Module

Process

The Process module is a collection of methods used to manipulate processes. Programs that want to manipulate real and effective user and group IDs should also look at the Process::GID, and Process::UID modules. Much of the functionality here is duplicated in the Process::Sys module.

Module constants

PRIO_PGRP	Process group priority.
PRIO_PROCESS	Process priority.
PRIO_USER	User priority.
WNOHANG	Does not block if no child has exited. Not available on all platforms.
WUNTRACED	Returns stopped children as well. Not available on all platforms.
RLIM[IT]_*xxx*	Used by getrlimit and setrlimit

.

Module methods

abort

abort
abort(*msg*)

Synonym for Kernel.abort.

daemon Process.daemon(*stay_in_dir* = false, *keep_stdio_open* = false) → 0 *or* -1

1.9

Puts the current process into the background (either by forking and calling Process.setssid or by using the daemon(2) call if available). Sets the current working directory to / unless *stay_in_dir* is true. Redirects standard input, output, and error to /dev/null unless keep_stdio_open is true. Not available on all platforms.

detach Process.detach(*pid*) → *thread*

Some operating systems retain the status of terminated child processes until the parent collects that status (normally using some variant of wait()). If the parent never collects this status, the child stays around as a *zombie* process. Process.detach prevents this by setting up a separate Ruby thread whose sole job is to reap the status of the process *pid* when it terminates. Use detach only when you do not intend to explicitly wait for the child to terminate. detach checks the status only periodically (currently once each second).

In this first example, we don't reap the first child process, so it appears as a zombie in the process status display.

```
pid = fork { sleep 0.1 }
sleep 1
system("ps -o pid,state -p #{pid}")
```

produces:

```
  PID STAT
85786 ZN+
```

P rocess

In the next example, Process.detach is used to reap the child automatically—no child processes are left running.

```
pid = fork { sleep 0.1 }
Process.detach(pid)
sleep 1
system("ps -o pid,state -p #{pid}")
```

produces:

```
  PID STAT
```

egid Process.egid → *int*

Returns the effective group ID for this process.

```
Process.egid   # =>   501
```

egid= Process.egid= *int* → *int*

Sets the effective group ID for this process.

euid Process.euid → *int*

Returns the effective user ID for this process.

```
Process.euid   # =>   501
```

euid= Process.euid= *int*

Sets the effective user ID for this process. Not available on all platforms.

exec Process.exec(*command* ⟨ , *args* ⟩)

1.9

Synonym for Kernel.exec.

exit Process.exit(*int*=0)

Synonym for Kernel.exit.

exit! Process.exit!(true | false | *status*=1)

Synonym for Kernel.exit!. No exit handlers are run. 0, 1, or *status* is returned to the underlying system as the exit status.

```
Process.exit!(0)
```

fork Process.fork ⟨ { *block* } ⟩ → *int* or nil

See Kernel.fork on page 570.

getpgid Process.getpgid(*int*) → *int*

Returns the process group ID for the given process ID. Not available on all platforms.

```
Process.getpgid(Process.ppid())   # =>   82263
```

getpgrp Process.getpgrp → *int*

Returns the process group ID for this process. Not available on all platforms.

```
Process.getpgid(0)   # =>   82263
Process.getpgrp      # =>   82263
```

getpriority Process.getpriority(*kind*, *int*) → *int*

Gets the scheduling priority for specified process, process group, or user. *kind* indicates the kind of entity to find: one of Process::PRIO_PGRP, Process::PRIO_USER, or Process::PRIO_PROCESS. *int* is an ID indicating the particular process, process group, or user (an ID of 0 means *current*). Lower priorities are more favorable for scheduling. Not available on all platforms.

```
Process.getpriority(Process::PRIO_USER, 0)      # =>   19
Process.getpriority(Process::PRIO_PROCESS, 0)   # =>   19
```

getrlimit Process.getrlimit(*name*) → [*current, max*]

1.9 Returns the current and maximum resource limit for the named resource. The name may be a symbol or a string from the following list. It may also be an operating-specific integer constant. The Process module defines constants corresponding to these integers: the constants are named RLIMIT_ followed by one of the following: AS, CORE, CPU, DATA, FSIZE, MEMLOCK, NOFILE, NPROC, RSS or STACK. Consult your operating systems *getrlimit(2)* man page for details. The return array may contain actual values, or one of the constants RLIM_INFINITY, RLIM_SAVED_CUR, or RLIM_SAVED_MAX. Not available on all platforms. See also Process.setrlimit.

```
Process.getrlimit(:STACK)                   # =>   [67104768, 67104768]
Process.getrlimit("STACK")                  # =>   [67104768, 67104768]
Process.getrlimit(Process::RLIMIT_STACK)    # =>   [67104768, 67104768]
```

gid Process.gid → *int*

Returns the group ID for this process.

```
Process.gid   # =>   501
```

gid= Process.gid= *int* → *int*

Sets the group ID for this process.

groups Process.groups → *groups*

Returns an array of integer supplementary group IDs. Not available on all platforms. See also Process.maxgroups.

```
Process.groups   # =>   [501, 98, 101, 102, 80]
```

groups= Process.groups = *array* → *groups*

Sets the supplementary group IDs from the given array, which may contain either numbers or group names (as strings). Not available on all platforms. Available only to superusers. See also Process.maxgroups.

