Package 'processx'

November 19, 2026

Config/rig/note This file was created by rig, do not edit it manually

Title Execute and Control System Processes

Version 3.10.0

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

URL https://processx.r-lib.org, https://github.com/r-lib/processx

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

Depends R (>= 3.4.0)

Imports ps (>= 1.9.3), R6, utils

Suggests callr (>= 3.7.3), cli (>= 3.3.0), codetools, covr, curl, debugme, parallel, pkgload, rlang (>= 1.0.2), testthat (>= 3.0.0), webfakes, withr

Config/Needs/check rcmdcheck

Config/Needs/coverage gaborcsardi/uncovr

Config/Needs/website pkgdown, tidyverse/tidytemplate

Config/testthat/edition 3

Config/usethis/last-upkeep 2025-04-25

Config/testthat/parallel true

Encoding UTF-8

Config/roxygen2/version 8.1.1

NeedsCompilation yes

Author Gábor Csárdi [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0001-7098-9676>),
Winston Chang [aut],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>),
Ascent Digital Services [cph, fnd]

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

Repository CRAN

Date/Publication 2026-11-19 06:30:02 UTC

Contents

base64_decode 2
conn_create_fd 3
conn_create_fifo 6
conn_create_unix_socket 8
curl_fds 9
default_pty_options 10
pipeline 10
poll 15
process 16
run 30

Index **35**

`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_file_name(con)

conn_create_pipepair(encoding = "", nonblocking = c(TRUE, FALSE))

conn_create_proc_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_bytes(con, n = -1)

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

processx_conn_read_bytes(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, ...)

is_valid_fd(fd)
```

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.
nonblocking	Whether the pipe should be non-blocking. For conn_create_pipepair() it must be a logical vector of length two, for both ends of the pipe.
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. For conn_create_fifo() on Windows, a \\?\pipe prefix is added to this, if it does not have such a prefix. For conn_create_fifo() it can also be NULL, in which case a random file name is used via tempfile().
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_file_name()` returns the name of the file associated with the connection. For connections that do not refer to a file in the file system it returns `NA_character()`. Except for named pipes on Windows, where it returns the full name of the pipe.

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

`conn_create_proc_pipepair()` creates a unidirectional pipe suitable for connecting two child processes: the first element is the write end (pass as `stdout` to the writing process) and the second is the read end (pass as `stdin` to the reading process). Unlike `conn_create_pipepair()`, both ends are synchronous (blocking), which is required for child-process stdin/stdout on Windows.

`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_bytes()` reads raw bytes from the connection into a raw vector. Unlike `conn_read_chars()`, it bypasses UTF-8 conversion, so null bytes and arbitrary binary data are preserved exactly. Calling this function switches the connection permanently to raw mode; after that, `conn_read_chars()` and `conn_read_lines()` must not be used on the same connection.

`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.

`is_valid_fd()` returns `TRUE` if `fd` is a valid open file descriptor. You can use it to check if the R process has standard input, output or error. E.g. R processes running in GUI (like RGui) might not have any of the standard streams available.

If a stream is redirected to the null device (e.g. in a callr subprocess), that is is still a valid file descriptor.

Examples

```
is_valid_fd(0L)      # stdin
is_valid_fd(1L)      # stdout
is_valid_fd(2L)      # stderr
```

conn_create_fifo *Processx FIFOs*

Description

[Experimental]

Create a FIFO for inter-process communication Note that these functions are currently experimental.

Usage

```
conn_create_fifo(
  filename = NULL,
  read = NULL,
  write = NULL,
  encoding = "",
  nonblocking = TRUE
)

conn_connect_fifo(
  filename,
  read = NULL,
  write = NULL,
  encoding = "",
  nonblocking = TRUE
)
```

Arguments

filename	File name of the FIFO. On Windows it the name of the pipe within the `\\?\pipe\` namespace, either the full name, or the part after that prefix. If `NULL`, then a random name is used, on Unix in the R temporary directory: `base::tempdir()`.
read	If `TRUE` then connect to the read end of the FIFO. Exactly one of `read` and `write` must be set to `TRUE`.
write	If `TRUE` then connect to the write end of the FIFO. Exactly one of `read` and `write` must be set to `TRUE`.
encoding	Encoding to assume.
nonblocking	Whether this should be a non-blocking FIFO. Note that blocking FIFOs are not well tested and might not work well with `poll()`, especially on Windows. We might remove this option in the future and make all FIFOs non-blocking.

Details

`conn_create_fifo()` creates a FIFO and connects to it. On Unix this is a proper FIFO in the file system, in the R temporary directory. On Windows it is a named pipe.

Use conn_file_name() to query the name of the FIFO, and conn_connect_fifo() to connect to the other end.

conn_connect_fifo() connects to a FIFO created with conn_create_fifo(), typically in another process. filename refers to the name of the pipe on Windows.

On Windows, conn_connect_fifo() may be successful even if the FIFO does not exist, but then later poll() or read/write operations will fail. We are planning on changing this behavior in the future, to make conn_connect_fifo() fail immediately, like on Unix.

Notes

In general Unix domain sockets work better than FIFOs, so we suggest:

you use sockets if you can. See conn_create_unix_socket().

Creating the read end of the FIFO:

This case is simpler. To wait for a writer to connect to the FIFO you can use poll() as usual. Then use conn_read_chars() or conn_read_lines() to read from the FIFO, as usual. Use conn_is_incomplete() *after* a read to check if there is more data, or the writer is done.

Creating the write end of the FIFO:

This is somewhat trickier. Creating the (non-blocking) FIFO does not block. However, there is no easy way to tell if a reader is connected to the other end of the FIFO or not. On Unix you can start using conn_write() to try to write to it, and this will succeed, until the buffer gets full, even if there is no reader. (When the buffer is full it will return the data that was not written, as usual.)

On Windows, using conn_write() to write to a FIFO without a reader fails with an error. This is not great, we are planning to improve it later.

Right now, one workaround for this behavior is for the reader to connunicate to the writer process independenctly that it has connected to the FIFO. (E.g. another FIFO in the opposite direction can do that.)

See Also

processx internals

Examples

