Package: later (via r-universe)

November 5, 2026

Type Package

Title Utilities for Scheduling Functions to Execute Later with Event Loops

Version 1.4.8.9000

Description Executes arbitrary R or C functions some time after the current time, after the R execution stack has emptied. The functions are scheduled in an event loop.

License MIT + file LICENSE

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

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

Depends R (>= 3.5)

Imports Rcpp (>= 1.0.10), rlang

Suggests knitr, nanonext, promises, rmarkdown, testthat (>= 3.0.0)

LinkingTo Rcpp

VignetteBuilder knitr

Config/build/compilation-database true

Config/Needs/website tidyverse/tidytemplate

Config/roxygen2/markdown TRUE

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Config/usethis/last-upkeep 2025-07-18

Encoding UTF-8

Repository https://posit-dev-shinycoreci.r-universe.dev

Date/Publication 2026-08-07 16:24:50 UTC

RemoteUrl https://github.com/r-lib/later

RemoteRef HEAD

RemoteSha f1d988deb5e9fd26b3af5adee6da454ae7386f9d

Contents

create_loop 2
later 3
later_fd 4
next_op_secs 6
run_now 7

Index **8**

`create_loop` *Private event loops*

Description

Normally, later uses a global event loop for scheduling and running functions. However, in some cases, it is useful to create a *private* event loop to schedule and execute tasks without disturbing the global event loop. For example, you might have asynchronous code that queries a remote data source, but want to wait for a full back-and-forth communication to complete before continuing in your code – from the caller's perspective, it should behave like synchronous code, and not do anything with the global event loop (which could run code unrelated to your operation). To do this, you would run your asynchronous code using a private event loop.

Usage

```
create_loop(parent = current_loop())

destroy_loop(loop)

exists_loop(loop)

current_loop()

with_temp_loop(expr)

with_loop(loop, expr)

global_loop()
```

Arguments

`parent`	The parent event loop for the one being created. Whenever the parent loop runs, this loop will also automatically run, without having to manually call `run_now()` on this loop. If `NULL`, then this loop will not have a parent event loop that automatically runs it; the only way to run this loop will be by calling `run_now()` on this loop.
`loop`	A handle to an event loop.
`expr`	An expression to evaluate.

Details

`create_loop` creates and returns a handle to a private event loop, which is useful when for scheduling tasks when you do not want to interfere with the global event loop.

`destroy_loop` destroys a private event loop.

`exists_loop` reports whether an event loop exists – that is, that it has not been destroyed.

`current_loop` returns the currently-active event loop. Any calls to `later()` or `run_now()` will use the current loop by default.

`with_loop` evaluates an expression with a given event loop as the currently-active loop.

`with_temp_loop` creates an event loop, makes it the current loop, then evaluates the given expression. Afterwards, the new event loop is destroyed.

`global_loop` returns a handle to the global event loop.

`later` *Executes a function later*

Description

Schedule an R function or formula to run after a specified period of time. Similar to JavaScript's `setTimeout` function. Like JavaScript, R is single-threaded so there's no guarantee that the operation will run exactly at the requested time, only that at least that much time will elapse.

Usage

```
later(func, delay = 0, loop = current_loop())
```

Arguments

`func`	A function or formula (see `rlang::as_function()`).
`delay`	Number of seconds in the future to delay execution. There is no guarantee that the function will be executed at the desired time, but it should not execute earlier.
`loop`	A handle to an event loop. Defaults to the currently-active loop.

Details

The mechanism used by this package is inspired by Simon Urbanek's background package and similar code in Rhttpd.

Value

A function, which, if invoked, will cancel the callback. The function will return `TRUE` if the callback was successfully cancelled and `FALSE` if not (this occurs if the callback has executed or has been cancelled already).

Note

To avoid bugs due to reentrancy, by default, scheduled operations only run when there is no other R code present on the execution stack; i.e., when R is sitting at the top-level prompt. You can force past-due operations to run at a time of your choosing by calling `run_now()`.

Error handling is not particularly well-defined and may change in the future. options(error=browser) should work and errors in `func` should generally not crash the R process, but not much else can be said about it at this point. If you must have specific behavior occur in the face of errors, put error handling logic inside of `func`.

Examples

```
# Example of formula style
later(~cat("Hello from the past\n"), 3)

# Example of function style
later(function() {
  print(summary(cars))
}, 2)
```

`later_fd` *Executes a function when a file descriptor is ready*

Description

Schedule an R function or formula to run after an indeterminate amount of time when file descriptors are ready for reading or writing, subject to an optional timeout.

