pypsutil

cptpcrd

Dec 31, 2022

CONTENTS:

CHAPTER

ONE

ABOUT

`pypsutil` is a partial reimplementation of the popular `psutil`. It is written in pure Python (when necessary, it calls library functions using `ctypes`).

CHAPTER

TWO

PYPSUTIL VS. PSUTIL

Reasons to use `pypsutil` instead of `psutil`:

- You do not want dependencies that require C extensions (for whatever reason)
- You need some of the extra features that `pypsutil` provides (such as *`Process.getgroups()`* and the increased availability of *`Process.rlimit()`*)
- You are using a type checker in your project (all of `pypsutil`'s public interfaces have type annotations, unlike `psutil`)

Reasons **not** to use `pypsutil` instead of `psutil`:

- You need to support Windows, Solaris, and/or AIX (`pypsutil` currently does not support these platforms)
- You need to support Python versions prior to 3.6 (`pypsutil` is Python 3.6+ only)
- You are concerned about speed (simple benchmarks have shown that `psutil` is faster, which is expected since it is partially written in C)
- You need professional support
- You need some of the features of `psutil` that `pypsutil` does not provide (check this documentation to see if the features you need are present)
- You want a drop-in replacement for `psutil` (see the note below)

`pypsutil` aims to implement many of the features of `psutil`; however, support is currently incomplete.

Important: `pypsutil` does **NOT** slavishly follow `psutil`'s API. Specifically:

- When possible, `pypsutil` avoids having a single function perform multiple different operations (mainly get vs. set) depending on the arguments it is passed. (Besides being better from a design perspective, this simplifies type annotations.)

  For example, `Process.nice()` (from `psutil`) is split into two methods in `pypsutil`: *`Process.getpriority()`* and *`Process.setpriority()`*. Similarly, `psutil`'s `cpu_times(percpu=True|False)` is split into two functions: *`cpu_times()`* and *`percpu_times()`*.

- If an interface has been added to the standard library that (at least partially) overlaps with an interface from `psutil`, `pypsutil` may either a) remove the interface entirely or b) remove the portion that overlaps with the standard library, possibly renaming it in the process.

  For example, `os.cpu_count()` was added in Python 3.4, and it retrieves the same information as `psutil.cpu_count(logical=True)`. As a result, `pypsutil` does not offer a `cpu_count()` function; instead, it offers a *`physical_cpu_count()`* function that covers the case of `psutil.cpu_count(logical=False)`.

CHAPTER

THREE

PLATFORM SUPPORT

Currently, the following platforms are supported:

- Linux
- macOS
- FreeBSD
- OpenBSD
- NetBSD

Not all platforms support all interfaces. Availability of different functions on different platforms is noted in the documentation.

Note that pypsutil is only regularly tested on x86_64 Linux, macOS, and FreeBSD. pypsutil *should* work on OpenBSD/NetBSD and on non-x86_64 architectures, but support is less reliable. (In particular, big-endian architectures may have trouble, especially with regards to networking APIs.)

CHAPTER

FOUR

PROCESS INFORMATION

class Process(*pid=None*)

Represents the process with the specified `pid`. (If `pid` is None, the PID of the current process is used.)

This class will retrieve the process's creation time and use the combination of PID + creation time to uniquely identify the process, helping to prevent bugs if the process exits and the PID is reused by the operating system.

pid

The process's PID. This attribute is read-only.

Type int

ppid()

Returns the parent process's PID.

Returns The PID of this process's parent process

Return type int

parent()

Returns a a *Process* object representing this process's parent process, or `None` if the parent process cannot be determined.

Note: This method preemptively checks if this process's PID has been reused.

Returns a *Process* object representing this process's parent process, or `None` if the parent process cannot be determined

Return type *Process* or None

parents()

Returns a list of this process's parents. This is a helper that is effectively equivalent to calling *parent()* repeatedly until it returns None.

Note: This method preemptively checks if this process's PID has been reused.

Returns a list of *Process* objects representing this process's parents

Return type list[*Process*]

children(*, *recursive=False*)

Get a list of the children of this process. If `recursive` is `True`, includes all descendants.

Parameters **recursive** (*bool*) – If `True`, include all descendants of this process's children

Returns a list of *Process* objects representing this process's children

Return type list[*Process*]

create_time()

Get the creation time of this process.

Returns The creation time of this process, in seconds since the Unix epoch

Return type float

`raw_create_time()`

> **Warning:** In nearly all cases, you want to use *create_time()* instead.

Get the "raw" creation time of this process. This is the value returned directly by the OS. For most intents and purposes, its value is completely meaningless.

The only guarantee made about this value is that two *Process* objects representing the same process will always have the same raw creation time. Any uses of this value beyond that are undefined behavior.

Returns The "raw" creation time returned directly by the OS.

Return type float

`pgid()`
Get the processs group ID of this process.

Returns The process group ID of this process

Return type int

`sid()`
Get the session ID of this process.

Returns The session ID of this process

Return type int

`status()`
Get the current process status as one of the members of the *ProcessStatus* enum.

Returns The current process status

Return type *ProcessStatus*

`name()`
Get the name of this process.

Returns The name of this process

Return type str

`exe`(*, *fallback_cmdline=True*)
Get the path to this process's executable.

On some platforms (such as OpenBSD) this cannot be obtained directly. On those platforms, if `fallback_cmdline` is `True`, this method will return the first command-line argument (if it is not an absolute path, a lookup will be performed on the system `PATH`).

If the path to the process's executable cannot be determined (for example, if the `PATH` lookup fails on OpenBSD), this function will return an empty string.

Parameters **fallback_cmdline** (*bool*) – Whether to fall back on checking the first command-line argument if the OS does not provide a way to get the executable path. (This is much less reliable.)

Returns The path to this process's executable

Return type str

`cmdline()`
A list of strings representing this process's command line.

