection itself is not UTF-8 encoded, it re-encodes it.

conn_read_lines() reads lines from a connection.

conn_is_incomplete() returns FALSE if the connection surely has no more data.

conn_write() writes a character or raw vector to the connection. It might not be able to write all bytes into the connection, in which case it returns the leftover bytes in a raw vector. Call conn_write() again with this raw vector.

conn_create_file() creates a connection to a file.

conn_set_stdout() set the standard output of the R process, to the specified connection.

conn_set_stderr() set the standard error of the R process, to the specified connection.

conn_get_fileno() return the integer file desciptor that belongs to the connection.

conn_disable_inheritance() can be called to disable the inheritance of all open handles. Call this function as soon as possible in a new process to avoid inheriting the inherited handles even further. The function is best effort to close the handles, it might still leave some handles open. It should work for stdin, stdout and stderr, at least.

---

# curl_fds *Create a pollable object from a curl multi handle's file descriptors*

---

## Description

Create a pollable object from a curl multi handle's file descriptors

## Usage

```
curl_fds(fds)
```

### Arguments

| | |
|---|---|
| fds | A list of file descriptors, as returned by curl::multi_fdset(). |

### Value

Pollable object, that be used with poll() directly.

---

## poll *Poll for process I/O or termination*

---

### Description

Wait until one of the specified connections or processes produce standard output or error, terminates, or a timeout occurs.

### Usage

```
poll(processes, ms)
```

### Arguments

| | |
|---|---|
| processes | A list of connection objects orprocess objects to wait on. (They can be mixed as well.) If this is a named list, then the returned list will have the same names. This simplifies the identification of the processes. |
| ms | Integer scalar, a timeout for the polling, in milliseconds. Supply -1 for an infitite timeout, and 0 for not waiting at all. |

### Value

A list of character vectors of length one or three. There is one list element for each connection/process, in the same order as in the input list. For connections the result is a single string scalar. For processes the character vectors' elements are named output, error and process. Possible values for each individual result are: nopipe, ready, timeout, closed, silent. See details about these below. process refers to the poll connection, see the poll_connection argument of the process initializer.

### Explanation of the return values

- nopipe means that the stdout or stderr from this process was not captured.
- ready means that the connection or the stdout or stderr from this process are ready to read from. Note that end-of-file on these outputs also triggers ready.
- timeout': the connections or processes are not ready to read from and a timeout happened.
- closed: the connection was already closed, before the polling started.
- silent: the connection is not ready to read from, but another connection was.

## Examples

```
## Different commands to run for windows and unix
## Not run:
cmd1 <- switch(
  .Platform$OS.type,
  "unix" = c("sh", "-c", "sleep 1; ls"),
  c("cmd", "/c", "ping -n 2 127.0.0.1 && dir /b")
)
cmd2 <- switch(
  .Platform$OS.type,
  "unix" = c("sh", "-c", "sleep 2; ls 1>&2"),
  c("cmd", "/c", "ping -n 2 127.0.0.1 && dir /b 1>&2")
)

## Run them. p1 writes to stdout, p2 to stderr, after some sleep
p1 <- process$new(cmd1[1], cmd1[-1], stdout = "|")
p2 <- process$new(cmd2[1], cmd2[-1], stderr = "|")

## Nothing to read initially
poll(list(p1 = p1, p2 = p2), 0)

## Wait until p1 finishes. Now p1 has some output
p1$wait()
poll(list(p1 = p1, p2 = p2), -1)

## Close p1's connection, p2 will have output on stderr, eventually
close(p1$get_output_connection())
poll(list(p1 = p1, p2 = p2), -1)

## Close p2's connection as well, no nothing to poll
close(p2$get_error_connection())
poll(list(p1 = p1, p2 = p2), 0)

## End(Not run)
```

---

`process` *External process*

---

## Description

Managing external processes from R is not trivial, and this class aims to help with this deficiency. It is essentially a small wrapper around the `system` base R function, to return the process id of the started process, and set its standard output and error streams. The process id is then used to manage the process.

