8

System Information and Resource Limits

8.1 Introduction

Solaris is a very scalable operating system capable of running on tiny uniprocessor workstations with as little as 16 MB of RAM (such as the SPARCstation 2) or large servers with more than 100 multicore processors and more than 512 GB of RAM (for example, the Sun Fire 25K). Remarkably, all this is achieved with complete binary compatibility throughout the whole range. Because of this, applications can't make any assumption about the capabilities of the machine they are running on.

Solaris has a number of facilities that enable a process to gather information about the kind of machine it is running on, like the number of processors, the amount of memory, and the architecture. Most of the facilities offered by Solaris are also available on other flavours of UNIX, although some of the specifics may vary.

UNIX has always offered a number of parameters that can be tuned by the system administrator, so that systems perform better under a particular workload. Some of these parameters are configured into the kernel, and on some versions of UNIX, changing them requires the kernel to be recompiled. Solaris, however, does not need to be recompiled if a kernel parameter is changed. Given that access to the Solaris source code outside of Sun is very restricted, this is a good thing. Instead, Solaris kernel tuning is performed by editing the file `/etc/system` (the details of which are beyond the scope of this text) and rebooting.

Historically, many of the adjustable parameters were defined in header files. This meant that whenever they were changed, the programs using them had to be recompiled. As the number of third party programs grew, it became clear that this idea wasn't very flexible. Instead, many of the parameters could be determined (and in some cases, changed) by using standard library calls.

The various UNIX standardization efforts (like POSIX, the Open Group, and the Single UNIX Specification) have also helped this situation by specifying standard interfaces and resource names.

In this chapter, we look at the facilities available to systems programmers that allow their programs to determine information about the machine they are running on. We'll also discuss how to get or set systemwide and per-process resource limits, and how a process can determine how much of a resource it has consumed.

8.2 System Information and Identification

We can call a number of functions to ascertain information about the machine a process is running on. Depending on the function called, we can retrieve the machine's hostname, its hostid, the OS version, and so on.

The `uname` Function

Each UNIX system maintains a number of pieces of information about itself, including the hostname, the operating system name and version, and the hardware type. We can use the `uname` function to obtain this information.

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

int uname (struct utsname *name);
```

Returns: non-negative if OK, –1 on error

The `uname` function stores information identifying the current operating system and machine in the `utsname` structure pointed to by *name*. The `utsname` structure has the following members:

```
struct utsname {
    char    sysname [SYS_NMLN];     /* Name of OS */
    char    nodename [SYS_NMLN];    /* Hostname */
    char    release [SYS_NMLN];     /* The OS release */
    char    version [SYS_NMLN];     /* The OS version */
    char    machine [SYS_NMLN];     /* The machine type */
};
```

The structure members have the following definitions:

`sysname` This is a NUL-terminated string containing the name of the operating system. On machines running Solaris, this field will be "SunOS".

`nodename` This is the NUL-terminated name of the machine as it is known on the network (i.e., its hostname).

`release` This is a NUL-terminated string containing the release number. On machines running Solaris, this field will be the SunOS version number (e.g., 5.9).

`version` This is a NUL-terminated string containing the operating system version. On machines running Solaris, this field contains the string "Generic", possibly followed by a number that specifies the kernel jumbo patch (KJP) number and revision. An example would be "Generic_112233-12", which is a Solaris 9 machine running revision 12 of the Solaris 9 KJP.

`machine` This is a NUL-terminated string identifying the type of hardware (i.e., the machine architecture). On Sun hardware, this would typically be something like "sun4u".

The `sysinfo` Function

The `uname` function can be used to retrieve only some system information. The `sysinfo` function performs a similar task, but can provide more information and enables some system information to be set.

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

long sysinfo (int command, char *buf, long count);
```

Returns: non-negative if OK, –1 on error

The `sysinfo` function copies information about the operating system, as requested by *command*, into the buffer pointed to by *buf*. The *count* parameter indicates the size of the buffer (257 is a good value to use for *count*, as it is likely to cover all strings returned). Upon successful completion, `sysinfo` returns the buffer size (in bytes) required to hold the complete value, including the terminating NUL. If this value is no greater than *count*, the entire string was copied. Otherwise, the string copied into *buf* has been truncated to *count* – 1 bytes, plus the terminating NUL.

The values of *command* are as follows (note that some values of *command* allow system information to be set from *buf*).

`SI_SYSNAME` Copy the name of the operating system into the buffer pointed to by *buf*. This is the same value returned by `uname` in the `sysname` field; on a Solaris system, it will be "SunOS".

`SI_HOSTNAME` Copy the name of the machine into the buffer pointed to by *buf*. This is the same value returned by `uname` in the `nodename` field. The name may or may not be fully qualified. Internet hostnames may be up to 256 bytes in length, plus the terminating NUL.

`SI_SET_HOSTNAME` If the effective user ID of the calling process is zero, this copies the NUL-terminated string pointed to by *buf* into the kernel. Subsequent calls to `sysinfo` with *command* set to `SI_HOSTNAME` will return the new value.

SI_RELEASE	Copy the release of the operating system into the buffer pointed to by *buf*. This is the same value returned by `uname` in the `release` field.
SI_VERSION	Copy the operating system version into the buffer pointed to by *buf*. This is the same value returned by `uname` in the `version` field.
SI_MACHINE	Copy a string that represents the machine type into the buffer pointed to by *buf*. This is the same value returned by `uname` in the `machine` field.
SI_ARCHITECTURE	Copy into the buffer pointed to by *buf* a string that represents the basic instruction architecture of the machine. Examples of values returned on a system running Solaris would be "sparc" or "i386".
SI_ISALIST	Copy a string that represents the variant instruction set architectures executable on the current system. The names are separated by spaces and are listed in order of performance (from best to worst). In other words, earlier named instruction sets may contain more instructions than later named ones. A program compiled for an earlier named instruction set will likely run faster on that machine than the same program compiled using a later named instruction set. For example, given the list "sparcv9 sparcv8 sparcv7", a program compiled for sparcv9 will likely run faster on a machine using the sparcv9 architecture than the same program compiled for sparcv7. Programs compiled for an instruction set that does not appear in this list will at best suffer a performance degradation, or at worst will not run at all.
SI_PLATFORM	A string describing the specific hardware model is copied into the buffer pointed to by *buf*. For example, "SUNW,Sun-Blade-100", "SUNW,Ultra-60", "SUNW,S240", or "SUNW,SPARCstation-20".
SI_HW_PROVIDER	This copies the name of the hardware manufacturer into the buffer pointed to by *buf*.
SI_HW_SERIAL	This copies into the buffer pointed to by *buf* a string that is the ASCII representation of the hardware-specific serial number of the machine that executes the function. On Sun hardware, this will usually be the hostid, which is not the same thing as the serial number of the actual machine.

A commonly asked Solaris programming question is "How do I programmatically obtain the serial number of a given machine?". The answer is: there's no way to do this. The only place the serial number of a machine is stored is on the case. The best we can hope to do is obtain the hostid.

It is not expected that manufacturers will issue the same "serial number" to more than one physical machine, so the pair of strings returned by `SI_HW_PROVIDER` and `SI_HW_SERIAL` is likely to be unique.

`SI_SRPC_DOMAIN` This copies the Secure Remote Procedure Call domain into the buffer pointed to by *buf*.

`SI_SET_SRPC_DOMAIN` If the effective user ID of the calling process is zero, this copies the NUL-terminated string pointed to by *buf* into the kernel. Subsequent calls to `sysinfo` with *command* set to `SI_SRPC_DOMAIN` will return the new value.

`SI_DHCP_CACHE` This copies into the buffer pointed to by *buf* an ASCII string that consists of the ASCII hexadecimal encoding of the name of the interface configured by `boot`(1M) followed by the `DHCPACK` reply from the server. This command is intended for use only by the `dhcpagent`(1M) DHCP client daemon for the purpose of adopting the DHCP maintenance of the interface configured by `boot`.

Not all values for *command* are supported on all versions of Solaris. Figure 8.1 summarizes their availability.

Example: Obtaining system information

Program 8.1 prints out the information obtained by the `uname` and `sysinfo` functions.

Notice how we've used the ISO C string creation operator (#) in our `psysinfo` macro, to generate the string version of each resource. When we say

```
psysinfo (SI_SYSNAME);
```

this is expanded by the C preprocessor to

```
print_sysinfo ("SI_SYSNAME", SI_SYSNAME);
```

Running Program 8.1 on one of the author's systems gave the following results:

```
$ ./sysinfo
Info from uname:
  sysname: SunOS
  nodename: grover
  release: 5.9
  version: Generic_112233-07
  machine: sun4u
```

Command	Solaris version					
	2.5	2.5.1	2.6	7	8	9
SI_SYSNAME	•	•	•	•	•	•
SI_HOSTNAME	•	•	•	•	•	•
SI_SET_HOSTNAME	•	•	•	•	•	•
SI_RELEASE	•	•	•	•	•	•
SI_VERSION	•	•	•	•	•	•
SI_MACHINE	•	•	•	•	•	•
SI_ARCHITECTURE	•	•	•	•	•	•
SI_ISALIST			•	•	•	•
SI_PLATFORM	•	•	•	•	•	•
SI_HW_PROVIDER	•	•	•	•	•	•
SI_HW_SERIAL	•	•	•	•	•	•
SI_SRPC_DOMAIN	•	•	•	•	•	•
SI_SET_SRPC_DOMAIN	•	•	•	•	•	•
SI_DHCP_CACHE					•	•

Figure 8.1 Availability of `sysinfo` commands.

```
Info from sysinfo
  SI_SYSNAME: SunOS
  SI_HOSTNAME: grover
  SI_RELEASE: 5.9
  SI_VERSION: Generic_Patch
  SI_MACHINE: sun4u
  SI_ARCHITECTURE: sparc
  SI_ISALIST: sparcv9+vis sparcv9 sparcv8plus+vis sparcv8plus sparcv8
      sparcv8-fsmuld sparcv7 sparc
  SI_PLATFORM: SUNW,Sun-Blade-100
  SI_HW_PROVIDER: Sun_Microsystems
  SI_HW_SERIAL: 2198662555
  SI_SRPC_DOMAIN:
  SI_DHCP_CACHE:
$ hostid
830ced9b
$ bc
obase=16
2198662555
830CED9B
```

Notice that we had to wrap the `SI_ISALIST` line to fit it onto the page. Also note that we used the `hostid` and `bc` commands to confirm that the serial number returned by the `SI_HW_SERIAL` command is the same as the hostid.

The `gethostname` and `sethostname` Functions

If we are just interested in retrieving the hostname of a machine, we can use the `gethostname` and `sethostname` functions.

sys_info/sysinfo.c

```
 1 #include <stdio.h>
 2 #include <sys/utsname.h>
 3 #include <sys/systeminfo.h>
 4 #include "ssp.h"

 5 #define psysinfo(name) print_sysinfo (#name, name)

 6 static void print_sysinfo (const char *name, int command);

 7 int main (void)
 8 {
 9     struct utsname utsname;

10     if (uname (&utsname) == -1)
11         err_msg ("uname failed");
12     printf ("Info from uname:\n");
13     printf ("  sysname: %s\n", utsname.sysname);
14     printf ("  nodename: %s\n", utsname.nodename);
15     printf ("  release: %s\n", utsname.release);
16     printf ("  version: %s\n", utsname.version);
17     printf ("  machine: %s\n\n", utsname.machine);

18     printf ("Info from sysinfo\n");
19     psysinfo (SI_SYSNAME);
20     psysinfo (SI_HOSTNAME);
21     psysinfo (SI_RELEASE);
22     psysinfo (SI_VERSION);
23     psysinfo (SI_MACHINE);
24     psysinfo (SI_ARCHITECTURE);
25 #ifdef SI_ISALIST
26     psysinfo (SI_ISALIST);
27 #endif
28     psysinfo (SI_PLATFORM);
29     psysinfo (SI_HW_PROVIDER);
30     psysinfo (SI_HW_SERIAL);
31     psysinfo (SI_SRPC_DOMAIN);
32 #ifdef SI_DHCP_CACHE
33     psysinfo (SI_DHCP_CACHE);
34 #endif

35     return (0);
36 }

37 static void print_sysinfo (const char *name, int command)
38 {
39     char buf [257];

40     if (sysinfo (command, buf, 257) == -1)
41         err_msg ("sysinfo (%s) failed", name);
42     printf ("  %s: %s\n", name, buf);
43 }
```

sys_info/sysinfo.c

Program 8.1 System information provided `uname` and `sysinfo`.

```
#include <unistd.h>

int gethostname (char *name, int namelen);

int sethostname (char *name, int namelen);
```

Both return: 0 if OK, –1 on error

The `gethostname` function copies the hostname of the current processor into the buffer pointed to by *name*, which is *namelen* bytes in size. The returned name is NUL-terminated unless insufficient space is provided in *name*.

This is the same value as those returned in the `nodename` member of the `utsname` structure returned by `uname`, and by calling `sysinfo` using the `SI_HOSTNAME` command.

If the effective user ID of the calling process is zero, the `sethostname` function sets the hostname of the current processor to that in the buffer pointed to by *name*, which is *namelen* bytes in size.

Hostnames are limited to `MAXHOSTNAMELEN` bytes, which is defined in `<netdb.h>`.

The `gethostid` Function

If we are just interested in retrieving the hostid of a machine, we can use the `gethostid` function.

```
#include <unistd.h>

long gethostid (void);
```

Returns: the host's hostid

The `gethostid` function returns the 32-bit identifier for the current host. In Sun machines, this identifier is taken from the CPU board's ID PROM. It is not guaranteed to be unique.

This is the same value as that returned by calling `sysinfo` using the `SI_HW_SERIAL` command.

8.3 System Resource Limits

There are many magic numbers and constants that are defined by a UNIX implementation. Historically, these have been hard coded into programs, or determined in an ad hoc manner. Part of the UNIX standardization effort of POSIX and the SUS was to come up with a more portable method of determining these magic numbers and any implementation-defined constants and limits.

We need to consider three categories:

1. Compile time options. An example of this would be whether job control is supported.
2. Compile time limits. An example of this would be the size of a pointer.
3. Run time limits. An example of this would be the memory page size.

Compile time options and limits can be defined in headers, which can be included when the program is compiled. But run time limits require the process to call a function to determine the value of the requested limit.

