Package 'ps'

July 23, 2025

Title List, Query, Manipulate System Processes

Version 1.9.1

Description List, query and manipulate all system processes, on 'Windows', 'Linux' and 'macOS'.

License MIT + file LICENSE

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

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

Depends R (>= 3.4)

Imports utils

Suggests callr, covr, curl, pillar, pingr, processx (>= 3.1.0), R6, rlang, testthat (>= 3.0.0), webfakes, withr

Biarch true

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation yes

Author Jay Loden [aut],
Dave Daeschler [aut],
Giampaolo Rodola' [aut],
Gábor Csárdi [aut, cre],
Posit Software, PBC [cph, fnd]

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

Repository CRAN

Date/Publication 2025-04-12 09:50:01 UTC

Contents

`CleanupReporter` *testthat reporter that checks if child processes are cleaned up in tests*

Description

`CleanupReporter` takes an existing testthat `Reporter` object, and wraps it, so it checks for leftover child processes, at the specified place, see the `proc_unit` argument below.

Usage

```
CleanupReporter(reporter = testthat::ProgressReporter)
```

Arguments

`reporter`	A testthat reporter to wrap into a new `CleanupReporter` class.

Details

Child processes can be reported via a failed expectation, cleaned up silently, or cleaned up and reported (the default).

If a `test_that()` block has an error, `CLeanupReporter` does not emit any expectations at the end of that block. The error will lead to a test failure anyway. It will still perform the cleanup, if requested, however.

The constructor of the `CleanupReporter` class has options:

- `file`: the output file, if any, this is passed to `reporter`.
- `proc_unit`: when to perform the child process check and cleanup. Possible values:
  - `"test"`: at the end of each `testthat::test_that()` block (the default),
  - `"testsuite"`: at the end of the test suite.
- `proc_cleanup`: Logical scalar, whether to kill the leftover processes, `TRUE` by default.
- `proc_fail`: Whether to create an expectation, that fails if there are any processes alive, `TRUE` by default.
- `proc_timeout`: How long to wait for the processes to quit. This is sometimes needed, because even if some kill signals were sent to child processes, it might take a short time for these to take effect. It defaults to one second.

- `rconn_unit`: When to perform the R connection cleanup. Possible values are `"test"` and `"testsuite"`, like for `proc_unit`.
- `rconn_cleanup`: Logical scalar, whether to clean up leftover R connections. `TRUE` by default.
- `rconn_fail`: Whether to fail for leftover R connections. `TRUE` by default.
- `file_unit`: When to check for open files. Possible values are `"test"` and `"testsuite"`, like for `proc_unit`.
- `file_fail`: Whether to fail for leftover open files. `TRUE` by default.
- `conn_unit`: When to check for open network connections. Possible values are `"test"` and `"testsuite"`, like for `proc_unit`.
- `conn_fail`: Whether to fail for leftover network connections. `TRUE` by default.

Value

New reporter class that behaves exactly like `reporter`, but it checks for, and optionally cleans up child processes, at the specified granularity.

Examples

This is how to use this reporter in `testthat.R`:

```
library(testthat)
library(mypackage)

if  (ps::ps_is_supported()) {
  reporter <- ps::CleanupReporter(testthat::ProgressReporter)$new(
    proc_unit = "test", proc_cleanup = TRUE)
} else {
  ## ps does not support this platform
  reporter <- "progress"
}

test_check("mypackage", reporter = reporter)
```

Note

Some IDEs, like RStudio, start child processes frequently, and sometimes crash when these are killed, only use this reporter in a terminal session. In particular, you can always use it in the idiomatic `testthat.R` file, that calls `test_check()` during `R CMD check`.

`errno` *List of 'errno' error codes*

Description

For the errors that are not used on the current platform, `value` is `NA_integer_`.

Usage

```
errno()
```

Details

A data frame with columns: `name`, `value`, `description`.

Examples

```
errno()
```

`ps` *Process table*

Description

Data frame with the currently running processes.

Usage

```
ps(user = NULL, after = NULL, columns = NULL)
```

Arguments

`user`	Username, to filter the results to matching processes.
`after`	Start time (`POSIXt`), to filter the results to processes that started after this.
`columns`	Columns to include in the result. If `NULL` (the default), then a default set of columns are returned, see below. The columns are shown in the same order they are specified in `columns`, but each column is included at most once. Use `"*"` to include all possible columns, and prefix a column name with `-` to remove it.

Details

Columns shown by default, if columns is not given or NULL:

- pid: Process ID.
- ppid: Process ID of parent process.
- name: Process name.
- username: Name of the user (real uid on POSIX).
- status: I.e. *running*, *sleeping*, etc.
- user: User CPU time.
- system: System CPU time.
- rss: Resident set size, the amount of memory the process currently uses. Does not include memory that is swapped out. It does include shared libraries.
- vms: Virtual memory size. All memory the process has access to.
- created: Time stamp when the process was created.
- ps_handle: ps_handle objects, in a list column.

Additional columns that can be requested via columns:

- cmdline: Command line, in a single string, from ps_cmdline().
- vcmdline: Like cmdline, but each command line argument in a separate string.
- cwd: Current working directory, from ps_cwd().
- exe: Path of the executable of the process, from ps_exe().
- num_fds: Number of open file descriptors, from ps_num_fds().
- num_threads: Number of threads, from ps_num_threads().
- cpu_children_user: See ps_cpu_times().
- cpu_children_system: See ps_cpu_times().
- terminal: Terminal device, from ps_terminal().
- uid_real: Real user id, from ps_uids().
- uid_effective: Effective user id, from ps_uids().
- uid_saved: Saved user id, from ps_uids().
- gid_real: Real group id, from ps_gids().
- gid_effective: Effective group id, from ps_gids().
- gid_saved: Saved group id, from ps_gids().
- mem_shared: See ps_memory_info().
- mem_text: See ps_memory_info().
- mem_data: See ps_memory_info().
- mem_lib: See ps_memory_info().
- mem_dirty: See ps_memory_info().
- mem_pfaults: See ps_memory_info().

- `mem_pageins`: See `ps_memory_info()`.
- `mem_maxrss`: See `ps_memory_full_info()`.
- `mem_uss`: See `ps_memory_full_info()`.
- `mem_pss`: See `ps_memory_full_info()`.
- `mem_swap`: See `ps_memory_full_info()`.

Use `"*"` in `columns` to include all columns.

Value

Data frame, see columns below.

`ps_apps` *List currently running applications*

Description

This function currently only works on macOS.

Usage

```
ps_apps()
```

Value

A data frame with columns:

- `pid`: integer process id.
- `name`: process name.
- `bundle_identifier`: bundle identifier, e.g. `com.apple.dock`.
- `bundle_url`: bundle URL, a `file://` URL to the app bundle.
- `arch`: executable architecture, possible values are arm64, i386, x86_64, ppc, ppc64.
- `executable_url`: `file://` URL to the executable file.
- `launch_date`: launch time stamp, a `POSIXct` object, may be `NA`.
- `finished_launching`: whether the app has finished launching.
- `active`: whether the app is active.
- `activation_policy`: one of the following values:
  - `regular`: the application is an ordinary app that appears in the Dock and may have a user interface.
  - `accessory`: the application doesn't appear in the Dock and doesn't have a menu bar, but it may be activated programmatically or by clicking on one of its windows.
  - `prohibited`: the application doesn't appear in the Dock and may not create windows or be activated.