## Usage

```
p <- process$new(command = NULL, args,
                 stdin = NULL, stdout = NULL, stderr = NULL,
```

```
                  connections = list(), poll_connection = NULL,
                  env = NULL, cleanup = TRUE, cleanup_tree = FALSE,
                  wd = NULL, echo_cmd = FALSE, supervise = FALSE,
                  windows_verbatim_args = FALSE,
                  windows_hide_window = FALSE,
                  encoding = "", post_process = NULL)

p$is_alive()
p$signal(signal)
p$interrupt()
p$kill(grace = 0.1, close_connections = TRUE)
p$kill_tree(grace = 0.1, close_connections = TRUE)
p$wait(timeout = -1)
p$get_pid()
p$get_exit_status()
p$get_start_time()

p$read_output(n = -1)
p$read_error(n = -1)
p$read_output_lines(n = -1)
p$read_error_lines(n = -1)
p$has_input_connection()
p$has_output_connection()
p$has_error_connection()
p$has_poll_connection()
p$get_input_connection()
p$get_output_connection()
p$get_error_connection()
p$get_poll_connection()
p$is_incomplete_output()
p$is_incomplete_error()
p$read_all_output()
p$read_all_error()
p$read_all_output_lines()
p$read_all_error_lines()
p$write_input(str, sep = "\n")
p$get_input_file()
p$get_output_file()
p$get_error_file()

p$poll_io(timeout)

p$get_result()

p$as_ps_handle()
p$get_name()
p$get_exe()
p$get_cmdline()
```

```
p$get_status()
p$get_username()
p$get_wd()
p$get_cpu_times()
p$get_memory_info()
p$suspend()
p$resume()

format(p)
print(p)
```

## Arguments

- p: process object.
- command: Character scalar, the command to run. Note that this argument is not passed to a shell, so no tilde-expansion or variable substitution is performed on it. It should not be quoted with base::shQuote(). See base::normalizePath() for tilde-expansion.
- args: Character vector, arguments to the command. They will be used as is, without a shell. They don't need to be escaped.
- stdin: What to do with the standard input. Possible values: NULL: set to the *null device*, i.e. no standard input is provided; a string, supply the specified string as standard input; "|": create a (writeable) connection for stdin.
- stdout: What to do with the standard output. Possible values: NULL: discard it; a string, redirect it to this file; "|": create a connection for it.
- stderr: What to do with the standard error. Possible values: NULL: discard it; a string, redirect it to this file; "|": create a connection for it; "2>&1": redirect it to the same connection (i.e. pipe or file) as stdout. "2>&1" is a way to keep standard output and error correctly interleaved.
- connections: A list of connections to pass to the child process. This is an experimental feature currently.
- poll_connection: Whether to create an extra connection to the process that allows polling, even if the standard input and standard output are not pipes. If this is NULL (the default), then this connection will be only created if standard output and standard error are not pipes, and connections is an empty list. If the poll connection is created, you can query it via p$get_poll_connection() and it is also included in the response to p$poll_io() and poll(). The numeric file descriptor of the poll connection comes right after stderr (2), and the connections listed in connections.
- env: Environment variables of the child process. If NULL, the parent's environment is inherited. On Windows, many programs cannot function correctly if some environment variables are not set, so we always set HOMEDRIVE, HOMEPATH, LOGONSERVER, PATH, SYSTEMDRIVE, SYSTEMROOT, TEMP, USERDOMAIN, USERNAME, USERPROFILE and WINDIR.
- cleanup: Whether to kill the process when the process object is garbage collected.
- cleanup_tree: Whether to kill the process and its child process tree when the process object is garbage collected.
- wd: working directory of the process. It must exist. If NULL, then the current working directory is used.

- `echo_cmd`: Whether to print the command to the screen before running it.
- `supervise`: Whether to register the process with a supervisor. If `TRUE`, the supervisor will ensure that the process is killed when the R process exits.
- `windows_verbatim_args`: Whether to omit quoting the arguments on Windows. It is ignored on other platforms.
- `windows_hide_window`: Whether to hide the application's window on Windows. It is ignored on other platforms.
- `signal`: An integer scalar, the id of the signal to send to the process. See `tools::pskill()` for the list of signals.
- `grace`: Currently not used.
- `close_connections`: Whether to close standard input, standard output, standard error connections and the poll connection, after killing the process.
- `timeout`: Timeout in milliseconds, for the wait or the I/O polling.
- `n`: Number of characters or lines to read.
- `str`: Character or raw vector to write to the standard input of the process. If a character vector with a marked encoding, it will be converted to `encoding`.
- `sep`: Separator to add between `str` elements if it is a character vector. It is ignored if `str` is a raw vector.
- `encoding`: The encoding to assume for `stdin`, `stdout` and `stderr`. By default the encoding of the current locale is used. Note that `processx` always reencodes the output of the `stdout` and `stderr` streams in UTF-8 currently. If you want to read them without any conversion, on all platforms, specify `"UTF-8"` as encoding.
- `post_process`: An optional function to run when the process has finished. Currently it only runs if `$get_result()` is called. It is only run once.