Returns This process's command line as a list of strings

Return type list[str]

`cwd()`
Get this process's current working directory.

Returns This process's current working directory

Return type str

`root()`
Get this process's root directory.

An empty string is returned if this cannot be obtained.

Returns This process's root directory

Return type str

Availability: Linux, macOS, FreeBSD, NetBSD

`environ()`
Return this process's environmental variables as a dictionary of strings.

Note: This may not reflect changes since the process was started.

Returns This process's environment as a dict

Return type dict[str, str]

`uids()`
Get the real, effective, and saved UIDs of this process

Returns A tuple containing the UIDs of this process

Return type tuple[int, int, int]

`gids()`
Get the real, effective, and saved GIDs of this process

Returns A tuple containing the GIDs of this process.

Return type tuple[int, int, int]

`fsuid()`
Get the filesystem UID of this process (Linux-specific).

Returns The filesystem UID of this process

Return type int

Availability: Linux

`fsgid()`
Get the filesystem GID of this process (Linux-specific).

Returns The filesystem GID of this process

Return type int

Availability: Linux

`getgroups()`
Get the supplementary group list of this process.

Note: Currently, on Windows Subsystem for Linux 1 (not on WSL 2), this function succeeds but always returns an empty list.

Note: On macOS, this function's behavior differs from that of os.getgroups(). Effectively, it always behaves as if the deployment target is less than 10.5.

Returns A list of this process's supplementary group IDs.

Return type list[int]

`username()`
Get the username of the user this process is running as.

Currently, this just takes the real UID and uses `pwd.getpwuid()` to look up the username. If that fails, it converts the real UID to a string and returns that.

Returns The username of the user this process is running as

Return type str

`umask()`
Get the umask of this process.

Returns `None` if it is not possible to get the umask on the current version of the operating system.

Note: On FreeBSD, this will raise *AccessDenied* for PID 0.

Returns The umask of this process

Return type int or None

Availability: Linux (4.7+), FreeBSD

`sigmasks`(*, *include_internal=False*)
Get the signal masks of this process.

This returns a dataclass with several attributes:

- `pending` (not on macOS): The signals that are pending for this process.
- `blocked` (not on macOS): The signals that are blocked for this process (i.e. the signal mask set with `pthread_sigmask()`).
- `ignored`: The signals that are ignored by this process (i.e. `SIG_IGN`).
- `caught`: The signals for which this process has registered signal handlers.
- `process_pending` (Linux-only): The signals that are pending for the process as a whole, not just this thread.

All of these are `set` objects.

Note: Currently, on Windows Subsystem for Linux 1 (not on WSL 2), this function succeeds but always returns empty sets for all fields.

Parameters `include_internal` (*bool*) – If this is `True`, then implementation-internal signals may be included – for example, on Linux this affects the two or three signals used by the glibc/musl POSIX threads implementations.

Returns The signal masks of this process.

Return type ProcessSignalMasks

`cpu_times()`

Get the accumulated process times.

This returns a dataclass with several attributes:

- `user`: Time spent in user mode
- `system`: Time spent in kernel mode
- `children_user`: Time spent in user mode by child processes (0 on macOS)
- `children_system`: Time spent in kernel mode (0 on macOS)

Note: On OpenBSD and NetBSD, `children_user` and `children_system` are both set to the combined user + system time.

Returns The accumulated process times

Return type ProcessCPUTimes

`memory_info()`

Return a dataclass containing information on the process's memory usage. Some attributes:

- `rss`: Non-swapped physical memory the process is using.
- `vms`: Total amount of virtual memory used by the process.
- `shared` (Linux): The amount of memory used in `tmpfs`-es.
- `text` (Linux, *BSD): The amount of memory used by executable code.
- `data` (Linux, *BSD): The amount of memory used by things other than executable code.
- `stack` (*BSD): The amount of memory used by the stack.
- `pfaults` (macOS): The number of page faults.
- `pageins` (macOS): The number of pageins.

Returns A dataclass containing information on the process's memory usage

Return type ProcessMemoryInfo

`memory_percent(`*memtype='rss'*`)`

Compare system-wide memory usage to the total system memory and return a process utilization percentage.

Returns The percent of system memory that is being used by the process as the given memory type.

Return type float

`memory_maps()`

Retrieve information on this process's memory maps.

> **Warning:** Unlike `psutil`'s `memory_maps()`, this method does NOT "group" mappings from the same file! See *memory_maps_grouped()* if that is what you want.

This returns a list of dataclasses, each of which represents a memory map.

The following metadata fields may included (there are slight variations across operating systems):

- `path` (not on OpenBSD): The path to the mapped file. If the mapping does not correspond to a file, this is a string describing the mapping, such as `[heap]`.
- `addr_start`: The start address of the mapping.
- `addr_end`: The start address of the mapping.
- `perms`: A string describing the permission set of the mapping. On Linux this is e.g. `rwxp` or `-w-s`; on other platforms it is e.g. `rw-` or `r-x`.
- `offset`: The offset into the file at which the mapping starts.
- `dev` (not on OpenBSD): The device number of the mapped file (or 0 if N/A).
- `ino` (not on OpenBSD): The device number of the mapped file (or 0 if N/A).
- `size`: The size of the mapping.

The fields containing statistics on the mapping vary wildly across operating systems:

Linux	FreeBSD	OpenBSD
rss	rss	wired_count
pss	private	
shared_clean	ref_count	
shared_dirty	shadow_count	
private_clean		
private_dirty		
referenced		
anonymous		
swap		

The following attributes

Returns A list of dataclasses containing information on the process's memory maps

Return type list[ProcessMemoryMap]

Availability: Linux, FreeBSD, OpenBSD

memory_maps_grouped()

Retrieve information on this process's memory maps, "grouping" mappings from the same file and summing the fields.

This is not available on systems where *memory_maps()* does not support retrieving the path of the mapped file, since it's impossible to group the mappings properly in that case.

