Package 'subprocess'

August 13, 2018

Type Package

Title Manage Sub-Processes in R

Version 0.8.3

Description Create and handle multiple sub-processes in R, exchange data over standard input and output streams, control their life cycle.

License MIT + file LICENSE

URL https://github.com/lbartnik/subprocess

BugReports https://github.com/lbartnik/subprocess/issues

Depends R (>= 3.2.0)

Suggests mockery, testthat, knitr, rmarkdown (>= 1.0)

Collate 'package.R' 'readwrite.R' 'signals.R' 'subprocess.R' 'tests.R' 'utils.R'

RoxygenNote 6.0.1

VignetteBuilder knitr

SystemRequirements C++11

NeedsCompilation yes

Author Lukasz Bartnik [aut, cre]

Maintainer Lukasz Bartnik <l.bartnik@gmail.com>

Repository CRAN

Date/Publication 2018-08-13 05:30:04 UTC

R topics documented:

`C_tests_utf8` *Run UTF8 tests implemented in C.*

Description

If there is no error in those tests a simple string message is returned. If there is at least one error, another message is returned.

Usage

```
C_tests_utf8()
```

Value

A string `"All C tests passed!"` if there are no errors.

`process_exists` *Check if process with a given id exists.*

Description

Check if process with a given id exists.

Usage

```
process_exists(x)
```

Arguments

`x`	A process handle returned by spawn_process or a OS-level process id.

Value

`TRUE` if process exists, `FALSE` otherwise.

readwrite *Communicating with a Child Process*

Description

`process_read()` reads data from one of the child process' streams, *standard output* or *standard error output*, and returns it as a `character` vector.

`process_write()` writes data into child's *standard input* stream.

`process_close_input()` closes the *write* end of the pipe whose *read* end is the standard input stream of the child process. This is a standard way to gracefully request the child process to exit.

`PIPE_STDOUT`: read from child's standard output.

`PIPE_STDERR`: read from child's standard error output.

`PIPE_BOTH`: read from both child's output streams: standard output and standard error output.

Usage

```
process_read(handle, pipe = PIPE_BOTH, timeout = TIMEOUT_IMMEDIATE,
  flush = TRUE)

process_write(handle, message)

process_close_input(handle)

PIPE_STDOUT

PIPE_STDERR

PIPE_BOTH
```

Arguments

`handle`	Process handle obtained from `spawn_process`.
`pipe`	Output stream identifier: `PIPE_STDOUT`, `PIPE_STDERR` or `PIPE_BOTH`.
`timeout`	Optional timeout in milliseconds.
`flush`	If there is any data within the given `timeout` try again with `timeout=0` until C buffer is empty.
`message`	Input for the child process.

Format

`PIPE_STDOUT`, `PIPE_STDERR` and `PIPE_BOTH` are single `character` values.

Details

If `flush=TRUE` in `process_read()` then the invocation of the underlying `read()` *system-call* will be repeated until the pipe buffer is empty.

If `pipe` is set to either `PIPE_STDOUT` or `PIPE_STDERR`, the returned value is a single list with a single key, `stdout` or `stderr`, respectively. If `pipe` is set to `PIPE_BOTH` the returned `list` contains both keys. Values in the list are `character` vectors of 0 or more elements, lines read from the respective output stream of the child process.

For details on `timeout` see terminating.

Value

`process_read` returns a `list` which contains either of or both keys: *stdout* and *stderr*; the value is in both cases a `character` vector which contains lines of child's output.

`process_write` returns the number of characters written.

`signal` *A helper function used in vignette.*

Description

A helper function used in vignette.

Usage

```
signal(signal, handler)
```

Arguments

`signal`	Signal number.
`handler`	Either `"default"` or `"ignore"`.

`signals` *Sending signals to the child process.*

Description

Sending signals to the child process.

Operating-System-level signals that can be sent via process_send_signal are defined in the 'subprocess::signals" list. It is a list that is generated when the package is loaded and it contains only signals supported by the current platform (Windows or Linux).

All signals, both supported and not supported by the current platform, are also exported under their names. If a given signal is not supported on the current platform, then its value is set to `NA`.

Calling process_kill() and process_terminate() invokes the appropriate OS routine (waitpid() or WaitForSingleObject(), closing the process handle, etc.) that effectively lets the operating system clean up after the child process. Calling process_send_signal() is not accompanied by such clean-up and if the child process exits it needs to be followed by a call to process_wait().

process_terminate() on Linux sends the SIGTERM signal to the process pointed to by handle. On Windows it calls TerminateProcess().

process_kill() on Linux sends the SIGKILL signal to handle. On Windows it is an alias for process_terminate().

process_send_signal() sends an OS-level signal to handle. In Linux all standard signal numbers are supported. On Windows supported signals are SIGTERM, CTRL_C_EVENT and CTRL_BREAK_EVENT. Those values will be available via the signals list which is also attached in the package namespace.

Usage

```
signals

process_terminate(handle)

process_kill(handle)

process_send_signal(handle, signal)

SIGABRT

SIGALRM

SIGCHLD

SIGCONT

SIGFPE

SIGHUP

SIGILL

SIGINT

SIGKILL

SIGPIPE

SIGQUIT

SIGSEGV

SIGSTOP
```

SIGTERM

SIGTSTP

SIGTTIN

SIGTTOU

SIGUSR1

SIGUSR2

CTRL_C_EVENT

CTRL_BREAK_EVENT

Arguments

handle	Process handle obtained from spawn_process().
signal	Signal number, one of names(signals).