## Details

`$new()` starts a new process in the background, and then returns immediately.

`$is_alive()` checks if the process is alive. Returns a logical scalar.

`$signal()` sends a signal to the process. On Windows only the `SIGINT`, `SIGTERM` and `SIGKILL` signals are interpreted, and the special 0 signal, The first three all kill the process. The 0 signal return `TRUE` if the process is alive, and `FALSE` otherwise. On Unix all signals are supported that the OS supports, and the 0 signal as well.

`$interrupt()` sends an interrupt to the process. On Unix this is a `SIGINT` signal, and it is usually equivalent to pressing CTRL+C at the terminal prompt. On Windows, it is a CTRL+BREAK keypress. Applications may catch these events. By default they will quit.

`$kill()` kills the process. It also kills all of its child processes, except if they have created a new process group (on Unix), or job object (on Windows). It returns `TRUE` if the process was killed, and `FALSE` if it was no killed (because it was already finished/dead when `processx` tried to kill it).

`$kill_tree()` performs process tree cleanup, it kills the process (if still alive), together with any child (or grandchild, etc.) processes. It uses the *ps* package, so that needs to be installed, and *ps* needs to support the current platform as well. Process tree cleanup works by marking the process with an environment variable, which is inherited in all child processes. This allows finding descendents, even if they are orphaned, i.e. they are not connected to the root of the tree cleanup in the

process tree any more. `$kill_tree()` returns a named integer vector of the process ids that were killed, the names are the names of the processes (e.g. `"sleep"`, `"notepad.exe"`, `"Rterm.exe"`, etc.).

`$wait()` waits until the process finishes, or a timeout happens. Note that if the process never finishes, and the timeout is infinite (the default), then R will never regain control. It returns the process itself, invisibly. In some rare cases, `$wait()` might take a bit longer than specified to time out. This happens on Unix, when another package overwrites the processx SIGCHLD signal handler, after the processx process has started. One such package is parallel, if used with fork clusters, e.g. through `parallel::mcparallel()`.

`$get_pid()` returns the process id of the process.

`$get_exit_status` returns the exit code of the process if it has finished and `NULL` otherwise. On Unix, in some rare cases, the exit status might be `NA`. This happens if another package (or R itself) overwrites the processx SIGCHLD handler, after the processx process has started. In these cases processx cannot determine the real exit status of the process. One such package is parallel, if used with fork clusters, e.g. through the `parallel::mcparallel()` function.

`$get_start_time()` returns the time when the process was started.

`$is_supervised()` returns whether the process is being tracked by supervisor process.

`$supervise()` if passed `TRUE`, tells the supervisor to start tracking the process. If `FALSE`, tells the supervisor to stop tracking the process. Note that even if the supervisor is disabled for a process, if it was started with `cleanup = TRUE`, the process will still be killed when the object is garbage collected.

`$read_output()` reads from the standard output connection of the process. If the standard output connection was not requested, then then it returns an error. It uses a non-blocking text connection. This will work only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_error()` is similar to `$read_output`, but it reads from the standard error stream.

`$read_output_lines()` reads lines from standard output connection of the process. If the standard output connection was not requested, then then it returns an error. It uses a non-blocking text connection. This will work only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_error_lines()` is similar to `$read_output_lines`, but it reads from the standard error stream.

`$has_input_connection()` return `TRUE` if there is a connection object for standard input; in other words, if `stdout="|"`. It returns `FALSE` otherwise.

`$has_output_connection()` returns `TRUE` if there is a connection object for standard output; in other words, if `stdout="|"`. It returns `FALSE` otherwise.

`$has_error_connection()` returns `TRUE` if there is a connection object for standard error; in other words, if `stderr="|"`. It returns `FALSE` otherwise.

`$has_poll_connection()` return `TRUE` if there is a poll connection, `FALSE` otherwise.

`$get_input_connection()` returns a connection object, to the standard input stream of the process.

`$get_output_connection()` returns a connection object, to the standard output stream of the process.

`$get_error_conneciton()` returns a connection object, to the standard error stream of the process.

`$get_poll_connetion()` returns the poll connection, if the process has one.