```
# Example for a non-blocking FIFO

# Need to open the reading end first, otherwise Unix fails
reader <- conn_create_fifo()

# Always use poll() before you read, with a timeout if you like.
# If you read before the other end of the FIFO is connected, then
# the OS (or processx?) assumes that the FIFO is done, and you cannot
# read anything.
# Now poll() tells us that there is no data yet.
poll(list(reader), 0)

writer <- conn_connect_fifo(conn_file_name(reader), write = TRUE)
conn_write(writer, "hello\nthere!\n")
```

```
poll(list(reader), 1000)
conn_read_lines(reader, 1)
conn_read_chars(reader)

conn_is_incomplete(reader)

close(writer)
conn_read_chars(reader)
conn_is_incomplete(reader)

close(reader)
```

`conn_create_unix_socket` *Unix domain sockets*

Description

[Experimental]

Cross platform point-to-point inter-process communication with Unix=domain sockets, implemented via named pipes on Windows. These connection are always bidirectional, i.e. you can read from them and also write to them.

Usage

```
conn_create_unix_socket(filename = NULL, encoding = "")

conn_connect_unix_socket(filename, encoding = "")

conn_accept_unix_socket(con)

conn_unix_socket_state(con)
```

Arguments

`filename`	File name of the socket. On Windows it the name of the pipe within the `\\?\pipe\` namespace, either the full name, or the part after that prefix. If `NULL`, then a random name is used, on Unix in the R temporary directory: `base::tempdir()`.
`encoding`	Encoding to assume when reading from the socket.
`con`	Connection. An error is thrown if not a socket connection.

Details

`conn_create_unix_socket()` creates a server socket. The new socket is listening at `filename`. See `filename` above.

`conn_connect_unix_socket()` creates a client socket and connects it to a server socket.

`conn_accept_unix_socket()` accepts a client connection at a server socket.

`conn_unix_socket_state()` returns the state of the socket. Currently it can return: `"listening"`, `"connected_server"`, `"connected_client"`. It is possible that other states (e.g. for a closed socket) will be added in the future.

Notes:

- `poll()` works on sockets, but only polls for data to read, and currently ignores the write-end of the socket.
- `poll()` also works for accepting client connections. It will return `"connect"`is a client connection is available for a server socket. After this you can call `conn_accept_unix_socket()` to accept the client connection.

Value

A new socket connection.

See Also

processx internals

`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.

default_pty_options *Default options for pseudo terminals (ptys)*

Description

Default options for pseudo terminals (ptys)

Usage

```
default_pty_options()
```

Value

Named list of default values of pty options.

Options and default values:

- `echo` whether to keep the echo on the terminal. `FALSE` turns echo off.
- `rows` the (initial) terminal size, number of rows.
- `cols` the (initial) terminal size, number of columns.

pipeline *Pipeline of processes connected with pipes*

Description

[Experimental]

A `pipeline` object represents a sequence of processes whose standard input and output streams are connected with pipes, like a Unix pipeline (`cmd1 | cmd2 | cmd3`). Data flows directly between the child processes via kernel-level pipes — the parent R process sees only the output of the final command (when `stdout = "|"`).

Methods

`pipeline$new(cmds, stdin, stdout, stderr, env, encoding, wd, cleanup, cleanup_tree)`
`$read_output(n = -1)`, `$read_output_lines(n = -1)`, `$read_all_output()`, `$read_all_output_lines()` — read from the last process (only meaningful when `stdout = "|"`).
`$poll_io(timeout)` — poll the last process's connections for I/O.
`$read_error(n = -1)`, `$read_error_lines(n = -1)`, `$read_all_error()`, `$read_all_error_lines()` — read stderr of the last process (only meaningful when `stderr = "|"`).
`$write_input(str, sep = "\n")` — write to first process stdin (only meaningful when `stdin = "|"`).
`$close_input()` — close the first process stdin, signalling EOF.
`$wait(timeout = -1)` — wait for all processes to finish.
`$kill(grace = 0, close_connections = TRUE)` — kill all processes.

`$kill_tree(grace = 0, close_connections = TRUE)` — kill all process trees.
`$is_alive()` — returns `TRUE` if any process is still running.
`$get_exit_statuses()` — list of exit codes (one per process; `NULL` if still running).
`$get_pids()` — integer vector of process IDs.
`$get_processes()` — list of process objects, one per command.
`$format()` — string representation of the pipeline.
`$print()` — print the pipeline to the screen.

Methods

Public methods:

- `pipeline$new()`
- `pipeline$read_output()`
- `pipeline$read_output_lines()`
- `pipeline$read_all_output()`
- `pipeline$read_all_output_lines()`
- `pipeline$poll_io()`
- `pipeline$read_error()`
- `pipeline$read_error_lines()`
- `pipeline$read_all_error()`
- `pipeline$read_all_error_lines()`
- `pipeline$write_input()`
- `pipeline$close_input()`
- `pipeline$wait()`
- `pipeline$kill()`
- `pipeline$kill_tree()`
- `pipeline$is_alive()`
- `pipeline$get_exit_statuses()`
- `pipeline$get_pids()`
- `pipeline$get_processes()`
- `pipeline$format()`
- `pipeline$print()`

`pipeline$new()`: Create a new pipeline.

Usage:

```
pipeline$new(
  cmds,
  stdin = NULL,
  stdout = "|",
  stderr = NULL,
  env = NULL,
  encoding = "utf-8",
  wd = NULL,
  cleanup = TRUE,
  cleanup_tree = FALSE
)
```

Arguments:

`cmds` A non-empty list of character vectors. Each vector is one command: the first element is the executable and the rest are its arguments. Example: `list(c("sort"), c("uniq", "-c"))`.

`stdin` Standard input for the *first* process. `NULL` to discard, `"|"` so the parent R process can write to it via `$write_input()`, or a file path.

`stdout` Standard output of the *last* process. `"|"` (the default) so the parent R process can read from it, `NULL` to discard, or a file path.

`stderr` Standard error for *all* processes. `NULL` (the default) to discard, `"|"` to create a separate readable pipe per process, `"2>&1"` to merge into stdout, or a file path. When `"|"`, use `$read_error()` to read from the last process; use `$get_processes()` to access individual process objects for other processes.

`env` Environment variables for all processes, or `NULL` to inherit the parent environment.

`encoding` Assumed encoding for stdin/stdout/stderr streams.

`wd` Working directory for all processes, or `NULL` for the current directory.

`cleanup` Whether to kill the processes on garbage collection.

`cleanup_tree` Whether to kill the full process trees on garbage collection.

`pipeline$read_output()`**:** Read output of the last process.

Usage:

```
pipeline$read_output(n = -1)
```

Arguments:

`n` Number of characters or lines to read. -1 means all available.

`pipeline$read_output_lines()`**:** Read output lines of the last process.

Usage:

```
pipeline$read_output_lines(n = -1)
```

Arguments:

`n` Number of characters or lines to read. -1 means all available.

`pipeline$read_all_output()`**:** Read all output of the last process.

Usage:

```
pipeline$read_all_output()
```

`pipeline$read_all_output_lines()`**:** Read all output lines of the last process.

Usage:

```
pipeline$read_all_output_lines()
```

`pipeline$poll_io()`**:** Poll the connections of the last process for I/O.

Usage:

```
pipeline$poll_io(timeout)
```

Arguments:

`timeout` Timeout in milliseconds. -1 means no timeout.

`pipeline$read_error()`: Read stderr of the last process.

Usage:

```
pipeline$read_error(n = -1)
```

Arguments:

`n` Number of characters or lines to read. -1 means all available.

`pipeline$read_error_lines()`: Read stderr lines of the last process.

Usage:

```
pipeline$read_error_lines(n = -1)
```

Arguments:

`n` Number of characters or lines to read. -1 means all available.

`pipeline$read_all_error()`: Read all stderr of the last process.

Usage:

```
pipeline$read_all_error()
```

`pipeline$read_all_error_lines()`: Read all stderr lines of the last process.

Usage:

```
pipeline$read_all_error_lines()
```

`pipeline$write_input()`: Write to the first process stdin.

Usage:

```
pipeline$write_input(str, sep = "\n")
```

Arguments:

`str` String to write to the process stdin.

`sep` Separator to add after `str`.

`pipeline$close_input()`: Close the first process stdin (signals EOF to the process).

Usage:

```
pipeline$close_input()
```

`pipeline$wait()`: Wait for all processes to finish.

Usage:

```
pipeline$wait(timeout = -1)
```

Arguments:

`timeout` Timeout in milliseconds. -1 means no timeout.

`pipeline$kill()`: Kill all processes.

Usage:

```
pipeline$kill(grace = 0, close_connections = TRUE)
```

Arguments:

`grace` Grace period in seconds, passed to the `$kill()` method of each process. See process for details.

`close_connections` Whether to close connections after killing.

`pipeline$kill_tree()`: Kill all process trees.

Usage:

```
pipeline$kill_tree(grace = 0, close_connections = TRUE)
```

Arguments:

`grace` Grace period in seconds, passed to the `$kill()` method of each process. See process for details.

`close_connections` Whether to close connections after killing.

`pipeline$is_alive()`: Check if any process is still alive.

Usage:

```
pipeline$is_alive()
```

`pipeline$get_exit_statuses()`: Return exit codes for all processes.

Usage:

```
pipeline$get_exit_statuses()
```

`pipeline$get_pids()`: Return PIDs for all processes.

Usage:

```
pipeline$get_pids()
```

`pipeline$get_processes()`: Return the list of process objects.

Usage:

```
pipeline$get_processes()
```

`pipeline$format()`: Format the pipeline as a string.

Usage:

```
pipeline$format()
```

`pipeline$print()`: Print the pipeline to the screen.

Usage:

```
pipeline$print(...)
```

Arguments:

`...` Not used, for compatibility with the generic.

Examples

```
## Not run:
# sort | uniq, reading from / writing to R
pl <- pipeline$new(
  list(c("sort"), c("uniq")),
  stdin = "|", stdout = "|"
)
pl$write_input("b\na\nb\na\n")
pl$close_input()
```

```
pl$read_all_output_lines()
pl$wait()
pl$get_exit_statuses()

## End(Not run)
```

`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 or`process` 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
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)
```

`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.

