36

PROCESS RESOURCES

Each process consumes system resources such as memory and CPU time. This chapter looks at resource-related system calls. We begin with the *getrusage()* system call, which allows a process to monitor the resources that it has used or that its children have used. We then look at the *setrlimit()* and *getrlimit()* system calls, which can be used to change and retrieve limits on the calling process's consumption of various resources.

36.1 Process Resource Usage

The *getrusage()* system call retrieves statistics about various system resources used by the calling process or by all of its children.

```
#include <sys/resource.h>

int getrusage(int who, struct rusage *res_usage);
```

Returns 0 on success, or −1 on error

The *who* argument specifies the process(es) for which resource usage information is to be retrieved. It has one of the following values:

RUSAGE_SELF
: Return information about the calling process.

RUSAGE_CHILDREN
: Return information about all children of the calling process that have terminated and been waited for.

RUSAGE_THREAD (since Linux 2.6.26)
: Return information about the calling thread. This value is Linux-specific.

The *res_usage* argument is a pointer to a structure of type *rusage*, defined as shown in Listing 36-1.

Listing 36-1: Definition of the *rusage* structure

```
struct rusage {
    struct timeval ru_utime;      /* User CPU time used */
    struct timeval ru_stime;      /* System CPU time used */
    long           ru_maxrss;     /* Maximum size of resident set (kilobytes)
                                     [used since Linux 2.6.32] */
    long           ru_ixrss;      /* Integral (shared) text memory size
                                     (kilobyte-seconds) [unused] */
    long           ru_idrss;      /* Integral (unshared) data memory used
                                     (kilobyte-seconds) [unused] */
    long           ru_isrss;      /* Integral (unshared) stack memory used
                                     (kilobyte-seconds) [unused] */
    long           ru_minflt;     /* Soft page faults (I/O not required) */
    long           ru_majflt;     /* Hard page faults (I/O required) */
    long           ru_nswap;      /* Swaps out of physical memory [unused] */
    long           ru_inblock;    /* Block input operations via file
                                     system [used since Linux 2.6.22] */
    long           ru_oublock;    /* Block output operations via file
                                     system [used since Linux 2.6.22] */
    long           ru_msgsnd;     /* IPC messages sent [unused] */
    long           ru_msgrcv;     /* IPC messages received [unused] */
    long           ru_nsignals;   /* Signals received [unused] */
    long           ru_nvcsw;      /* Voluntary context switches (process
                                     relinquished CPU before its time slice
                                     expired) [used since Linux 2.6] */
    long          ru_nivcsw;      /* Involuntary context switches (higher
                                     priority process became runnable or time
                                     slice ran out) [used since Linux 2.6] */
};
```

As indicated in the comments in Listing 36-1, on Linux, many of the fields in the *rusage* structure are not filled in by *getrusage()* (or *wait3()* and *wait4()*), or they are filled in only by more recent kernel versions. Some of the fields that are unused on Linux are used on other UNIX implementations. These fields are provided on Linux so that, if they are implemented at a future date, the *rusage* structure does not need to undergo a change that would break existing application binaries.

> Although *getrusage()* appears on most UNIX implementations, it is only weakly specified in SUSv3 (which specifies only the fields *ru_utime* and *ru_stime*). In part, this is because the meaning of much of the information in the *rusage* structure is implementation-dependent.

The *ru_utime* and *ru_stime* fields are structures of type *timeval* (Section 10.1), which return the number of seconds and microseconds of CPU time consumed by a process in user mode and kernel mode, respectively. (Similar information is retrieved by the *times()* system call described in Section 10.7.)

> The Linux-specific `/proc/PID/stat` files expose some resource usage information (CPU time and page faults) about all processes on the system. See the *proc(5)* manual page for further details.

The *rusage* structure returned by the *getrusage()* `RUSAGE_CHILDREN` operation includes the resource usage statistics of all of the descendants of the calling process. For example, if we have three processes related as parent, child, and grandchild, then, when the child does a *wait()* on the grandchild, the resource usage values of the grandchild are added to the child's `RUSAGE_CHILDREN` values; when the parent performs a *wait()* for the child, the resource usage values of both the child and the grandchild are added to the parent's `RUSAGE_CHILDREN` values. Conversely, if the child does not *wait()* on the grandchild, then the grandchild's resource usages are not recorded in the `RUSAGE_CHILDREN` values of the parent.