Examples

```
ps_apps()
```

`ps_boot_time` *Boot time of the system*

Description

Boot time of the system

Usage

```
ps_boot_time()
```

Value

A `POSIXct` object.

`ps_children` *List of child processes (process objects) of the process. Note that this typically requires enumerating all processes on the system, so it is a costly operation.*

Description

List of child processes (process objects) of the process. Note that this typically requires enumerating all processes on the system, so it is a costly operation.

Usage

```
ps_children(p = ps_handle(), recursive = FALSE)
```

Arguments

`p`	Process handle.
`recursive`	Whether to include the children of the children, etc.

Value

List of `ps_handle` objects.

See Also

Other process handle functions: ps_cmdline(), ps_connections(), ps_cpu_times(), ps_create_time(), ps_cwd(), ps_descent(), ps_environ(), ps_exe(), ps_handle(), ps_interrupt(), ps_is_running(), ps_kill(), ps_memory_info(), ps_name(), ps_num_fds(), ps_num_threads(), ps_open_files(), ps_pid(), ps_ppid(), ps_resume(), ps_send_signal(), ps_shared_libs(), ps_status(), ps_suspend(), ps_terminal(), ps_terminate(), ps_uids(), ps_username()

Examples

```
p <- ps_parent(ps_handle())
ps_children(p)
```

---

ps_cmdline *Command line of the process*

Description

Command line of the process, i.e. the executable and the command line arguments, in a character vector. On Unix the program might change its command line, and some programs actually do it.

Usage

```
ps_cmdline(p = ps_handle())
```

Arguments

p	Process handle.

Details

For a zombie process it throws a `zombie_process` error.

Value

Character vector.

See Also

Other process handle functions: ps_children(), ps_connections(), ps_cpu_times(), ps_create_time(), ps_cwd(), ps_descent(), ps_environ(), ps_exe(), ps_handle(), ps_interrupt(), ps_is_running(), ps_kill(), ps_memory_info(), ps_name(), ps_num_fds(), ps_num_threads(), ps_open_files(), ps_pid(), ps_ppid(), ps_resume(), ps_send_signal(), ps_shared_libs(), ps_status(), ps_suspend(), ps_terminal(), ps_terminate(), ps_uids(), ps_username()

Examples

```
p <- ps_handle()
p
ps_name(p)
ps_exe(p)
ps_cmdline(p)
```

---

ps_connections *List network connections of a process*

Description

For a zombie process it throws a `zombie_process` error.

Usage

```
ps_connections(p = ps_handle())
```

Arguments

`p`	Process handle.

Value

Data frame, with columns:

- `fd`: integer file descriptor on POSIX systems, `NA` on Windows.
- `family`: Address family, string, typically `AF_UNIX`, `AF_INET` or `AF_INET6`.
- `type`: Socket type, string, typically `SOCK_STREAM` (TCP) or `SOCK_DGRAM` (UDP).
- `laddr`: Local address, string, `NA` for UNIX sockets.
- `lport`: Local port, integer, `NA` for UNIX sockets.
- `raddr`: Remote address, string, `NA` for UNIX sockets. This is always `NA` for `AF_INET` sockets on Linux.
- `rport`: Remote port, integer, `NA` for UNIX sockets.
- `state`: Socket state, e.g. `CONN_ESTABLISHED`, etc. It is `NA` for UNIX sockets.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
ps_connections(p)
sc <- socketConnection("httpbin.org", port = 80)
ps_connections(p)
close(sc)
ps_connections(p)
```

`ps_cpu_count` *Number of logical or physical CPUs*

Description

If cannot be determined, it returns `NA`. It also returns `NA` on older Windows systems, e.g. Vista or older and Windows Server 2008 or older.

Usage

```
ps_cpu_count(logical = TRUE)
```

Arguments

`logical`	Whether to count logical CPUs.

Value

Integer scalar.

Examples

```
ps_cpu_count(logical = TRUE)
ps_cpu_count(logical = FALSE)
```

`ps_cpu_times` *CPU times of the process*

Description

All times are measured in seconds:

- `user`: Amount of time that this process has been scheduled in user mode.
- `system`: Amount of time that this process has been scheduled in kernel mode
- `children_user`: On Linux, amount of time that this process's waited-for children have been scheduled in user mode.
- `children_system`: On Linux, Amount of time that this process's waited-for children have been scheduled in kernel mode.

Usage

```
ps_cpu_times(p = ps_handle())
```

Arguments

p	Process handle.

Details

Throws a `zombie_process()` error for zombie processes.

Value

Named real vector or length four: `user`, `system`, `children_user`, `children_system`. The last two are `NA` on non-Linux systems.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_cpu_times(p)
proc.time()
```

`ps_create_time` *Start time of a process*

Description

The pid and the start time pair serves as the identifier of the process, as process ids might be reused, but the chance of starting two processes with identical ids within the resolution of the timer is minimal.

Usage

```
ps_create_time(p = ps_handle())
```

Arguments

p	Process handle.

Details

This function works even if the process has already finished.

Value

`POSIXct` object, start time, in GMT.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_create_time(p)
```

`ps_cwd` *Process current working directory as an absolute path.*

Description

For a zombie process it throws a `zombie_process` error.

Usage

```
ps_cwd(p = ps_handle())
```

Arguments

`p`	Process handle.

Value

String scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_cwd(p)
```

`ps_descent` *Query the ancestry of a process*

Description

Query the parent processes recursively, up to the first process. (On some platforms, like Windows, the process tree is not a tree and may contain loops, in which case `ps_descent()` only goes up until the first repetition.)

Usage

```
ps_descent(p = ps_handle())
```

Arguments

`p`	Process handle.