Run time limits can be subdivided into two types:

1. Those that are associated with a file or directory
2. Those that are not associated with a file or directory

As an example of the former, consider the maximum length of a filename. Older versions of Solaris supported the System V file system, `s5fs`, as well as the usual UFS file system. The `s5fs` file system allows only 14 characters for a filename, whereas UFS allows 256. If we want our code to work with `s5fs` and UFS, we can't rely on compile time limits, and must use run time limits.

Because there are two types of run time limits, two types of function are used to determine them; `sysconf` is used to determine run time limits that are not associated with a file or directory, and `pathconf` and `fpathconf` are used to determine run time limits that are associated with a file or directory.

The `sysconf` Function

We use the `sysconf` function to determine the value of configurable system variables.

```
#include <unistd.h>

long sysconf (int name);
```

Returns: see text

The `sysconf` function returns the current value of a configurable system limit of option (variable). On successful completion, `sysconf` returns the current value of the variable. This value will not be more restrictive than the corresponding value available at compile time in `<limits.h>`, `<unistd.h>`, or `<time.h>`. The value of most of these variables will not change during the lifetime of the calling process.

In the event of an error, −1 is returned and `errno` is set appropriately. We should be aware that −1 is also a legal return value when no error occurs, so programs wanting to check for errors should set `errno` to 0 before calling `sysconf`, and then check it if −1 is returned.

The parameter *name* specifies the name of the run time limit we are interested in, and must be one of the following values (the values returned are integers unless otherwise stated):

_SC_2_C_BIND	This is a Boolean value that indicates whether the POSIX C language bindings are supported. _POSIX2_C_BIND is the returned value.
_SC_2_C_DEV	This is a Boolean value that indicates whether the POSIX C language development utilities options is supported. The returned value is _POSIX2_C_DEV.
_SC_2_C_VERSION	This indicates which version of the ISO POSIX.2 standard (Commands) is supported. The returned value is _POSIX2_C_VERSION.
_SC_2_CHAR_TERM	This is a Boolean value that indicates whether at least one terminal is supported. The returned value is _POSIX2_CHAR_TERM.
_SC_2_FORT_DEV	This is a Boolean value that indicates whether the FORTRAN development utilities option is supported. _POSIX2_FORT_DEV is the returned value.
_SC_2_FORT_RUN	This is a Boolean value that indicates whether the FORTRAN run time utilities option is supported. _POSIX2_FORT_RUN is the returned value.
_SC_2_LOCALEDEF	This is a Boolean value that indicates whether the creation of locales using the localedef utility is supported. The returned value is _POSIX2_LOCALEDEF.
_SC_2_SW_DEV	This is a Boolean value that indicates whether the software development utilities option is supported. _POSIX2_SW_DEV is the value returned.
_SC_2_UPE	This is a Boolean value that indicates whether the user portability utilities option is supported. _POSIX2_UPE is the value returned.
_SC_2_VERSION	This indicates which version of the ISO POSIX.2 standard (C language binding) is supported. The returned value is _POSIX2_VERSION.
_SC_AIO_LISTIO_MAX	This indicates the maximum number of I/O operations in a single call that is supported. The returned value is AIO_LISTIO_MAX.
_SC_AIO_MAX	This indicates the maximum number of outstanding asynchronous I/O operations that are supported. (We talk about asynchronous I/O in Chapter 13). The returned value is AIO_MAX.
_SC_AIO_PRIO_DELTA_MAX	This indicates the maximum amount by which a process can decrease its asynchronous I/O priority level from its own scheduling priority. The returned value is AIO_PRIO_DELTA_MAX.

_SC_ARG_MAX	This is the maximum size of argv [] plus argp []. The returned value is ARG_MAX.
_SC_ASYNCHRONOUS_IO	This is a Boolean value that indicates whether asynchronous I/O is supported. The returned value is _POSIX_ASYNCHRONOUS_IO.
_SC_ATEXIT_MAX	This is the maximum number of functions that can be registered with atexit. These functions are called when a process terminates. The returned value is ATEXIT_MAX.
_SC_AVPHYS_PAGES	This is the number of physical memory pages not currently in use by the system.
_SC_BC_BASE_MAX	This is the maximum ibase and obase values allowed by the bc command. The returned value is BC_BASE_MAX.
_SC_BC_DIM_MAX	This is the maximum number of elements permitted in an array by the bc command. The returned value is BC_DIM_MAX.
_SC_BC_SCALE_MAX	This is the maximum scale value allowed by the bc command. The returned value is BC_SCALE_MAX.
_SC_BC_STRING_MAX	This is the maximum length of a string constant allowed by the bc command. The returned value is BC_STRING_MAX.
_SC_CHILD_MAX	This is the maximum number of processes allowed to each user ID. The value returned is CHILD_MAX.
_SC_CLK_TCK	This is the number of clock ticks per second. The returned value is CLK_TCK.
_SC_COLL_WEIGHTS_MAX	This is the maximum number of weights that can be assigned to an entry of the LC_COLLATE order keyword in a locale definition file. The returned value is COLL_WEIGHTS_MAX.
_SC_CPUID_MAX	This is the maximum possible processor ID.
_SC_DELAYTIMER_MAX	This is the maximum number of timer expiration overruns. DELAYTIMER_MAX is the returned value.
_SC_EXPR_NEST_MAX	This is the maximum number of parentheses allowed by the expr command. The returned value is EXPR_NEST_MAX.
_SC_FSYNC	This is a Boolean value that indicates whether file synchronization is supported. The returned value is _POSIX_FSYNC.
_SC_GETGR_R_SIZE_MAX	This is the maximum size of the group entry in /etc/nsswitch.conf, the name service switch file. NSS_BUFLEN_GROUP is the returned value.

_SC_GETPW_R_SIZE_MAX	This is the maximum size of the password entry in /etc/nsswitch.conf, the name service switch file. NSS_BUFLEN_PASSWD is the returned value.
_SC_IOV_MAX	This is the maximum number of iovec structures available to each process for use with readv and writev. The returned value is IOV_MAX.
_SC_JOB_CONTROL	This is a Boolean value that indicates whether job control is supported. The returned value is _POSIX_JOB_CONTROL.
_SC_LINE_MAX	This is maximum length of an input line. The returned value is LINE_MAX.
_SC_LOGIN_NAME_MAX	This is the maximum length of a login name, including the terminating NUL. The value returned is LOGNAME_MAX + 1.
_SC_LOGNAME_MAX	This is the maximum length of a login name, excluding the terminating NUL. The value returned is LOGNAME_MAX.
_SC_MAPPED_FILES	This is a Boolean value that indicates whether memory-mapped files are supported. The returned value is _POSIX_MAPPED_FILES.
_SC_MAXPID	This is the maximum value of a process ID.
_SC_MEMLOCK	This is a Boolean value that indicates whether memory locking is supported. The returned value is _POSIX_MEMLOCK.
_SC_MEMLOCK_RANGE	This is a Boolean value that indicates whether range memory locking is supported. The returned value is _POSIX_MEMLOCK_RANGE.
_SC_MEMORY_PROTECTION	This is a Boolean value that indicates whether memory protection is supported. The returned value is _POSIX_MEMORY_PROTECTION.
_SC_MESSAGE_PASSING	This is a Boolean value that indicates whether message passing is supported. The returned value is _POSIX_MESSAGE_PASSING.
_SC_MQ_OPEN_MAX	This is the maximum number of open messages a process can hold. The returned value is MQ_OPEN_MAX.
_SC_MQ_PRIO_MAX	This is the maximum number of message priorities supported. The returned value is MQ_PRIO_MAX.
_SC_NGROUPS_MAX	The is the maximum number of groups to which a user can belong. The returned value is NGROUPS_MAX.

_SC_NPROCESSORS_CONF	This is the number of processors currently configured (i.e., actually installed in the system).
_SC_NPROCESSORS_MAX	This is the maximum number of processors supported by the current platform.
_SC_NPROCESSORS_ONLN	This is the number of processors currently online.
_SC_OPEN_MAX	This is the maximum number of open files per process. The returned value is OPEN_MAX.
_SC_PAGESIZE	This is the system memory page size. The returned value is PAGESIZE.
_SC_PAGE_SIZE	This is a synonym for _SC_PAGESIZE. The returned value is PAGESIZE.
_SC_PASS_MAX	This is the maximum number of significant bytes in a password. It does not take into account the extended password lengths available from Solaris 9 12/02 and later, if different encryption algorithms are used. The returned value is PASS_MAX.
_SC_PHYS_PAGES	This is the number of pages of physical memory in the system.
_SC_PRIORITIZED_IO	This is a Boolean value that indicates whether POSIX prioritized I/O is supported. The returned value is _POSIX_PRIORITIZED_IO.
_SC_PRIORITY_SCHEDULING	This is a Boolean value that indicates whether process scheduling is supported. The returned value is _POSIX_PRIORITY_SCHEDULING.
_SC_RE_DUP_MAX	This is the maximum number of repeated occurrences of a regular expression permitted when using interval notation {m,n}. The returned value is RE_DUP_MAX.
_SC_REALTIME_SIGNALS	This is a Boolean value that indicates whether realtime signals are supported. The returned value is _POSIX_REALTIME_SIGNALS.
_SC_RTSIG_MAX	This is the maximum number of realtime signals reserved for application use. The returned value is RTSIG_MAX.
_SC_SAVED_IDS	This is a Boolean value that indicates whether saved set-user-IDs are supported. The returned value is _POSIX_SAVED_IDS.
_SC_SEM_NSEMS_MAX	This is the maximum number of POSIX semaphores available to each process. The returned value is SEM_NSEMS_MAX.
_SC_SEM_VALUE_MAX	This is the maximum value a POSIX semaphore can have. SEM_VALUE_MAX is the returned value.

_SC_SEMAPHORES	This is a Boolean value that indicates whether POSIX semaphores are supported. The returned value is _POSIX_SEMAPHORES.
_SC_SHARED_MEMORY_OBJECTS	This is a Boolean value that indicates whether POSIX shared memory objects are supported. _POSIX_SHARED_MEMORY_OBJECTS is the returned value.
_SC_SIGQUEUE_MAX	This is the maximum number of queued signals that a process can send and have pending at receivers at a time. The returned value is SIGQUEUE_MAX.
_SC_STACK_PROT	This is the default stack protection.
_SC_STREAM_MAX	This is the maximum number of streams a process can have open at a time. The returned value is STREAM_MAX.
_SC_SYNCHRONIZED_IO	This is a Boolean value that indicates whether POSIX synchronized I/O is supported. The returned value is _POSIX_SYNCHRONIZED_IO.
_SC_THREAD_ATTR_STACKADDR	This is a Boolean value that indicates whether the POSIX thread stack address attribute option is supported. The returned value is _POSIX_THREAD_ATTR_STACKADDR.
_SC_THREAD_ATTR_STACKSIZE	This is a Boolean value that indicates whether the POSIX thread stack size attribute option is supported. The returned value is _POSIX_THREAD_ATTR_STACKSIZE.
_SC_THREAD_DESTRUCTOR_ITERATIONS	This is the number of attempts to destroy thread-specific data when a thread exits. PTHREAD_DESTRUCTOR_ITERATIONS is the returned value.
_SC_THREAD_KEYS_MAX	This is the maximum number of data keys per process. PTHREAD_KEYS_MAX is the returned value.
_SC_THREAD_PRIO_INHERIT	This is a Boolean value that indicates whether the POSIX threads priority inheritance option is supported. _POSIX_THREAD_PRIO_INHERIT is the returned value.
_SC_THREAD_PRIO_PROTECT	This is a Boolean value that indicates whether the POSIX threads priority protection option is supported. _POSIX_THREAD_PRIO_PROTECT is the returned value.
_SC_THREAD_PRIORITY_SCHEDULING	This is a Boolean value that indicates whether the POSIX threads execution scheduling option is supported. The returned value is _POSIX_THREAD_PRIORITY_SCHEDULING.

_SC_THREAD_PROCESS_SHARED	This is a Boolean value that indicates whether the POSIX threads process shared synchronization option is supported. _POSIX_THREAD_PROCESS_SHARED is the returned value.
_SC_THREAD_SAFE_FUNCTIONS	This is a Boolean value that indicates whether the POSIX threads thread safe functions option is supported. The returned value is _POSIX_THREAD_SAFE_FUNCTIONS.
_SC_THREAD_STACK_MIN	This is the minimum number of bytes of thread stack storage. PTHREAD_STACK_MIN is the returned value.
_SC_THREAD_THREADS_MAX	This is the maximum number of threads per process. PTHREAD_THREADS_MAX is the value returned.
_SC_THREADS	This is a Boolean value that indicates whether POSIX threads are supported. The returned value is _POSIX_THREADS.
_SC_TIMER_MAX	This is the maximum number of timers per process. The returned value is TIMER_MAX.
_SC_TIMERS	This is a Boolean value that indicates whether POSIX timers are supported. The returned value is _POSIX_TIMERS.
_SC_TTY_NAME_MAX	This is the maximum length of a TTY device name. The returned value is TTYNAME_MAX.
_SC_TZNAME_MAX	This is the maximum number of bytes supported for the name of time zones. The returned value is TZNAME_MAX.
_SC_VERSION	This indicates which version of POSIX.1 is supported. _POSIX_VERSION is the returned value.
_SC_XBS5_ILP32_OFF32	This is a Boolean value that indicates whether the X/Open ILP32 with 32-bit file offsets build environment is supported. The returned value is _XBS_ILP32_OFF32.
_SC_XBS5_ILP32_OFFBIG	This is a Boolean value that indicates whether the X/Open ILP32 with 64-bit file offsets build environment is supported. The returned value is _XBS5_ILP32_OFFBIG.
_SC_XBS5_LP64_OFF64	This is a Boolean value that indicates whether the X/Open LP64 with 64-bit file offsets build environment is supported. The returned value is _XBS5_LP64_OFF64.
_SC_XBS5_LPBIG_OFFBIG	This is a synonym for _SC_XBS5_LP64_OFF64. The returned value is _XBS5_LP64_OFF64.

