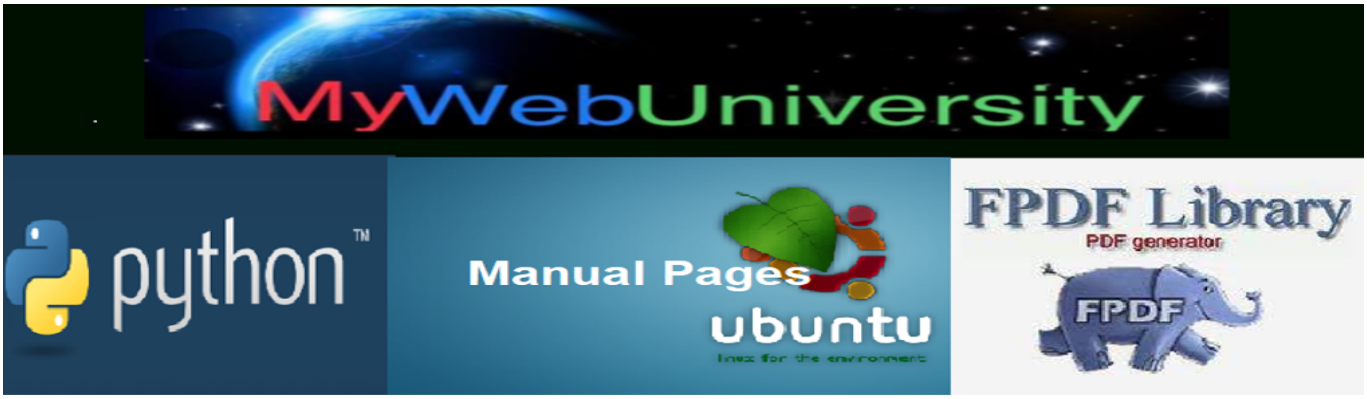

Rocky Enterprise Linux 9.2 Manual Pages on command 'BSD::Resource.3pm'

$ man BSD::Resource.3pm

Resource(3pm) User Contributed Perl Documentation Resource(3pm)

NAME

BSD::Resource - BSD process resource limit and priority functions

SYNOPSIS

```
use BSD::Resource;
#
# the process resource consumption so far
#
($usertime, $systemtime,
 $maxrss, $ixrss, $idrss, $isrss, $minflt, $majflt, $nswap,
 $inblock, $oublock, $msgsnd, $msgrcv,
 $nsignals, $nvcsw, $nivcsw) = getrusage($ru_who);
$rusage = getrusage($ru_who);
#
# the process resource limits
#
($nowsoft, $nowhard) = getrlimit($resource);
$rlimit = getrlimit($resource);
$success = setrlimit($resource, $newsoft, $newhard);
#
# the process scheduling priority
#
$nowpriority = getpriority($pr_which, $pr_who);
```

```
    $success = setpriority($pr_which, $pr_who, $priority);
    # The following is not a BSD function.
    # It is a Perlish utility for the users of BSD::Resource.
    $rlimits = get_rlimits();
```

DESCRIPTION

getrusage

```
    ($usertime, $systemtime,
     $maxrss, $ixrss, $idrss, $isrss, $minflt, $majflt, $nswap,
     $inblock, $oublock, $msgsnd, $msgrcv,
     $nsignals, $nvcsw, $nivcsw) = getrusage($ru_who);
    $rusage = getrusage($ru_who);
    # $ru_who argument is optional; it defaults to RUSAGE_SELF
    $rusage = getrusage();
```

The $ru_who argument is either "RUSAGE_SELF" (the current process) or "RUSAGE_CHILDREN" (all the child processes of the current process) or it maybe left away in which case "RUSAGE_SELF" is used.

The "RUSAGE_CHILDREN" is the total sum of all the so far terminated (either successfully or unsuccessfully) child processes: there is no way to find out information about child processes still running.

On some systems (those supporting both getrusage() with the POSIX threads) there can also be "RUSAGE_THREAD". The BSD::Resource supports the "RUSAGE_THREAD" if it is present but understands nothing more about the POSIX threads themselves. Similarly for "RUSAGE_BOTH": some systems support retrieving the sums of the self and child resource consumptions simultaneously.

In list context getrusage() returns the current resource usages as a list. On failure it returns an empty list.

The elements of the list are, in order: index name meaning usually (quite system dependent)

0	utime	user time
1	stime	system time
2	maxrss	maximum shared memory or current resident set
3	ixrss	integral shared memory
4	idrss	integral or current unshared data

5	isrss	integral or current unshared stack
6	minflt	page reclaims
7	majflt	page faults
8	nswap	swaps
9	inblock	block input operations
10	oublock	block output operations
11	msgsnd	messages sent
12	msgrcv	messaged received
13	nsignals	signals received
14	nvcsw	voluntary context switches
15	nivcsw	involuntary context switches

In scalar context getrusage() returns the current resource usages as a an object. The object can be queried via methods named exactly like the middle column, name, in the above table.

```
$ru = getrusage();
print $ru->stime, "\n";
$total_context_switches = $ru->nvcsw + $ru->nivcsw;
```

For a detailed description about the values returned by getrusage() please consult your usual C programming documentation about getrusage() and also the header file "<sys/resource.h>". (In Solaris, this might be "<sys/rusage.h>").

