Red Hat Enterprise Linux Release 9.2 Manual Pages on 'vlimit.3' command

$ man vlimit.3

GETRLIMIT(2) Linux Programmer's Manual GETRLIMIT(2)

NAME

getrlimit, setrlimit, prlimit - get/set resource limits

SYNOPSIS

#include <sys/time.h>

#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);

int setrlimit(int resource, const struct rlimit *rlim);

int prlimit(pid_t pid, int resource, const struct rlimit *new_limit,

struct rlimit *old_limit);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

prlimit(): _GNU_SOURCE

DESCRIPTION

The getrlimit() and setrlimit() system calls get and set resource lim?

its. Each resource has an associated soft and hard limit, as defined

by the rlimit structure:

```
struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

The soft limit is the value that the kernel enforces for the corre?

sponding resource. The hard limit acts as a ceiling for the soft

limit: an unprivileged process may set only its soft limit to a value in the range from 0 up to the hard limit, and (irreversibly) lower its hard limit. A privileged process (under Linux: one with the CAP_SYS_RESOURCE capability in the initial user namespace) may make ar? bitrary changes to either limit value.

The value RLIM_INFINITY denotes no limit on a resource (both in the structure returned by getrlimit() and in the structure passed to setr? limit()).

The resource argument must be one of:

RLIMIT_AS

This is the maximum size of the process's virtual memory (ad? dress space). The limit is specified in bytes, and is rounded down to the system page size. This limit affects calls to brk(2), mmap(2), and mremap(2), which fail with the error ENOMEM upon exceeding this limit. In addition, automatic stack expan? sion fails (and generates a SIGSEGV that kills the process if no alternate stack has been made available via sigaltstack(2)). Since the value is a long, on machines with a 32-bit long either this limit is at most 2 GiB, or this resource is unlimited.

RLIMIT_CORE

This is the maximum size of a core file (see core(5)) in bytes that the process may dump. When 0 no core dump files are cre? ated. When nonzero, larger dumps are truncated to this size.

RLIMIT_CPU

This is a limit, in seconds, on the amount of CPU time that the process can consume. When the process reaches the soft limit, it is sent a SIGXCPU signal. The default action for this signal is to terminate the process. However, the signal can be caught, and the handler can return control to the main program. If the process continues to consume CPU time, it will be sent SIGXCPU once per second until the hard limit is reached, at which time it is sent SIGKILL. (This latter point describes Linux behav? ior. Implementations vary in how they treat processes which continue to consume CPU time after reaching the soft limit.

Portable applications that need to catch this signal should per?
form an orderly termination upon first receipt of SIGXCPU.)

RLIMIT_DATA

This is the maximum size of the process's data segment (initial? ized data, uninitialized data, and heap). The limit is speci? fied in bytes, and is rounded down to the system page size. This limit affects calls to brk(2), sbrk(2), and (since Linux 4.7) mmap(2), which fail with the error ENOMEM upon encountering the soft limit of this resource.

RLIMIT_FSIZE

This is the maximum size in bytes of files that the process may create. Attempts to extend a file beyond this limit result in delivery of a SIGXFSZ signal. By default, this signal termi? nates a process, but a process can catch this signal instead, in which case the relevant system call (e.g., write(2), trun? cate(2)) fails with the error EFBIG.

RLIMIT_LOCKS (Linux 2.4.0 to 2.4.24)

This is a limit on the combined number of flock(2) locks and fc? ntl(2) leases that this process may establish.

RLIMIT_MEMLOCK

This is the maximum number of bytes of memory that may be locked into RAM. This limit is in effect rounded down to the nearest multiple of the system page size. This limit affects mlock(2), mlockall(2), and the mmap(2) MAP_LOCKED operation. Since Linux 2.6.9, it also affects the shmctl(2) SHM_LOCK operation, where it sets a maximum on the total bytes in shared memory segments (see shmget(2)) that may be locked by the real user ID of the calling process. The shmctl(2) SHM_LOCK locks are accounted for separately from the per-process memory locks established by mlock(2), mlockall(2), and mmap(2) MAP_LOCKED; a process can lock bytes up to this limit in each of these two categories.

In Linux kernels before 2.6.9, this limit controlled the amount of memory that could be locked by a privileged process. Since

Linux 2.6.9, no limits are placed on the amount of memory that a privileged process may lock, and this limit instead governs the amount of memory that an unprivileged process may lock.

RLIMIT_MSGQUEUE (since Linux 2.6.8)

This is a limit on the number of bytes that can be allocated for POSIX message queues for the real user ID of the calling process. This limit is enforced for mq_open(3). Each message queue that the user creates counts (until it is removed) against this limit according to the formula:

```
Since Linux 3.5:
    bytes = attr.mq_maxmsg * sizeof(struct msg_msg) +
            min(attr.mq_maxmsg, MQ_PRIO_MAX) *
                sizeof(struct posix_msg_tree_node)+
                        /* For overhead */
            attr.mq_maxmsg * attr.mq_msgsize;
                        /* For message data */
Linux 3.4 and earlier:
    bytes = attr.mq_maxmsg * sizeof(struct msg_msg *) +
                        /* For overhead */
            attr.mq_maxmsg * attr.mq_msgsize;
                        /* For message data */
```

where attr is the mq_attr structure specified as the fourth ar? gument to mq_open(3), and the msg_msg and posix_msg_tree_node structures are kernel-internal structures.

The "overhead" addend in the formula accounts for overhead bytes required by the implementation and ensures that the user cannot create an unlimited number of zero-length messages (such mes? sages nevertheless each consume some system memory for bookkeep? ing overhead).

RLIMIT_NICE (since Linux 2.6.12, but see BUGS below)

This specifies a ceiling to which the process's nice value can be raised using setpriority(2) or nice(2). The actual ceiling for the nice value is calculated as 20 - rlim_cur. The useful range for this limit is thus from 1 (corresponding to a nice value of 19) to 40 (corresponding to a nice value of -20). This unusual choice of range was necessary because negative numbers cannot be specified as resource limit values, since they typi? cally have special meanings. For example, RLIM_INFINITY typi? cally is the same as -1. For more detail on the nice value, see sched(7).

RLIMIT_NOFILE

This specifies a value one greater than the maximum file de? scriptor number that can be opened by this process. Attempts (open(2), pipe(2), dup(2), etc.) to exceed this limit yield the error EMFILE. (Historically, this limit was named RLIMIT_OFILE on BSD.)

Since Linux 4.5, this limit also defines the maximum number of file descriptors that an unprivileged process (one without the CAP_SYS_RESOURCE capability) may have "in flight" to other pro? cesses, by being passed across UNIX domain sockets. This limit applies to the sendmsg(2) system call. For further details, see unix(7).

RLIMIT_NPROC

This is a limit on the number of extant process (or, more pre? cisely on Linux, threads) for the real user ID of the calling process. So long as the current number of processes belonging to this process's real user ID is greater than or equal to this limit, fork(2) fails with the error EAGAIN.

The RLIMIT_NPROC limit is not enforced for processes that have either the CAP_SYS_ADMIN or the CAP_SYS_RESOURCE capability.

RLIMIT_RSS

This is a limit (in bytes) on the process's resident set (the number of virtual pages resident in RAM). This limit has effect only in Linux 2.4.x, x < 30, and there affects only calls to madvise(2) specifying MADV_WILLNEED.

RLIMIT_RTPRIO (since Linux 2.6.12, but see BUGS)

This specifies a ceiling on the real-time priority that may be set for this process using sched_setscheduler(2) and sched_set?param(2).

For further details on real-time scheduling policies, see sched(7)

RLIMIT_RTTIME (since Linux 2.6.25)

This is a limit (in microseconds) on the amount of CPU time that a process scheduled under a real-time scheduling policy may con?sume without making a blocking system call. For the purpose of this limit, each time a process makes a blocking system call, the count of its consumed CPU time is reset to zero. The CPU time count is not reset if the process continues trying to use the CPU but is preempted, its time slice expires, or it calls sched_yield(2).

Upon reaching the soft limit, the process is sent a SIGXCPU sig?nal. If the process catches or ignores this signal and contin?ues consuming CPU time, then SIGXCPU will be generated once each second until the hard limit is reached, at which point the process is sent a SIGKILL signal.

The intended use of this limit is to stop a runaway real-time process from locking up the system.

For further details on real-time scheduling policies, see sched(7)

RLIMIT_SIGPENDING (since Linux 2.6.8)

This is a limit on the number of signals that may be queued for the real user ID of the calling process. Both standard and real-time signals are counted for the purpose of checking this limit. However, the limit is enforced only for sigqueue(3); it is always possible to use kill(2) to queue one instance of any of the signals that are not already queued to the process.

RLIMIT_STACK

This is the maximum size of the process stack, in bytes. Upon reaching this limit, a SIGSEGV signal is generated. To handle this signal, a process must employ an alternate signal stack (sigaltstack(2)).

Since Linux 2.6.23, this limit also determines the amount of space used for the process's command-line arguments and environ? ment variables; for details, see execve(2).

prlimit()

The Linux-specific prlimit() system call combines and extends the func? tionality of setrlimit() and getrlimit(). It can be used to both set and get the resource limits of an arbitrary process.

The resource argument has the same meaning as for setrlimit() and getr? limit().

If the new_limit argument is a not NULL, then the rlimit structure to which it points is used to set new values for the soft and hard limits for resource. If the old_limit argument is a not NULL, then a success? ful call to prlimit() places the previous soft and hard limits for re? source in the rlimit structure pointed to by old_limit.

The pid argument specifies the ID of the process on which the call is to operate. If pid is 0, then the call applies to the calling process.

To set or get the resources of a process other than itself, the caller must have the CAP_SYS_RESOURCE capability in the user namespace of the process whose resource limits are being changed, or the real, effec? tive, and saved set user IDs of the target process must match the real user ID of the caller and the real, effective, and saved set group IDs of the target process must match the real group ID of the caller.