_SC_XOPEN_CRYPT	This is a Boolean value that indicates whether the X/Open encryption feature group is supported. _XOPEN_CRYPT is the returned value.
_SC_XOPEN_ENH_I18N	This is a Boolean value that indicates whether the X/Open enhanced internationalization feature group is supported. The returned value is _XOPEN_ENH_I18N.
_SC_XOPEN_LEGACY	This is a Boolean value that indicates whether the X/Open legacy feature group is supported. _XOPEN_LEGACY is the returned value.
_SC_XOPEN_REALTIME	This is a Boolean value that indicates whether the X/Open POSIX realtime feature group is supported. _XOPEN_REALTIME. is the returned value.
_SC_XOPEN_REALTIME_THREADS	This is a Boolean value that indicates whether the X/Open POSIX realtime threads feature group is supported. The returned value is _XOPEN_REALTIME_THREADS.
_SC_XOPEN_SHM	This is a Boolean value that indicates whether the X/Open shared memory feature group is supported. The returned value is _XOPEN_SHM.
_SC_XOPEN_UNIX	This is a Boolean value that indicates whether the X/Open CAE Specification, August 1994, System Interfaces and Headers, Issue 4, Version 2 is supported. _XOPEN_UNIX is the returned value.
_SC_XOPEN_VERSION	This indicates which version of the X/Open Portability Guide the implementation conforms to. The returned value is _XOPEN_VERSION.
_SC_XOPEN_XCU_VERSION	This indicates which version of the XCU specification the implementation conforms to. The returned value is _XOPEN_XCU_VERSION.

Some of the preceding values for *name* return –1 without setting errno. This is because no maximum limit can be determined. Although the system supports at least the minimum values, higher values can be supported depending on system resources. Figure 8.2 summarizes the variables this applies to.

We must be aware of a number points when we use the sysconf function:

1. The value of CLK_TCK can be variable, so we should not assume that it is a compile time constant.

Variable	Minimum supported value
_SC_AIO_MAX	_POSIX_AIO_MAX
_SC_ATEXIT_MAX	32
_SC_THREAD_THREADS_MAX	_POSIX_THREAD_THREADS_MAX
_SC_THREAD_KEYS_MAX	_POSIX_THREAD_KEYS_MAX
_SC_THREAD_DESTRUCTOR_ITERATIONS	_POSIX_THREAD_DESTRUCTOR_ITERATIONS

Figure 8.2 Variables that return –1 without setting errno.

2. A call to setrlimit (see Section 8.4) can cause the value of OPEN_MAX to change.
3. Determining the amount of physical memory (in bytes) in the system by multiplying sysconf (_SC_PHYS_PAGES) by sysconf (_SC_PAGESIZE) can return a value that exceeds the maximum value representable by a long or an unsigned long in a 32-bit process. The same applies to determining the amount of free physical memory by multiplying sysconf (_SC_PAGESIZE) by sysconf (_SC_AVPHYS_PAGES).

   There are two workarounds for this problem. The first is to compile our program as a 64-bit process (so that a long is large enough to hold the result of the multiplication), but the disadvantage to this is that the resulting program will run only on a 64-bit kernel. The second workaround is to store the result of the multiplication in a 64-bit quantity, like a longlong_t. This has the advantage of working correctly on both 32-bit and 64-bit kernels.

The variables _SC_CPUID_MAX and _SC_NPROCESSORS_MAX were added in Solaris 9.

Example: Printing CPU and memory information

Program 8.2 shows how we can use the sysconf function to determine the total amount of memory installed in a machine and how much is free. It also shows how many CPUs are installed and how many are online.

Running Program 8.2 on a dual-CPU Ultra 60 gives the following results:

```
$ ./sysconf
2 CPUs installed, 2 online
2048 MB physical memory, 1530 MB free
$ su
Password:
# psradm -f 2                              Take CPU 2 offline
# ./sysconf
2 CPUs installed, 1 online
2048 MB physical memory, 1530 MB free
```

sys_info/sysconf.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>

 3 #define ONE_MB (1024 * 1024)

 4 int main (void)
 5 {
 6     long num_procs;
 7     long procs_online;
 8     long page_size;
 9     long num_pages;
10     long free_pages;
11     longlong_t mem;
12     longlong_t free_mem;

13     num_procs = sysconf (_SC_NPROCESSORS_CONF);
14     procs_online = sysconf (_SC_NPROCESSORS_ONLN);
15     page_size = sysconf (_SC_PAGESIZE);
16     num_pages = sysconf (_SC_PHYS_PAGES);
17     free_pages = sysconf (_SC_AVPHYS_PAGES);

18     mem = (longlong_t) ((longlong_t) num_pages * (longlong_t) page_size);
19     mem /= ONE_MB;
20     free_mem = (longlong_t) free_pages * (longlong_t) page_size;
21     free_mem /= ONE_MB;

22     printf ("%ld CPU%s installed, %ld online\n", num_procs,
23         (num_procs > 1) ? "s" : "", procs_online);
24     printf ("%lld MB physical memory, %lld MB free\n", mem, free_mem);

25     return (0);
26 }
```

sys_info/sysconf.c

Program 8.2 Displaying CPU and memory information.

The `pathconf` and `fpathconf` Functions

As we stated previously, the `pathconf` and `fpathconf` functions are used to determine the value of run time limits associated with a file or directory.

```
#include <unistd.h>

long pathconf (const char *path, int name);

long fpathconf (int fildes, int name);
```

Both return: see text

The `pathconf` function returns the current value of a configurable limit or option (variable) associated with a file or directory identified by *path*.

The `fpathconf` function returns the same information, but for the file associated with the file descriptor *fildes*.

For both functions, the *name* argument represents the name of the variable we are interested in. In the event of an error, −1 is retuned and `errno` is set appropriately. We should be aware that −1 is also a legal return value when no error occurs, so programs wanting to check for errors should set `errno` to 0 before calling `pathconf` or `fpathconf`, and then check it if −1 is returned.

The parameter *name* specifies the name of the limit we are interested in, and must be one of the following values (the values returned are integers unless otherwise stated):

`_PC_FILESIZEBITS` This is the maximum number of bits that can be used to store the size of a file. If *path* or *fildes* refer to a directory, the returned value applies to files within that directory. The actual value returned depends on the underlying file system type. A UFS file system that does not support large files will always return 32. UFS file systems that support large files will return 41, and NFS v3 file systems return whatever is in the `maxfilesize` member of a successful `FSINFO` request; `FSINFO` is a procedure that retrieves nonvolatile file system information and general information about the NFS v3 protocol server implementation.

`_PC_LINK_MAX` This is the maximum number of links the file or directory may have. If *path* or *fildes* refers to a directory, the returned value applies to the directory itself. The value returned is `LINK_MAX`.

`_PC_MAX_CANON` This is the maximum number of characters in a line from a terminal in canonical mode (we talk about terminal modes in Chapter 12). If *path* or *fildes* does not refer to a terminal device, the returned value is meaningless; otherwise, the returned value is `MAX_CANON`.

`_PC_MAX_INPUT` This is the maximum number of characters in a terminal input queue. If *path* or *fildes* does not refer to a terminal device, the returned value is meaningless; otherwise, the returned value is `MAX_INPUT`.

`_PC_NAME_MAX` This is the maximum number of characters in a filename. If *path* or *fildes* refers to a directory, the returned value applies to files within that directory.

`_PC_PATH_MAX` This is the maximum number of characters in a pathname. If *path* or *fildes* refers to a directory, the returned value is the maximum length of a relative pathname when the specified directory is the working directory.

`_PC_PIPE_BUF` This is the maximum number of bytes we can write atomically to a pipe or FIFO. If *path* or *fildes* refers to a

	pipe or FIFO, the returned value applies to that pipe or FIFO. If *path* or *fildes* refers to a directory, then the returned value applies to any FIFO that exists or can be created in that directory. In all of the preceding cases, the returned value is `PIPE_BUF`.
`_PC_XATTR_ENABLED`	This is a Boolean value that indicates whether the file system containing *path* or *fildes* supports extended file attributes. We talk about extended file attributes in Section 13.38.
`_PC_XATTR_EXISTS`	This is a Boolean value that indicates whether *path* or *fildes* has one or more extended file attributes. We talk about extended file attributes in Section 13.38.
`_PC_CHOWN_RESTRICTED`	This is a Boolean value that indicates whether unprivileged users may use the `chown` function to change the ownership of files they currently own. If *path* or *fildes* refers to a directory, the returned value applies to any files, other than directories, that exist or can be created within that directory. The returned value is `_POSIX_CHOWN_RESTRICTED`.
`_PC_NO_TRUNC`	This is a Boolean value that indicates whether pathnames whose components are longer than `_PC_NAME_MAX` characters will generate an error. If *path* or *fildes* refers to a directory, the returned value applies to filenames within that directory and is `_POSIX_NO_TRUNC`.
`_PC_VDISABLE`	This is a Boolean value that indicates whether special terminal input characters can be disabled (see Chapter 12). If *path* or *fildes* does not refer to a terminal device, the returned value is meaningless; otherwise, the returned value is `_POSIX_VDISABLE`.
`_PC_ASYNC_IO`	This is a Boolean value that indicates whether asynchronous I/O may be performed on the file. The returned value is `_POSIX_ASYNC_IO`.
`_PC_PRIO_IO`	This is a Boolean value that indicates whether prioritized I/O may be performed on the file. If *path* or *fildes* refers to a directory, the returned value is meaningless, otherwise, the returned value is `_POSIX_PRIO_IO`.
`_PC_SYNC_IO`	This is a Boolean value that indicates whether synchronous I/O may be performed on the file. The returned value is `_POSIX_SYNC_IO`.

The variables `_PC_XATTR_ENABLED` and `_PC_XATTR_EXISTS` were added in Solaris 9.

Example: Printing some run time file limits

Program 8.3 shows how we can use the `pathconf` function to determine some run time file limits.

sys_info/pathconf.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include "ssp.h"

 4 int main (int argc, char **argv)
 5 {
 6     long filesize_bits;
 7     long max_links;

 8     if (argc != 2)
 9         err_quit ("Usage: pathconf path");

10     filesize_bits = pathconf (argv [1], _PC_FILESIZEBITS);
11     max_links = pathconf (argv [1], _PC_LINK_MAX);

12     printf ("Maximum file size bits = %ld\n", filesize_bits);
13     printf ("Maximum links to a file = %ld\n", max_links);

14     return (0);
15 }
```

sys_info/pathconf.c

Program 8.3 Displaying some run time file limits.

Let's see what happens when we run this simple program against a few directories.

```
$ ./pathconf /
Maximum file size bits = 41
Maximum links to a file = 32767
$ ./pathconf /home
Maximum file size bits = -1
Maximum links to a file = 32767
$ ./pathconf /home/rich
Maximum file size bits = 40
Maximum links to a file = 32767
```

The program output shows that the root directory is on a UFS file system that supports large files, `/home` has essentially no file size limit (although it would be reasonable to assume that file offsets are limited to 64 bits in practice), and that `/home/rich` is an NFS-mounted directory that limits the size of files to 40 bits. All of the directories tested support up to 32,767 links to a single file. This also means that directories are limited to containing 32,765 subdirectories; two links are reserved for the directory itself and its parent. (There are no limits on how many files a directory may contain, except for the number of directory entries that will fit into a file, and the availability of inodes.)

The `getpagesize` Function

Another way we can get the page size is to use `getpagesize`.

```
#include <unistd.h>

int getpagesize (void);
```

Returns: the number of bytes in a page

The `getpagesize` function returns the number of bytes in a page. This page size is the system page size, which is not necessarily the same size as the underlying hardware.

The page size is the granularity of many memory management functions. However, the value returned by `getpagesize` is not the smallest value that can be allocated by `malloc`.

Calling `getpagesize` is equivalent to calling the `sysconf` function with an argument of `_SC_PAGESIZE` or `_SC_PAGE_SIZE`.

The `getpagesizes` Function

Solaris 9 introduced the feature of multiple user page sizes. We can get a list of the supported user page sizes by using `getpagesizes`.

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

int getpagesizes (size_t pagesize [], int nelem);
```

Returns: the number of page sizes supported or retrieved if OK, –1 on error

We can use the `getpagesizes` function in two ways: we can either determine how many different user page sizes are supported, or we can get a list of the supported user page sizes.

If we call `getpagesizes` with a NULL pointer for *pagesize* and *nelem* set to 0, the number of supported user page sizes is returned. Otherwise, up to *nelem* page sizes are retrieved and placed into successive elements of *pagesize*.

Not all processors support all page sizes or combinations of page sizes with equal efficiency, so we must take this into consideration when using `getpagesizes`.

Example: Listing the supported user page sizes

Program 8.4 lists all the supported user page sizes.

Get the number of user page sizes

11–12 Call `getpagsizes` with *pagesize* and *nelem* set to NULL and 0 respectively to determine the number of supported user page sizes.

Allocate sufficient memory to hold the list

13–14 Call `malloc` to allocate a memory buffer big enough to hold the list of supported user page sizes.

Print the list of supported user page sizes

15–16 Call `getpagesizes` again to retrieve the list of supported user page sizes.

sys_info/getpagesizes.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/mman.h>
 4 #include "ssp.h"

 5 int main (void)
 6 {
 7     int num_page_sizes;
 8     int res;
 9     int i;
10     size_t *page_sizes;

11     if ((num_page_sizes = getpagesizes (NULL, 0)) == -1)
12         err_msg ("getpagesizes (NULL, 0) failed");

13     if ((page_sizes = malloc (sizeof (size_t) * num_page_sizes)) == NULL)
14         err_msg ("malloc failed");

15     if ((res = getpagesizes (page_sizes, num_page_sizes)) == -1)
16         err_msg ("getpagesizes failed");

17     printf ("Supported page sizes:");
18     for (i = 0; i < res; i++)
19         printf (" %d", page_sizes [i]);
20     printf ("\n");

21     free (page_sizes);

22     return (0);
23 }
```

sys_info/getpagesizes.c

Program 8.4 Listing the supported user page sizes.

17–20 Print the list we just retrieved.

Running Program 8.4 on one of the author's machines (an Ultra 60) gives the following results:

```
$ ./getpagesizes
Supported page sizes: 8192 65536 524288 4194304
```

Running the command `pagesize -a` confirms these results.

8.4 Per-Process Resource Limits

Beyond the systemwide resource limits we discussed in the previous section, a number of limits are applied on a per-process basis, like the maximum amount of CPU time, the number of file descriptors, and the size of the stack. Many of these limits can be changed by the process, and are intended to prevent a runaway process from consuming all of the machine's resources.

There are two types of limit for each resource: a soft (current) limit, and a hard (maximum) limit. Any process can change soft limits to any value less than or equal to

the hard limit, but only privileged process can raise the hard limit. Any process can (permanently) lower its hard limits.