Value

A list of process handles, starting with `p`, each one is the parent process of the previous one.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
ps_descent()
```

`ps_disk_io_counters` *System-wide disk I/O counters*

Description

Returns a data.frame of system-wide disk I/O counters.

Usage

```
ps_disk_io_counters()
```

Details

Includes the following non-NA fields for all supported platforms:

- `read_count`: number of reads
- `write_count`: number of writes
- `read_bytes`: number of bytes read
- `write_bytes`: number of bytes written

And for only some platforms:

- `read_time`: time spent reading from disk (in milliseconds)
- `write_time`: time spent writing to disk (in milliseconds)
- `busy_time`: time spent doing actual I/Os (in milliseconds)
- `read_merged_count`: number of merged reads (see iostats doc)
- `write_merged_count`: number of merged writes (see iostats doc)

Value

A data frame of one row per disk of I/O stats, with columns name, `read_count` `read_merged_count` `read_bytes`, `read_time`, `write_count`, `write_merged_count`, `write_bytes` `write_time`, and `busy_time`.

See Also

Other disk functions: `ps_disk_partitions()`, `ps_disk_usage()`

Examples

```
ps_disk_io_counters()
```

ps_disk_partitions *List all mounted partitions*

Description

The output is similar the Unix mount and df commands.

Usage

```
ps_disk_partitions(all = FALSE)
```

Arguments

all	Whether to list virtual devices as well. If FALSE, on Linux it will still list overlay and grpcfuse file systems, to provide some useful information in Docker containers.

Value

A data frame with columns device, mountpoint, fstype and options.

See Also

Other disk functions: ps_disk_io_counters(), ps_disk_usage()

Examples

```
ps_disk_partitions(all = TRUE)
ps_disk_partitions()
```

ps_disk_usage *Disk usage statistics, per partition*

Description

The output is similar to the Unix df command.

Usage

```
ps_disk_usage(paths = ps_disk_partitions()$mountpoint)
```

Arguments

paths	The mounted file systems to list. By default all file systems returned by ps_disk_partitions() is listed.

Details

Note that on Unix a small percentage of the disk space (5% typically) is reserved for the superuser. `ps_disk_usage()` returns the space available to the calling user.

Value

A data frame with columns `mountpoint`, `total`, `used`, `available` and `capacity`.

See Also

Other disk functions: `ps_disk_io_counters()`, `ps_disk_partitions()`

Examples

```
ps_disk_usage()
```

`ps_environ` *Environment variables of a process*

Description

`ps_environ()` returns the environment variables of the process, in a named vector, similarly to the return value of `Sys.getenv()` (without arguments).

Usage

```
ps_environ(p = ps_handle())

ps_environ_raw(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

Note: this usually does not reflect changes made after the process started.

`ps_environ_raw()` is similar to `p$environ()` but returns the unparsed `"var=value"` strings. This is faster, and sometimes good enough.

These functions throw a `zombie_process` error for zombie processes.

Value

`ps_environ()` returns a named character vector (that has a `Dlist` class, so it is printed nicely), `ps_environ_raw()` returns a character vector.

macOS issues

`ps_environ()` usually does not work on macOS nowadays. This is because macOS does not allow reading the environment variables of another process. Accoding to the Darwin source code, `ps_environ` will work is one of these conditions hold:

- You are running a development or debug kernel, i.e. if you are debugging the macOS kernel itself.
- The target process is same as the calling process.
- SIP if off.
- The target process is not restricted, e.g. it is running a binary that was not signed.
- The calling process has the `com.apple.private.read-environment-variables` entitlement. However adding this entitlement to the R binary makes R crash on startup.

Otherwise `ps_environ` will return an empty set of environment variables on macOS.

Issue 121 might have more information about this.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
env <- ps_environ(p)
env[["R_HOME"]]
```

`ps_exe` *Full path of the executable of a process*

Description

Path to the executable of the process. May also be an empty string or `NA` if it cannot be determined.

Usage

```
ps_exe(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

For a zombie process it throws a `zombie_process` error.

Value

Character scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_name(p)
ps_exe(p)
ps_cmdline(p)
```

`ps_fs_info` *File system information for files*

Description

File system information for files

Usage

```
ps_fs_info(paths = "/")
```

Arguments

`paths`	A path or a vector of paths. `ps_fs_info()` returns information about the file systems of all paths. `path` may contain direcories as well.

Value

Data frame with file system information for each path in `paths`, one row per path. Common columns for all operating systems:

- `path`: The input paths, i.e. the `paths` argument.

- `mountpoint`: Directory where the file system is mounted. On Linux there is a small chance that it was not possible to look this up, and it is `NA_character_`. This is the drive letter or the mount directory on Windows, with a trailing `\`.
- `name`: Device name. On Linux there is a small chance that it was not possible to look this up, and it is `NA_character_`. On Windows this is the volume GUID path of the form `\\?\Volume{GUID}\`.
- `type`: File system type (character). On Linux there is a tiny chance that it was not possible to look this up, and it is `NA_character_`.
- `block_size`: File system block size. This is the sector size on Windows, in bytes.
- `transfer_block_size`: Pptimal transfer block size. On Linux it is currently always the same as `block_size`. This is the cluster size on Windows, in bytes.
- `total_data_blocks`: Total data blocks in file system. On Windows this is the number of sectors.
- `free_blocks`: Free blocks in file system. On Windows this is the number of free sectors.
- `free_blocks_non_superuser`: Free blocks for a non-superuser, which might be different on Unix. On Windows this is the number of free sectors for the calling user.
- `id`: File system id. This is a raw vector. On Linux it is often all zeros. It is always `NULL` on Windows.
- `owner`: User that mounted the file system. On Linux and Windows this is currently always `NA_real_`.
- `type_code`: Type of file system, a numeric code. On Windows this this is `NA_real_`.
- `subtype_code`: File system subtype (flavor). On Linux and Windows this is always `NA_real_`.

The rest of the columns are flags, and they are operating system dependent.

macOS:

- `RDONLY`: A read-only filesystem.
- `SYNCHRONOUS`: File system is written to synchronously.
- `NOEXEC`: Can't exec from filesystem.
- `NOSUID`: Setuid bits are not honored on this filesystem.
- `NODEV`: Don't interpret special files.
- `UNION`: Union with underlying filesysten.
- `ASYNC`: File system written to asynchronously.
- `EXPORTED`: File system is exported.
- `LOCAL`: File system is stored locally.
- `QUOTA`: Quotas are enabled on this file system.
- `ROOTFS`: This file system is the root of the file system.
- `DOVOLFS`: File system supports volfs.
- `DONTBROWSE`: File system is not appropriate path to user data.
- `UNKNOWNPERMISSIONS`: VFS will ignore ownership information on filesystem filesystemtem objects.

- `AUTOMOUNTED`: File system was mounted by automounter.
- `JOURNALED`: File system is journaled.
- `DEFWRITE`: File system should defer writes.
- `MULTILABEL`: MAC support for individual labels.
- `CPROTECT`: File system supports per-file encrypted data protection.

Linux:

- `MANDLOCK`: Mandatory locking is permitted on the filesystem (see `fcntl(2)`).
- `NOATIME`: Do not update access times; see `mount(2)`.
- `NODEV`: Disallow access to device special files on this filesystem.
- `NODIRATIME`: Do not update directory access times; see mount(2).
- `NOEXEC`: Execution of programs is disallowed on this filesystem.
- `NOSUID`: The set-user-ID and set-group-ID bits are ignored by `exec(3)` for executable files on this filesystem
- `RDONLY`: This filesystem is mounted read-only.
- `RELATIME`: Update atime relative to mtime/ctime; see `mount(2)`.
- `SYNCHRONOUS`: Writes are synched to the filesystem immediately (see the description of `O_SYNC` in ‘open(2)“).
- `NOSYMFOLLOW`: Symbolic links are not followed when resolving paths; see ‘mount(2)“.

Windows:

- `CASE_SENSITIVE_SEARCH`: Supports case-sensitive file names.
- `CASE_PRESERVED_NAMES`: Supports preserved case of file names when it places a name on disk.
- `UNICODE_ON_DISK`: Supports Unicode in file names as they appear on disk.
- `PERSISTENT_ACLS`: Preserves and enforces access control lists (ACL). For example, the NTFS file system preserves and enforces ACLs, and the FAT file system does not.
- `FILE_COMPRESSION`: Supports file-based compression.
- `VOLUME_QUOTAS`: Supports disk quotas.
- `SUPPORTS_SPARSE_FILES`: Supports sparse files.
- `SUPPORTS_REPARSE_POINTS`: Supports reparse points.
- `SUPPORTS_REMOTE_STORAGE`: Supports remote storage.
- `RETURNS_CLEANUP_RESULT_INFO`: On a successful cleanup operation, the file system returns information that describes additional actions taken during cleanup, such as deleting the file. File system filters can examine this information in their post-cleanup callback.
- `SUPPORTS_POSIX_UNLINK_RENAME`: Supports POSIX-style delete and rename operations.
- `VOLUME_IS_COMPRESSED`: It is a compressed volume, for example, a DoubleSpace volume.
- `SUPPORTS_OBJECT_IDS`: Supports object identifiers.
- `SUPPORTS_ENCRYPTION`: Supports the Encrypted File System (EFS).
- `NAMED_STREAMS`: Supports named streams.

- `READ_ONLY_VOLUME`: It is read-only.
- `SEQUENTIAL_WRITE_ONCE`: Supports a single sequential write.
- `SUPPORTS_TRANSACTIONS`: Supports transactions.
- `SUPPORTS_HARD_LINKS`: The volume supports hard links.
- `SUPPORTS_EXTENDED_ATTRIBUTES`: Supports extended attributes.
- `SUPPORTS_OPEN_BY_FILE_ID`: Supports open by FileID.
- `SUPPORTS_USN_JOURNAL`: Supports update sequence number (USN) journals.
- `SUPPORTS_INTEGRITY_STREAMS`: Supports integrity streams.
- `SUPPORTS_BLOCK_REFCOUNTING`: The volume supports sharing logical clusters between files on the same volume.
- `SUPPORTS_SPARSE_VDL`: The file system tracks whether each cluster of a file contains valid data (either from explicit file writes or automatic zeros) or invalid data (has not yet been written to or zeroed).
- `DAX_VOLUME`: The volume is a direct access (DAX) volume.
- `SUPPORTS_GHOSTING`: Supports ghosting.

Examples