Usage

```
later_fd(
  func,
  readfds = integer(),
  writefds = integer(),
  exceptfds = integer(),
  timeout = Inf,
  loop = current_loop()
)
```

Arguments

`func`	A function that takes a single argument, a logical vector that indicates which file descriptors are ready (a concatenation of `readfds`, `writefds` and `exceptfds`). This may be all `FALSE` if the `timeout` argument is non-`Inf`. File descriptors with error conditions pending are represented as `NA`, as are invalid file descriptors such as those already closed.

readfds	Integer vector of file descriptors, or Windows SOCKETs, to monitor for being ready to read.
writefds	Integer vector of file descriptors, or Windows SOCKETs, to monitor being ready to write.
exceptfds	Integer vector of file descriptors, or Windows SOCKETs, to monitor for error conditions pending.
timeout	Number of seconds to wait before giving up, and calling func with all FALSE. The default Inf implies waiting indefinitely. Specifying 0 will check once without blocking, and supplying a negative value defaults to a timeout of 1s.
loop	A handle to an event loop. Defaults to the currently-active loop.

Details

On the occasion the system-level poll (on Windows WSAPoll) returns an error, the callback will be made on a vector of all NAs. This is indistinguishable from a case where the poll succeeds but there are error conditions pending against each file descriptor.

If no file descriptors are supplied, the callback is scheduled for immediate execution and made on the empty logical vector logical(0).

Value

A function, which, if invoked, will cancel the callback. The function will return TRUE if the callback was successfully cancelled and FALSE if not (this occurs if the callback has executed or has been cancelled already).

Note

To avoid bugs due to reentrancy, by default, scheduled operations only run when there is no other R code present on the execution stack; i.e., when R is sitting at the top-level prompt. You can force past-due operations to run at a time of your choosing by calling run_now().

Error handling is not particularly well-defined and may change in the future. options(error=browser) should work and errors in func should generally not crash the R process, but not much else can be said about it at this point. If you must have specific behavior occur in the face of errors, put error handling logic inside of func.

Examples

```
# create nanonext sockets
s1 <- nanonext::socket(listen = "inproc://nano")
s2 <- nanonext::socket(dial = "inproc://nano")
fd1 <- nanonext::opt(s1, "recv-fd")
fd2 <- nanonext::opt(s2, "recv-fd")

# 1. timeout: prints FALSE, FALSE
later_fd(print, c(fd1, fd2), timeout = 0.1)
Sys.sleep(0.2)
run_now()

# 2. fd1 ready: prints TRUE, FALSE
```

```
later_fd(print, c(fd1, fd2), timeout = 1)
res <- nanonext::send(s2, "msg")
Sys.sleep(0.1)
run_now()

# 3. both ready: prints TRUE, TRUE
res <- nanonext::send(s1, "msg")
later_fd(print, c(fd1, fd2), timeout = 1)
Sys.sleep(0.1)
run_now()

# 4. fd2 ready: prints FALSE, TRUE
res <- nanonext::recv(s1)
later_fd(print, c(fd1, fd2), timeout = 1)
Sys.sleep(0.1)
run_now()

# 5. fds invalid: prints NA, NA
close(s2)
close(s1)
later_fd(print, c(fd1, fd2), timeout = 0)
Sys.sleep(0.1)
run_now()
```

`next_op_secs` *Relative time to next scheduled operation*

Description

Returns the duration between now and the earliest operation that is currently scheduled, in seconds. If the operation is in the past, the value will be negative. If no operation is currently scheduled, the value will be `Inf`.

Usage

```
next_op_secs(loop = current_loop())
```

Arguments

`loop`	A handle to an event loop.

run_now *Execute scheduled operations*

Description

Normally, operations scheduled with `later()` will not execute unless/until no other R code is on the stack (i.e. at the top-level). If you need to run blocking R code for a long time and want to allow scheduled operations to run at well-defined points of your own operation, you can call `run_now()` at those points and any operations that are due to run will do so.

Usage

```
run_now(timeoutSecs = 0L, all = TRUE, loop = current_loop())
```

Arguments

`timeoutSecs`	Wait (block) for up to this number of seconds waiting for an operation to be ready to run. If `0`, then return immediately if there are no operations that are ready to run. If `Inf` or negative, then wait as long as it takes (if none are scheduled, then this will block forever).
`all`	If `FALSE`, `run_now()` will execute at most one scheduled operation (instead of all eligible operations). This can be useful in cases where you want to interleave scheduled operations with your own logic.
`loop`	A handle to an event loop. Defaults to the currently-active loop.

Details

If one of the callbacks throws an error, the error will *not* be caught, and subsequent callbacks will not be executed (until `run_now()` is called again, or control returns to the R prompt). You must use your own tryCatch if you want to handle errors.

Value

A logical indicating whether any callbacks were actually run.

Index

create_loop, 2
current_loop *(*create_loop*)*, 2

destroy_loop *(*create_loop*)*, 2

exists_loop *(*create_loop*)*, 2

global_loop *(*create_loop*)*, 2

later, 3, *3*
later(), *7*
later_fd, 4

next_op_secs, 6

rlang::as_function(), *3*
run_now, *2*, *3*, 7
run_now(), *4*, *5*

tryCatch, *7*

with_loop *(*create_loop*)*, 2
with_temp_loop *(*create_loop*)*, 2