For the `RUSAGE_CHILDREN` operation, the *ru_maxrss* field returns the maximum resident set size among all of the descendants of the calling process (rather than a sum for all descendants).

> SUSv3 specifies that if `SIGCHLD` is being ignored (so that children are not turned into zombies that can be waited on), then the child statistics should not be added to the values returned by `RUSAGE_CHILDREN`. However, as noted in Section 26.3.3, in kernels before 2.6.9, Linux deviates from this requirement—if `SIGCHLD` is ignored, then the resource usage values for dead children *are* included in the values returned for `RUSAGE_CHILDREN`.

36.2 Process Resource Limits

Each process has a set of resource limits that can be used to restrict the amounts of various system resources that the process may consume. For example, we may want to set resource limits on a process before execing an arbitrary program, if we are concerned that it may consume excessive resources. We can set the resource limits of the shell using the *ulimit* built-in command (*limit* in the C shell). These limits are inherited by the processes that the shell creates to execute user commands.

> Since kernel 2.6.24, the Linux-specific `/proc/PID/limits` file can be used to view all of the resource limits of any process. This file is owned by the real user ID of the corresponding process and its permissions allow reading only by that user ID (or by a privileged process).

The *getrlimit()* and *setrlimit()* system calls allow a process to fetch and modify its resource limits.

```
#include <sys/resource.h>

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

Both return 0 on success, or −1 on error

The *resource* argument identifies the resource limit to be retrieved or changed. The *rlim* argument is used to return resource limit values (*getrlimit()*) or to specify new resource limit values (*setrlimit()*), and is a pointer to a structure containing two fields:

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

These fields correspond to the two associated limits for a resource: the *soft* (*rlim_cur*) and *hard* (*rlim_max*) limits. (The *rlim_t* data type is an integer type.) The soft limit governs the amount of the resource that may be consumed by the process. A process can adjust the soft limit to any value from 0 up to the hard limit. For most resources, the sole purpose of the hard limit is to provide this ceiling for the soft limit. A privileged (`CAP_SYS_RESOURCE`) process can adjust the hard limit in either direction (as long as its value remains greater than the soft limit), but an unprivileged process can adjust the hard limit only to a lower value (irreversibly). The value `RLIM_INFINITY` in *rlim_cur* or *rlim_max* means infinity (no limit on the resource), both when retrieved via *getrlimit()* and when set via *setrlimit()*.

In most cases, resource limits are enforced for both privileged and unprivileged processes. They are inherited by child processes created via *fork()* and are preserved across an *exec()*.

The values that can be specified for the *resource* argument of *getrlimit()* and *setrlimit()* are summarized in Table 36-1 and detailed in Section 36.3.

Although a resource limit is a per-process attribute, in some cases, the limit is measured against not just that process's consumption of the corresponding resource, but also against the sum of resources consumed by all processes with the same real user ID. The `RLIMIT_NPROC` limit, which places a limit on the number of processes that can be created, is a good example of the rationale for this approach. Applying this limit against only the number of children that the process itself created would be ineffective, since each child that the process created would also be able to create further children, which could create more children, and so on. Instead, the limit is measured against the count of all processes that have the same real user ID. Note, however, that the resource limit is checked only in the processes where it has been set (i.e., the process itself and its descendants, which inherit the limit). If another process owned by the same real user ID has not set the limit (i.e., the limit is infinite) or has set a different limit, then that process's capacity to create children will be checked according to the limit that it has set.

As we describe each resource limit below, we note those limits that are measured against the resources consumed by all processes with the same real user ID. If not otherwise specified, then a resource limit is measured only against the process's own consumption of the resource.

Be aware that, in many cases, the shell commands for getting and setting resource limits (*ulimit* in *bash* and the Korn shell, and *limit* in the C shell) use different units from those used in *getrlimit()* and *setrlimit()*. For example, the shell commands typically express the limits on the size of various memory segments in kilobytes.

Table 36-1: Resource values for *getrlimit()* and *setrlimit()*

resource	**Limit on**	**SUSv3**
RLIMIT_AS	Process virtual memory size (bytes)	•
RLIMIT_CORE	Core file size (bytes)	•
RLIMIT_CPU	CPU time (seconds)	•
RLIMIT_DATA	Process data segment (bytes)	•
RLIMIT_FSIZE	File size (bytes)	•
RLIMIT_MEMLOCK	Locked memory (bytes)	
RLIMIT_MSGQUEUE	Bytes allocated for POSIX message queues for real user ID (since Linux 2.6.8)	
RLIMIT_NICE	Nice value (since Linux 2.6.12)	
RLIMIT_NOFILE	Maximum file descriptor number plus one	•
RLIMIT_NPROC	Number of processes for real user ID	
RLIMIT_RSS	Resident set size (bytes; not implemented)	
RLIMIT_RTPRIO	Realtime scheduling priority (since Linux 2.6.12)	
RLIMIT_RTTIME	Realtime CPU time (microseconds; since Linux 2.6.25)	
RLIMIT_SIGPENDING	Number of queued signals for real user ID (since Linux 2.6.8)	
RLIMIT_STACK	Size of stack segment (bytes)	•

Example program

Before going into the specifics of each resource limit, we look at a simple example of the use of resource limits. Listing 36-2 defines the function *printRlimit()*, which displays a message, along with the soft and hard limits for a specified resource.

The *rlim_t* data type is typically represented in the same way as *off_t*, to handle the representation of RLIMIT_FSIZE, the file size resource limit. For this reason, when printing *rlim_t* values (as in Listing 36-2), we cast them to *long long* and use the %lld *printf()* specifier, as explained in Section 5.10.

The program in Listing 36-3 calls *setrlimit()* to set the soft and hard limits on the number of processes that a user may create (RLIMIT_NPROC), uses the *printRlimit()* function of Listing 36-2 to display the limits before and after the change, and then creates as many processes as possible. When we run this program, setting the soft limit to 30 and the hard limit to 100, we see the following:

```
$ ./rlimit_nproc 30 100
Initial maximum process limits:  soft=1024; hard=1024
New maximum process limits:      soft=30; hard=100
Child 1 (PID=15674) started
Child 2 (PID=15675) started
Child 3 (PID=15676) started
Child 4 (PID=15677) started
ERROR [EAGAIN Resource temporarily unavailable] fork
```

In this example, the program managed to create only 4 new processes, because 26 processes were already running for this user.

Listing 36-2: Displaying process resource limits

procres/print_rlimit.c

```
#include <sys/resource.h>
#include "print_rlimit.h"           /* Declares function defined here */
#include "tlpi_hdr.h"

int                     /* Print 'msg' followed by limits for 'resource' */
printRlimit(const char *msg, int resource)
{
    struct rlimit rlim;

    if (getrlimit(resource, &rlim) == -1)
        return -1;

    printf("%s soft=", msg);
    if (rlim.rlim_cur == RLIM_INFINITY)
        printf("infinite");
#ifdef RLIM_SAVED_CUR               /* Not defined on some implementations */
    else if (rlim.rlim_cur == RLIM_SAVED_CUR)
        printf("unrepresentable");
#endif
    else
        printf("%lld", (long long) rlim.rlim_cur);

    printf("; hard=");
    if (rlim.rlim_max == RLIM_INFINITY)
        printf("infinite\n");
#ifdef RLIM_SAVED_MAX               /* Not defined on some implementations */
    else if (rlim.rlim_max == RLIM_SAVED_MAX)
        printf("unrepresentable");
#endif
    else
        printf("%lld\n", (long long) rlim.rlim_max);

    return 0;
}
```

procres/print_rlimit.c

Listing 36-3: Setting the `RLIMIT_NPROC` resource limit

procres/rlimit_nproc.c

```
#include <sys/resource.h>
#include "print_rlimit.h"               /* Declaration of printRlimit() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct rlimit rl;
    int j;
    pid_t childPid;
```

procres/rlimit_nproc.c

```
    if (argc < 2 || argc > 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s soft-limit [hard-limit]\n", argv[0]);

    printRlimit("Initial maximum process limits: ", RLIMIT_NPROC);

    /* Set new process limits (hard == soft if not specified) */

    rl.rlim_cur = (argv[1][0] == 'i') ? RLIM_INFINITY :
                                getInt(argv[1], 0, "soft-limit");
    rl.rlim_max = (argc == 2) ? rl.rlim_cur :
                (argv[2][0] == 'i') ? RLIM_INFINITY :
                                getInt(argv[2], 0, "hard-limit");
    if (setrlimit(RLIMIT_NPROC, &rl) == -1)
        errExit("setrlimit");

    printRlimit("New maximum process limits:     ", RLIMIT_NPROC);