The `ulimit` Function

The `ulimit` function can be used to get and set certain process limits.

```
#include <ulimit.h>

long ulimit (int cmd, /* long newlimit */...);
```

Returns: the value of the requested limit if OK, –1 on error

The `ulimit` function allows a process to get or set some of its limits. The operation performed is determined by the value of the *cmd* argument, which must be one of the following four values:

`UL_GETFSIZE`	This returns the maximum file size in units of 512-byte blocks that the process can create. The number returned is the integer part of the soft limit divided by 512. Note that this limit only affects writing to a file; files of any size can be read, with the exception of a non-large-file-aware program trying to read a large file (see Section 1.12).
`UL_SETFSIZE`	This sets the hard and soft limits for output operations to the value specified by *newlimit*. The file size limits are set to the value of *newlimit* multiplied by 512. The new file size limit is returned.
`UL_GMEMLIM`	This returns the maximum amount of memory the process may use.
`UL_GDESLIM`	This returns the maximum number of file descriptors the process may open.

On successful completion, the requested limit is returned. Otherwise, –1 is returned and the limit is not changed.

The `getrlimit` and `setrlimit` Functions

The `getrlimit` and `setrlimit` functions provide a more general interface for controlling process limits.

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

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

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

Both return: 0 if OK, –1 on error

The getrlimit function gets the hard and soft limits associated with the resource identified by *resource* and places them in the rlimit structure pointed to by *rlp*. The rlimit structure has the following members:

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

As well as specifying a specific limit, an "infinite" limit may be applied by setting the appropriate member of the rlimit structure to the special value of RLIM_INFINITY.

The legal values of *resource* are:

RLIMIT_AS — This is a synonym for RLIMIT_VMEM.

RLIMIT_CORE — This is the maximum size of a core file in bytes that may be created. Setting this limit to 0 will prevent the creation of a core file. The writing of the core file terminates when it reaches this size, even if it is incomplete.

RLIMIT_CPU — This is the maximum amount of CPU time (measured in seconds) the process is allowed to use. This a soft limit only; there is no hard limit. When the time limit is exceeded, the process is sent a SIGXCPU signal (see Chapter 17). If SIGXCPU is being held or ignored, the behaviour is scheduling-class defined.

RLIMIT_DATA — This is the maximum size of the process' heap in bytes. The heap is the data segment of a process.

RLIMIT_FSIZE — This is the maximum size of a file in bytes that the process may create. Setting this limit to 0 will prevent the creation of files. When this limit is exceeded, the process is sent a SIGXFSZ signal. If this signal is being held or ignored, continued attempts to increase the size of the file will fail with errno set to EFBIG.

RLIMIT_NOFILE — This is the maximum value that the kernel may assign to a file descriptor, effectively limiting the number of file descriptors (and hence, the number of open files) for the calling process to this number.

RLIMIT_STACK — This is the maximum size of the process' stack in bytes. The kernel will not automatically grow the stack beyond this limit. When this limit is exceeded, a SIGSEGV signal is sent to the process. If this signal is being held or ignored, or is being caught without arrangements being made to us an alternate stack, the signal's disposition is set to SIG_DFL before it is sent to the process. (We talk about signals in Chapter 17.)

Increasing this limit will increase the size of the stack, but current memory segments will not be moved to allow this

growth. The only way we can guarantee that the process' stack will be able to grow to the new limit is to increase the limit before we execute the process in which the new stack is to be used.

In a multithreaded process, setting this limit has no effect if the calling thread is not the main one. Calling `setrlimit` for `RLIMIT_STACK` only impacts the main thread's stack, and should be made only from the main thread, if at all.

`RLIMIT_VMEM` This is the maximum size of the process' mapped address space in bytes. When this limit is exceeded, the `malloc` and other memory allocation functions will fail, as will calls to `mmap`. Additionally, the automatic stack growth will fail, with the effects described under `RLIMIT_STACK`.

As we stated previously, setting a limit to `RLIM_INFINITY` means that there is "no limit" on the resource. Requesting a limit of `RLIM_SAVED_MAX` will set the limit to the corresponding saved hard limit. Similarly, requesting a limit of `RLIM_SAVED_CUR` will set the limit to the corresponding saved soft limit.

The current value of two limits affect their corresponding implementation-defined constants. These are summarized in Figure 8.3.

Limit	Constant
RLIMIT_FSIZE	FCHR_MAX
RLIMIT_NOFILE	OPEN_MAX

Figure 8.3 The effect of limits on implementation-defined constants.

Example: Printing the current resource limits

Program 8.5 prints out the current soft and hard resource limits.

Note how we've used the ISO C string creation operator again, as we did in Program 8.1.

Running Program 8.5 gives us the following results:

```
$ ./getrlimit
RLIMIT_AS      unlimited  unlimited
RLIMIT_CORE    unlimited  unlimited
RLIMIT_CPU     unlimited  unlimited
RLIMIT_DATA    unlimited  unlimited
RLIMIT_FSIZE   unlimited  unlimited
RLIMIT_NOFILE        256      65536
RLIMIT_STACK     8388608  unlimited
RLIMIT_VMEM    unlimited  unlimited
```

sys_info/getrlimit.c

```
 1 #include <stdio.h>
 2 #include <sys/resource.h>
 3 #include "ssp.h"

 4 #define plimit(name) print_limits (#name, name)

 5 static void print_limits (const char *name, int resource);

 6 int main (void)
 7 {
 8     plimit (RLIMIT_AS);
 9     plimit (RLIMIT_CORE);
10     plimit (RLIMIT_CPU);
11     plimit (RLIMIT_DATA);
12     plimit (RLIMIT_FSIZE);
13     plimit (RLIMIT_NOFILE);
14     plimit (RLIMIT_STACK);
15     plimit (RLIMIT_VMEM);

16     return (0);
17 }

18 static void print_limits (const char *name, int resource)
19 {
20     struct rlimit limits;

21     if (getrlimit (resource, &limits) == -1)
22         err_quit ("getrlimit (%s) failed", name);

23     printf ("%-13s  ", name);

24     if (limits.rlim_cur == RLIM_INFINITY)
25         printf ("unlimited  ");
26     else
27         printf ("%9ld  ", limits.rlim_cur);

28     if (limits.rlim_max == RLIM_INFINITY)
29         printf ("unlimited\n");
30     else
31         printf ("%9ld\n", limits.rlim_max);
32 }
```

sys_info/getrlimit.c

Program 8.5 Print the current soft and hard resource limits.

The `getdtablesize` Function

Another way of determining the maximum number of file descriptors available to a process is by using the `getdtablesize` function.

```
#include <unistd.h>

int getdtablesize (void);
```

Returns: the soft file descriptor limit

The `getdtablesize` function returns the current soft limit of the maximum file descriptor number as if obtained by calling `getrlimit` with *resource* set to `RLIMIT_NOFILE`.

8.5 The Resource Control Facility

Solaris 9 introduced the resource control facility, which provides more general purpose resource control functionality than the `getrlimit` (rlimit) mechanism. (We should note that the resource control facility is supplied in addition to rlimits, not instead of. In fact, the latter is built on top of the former.)

One of the limitations of the rlimit method of resource control is that the granularity of the limits is restricted to processes only. The resource controls facility extends the concept of limits to the task and project entities described in Chapter 6 of [Sun Microsystems 2002a]. Another advantage of the resource control facility is that it can provide information about encountered restraints without necessarily denying access to the requested resource. This information can be used to help the capacity planning process.

Resource Controls

The resource control attributes are set in the final field of the `project` database entry (in a files-based environment, the project database is stored in `/etc/project`). The values associated with each resource control are enclosed in parentheses and appear as plain text separated by commas. The value in parentheses is called an *action clause*, and is made up of a privilege level, a threshold value, and an action associated with the given threshold value. Each resource control can have multiple action clauses, which are also separated by commas. The following example resource control defines a per-process address space limit and a per-task LWP limit on a project entry (note that we had to wrap the line so it fits on the page; each entry must be on one and only one line):

```
dev:100::::task.max-lwps=(privileged,10,deny);
    process.max-address-space=(privileged,209715200,deny)
```

Figure 8.4 lists the standard resource controls available in Solaris 9 (later versions added numerous others).

The resource control threshold values constitute an enforcement point where local or global actions can occur. Each threshold value must be associated with one of the following privilege levels:

- Basic, which can be modified by the owner of the calling process

Control name	Description	Default unit
`project.cpu-shares`	The number of CPU shares granted to this project for use with the fair-share scheduler	Quantity (shares)
`task.max-cpu-time`	Maximum CPU time available to this task's processes	Time (milliseconds)
`task.max-lwps`	Maximum number of LWPs simultaneously available to this task's processes	Quantity (LWPs)
`process.max-cpu-time`	Maximum CPU time available to this process	Time (milliseconds)
`process.max-file-descriptor`	Maximum file descriptor index available to this process	Index
`process.max-file-size`	Maximum file offset available for writing by this process	Size (bytes)
`process.max-core-size`	Maximum size of a core file that may be created by this process	Size (bytes)
`process.max-data-size`	Maximum size of heap available to this process	Size (bytes)
`process.max-stack-size`	Maximum stack memory segment available to this process	Size (bytes)
`process.max-address-space`	Maximum address space, as summed over segment size, available to this process	Size (bytes)

Figure 8.4 Resource controls available in Solaris 9.

- Privileged, which can be modified only by privileged callers, i.e., `root`
- System, which is fixed for the duration of the OS instance

Each resource control is guaranteed to have at least a system value associated with it. This value represents how much of the resource the current instance of the OS is capable of providing. Any number of privileged values can be defined, but only one basic value is allowed.

The resource value is stored as an unsigned 64-bit quantity. When the resource threshold of a resource is exceeded, one or more actions can be invoked:

1. Requests for an amount of the resource that will exceed the threshold can be denied.
2. A signal can be sent to the violating or observing process if the threshold value is reached.

Resource controls also have associated with them a set of global and local flags, which we discuss below.

From the perspective of a program, resource control blocks are used to implement the resource control facility. Each of these blocks is stored in an opaque data structure called an `rctlblk_t`. (When we say a data structure is opaque, we mean that we have no idea what members it has; the only way we can manipulate the members is to use the functions provided for that purpose.)

The `rctlblk_size` Function

Because a resource control block is implemented as an opaque data structure, we can't simply use something like

```
struct rctlblk rblk;
```

to allocate the correct amount of memory for it. Instead, we must use `malloc`; the number of bytes we need to allocate is given by the `rctlblk_size` function.

```
#include <rctl.h>

size_t rctlblk_size (void);
```

Returns: the size of a resource control block

The `rctlblk_size` function returns the size of a resource control block for use in memory allocation. Using `malloc` (or a related function) to allocate the number of bytes returned by `rctlblk_size` is the only safe way to allocate memory for a resource control block.

The `getrctl` Function

We use the `getrctl` function to get resource values.

```
#include <rctl.h>

int getrctl (const char *controlname, rctlblk_t *old_blk, rctlblk_t *new_blk,
    uint_t flags);
```

Returns: 0 if OK, −1 on error

The `getrctl` function enables us to retrieve resource control values on active entities on the system, such as processes, tasks, and projects. As we stated earlier, each resource control is an unsigned 64-bit quantity, although a number of flags modify the resource control that is retrieved.

The name of the resource control we are interested in must be stored in the buffer pointed to by *controlname*, and the retrieved resource control is stored in the `rctlblk` structure pointed to by *new_blk*.

The values of *old_blk* and *flags* determine which resource control associated with *controlname* is retrieved (recall that each resource control may have several values associated with it). Setting *old_blk* to NULL and *flags* to `RCTL_FIRST` will retrieve the first value associated with *controlname*. Subsequent resource control values can be retrieved by setting *old_blk* to the previously retrieved value and setting *flags* to `RCTL_NEXT`. When the list of values is exhausted, `getrctl` fails, setting `errno` to `ENOENT`.

The current usage of a controlled resource can be retrieved by setting *flags* to `RCTL_USAGE`.

Once we have retrieved the resource control we are interested in, we can get or modify the values associated with it using the `rctlblk_get` and `rctlblk_set` family of functions discussed next.

The `setrctl` Function

We use the `setrctl` function to set resource values.

```
#include <rctl.h>

int setrctl (const char *controlname, rctlblk_t *old_blk, rctlblk_t *new_blk,
    uint_t flags);
```

Returns: 0 if OK, –1 on error

The `setrctl` function enables us to modify resource control values. In other words, it allows us to create, modify, or delete the action-value pairs of a given resource control.

The name of the resource control we want to operate on is stored in the buffer pointed to by *controlname*. The operation performed on the resource control is determined by the value of *flags*, *old_blk*, and *new_blk*.

If *flags* is set to `RCTL_INSERT`, the resource control block pointed to by *new_blk* is inserted into the sequence. Conversely, if *flags* is set to `RCTL_DELETE`, the resource control block pointed to by *new_blk* is removed from the sequence. Setting *flags* to `RCTL_REPLACE` will cause the resource control matching the one pointed to by *old_blk* to be replaced by the one pointed to by *new_blk*.

We must bear in mind several points when manipulating resource control blocks. Resource control blocks are matched on the privilege as well as the value fields. Resource control operations are performed on the first matching resource control block; multiple blocks of equal privilege and value will likely need to be deleted and reinserted, rather than replaced, to have the desired outcome. The resource control blocks are sorted such that all blocks with the same value that do not have the `RCTL_LOCAL_DENY` flag set precede those that do.

Because the resource control facility is used by both `[gs]etrlimit` and `[gs]etrctl`, the ordering issues we have just discussed and the limit equivalencies we discuss in the next paragraph must be considered if we use both interfaces in the same program (these issues are of no concern if either interface is used exclusively).

As we stated previously, the resource control facility is used to implement the hard and soft process limits made available with `getrlimit` and `setrlimit`. An rlimit has two (and only two) values associated with it, but the `RCTL_INSERT` and `RCTL_DELETE` operations allow a resource control to have an arbitrary number of values associated with it. In the event of the number of values associated with a resource control not being equal to two, the lowest priority resource control value with the `RCTL_LOCAL_DENY` flag set is taken as the soft limit, and the lowest priority resource control value with a priority equal to or exceeding `RCPRIV_PRIVILEGED` with the `RCTL_LOCAL_DENY` flag set is taken as the hard limit. If no identifiable soft limit exists on the resource control, and `setrlimit` is called, a new resource control value will be created. If a resource control does not have the global `RCTL_GLOBAL_LOWERABLE` property set, its hard limit will not allow lowering by unprivileged callers.

Now that we've discussed resource control blocks and how to get and set them, let's take a closer look at the functions we need to manipulate the attributes of a resource control block.