The returned structures have all of the same fields as the structures returned by *memory_maps()*, **except** for `addr_start`, `addr_end`, `perms`, and `offset` (since they are mapping-specific).

Returns A list of dataclasses containing information on the process's "grouped" memory maps

Return type list[ProcessMemoryMapGrouped]

Availability: Linux, FreeBSD

rlimit(*res*, *new_limits=None*)

Get/set the soft/hard resource limits of the process. Equivalent to `resource.prlimit(proc.pid, res, new_limits)`, but may be implemented on more platforms.

In addition, if this method is used to *set* the resource limits, it preemptively checks for PID reuse.

> **Warning:** On some platforms, this method may not be able to get/set the limits atomically, or to set the soft and hard resource limits together.
>
> Aside from the potential race conditions this creates, if this method raises an error, one or both of the limits may have been changed before the error occurred. In this case, no attempts are made to revert the changes.

Note: A special boolean attribute, `is_atomic`, is set on this method. It is `True` if the implementation of *rlimit()* is able to get/set the soft/hard limits atomically, and is not vulnerable to the issues described above.

Parameters

- **res** (*int*) – The number of the resource to set (one of the `resource.RLIMIT_*` constants)
- **new_limits** (*tuple[int, int] or None*) – The new `(soft, hard)` limits to set (or `None` to only get the old resource limits). Use `resource.RLIM_INFINITY` for infinite limits.

Returns A tuple of the old `(soft, hard)` resource limits

Return type tuple[int, int]

Availability: Linux, FreeBSD, NetBSD

getrlimit(*res*)

Get the current soft/hard resource limits of the process. Equivalent to `resource.prlimit(proc.pid, res)`, but may be implemented on more platforms.

Currently, the availability of this method is the same as for *rlimit()*. However, that may change if `pypsutil` adds supports for platforms that allow for getting, but not setting, resource limits for other processes.

Note: As with *rlimit()*, this method may not be able to get the soft and hard resource limits together. As a result, there is a race condition: If the process's resource limits are changed while this method is reading them, this method may return a combination such as `(old soft, new hard)` or `(new soft, old hard)` (where "old" means the values before the change and "new" means the values after the change).

To aid in detection, this method has an `is_atomic` attribute similar to the one set on *rlimit()*.

Parameters **res** (*int*) – The number of the resource to set (one of the `resource.RLIMIT_*` constants)

Returns A tuple of the current `(soft, hard)` resource limits

Return type tuple[int, int]

Availability: Linux, FreeBSD, NetBSD

`has_terminal()`
Check whether this process has a controlling terminal. `proc.has_terminal()` is exactly equivalent to `proc.terminal() is not None`, but it is more efficient if you don't need the name of the terminal (just whether or not the process has one).

Note: See the note on *terminal()* for an explanation of how this differs from `[ -t 0 ]`, `tty -s`, or `isatty(0)`.

Returns Whether this process has a controlling terminal

Return type bool

`terminal()`
Get the name of this process's controlling terminal. Returns `None` if the process has no controlling terminal, or an empty string if the process has a controlling terminal but its name cannot be found.

Note: Usually, the name returned by this function will be the same as with the `tty` command or `ttyname(0)`. However, this function returns the name of the process's *controlling terminal*, while `tty` and `ttyname(0)` return the name of *the terminal connected to standard input* (if the process's standard input is a terminal).

In most cases, these will be the same thing. However, they are not technically *required* to be, and in some edge cases they may be different.

Returns The name of this process's controlling terminal

Return type str or None

`cpu_num()`
Get number of the CPU this process is running on (or was last running on if it is not currently running). Note that the CPU number can change at any time, so the returned value may already be outdated.

This will return -1 if the CPU number cannot be determined (for example, on FreeBSD with certain kernel processes).

Returns The number of the CPU this process is running on (or was last running on)

Return type int

Availability: Linux, FreeBSD, OpenBSD, NetBSD

`cpu_getaffinity()`
Get the CPU affinity of this process.

On Linux, this is equivalent to `os.sched_getaffinity(proc.pid)` (see os.sched_getaffinity() for more information). However, it may support other platforms which have APIs that allow similar functionality.

Returns The set of CPUs this process is eligible to run on

Return type set[int]

Availablity: Linux, FreeBSD

`cpu_setaffinity(`*cpus*`)`
Set the CPU affinity of this process.

On Linux, this is equivalent to `os.sched_setaffinity(proc.pid, cpus)` (see os.sched_setaffinity() for more information). However, it may support other platforms which have APIs that allow similar functionality. It also preemptively checks for PID reuse.

If the `cpus` list is empty, this method will reset the CPU affinity to the set of all available CPUs which this process can run on.

Parameters **cpus** (*iterable[int]*) – The new set of CPUs that the process should be eligible to run on

Availablity: Linux, FreeBSD

getpriority()

Equivalent to `os.getpriority(os.PRIO_PROCESS, proc.pid)` in most cases. (However, on systems where the kernel appears as PID 0, `Process(0).getpriority()` will actually operate on PID 0.)

Returns The process's scheduling priority (a.k.a. nice value)

Return type int

setpriority(*prio*)

Equivalent to `os.setpriority(os.PRIO_PROCESS, proc.pid, prio)`, but preemptively checks for PID reuse.

(Note: on systems where the kernel appears as PID 0, attempting to set the priority of PID 0 will always fail with an *AccessDenied* exception.)

Parameters **prio** (*int*) – The new scheduling priority (a.k.a. nice value) for the process

send_signal(*sig*)

Send the specified signal to this process, preemptively checking for PID reuse.

Other than the PID reuse check, this is equivalent to `os.kill(proc.pid, sig)`.