Batch files

Running Windows batch files (`.bat` or `.cmd` files) may be complicated because of the `cmd.exe` command line parsing rules. For example you cannot easily have whitespace in both the command (path) and one of the arguments. To work around these limitations you need to start a `cmd.exe` shell explicitly and use its `call` command. For example:

```
process$new("cmd.exe", c("/c", "call", bat_file, "arg 1", "arg 2"))
```

This works even if `bat_file` contains whitespace characters. For more information about this, see this processx issue: https://github.com/r-lib/processx/issues/301

The detailed parsing rules are at https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cmd

A very good practical guide is at https://ss64.com/nt/syntax-esc.html

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).

Always call `$poll_io()` (or `poll()`) before reading from the stdout or stderr pipes. The OS pipe buffer is finite (typically 64KB on Linux/macOS, ~76KB on Windows). If the child process fills the pipe buffer before the parent reads from it, the child blocks waiting for the buffer to drain, while the parent may be waiting for the child — resulting in a deadlock. Polling drains the buffer and prevents this. Even a zero-timeout poll (`$poll_io(0)`) is sufficient when you know output is available; use a positive timeout (or `-1` to wait indefinitely) when you need to wait for output to arrive.

Note also that `$read_output()` and `$read_error()` may return *less* data than requested: a single call is not guaranteed to return all buffered output. Call them in a loop (polling before each read) until `$is_incomplete_output()` / `$is_incomplete_error()` returns `FALSE` to collect everything. The `$read_all_output()` and `$read_all_error()` helpers already do this for you.

Cleaning up background processes

processx provides several mechanisms to clean up background processes. See the Process cleanup article for a full discussion. A brief summary:

- **Explicit cleanup** (most reliable): call `$kill()` or `$kill_tree()` from an `on.exit()` expression or 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.

- **Automatic GC cleanup** (`cleanup = TRUE`, the default): the process is killed when the `process` R object is garbage collected. On Unix, `kill(-pid, SIGKILL)` is used, which kills the child's whole process group (since the child calls `setsid()` on startup). On Windows, the child is added to a global Job Object with `JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE`, so it is also killed if R exits or crashes. GC timing is non-deterministic; prefer `on.exit()` when determinism matters.
- **Process tree cleanup** (`cleanup_tree = TRUE`): kills the process and all its descendants, including orphaned ones. processx marks each child with a unique environment variable (`PROCESSX_<id>=YES`) that is inherited by all descendants; `$kill_tree()` uses the *ps* package to find and kill every process carrying that variable. On macOS, system restrictions may prevent reading other processes' environment, so tree cleanup may not work reliably.
- **Linux parent-death signal** (`linux_pdeathsig`): on Linux, the kernel can send a signal (e.g. `SIGTERM`) to the child when the parent R process exits, including on crash. Pass `linux_pdeathsig = TRUE` for `SIGTERM`, or an integer signal number. Ignored on non-Linux platforms.
- **Supervisor** (`supervise = TRUE`): a separate native process that polls every 200 ms and kills registered children if the parent R process dies (including crashes). On Unix it sends SIGTERM then (after 5 s) SIGKILL. On Windows it sends CTRL+C / WM_CLOSE then hard-kills. Note: on Windows, antivirus software may block `supervisor.exe`.

Methods

Public methods:

- `process$new()`
- `process$kill()`
- `process$kill_tree()`
- `process$signal()`
- `process$interrupt()`
- `process$get_pid()`
- `process$is_alive()`
- `process$wait()`
- `process$get_exit_status()`
- `process$format()`
- `process$print()`
- `process$get_start_time()`
- `process$get_end_time()`
- `process$is_supervised()`
- `process$supervise()`
- `process$read_output()`
- `process$read_error()`
- `process$read_output_bytes()`
- `process$read_error_bytes()`
- `process$read_output_lines()`

- process$read_error_lines()
- process$is_incomplete_output()
- process$is_incomplete_error()
- process$has_input_connection()
- process$has_output_connection()
- process$has_error_connection()
- process$has_poll_connection()
- process$get_input_connection()
- process$get_output_connection()
- process$get_error_connection()
- process$read_all_output()
- process$read_all_error()
- process$read_all_output_lines()
- process$read_all_error_lines()
- process$write_input()
- process$get_input_file()
- process$get_output_file()
- process$get_error_file()
- process$poll_io()
- process$get_poll_connection()
- process$get_result()
- process$as_ps_handle()
- process$get_name()
- process$get_exe()
- process$get_cmdline()
- process$get_status()
- process$get_username()
- process$get_wd()
- process$get_cpu_times()
- process$get_memory_info()
- process$suspend()
- process$resume()
- process$clone()

`process$new()`**:** Start a new process in the background, and then return immediately.

