Package 'processx'

July 30, 2017

Title Execute and Control System Processes

Version 2.0.0.1

Author Gábor Csárdi

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

Description Portable 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 and their children; restart 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-pkgs/processx#readme

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

RoxygenNote 6.0.1.9000

Suggests covr, testthat, withr

Imports assertthat, crayon, debugme, R6, utils

Encoding UTF-8

NeedsCompilation yes

Repository CRAN

Date/Publication 2017-07-30 06:34:51 UTC

R topics documented:

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

Description

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

Usage

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

Arguments

`processes`	A list of `process` objects to wait on. If this is a named list, then the returned list will have the same names. This simplifies the identification of the processes. If an empty list, then the
`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 two. There is one list element for each process, in the same order as in the input list. The character vectors' elements are named `output` and `error` and their possible values are: `nopipe`, `ready`, `timeout`, `closed`, `silent`. See details about these below.

Explanation of the return values

- `nopipe` means that the stdout or stderr from this process was not captured.
- `ready` means that stdout or stderr from this process are ready to read from. Note that end-of-file on these outputs also triggers `ready`.
- timeout‘: the 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.

Known issues

You cannot wait on the termination of a process directly. It is only signalled through the closed stdout and stderr pipes. This means that if both stdout and stderr are ignored or closed for a process, then you will not be notified when it exits.

Examples

```
## Different commands to run for windows and unix
## Not run:
cmd1 <- switch(
  .Platform$OS.type,
  "unix" = "sleep 1; ls",
  "ping -n 2 127.0.0.1 && dir /b"
)
cmd2 <- switch(
  .Platform$OS.type,
  "unix" = "sleep 2; ls 1>&2",
  "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$new(commandline = cmd1, stdout = "|")
p2 <- process$new(commandline = cmd2, 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, commandline = NULL,
                 stdout = TRUE, stderr = TRUE, cleanup = TRUE,
```

```
                  echo_cmd = FALSE, windows_verbatim_args = FALSE,
                  windows_hide_window = FALSE)

p$is_alive()
p$signal(signal)
p$kill(grace = 0.1)
p$wait(timeout = -1)
p$get_pid()
p$get_exit_status()
p$restart()
p$get_start_time()

p$read_output_lines(...)
p$read_error_lines(...)
p$get_output_connection()
p$get_error_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$poll_io(timeout)

print(p)
```

Arguments

p A `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 `shQuote`. See `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.

commandline A character scalar, a full command line. On Unix systems it runs the a shell: `sh -c <commandline>`. On Windows it uses the `cmd` shell: `cmd /c <commandline>`. If you want more control, then call your chosen shell directly.

stdout What to do with the standard output. Possible values: `FALSE`: discard it; a string, redirect it to this file, `TRUE`: redirect it to a temporary file, `"|"`: create an R connection for it.

stderr What to do with the standard error. Possible values: `FALSE`: discard it; a string, redirect it to this file, `TRUE`: redirect it to a temporary file, `"|"`: create an R connection for it.

cleanup Whether to kill the process (and its children) if the `process` object is garbage collected.

echo_cmd Whether to print the command to the screen before running it.

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 `pskill` for the list of signals.

grace Currently not used.

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

... Extra arguments are passed to the `readLines` function.

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.

`$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).

`$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.

`$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.

`$restart()` restarts a process. It returns the process itself.

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

`$read_output_lines()` reads 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.

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

`$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.

`$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.

`$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.

`$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.

`$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.

`$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.

`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).

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$restart()
p$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(), commandline = NULL,
  error_on_status = TRUE, echo_cmd = FALSE, echo = FALSE,
  spinner = FALSE, timeout = Inf, stdout_line_callback = NULL,
  stdout_callback = NULL, stderr_line_callback = NULL,
  stderr_callback = NULL, windows_verbatim_args = FALSE,
  windows_hide_window = FALSE)
```

Arguments

command	Character scalar, the command to run. It will be escaped via base::shQuote.
args	Character vector, arguments to the command. They will be escaped via base::shQuote.
commandline	A character scalar, a full command line. No escaping will be performed on it.
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.
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 reassusing 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.
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.

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.

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(commandline = "sleep 10", timeout = 1,
    error_on_status = FALSE))
  system.time(
    run(
      commandline = "for i in 1 2 3 4 5; do echo $i; sleep 1; done",
      timeout=2, error_on_status = FALSE
    )
  )
} else {
  run(commandline = "ping -n 1 127.0.0.1")
  run(commandline = "ping -n 6 127.0.0.1", timeout = 1,
    error_on_status = FALSE)
}

## End(Not run)
```

Index