    /* Create as many children as possible */

    for (j = 1; ; j++) {
        switch (childPid = fork()) {
        case -1: errExit("fork");

        case 0: _exit(EXIT_SUCCESS);            /* Child */

        default:        /* Parent: display message about each new child
                           and let the resulting zombies accumulate */
            printf("Child %d (PID=%ld) started\n", j, (long) childPid);
            break;
        }
    }
}
```

procres/rlimit_nproc.c

Unrepresentable limit values

In some programming environments, the *rlim_t* data type may not be able to represent the full range of values that could be maintained for a particular resource limit. This may be the case on a system that offers multiple programming environments in which the size of the *rlim_t* data type differs. Such systems can arise if a large-file compilation environment with a 64-bit *off_t* is added to a system on which *off_t* was traditionally 32 bits. (In each environment, *rlim_t* would be the same size as *off_t*.) This leads to the situation where a program with a small *rlim_t* can, after being execed by a program with a 64-bit *off_t*, inherit a resource limit (e.g., the file size limit) that is greater than the maximum *rlim_t* value.

To assist portable applications in handling the possibility that a resource limit may be unrepresentable, SUSv3 specifies two constants to indicate unrepresentable limit values: `RLIM_SAVED_CUR` and `RLIM_SAVED_MAX`. If a soft resource limit can't be represented in *rlim_t*, then *getrlimit()* will return `RLIM_SAVED_CUR` in the *rlim_cur* field. `RLIM_SAVED_MAX` performs an analogous function for an unrepresentable hard limit returned in the *rlim_max* field.

If all possible resource limit values can be represented in *rlim_t*, then SUSv3 permits an implementation to define RLIM_SAVED_CUR and RLIM_SAVED_MAX to be the same as RLIM_INFINITY. This is how these constants are defined on Linux, implying that all possible resource limit values can be represented in *rlim_t*. However, this is not the case on 32-bit architectures such as x86-32. On those architectures, in a large-file compilation environment (i.e., setting the _FILE_OFFSET_BITS feature test macro to 64 as described in Section 5.10), the *glibc* definition of *rlim_t* is 64 bits wide, but the kernel data type for representing a resource limit is *unsigned long*, which is only 32 bits wide. Current versions of *glibc* deal with this situation as follows: if a program compiled with _FILE_OFFSET_BITS=64 tries to set a resource limit to a value larger than can be represented in a 32-bit *unsigned long*, then the *glibc* wrapper for *setrlimit()* silently converts the value to RLIM_INFINITY. In other words, the requested setting of the resource limit is not honored.

> Because utilities that handle files are normally compiled with _FILE_OFFSET_BITS=64 in many x86-32 distributions, the failure to honor resource limits larger that the value that can be represented in 32 bits is a problem that can affect not only application programmers, but also end users.
>
> One could argue that it might be better for the *glibc setrlimit()* wrapper to give an error if the requested resource limit exceeds the capacity of a 32-bit *unsigned long*. However, the fundamental problem is a kernel limitation, and the behavior described in the main text is the approach that the *glibc* developers have taken to dealing with it.

36.3 Details of Specific Resource Limits

In this section, we provide details on each of the resource limits available on Linux, noting those that are Linux-specific.

RLIMIT_AS

The RLIMIT_AS limit specifies the maximum size for the process's virtual memory (address space), in bytes. Attempts (*brk()*, *sbrk()*, *mmap()*, *mremap()*, and *shmat()*) to exceed this limit fail with the error ENOMEM. In practice, the most common place where a program may hit this limit is in calls to functions in the *malloc* package, which make use of *sbrk()* and *mmap()*. Upon encountering this limit, stack growth can also fail with the consequences listed below for RLIMIT_STACK.

RLIMIT_CORE

The RLIMIT_CORE limit specifies the maximum size, in bytes, for core dump files produced when a process is terminated by certain signals (Section 22.1). Production of a core dump file will stop at this limit. Specifying a limit of 0 prevents creation of core dump files, which is sometimes useful because core dump files can be very large, and end users usually don't know what to do with them. Another reason for disabling core dumps is security—to prevent the contents of a program's memory from being dumped to disk. If the RLIMIT_FSIZE limit is lower than this limit, core dump files are limited to RLIMIT_FSIZE bytes.