Parameters **sig** (*int*) – The signal number (one of the `signal.SIG*` constants)

suspend()

Suspends process execution, preemptively checking for PID reuse. Equivalent to `proc.send_signal(signal.SIGSTOP)`.

resume()

Resumes process execution, preemptively checking for PID reuse. Equivalent to `proc.send_signal(signal.SIGCONT)`.

terminate()

Terminates the process, preemptively checking for PID reuse. Equivalent to `proc.send_signal(signal.SIGTERM)`.

kill()

Kills the process, preemptively checking for PID reuse. Equivalent to `proc.send_signal(signal.SIGKILL)`.

num_threads()

The number of threads in this process (including the main thread).

Returns The number of threads in this process

Return type int

threads()

Returns a list of *ThreadInfo* structures with information on the threads in this process.

Returns A list of *ThreadInfo* structures with information on this process's threads

Return type list[*ThreadInfo*]

`num_fds`()
Get the number of file descriptors this process has open.

Returns The number of file descriptors this process has open

Return type int

`open_files`()
Return a list of dataclasses containing information on all the regular files this process has open. Each entry has the following attributes.

- `path`: The absolute path to the file.
- `fd`: The file descriptor number.
- `position` (Linux-only): The current seek position.
- `flags` (Linux-only): The flags passed to the underlying `open()` C call.
- `mode` (Linux-only): A string, derived from `flags`, that approximates the likely `mode` argument as for `open()`. Possible values are `"r"`, `"w"`, `"a"`, `"r+"`, `"a+"`.

Returns A list of dataclasses containing information on all the regular files this process has open

Return type list[ProcessOpenFile]

`iter_fds`()
Return an iterator that yields a series of *ProcessFd* s containing information on all the file descriptors this process has open.

Returns An iterator yielding *ProcessFd* s

Return type iter[*ProcessFd*]

`connections`(*kind='inet'*)
Return a list of *Connection* s representing sockets opened by this process.

`kind` specifies which sockets should be returned:

- `"inet"`: All inet sockets (TCP/UDP, IPv4/IPv6)
- `"inet4"`: All IPv4 sockets (TCP/UDP)
- `"inet6"`: All IPv6 sockets (TCP/UDP)
- `"tcp"`: All TCP sockets (IPv4/IPv6)
- `"tcp4"`: All TCP, IPv4 sockets
- `"tcp6"`: All TCP, IPv6 sockets
- `"udp"`: All UDP sockets (IPv4/IPv6)
- `"udp4"`: All UDP, IPv4 sockets
- `"udp6"`: All UDP, IPv6 sockets
- `"unix"`: All Unix sockets
- `"all"`: All of the above

On some OSes, root privileges may be required to list connection information for processes created by other users:

- On Linux and macOS, root privileges **ARE** required to list connection information for these processes.

- On FreeBSD, root privileges are **NOT** required to list connection information for these processes, but having root privileges allows this method to use a much more efficient implementation (~5x faster).
- On OpenBSD and NetBSD, root privileges are **NOT** required to list connection information for these processes.

Returns A list of *Connection* s

Return type list[*Connection*]

Availability: Linux, macOS, FreeBSD, OpenBSD, NetBSD

`is_running`()

Checks if the process is still running. Unlike `pid_exists(proc.pid)`, this also checks for PID reuse.

Note: The following methods preemptively check whether the process is still running and raise *NoSuchProcess* if it has exited:

- *parent()*
- *parents()*
- *children()*
- *rlimit()* (when setting limits)
- *setpriority()*
- *send_signal()*
- *suspend()*
- *resume()*
- *terminate()*
- *kill()*

Returns Whether the process is still running

Return type int

`wait`(*, *timeout=None*)

Wait for the process to exit. If this process was a child of the current process, its exit code is returned.

Raises *TimeoutExpired* if the timeout expires.

Parameters **`timeout`** (*int or float or None*) – The maximum amount of time to wait for the process to exit (`None` signifies no limit).

Raises ***TimeoutExpired*** – If the timeout expires

Returns The exit code of the process if it can be determined; otherwise `None`

Return type int or None

`oneshot`()

This is a context manager which enables caching pieces of information that can be obtained via the same method.

Here is a table, in the same format as psutil.Process.oneshot()'s table, that shows which methods can be grouped together for greater efficiency:

Linux	macOS	FreeBSD/OpenBSD/NetBSD
name()[1]	*name()*	*name()*
status()	*status()*	*status()*
ppid()	*ppid()*	*ppid()*
terminal()[2]	*pgid()*?	*pgid()*?
has_terminal()	*uids()*	*sid()*?
cpu_times()	*gids()*	*uids()*
cpu_num()	*username()*Page 18, 2	*gids()*
num_threads()?	*getgroups()*	*username()*Page 18, 2
	*terminal()*Page 18, 2	*getgroups()*[3]
uids()	*has_terminal()*	*terminal()*?
gids()	*sigmasks()*	*has_terminal()*
username()?		*sigmasks()*
getgroups()	*cmdline()*	*cpu_times()*
umask()	*environ()*	*memory_info()*
sigmasks()		*cpu_num()*
num_ctx_switches()	*cpu_times()*	num_ctx_switches()
	memory_info()	*num_threads()* (on FreeBSD)
	num_threads()	
	num_ctx_switches()	
	cwd()	
	root()	

class ProcessStatus

An enum representing a process's status.

RUNNING

SLEEPING

DISK_SLEEP

ZOMBIE

STOPPED

TRACING_STOP

DEAD

WAKE_KILL

WAKING

PARKED

IDLE

LOCKED

WAITING

[1] These functions, when called inside a *oneshot()* context manager, will retrieve the requested information in a different way that collects as much extra information as possible about the process for later use.

[2] *terminal()* and *username()* have to do additional processing after retrieving the cached information, so they will likely only see minor speedups.

[3] On FreeBSD, calling *getgroups()* inside a *oneshot()* will first attempt to retrieve the group list via a method that collects as much extra information as possible. However, this method may truncate the returned group list. In this case, *getgroups()* will fall back on the normal method, which avoids truncation.