```
ps_fs_info(c("/", "~", "."))
```

`ps_fs_mount_point` *Find the mount point of a file or directory*

Description

Find the mount point of a file or directory

Usage

```
ps_fs_mount_point(paths)
```

Arguments

`paths`	Paths to files, directories, devices, etc. They must exist. They are normalized using `base::normalizePath()`.

Value

Character vector, paths to the mount points of the input `paths`.

Examples

```
ps_fs_mount_point(".")
```

`ps_fs_stat` *File status*

Description

This function is currently not implemented on Windows.

Usage

```
ps_fs_stat(paths, follow = TRUE)
```

Arguments

`paths`	Paths to files, directories, devices, etc. They must exist. They are expanded using `base::path.expand()`.
`follow`	Whether to follow symbolic links. If `FALSE` it returns information on the links themselves.

Value

Data frame with one row for each path in `paths`. Columns:

- `path`: Expanded `paths`.
- `dev_major`: Major device ID of the device the path resides on.
- `dev_minor`: Minor device ID of the device the path resodes on.
- `inode`: Inode number.
- `mode`: File type and mode (permissions). It is easier to use the `type` and `permissions` columns.
- `type`: File type, character. One of regular file, directory, character device, block device, FIFO, symbolic link, socket.
- `permissions`: Permissions, numeric code in an integer column.
- `nlink`: Number of hard links.
- `uid`: User id of owner.
- `gid`: Group id of owner.
- `rdev_major`: If the path is a device, its major device id, otherwise `NA_integer_`.
- `rdev_minor`: IF the path is a device, its minor device id, otherwise `NA_integer_`.
- `size`: File size in bytes.
- `block_size`: Block size for filesystem I/O.
- `blocks`: Number of 512B blocks allocated.
- `access_time`: Time of last access.
- `modification_time`: Time of last modification.
- `change_time`: Time of last status change.

Examples

```
ps_fs_stat(c(".", tempdir()))
```

`ps_get_cpu_affinity` *Query or set CPU affinity*

Description

`ps_get_cpu_affinity()` queries the CPU affinity of a process. `ps_set_cpu_affinity()` sets the CPU affinity of a process.

Usage

```
ps_get_cpu_affinity(p = ps_handle())

ps_set_cpu_affinity(p = ps_handle(), affinity)
```

Arguments

`p`	Process handle.
`affinity`	Integer vector of CPU numbers to restrict a process to. CPU numbers start with zero, and they have to be smaller than the number of (logical) CPUs, see `ps_cpu_count()`.

Details

CPU affinity consists in telling the OS to run a process on a limited set of CPUs only (on Linux cmdline, the `taskset` command is typically used).

These functions are only supported on Linux and Windows. They error on macOS.

Value

`ps_get_cpu_affinity()` returns an integer vector of CPU numbers, starting with zero.

`ps_set_cpu_affinity()` returns `NULL`, invisibly.

Examples

```
# current
orig <- ps_get_cpu_affinity()
orig

# restrict
ps_set_cpu_affinity(affinity = 0:0)
ps_get_cpu_affinity()

# restore
```

```
ps_set_cpu_affinity(affinity = orig)
ps_get_cpu_affinity()
```

`ps_handle` *Create a process handle*

Description

Create a process handle

Usage

```
ps_handle(pid = NULL, time = NULL)

## S3 method for class 'ps_handle'
as.character(x, ...)

## S3 method for class 'ps_handle'
format(x, ...)

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

Arguments

`pid`	Process id. Integer scalar. `NULL` means the current R process.
`time`	Start time of the process. Usually `NULL` and ps will query the start time.
`x`	Process handle.
`...`	Not used currently.

Value

`ps_handle()` returns a process handle (class `ps_handle`).

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
```

ps_interrupt — *Interrupt a process*

Description

Sends `SIGINT` on POSIX, and 'CTRL+C' or 'CTRL+BREAK' on Windows.

Usage

```
ps_interrupt(p = ps_handle(), ctrl_c = TRUE)
```

Arguments

`p`	Process handle or a list of process handles.
`ctrl_c`	On Windows, whether to send 'CTRL+C'. If `FALSE`, then 'CTRL+BREAK' is sent. Ignored on non-Windows platforms.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

ps_is_running — *Checks whether a process is running*

Description

It returns `FALSE` if the process has already finished.

Usage

```
ps_is_running(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

It uses the start time of the process to work around pid reuse. I.e.