Format

An object of class list.

Details

In Windows, signals are delivered either only to the child process or to the child process and all its descendants. This behavior is controlled by the termination_mode argument of the subprocess::spawn_process() function. Setting it to TERMINATION_GROUP results in signals being delivered to the child and its descendants.

See Also

spawn_process()

Examples

```
## Not run:
# send the SIGKILL signal to bash
h <- spawn_process('bash')
process_signal(h, signals$SIGKILL)
process_signal(h, SIGKILL)

# is SIGABRT supported on the current platform?
is.na(SIGABRT)

## End(Not run)

## Not run:
```

```
# Windows
process_send_signal(h, SIGTERM)
process_send_signal(h, CTRL_C_EVENT)
process_send_signal(h, CTRL_BREAK_EVENT)

## End(Not run)
```

---

spawn_process *Start a new child process.*

Description

In Linux, the usual combination of `fork()` and `exec()` is used to spawn a new child process. Standard streams are redirected over regular unnamed `pipe`s.

In Windows a new process is spawned with `CreateProcess()` and streams are redirected over unnamed pipes obtained with `CreatePipe()`. However, because non-blocking (*overlapped* in Windows-speak) read/write is not supported for unnamed pipes, two reader threads are created for each new child process. These threads never touch memory allocated by R and thus they will not interfere with R interpreter's memory management (garbage collection).

`is_process_handle()` verifies that an object is a valid *process handle* as returned by `spawn_process()`.

`TERMINATION_GROUP`: `process_terminate(handle)` and `process_kill(handle)` deliver the signal to the child process pointed to by `handle` and all of its descendants.

`TERMINATION_CHILD_ONLY`: `process_terminate(handle)` and `process_kill(handle)` deliver the signal only to the child process pointed to by `handle` but to none of its descendants.

Usage

```
spawn_process(command, arguments = character(), environment = character(),
  workdir = "", termination_mode = TERMINATION_GROUP)

## S3 method for class 'process_handle'
print(x, ...)

is_process_handle(x)

TERMINATION_GROUP

TERMINATION_CHILD_ONLY
```

Arguments

`command`	Path to the executable.
`arguments`	Optional arguments for the program.
`environment`	Optional environment.
`workdir`	Optional new working directory.

termination_mode
: Either TERMINATION_GROUP or TERMINATION_CHILD_ONLY.

x
: Object to be printed or tested.

...
: Other parameters passed to the print method.

Format

TERMINATION_GROUP and TERMINATION_CHILD_ONLY are single character values.

Details

command is always prepended to arguments so that the child process can correcty recognize the name of its executable via its argv vector. This is done automatically by spawn_process.

environment can be passed as a character vector whose elements take the form "NAME=VALUE", a named character vector or a named list.

workdir is the path to the directory where the new process is ought to be started. NULL and "" mean that working directory is inherited from the parent.

Value

spawn_process() returns an object of the *process handle* class.

Termination

The termination_mode specifies what should happen when process_terminate() or process_kill() is called on a subprocess. If it is set to TERMINATION_GROUP, then the termination signal is sent to the parent and all its descendants (sub-processes). If termination mode is set to TERMINATION_CHILD_ONLY, only the child process spawned directly from the R session receives the signal.

In Windows this is implemented with the job API, namely CreateJobObject(), AssignProcessToJobObject() and TerminateJobObject(). In Linux, the child calls setsid() after fork() but before execve(), and kill() is called with the negate process id.

---

subprocess *Manage Subprocesses in R*

Description

Cross-platform child process management modelled after Python's subprocess module.

Details

This R package extends R's capabilities of starting and handling child processes. It brings the capability of alternating read from and write to a child process, communicating via signals, terminating it and handling its exit status (return code).

With R's standard base::system and base::system2 functions one can start a new process and capture its output but cannot directly write to its standard input. Another tool, the parallel::mclapply function, is aimed at replicating the current session and is limited to operating systems that come with the fork() system call.

References

http://github.com/lbartnik/subprocess

http://docs.python.org/3/library/subprocess.html

`terminating` *Terminating a Child Process.*

Description

These functions give access to the state of the child process and to its exit status (return code).

The `timeout` parameter can take one of three values:

- `0` which means no timeout
- `-1` which means "wait until there is data to read"
- a positive integer, which is the actual timeout in milliseconds

`TIMEOUT_INFINITE` denotes an "infinite" timeout (that is, wait until response is available) when waiting for an operation to complete.

`TIMEOUT_IMMEDIATE` denotes an "immediate" timeout (in other words, no timeout) when waiting for an operation to complete.

Usage

```
process_wait(handle, timeout = TIMEOUT_INFINITE)

process_state(handle)

process_return_code(handle)

TIMEOUT_INFINITE

TIMEOUT_IMMEDIATE
```

Arguments

`handle`	Process handle obtained from `spawn_process`.
`timeout`	Optional timeout in milliseconds.

Format

An object of class `integer` of length 1.

Details

`process_wait()` checks the state of the child process by invoking the system call `waitpid()` or `WaitForSingleObject()`.

`process_state()` refreshes the handle by calling `process_wait()` with no timeout and returns one of these values: `"not-started"`. `"running"`, `"exited"`, `"terminated"`.

`process_return_code()` gives access to the value returned also by `process_wait()`. It does not invoke `process_wait()` behind the scenes.

Value

`process_wait()` returns an `integer` exit code of the child process or `NA` if the child process has not exited yet. The same value can be accessed by `process_return_code()`.

See Also

`spawn_process()`, `process_read()` `signals()`

Index