NAME

getrlimit, setrlimit − 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***);**

DESCRIPTION

getrlimit() and **setrlimit**() get and set resource limits respectively. Each resource has an associated soft and hard limit, as defined by the *rlimit* structure (the *rlim* argument to both **getrlimit**() and **setrlimit**()):

```
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 corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may only set 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) may make arbitrary 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 **setrlimit**()).

resource must be one of:

RLIMIT_AS
The maximum size of the process's virtual memory (address space) in bytes. This limit affects calls to **brk**(2), **mmap**(2) and **mremap**(2), which fail with the error **ENOMEM** upon exceeding this limit. Also automatic stack expansion will fail (and generate 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
Maximum size of *core* file. When 0 no core dump files are created. When non-zero, larger dumps are truncated to this size.

RLIMIT_CPU
CPU time limit in seconds. 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 2.2 through 2.6 behavior. 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 perform an orderly termination upon first receipt of **SIGXCPU**.)

RLIMIT_DATA
The maximum size of the process's data segment (initialized data, uninitialized data, and heap). This limit affects calls to **brk**(2) and **sbrk**(2), which fail with the error **ENOMEM** upon encountering the soft limit of this resource.

RLIMIT_FSIZE
The maximum size 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 terminates a process, but a process can catch this signal instead, in which case the relevant system call (e.g., **write**(2), **truncate**(2)) fails with the error **EFBIG**.

RLIMIT_LOCKS (Early Linux 2.4 only)
A limit on the combined number of **flock**(2) locks and **fcntl**(2) leases that this process may establish.

RLIMIT_MEMLOCK
The maximum number of bytes of memory that may be locked into RAM. In effect this limit is rounded down to the nearest multiple of the system page size. This limit affects **mlock**(2) and **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)
Specifies the 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:

```
bytes = attr.mq_maxmsg * sizeof(struct msg_msg *) +
        attr.mq_maxmsg * attr.mq_msgsize
```

where *attr* is the *mq_attr* structure specified as the fourth argument to **mq_open**(3).

The first addend in the formula, which includes *sizeof(struct msg_msg *)* (4 bytes on Linux/i386), ensures that the user cannot create an unlimited number of zero-length messages (such messages nevertheless each consume some system memory for bookkeeping overhead).

RLIMIT_NICE (since Linux 2.6.12, but see BUGS below)
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 $20 - rlim\_cur$. (This strangeness occurs because negative numbers cannot be specified as resource limit values, since they typically have special meanings. For example, **RLIM_INFINITY** typically is the same as −1.)

RLIMIT_NOFILE
Specifies a value one greater than the maximum file descriptor 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.)

RLIMIT_NPROC
The maximum number of processes (or, more precisely on Linux, threads) that can be created for the real user ID of the calling process. Upon encountering this limit, **fork**(2) fails with the error **EAGAIN**.

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

RLIMIT_RTPRIO (Since Linux 2.6.12, but see BUGS)
Specifies a ceiling on the real-time priority that may be set for this process using **sched_setscheduler**(2) and **sched_setparam**(2).

RLIMIT_RTTIME (Since Linux 2.6.25)
Specifies a limit on the amount of CPU time that a process scheduled under a real-time scheduling policy may consume 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** signal. If the process catches or ignores this signal and continues 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.

RLIMIT_SIGPENDING (Since Linux 2.6.8)
Specifies the 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 only enforced for **sigqueue**(2); 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
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 environment variables; for details, see **execve**(2).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EFAULT
rlim points outside the accessible address space.

EINVAL
resource is not valid; or, for **setrlimit**(): *rlim*−>*rlim_cur* was greater than *rlim*−>*rlim_max*.

EPERM
An unprivileged process tried to use **setrlimit**() to increase a soft or hard limit above the current hard limit; the **CAP_SYS_RESOURCE** capability is required to do this. Or, the process tried to use **setrlimit**() to increase the soft or hard **RLIMIT_NOFILE** limit above the current kernel maximum (**NR_OPEN**).

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001. **RLIMIT_MEMLOCK** and **RLIMIT_NPROC** derive from BSD and are not specified in POSIX.1-2001; they are present on the BSDs and Linux, but on few other implementations. **RLIMIT_RSS** derives from BSD and is not specified in POSIX.1-2001; 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 limits. Resource limits are preserved across **execve**(2).

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.

BUGS

In older Linux kernels, the **SIGXCPU** and **SIGKILL** signals delivered when a process encountered the soft and hard **RLIMIT_CPU** limits were delivered 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, setting 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 actual ceiling for the nice value was calculated as *19 − rlim_cur*. This was fixed in kernel 2.6.13.

Kernels before 2.4.22 did not diagnose the error **EINVAL** for **setrlimit**() when *rlim−>rlim_cur* was greater than *rlim−>rlim_max*.

SEE ALSO

dup(2), **fcntl**(2), **fork**(2), **getrusage**(2), **mlock**(2), **mmap**(2), **open**(2), **quotactl**(2), **sbrk**(2), **shmctl**(2), **sigqueue**(2), **malloc**(3), **ulimit**(3), **core**(5), **capabilities**(7), **signal**(7)

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.