SUSPENDED

class ThreadInfo

A dataclass containing information on the threads in this process.

id

type int

The thread ID.

user_time

type float

The time this thread spent in user mode.

system_time

type float

The time this thread spent in system mode.

class ProcessFd

Note: Availability of `dev`, `ino`, and `rdev`, `size`, and `mode`:

- Always set on Linux (though they are meaningless for some file types)
- Available on NetBSD for *ProcessFdType.FILE* s and *ProcessFdType.FIFO* s if procfs is mounted
- Availability on macOS/FreeBSD/OpenBSD varies depending on the file type. They are usually only available for *ProcessFdType.FILE* s.

Warning: On macOS, `position` and `flags` may be -1 for some special file descriptors.

path

type str

The path to the file. This will be empty if this cannot be retrieved, or if it is not meaningful for the file type in question.

Note that this is always empty on OpenBSD, and on NetBSD it is only set for directories.

fd

type int

The file descriptor number.

fdtype

Type *ProcessFdType*

The file descriptor type (one of the values in the *ProcessFdType* enum).

dev

type int or None

The device ID of the device containing the file, or `None` if not available.

ino

type int or None

The inode of the file, or `None` if not available.

rdev

type int or None

The device ID of the file (if it is a special file that represents a device) or `None` if not available.

mode

type int or None

The mode of the file (i.e. `st_mode`), or `None` if not available. For *ProcessFdType.FILE* s, this can be used to get the file type.

size

type int or None

The size of the file, or `None` if not available.

On NetBSD, this is only set for type *ProcessFdType.FILE* s.

position

type int

The current offset of the file.

flags

type int

The flags passed to the underlying `open()` C call.

This will include `O_CLOEXEC` if the file descriptor's close-on-exec flag is set (except on FreeBSD).

On Linux, this will never include the `O_LARGEFILE` flag.

On macOS/*BSD, only the following flags will be preserved: - `O_RDONLY` - `O_WRONLY` - `O_RDWR` - `O_APPEND` - `O_ASYNC` - `O_FSYNC` - `O_NONBLOCK` - `O_EVTONLY` (macOS) - `O_DSYNC` (macOS, FreeBSD, NetBSD) - `O_DIRECT` (FreeBSD, NetBSD) - `O_EXEC` (FreeBSD) - `O_PATH` (FreeBSD 14+) - `O_RSYNC` (NetBSD) - `O_ALT_IO` (NetBSD) - `O_NOSIGPIPE` (NetBSD)

open_mode

type str

A string, derived from `flags`, that approximates the likely `mode` argument as for `open()`. Possible values are `"r"`, `"w"`, `"a"`, `"r+"`,

extra_info

type dict[str, Any]

A dictionary containing extra information about the file descriptor.

The contents of this dictionary are highly dependent on the OS and the file descriptor type. See *ProcessFdType* for more information on the data that may be stored in here for different file descriptor types. (On Linux, they generally correspond to fields in `/proc/$PID/fdinfo/$FD`.)

class ProcessFdType

An enum representing the possible file types that can be returned for a *ProcessFd*.

Note that all of the variants here are added on all OSes. For example, *ProcessFdType.KQUEUE* is available on Linux, though it is BSD/macOS-specific. It will simply never actually be returned.

Additionally, new variants may be added to this enum at any time. Treat unrecognized variants as equivalent to *ProcessFdType.UNKNOWN*.

FILE
A file on the disk. Note that this doesn't necessarily mean a *regular* file, it could still be a directory, block device, etc.

On Linux, macOS, and OpenBSD, *ProcessFd.extra_info* may contain an `nlink` field specifying the number of links to the file.

SOCKET
A socket. Note that for Unix sockets, whether or not the path is returned in *ProcessFd.path* is platform-dependent.

On macOS, FreeBSD, and OpenBSD, *ProcessFd.extra_info* will contain `domain`, `type`, `protocol`, `recvq`, and `sendq` fields indicating these attributes of the socket.

PIPE
A pipe.

On macOS and FreeBSD, *ProcessFd.extra_info* will contain a `buffer_cnt` field specifying the amount of data in the pipe's read buffer. (Note that since pipes are bidirectional on FreeBSD, this means the amount of data waiting to be read by *this* end of the pipe.)

On macOS, *ProcessFd.extra_info* will also contain a `buffer_max` field specifying the size of the pipe's read buffer.

FIFO
A named pipe.

KQUEUE
(macOS/*BSD) A kqueue instance.

On macOS and OpenBSD, *ProcessFd.extra_info* will contain a `kq_count` field indicating the number of pending events on the kqueue instance.

PROCDESC
(FreeBSD) A process descriptor.

On FreeBSD 12+, *ProcessFd.extra_info* will contain a `pid` field containing the PID of the process referred to by this process descriptor.

INOTIFY
(Linux) An inotify instance.

SIGNALFD
(Linux) A signalfd instance.

On Linux 3.8+, *ProcessFd.extra_info* will contain a `sigmask` field with a set containing the signals that this signalfd instance is monitoring.

EPOLL
(Linux) An epoll instance.

On Linux 3.8+, *ProcessFd.extra_info* will contain a `tfds` field with a dictionary of information on the file descriptors being monitored by this epoll instance.

TIMERFD
(Linux) A timerfd instance.

On Linux 3.17+, *ProcessFd.extra_info* will contain the following extra fields:

- `clockid`: The clock ID used to mark the progress of the timer; e.g. `time.CLOCK_REALTIME`.

- `ticks`: The number of timer expirations that have occurred.
- `settime_flags`: The integer flags that were last used to arm the timer using `timerfd_settime()`.
- `it_value` The amount of time until the timer exipres (in seconds, represented as a float).
- `it_value_ns`: The same value as `it_value`, but in nanoseconds.
- `it_interval`: The interval of the timer (in seconds, represented as a float).
- `it_interval_ns`: The same value as `it_interval`, but in nanoseconds.