Value

Logical scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_is_running(p)
```

`ps_kill` *Kill one or more processes*

Description

Kill the process with SIGKILL preemptively checking whether PID has been reused. On Windows it uses `TerminateProcess()`.

Usage

```
ps_kill(p = ps_handle(), grace = 200)
```

Arguments

`p`	Process handle, or a list of process handles.
`grace`	Grace period, in milliseconds, used on Unix. If it is not zero, then `ps_kill()` first sends a `SIGTERM` signal to all processes in `p`. If some proccesses do not terminate within `grace` milliseconds after the `SIGTERM` signal, `ps_kill()` kills them by sending `SIGKILL` signals.

Details

Note that since ps version 1.8, `ps_kill()` does not error if the `p` process (or some processes if `p` is a list) are already terminated.

Value

Character vector, with one element for each process handle in `p`. If the process was already dead before `ps_kill()` tried to kill it, the corresponding return value is `"dead"`. If `ps_kill()` just killed it, it is `"killed"`.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
px <- processx::process$new("sleep", "10")
p <- ps_handle(px$get_pid())
p
ps_kill(p)
p
ps_is_running(p)
px$get_exit_status()
```

ps_loadavg — *Return the average system load over the last 1, 5 and 15 minutes as a tuple.*

Description

The “load” represents the processes which are in a runnable state, either using the CPU or waiting to use the CPU (e.g. waiting for disk I/O). On Windows this is emulated by using a Windows API that spawns a thread which keeps running in background and updates results every 5 seconds, mimicking the UNIX behavior. Thus, on Windows, the first time this is called and for the next 5 seconds it will return a meaningless (0.0, 0.0, 0.0) vector. The numbers returned only make sense if related to the number of CPU cores installed on the system. So, for instance, a value of 3.14 on a system with 10 logical CPUs means that the system load was 31.4% percent over the last N minutes.

Usage

```
ps_loadavg()
```

Value

Numeric vector of length 3.

Examples

```
ps_loadavg()
```

`ps_mark_tree` *Mark a process and its (future) child tree*

Description

`ps_mark_tree()` generates a random environment variable name and sets it in the current R process. This environment variable will be (by default) inherited by all child (and grandchild, etc.) processes, and will help finding these processes, even if and when they are (no longer) related to the current R process. (I.e. they are not connected in the process tree.)

Usage

```
ps_mark_tree()

with_process_cleanup(expr)

ps_find_tree(marker)

ps_kill_tree(marker, sig = signals()$SIGKILL, grace = 200)
```

Arguments

`expr`	R expression to evaluate in the new context.
`marker`	String scalar, the name of the environment variable to use to find the marked processes.
`sig`	The signal to send to the marked processes on Unix. On Windows this argument is ignored currently.
`grace`	Grace period, in milliseconds, used on Unix, if `sig` is `SIGKILL`. If it is not zero, then `ps_kill_tree()` first sends a `SIGTERM` signal to all processes. If some proccesses do not terminate within `grace` milliseconds after the `SIGTERM` signal, `ps_kill_tree()` kills them by sending `SIGKILL` signals.

Details

`ps_find_tree()` finds the processes that set the supplied environment variable and returns them in a list.

`ps_kill_tree()` finds the processes that set the supplied environment variable, and kills them (or sends them the specified signal on Unix).

`with_process_cleanup()` evaluates an R expression, and cleans up all external processes that were started by the R process while evaluating the expression. This includes child processes of child processes, etc., recursively. It returns a list with entries: `result` is the result of the expression, `visible` is TRUE if the expression should be printed to the screen, and `process_cleanup` is a named integer vector of the cleaned pids, names are the process names.

If `expr` throws an error, then so does `with_process_cleanup()`, the same error. Nevertheless processes are still cleaned up.

Value

`ps_mark_tree()` returns the name of the environment variable, which can be used as the `marker` in `ps_kill_tree()`.

`ps_find_tree()` returns a list of `ps_handle` objects.

`ps_kill_tree()` returns the pids of the killed processes, in a named integer vector. The names are the file names of the executables, when available.

`with_process_cleanup()` returns the value of the evaluated expression.

macOS issues

These functions do not work on macOS, unless specific criteria are met. See `ps_environ()` for details.

Note

Note that `with_process_cleanup()` is problematic if the R process is multi-threaded and the other threads start subprocesses. `with_process_cleanup()` cleans up those processes as well, which is probably not what you want. This is an issue for example in RStudio. Do not use `with_process_cleanup()`, unless you are sure that the R process is single-threaded, or the other threads do not start subprocesses. E.g. using it in package test cases is usually fine, because RStudio runs these in a separate single-threaded process.

The same holds for manually running `ps_mark_tree()` and then `ps_find_tree()` or `ps_kill_tree()`.

A safe way to use process cleanup is to use the processx package to start subprocesses, and set the `cleanup_tree = TRUE` in `processx::run()` or the processx::process constructor.

`ps_memory_info` *Memory usage information*

Description

Memory usage information

Usage

```
ps_memory_info(p = ps_handle())

ps_memory_full_info(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

`ps_memory_info()` returns information about memory usage.

It returns a named vector. Portable fields:

- `rss`: "Resident Set Size", this is the non-swapped physical memory a process has used (bytes). On UNIX it matches "top"'s 'RES' column (see doc). On Windows this is an alias for `wset` field and it matches "Memory" column of `taskmgr.exe`.
- `vmem`: "Virtual Memory Size", this is the total amount of virtual memory used by the process (bytes). On UNIX it matches "top"'s 'VIRT' column (see doc). On Windows this is an alias for the `pagefile` field and it matches the "Working set (memory)" column of `taskmgr.exe`.

Non-portable fields:

- `shared`: (Linux) memory that could be potentially shared with other processes (bytes). This matches "top"'s 'SHR' column (see doc).
- `text`: (Linux): aka 'TRS' (text resident set) the amount of memory devoted to executable code (bytes). This matches "top"'s 'CODE' column (see doc).
- `data`: (Linux): aka 'DRS' (data resident set) the amount of physical memory devoted to other than executable code (bytes). It matches "top"'s 'DATA' column (see doc).
- `lib`: (Linux): the memory used by shared libraries (bytes).
- `dirty`: (Linux): the amount of memory in dirty pages (bytes).
- `pfaults`: (macOS): number of page faults.
- `pageins`: (macOS): number of actual pageins.

For the explanation of Windows fields see the PROCESS_MEMORY_COUNTERS_EX structure.

`ps_memory_full_info()` returns all fields as `ps_memory_info()`, plus additional information, but typically takes slightly longer to run, and might not have access to some processes that `ps_memory_info()` can query:

- `maxrss` maximum resident set size over the process's lifetime. This only works for the calling process, otherwise it is `NA_real_`.
- `uss`: Unique Set Size, this is the memory which is unique to a process and which would be freed if the process was terminated right now.
- `pss` (Linux only): Proportional Set Size, is the amount of memory shared with other processes, accounted in a way that the amount is divided evenly between the processes that share it. I.e. if a process has 10 MBs all to itself and 10 MBs shared with another process its PSS will be 15 MBs.
- `swap` (Linux only): amount of memory that has been swapped out to disk.

They both throw a `zombie_process()` error for zombie processes.

Value

Named real vector.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_memory_info(p)
ps_memory_full_info(p)
```

`ps_name` *Process name*

Description

The name of the program, which is typically the name of the executable.