`$is_incomplete_output()` return `FALSE` if the other end of the standard output connection was closed (most probably because the process exited). It return `TRUE` otherwise.

`$is_incomplete_error()` return `FALSE` if the other end of the standard error connection was closed (most probably because the process exited). It return `TRUE` otherwise.

`$read_all_output()` waits for all standard output from the process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character scalar. This will return content only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_all_error()` waits for all standard error from the process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character scalar. This will return content only if `stderr="|"` was used. Otherwise, it will throw an error.

`$read_all_output_lines()` waits for all standard output lines from a process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character vector. This will return content only if `stdout="|"` was used. Otherwise, it will throw an error.

`$read_all_error_lines()` waits for all standard error lines from a process. It does not return until the process has finished. Note that this process involves waiting for the process to finish, polling for I/O and potentically several `readLines()` calls. It returns a character vector. This will return content only if `stderr="|"` was used. Otherwise, it will throw an error.

`$write_input()` writes the character vector (separated by `sep`) to the standard input of the process. It will be converted to the specified encoding. This operation is non-blocking, and it will return, even if the write fails (because the write buffer is full), or if it suceeds partially (i.e. not the full string is written). It returns with a raw vector, that contains the bytes that were not written. You can supply this raw vector to `$write_input()` again, until it is fully written, and then the return value will be `raw(0)` (invisibly).

`$get_input_file()` if the `stdin` argument was a filename, this returns the absolute path to the file. If `stdin` was `"|"` or `NULL`, this simply returns that value.

`$get_output_file()` if the `stdout` argument was a filename, this returns the absolute path to the file. If `stdout` was `"|"` or `NULL`, this simply returns that value.

`$get_error_file()` if the `stderr` argument was a filename, this returns the absolute path to the file. If `stderr` was `"|"` or `NULL`, this simply returns that value.

`$poll_io()` polls the process's connections for I/O. See more in the *Polling* section, and see also the `poll()` function to poll on multiple processes.

`$get_result()` returns the result of the post processesing function. It can only be called once the process has finished. If the process has no post-processing function, then `NULL` is returned.

`$as_ps_handle()` returns a ps::ps_handle object, corresponding to the process. The following methods use the ps package on a `ps_handle` object, created automatically:

- `p$get_name()` calls `ps::ps_name()` to get the process name.
- `p$get_exe()` calls `ps::ps_exe()` to get the path of the executable.
- `p$get_cmdline()` calls `ps::ps_cmdline()` to get the command line.
- `p$get_status()` calls `ps::ps_status()` to get the process status.
- `p$get_username()` calls `ps::ps_username()` to get the username.

- p$get_wd() calls ps::ps_cwd() to get the current working directory.
- p$get_cpu_times() calls ps::ps_cpu_times() to get CPU usage data.
- p$get_memory_info() calls ps::ps_memory_info() to get memory usage data.
- p$suspend() calls ps::ps_suspend() to suspend the process.
- p$resume() calls ps::ps_resume() to resume a suspended process.

format(p) or p$format() creates a string representation of the process, usually for printing.

print(p) or p$print() shows some information about the process on the screen, whether it is running and it's process id, etc.

## Polling

The poll_io() function polls the standard output and standard error connections of a process, with a timeout. If there is output in either of them, or they are closed (e.g. because the process exits) poll_io() returns immediately.

In addition to polling a single process, the poll() function can poll the output of several processes, and returns as soon as any of them has generated output (or exited).

## Cleaning up background processes

processx kills processes that are not referenced any more (if cleanup is set to TRUE), or the whole subprocess tree (if cleanup_tree is also set to TRUE).

The cleanup happens when the references of the processes object are garbage collected. To clean up earlier, you can call the kill() or kill_tree() method of the process(es), from an on.exit() expression, or an error handler:

```
process_manager <- function() {
  on.exit({
    try(p1$kill(), silent = TRUE)
    try(p2$kill(), silent = TRUE)
  }, add = TRUE)
  p1 <- process$new("sleep", "3")
  p2 <- process$new("sleep", "10")
  p1$wait()
  p2$wait()
}
process_manager()
```

If you interrupt process_manager() or an error happens then both p1 and p2 are cleaned up immediately. Their connections will also be closed. The same happens at a regular exit.

## Examples

```
# CRAN does not like long-running examples
## Not run:
p <- process$new("sleep", "2")
p$is_alive()
p
```

```
p$kill()
p$is_alive()

p <- process$new("sleep", "2")
p$is_alive()
Sys.sleep(3)
p$is_alive()

## End(Not run)
```

---

run *Run external command, and wait until finishes*

---

## Description

run provides an interface similar to base::system() and base::system2(), but based on the process class. This allows some extra features, see below.