See `timerfd_settime(2)` and `timerfd_gettime(2)` for more information.

PIDFD
(Linux) An PID file descriptor.

ProcessFd.extra_info will contain a `pid` field containing the PID of the process referred to by this process descriptor.

EVENTFD
(Linux/FreeBSD) An eventfd.

On Linux 3.8+, *ProcessFd.extra_info* will contain an `eventfd-count` field with the current value of the eventfd's counter.

On FreeBSD, *ProcessFd.extra_info* will contain an `eventfd_value` field with the current value of the eventfd's counter, and an `eventfd_flags` field with the flags used when creating the eventfd.

UNKNOWN
An unknown file type.

On macOS, `flags` and `position` may be -1 for `UNKNOWN` files.

class Connection
A dataclass representing a network connection.

family

Type int

The address family; one of `socket.AF_INET`, `socket.AF_INET6`, or `socket.AF_UNIX`.

type

Type int

The address type; one of `socket.SOCK_STREAM`, `socket.SOCK_DGRAM`, or `socket.SOCK_SEQPACKET`.

laddr

Type tuple[str, int] or str

The local address. For inet sockets this is in the format `(ip, port)` (it's `("", 0)` if the socket is not connected or the address cannot be determined). For Unix sockets this is a string path, or `""` if the socket is not connected or the address cannot be determined.

raddr

Type tuple[str, int] or str

The remote address, in the same format as `laddr`.

Note: On Linux and OpenBSD, this is always `""` for Unix sockets.

---

status

Type *ConnectionStatus* or None

The TCP connection status (for inet TCP sockets only).

fd

Type int

The socket file descriptor number. This is -1 if N/A or it cannot be determined.

pid

Type int or None

The PID of the process that created the socket. This is `None` if it cannot be determined.

class ConnectionStatus

An enum representing a TCP connection status.

ESTABLISHED

SYN_SENT

SYN_RECV

FIN_WAIT1

FIN_WAIT2

TIME_WAIT

CLOSE

CLOSE_WAIT

LAST_ACK

LISTEN

CLOSING

pids()

Get a list of the PIDs of running processes.

Returns A list of the PIDs of running processes.

Return type list[int]

process_iter()

Return an iterator over the running processes.

Note: On Linux, if `/proc` is mounted with `hidepid=1`, it is not possible to get the process creation time (or indeed, any information except the PID/UID/GID of the process) of other users' processes when running an unprivileged user. This function will raise a *AccessDenied* if that occurs; if you wish to simply skip these processes then use *process_iter_available()* instead.

Return type iterator[*Process*]

process_iter_available()

Return an iterator over the running processes, except that a process will be skipped if a permission error is encountered while trying to retrieve its creation time. (This can happen, for example, on Linux if `/proc` is mounted with `hidepid=1`.)

Return type iterator[*Process*]

pid_exists(*pid*)

Checks whether a process with the given PID exists. This function will use the most efficient method possible to perform this check.

Note: Use *Process.is_running()* if you want to check whether a *Process* has exited.

Parameters **pid** (*int*) – The PID to check for existence

Returns Whether the process with the given PID exists

Return type bool

wait_procs(*procs*, *timeout=None*, *callback=None*)

Wait for several *Process* instances to terminate, and returns a `(gone, alive)` tuple indicating which have terminated and which are still alive.

As each process terminates, a `returncode` attribute will be set on it. If the process was a child of the current process, this will be set to the return code of the process; otherwise, it will be set to `None`.

If `callback` is not `None`, it should be a function that will be called as each process terminates (after the `returncode` atribute is set).

If the `timeout` expires, this function will not raise *TimeoutExpired*; it will simply return the current `(gone, alive)` tuple of processes.

Parameters

- **procs** (*iterable[int]*) – The processes that should be waited for
- **timeout** (*int or float or None*) – The maximum amount of time to wait for the processes to terminate
- **callback** – A function which will be called with the *Process* as an argument when one of the processes exits.

Return type tuple[list[*Process*], list[*Process*]]

CHAPTER

FIVE

SYSTEM INFORMATION

boot_time()
Get the system boot time as a number of seconds since the Unix epoch.

Returns The system boot time as a number of seconds since the Unix epoch

Return type float

time_since_boot()
Get the number of seconds elapsed since the system booted.

Usually, this is approximately the same as `time.time() - pypsutil.boot_time()`. However, it may be more efficient. (On some systems, *boot_time()* may just return `time.time() - pypsutil.time_since_boot()`!)

Returns The number of seconds elapsed since the system booted

Return type float

uptime()
Get the system uptime. This is similar to `time_since_boot()`, but it does not count time the system was suspended.

Returns The system uptime

Return type float

Availability: Linux, macOS, FreeBSD, OpenBSD

virtual_memory()
Return a dataclass containing system memory statistics. Currently, the following fields are available:

- `total`: Total physical memory in bytes
- `available`: Amount of memory that can be made available without using swap (or killing programs)
- `used`: Used memory in bytes
- `free`: Free memory (immediately available) in bytes
- `active`: Memory currently in use or recently used (unlikely to be reclaimed)
- `inactive`: Memory not recently used (more likely to be reclaimed)
- `buffers`: Temporary disk storage
- `cached`: Disk cache
- `shared`: Memory used for shared objects (`tmpfs`-es on Linux)
- `slab`: In-kernel data structure cache

The dataclass also has a `percent` property that returns the usage percentage (0-100).

In most cases, you should only use `total`, `available` and `percent`.

Returns A dataclass containing system memory statistics

Return type VirtualMemoryInfo

`swap_memory()`
Return a dataclass containing system swap memory statistics. Currently, the following fields are available:

- `total`: Total swap memory in bytes
- `used`: Used swap memory in bytes
- `free`: Free swap memory in bytes
- `sin`: Cumulative number of bytes the system has swapped in from the disk
- `sout`: Cumulative number of bytes the system has swapped out from the disk

The dataclass also has a `percent` property that returns the usage percentage (0-100).