Usage

```
ps_name(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

On Unix this can change, e.g. via an exec*() system call.

`ps_name()` works on zombie processes.

Value

Character scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_name(p)
ps_exe(p)
ps_cmdline(p)
```

`ps_num_fds` *Number of open file descriptors*

Description

Note that in some IDEs, e.g. RStudio or R.app on macOS, the IDE itself opens files from other threads, in addition to the files opened from the main R thread.

Usage

```
ps_num_fds(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

For a zombie process it throws a `zombie_process` error.

Value

Integer scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
ps_num_fds(p)
f <- file(tmp <- tempfile(), "w")
ps_num_fds(p)
close(f)
unlink(tmp)
ps_num_fds(p)
```

`ps_num_threads` *Number of threads*

Description

Throws a `zombie_process()` error for zombie processes.

Usage

```
ps_num_threads(p = ps_handle())
```

Arguments

`p`	Process handle.

Value

Integer scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_num_threads(p)
```

`ps_open_files` *Open files of a process*

Description

Note that in some IDEs, e.g. RStudio or R.app on macOS, the IDE itself opens files from other threads, in addition to the files opened from the main R thread.

Usage

```
ps_open_files(p = ps_handle())
```

Arguments

p	Process handle.

Details

For a zombie process it throws a `zombie_process` error.

Value

Data frame with columns: `fd` and `path`. `fd` is numeric file descriptor on POSIX systems, `NA` on Windows. `path` is an absolute path to the file.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
ps_open_files(p)
f <- file(tmp <- tempfile(), "w")
ps_open_files(p)
close(f)
unlink(tmp)
ps_open_files(p)
```

---

ps_os_type *Query the type of the OS*

Description

Query the type of the OS

Usage

```
ps_os_type()

ps_is_supported()
```

Value

`ps_os_type` returns a named logical vector. The rest of the functions return a logical scalar.

`ps_is_supported()` returns `TRUE` if ps supports the current platform.

Examples

```
ps_os_type()
ps_is_supported()
```

---

ps_pid *Pid of a process handle*

Description

This function works even if the process has already finished.

Usage

```
ps_pid(p = ps_handle())
```

Arguments

p	Process handle.

Value

Process id.

See Also

Other process handle functions: ps_children(), ps_cmdline(), ps_connections(), ps_cpu_times(), ps_create_time(), ps_cwd(), ps_descent(), ps_environ(), ps_exe(), ps_handle(), ps_interrupt(), ps_is_running(), ps_kill(), ps_memory_info(), ps_name(), ps_num_fds(), ps_num_threads(), ps_open_files(), ps_ppid(), ps_resume(), ps_send_signal(), ps_shared_libs(), ps_status(), ps_suspend(), ps_terminal(), ps_terminate(), ps_uids(), ps_username()

Examples

```
p <- ps_handle()
p
ps_pid(p)
ps_pid(p) == Sys.getpid()
```

`ps_pids` *Ids of all processes on the system*

Description

Ids of all processes on the system

Usage

```
ps_pids()
```

Value

Integer vector of process ids.

`ps_ppid` *Parent pid or parent process of a process*

Description

`ps_ppid()` returns the parent pid, `ps_parent()` returns a `ps_handle` of the parent.

Usage

```
ps_ppid(p = ps_handle())

ps_parent(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

On POSIX systems, if the parent process terminates, another process (typically the pid 1 process) is marked as parent. `ps_ppid()` and `ps_parent()` will return this process then.

Both `ps_ppid()` and `ps_parent()` work for zombie processes.

Value

`ps_ppid()` returns and integer scalar, the pid of the parent of `p`. `ps_parent()` returns a `ps_handle`.

See Also

Other process handle functions: ps_children(), ps_cmdline(), ps_connections(), ps_cpu_times(), ps_create_time(), ps_cwd(), ps_descent(), ps_environ(), ps_exe(), ps_handle(), ps_interrupt(), ps_is_running(), ps_kill(), ps_memory_info(), ps_name(), ps_num_fds(), ps_num_threads(), ps_open_files(), ps_pid(), ps_resume(), ps_send_signal(), ps_shared_libs(), ps_status(), ps_suspend(), ps_terminal(), ps_terminate(), ps_uids(), ps_username()

Examples

```
p <- ps_handle()
p
ps_ppid(p)
ps_parent(p)
```

---

ps_resume *Resume (continue) a stopped process*

Description

Resume process execution with SIGCONT preemptively checking whether PID has been reused. On Windows this has the effect of resuming all process threads.

Usage

```
ps_resume(p = ps_handle())
```

Arguments

p	Process handle or a list of process handles.

See Also

Other process handle functions: ps_children(), ps_cmdline(), ps_connections(), ps_cpu_times(), ps_create_time(), ps_cwd(), ps_descent(), ps_environ(), ps_exe(), ps_handle(), ps_interrupt(), ps_is_running(), ps_kill(), ps_memory_info(), ps_name(), ps_num_fds(), ps_num_threads(), ps_open_files(), ps_pid(), ps_ppid(), ps_send_signal(), ps_shared_libs(), ps_status(), ps_suspend(), ps_terminal(), ps_terminate(), ps_uids(), ps_username()

Examples

```
px <- processx::process$new("sleep", "10")
p <- ps_handle(px$get_pid())
p
ps_suspend(p)
ps_status(p)
ps_resume(p)
ps_status(p)
ps_kill(p)
```

`ps_send_signal` *Send signal to a process*

Description

Send a signal to the process. Not implemented on Windows. See `signals()` for the list of signals on the current platform.

Usage

```
ps_send_signal(p = ps_handle(), sig)
```

Arguments

`p`	Process handle, or a list of process handles.
`sig`	Signal number, see `signals()`.

Details

It checks if the process is still running, before sending the signal, to avoid signalling the wrong process, because of pid reuse.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
px <- processx::process$new("sleep", "10")
p <- ps_handle(px$get_pid())
p
ps_send_signal(p, signals()$SIGINT)
p
ps_is_running(p)
px$get_exit_status()
```

ps_shared_libs — *List the dynamically loaded libraries of a process*

Description

Note: this function currently only works on Windows.

Usage

```
ps_shared_libs(p = ps_handle())
```

Arguments

p	Process handle.