See also "KNOWN ISSUES".

getrlimit

```
($nowsoft, $nowhard) = getrlimit($resource);
$rlimit = getrlimit($resource);
```

The $resource argument can be one of

$resource	usual meaning	usual unit
RLIMIT_CPU	CPU time	seconds
RLIMIT_FSIZE	file size	bytes
RLIMIT_DATA	data size	bytes
RLIMIT_STACK	stack size	bytes
RLIMIT_CORE	coredump size	bytes
RLIMIT_RSS	resident set size	bytes
RLIMIT_MEMLOCK	memory locked data size	bytes

RLIMIT_NPROC	number of processes	1
RLIMIT_NOFILE	number of open files	1
RLIMIT_OFILE	number of open files	1
RLIMIT_OPEN_MAX	number of open files	1
RLIMIT_LOCKS	number of file locks	1
RLIMIT_AS	(virtual) address space	bytes
RLIMIT_VMEM	virtual memory (space)	bytes
RLIMIT_PTHREAD	number of pthreads	1
RLIMIT_TCACHE	maximum number of cached threads	1
RLIMIT_AIO_MEM	maximum memory locked for POSIX AIO	bytes
RLIMIT_AIO_OPS	maximum number for POSIX AIO ops	1
RLIMIT_FREEMEM	portion of the total memory	
RLIMIT_NTHR	maximum number of threads	1
RLIMIT_NPTS	maximum number of pseudo-terminals	1
RLIMIT_RSESTACK	RSE stack size	bytes
RLIMIT_SBSIZE	socket buffer size	bytes
RLIMIT_SWAP	maximum swap size	bytes
RLIMIT_MSGQUEUE	POSIX mq size	bytes
RLIMIT_RTPRIO	maximum RT priority	1
RLIMIT_RTTIME	maximum RT time	microseconds
RLIMIT_SIGPENDING	pending signals	1

What limits are available depends on the operating system.

See below for "get_rlimits()" on how to find out which limits are available, for the exact documentation consult the documentation of your operating system (setrlimit documentation, usually).

The two groups ("NOFILE", "OFILE", "OPEN_MAX") and ("AS", "VMEM") are aliases within themselves.

Two meta-resource-symbols might exist

RLIM_NLIMITS

RLIM_INFINITY

"RLIM_NLIMITS" being the number of possible (but not necessarily fully supported) resource limits, see also the get_rlimits() call below. "RLIM_INFINITY" is useful in setrlimit(), the "RLIM_INFINITY" is often represented as minus one (-1).

In list context "getrlimit()" returns the current soft and hard resource limits as a list. On failure it returns an empty list.

Processes have soft and hard resource limits. On crossing the soft limit they receive a signal (for example the "SIGXCPU" or "SIGXFSZ", corresponding to the "RLIMIT_CPU" and "RLIMIT_FSIZE", respectively). The processes can trap and handle some of these signals, please see "Signals" in perlipc. After the hard limit the processes will be ruthlessly killed by the "KILL" signal which cannot be caught.

NOTE: the level of 'support' for a resource varies. Not all the systems

a) even recognise all those limits

b) really track the consumption of a resource

c) care (send those signals) if a resource limit is exceeded

Again, please consult your usual C programming documentation.

One notable exception for the better: officially HP-UX does not support getrlimit() at all but for the time being, it does seem to.

In scalar context "getrlimit()" returns the current soft limit. On failure it returns "undef".

getpriority

```
# $pr_which can be PRIO_USER, PRIO_PROCESS, or PRIO_PGRP,
# and in some systems PRIO_THREAD
$nowpriority = getpriority($pr_which, $pr_who);
# the default $pr_who is 0 (the current $pr_which)
$nowpriority = getpriority($pr_which);
# the default $pr_which is PRIO_PROCESS (the process priority)
$nowpriority = getpriority();
```

getpriority() returns the current priority. NOTE: getpriority() can return zero or negative values completely legally. On failure getpriority() returns "undef" (and $! is set as usual).

The priorities returned by getpriority() are in the (inclusive) range

"PRIO_MIN"..."PRIO_MAX". The $pr_which argument can be any of PRIO_PROCESS (a process) "PRIO_USER" (a user), or "PRIO_PGRP" (a process group). The $pr_who argument tells which process/user/process group, 0 signifying the current one.

Usual values for "PRIO_MIN", "PRIO_MAX", are -20, 20. A negative value means better priority (more impolite process), a positive value means worse priority (more polite process).

setrlimit

```
$success = setrlimit($resource, $newsoft, $newhard);
```

setrlimit() returns true on success and "undef" on failure.

NOTE: A normal user process can only lower its resource limits. Soft or hard limit "RLIM_INFINITY" means as much as possible, the real hard limits are normally buried inside the kernel and are very system-dependent.