Usage:

```
process$new(
  command = NULL,
  args = character(),
  stdin = NULL,
  stdout = NULL,
  stderr = NULL,
  pty = FALSE,
```

```
  pty_options = list(),
  connections = list(),
  poll_connection = NULL,
  env = NULL,
  cleanup = TRUE,
  cleanup_tree = FALSE,
  cleanup_grace = 0,
  wd = NULL,
  echo_cmd = FALSE,
  supervise = FALSE,
  windows_verbatim_args = FALSE,
  windows_hide_window = FALSE,
  windows_detached_process = !cleanup,
  encoding = "",
  post_process = NULL,
  linux_pdeathsig = FALSE
)
```

Arguments:

`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. If you want to run `.bat` or `.cmd` files on Windows, make sure you read the 'Batch files' section above.

`args` Character vector, arguments to the command. They will be passed to the process as is, without a shell transforming them, 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 file name, use this file as standard input;
- `"|"`: create a (writeable) connection for stdin.
- `""` (empty string): inherit it from the main R process. If the main R process does not have a standard input stream, e.g. in RGui on Windows, then an error is thrown.

`stdout` What to do with the standard output. Possible values:

- `NULL`: discard it;
- A string starting with `">>"`, e.g. `">>output.txt"`: append it to this file. The file is created if it does not exist.
- A string (not starting with `">>"`), redirect it to this file, truncating the file first. Note that if you specify a relative path, it will be relative to the current working directory, even if you specify another directory in the `wd` argument. (See issue 324.)
- `"|"`: create a connection for it.
- `""` (empty string): inherit it from the main R process. If the main R process does not have a standard output stream, e.g. in RGui on Windows, then an error is thrown.

`stderr` What to do with the standard error. Possible values:

- `NULL`: discard it.
- A string starting with `">>"`, e.g. `">>error.txt"`: append it to this file. The file is created if it does not exist.
- A string (not starting with `">>"`), redirect it to this file, truncating the file first. Note that if you specify a relative path, it will be relative to the current working directory, even if you specify another directory in the `wd` argument. (See issue 324.)

- "|": 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.
- "" (empty string): inherit it from the main R process. If the main R process does not have a standard error stream, e.g. in RGui on Windows, then an error is thrown.

pty Whether to create a pseudo terminal (pty) for the background process. This is currently only supported on Unix systems, but not supported on Solaris. If it is TRUE, then the stdin, stdout and stderr arguments must be NULL. If a pseudo terminal is created, then processx will create pipes for standard input and standard output. There is no separate pipe for standard error, because there is no way to distinguish between stdout and stderr on a pty. Note that the standard output connection of the pty is *blocking*, so we always poll the standard output connection before reading from it using the $read_output() method. Also, because $read_output_lines() could still block if no complete line is available, this function always fails if the process has a pty. Use $read_output() to read from ptys.

pty_options Unix pseudo terminal options, a named list. see default_pty_options() for details and defaults.

connections A list of processx 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. To append new environment variables to the ones set in the current process, specify "current" in env, without a name, and the appended ones with names. The appended ones can overwrite the current ones.

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.

cleanup_grace Grace period between SIGTERM and SIGKILL. Only has an effect on Unix platforms. Set to 0 to terminate abruptly with SIGKILL only. Currently defaults to 0 until we implement a better approach on session quit.

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.

`windows_detached_process` Whether to use the `DETACHED_PROCESS` flag on Windows. If this is `TRUE`, then the child process will have no attached console, even if the parent had one.

`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. Use `"binary"` to disable text conversion entirely: `$read_output()` and `$read_error()` will return raw vectors instead of character strings, preserving all bytes including null bytes and non-UTF-8 byte sequences.

`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.

`linux_pdeathsig` On Linux, send this signal to the child process when the parent R process exits. `FALSE` (the default) disables this. `TRUE` sends `SIGTERM`. An integer signal number, e.g. `tools::SIGTERM` or `tools::SIGKILL`, sends that signal. Ignored on non-Linux platforms.

Returns: R6 object representing the process.

`process$kill()`**:** Terminate the process. It also terminate 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 terminated, and `FALSE` if it was not (because it was already finished/dead when `processx` tried to terminate it).

Usage:

```
process$kill(grace = 0, close_connections = TRUE)
```

Arguments:

`grace` Grace period in seconds. For `$kill()` on Unix, if it is positive, processx first sends `SIGTERM`, and sends `SIGKILL` if the process is still alive after `grace` seconds. If it is zero (the default), processx sends `SIGKILL` right away. Ignored on Windows and by `$kill_tree()`.

`close_connections` Whether to close standard input, standard output, standard error connections and the poll connection, after killing the process.

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

Usage:

```
process$kill_tree(grace = 0, close_connections = TRUE)
```

Arguments:

`grace` Grace period in seconds. For `$kill()` on Unix, if it is positive, processx first sends `SIGTERM`, and sends `SIGKILL` if the process is still alive after `grace` seconds. If it is zero (the default), processx sends `SIGKILL` right away. Ignored on Windows and by `$kill_tree()`.

`close_connections` Whether to close standard input, standard output, standard error connections and the poll connection, after killing the process.

`process$signal()`: Send 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 returns `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.

Usage:

```
process$signal(signal, group = TRUE)
```

Arguments:

`signal` An integer scalar, the id of the signal to send to the process. See `tools::pskill()` for the list of signals.