RETURN VALUE

On success, these system calls return 0. On error, -1 is returned, and errno is set appropriately.

ERRORS

EFAULT A pointer argument points to a location outside the accessible address space.

EINVAL The value specified in resource is not valid; or, for setr? limit() or prlimit(): rlim->rlim_cur was greater than rlim->rlim_max.

EPERM An unprivileged process tried to raise the hard limit; the

CAP_SYS_RESOURCE capability is required to do this.

EPERM The caller tried to increase the hard RLIMIT_NOFILE limit above

the maximum defined by /proc/sys/fs/nr_open (see proc(5))

EPERM (prlimit()) The calling process did not have permission to set

limits for the process specified by pid.

ESRCH Could not find a process with the ID specified in pid.

VERSIONS

The prlimit() system call is available since Linux 2.6.36. Library

support is available since glibc 2.13.

ATTRIBUTES

For an explanation of the terms used in this section, see at?

tributes(7).

???

?Interface ? Attribute ? Value ?

???

?getrlimit(), setrlimit(), prlimit() ? Thread safety ? MT-Safe ?

???

CONFORMING TO

getrlimit(), setrlimit(): POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

prlimit(): Linux-specific.

RLIMIT_MEMLOCK and RLIMIT_NPROC derive from BSD and are not specified

in POSIX.1; they are present on the BSDs and Linux, but on few other

implementations. RLIMIT_RSS derives from BSD and is not specified in

POSIX.1; it is nevertheless present on most implementations.

RLIMIT_MSGQUEUE, RLIMIT_NICE, RLIMIT_RTPRIO, RLIMIT_RTTIME, and

RLIMIT_SIGPENDING are Linux-specific.

NOTES

A child process created via fork(2) inherits its parent's resource lim?

its. Resource limits are preserved across execve(2).

Resource limits are per-process attributes that are shared by all of

the threads in a process.

Lowering the soft limit for a resource below the process's current con?

sumption of that resource will succeed (but will prevent the process from further increasing its consumption of the resource).

One can set the resource limits of the shell using the built-in ulimit command (limit in csh(1)). The shell's resource limits are inherited by the processes that it creates to execute commands.

Since Linux 2.6.24, the resource limits of any process can be inspected via /proc/[pid]/limits; see proc(5).

Ancient systems provided a vlimit() function with a similar purpose to setrlimit(). For backward compatibility, glibc also provides vlimit(). All new applications should be written using setrlimit().

C library/kernel ABI differences

Since version 2.13, the glibc getrlimit() and setrlimit() wrapper func? tions no longer invoke the corresponding system calls, but instead em? ploy prlimit(), for the reasons described in BUGS.

The name of the glibc wrapper function is prlimit(); the underlying system call is prlimit64().

BUGS

In older Linux kernels, the SIGXCPU and SIGKILL signals delivered when a process encountered the soft and hard RLIMIT_CPU limits were deliv? ered one (CPU) second later than they should have been. This was fixed in kernel 2.6.8.

In 2.6.x kernels before 2.6.17, a RLIMIT_CPU limit of 0 is wrongly treated as "no limit" (like RLIM_INFINITY). Since Linux 2.6.17, set? ting a limit of 0 does have an effect, but is actually treated as a limit of 1 second.

A kernel bug means that RLIMIT_RTPRIO does not work in kernel 2.6.12; the problem is fixed in kernel 2.6.13.

In kernel 2.6.12, there was an off-by-one mismatch between the priority ranges returned by getpriority(2) and RLIMIT_NICE. This had the effect that the actual ceiling for the nice value was calculated as 19 - rlim_cur. This was fixed in kernel 2.6.13.

Since Linux 2.6.12, if a process reaches its soft RLIMIT_CPU limit and has a handler installed for SIGXCPU, then, in addition to invoking the signal handler, the kernel increases the soft limit by one second.

This behavior repeats if the process continues to consume CPU time, un?

til the hard limit is reached, at which point the process is killed.

Other implementations do not change the RLIMIT_CPU soft limit in this

manner, and the Linux behavior is probably not standards conformant;

portable applications should avoid relying on this Linux-specific be?

havior. The Linux-specific RLIMIT_RTTIME limit exhibits the same be?

havior when the soft limit is encountered.

Kernels before 2.4.22 did not diagnose the error EINVAL for setrlimit()

when rlim->rlim_cur was greater than rlim->rlim_max.

Linux doesn't return an error when an attempt to set RLIMIT_CPU has

failed, for compatibility reasons.

Representation of "large" resource limit values on 32-bit platforms

The glibc getrlimit() and setrlimit() wrapper functions use a 64-bit

rlim_t data type, even on 32-bit platforms. However, the rlim_t data

type used in the getrlimit() and setrlimit() system calls is a (32-bit)

unsigned long. Furthermore, in Linux, the kernel represents resource

limits on 32-bit platforms as unsigned long. However, a 32-bit data

type is not wide enough. The most pertinent limit here is

RLIMIT_FSIZE, which specifies the maximum size to which a file can

grow: to be useful, this limit must be represented using a type that is

as wide as the type used to represent file offsets?that is, as wide as

a 64-bit off_t (assuming a program compiled with _FILE_OFFSET_BITS=64).

To work around this kernel limitation, if a program tried to set a re?

source limit to a value larger than can be represented in a 32-bit un?

signed long, then the glibc setrlimit() wrapper function silently con?

verted the limit value to RLIM_INFINITY. In other words, the requested

resource limit setting was silently ignored.

Since version 2.13, glibc works around the limitations of the getr?

limit() and setrlimit() system calls by implementing setrlimit() and

getrlimit() as wrapper functions that call prlimit().

EXAMPLES

The program below demonstrates the use of prlimit().