## Usage

```
run(command = NULL, args = character(), error_on_status = TRUE,
  wd = NULL, echo_cmd = FALSE, echo = FALSE, spinner = FALSE,
  timeout = Inf, stdout_line_callback = NULL, stdout_callback = NULL,
  stderr_line_callback = NULL, stderr_callback = NULL,
  stderr_to_stdout = FALSE, env = NULL,
  windows_verbatim_args = FALSE, windows_hide_window = FALSE,
  encoding = "", cleanup_tree = FALSE)
```

## Arguments

| | |
|---|---|
| command | Character scalar, the command to run. |
| args | Character vector, arguments to the command. |
| error_on_status | Whether to throw an error if the command returns with a non-zero status, or it is interrupted. The error clases are system_command_status_error and system_command_timeout_error, respectively, and both errors have class system_command_error as well. |
| wd | Working directory of the process. If NULL, the current working directory is used. |
| echo_cmd | Whether to print the command to run to the screen. |
| echo | Whether to print the standard output and error to the screen. Note that the order of the standard output and error lines are not necessarily correct, as standard output is typically buffered. |
| spinner | Whether to show a reassuring spinner while the process is running. |
| timeout | Timeout for the process, in seconds, or as a difftime object. If it is not finished before this, it will be killed. |

`stdout_line_callback`
: `NULL`, or a function to call for every line of the standard output. See `stdout_callback` and also more below.

`stdout_callback`
: `NULL`, or a function to call for every chunk of the standard output. A chunk can be as small as a single character. At most one of `stdout_line_callback` and `stdout_callback` can be non-`NULL`.

`stderr_line_callback`
: `NULL`, or a function to call for every line of the standard error. See `stderr_callback` and also more below.

`stderr_callback`
: `NULL`, or a function to call for every chunk of the standard error. A chunk can be as small as a single character. At most one of `stderr_line_callback` and `stderr_callback` can be non-`NULL`.

`stderr_to_stdout`
: Whether to redirect the standard error to the standard output. Specifying `TRUE` here will keep both in the standard output, correctly interleaved. However, it is not possible to deduce where pieces of the output were coming from. If this is `TRUE`, the standard error callbacks (if any) are never called.

`env`
: Environment of the child process, a named character vector. IF `NULL`, the environment of the parent is inherited.

`windows_verbatim_args`
: Whether to omit the escaping of the command and the arguments on windows. Ignored on other platforms.

`windows_hide_window`
: Whether to hide the window of the application on windows. Ignored on other platforms.

`encoding`
: The encoding to assume for `stdout` and `stderr`. By default the encoding of the current locale is used. Note that `processx` always reencodes the output of both streams in UTF-8 currently.

`cleanup_tree`
: Whether to clean up the child process tree after the process has finished.

## Details

`run` supports

- Specifying a timeout for the command. If the specified time has passed, and the process is still running, it will be killed (with all its child processes).
- Calling a callback function for each line or each chunk of the standard output and/or error. A chunk may contain multiple lines, and can be as short as a single character.
- Cleaning up the subprocess, or the whole process tree, before exiting.

## Value

A list with components:

- status The exit status of the process. If this is `NA`, then the process was killed and had no exit status.

- stdout The standard output of the command, in a character scalar.
- stderr The standard error of the command, in a character scalar.
- timeout Whether the process was killed because of a timeout.

## Callbacks

Some notes about the callback functions. The first argument of a callback function is a character scalar (length 1 character), a single output or error line. The second argument is always the process object. You can manipulate this object, for example you can call `$kill()` on it to terminate it, as a response to a message on the standard output or error.

## Examples

```
## Different examples for Unix and Windows
## Not run:
if (.Platform$OS.type == "unix") {
  run("ls")
  system.time(run("sleep", "10", timeout = 1,
    error_on_status = FALSE))
  system.time(
    run(
      "sh", c("-c", "for i in 1 2 3 4 5; do echo $i; sleep 1; done"),
      timeout = 2, error_on_status = FALSE
    )
  )
} else {
  run("ping", c("-n", "1", "127.0.0.1"))
  run("ping", c("-n", "6", "127.0.0.1"), timeout = 1,
    error_on_status = FALSE)
}

## End(Not run)
```

# Index