`group` Whether to send the signal to the whole process group of the child. The child is started in its own process group (via `setsid()` on Unix), so by default the signal is delivered to the child and any descendants that have not started a new group of their own. Set to `FALSE` to send the signal to the child process only. Ignored on Windows.

`process$interrupt()`: Send 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.

Usage:

```
process$interrupt(group = TRUE)
```

Arguments:

`group` Whether to send the interrupt to the whole process group of the child. See `$signal()` for details. Ignored on Windows, where the CTRL+BREAK event is always delivered to all processes attached to the child's console.

`process$get_pid()`: Query the process id.

Usage:

```
process$get_pid()
```

Returns: Integer scalar, the process id of the process.

`process$is_alive()`: Check if the process is alive.

Usage:

```
process$is_alive()
```

Returns: Logical scalar.

`process$wait()`: Wait until the process finishes, or a timeout happens. Note that if the process never finishes, and the timeout is infinite (the default), then R will never regain control. In some rare cases, `$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()`.

Usage:

```
process$wait(timeout = -1)
```

Arguments:

`timeout` Timeout in milliseconds, for the wait or the I/O polling.

Returns: It returns the process itself, invisibly.

`process$get_exit_status()`: `$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.

Usage:

```
process$get_exit_status()
```

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

Usage:

```
process$format()
```

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

Usage:

```
process$print()
```

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

Usage:

```
process$get_start_time()
```

`process$get_end_time()`: `$get_end_time()` returns the time when the process finished, or `NULL` if it is still running. On Unix the timestamp is recorded when R first notices the exit (via the `SIGCHLD` handler or a call to `$is_alive()`, `$get_exit_status()`, or `$wait()`), so it may be slightly later than the actual kernel exit time. On Windows the exact kernel exit time is used.

Usage:

```
process$get_end_time()
```

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

Usage:

```
process$is_supervised()
```

`process$supervise()`: `$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.

Usage:

```
process$supervise(status)
```

Arguments:

`status` Whether to turn on of off the supervisor for this process.

`process$read_output()`: `$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. When the process was started with `encoding = "binary"`, returns a raw vector instead of a character string.

A single call may return less data than requested (or an empty string) even when more output will eventually arrive: the OS pipe buffer is finite, and `$read_output()` only returns what is already buffered. Always call `$poll_io()` (or `poll()`) before reading to avoid deadlocking when the child fills the pipe buffer (see the *Polling* section for details). To read *all* output call `$read_all_output()`.

Usage:

```
process$read_output(n = -1)
```

Arguments:

`n` Number of characters or lines to read.

`process$read_error()`: `$read_error()` is similar to `$read_output()`, but reads from the standard error stream. Returns a raw vector when `encoding = "binary"` was used. The same polling requirement applies as for `$read_output()` (see the *Polling* section).

Usage:

```
process$read_error(n = -1)
```

Arguments:

`n` Number of characters or lines to read.

`process$read_output_bytes()`: `$read_output_bytes()` reads from the standard output connection of the process and returns the result as a raw vector, preserving all bytes including null bytes and other binary data. Switches the underlying connection to raw mode; do not mix with `$read_output()`. This will work only if `stdout="|"` was used.

Usage:

```
process$read_output_bytes(n = -1)
```

Arguments:

`n` Number of characters or lines to read.

`process$read_error_bytes()`: `$read_error_bytes()` is similar to `$read_output_bytes()`, but reads from the standard error stream.

Usage:

```
process$read_error_bytes(n = -1)
```

Arguments:

`n` Number of characters or lines to read.

`process$read_output_lines()`: `$read_output_lines()` reads lines from standard output connection of the process. If the standard output connection was not requested, 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.

Because `$read_output_lines()` only returns complete lines already in the buffer, it may return zero lines even when the process has produced output — for example when a line is longer than the

pipe buffer (~64KB on Linux/macOS, ~76KB on Windows) or when the line is not yet terminated. Always call `$poll_io()` before reading to avoid deadlocking (see the *Polling* section), and use `$read_output()` when lines may be very long.

Usage:

```
process$read_output_lines(n = -1)
```

Arguments:

`n` Number of characters or lines to read.

`process$read_error_lines()`: `$read_error_lines()` is similar to `$read_output_lines`, but it reads from the standard error stream. The same polling requirement applies (see the *Polling* section).

Usage:

```
process$read_error_lines(n = -1)
```

Arguments:

`n` Number of characters or lines to read.

`process$is_incomplete_output()`: `$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.

Usage:

```
process$is_incomplete_output()
```

`process$is_incomplete_error()`: `$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.

Usage:

```
process$is_incomplete_error()
```

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

Usage:

```
process$has_input_connection()
```

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

Usage:

```
process$has_output_connection()
```

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

Usage:

```
process$has_error_connection()
```

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

Usage:

```
process$has_poll_connection()
```

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

Usage:

```
process$get_input_connection()
```

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

Usage:

```
process$get_output_connection()
```

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

Usage:

```
process$get_error_connection()
```

`process$read_all_output()`: `$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 potentially several `readLines()` calls. It returns a character scalar. This will return content only if `stdout="|"` was used. Otherwise, it will throw an error.

Usage:

```
process$read_all_output()
```

`process$read_all_error()`: `$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 potentially several `readLines()` calls. It returns a character scalar. This will return content only if `stderr="|"` was used. Otherwise, it will throw an error.