```
#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <stdint.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>
#define errExit(msg) do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)
int
main(int argc, char *argv[])
{
    struct rlimit old, new;
    struct rlimit *newp;
    pid_t pid;
    if (!(argc == 2 || argc == 4)) {
        fprintf(stderr, "Usage: %s <pid> [<new-soft-limit> "
                "<new-hard-limit>]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    pid = atoi(argv[1]);        /* PID of target process */
    newp = NULL;
    if (argc == 4) {
        new.rlim_cur = atoi(argv[2]);
        new.rlim_max = atoi(argv[3]);
        newp = &new;
    }
    /* Set CPU time limit of target process; retrieve and display
       previous limit */
    if (prlimit(pid, RLIMIT_CPU, newp, &old) == -1)
        errExit("prlimit-1");
    printf("Previous limits: soft=%jd; hard=%jd\n",
```

```
            (intmax_t) old.rlim_cur, (intmax_t) old.rlim_max);

    /* Retrieve and display new CPU time limit */

    if (prlimit(pid, RLIMIT_CPU, NULL, &old) == -1)

        errExit("prlimit-2");

    printf("New limits: soft=%jd; hard=%jd\n",

            (intmax_t) old.rlim_cur, (intmax_t) old.rlim_max);

    exit(EXIT_SUCCESS);

}
```

SEE ALSO

prlimit(1), dup(2), fcntl(2), fork(2), getrusage(2), mlock(2), mmap(2), open(2), quotactl(2), sbrk(2), shmctl(2), malloc(3), sigqueue(3), ulimit(3), core(5), capabilities(7), cgroups(7), credentials(7), sig? nal(7)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.