Returns A dataclass containing system swap memory statistics

Return type SwapInfo

`disk_usage(`*path*`)`
Return disk usage statistics about the filesystem which contains the given `path`.

Current attributes:

- `total`: Total disk space in bytes
- `used`: Total used disk space in bytes
- `free`: Disk space in bytes that is free and available for use by unprivileged users
- `percent`: Percentage of disk space used (out of the space available to unprivileged users)

Returns A dataclass containing disk usage statistics about the filesystem which contains the given `path`

Return type DiskUsage

`physical_cpu_count()`
Get the number of physical CPUs in the system (i.e. excluding Hyper Threading cores) or `None` if that cannot be determined.

Currently, this always returns `None` on OpenBSD and NetBSD.

Returns The number of physical CPUs in the system (or `None` if unable to determine)

Return type int or None

`cpu_freq()`
Returns an instance of a dataclass with `current`, `min`, and `max` attributes, representing the current, minimum, and maximum CPU frequencies.

If the frequencies cannot be determined, returns `None`. If only the current frequency can be determined, `min` and `max` will be returned as `0.0`.

Returns An instance of a dataclass containing the current, minimum, and maximum CPU frequencies.

Return type CPUFrequencies or None

Availability: Linux, macOS, FreeBSD, OpenBSD

percpu_freq()
Identical to *cpu_freq()*, but returns a list representing the frequencies for each CPU.

If the frequencies cannot be determined, returns an empty list.

Returns A list of the frequencies of each CPU.

Return type list[CPUFrequencies]

Availability: Linux, FreeBSD

cpu_times()
Returns a dataclass containing information about system CPU times. Each attribute represents the time in seconds that the CPU has spent in the corresponding mode (since boot):

- `user`: Time spent in user mode (includes `guest` time on Linux)
- `system`: Time spent in kernel mode
- `idle`: Time spent doing nothing

Extra platform-specific fields:

- `nice` (Linux/BSDs/macOS): Time spent by prioritized processes in user mode (includes `guest_nice` time on Linux)
- `iowait` (Linux): Time spent waiting for I/O to complete
- `irq` (Linux/BSDs): Time spent servicing hardware interrupts
- `softirq` (Linux): Time spent servicing software interrupts
- `lock_spin` (OpenBSD): Time spent "spinning" on a lock
- `steal` (Linux): Time spent running other operating systems in a virtualized environment
- `guest` (Linux): Time spent running a virtual CPU for a guest operating system
- `guest_nice` (Linux): Time spent running a niced guest

Returns A dataclass containing information about system CPU times.

Return type CPUTimes

percpu_times()
Identical to *cpu_times()*, but returns a list representing the times for each CPU.

Returns A list of the times of each CPU.

Return type list[CPUTimes]

cpu_stats()
Return a dataclass containing various statistics:

- `ctx_switches`: The number of context switches since boot.
- `interrupts`: The number of interrupts since boot.
- `soft_interrupts`: The number of software interrupts since boot.
- `syscalls`: The number of system calls since boot (always 0 on Linux)

Returns A dataclass containing some CPU statistics.

Return type CPUStats

Availablity: Linux, FreeBSD, OpenBSD, NetBSD

net_connections(*kind='inet'*)

Return a list of *Connection* s representing all sockets opened system-wide.

See *Process.connections()* for information on the possible values of `kind`.

On Linux and macOS, this function will succeed if not running as root, but `pid` and `fd` cannot be determined for *Connection* s opened by other users (they will be `None` and `-1`, respectively.)

Returns A list of *Connection* s

Return type list[*Connection*]

Availability: Linux, macOS, FreeBSD, OpenBSD, NetBSD

net_io_counters(*nowrap=True*)

Returns a dataclass containing various network I/O statisitcs:

- `bytes_sent`: the number of bytes sent
- `bytes_recv`: the number of bytes received
- `packets_sent`: the number of packets sent
- `packets_recv`: the number of packets received
- `errin`: the number of errors encountered while receiving
- `errout`: the number of errors encountered while sending
- `dropin`: the number of dropped incoming packets
- `dropout`: the number of dropped outgoing packets

Note that some of these counts may overflow on long-running systems. Unlike `psutil`, `pypsutil` currently has no methods to work around such overflows.

If the system has no network interfaces, returns `None`.

If `nowrap` is `True`, pypsutil will attempt to detect if these counters overflow and wrap around to 0, and then adjust them to ensure that the reported values never decrease.

Parameters **nowrap** (*bool*) – Whether to adjust the counters across calls to ensure that they always increase or stay the same.

Returns A dataclass containg network I/O statistics

Return type NetIOCounts or None

Availability: Linux, FreeBSD

pernic_net_io_counters(*nowrap=True*)

Identical to *net_io_counters()*, but returns a dictionary mapping interface names to network I/O statistics.

If the system has no network interfaces, returns an empty dict.

Parameters **nowrap** (*bool*) – Whether to adjust the counters across calls to ensure that they always increase or stay the same.

Returns A dictionary mapping interface names to `NetIOCounts` s

Return type dict[str, NetIOCounts]

Availability: Linux, FreeBSD

net_if_addrs()
Return a dictionary mapping interface names to addresses.

Each key is the name of an interface, and each value is a list of dataclasses representing addresses, each of which have the following attributes:

- family: The address family. This is AF_LINK for MAC addresses, socket.AF_INET for IPv4 addresses, and socket.AF_INET for IPv6 addresses.
- address: A string representation of the IP/MAC address.
- netmask: A string representation of the corresponding netmask (or None).
- broadcast: A string representation of the broadcast address (or None).
- ptp: The destination address for point-to-point interfaces (or None).

ptp and broadcast are mutually exclusive.