RLIMIT_CPU

The RLIMIT_CPU limit specifies the maximum number of seconds of CPU time (in both system and user mode) that can be used by the process. SUSv3 requires that the SIGXCPU signal be sent to the process when the soft limit is reached, but leaves other details unspecified. (The default action for SIGXCPU is to terminate a process with a core dump.) It is possible to establish a handler for SIGXCPU that does whatever processing is desired and then returns control to the main program. Thereafter, (on Linux) SIGXCPU is sent once per second of consumed CPU time. If the process continues executing until the hard CPU limit is reached, then the kernel sends it a SIGKILL signal, which always terminates the process.

UNIX implementations vary in the details of how they deal with processes that continue consuming CPU time after handling a SIGXCPU signal. Most continue to deliver SIGXCPU at regular intervals. If aiming for portable use of this signal, we should code an application so that, on first receipt of this signal, it does whatever cleanup is required and terminates. (Alternatively, the program could change the resource limit after receiving the signal.)

RLIMIT_DATA

The RLIMIT_DATA limit specifies the maximum size, in bytes, of the process's data segment (the sum of the initialized data, uninitialized data, and heap segments described in Section 6.3). Attempts (*sbrk()* and *brk()*) to extend the data segment (program break) beyond this limit fail with the error ENOMEM. As with RLIMIT_AS, the most common place where a program may hit this limit is in calls to functions in the *malloc* package.

RLIMIT_FSIZE

The RLIMIT_FSIZE limit specifies the maximum size of files that the process may create, in bytes. If a process attempts to extend a file beyond the soft limit, it is sent a SIGXFSZ signal, and the system call (e.g., *write()* or *truncate()*) fails with the error EFBIG. The default action for SIGXFSZ is to terminate a process and produce a core dump. It is possible to instead catch this signal and return control to the main program. However, any further attempt to extend the file will yield the same signal and error.

RLIMIT_MEMLOCK

The RLIMIT_MEMLOCK limit (BSD-derived; absent from SUSv3 and available only on Linux and the BSDs) specifies the maximum number of bytes of virtual memory that a process may lock into physical memory, to prevent the memory from being swapped out. This limit affects the *mlock()* and *mlockall()* system calls, and the locking options for the *mmap()* and *shmctl()* system calls. We describe the details in Section 50.2.

If the MCL_FUTURE flag is specified when calling *mlockall()*, then the RLIMIT_MEMLOCK limit may also cause later calls to *brk()*, *sbrk()*, *mmap()*, or *mremap()* to fail.

RLIMIT_MSGQUEUE

The RLIMIT_MSGQUEUE limit (Linux-specific; since Linux 2.6.8) specifies the maximum number of bytes that can be allocated for POSIX message queues for the real user

ID of the calling process. When a POSIX message queue is created using *mq_open()*, bytes are deducted against this limit according to the following formula:

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

In this formula, *attr* is the *mq_attr* structure that is passed as the fourth argument to *mq_open()*. The addend that includes *sizeof(struct msg_msg *)* ensures that the user can't queue an unlimited number of zero-length messages. (The *msg_msg* structure is a data type used internally by the kernel.) This is necessary because, although zero-length messages contain no data, they do consume some system memory for bookkeeping overhead.

The RLIMIT_MSGQUEUE limit affects only the calling process. Other processes belonging to this user are not affected unless they also set this limit or inherit it.

RLIMIT_NICE

The RLIMIT_NICE limit (Linux-specific; since Linux 2.6.12) specifies a ceiling on the nice value that may be set for this process using *sched_setscheduler()* and *nice()*. The ceiling is calculated as *20 − rlim_cur*, where *rlim_cur* is the current RLIMIT_NICE soft resource limit. Refer to Section 35.1 for further details.

RLIMIT_NOFILE

The RLIMIT_NOFILE limit specifies a number one greater than the maximum file descriptor number that a process may allocate. Attempts (e.g., *open()*, *pipe()*, *socket()*, *accept()*, *shm_open()*, *dup()*, *dup2()*, *fcntl(F_DUPFD)*, and *epoll_create()*) to allocate descriptors beyond this limit fail. In most cases, the error is EMFILE, but for *dup2(fd, newfd)* it is EBADF, and for *fcntl(fd, F_DUPFD, newfd)* with *newfd* is greater than or equal to the limit, it is EINVAL.