Manipulating a Resource Control's Privilege Level

We mentioned previously that resource control blocks have a privilege level associated with them. We manipulate this privilege level by using the `rctlblk_get_privilege` and `rctlblk_set_privilege` functions.

```
#include <rctl.h>

rctl_priv_t rctlblk_get_privilege (rctlblk_t *rblk);
                                        Returns: resource control block's privilege

void rctlblk_set_privilege (rctlblk_t *rblk, rctl_priv_t privilege);
```

The `rctlblk_get_privilege` returns the privilege level of the resource control block pointed to by *rblk*. Three privilege levels are defined:

`RCPRIV_BASIC`	The resource control can be modified by the owner of the calling process.
`RCPRIV_PRIVILEGED`	The resource control can be modified only by a privileged process (i.e., one whose effective user ID is 0); for other users, the resource limit is read-only. The only exception to this is if the `RCTL_GLOBAL_LOWERABLE` global flag is set for the resource, in which case unprivileged applications can lower the value of the resource limit.
`RCPRIV_SYSTEM`	System resource controls are read-only and are fixed for the duration of the OS instance. They represent the amount of the resource the current instance of the OS is capable of providing.

The `rctlblk_set_privilege` function enables us to set the privilege of the resource control pointed to by *rblk* to the value specified by *privilege* (which must be one of the three values we describe in the previous paragraph). No errors are returned by `rctlblk_set_privilege`; any errors are reported by the `setrctl` function.

Manipulating a Resource Control's Values

The value of a resource control is manipulated using the `rctlblk_get_value`, `rctlblk_get_enforced_value`, and `rctlblk_set_value` functions.

```
#include <rctl.h>

rctl_qty_t rctlblk_get_value (rctlblk_t *rblk);

rctl_qty_t rctlblk_get_enforced_value (rctlblk_t *rblk);
                                        Both return: resource control block's value

void rctlblk_set_value (rctlblk_t *rblk, rctl_qty_t value);
```

The `rctlblk_get_value` function returns the value associated with the resource control block pointed to by *rblk*. The `rctlblk_get_enforced_value` function also returns the value associated with the resource control block pointed to by *rblk*.

Most of the time, the values returned by `rctlblk_get_value` and `rctlblk_get_enforced_value` will be the same. However, in cases where the process, task, or project associated with the control possesses fewer capabilities than allowable by the current value, the value returned by `rctlblk_get_enforced_value` will differ from that returned by `rctlblk_get_value`. An example of when this can happen is if the calling process is using an address space model smaller than the maximum address space model supported by the system (e.g., running a 32-bit process on a 64-bit kernel).

The `rctlblk_set_value` enables us to set the value associated with the resource control block pointed to by *rblk* to the value specified by *value*. No errors are returned by `rctlblk_set_value`; any errors are reported by the `setrctl` function.

Manipulating a Resource Control's Flags

Each resource control has two sets of flags associated with it: the global flags and the local flags. We can manipulate them by using the `rctlblk_get_global_flags`, `rctlblk_get_local_flags`, and `rctlblk_set_local_flags` functions.

```
#include <rctl.h>

int rctlblk_get_global_flags (rctlblk_t *rblk);

int rctlblk_get_local_flags (rctlblk_t *rblk);
                                        Both return: resource control block's flags

void rctlblk_set_local_flags (rctlblk_t *rblk, int flags);
```

The `rctlblk_get_global_flags` function returns the global flags associated with the resource control block pointed to by *rblk*. Global flags are set using the `rctladm` command, and are generally a published property of the control and hence not modifiable. The returned value is the bitwise-OR of zero or more of the following flags:

RCTL_GLOBAL_DENY_ALWAYS	The action taken when a control value is exceeded on this control will always include denial of the resource.
RCTL_GLOBAL_DENY_NEVER	The action taken when a control value is exceeded on this control will never include denial of the resource, although other actions can also be taken.
RCTL_GLOBAL_CPU_TIME	The list of valid signals available as local actions includes SIGXCPU.
RCTL_GLOBAL_FILE_SIZE	The list of valid signals available as local actions includes SIGXFSZ.
RCTL_GLOBAL_INFINITE	The resource supports the concept of unlimited value. This is usually true of only accumulation oriented resources, such as CPU time.
RCTL_GLOBAL_LOWERABLE	This means that unprivileged callers are able to lower the value of privileged resource control values on this control.
RCTL_GLOBAL_NOBASIC	This means that no values with the RCPRIV_BASIC are allowed on this control.
RCTL_GLOBAL_NOLOCALACTION	No local actions are permitted on this control.
RCTL_GLOBAL_UNOBSERVABLE	The resource control (usually on a task- or project-related control) does not support observational control values. A control value with a privilege of RCPRIV_BASIC that is placed by a process on the task or process will generate an action only if the value is exceeded by that process.

As its name suggests, the rctlblk_get_local_flags returns the local flags associated with the resource control block pointed to by *rblk*. At present, only one local flag is defined:

RCTL_LOCAL_MAXIMAL	Setting this flag indicates that this resource control value represents a request for the maximum amount of resource for this control. If the RCTL_GLOBAL_INFINITE global flag is set for this resource control, setting the RCTL_LOCAL_MAXIMAL flag indicates an unlimited resource control value, which can't be exceeded.

The rctlblk_set_local_flags function enables us to set the local flags associated with the resource control block pointed to by *rblk* to the value specified by *flags*. No errors are returned by rctlblk_set_local_flags; any errors are reported by the setrctl function.

Manipulating a Resource Control's Actions

When a resource control's threshold value is exceeded, one of several actions can be triggered. These actions can be manipulated using the `rctlblk_get_global_action`, `rctlblk_get_local_action`, and `rctlblk_set_local_action` functions.

```
#include <rctl.h>

int rctlblk_get_global_action (rctlblk_t *rblk);

int rctlblk_get_local_action (rctlblk_t *rblk, int *signalp);

                                        Both return: resource control block's actions

void rctlblk_set_local_action (rctlblk_t *rblk, rctl_action_t action,
    int signal);
```

The `rctlblk_get_global_action` function returns the global action associated with the resource control block pointed to by *rblk*. Global actions are set using the `rctladm` command. The returned value will be one of the following values:

`RCTL_GLOBAL_NOACTION`	No global action will be taken when a resource control value is exceeded on this control.
`RCTL_GLOBAL_SYSLOG`	A message will be logged using the `syslog` facility when any resource control value on a sequence associated with this control is exceeded.

Similarly, the `rctlblk_get_local_action` function returns the local action associated with the resource control block pointed to by *rblk*. The value returned will be a bitwise-OR of one or more of the following values:

`RCTL_LOCAL_DENY`	If this action is specified, resource requests that exceed the threshold value associated with the resource control will be denied. If the `RCTL_GLOBAL_DENY_ALWAYS` global flag is set for this control, the local action `RCTL_LOCAL_DENY` will always be set on all values of this control. On the contrary, setting the global flag `RCTL_GLOBAL_DENY_NEVER` for this control will always clear the `RCTL_LOCAL_DENY` action on all values of this control.
`RCTL_LOCAL_NOACTION`	No local action will be taken when this resource control is exceeded.
`RCTL_LOCAL_SIGNAL`	The signal specified (see the following paragraph) will be sent to the process that placed this resource control in the value sequence when consumption of the resource exceeds the value associated with the resource control.

If the local action includes sending a signal to the process, the number of the signal that will be sent is stored in the integer pointed to by *signalp*.

The `rctlblk_set_local_action` function enables us to determine the actions that occur when the resource value associated with resource control block pointed to by *rblk* is exceeded. The *action* argument specifies the action to be taken (as described previously), and the *signal* argument enumerates the signal associated with the `RCTL_LOCAL_SIGNAL` action. The set of valid signals is: `SIGABRT`, `SIGXRES`, `SIGHUP`, `SIGSTOP`, `SIGTERM`, and `SIGKILL` (although other signals—specifically `SIGXCPU` and `SIGXFSZ`—may be permitted because of the global properties of a specific control). No errors are returned by `rctlblk_set_local_action`; any errors are reported by the `setrctl` function.

The `rctlblk_get_firing_time` Function

If we want to know if (or when) a resource control has exceeded one of its action values, we can use the `rctlblk_get_firing_time` function.

```
#include <rctl.h>

hrtime_t rctlblk_get_firing_time (rctlblk_t *rblk);
```

Returns: see text

The `rctlblk_get_firing_time` function returns the value of `gethrtime` at the moment the action on the resource control pointed to by *rblk* was taken. This time is measured in nanoseconds since the system was booted.

If the action value for the resource control has not been exceeded for its lifetime on the process, `rctlblk_get_firing_time` returns 0.

The `rctlblk_get_recipient_pid` Function

If we want to know which process ID placed a resource control, we use the `rctlblk_get_recipient_pid` function.

```
#include <rctl.h>

id_t rctlblk_get_recipient_pid (rctlblk_t *rblk);
```

Returns: see text

The `rctlblk_get_recipient_pid` function returns the process ID that placed the resource control pointed to by *rblk*. The process ID is set automatically by the kernel when a process calls `setrctl`.

The `rctl_walk` Function

We can visit all the registered resource controls on the system by using the `rctl_walk` function.

```
#include <rctl.h>

int rctl_walk (int (*callback) (const char *rctlname, void *walk_data),
    void *init_data);
```

Returns: 0 if OK, –1 on error

The `rctl_walk` function walks through all the active resource controls on the system. For each resource control, the function referred to by *callback* is called and is passed two arguments. The first argument, *rctlname*, is the name of the current resource control. The second argument, *walk_data*, can be used by *callback* to record its own state. The callback function should return a non-zero result in the event of an error or if it wants to prematurely terminate the walk; otherwise, it should return zero.

Upon successful completion, `rctl_walk` returns 0. If *callback* returns a non-zero result, or an error occurs when performing the walk, –1 is returned and `errno` is set to indicate the error.

8.6 Resource Control Examples

This section presents a couple of example programs that illustrate some of the points we've made about resource controls.

The examples in this section use two functions to print out resource control blocks; `print_rctls` prints all the resource control blocks associated with the named resource, and `print_rctl` prints out a single resource control block.

Our `print_rctls` Function

Program 8.6 shows our implementation of `print_rctls`.

Print each resource control block

9–12 Allocate some space for a resource control block, and get the first one that matches the resource control name. Return –1 if either fails.

13–14 Print the name of the resource control and its attributes.

15–16 Print the attributes of all the other resource controls of the same name.

17 Free the memory we previously allocated.

sys_info/print_rctl.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <signal.h>
 4 #include <rctl.h>

 5 void print_rctl (rctlblk_t *rblk);

 6 int print_rctls (const char *name)
 7 {
 8     rctlblk_t *rblk;

 9     if ((rblk = malloc (rctlblk_size ())) == NULL)
10         return (-1);

11     if (getrctl (name, NULL, rblk, RCTL_FIRST) == -1)
12         return (-1);

13     printf ("%s:\n", name);
14     print_rctl (rblk);

15     while (getrctl (name, rblk, rblk, RCTL_NEXT) != -1)
16         print_rctl (rblk);

17     free (rblk);

18     return (0);
19 }
```

sys_info/print_rctl.c

Program 8.6 Our implementation of the `print_rctls` function.

Our `print_rctl` Function

Program 8.7 shows the implementation of `print_rctl`.

Print resource control's process ID

24 Print the process ID of the process that placed the resource control. The process ID is retrieved using `rctlblk_get_recipient_pid`.

Print the resource control's privilege

25–39 Get the privilege level using the `rctlblk_get_privilege` function, and print out its name.

Print resource control's values

40–41 Print the value and enforced value of the resource control, getting the values by calling `rctlblk_get_value` and `rctlblk_get_enforced_value` respectively.

Print resource control's global flags

42–62 Retrieve the global flags using `rctlblk_get_global_flags`, and print the name of each flag that is set.

Print resource control's global actions

63–69 Print out the global actions, having first retrieved them using the `rctlblk_get_global_action` function.

sys_info/print_rctl.c

```
20 void print_rctl (rctlblk_t *rblk)
21 {
22     int tmp;
23     int sig;

24     printf ("  Process ID: %ld\n", (long) rctlblk_get_recipient_pid (rblk));

25     printf ("  Privilege: ");
26     switch (rctlblk_get_privilege (rblk)) {
27         case RCPRIV_BASIC:
28             printf ("RCPRIV_BASIC\n");
29             break;

30         case RCPRIV_PRIVILEGED:
31             printf ("RCPRIV_PRIVILEGED\n");
32             break;

33         case RCPRIV_SYSTEM:
34             printf ("RCPRIV_SYSTEM\n");
35             break;

36         default:
37             printf ("Unknown privilege\n");
38             break;
39     }

40     printf ("  Value: %llu\n", rctlblk_get_value (rblk));
41     printf ("  Enforced value: %llu\n", rctlblk_get_enforced_value (rblk));

42     printf ("  Global flags: ");
43     tmp = rctlblk_get_global_flags (rblk);
44     if (tmp & RCTL_GLOBAL_DENY_ALWAYS)
45         printf ("RCTL_GLOBAL_DENY_ALWAYS ");
46     if (tmp & RCTL_GLOBAL_DENY_NEVER)
47         printf ("RCTL_GLOBAL_DENY_NEVER ");
48     if (tmp & RCTL_GLOBAL_CPU_TIME)
49         printf ("RCTL_GLOBAL_CPU_TIME ");
50     if (tmp & RCTL_GLOBAL_FILE_SIZE)
51         printf ("RCTL_GLOBAL_FILE_SIZE ");
52     if (tmp & RCTL_GLOBAL_INFINITE)
53         printf ("RCTL_GLOBAL_INFINITE ");
54     if (tmp & RCTL_GLOBAL_LOWERABLE)
55         printf ("RCTL_GLOBAL_LOWERABLE ");
56     if (tmp & RCTL_GLOBAL_NOBASIC)
57         printf ("RCTL_GLOBAL_NOBASIC ");
58     if (tmp & RCTL_GLOBAL_NOLOCALACTION)
59         printf ("RCTL_GLOBAL_NOLOCALACTION ");
60     if (tmp & RCTL_GLOBAL_UNOBSERVABLE)
61         printf ("RCTL_GLOBAL_UNOBSERVABLE ");
62     printf ("\n");

63     printf ("  Global actions: ");
64     tmp = rctlblk_get_global_action (rblk);
65     if (tmp & RCTL_GLOBAL_NOACTION)
66         printf ("RCTL_GLOBAL_NOACTION ");
67     if (tmp & RCTL_GLOBAL_SYSLOG)
68         printf ("RCTL_GLOBAL_SYSLOG ");
69     printf ("\n");
```