Returns A dictionary mapping interface names to lists of NICAddr s

Return type dict[str, list[NICAddr]]

Availability: Linux, FreeBSD

net_if_stats()
Return information about each network interface.

Each key is the name of an interface, and each value is a dataclass which has the following attributes:

- isup: Whether or not the NIC is up and running.
- duplex: The NICs' duplex status. This is one of NICDuplex.FULL, NICDuplex.HALF, or NICDuplex.UNKNOWN.
- speed: The NIC's speed in MB/s, or 0 if unknown.
- mtu: The NIC's maximum transmission unit, in bytes.

Returns A dictionary mapping interface names to lists of NICStats s

Return type dict[str, list[NICStats]]

Availability: Linux

CHAPTER

SIX

SENSOR INFORMATION

sensors_power()

Get information on power supplies connected to the current system.

This returns a dataclass with the following attributes:

- batteries: A list of BatteryInfo objects representing any batteries connected to the current system.
- ac_supplies: A list of ACPowerInfo objects representing any mains power supplies connected to the current system.
- is_on_ac_power: True if the system is on AC power, False if it is not, and None if this cannot be determined

ACPowerInfo objects have the following attributes:

- name: A semi-meaningless name.
- is_online: Whether the power supply is online.

BatteryInfo objects have the following attributes:

- name: A semi-meaningless name (should be unique between batteries, but may change if one battery is unplugged in a multi-battery system).
- status: One of the elements of the BatteryStatus enum (listed below) indicating the current battery status.
- power_plugged: This is True if it can be confirmed that AC power is connected, False if it can be confirmed that AC power is disconnected, and None if it cannot be determined. This is provided for compatibility with psutil; it is recommended to use status instead for most cases. *sensors_power()* will only set this to a value other than None if the battery is either charging or discharging; other sensor information functions may set this based on the AC adapter status.
- percent: The percentage capacity of the battery, as a floating point number,
- energy_full: The amount of energy the battery normally contains when full, in uWh (or None if not available).
- energy_now: The amount of energy the battery currently holds, in uWh (or None if not available).
- power_now: The amount of power currently flowing into or out of the battery (this value is always positive; check whether the battery is charging or discharging to determine the direction) in uW or None if not available.
- secsleft: The number of seconds left until the battery is empty. If the battery is either charging or full, this is float("inf"); if the information cannot be determined (or the battery is in the "unknown" state) it is None.

- `secsleft_full`: The number of seconds left until the battery is full. If the battery is full, this is 0; if the the information cannot be determined (or the battery is in the "unknown" or "discharging" states) it is `None`.
- `temperature`: The temperature (in Celsius) of the battery, or `None` if unknown.
- `temperature_fahrenheit`: The temperature (in Fahrenheit) of the battery, or `None` if unknown.

The elements of the `BatteryStatus` enum are as follows:

- `CHARGING`: The battery is actively charging.
- `DISCHARGING`: The battery is actively discharging.
- `FULL`: The battery is at 100% capacity and neither charging nor discharging.
- `UNKNOWN`: The battery state is unknown.

Returns Information on power supplies connected to the current system.

Return type PowerSupplySensorInfo or None

Availability: Linux, FreeBSD

sensors_is_on_ac_power()

Detect whether the system is on AC power.

This is equivalent to `sensors_power().is_on_ac_power` (except that it returns `None` if *sensors_power()* would return `None`) but it may be more efficient.

In some cases, it may also succeed if `sensors_power()` would return `None`.

Returns True if the computer is on AC power, False if it is not, and None if this cannot be determined.

Return type bool or None

Availability: Linux, FreeBSD

sensors_battery()

Return battery status information (or `None` if no battery is installed).

Internally, this just calls *sensors_power()*, extracts the first battery's information, and then sets `battery.power_plugged` based on the `is_on_ac_power` attribute of the dataclass returned by *sensors_power()*. If that fails, it may fall back on methods that will return the same results as for *sensors_battery_total()*.

Essentially, this function says "let's assume the system has at most one battery, and return results based on that." On systems that may have more than one battery, you should use *sensors_power()* or *sensors_battery_total()* instead.

Returns Battery information

Return type BatteryInfo or None

Availability: Linux, FreeBSD

sensors_battery_total()

Collect system-wide battery information.

If the system has only one battery (or no batteries), this should be roughly equivalent to *sensors_battery()*. If the system has more than one battery, this function will return totaled statistics for all batteries.

It also sets the `power_plugged` attribute similarly to how *sensors_battery()* does it.

Returns Totaled battery information

Return type BatteryInfo or None

Availability: Linux, FreeBSD

CHAPTER

SEVEN

PATH CONSTANTS THAT MODIFY BEHAVIOR

These constants can be set to modify `pypsutil`'s behavior; e.g. partially examine the state of a Docker container on Linux.

PROCFS_PATH
The path at which a procfs is mounted (if applicable). Defaults to `/proc`.

This attribute is set and can be modified even on systems which do not have a procfs; on those systems it is ignored.

Currently, it is only used on Linux and NetBSD.

DEVFS_PATH
The path at which a `/dev`-like file layout (e.g. `tty*`) can be found.

SYSFS_PATH
On Linux, the path at which a sysfs is mounted. Defaults to `/sys`.

Availability: Linux

CHAPTER

EIGHT

EXCEPTIONS

class Error
Base exception class

class NoSuchProcess(*pid*)
Raised by methods of *Process* when no process with the given PID is found.

class ZombieProcess(*pid*)
Raised by methods of *Process* if 1) the process has become a zombie process and 2) it is not possible to retrieve the requested information for zombie processes.

This is a subclass of *NoSuchProcess*.

class AccessDenied(*pid*)
Raised by methods of *Process* when permission to perform an action is denied.

class TimeoutExpired(*seconds*, *pid=None*)
Raised if a timeout expires.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

R

S

T

U

V

W

Z