initgroups Process.initgroups(*user*, *base_group*) → *groups*

Initializes the group access list using the operating system's initgroups call. Not available on all platforms. May require superuser privilege.

```
Process.initgroups("dave", 500)
```

kill Process.kill(*signal*, ⟨ *pid* ⟩$^+$) → *int*

1.9 Sends the given signal to the specified process ID(s) or to the current process if *pid* is zero. *signal* may be an integer signal number or a string or symbol representing a POSIX signal name (either with or without a SIG prefix). If *signal* is negative (or starts with a – sign), kills process groups instead of processes. Not all signals are available on all platforms.

```
pid = fork do
   Signal.trap(:USR1) { puts "Ouch!"; exit }
   # ... do some work ...
end
# ...
Process.kill(:USR1, pid)
Process.wait
```

produces:

```
Ouch!
```

maxgroups Process.maxgroups → *count*

The Process module has a limit on the number of supplementary groups it supports in the calls Process.groups and Process.groups=. The maxgroups call returns that limit (by default 32), and the maxgroups= call sets it.

```
Process.maxgroups    # =>   32
Process.maxgroups = 64
Process.maxgroups    # =>   64
```

maxgroups= Process.maxgroups= *limit* → *count*

Sets the maximum number of supplementary group IDs that can be processed by the groups and groups= methods. If a number larger that 4096 is given, 4096 will be used.

pid Process.pid → *int*

Returns the process ID of this process. Not available on all platforms.

```
Process.pid   # =>   85816
```

ppid Process.ppid → *int*

Returns the process ID of the parent of this process. Always returns 0 on Windows. Not available on all platforms.

```
puts "I am #{Process.pid}"
Process.fork { puts "Dad is #{Process.ppid}" }
```

produces:

```
I am 85818
Dad is 85818
```

setpgid Process.setpgid(*pid*, *int*) → 0

Sets the process group ID of *pid* (0 indicates this process) to *int*. Not available on all platforms.

setpgrp Process.setpgrp → 0

Equivalent to setpgid(0,0). Not available on all platforms.

setpriority Process.setpriority(*kind*, *int*, *int_priority*) → 0

See Process#getpriority.

```
Process.setpriority(Process::PRIO_USER, 0, 19)      # =>   0
Process.setpriority(Process::PRIO_PROCESS, 0, 19)   # =>   0
Process.getpriority(Process::PRIO_USER, 0)          # =>   19
Process.getpriority(Process::PRIO_PROCESS, 0)       # =>   19
```

setrlimit Process.setrlimit(*name*, *soft_limit*, *hard_limit=soft_limit*) → nil

1.9 Sets the limit for the named resource. See Process.getrlimit for a description of resource naming. See your system's man page for setrlimit(2) for a description of the limits. Not available on all platforms.

setsid Process.setsid → *int*

Establishes this process as a new session and process group leader, with no controlling tty. Returns the session ID. Not available on all platforms.

```
Process.setsid   # =>   85823
```

spawn Process.spawn(*command* ⟨ , *args* ⟩*) → *pid*

1.9 Synonym for Kernel.spawn.

times Process.times → *struct_tms*

Returns a Tms structure (see Struct::Tms on page 700) that contains user and system CPU times for this process.

```
t = Process.times
[ t.utime, t.stime ]   # =>   [0.0, 0.0]
```

uid Process.uid → *int*

Returns the user ID of this process.

```
Process.uid   # =>   501
```

uid= Process.uid= *int* → *numeric*

Sets the (integer) user ID for this process. Not available on all platforms.

wait Process.wait → *int*

Waits for any child process to exit and returns the process ID of that child. Also sets $? to the Process::Status object containing information on that process. Raises a SystemError if there are no child processes. Not available on all platforms.

```
Process.fork { exit 99 }   # =>   85830
Process.wait               # =>   85830
$?.exitstatus              # =>   99
```

waitall Process.waitall → [[*pid1*,*status*], . . .]

Waits for all children, returning an array of *pid*/*status* pairs (where *status* is an object of class Process::Status).

```
fork { sleep 0.2; exit 2 }   # =>   85833
fork { sleep 0.1; exit 1 }   # =>   85834
fork {            exit 0 }   # =>   85835
Process.waitall              # =>   [[85835, #<Process::Status: pid 85835
                                    exit 0>], [85834, #<Process::Status:
                                    pid 85834 exit 1>], [85833,
                                    #<Process::Status: pid 85833 exit 2>]]
```

wait2 Process.wait2 → [*pid*, *status*]

Waits for any child process to exit and returns an array containing the process ID and the exit status (a Process::Status object) of that child. Raises a SystemError if no child processes exist.

```
Process.fork { exit 99 }   # =>   85838
pid, status = Process.wait2
pid                        # =>   85838
status.exitstatus          # =>   99
```

waitpid Process.waitpid(*pid*, *int*=0) → *pid*

Waits for a child process to exit depending on the value of *pid*:

< -1	Any child whose progress group ID equals the absolute value of *pid*.
-1	Any child (equivalent to wait).
0	Any child whose process group ID equals that of the current process.
> 0	The child with the given PID.

int may be a logical or of the flag values Process::WNOHANG (do not block if no child available) or Process::WUNTRACED (return stopped children that haven't been reported). Not all flags are available on all platforms, but a flag value of zero will work on all platforms.

```
include Process
pid = fork { sleep 3 }            # =>   85841
Time.now                          # =>   2009-04-13 13:26:49 -0500
waitpid(pid, Process::WNOHANG)    # =>   nil
Time.now                          # =>   2009-04-13 13:26:49 -0500
waitpid(pid, 0)                   # =>   85841
Time.now                          # =>   2009-04-13 13:26:52 -0500
```

waitpid2 Process.waitpid2(*pid*, *int*=0) → [*pid*, *status*]

Waits for the given child process to exit, returning that child's process ID and exit status (a Process::Status object). *int* may be a logical or of the values Process::WNOHANG (do not block if no child available) or Process::WUNTRACED (return stopped children that haven't been reported). Not all flags are available on all platforms, but a flag value of zero will work on all platforms.