Usage:

```
process$read_all_error()
```

`process$read_all_output_lines()`: `$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 potentially several `readLines()` calls. It returns a character vector. This will return content only if `stdout="|"` was used. Otherwise, it will throw an error.

Usage:

```
process$read_all_output_lines()
```

`process$read_all_error_lines()`: `$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 potentially several `readLines()` calls. It returns a character vector. This will return content only if `stderr="|"` was used. Otherwise, it will throw an error.

Usage:

```
process$read_all_error_lines()
```

`process$write_input()`: `$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).

Usage:

```
process$write_input(str, sep = "\n")
```

Arguments:

`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.

Returns: Leftover text (as a raw vector), that was not written.

`process$get_input_file()`: `$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.

Usage:

```
process$get_input_file()
```

`process$get_output_file()`: `$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.

Usage:

```
process$get_output_file()
```

`process$get_error_file()`: `$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.

Usage:

```
process$get_error_file()
```

`process$poll_io()`: `$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.

Usage:

```
process$poll_io(timeout)
```

Arguments:

`timeout` Timeout in milliseconds, for the wait or the I/O polling.

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

Usage:

```
process$get_poll_connection()
```

`process$get_result()`: `$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.

Usage:

```
process$get_result()
```

`process$as_ps_handle()`: `$as_ps_handle()` returns a ps::ps_handle object, corresponding to the process.

Usage:

```
process$as_ps_handle()
```

`process$get_name()`: Calls `ps::ps_name()` to get the process name.

Usage:

```
process$get_name()
```

`process$get_exe()`: Calls `ps::ps_exe()` to get the path of the executable.

Usage:

```
process$get_exe()
```

`process$get_cmdline()`: Calls `ps::ps_cmdline()` to get the command line.

Usage:

```
process$get_cmdline()
```

`process$get_status()`: Calls `ps::ps_status()` to get the process status.

Usage:

```
process$get_status()
```

`process$get_username()`: calls `ps::ps_username()` to get the username.

Usage:

```
process$get_username()
```

`process$get_wd()`: Calls `ps::ps_cwd()` to get the current working directory.

Usage:

```
process$get_wd()
```

`process$get_cpu_times()`: Calls `ps::ps_cpu_times()` to get CPU usage data.

Usage:

```
process$get_cpu_times()
```

`process$get_memory_info()`: Calls `ps::ps_memory_info()` to get memory data.

Usage:

```
process$get_memory_info()
```

`process$suspend()`: Calls `ps::ps_suspend()` to suspend the process.

Usage:

```
process$suspend()
```

`process$resume()`: Calls `ps::ps_resume()` to resume a suspended process.

Usage:

```
process$resume()
```

`process$clone()`: The objects of this class are cloneable with this method.

Usage:

```
process$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
p <- process$new("sleep", "2")
p$is_alive()
p
p$kill()
p$is_alive()

p <- process$new("sleep", "1")
p$is_alive()
Sys.sleep(2)
p$is_alive()
```

`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 = "|",
  stderr = "|",
```

```
  stdout_line_callback = NULL,
  stdout_callback = NULL,
  stderr_line_callback = NULL,
  stderr_callback = NULL,
  stderr_to_stdout = FALSE,
  stdin = NULL,
  env = NULL,
  windows_verbatim_args = FALSE,
  windows_hide_window = FALSE,
  encoding = "",
  cleanup_tree = FALSE,
  cleanup_grace = 0.1,
  pty = FALSE,
  pty_options = list(),
  ...
)
```

Arguments

command	Character scalar, the command to run. If you are running `.bat` or `.cmd` files on Windows, make sure you read the 'Batch files' section in the process manual page.
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 classes are `system_command_status_error` and `system_command_timeout_error`, respectively, and both errors have class `system_command_error` as well. See also "Error conditions" below.
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. If the standard output and/or error is redirected to a file or they are ignored, then they also not echoed.
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	What to do with the standard output. By default it is collected in the result, and you can also use the `stdout_line_callback` and `stdout_callback` arguments to pass callbacks for output. If it is the empty string (`""`), then the child process inherits the standard output stream of the R process. (If the main R process does not have a standard output stream, e.g. in RGui on Windows, then an error is thrown.) If it is `NULL`, then standard output is discarded. If it is a string other than `"|"` and `""`, then it is taken as a file name and the output is redirected to this file.

stderr What to do with the standard error. By default it is collected in the result, and you can also use the stderr_line_callback and stderr_callback arguments to pass callbacks for output. If it is the empty string (""), then the child process inherits the standard error stream of the R process. (If the main R process does not have a standard error stream, e.g. in RGui on Windows, then an error is thrown.) If it is NULL, then standard error is discarded. If it is a string other than "|" and "", then it is taken as a file name and the standard error is redirected to this file.

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.

stdin What to do with the standard input. By default it is ignored (NULL). It can be a file name, to redirect the contents of a file to the standard input. When pty = TRUE, stdin can only be NULL (no input) or a file path (whose contents are fed to the process via the PTY).

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. To append new environment variables to the ones set in the current process, specify "current" in env, without a name, and the appended ones with names. The appended ones can overwrite the current ones.

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. Use "binary" to collect the raw bytes without any

	conversion: `stdout` and `stderr` in the return value will be raw vectors instead of character strings. Line callbacks are not supported in binary mode.
`cleanup_tree`	Whether to clean up the child process tree after the process has finished.
`cleanup_grace`	Passed to `kill()` or `kill_tree()` on cleanup.
`pty`	Whether to use a pseudo-terminal (PTY) for the process. Supported on Unix and on Windows 10 version 1809 or later (via ConPTY). When `TRUE`, stdout and stderr are merged into a single stream (accessible via `$stdout` in the result), and `$stderr` is always `NULL`. The process sees a real terminal, so programs that disable colour or interactive features when not attached to a terminal will behave as if they are. `stdout` and `stderr` must be left at their defaults (`"|"`), and `stderr_to_stdout`, `stderr_callback`, and `stderr_line_callback` must not be set.
`pty_options`	Options for the PTY, a named list. See `default_pty_options()` for the available options and their defaults.
`...`	Extra arguments are passed to `process$new()`, see process. Note that you cannot pass `stout` or `stderr` here, because they are used internally by `run()`. You can use the `stdout_callback`, `stderr_callback`, etc. arguments to manage the standard output and error, or the process class directly if you need more flexibility.

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.

Error conditions

`run()` throws error condition objects if the process is interrupted, timeouts or fails (if `error_on_status` is `TRUE`):

- On interrupt, a condition with classes `system_command_interrupt`, `interrupt`, `condition` is signalled. This can be caught with `tryCatch(..., interrupt = ...)`.
- On timeout, a condition with classes `system_command_timeout_error`, `system_command_error`, `error`, `condition` is thrown.
- On error (if `error_on_status` is `TRUE`), an error with classes `system_command_status_error`, `system_command_error`, `error`, `condition` is thrown.

All of these conditions have the fields:

- `message`: the error message,
- `stderr`: the standard error of the process, or the standard output of the process if `stderr_to_stdout` was `TRUE`.
- `call`: the captured call to `run()`.
- `echo`: the value of the `echo` argument.
- `stderr_to_stdout`: the value of the `stderr_to_stdout` argument.
- `status`: the exit status for `system_command_status_error` errors.

Examples