NOTE: Even the soft limit that is actually set might be lower than what requested for various reasons. One possibility is that the actual limit on a resource might be controlled by some system variable (e.g. in BSD systems the RLIMIT_NPROC can be capped by the system variable "maxprocperuid", try "sysctl -a kern.maxprocperuid"), or in many environments core dumping has been disabled from normal user processes. Another possibility is that a limit is rounded down to some alignment or granularity, for example the memory limits might be rounded down to the closest 4 kilobyte boundary. In other words, do not expect to be able to setrlimit() a limit to a value and then be able to read back the same value with getrlimit().

setpriority

```
$success = setpriority($pr_which, $pr_who, $priority);
# NOTE! If there are two arguments the second one is
# the new $priority (not $pr_who) and the $pr_who is
# defaulted to 0 (the current $pr_which)
$success = setpriority($pr_which, $priority);
# The $pr_who defaults to 0 (the current $pr_which) and
# the $priority defaults to half of the PRIO_MAX, usually
# that amounts to 10 (being a nice $pr_which).
$success = setpriority($pr_which);
# The $pr_which defaults to PRIO_PROCESS.
$success = setpriority();
```

setpriority() is used to change the scheduling priority. A positive priority means a more polite process/process group/user; a negative priority means a more impolite process/process group/user. The priorities handled by setpriority() are ["PRIO_MIN","PRIO_MAX"]. A normal user process can only lower its priority (make it more positive).

NOTE: A successful call returns 1, a failed one 0.

See also "KNOWN ISSUES".

times

```
use BSD::Resource qw(times);
($user, $system, $child_user, $child_system) = times();
```

The BSD::Resource module offers a times() implementation that has usually slightly better time granularity than the times() by Perl core. The time granularity of the latter is usually 1/60 seconds while the former may achieve submilliseconds.

NOTE: The current implementation uses two getrusage() system calls: one with RUSAGE_SELF and one with RUSAGE_CHILDREN. Therefore the operation is not `atomic': the times for the children are recorded a little bit later.

NOTE: times() is not imported by default by BSD::Resource. You need to tell that you want to use it.

NOTE: times() is not a "real BSD" function. It is older UNIX.

get_rlimits

```
use BSD::Resource qw{get_rlimits};
my $limits = get_rlimits();
```

NOTE: This is not a real BSD function. It is a convenience function introduced by BSD::Resource.

get_rlimits() returns a reference to hash which has the names of the available resource limits as keys and their indices (those which are needed as the first argument to getrlimit() and setrlimit()) as values. For example:

```
use BSD::Resource qw{get_rlimits};
my $limits = get_rlimits();
for my $name (keys %$limits) {
  my ($soft, $hard) = BSD::Resource::getrlimit($limits->{$name});
  print "$name soft $soft hard $hard\n";
}
```

Note that a limit of -1 means unlimited.

ERRORS

?

Your vendor has not defined BSD::Resource macro ...

The code tried to call getrlimit/setrlimit for a resource limit that your operating system vendor/supplier does not support. Portable code should use get_rlimits() to check which resource limits are defined.

EXAMPLES

```
# the user and system times so far by the process itself
($usertime, $systemtime) = getrusage();
# ditto in OO way
$ru = getrusage();
$usertime   = $ru->utime;
$systemtime = $ru->stime;
# get the current priority level of this process
$currprio = getpriority();
```

KNOWN ISSUES

In AIX (at least version 3, maybe later also releases) if the BSD compatibility library is not installed or not found by the BSD::Resource installation procedure and when using the getpriority() or setpriority(), the "PRIO_MIN" is 0 (corresponding to -20) and "PRIO_MAX" is 39 (corresponding to 19, the BSD priority 20 is unreachable).

In HP-UX the getrusage() is not Officially Supported at all but for the time being, it does seem to be.

In Mac OS X a normal user cannot raise the "RLIM_NPROC" over the maxprocperuid limit (the default value is 266, try the command "sysctl -a kern.maxprocperuid").

In NetBSD "RLIMIT_STACK" setrlimit() calls fail.

In Cygwin "RLIMIT_STACK" setrlimit calls fail. Also, setrlimit() "RLIMIT_NOFILE/RLIMIT_OFILE/RLIMIT_OFILE" calls return success, but then the subsequent getrlimit calls show that the limits didn't really change.

Because not all UNIX kernels are BSD and also because of the sloppy support of getrusage() by many vendors many of the getrusage() values may not be correctly updated. For example Solaris 1 claims in "<sys/rusage.h>" that the "ixrss" and the "isrss" fields are always zero. In SunOS 5.5 and 5.6 the getrusage() leaves most of the fields zero and therefore

getrusage() is not even used, instead of that the /proc interface is used. The mapping is not perfect: the "maxrss" field is really the current resident size instead of the maximum, the "idrss" is really the current heap size instead of the integral data, and the "isrss" is really the current stack size instead of the integral stack. The ixrss has no sensible counterpart at all so it stays zero.

COPYRIGHT AND LICENSE

AUTHOR

Jarkko Hietaniemi, "jhi@iki.fi"