Value

Data frame with one column currently: `path`, the absolute path to the loaded module or shared library. On Windows the list includes the executable file itself.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Other shared library tools: `ps_shared_lib_users()`

Examples

```
# The loaded DLLs of the current process
ps_shared_libs()
```

ps_shared_lib_users — *List all processes that loaded a shared library*

Description

List all processes that loaded a shared library

Usage

```
ps_shared_lib_users(paths, user = ps_username(), filter = NULL)
```

Arguments

`paths`	Character vector of paths of shared libraries to look up. They must be absolute paths. They don't need to exist. Forward slashes are converted to backward slashes on Windows, and the output will always have backward slashes in the paths.
`user`	Character scalar or `NULL`. If not `NULL`, then only the processes of this user are considered. It defaults to the current user.
`filter`	Character vector or `NULL`. If not NULL, then it is a vector of glob expressions, used to filter the process names.

Details

Notes::

This function currently only works on Windows.

On Windows, a 32 bit R process can only list other 32 bit processes. Similarly, a 64 bit R process can only list other 64 bit processes. This is a limitation of the Windows API.

Even though Windows file systems are (almost always) case insensitive, the matching of `paths`, `user` and also `filter` are case sensitive. This might change in the future.

This function can be very slow on Windows, because it needs to enumerate all shared libraries of all processes in the system, unless the `filter` argument is set. Make sure you set `filter` if you can.

If you want to look up multiple shared libraries, list all of them in `paths`, instead of calling `ps_shared_lib_users` for each individually.

If you are after libraries loaded by R processes, you might want to set `filter` to `c("Rgui.exe", "Rterm.exe", "rsession.exe")` The last one is for RStudio.

Value

A data frame with columns:

- `dll`: the file name of the dll file, without the path,
- `path`: path to the shared library,
- `pid`: process ID of the process,
- `name`: name of the process,
- `username`: username of process owner,
- `ps_handle`: `ps_handle` object, that can be used to further query and manipulate the process.

See Also

Other shared library tools: `ps_shared_libs()`

Examples

```
dlls <- vapply(getLoadedDLLs(), "[[", character(1), "path")
psdll <- dlls[["ps"]][[1]]
r_procs <- c("Rgui.exe", "Rterm.exe", "rsession.exe")
ps_shared_lib_users(psdll, filter = r_procs)
```

ps_status *Current process status*

Description

One of the following:

- `"idle"`: Process being created by fork, or process has been sleeping for a long time. macOS only.
- `"running"`: Currently runnable on macOS and Windows. Actually running on Linux.
- `"sleeping"` Sleeping on a wait or poll.
- `"disk_sleep"` Uninterruptible sleep, waiting for an I/O operation (Linux only).
- `"stopped"` Stopped, either by a job control signal or because it is being traced.
- `"uninterruptible"` Process is in uninterruptible wait. macOS only.
- `"tracing_stop"` Stopped for tracing (Linux only).
- `"zombie"` Zombie. Finished, but parent has not read out the exit status yet.
- `"dead"` Should never be seen (Linux).
- `"wake_kill"` Received fatal signal (Linux only).
- `"waking"` Paging (Linux only, not valid since the 2.6.xx kernel).

Usage

```
ps_status(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

It might return `NA_character_` on macOS.

Works for zombie processes.

Value

Character scalar.

Note on macOS

On macOS `ps_status()` often falls back to calling the external `ps` program, because macOS does not let R access the status of most other processes. Notably, it is usually able to access the status of other R processes.

The external `ps` program always runs as the root user, and it also has special entitlements, so it can typically access the status of most processes.

If this behavior is problematic for you, e.g. because calling an external program is too slow, set the `ps.no_external_ps` option to `TRUE`:

```
options(ps.no_external_ps = TRUE)
```

Note that setting this option to `TRUE` will cause `ps_status()` to return `NA_character_` for most processes.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_status(p)
```

`ps_suspend` *Suspend (stop) the process*

Description

Suspend process execution with `SIGSTOP` preemptively checking whether PID has been reused. On Windows this has the effect of suspending all process threads.

Usage

```
ps_suspend(p = ps_handle())
```

Arguments

`p`	Process handle or a list of process handles.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
px <- processx::process$new("sleep", "10")
p <- ps_handle(px$get_pid())
p
ps_suspend(p)
ps_status(p)
ps_resume(p)
ps_status(p)
ps_kill(p)
```

`ps_system_cpu_times` *System CPU times.*

Description

Every attribute represents the seconds the CPU has spent in the given mode. The attributes availability varies depending on the platform:

- `user`: time spent by normal processes executing in user mode; on Linux this also includes guest time.
- `system`: time spent by processes executing in kernel mode.
- `idle`: time spent doing nothing.

Usage

```
ps_system_cpu_times()
```

Details

Platform-specific fields:

- `nice` (UNIX): time spent by niced (prioritized) processes executing in user mode; on Linux this also includes guest_nice time.
- `iowait` (Linux): time spent waiting for I/O to complete. This is not accounted in idle time counter.
- `irq` (Linux): time spent for servicing hardware interrupts.
- `softirq` (Linux): time spent for servicing software interrupts.
- `steal` (Linux 2.6.11+): time spent by other operating systems running in a virtualized environment.
- `guest` (Linux 2.6.24+): time spent running a virtual CPU for guest operating systems under the control of the Linux kernel.
- `guest_nice` (Linux 3.2.0+): time spent running a niced guest (virtual CPU for guest operating systems under the control of the Linux kernel).

Value

Named list

Examples

```
ps_system_cpu_times()
```

`ps_system_memory` *Statistics about system memory usage*

Description

Statistics about system memory usage

Usage

```
ps_system_memory()
```

Value

Named list. All numbers are in bytes:

- `total`: total physical memory (exclusive swap).
- `avail` the memory that can be given instantly to processes without the system going into swap. This is calculated by summing different memory values depending on the platform and it is supposed to be used to monitor actual memory usage in a cross platform fashion.
- `percent`: Percentage of memory that is taken.
- `used`: memory used, calculated differently depending on the platform and designed for informational purposes only. `total` - `free` does not necessarily match `used`.
- `free`: memory not being used at all (zeroed) that is readily available; note that this doesn't reflect the actual memory available (use `available` instead). `total` - `used` does not necessarily match `free`.
- `active`: (Unix only) memory currently in use or very recently used, and so it is in RAM.
- `inactive`: (Unix only) memory that is marked as not used.
- `wired`: (macOS only) memory that is marked to always stay in RAM. It is never moved to disk.
- `buffers`: (Linux only) cache for things like file system metadata.
- `cached`: (Linux only) cache for various things.
- `shared`: (Linux only) memory that may be simultaneously accessed by multiple processes.
- `slab`: (Linux only) in-kernel data structures cache.

See Also

Other memory functions: `ps_system_swap()`

Examples

```
ps_system_memory()
```

`ps_system_swap` *System swap memory statistics*

Description

System swap memory statistics

Usage

```
ps_system_swap()
```

Value

Named list. All numbers are in bytes:

- `total`: total swap memory.
- `used`: used swap memory.
- `free`: free swap memory.
- `percent`: the percentage usage.
- `sin`: the number of bytes the system has swapped in from disk (cumulative). This is `NA` on Windows.
- `sout`: the number of bytes the system has swapped out from disk (cumulative). This is `NA` on Windows.

See Also

Other memory functions: `ps_system_memory()`

Examples

```
ps_system_swap()
```

`ps_terminal` *Terminal device of the process*

Description

Returns the terminal of the process. Not implemented on Windows, always returns `NA_character_`. On Unix it returns `NA_character_` if the process has no terminal.

Usage

```
ps_terminal(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

Works for zombie processes.