```
# This works on Unix systems
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
  )
)


# This works on Windows systems, if the ping command is available
run("ping", c("-n", "1", "127.0.0.1"))
run("ping", c("-n", "6", "127.0.0.1"), timeout = 1,
    error_on_status = FALSE)
```

Index

∗ **processx connections**
    conn_create_fd, 3

base64_decode, 2
base64_encode *(*base64_decode*)*, 2
base::normalizePath(), *20*
base::shQuote(), *20*
base::system(), *30*
base::system2(), *30*
base::tempdir(), *6*, *8*

close.processx_connection
        *(*conn_create_fd*)*, 3
conn_accept_unix_socket
        *(*conn_create_unix_socket*)*, 8
conn_connect_fifo *(*conn_create_fifo*)*, 6
conn_connect_unix_socket
        *(*conn_create_unix_socket*)*, 8
conn_create_fd, 3
conn_create_fifo, 6
conn_create_file *(*conn_create_fd*)*, 3
conn_create_pipepair *(*conn_create_fd*)*, 3
conn_create_proc_pipepair
        *(*conn_create_fd*)*, 3
conn_create_unix_socket, 8
conn_create_unix_socket(), *7*
conn_disable_inheritance
        *(*conn_create_fd*)*, 3
conn_file_name *(*conn_create_fd*)*, 3
conn_file_name(), *7*
conn_get_fileno *(*conn_create_fd*)*, 3
conn_is_incomplete *(*conn_create_fd*)*, 3
conn_is_incomplete(), *7*
conn_read_bytes *(*conn_create_fd*)*, 3
conn_read_chars *(*conn_create_fd*)*, 3
conn_read_chars(), *7*
conn_read_lines *(*conn_create_fd*)*, 3
conn_read_lines(), *7*
conn_set_stderr *(*conn_create_fd*)*, 3
conn_set_stdout *(*conn_create_fd*)*, 3
conn_unix_socket_state
        *(*conn_create_unix_socket*)*, 8
conn_write *(*conn_create_fd*)*, 3
conn_write(), *7*
curl::multi_fdset(), *9*
curl_fds, 9

default_pty_options, 10
default_pty_options(), *21*, *33*

is_valid_fd *(*conn_create_fd*)*, 3

pipeline, 10
poll, 15
poll(), *6*, *7*, *9*, *17*, *21*, *25*, *28*
process, *11*, *13*, *14*, 16, *30*, *31*, *33*
processx_conn_close *(*conn_create_fd*)*, 3
processx_conn_is_incomplete
        *(*conn_create_fd*)*, 3
processx_conn_read_bytes
        *(*conn_create_fd*)*, 3
processx_conn_read_chars
        *(*conn_create_fd*)*, 3
processx_conn_read_lines
        *(*conn_create_fd*)*, 3
processx_conn_write *(*conn_create_fd*)*, 3
ps::ps_cmdline(), *29*
ps::ps_cpu_times(), *29*
ps::ps_cwd(), *29*
ps::ps_exe(), *29*
ps::ps_handle, *29*
ps::ps_memory_info(), *29*
ps::ps_name(), *29*
ps::ps_resume(), *30*
ps::ps_status(), *29*
ps::ps_suspend(), *29*
ps::ps_username(), *29*

run, 30

tools::pskill(), *23*