```
 70     printf ("  Local flags: ");
 71     tmp = rctlblk_get_local_flags (rblk);
 72     if (tmp & RCTL_LOCAL_MAXIMAL)
 73         printf ("RCTL_LOCAL_MAXIMAL ");
 74     printf ("\n");
 75     printf ("  Local actions: ");
 76     tmp = rctlblk_get_local_action (rblk, &sig);
 77     if (tmp & RCTL_LOCAL_DENY)
 78         printf ("RCTL_LOCAL_DENY ");
 79     if (tmp & RCTL_LOCAL_NOACTION)
 80         printf ("RCTL_LOCAL_NOACTION ");
 81     if (tmp & RCTL_LOCAL_SIGNAL) {
 82         printf ("RCTL_LOCAL_SIGNAL ");
 83         switch (sig) {
 84             case SIGABRT:
 85                 printf ("(SIGABRT)");
 86                 break;
 87             case SIGXRES:
 88                 printf ("(SIGXRES)");
 89                 break;
 90             case SIGHUP:
 91                 printf ("(SIGHUP)");
 92                 break;
 93             case SIGSTOP:
 94                 printf ("(SIGSTOP)");
 95                 break;
 96             case SIGTERM:
 97                 printf ("(SIGTERM)");
 98                 break;
 99             case SIGKILL:
100                 printf ("(SIGKILL)");
101                 break;
102             case SIGXCPU:
103                 printf ("(SIGXCPU)");
104                 break;
105             case SIGXFSZ:
106                 printf ("(SIGXFSZ)");
107                 break;
108             default:
109                 printf ("(Illegal signal)");
110                 break;
111         }
112     }
113     printf ("\n");
114     printf ("  Firing time: %llu\n\n", rctlblk_get_firing_time (rblk));
115 }
```

sys_info/print_rctl.c

Program 8.7 Our implementation of the `print_rctl` function.

Print resource control's local flags

70–74 Get the local flags using the `rctlblk_get_local_flags` function, and if any are set, print their names.

Print resource control's local actions

75–113 Print the local actions after retrieving them using `rctlblk_get_local_action`. If a signal is to be sent to the process, print the name of the signal to be sent.

Print resource control's firing time

114 Get the most recent time the resource control was triggered (using the `rctlblk_get_firing_time` function to retrieve it), and print it out.

Example: Printing the values of resource controls

Program 8.8 prints the details of all the default resource controls.

sys_info/rctl_walk.c

```
 1 #include <rctl.h>
 2 #include "ssp.h"

 3 extern int print_rctls (const char *name);
 4 static int callback (const char *name, void *pvt);

 5 int main (void)
 6 {
 7     if (rctl_walk (callback, NULL) == -1)
 8         err_msg ("callback failed");

 9     return (0);
10 }

11 static int callback (const char *name, void *pvt)
12 {
13     return (print_rctls (name));
14 }
```

sys_info/rctl_walk.c

Program 8.8 Printing all the resource controls.

When we run Program 8.8, we get results like the following:

```
$ ./rctl_walk
process.max-address-space:
  Process ID: -1
  Privilege: RCPRIV_PRIVILEGED
  Value: 18446744073709551615
  Enforced value: 4294967295
  Global flags: RCTL_GLOBAL_DENY_ALWAYS RCTL_GLOBAL_LOWERABLE
      RCTL_GLOBAL_NOLOCALACTION
  Global actions:
  Local flags: RCTL_LOCAL_MAXIMAL
  Local actions: RCTL_LOCAL_DENY
  Firing time: 0
```

```
    Process ID: -1
    Privilege: RCPRIV_SYSTEM
    Value: 18446744073709551615
    Enforced value: 4294967295
    Global flags: RCTL_GLOBAL_DENY_ALWAYS RCTL_GLOBAL_LOWERABLE
        RCTL_GLOBAL_NOLOCALACTION
    Global actions:
    Local flags: RCTL_LOCAL_MAXIMAL
    Local actions: RCTL_LOCAL_DENY
    Firing time: 0

process.max-file-descriptor:
    Process ID: 2481
    Privilege: RCPRIV_BASIC
    Value: 256
    Enforced value: 256
    Global flags: RCTL_GLOBAL_DENY_ALWAYS RCTL_GLOBAL_LOWERABLE
    Global actions:
    Local flags:
    Local actions: RCTL_LOCAL_DENY
    Firing time: 0

...                                        Output cut for brevity
```

We've had to wrap some of the longer lines so that they will fit onto the page, and trimmed most of the output to save space. There are some points of interest we should note:

1. The enforced values for the `process.max-address-space` resource controls are 4 GB. This is because we compiled Program 8.8 as a 32-bit program. If we recompile Program 8.8 as a 64-bit program, the enforced value of these resource controls reflects the full 64-bit address space.
2. The last block of output from our example is associated with process ID 2481; this is the process ID of the `rctl_walk` program.

Example: Modifying a resource control

Program 8.9 modifies the resource control block for the resource that controls the maximum number of file descriptors for a process: `process.max-file-descriptor`. Our new resource control block replaces the default one, which has a value equal to the current soft limit (256 in our example).

Let's take a closer look at some of the implementation details of Program 8.9.

Allocate memory

17–19 We allocate the memory we need for the two resource control blocks. We need two resource control blocks in this example because of how we've chosen to order the code. If we delete the current resource control before modifying it with our new values, we could use just one resource control block.

sys_info/setrctl.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <signal.h>
 5 #include <rctl.h>
 6 #include <sys/types.h>
 7 #include <sys/stat.h>
 8 #include <fcntl.h>
 9 #include "ssp.h"

10 extern void print_rctl (rctlblk_t *rblk);

11 int main (void)
12 {
13     rctlblk_t *old_rblk;
14     rctlblk_t *new_rblk;
15     char *name = "process.max-file-descriptor";
16     int fd;

17     if (((old_rblk = malloc (rctlblk_size ())) == NULL) ||
18         ((new_rblk = malloc (rctlblk_size ())) == NULL))
19         err_msg ("malloc failed");

20     if (getrctl (name, NULL, old_rblk, RCTL_FIRST) == -1)
21         err_msg ("getrctl failed");

22     memcpy (new_rblk, old_rblk, rctlblk_size ());

23     printf ("Before...\n");
24     print_rctl (old_rblk);

25     rctlblk_set_value (new_rblk, 10);
26     rctlblk_set_local_action (new_rblk, RCTL_LOCAL_DENY | RCTL_LOCAL_SIGNAL,
27         SIGTERM);

28     if (setrctl (name, NULL, old_rblk, RCTL_DELETE) == -1)
29         err_msg ("setrctl (RCTL_DELETE) failed");
30     if (setrctl (name, NULL, new_rblk, RCTL_INSERT) == -1)
31         err_msg ("setrctl (RCTL_INSERT) failed");

32     if (getrctl (name, NULL, new_rblk, RCTL_FIRST) == -1)
33         err_msg ("getrctl failed");

34     printf ("After...\n");
35     print_rctl (new_rblk);

36     for (;;) {
37         fd = open ("/tmp", O_RDONLY);
38         printf ("Returned file descriptor = %d\n", fd);
39     }
40 }
```

sys_info/setrctl.c

Program 8.9 Modifying a resource control.

Get the current resource control

20–21 Get the current value of the first resource control matching the one we are interested in.

22 Use the current resource control as a template for our new one.

Print the resource control before changing it

23–24 Call `print_rctl` to print the resource control before we modify it.

Modify the resource control

25 Set the threshold value for this resource control to 10.

26–27 Set up the local action flags to deny access to the resource and to send the signal `SIGTERM` to the process when the threshold limit is exceeded. We must specify the `RCTL_LOCAL_DENY` action because the `process.max-file-descriptor` resource control has the `RCTL_GLOBAL_DENY_ALWAYS` flag set; not doing so will cause the `setrctl` function to fail. We also specify `RCTL_LOCAL_SIGNAL` to arrange for a signal to be sent at the appropriate time (i.e., when an attempt to use more than 10 file descriptors is made by the process).

Replace the resource control

28–31 The original resource control is deleted, and then a new one with our new threshold value and actions is inserted. Note that we must perform this task as two separate steps; although intuition might tell us otherwise, we can't simply replace the original resource control with our new one by passing `RCTL_REPLACE` to the `setrctl` function.

Print the resource control after changing it

34–35 Call `print_rctl` to print the resource control before we modify it.

Use up all available file descriptors

36–39 This infinite loop repeatedly calls `open` to consume file descriptors. When the number of file descriptors exceeds the threshold value (which in this example is 10), a `SIGTERM` signal is sent to the process, terminating it. A real application would likely do something more useful than just terminating, like logging an error or trying to free up some of the overallocated resource.

Although we used `SIGTERM` in this example, `SIGXRES` is a more logical choice. By default, `SIGXRES` is ignored, so we would have to set up a signal handler to do anything useful when the signal is received. We show how to do this in Chapter 17.

Let's see what happens when we run Program 8.9:

```
$ ./setrctl
Before...
  Process ID: 1460
  Privilege: RCPRIV_BASIC
  Value: 256
  Enforced value: 256
  Global flags: RCTL_GLOBAL_DENY_ALWAYS RCTL_GLOBAL_LOWERABLE
  Global actions:
  Local flags:
  Local actions: RCTL_LOCAL_DENY
  Firing time: 0
```

```
After...
  Process ID: 1460
  Privilege: RCPRIV_BASIC
  Value: 10
  Enforced value: 10
  Global flags: RCTL_GLOBAL_DENY_ALWAYS RCTL_GLOBAL_LOWERABLE
  Global actions:
  Local flags:
  Local actions: RCTL_LOCAL_DENY RCTL_LOCAL_SIGNAL (SIGTERM)
  Firing time: 0

Returned file descriptor = 3
Returned file descriptor = 4
Returned file descriptor = 5
Returned file descriptor = 6
Returned file descriptor = 7
Returned file descriptor = 8
Returned file descriptor = 9
Terminated
```

Notice that the enforced value was automatically lowered to match the new value.

8.7 Resource Usage Information

Solaris provides two methods that enable us to monitor our resource consumption: the `times`, `clock`, and `getrusage` functions, and the `/proc` file system.

The `times` Function

We can use the `times` function to determine the CPU usage for the calling process and its children.

```
#include <sys/times.h>
#include <limits.h>

clock_t times (struct tms *buffer);
```

Returns: the elapsed time since the system booted if OK, –1 on error

The `times` function retrieves the system and user CPU time for the calling process and its children, placing the result in the `tms` structure pointed to by *buffer*. The `tms` structure has the following members:

```
struct tms {
    clock_t tms_utime;      /* User time */
    clock_t tms_stime;      /* System time */
    clock_t tms_cutime;     /* User time, children */
    clock_t tms_cstime;     /* System time, children */
};
```

The definition of the members is a follows:

`tms_utime`	This is the amount of CPU time spent executing instructions in the user space of the calling process.
`tms_stime`	This is the amount of CPU time spent executing kernel code on behalf of the calling process (i.e., executing system calls).
`tms_cutime`	This is the sum of the `tms_utime` and `tms_cutime` values for all of the children of the calling process.
`tms_cstime`	This is the sum of the `tms_stime` and `tms_cstime` values for all of the children of the calling process.

All of the times are reported in clock ticks; there are `CLK_TCK` clock ticks per second.

The times of terminated children are included in the `tms_cutime` and `tms_cstime` members of the parent when one of the `wait` functions we describe in Chapter 15 returns the process ID of the terminated child. If a process doesn't `wait` for its children, their times will not be included in its times.

Upon successful completion, the `times` function returns the elapsed real time since some arbitrary point in the past (e.g., the system boot time), in clock ticks. This point does not change from one invocation of `times` in a process to another. If an error occurs, –1 is returned.

Although the Solaris implementation of `times` uses the system boot as its "arbitrary point in the past", this event is not codified by any standards, so portable applications shouldn't rely on it.

Example: Timing the execution of command line arguments

Program 8.10 uses the `system` function we describe in Section 15.13 to run each of its command line arguments, timing how long the command takes to run and printing the results. It also calls the `print_term_status` function we show in Section 15.6.

Running Program 8.10 gives us the following results:

```
$ ./ssp_time "sleep 2" date
Command: sleep 2
  Real:   2.040
  User:   0.000
  Sys:    0.000
  Child user:   0.010
  Child sys:    0.020
Normal termination; exit status = 0

Command: date
Mon Jan 27 12:25:33 PST 2003
  Real:   0.060
  User:   0.000
  Sys:    0.000
  Child user:   0.010
  Child sys:    0.030
Normal termination; exit status = 0
```

sys_info/ssp_time.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/times.h>
 5 #include <limits.h>
 6 #include "ssp.h"

 7 static void proc_cmd (char *cmd);
 8 static void print_times (clock_t real, struct tms *start, struct tms *end);

 9 int main (int argc, char **argv)
10 {
11     int i;

12     for (i = 1; i < argc; i++)
13         proc_cmd (argv [i]);

14     return (0);
15 }

16 static void proc_cmd (char *cmd)
17 {
18     struct tms start_tms;
19     struct tms end_tms;
20     clock_t start;
21     clock_t end;
22     int status;

23     printf ("Command: %s\n", cmd);

24     if ((start = times (&start_tms)) == -1)
25         err_msg ("Start times failed");

26     if ((status = system (cmd)) == -1)
27         err_msg ("system failed");

28     if ((end = times (&end_tms)) == -1)
29         err_msg ("End times failed");

30     print_times (end - start, &start_tms, &end_tms);
31     print_term_status (status);
32     printf ("\n");
33 }

34 static void print_times (clock_t real, struct tms *start, struct tms *end)
35 {
36     static double tps = 0.0;

37     if (tps == 0.0)
38         if ((tps = (double) sysconf (_SC_CLK_TCK)) == -1)
39             err_msg ("sysconf failed");

40     printf ("  Real: %7.3f\n", real / tps);
41     printf ("  User: %7.3f\n", (end -> tms_utime - start -> tms_utime) / tps);
42     printf ("  Sys:  %7.3f\n", (end -> tms_stime - start -> tms_stime) / tps);
43     printf ("  Child user: %7.3f\n",
44         (end -> tms_cutime - start -> tms_cutime) / tps);
```

```
45     printf ("  Child sys:  %7.3f\n",
46         (end -> tms_cstime - start -> tms_cstime) / tps);
47 }
```

sys_info/ssp_time.c

Program 8.10 Time how long each command line argument takes to run.

Note that in this example, all the CPU time is attributed to the child process. This is because it is the child process that runs the shell and the specified command.

We can compare the output from Program 8.10 to the `time` command:

```
$ time ls -lR > /dev/null

real    0m9.00s
user    0m2.78s
sys     0m1.75s
$ ./ssp_time "ls -lR > /dev/null"
Command: ls -lR > /dev/null
  Real:   6.510
  User:   0.000
  Sys:    0.000
  Child user:   2.270
  Child sys:    1.300
Normal termination; exit status = 0
```

As we expect, the CPU times are very similar. The reason for the disparity with the real time is that the second `ls` command benefits from the fact that the results of the directory lookups from the first command are cached by the kernel.

The `clock` Function

The `clock` function enables us to determine the CPU usage for the caller and its reaped children, starting from a specific event.