Value

Character scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminate()`, `ps_uids()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_terminal(p)
```

ps_terminate *Terminate a Unix process*

Description

Send a SIGTERM signal to the process. Not implemented on Windows.

Usage

```
ps_terminate(p = ps_handle())
```

Arguments

p	Process handle or a list of process handles.

Details

Checks if the process is still running, to work around pid reuse.

See Also

Other process handle functions: ps_children(), ps_cmdline(), ps_connections(), ps_cpu_times(), ps_create_time(), ps_cwd(), ps_descent(), ps_environ(), ps_exe(), ps_handle(), ps_interrupt(), ps_is_running(), ps_kill(), ps_memory_info(), ps_name(), ps_num_fds(), ps_num_threads(), ps_open_files(), ps_pid(), ps_ppid(), ps_resume(), ps_send_signal(), ps_shared_libs(), ps_status(), ps_suspend(), ps_terminal(), ps_uids(), ps_username()

Examples

```
px <- processx::process$new("sleep", "10")
p <- ps_handle(px$get_pid())
p
ps_terminate(p)
p
ps_is_running(p)
px$get_exit_status()
```

ps_tty_size *Query the size of the current terminal*

Description

If the standard output of the current R process is not a terminal, e.g. because it is redirected to a file, or the R process is running in a GUI, then it will throw an error. You need to handle this error if you want to use this function in a package.

Usage

```
ps_tty_size()
```

Details

If an error happens, the error message is different depending on what type of device the standard output is. Some common error messages are:

- "Inappropriate ioctl for device."
- "Operation not supported on socket."
- "Operation not supported by device."

Whatever the error message, `ps_tty_size` always fails with an error of class `ps_unknown_tty_size`, which you can catch.

Examples

```
# An example that falls back to the 'width' option
tryCatch(
  ps_tty_size(),
  ps_unknown_tty_size = function(err) {
    c(width = getOption("width"), height = NA_integer_)
  }
)
```

---

ps_uids *User ids and group ids of the process*

Description

User ids and group ids of the process. Both return integer vectors with names: `real`, `effective` and `saved`.

Usage

```
ps_uids(p = ps_handle())

ps_gids(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

Both work for zombie processes.

They are not implemented on Windows, they throw a `not_implemented` error.

Value

Named integer vector of length 3, with names: `real`, `effective` and `saved`.

See Also

`ps_username()` returns a user *name* and works on all platforms.

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_username()`

Examples

```
p <- ps_handle()
p
ps_uids(p)
ps_gids(p)
```

`ps_username` *Owner of the process*

Description

The name of the user that owns the process. On Unix it is calculated from the real user id.

Usage

```
ps_username(p = ps_handle())
```

Arguments

`p`	Process handle.

Details

On Unix, a numeric uid id returned if the uid is not in the user database, thus a username cannot be determined.

Works for zombie processes.

Value

String scalar.

See Also

Other process handle functions: `ps_children()`, `ps_cmdline()`, `ps_connections()`, `ps_cpu_times()`, `ps_create_time()`, `ps_cwd()`, `ps_descent()`, `ps_environ()`, `ps_exe()`, `ps_handle()`, `ps_interrupt()`, `ps_is_running()`, `ps_kill()`, `ps_memory_info()`, `ps_name()`, `ps_num_fds()`, `ps_num_threads()`, `ps_open_files()`, `ps_pid()`, `ps_ppid()`, `ps_resume()`, `ps_send_signal()`, `ps_shared_libs()`, `ps_status()`, `ps_suspend()`, `ps_terminal()`, `ps_terminate()`, `ps_uids()`

Examples

```
p <- ps_handle()
p
ps_username(p)
```

`ps_users` *List users connected to the system*

Description

List users connected to the system

Usage

```
ps_users()
```

Value

A data frame with columns `username`, `tty`, `hostname`, `start_time`, `pid`. `tty` and `pid` are `NA` on Windows. `pid` is the process id of the login process. For local users the `hostname` column is the empty string.

`ps_wait` *Wait for one or more processes to terminate, with a timeout*

Description

This function supports interruption with SIGINT on Unix, or CTRL+C or CTRL+BREAK on Windows.

Usage

```
ps_wait(p, timeout = -1)
```

Arguments

p	A process handle, or a list of process handles. The process(es) to wait for.
timeout	Timeout in milliseconds. If -1, `ps_wait()` will wait indefinitely (or until it is interrupted). If 0, then it checks which processes have already terminated, and returns immediately.

Value

Logical vector, with one value of each process in `p`. For processes that terminated it contains a `TRUE` value. For processes that are still running it contains a `FALSE` value.

Examples

```
# this example calls `sleep`, so it only works on Unix
p1 <- processx::process$new("sleep", "100")
p2 <- processx::process$new("sleep", "100")

# returns c(FALSE, FALSE) immediately if p1 and p2 are running
ps_wait(list(p1$as_ps_handle(), p2$as_ps_handle()), 0)

# timeouts at one second
ps_wait(list(p1$as_ps_handle(), p2$as_ps_handle()), 1000)

p1$kill()
p2$kill()
# returns c(TRUE, TRUE) immediately
ps_wait(list(p1$as_ps_handle(), p2$as_ps_handle()), 1000)
```

ps_windows_nice_values

Get or set the priority of a process

Description

`ps_get_nice()` returns the current priority, `ps_set_nice()` sets a new priority, `ps_windows_nice_values()` list the possible priority values on Windows.

Usage

```
ps_windows_nice_values()

ps_get_nice(p = ps_handle())

ps_set_nice(p = ps_handle(), value)
```

Arguments

`p`	Process handle.
`value`	On Windows it must be a string, one of the values of `ps_windows_nice_values()`. On Unix it is a priority value that is smaller than or equal to 20.

Details

Priority values are different on Windows and Unix.

On Unix, priority is an integer, which is maximum 20. 20 is the lowest priority.

Rules::

- On Windows you can only set the priority of the processes the current user has `PROCESS_SET_INFORMATION` access rights to. This typically means your own processes.
- On Unix you can only set the priority of the your own processes. The superuser can set the priority of any process.
- On Unix you cannot set a higher priority, unless you are the superuser. (I.e. you cannot set a lower number.)
- On Unix the default priority of a process is zero.

Value

`ps_windows_nice_values()` return a character vector of possible priority values on Windows.

`ps_get_nice()` returns a string from `ps_windows_nice_values()` on Windows. On Unix it returns an integer smaller than or equal to 20.

`ps_set_nice()` return `NULL` invisibly.

`signals` *List of all supported signals*

Description

Only the signals supported by the current platform are included.

Usage

```
signals()
```

Value

List of integers, named by signal names.

Index