Changes to the RLIMIT_NOFILE limit are reflected in the value returned by *sysconf(_SC_OPEN_MAX)*. SUSv3 permits, but doesn't require, an implementation to return different values for a call to *sysconf(_SC_OPEN_MAX)* before and after changing the RLIMIT_NOFILE limit; other implementations may not behave the same as Linux on this point.

> SUSv3 states that if an application sets the soft or hard RLIMIT_NOFILE limit to a value less than or equal to the number of the highest file descriptor that the process currently has open, unexpected behavior may occur.
>
> On Linux, we can check which file descriptors a process currently has open by using *readdir()* to scan the contents of the /proc/*PID*/fd directory, which contains symbolic links for each of the file descriptors currently opened by the process.

The kernel imposes a ceiling on the value to which the RLIMIT_NOFILE limit may be raised. In kernels before 2.6.25, this ceiling is a hard-coded value defined by the kernel constant NR_OPEN, whose value is 1,048,576. (A kernel rebuild is required to raise this ceiling.) Since kernel 2.6.25, the limit is defined by the value in the Linux-specific /proc/sys/fs/nr_open file. The default value in this file is 1,048,576; this can be modified by the superuser. Attempts to set the soft or hard RLIMIT_NOFILE limit higher than the ceiling value yield the error EPERM.

There is also a system-wide limit on the total number of files that may be opened by all processes. This limit can be retrieved and modified via the Linux-specific `/proc/sys/fs/file-max` file. (Referring to Section 5.4, we can define `file-max` more precisely as a system-wide limit on the number of open file descriptions.) Only privileged (`CAP_SYS_ADMIN`) processes can exceed the `file-max` limit. In an unprivileged process, a system call that encounters the `file-max` limit fails with the error `ENFILE`.

RLIMIT_NPROC

The `RLIMIT_NPROC` limit (BSD-derived; absent from SUSv3 and available only on Linux and the BSDs) specifies the maximum number of processes that may be created for the real user ID of the calling process. Attempts (*fork()*, *vfork()*, and *clone()*) to exceed this limit fail with the error `EAGAIN`.

The `RLIMIT_NPROC` limit affects only the calling process. Other processes belonging to this user are not affected unless they also set or inherit this limit. This limit is not enforced for privileged (`CAP_SYS_ADMIN` or `CAP_SYS_RESOURCE`) processes.

> Linux also imposes a system-wide limit on the number of processes that can be created by all users. On Linux 2.4 and later, the Linux-specific `/proc/sys/kernel/threads-max` file can be used to retrieve and modify this limit.
>
> To be precise, the `RLIMIT_NPROC` resource limit and the `threads-max` file are actually limits on the numbers of threads that can be created, rather than the number of processes.

The manner in which the default value for the `RLIMIT_NPROC` resource limit is set has varied across kernel versions. In Linux 2.2, it was calculated according to a fixed formula. In Linux 2.4 and later, it is calculated using a formula based on the amount of available physical memory.

> SUSv3 doesn't specify the `RLIMIT_NPROC` resource limit. The SUSv3-mandated method for retrieving (but not changing) the maximum number of processes permitted to a user ID is via the call *sysconf(_SC_CHILD_MAX)*. This *sysconf()* call is supported on Linux, but in kernel versions before 2.6.23, the call does not return accurate information—it always returns the value 999. Since Linux 2.6.23 (and with *glibc* 2.4 and later), this call correctly reports the limit (by checking the value of the `RLIMIT_NPROC` resource limit).
>
> There is no portable way of discovering how many processes have already been created for a specific user ID. On Linux, we can try scanning all of the `/proc/`*PID*`/status` files on the system and examining the information under the `Uid` entry (which lists the four process user IDs in the order: real, effective, saved set, and file system) in order to estimate the number of processes currently owned by a user. Be aware, however, that by the time we have completed such a scan, this information may already have changed.

RLIMIT_RSS

The `RLIMIT_RSS` limit (BSD-derived; absent from SUSv3, but widely available) specifies the maximum number of pages in the process's resident set; that is, the total number of virtual memory pages currently in physical memory. This limit is provided on Linux, but it currently has no effect.

In older Linux 2.4 kernels (up to and including 2.4.29), RLIMIT_RSS did have an effect on the behavior of the *madvise()* MADV_WILLNEED operation (Section 50.4). If this operation could not be performed as a result of encountering the RLIMIT_RSS limit, the error EIO was returned in *errno*.

RLIMIT_RTPRIO

The RLIMIT_RTPRIO limit (Linux-specific; since Linux 2.6.12) specifies a ceiling on the realtime priority that may be set for this process using *sched_setscheduler()* and *sched_setparam()*. Refer to Section 35.3.2 for further details.

RLIMIT_RTTIME

The RLIMIT_RTTIME limit (Linux-specific; since Linux 2.6.25) specifies the maximum amount of CPU time in microseconds that a process running under a realtime scheduling policy may consume without sleeping (i.e., performing a blocking system call). The behavior if this limit is reached is the same as for RLIMIT_CPU: if the process reaches the soft limit, then a SIGXCPU signal is sent to the process, and further SIGXCPU signals are sent for each additional second of CPU time consumed. On reaching the hard limit, a SIGKILL signal is sent. Refer to Section 35.3.2 for further details.

RLIMIT_SIGPENDING

The RLIMIT_SIGPENDING limit (Linux-specific; since Linux 2.6.8) specifies the maximum number of signals that may be queued for the real user ID of the calling process. Attempts (*sigqueue()*) to exceed this limit fail with the error EAGAIN.

The RLIMIT_SIGPENDING limit affects only the calling process. Other processes belonging to this user are not affected unless they also set or inherit this limit.

As initially implemented, the default value for the RLIMIT_SIGPENDING limit was 1024. Since kernel 2.6.12, the default value has been changed to be the same as the default value for RLIMIT_NPROC.

For the purposes of checking the RLIMIT_SIGPENDING limit, the count of queued signals includes both realtime and standard signals. (Standard signals can be queued only once to a process.) However, this limit is enforced only for *sigqueue()*. Even if the number of signals specified by this limit has already been queued to processes belonging to this real user ID, it is still possible to use *kill()* to queue one instance of each of the signals (including realtime signals) that are not already queued to a process.

From kernel 2.6.12 onward, the SigQ field of the Linux-specific /proc/*PID*/status file displays the current and maximum number of queued signals for the real user ID of the process.

RLIMIT_STACK

The RLIMIT_STACK limit specifies the maximum size of the process stack, in bytes. Attempts to grow the stack beyond this limit result in the generation of a SIGSEGV signal for the process. Since the stack is exhausted, the only way to catch this signal is by establishing an alternate signal stack, as described in Section 21.3.

Since Linux 2.6.23, the RLIMIT_STACK limit also determines the amount of space available for holding the process's command-line arguments and environment variables. See the *execve(2)* manual page for details.

36.4 Summary

Processes consume various system resources. The *getrusage()* system call allows a process to monitor certain of the resources consumed by itself and by its children.

The *setrlimit()* and *getrlimit()* system calls allow a process to set and retrieve limits on its consumption of various resources. Each resource limit has two components: a soft limit, which is what the kernel enforces when checking a process's resource consumption, and a hard limit, which acts as a ceiling on the value of the soft limit. An unprivileged process can set the soft limit for a resource to any value in the range from 0 up to the hard limit, but can only lower the hard limit. A privileged process can make any changes to either limit value, as long as the soft limit is less than or equal to the hard limit. If a process encounters a soft limit, it is typically informed of the fact either by receiving a signal or via failure of the system call that attempts to exceed the limit.

36.5 Exercises

36-1. Write a program that shows that the *getrusage()* `RUSAGE_CHILDREN` flag retrieves information about only the children for which a *wait()* call has been performed. (Have the program create a child process that consumes some CPU time, and then have the parent call *getrusage()* before and after calling *wait()*.)

36-2. Write a program that executes a command and then displays its resource usage. This is analogous to what the *time(1)* command does. Thus, we would use this program as follows:

```
$ ./rusage command arg...
```

36-3. Write programs to determine what happens if a process's consumption of various resources already exceeds the soft limit specified in a call to *setrlimit()*.