```
#include <time.h>

clock_t clock (void);
```

Returns: the CPU usage of the caller and its reaped children if OK, –1 on error

The `clock` function returns the total amount of user and system CPU time (in microseconds) used since the first call to `clock` in the calling process. The time returned is the sum of the user and system time for the calling process and all of its terminated children that have been waited for. Dividing the value returned by the constant `CLOCKS_PER_SEC` will convert the time into seconds.

Because the value returned is measured in microseconds, it is possible that the counter will wraparound in long-running 32-bit processes; this wraparound will occur after 2147 seconds (about 36 minutes) of CPU time have accumulated.

The `getrusage` Function

A process can determine its resource usage (or that of its terminated children) by calling the `getrusage` function.

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

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

Returns: 0 if OK, –1 on error

The `getrusage` function is used to gather comprehensive process resource usage information. The value of the *who* argument determines which process the resource usage information is gathered for: the calling process, or its children. If *who* is `RUSAGE_SELF`, then resource usage information for the calling process is returned, and if *who* is `RUSAGE_CHILDREN`, then resource usage information for the terminated and waited for children of the calling process is returned. If a child is not waited for (for example, if the parent sets `SA_NOCLDWAIT` or ignores the `SIGCHLD` signal), the resource usage information for the child is discarded and not included in the resource usage information provided by `getrusage`.

The resource usage information is stored in the `rusage` structure pointed to by *r_usage*. The `rusage` structure has the following members:

```
struct rusage {
    struct timeval  ru_utime;       /* User time */
    struct timeval  ru_stime;       /* System time */
    long            ru_maxrss;      /* Maximum resident set size */
    long            ru_ixrss;       /* Integral shared memory size */
    long            ru_idrss;       /* Integral unshared data size */
    long            ru_isrss;       /* Integral unshared stack size */
    long            ru_minflt;      /* Minor page faults */
    long            ru_majflt;      /* Major page faults */
    long            ru_nswap;       /* Number of swaps */
    long            ru_inblock;     /* Block input operations */
    long            ru_oublock;     /* Block output operations */
    long            ru_msgsnd;      /* Messages sent */
    long            ru_msgrcv;      /* Messages received */
    long            ru_nsignals;    /* Number of signals received */
    long            ru_nvcsw;       /* Voluntary context switches */
    long            ru_nivcsw;      /* Involuntary context switches */
};
```

The members of the `rusage` structure are interpreted as follows:

`ru_utime` This is the amount of CPU time spent executing instructions in user space. The time is measured in seconds and microseconds.

`ru_stime` This is the amount of CPU time spent executing instructions in the kernel. The time is measured in seconds and microseconds.

`ru_maxrss`	This is the maximum resident set size measured in pages.
`ru_idrss`	This is an "integral" value indicating the amount of memory in use by a process while it is running. This value is the sum of the resident set sizes of the process when a clock tick occurs. The value is given in pages multiplied by clock ticks. It does not take shared pages into account.
`ru_minflt`	This is the number of minor page faults serviced. A *minor page fault* is one that does not require any physical I/O activity. An example of a minor page fault would be when a process starts up and refers to pages already in memory (e.g., those in a shared library like `libc.so`).
`ru_majflt`	This is the number of major page faults serviced. A *major page fault* is one that requires physical I/O activity. An example of a major page fault would be when a process starts up for the first time since the system was booted; the pages of the executable must be paged in from disk before the program can run.
`ru_nswap`	This is the number of times the process was swapped out of physical memory.
`ru_inblock`	This is the number of times the kernel had to perform input when servicing a `read` request.
`ru_oublock`	This is the number of times the kernel had to perform output when servicing a `write` request.
`ru_msgsnd`	This is the number of messages sent over sockets.
`ru_msgrcv`	This is the number of messages received over sockets.
`ru_nsignals`	This is the number of signals that have been delivered.
`ru_nvcsw`	This is the number of voluntary context switches. A voluntary context switch occurs when a process gives up the CPU before its time slice has expired. This is usually because the process is awaiting the availability of a resource.
`ru_nivcsw`	This is the number of involuntary context switches. An involuntary context switch occurs when a higher priority process has become runnable, or because the process has exceeded its time slice.

In the Solaris implementation of `getrusage`, the `ru_maxrss`, `ru_ixrss`, `ru_idrss`, and `ru_isrss` members of the `rusage` structure are set to 0 (these members are present only to maintain backward source compatibility with SunOS 4.x). The `psinfo` object in the `/proc` file system can be read if we want to know the resident set size of a process. (Note that the presence of the `getrusage` functions is mandated by various industry standards, but the information it returns is not.)

The most flexible way to determine the resource usage of a process is to use the `/proc` file system, which we describe next.

8.8 Determining Resource Usage Using the /proc File System

Historically, process data such as that obtained using the ps utility could be obtained only by directly reading kernel memory. The problem with this approach is that as well as being inherently non-portable, it requires the use of superuser privileges. To get around these problems, the /proc file system provides a general interface to the memory image of processes.

The /proc file system contains one directory for each process currently running on the system; the name of the directory is the same as the process ID of the process to which it refers. The owner and group of the directory are set to the real user ID and group ID of the process the directory is associated with. (There is another invisible alias a process can use to refer to itself; opening /proc/self is the same as opening /proc/*PID*, where *PID* is the process ID of the process. /proc/self is invisible in the sense that the name self does not appear in directory listing of /proc.)

Inside each directory is a number of files and subdirectories that contain information about the process. For example, the file as contains the address space of the process, psinfo contains information used by the ps command, and (of most interest to us in this discussion) usage, which contains the resource usage of the process.

One of the directories under /proc/*PID* is called lwp, and it contains one directory for each lightweight process (LWP) associated with the process. Each of those directories contains a number of files holding LWP-specific information.

From the perspective of a user process, a *lightweight process* can be thought of as a virtual CPU. LWPs are actually kernel entities, similar to kernel threads (kthreads). A single threaded program will have exactly one LWP and one kthread in its address space, but there is not necessarily a one-to-one correspondence between the number of threads in a process and the number of LWPs. LWPs maintain a control structure in which they store the hardware context (i.e., CPU registers) when a thread is context-switched off a processor. Readers interested in an in-depth discussion of the Solaris multithreaded architecture are encouraged to read Chapter 8 of [Mauro and McDougall 2001].

Most files in the /proc hierarchy can be opened only for reading, and although process state—and therefore the contents of /proc files—can change from instant to instant, a single read of a file in /proc is guaranteed to be sane. In other words, the read will be atomic with respect to the state of the process. The only exception to this is that atomicity is not guaranteed for I/O applied to the as (address space) file for a running process, or for a process whose address space contains memory shared with other running processes.

Standard system calls are used to interface with the /proc files, including open, close, read, and write.

> Prior to Solaris 2.6, /proc was not a directory hierarchy. Instead, the entries in /proc were files—one for each process. Rather than reading a file to obtain the required information, ioctl commands were used. For example, the PIOCUSAGE command was used to obtain the resource usage of the process. Although the ioctl method of obtaining process information is still supported for binary compatibility reasons, new programs should use the directory hierarchy method we describe in this text.

As we said earlier, each process directory contains a number of files that contain information about the process. The contents of many of these files can be read into a structure (which structure depends of course on the file we are reading). For the purpose of obtaining the resource usage information for a process, we must read the contents of the file `usage` into a `prusage` structure. This structure is defined in `<sys/procfs.h>` (which is included by `<procfs.h>`), and has the following members:

```
typedef struct prusage {
    id_t        pr_lwpid;       /* LWP ID */
    int         pr_count;       /* Number of contributing LWPs */
    timestruc_t pr_tstamp;      /* Current time stamp */
    timestruc_t pr_create;      /* Process/LWP creation time stamp */
    timestruc_t pr_term;        /* Process/LWP termination time */
    timestruc_t pr_rtime;       /* Total LWP real (elapsed) time */
    timestruc_t pr_utime;       /* User level CPU time */
    timestruc_t pr_stime;       /* System call CPU time */
    timestruc_t pr_ttime;       /* Other system trap CPU time */
    timestruc_t pr_tftime;      /* Text page fault sleep time */
    timestruc_t pr_dftime;      /* Data page fault sleep time */
    timestruc_t pr_kftime;      /* Kernel page fault sleep time */
    timestruc_t pr_ltime;       /* User local wait sleep time */
    timestruc_t pr_slptime;     /* All other sleep time */
    timestruc_t pr_wtime;       /* Wait-CPU (latency) time */
    timestruc_t pr_stoptime;    /* Stopped time */
    timestruc_t filltime [6];   /* Filler for future expansion */
    ulong_t     pr_minf;        /* Minor page faults */
    ulong_t     pr_majf;        /* Major page faults */
    ulong_t     pr_nswap;       /* Swaps */
    ulong_t     pr_inblk;       /* Input blocks */
    ulong_t     pr_oublk;       /* Output blocks */
    ulong_t     pr_msnd;        /* Messages sent */
    ulong_t     pr_mrcv;        /* Messages received */
    ulong_t     pr_sigs;        /* Signals received */
    ulong_t     pr_vctx;        /* Voluntary context switches */
    ulong_t     pr_ictx;        /* Involuntary context switches */
    ulong_t     pr_sysc;        /* System calls */
    ulong_t     pr_ioch;        /* Characters read and written */
    ulong_t     filler [10];    /* Filler for future expansion */
} prusage_t;
```

The members of the `prusage` structure are interpreted as follows:

`pr_lwpid` This is the lightweight process ID. This will be 0 for regular processes.

`pr_count` This is the number of LWPs that contribute to the process totals.

`pr_tstamp` This is a time stamp; it contains the time at which the `read` of the `prusage` structure was performed, measured in seconds and nanoseconds since the last reboot.

`pr_create` This is the creation time of the process or LWP, measured in seconds and nanoseconds since the last reboot.

pr_term	This is the termination time of the process or LWP, measured in seconds and nanoseconds since the last reboot.
pr_rtime	This is the total elapsed time of the process or LWP. Like all time values in this structure, the time is measured in seconds and nanoseconds.
pr_utime	This is the amount of CPU time spent executing instructions in user space.
pr_stime	This is the amount of CPU time spent executing instructions in the kernel.
pr_ttime	This is the amount of CPU time spent performing system trap instructions, other than that covered by pr_stime.
pr_tftime	This is the amount of time spent waiting for program text page faults to be serviced.
pr_dftime	This is the amount of time spent waiting for data page faults to be serviced.
pr_kftime	This is the amount of time spent waiting for kernel page faults to be serviced.
pr_ltime	This is the amount of time spent waiting for user locks.
pr_slptime	This is the total amount of time spent sleeping for reasons not covered by pr_tftime, pr_dftime, pr_kftime, and pr_ltime.
pr_wtime	This is the amount of time spent waiting for the CPU (in other words, the CPU latency).
pr_stoptime	This is the amount time the process was stopped for.
pr_minf	This is the number of minor faults serviced.
pr_majf	This is the number of major faults serviced.
pr_nswap	This is the number of times the process was swapped out of physical memory.
pr_inblk	This is the number of blocks of physical input required to service read requests.
pr_oublk	This is the number of blocks of physical output required to service write requests.
pr_msnd	This is the number of messages sent over sockets.
pr_mrcv	This is the number of messages received over sockets.
pr_sigs	This is the number of signals that have been delivered.
pr_vctx	This is the number of voluntary context switches.
pr_ictx	This is the number of involuntary context switches.
pr_sysc	This is the number of system calls that have been made.

pr_ioch	This is the number of characters that have been read or written.

If microstate accounting has not been enabled, the times reported for the various states are only estimates.

Example: Implementing the getprusage function

Program 8.11 shows our implementation of a function similar to getrusage called getprusage. Our function takes two arguments. The first is the process ID that we want to get the resource usage for, and the second is a pointer to a prusage structure, which is used to hold the results. Setting the process ID to −1 will return the resource usage for the calling process.

If the resource usage for the requested process ID is successfully acquired, getprusage returns 0. Otherwise −1 is returned, and errno is set to indicate the error.

Running Program 8.11 to obtain the resource usage information for a long running process (in this case, init) gives the following results:

```
$ ./getprusage1 1
Resource usage for PID 1:
  LWP ID: 0
  Number of LWPs: 1
  Timestamp: 1795343.11543115
  Creation time: 54.248659987
  Termination time: 0.0
  Real (elapsed) time: 1795288.757557953
  User CPU time: 0.230000000
  System CPU time: 0.220000000
  System trap CPU time: 0.0
  Text page fault CPU time: 0.0
  Data page fault CPU time: 0.0
  Kernel page fault CPU time: 0.0
  User lock wait time: 0.0
  Other sleep time: 1795288.310000000
  CPU latency time: 0.0
  Stopped time: 0.0
  Minor faults: 0
  Major faults: 80
  Number of swaps: 0
  Input blocks: 94
  Output blocks: 1
  Messages sent: 0
  Messages received: 0
  Signals received: 6517
  Voluntary context switches: 6519
  Involuntary context switches: 79
  System calls: 143924
  Characters read/written: 13803067
```

We can make a number of observations about these results:

sys_info/getprusage1.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <fcntl.h>
 5 #include <sys/types.h>
 6 #include <sys/stat.h>
 7 #include <sys/resource.h>
 8 #include <procfs.h>
 9 #include <limits.h>
10 #include "ssp.h"

11 static int getprusage (pid_t pid, prusage_t *pr_usage);
12 static void print_rusage (pid_t pid, prusage_t *buf);

13 int main (int argc, char **argv)
14 {
15     pid_t pid;
16     prusage_t buf;
17     int i;

18     if (argc == 1) {
19         if (getprusage (-1, &buf) == -1)
20             err_msg ("getprusage failed");
21         print_rusage (getpid (), &buf);
22     }
23     else {
24         for (i = 1; i < argc; i++) {
25             pid = atoi (argv [i]);
26             if (getprusage (pid, &buf) == -1)
27                 err_ret ("getprusage failed");
28             else
29                 print_rusage (pid, &buf);
30         }
31     }

32     return (0);
33 }

34 static int getprusage (pid_t pid, prusage_t *pr_usage)
35 {
36     int fd;
37     char name [PATH_MAX];

38     if (pid == -1)
39         snprintf (name, PATH_MAX, "/proc/self/usage");
40     else
41         snprintf (name, PATH_MAX, "/proc/%ld/usage", (long) pid);

42     if ((fd = open (name, O_RDONLY)) == -1)
43         return (-1);

44     if (read (fd, pr_usage, sizeof (prusage_t)) == -1) {
45         close (fd);
46         return (-1);
47     }
```

```
48      else {
49          close (fd);
50          return (0);
51      }
52  }
53  static void print_rusage (pid_t pid, prusage_t *buf)
54  {
55      printf ("Resource usage for PID %ld:\n", (long) pid);
56      printf ("  LWP ID: %ld\n", (long) buf -> pr_lwpid);
57      printf ("  Number of LWPs: %d\n", buf -> pr_count);
58      printf ("  Timestamp: %ld.%ld\n", buf -> pr_tstamp.tv_sec,
59          buf -> pr_tstamp.tv_nsec);
60      printf ("  Creation time: %ld.%ld\n", buf -> pr_create.tv_sec,
61          buf -> pr_create.tv_nsec);
62      printf ("  Termination time: %ld.%ld\n", buf -> pr_term.tv_sec,
63          buf -> pr_term.tv_nsec);
64      printf ("  Real (elapsed) time: %ld.%ld\n", buf -> pr_rtime.tv_sec,
65          buf -> pr_rtime.tv_nsec);
66      printf ("  User CPU time: %ld.%ld\n", buf -> pr_utime.tv_sec,
67          buf -> pr_utime.tv_nsec);
68      printf ("  System CPU time: %ld.%ld\n", buf -> pr_stime.tv_sec,
69          buf -> pr_stime.tv_nsec);
70      printf ("  System trap CPU time: %ld.%ld\n", buf -> pr_ttime.tv_sec,
71          buf -> pr_ttime.tv_nsec);
72      printf ("  Text page fault CPU time: %ld.%ld\n", buf -> pr_tftime.tv_sec,
73          buf -> pr_tftime.tv_nsec);
74      printf ("  Data page fault CPU time: %ld.%ld\n", buf -> pr_dftime.tv_sec,
75          buf -> pr_dftime.tv_nsec);
76      printf ("  Kernel page fault CPU time: %ld.%ld\n", buf -> pr_kftime.tv_sec,
77          buf -> pr_kftime.tv_nsec);
78      printf ("  User lock wait time: %ld.%ld\n", buf -> pr_ltime.tv_sec,
79          buf -> pr_ltime.tv_nsec);
80      printf ("  Other sleep time: %ld.%ld\n", buf -> pr_slptime.tv_sec,
81          buf -> pr_slptime.tv_nsec);
82      printf ("  CPU latency time: %ld.%ld\n", buf -> pr_wtime.tv_sec,
83          buf -> pr_wtime.tv_nsec);
84      printf ("  Stopped time: %ld.%ld\n", buf -> pr_stoptime.tv_sec,
85          buf -> pr_stoptime.tv_nsec);
86      printf ("  Minor faults: %ld\n", buf -> pr_minf);
87      printf ("  Major faults: %ld\n", buf -> pr_majf);
88      printf ("  Number of swaps: %ld\n", buf -> pr_nswap);
89      printf ("  Input blocks: %ld\n", buf -> pr_inblk);
90      printf ("  Output blocks: %ld\n", buf -> pr_oublk);
91      printf ("  Messages sent: %ld\n", buf -> pr_msnd);
92      printf ("  Messages received: %ld\n", buf -> pr_mrcv);
93      printf ("  Signals received: %ld\n", buf -> pr_sigs);
94      printf ("  Voluntary context switches: %ld\n", buf -> pr_vctx);
95      printf ("  Involuntary context switches: %ld\n", buf -> pr_ictx);
96      printf ("  System calls: %ld\n", buf -> pr_sysc);
97      printf ("  Characters read/written: %ld\n", buf -> pr_ioch);
98  }
```

sys_info/getprusage1.c

Program 8.11 Our implementation of `getprusage`.

1. The init process started less than one minute after the machine booted. This isn't surprising, because init is the first non-system process created when the Solaris kernel starts running.
2. Despite running for about 20 days, less than one second of CPU time has been consumed by the process.
3. More than 6500 signals have been received, and nearly 144,000 system calls have been made.

Example: Implementing getprusage using the PIOCUSAGE ioctl

Program 8.12 shows another implementation of our getprusage function. This time, however, we have used the PIOCUSAGE ioctl. We've done this to show how to use the older ioctl method of obtaining process information from /proc.

Program 8.12 is almost the same as Program 8.11. The only differences are the implementation of the getprusage function, and that we must include <sys/old_procfs.h> instead of <procfs.h> (if we are compiling Program 8.12 on Solaris 2.5 or Solaris 2.5.1, we must include <sys/procfs.h>; we perform this behind the scenes sleight of hand by checking if NEED_SNPRINTF—which we define only when compiling on Solaris 2.5 or 2.5.1—is defined).

Let's see what happens when we run Program 8.12:

```
$ ./getprusage2 1
getprusage failed: Permission denied
$ ps
   PID TTY      TIME CMD
   503 pts/10   0:01 ksh
 18503 pts/10   0:00 ps
$ ./getprusage2 503
Resource usage for PID 503:
  LWP ID: 0
  Number of LWPs: 1
  Timestamp: 267011.76709783
  Creation time: 1977.486278389
  Termination time: 0.0
  Real (elapsed) time: 265033.589329198
  User CPU time: 0.228333337
  System CPU time: 0.650211029
  System trap CPU time: 0.7082816
  Text page fault CPU time: 0.0
  Data page fault CPU time: 0.0
  Kernel page fault CPU time: 0.0
  User lock wait time: 0.0
  Other sleep time: 265032.598291161
  CPU latency time: 0.109213722
  Stopped time: 0.0
  Minor faults: 0
  Major faults: 8
  Number of swaps: 0
  Input blocks: 8
```

sys_info/getprusage2.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <stdlib.h>
 4 #include <fcntl.h>
 5 #include <sys/types.h>
 6 #include <sys/stat.h>
 7 #include <sys/resource.h>
 8 #ifdef NEED_SNPRINTF
 9 #include <sys/procfs.h>
10 #else
11 #include <sys/old_procfs.h>
12 #endif
13 #include <limits.h>
14 #include "ssp.h"

15 static int getprusage (pid_t pid, prusage_t *pr_usage);
16 static void print_rusage (pid_t pid, prusage_t *buf);

17 int main (int argc, char **argv)
18 {
19     pid_t pid;
20     prusage_t buf;
21     int i;

22     if (argc == 1) {
23         if (getprusage (-1, &buf) == -1)
24             err_msg ("getprusage failed");
25         print_rusage (getpid (), &buf);
26     }
27     else {
28         for (i = 1; i < argc; i++) {
29             pid = atoi (argv [i]);
30             if (getprusage (pid, &buf) == -1)
31                 err_ret ("getprusage failed");
32             else
33                 print_rusage (pid, &buf);
34         }
35     }

36     return (0);
37 }

38 static int getprusage (pid_t pid, prusage_t *pr_usage)
39 {
40     int fd;
41     char name [PATH_MAX];

42     if (pid == -1)
43         snprintf (name, PATH_MAX, "/proc/%ld", (long) getpid ());
44     else
45         snprintf (name, PATH_MAX, "/proc/%ld", (long) pid);

46     if ((fd = open (name, O_RDONLY)) == -1)
47         return (-1);
```

```
48      if (ioctl (fd, PIOCUSAGE, pr_usage) == -1) {
49          close (fd);
50          return (-1);
51      }
52      else {
53          close (fd);
54          return (0);
55      }
56  }
57  static void print_rusage (pid_t pid, prusage_t *buf)
58  {
59      printf ("Resource usage for PID %ld:\n", (long) pid);
60      printf ("  LWP ID: %ld\n", (long) buf -> pr_lwpid);
61      printf ("  Number of LWPs: %d\n", (int) buf -> pr_count);
62      printf ("  Timestamp: %ld.%ld\n", buf -> pr_tstamp.tv_sec,
63          buf -> pr_tstamp.tv_nsec);
64      printf ("  Creation time: %ld.%ld\n", buf -> pr_create.tv_sec,
65          buf -> pr_create.tv_nsec);
66      printf ("  Termination time: %ld.%ld\n", buf -> pr_term.tv_sec,
67          buf -> pr_term.tv_nsec);
68      printf ("  Real (elapsed) time: %ld.%ld\n", buf -> pr_rtime.tv_sec,
69          buf -> pr_rtime.tv_nsec);
70      printf ("  User CPU time: %ld.%ld\n", buf -> pr_utime.tv_sec,
71          buf -> pr_utime.tv_nsec);
72      printf ("  System CPU time: %ld.%ld\n", buf -> pr_stime.tv_sec,
73          buf -> pr_stime.tv_nsec);
74      printf ("  System trap CPU time: %ld.%ld\n", buf -> pr_ttime.tv_sec,
75          buf -> pr_ttime.tv_nsec);
76      printf ("  Text page fault CPU time: %ld.%ld\n", buf -> pr_tftime.tv_sec,
77          buf -> pr_tftime.tv_nsec);
78      printf ("  Data page fault CPU time: %ld.%ld\n", buf -> pr_dftime.tv_sec,
79          buf -> pr_dftime.tv_nsec);
80      printf ("  Kernel page fault CPU time: %ld.%ld\n", buf -> pr_kftime.tv_sec,
81          buf -> pr_kftime.tv_nsec);
82      printf ("  User lock wait time: %ld.%ld\n", buf -> pr_ltime.tv_sec,
83          buf -> pr_ltime.tv_nsec);
84      printf ("  Other sleep time: %ld.%ld\n", buf -> pr_slptime.tv_sec,
85          buf -> pr_slptime.tv_nsec);
86      printf ("  CPU latency time: %ld.%ld\n", buf -> pr_wtime.tv_sec,
87          buf -> pr_wtime.tv_nsec);
88      printf ("  Stopped time: %ld.%ld\n", buf -> pr_stoptime.tv_sec,
89          buf -> pr_stoptime.tv_nsec);
90      printf ("  Minor faults: %ld\n", buf -> pr_minf);
91      printf ("  Major faults: %ld\n", buf -> pr_majf);
92      printf ("  Number of swaps: %ld\n", buf -> pr_nswap);
93      printf ("  Input blocks: %ld\n", buf -> pr_inblk);
94      printf ("  Output blocks: %ld\n", buf -> pr_oublk);
95      printf ("  Messages sent: %ld\n", buf -> pr_msnd);
96      printf ("  Messages received: %ld\n", buf -> pr_mrcv);
97      printf ("  Signals received: %ld\n", buf -> pr_sigs);
98      printf ("  Voluntary context switches: %ld\n", buf -> pr_vctx);
99      printf ("  Involuntary context switches: %ld\n", buf -> pr_ictx);
100     printf ("  System calls: %ld\n", buf -> pr_sysc);
101     printf ("  Characters read/written: %ld\n", buf -> pr_ioch);
102 }
```

sys_info/getprusage2.c

Program 8.12 Implementing `getprusage` using `ioctl`.

```
Output blocks: 3
Messages sent: 0
Messages received: 0
Signals received: 6
Voluntary context switches: 1169
Involuntary context switches: 686
System calls: 10673
Characters read/written: 178966
```

Notice that when we try to get the resource usage information for `init` this time, we get an error (Permission denied). This is because the `ioctl` method of obtaining information from `/proc` only works for processes whose effective user ID is the same as the program trying to obtain the information (running Program 8.12 as `root` would of course work around this restriction).

In our example output, we use the `ps` command to list some processes we can get the resource usage for. We chose the process for our shell, the process ID of which is 503. This time, the program works as expected.

8.9 Determining the System's Load Average

It is sometimes useful to determine the system's current load average. A process can use this information to decided whether to perform a given task. For example, Sendmail can be configured to refuse connections when the load average exceeds a certain threshold. But before we show how to determine the current load average, we need to describe what a load average is.

The *load average* is the number of runnable and running processes (the *load*), averaged over a certain time interval. Solaris (and many other implementations of UNIX) uses time intervals of 60, 300, and 900 seconds (i.e., 1, 5, and 15 minutes respectively) to calculate the load average. Note that the load average does not take into account the number of CPUs the system has; whether or not a given load average is cause for concern is dependant on this piece of information (which we can determine by calling the `sysconf` function we describe in Section 8.3). For example, a load average of 10 on a uniprocessor machine is likely to be a problem, but that same load average on a 16-CPU system is not an issue. (As a general rule of thumb, a load average that is less than four times the number of CPUs is acceptable.)

From Solaris 7, the load average for the last 60, 300, and 900 seconds can be ascertained by calling the `getloadavg` function.

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

int getloadavg (double loadavg [], int nelem);
```

Returns: the number of samples retrieved if OK, –1 on error

The `getloadavg` function retrieves up to *nelem* load averages, placing them in successive elements of *loadavg*. The system imposes a maximum of three samples, representing the load averaged over the last 1, 5, and 15 minutes respectively.

Three constants defined in <sys/loadavg.h> can be used to extract data from the appropriate element of the *loadavg* array. These are LOADAVG_1MIN, LOADAVG_5MIN, and LOADAVG_15MIN.

Example: Printing the system's load averages

Program 8.13 retrieves the current load averages for the system, and then displays them.

sys_info/loadav.c

```
 1 #include <stdio.h>
 2 #include <sys/loadavg.h>
 3 #include "ssp.h"

 4 int main (void)
 5 {
 6     double load_av [3];

 7     if (getloadavg (load_av, 3) == -1)
 8         err_msg ("getloadavg failed");

 9     printf ("Load averaged over 1 minute: %.2f\n", load_av [LOADAVG_1MIN]);
10     printf ("Load averaged over 5 minutes: %.2f\n", load_av [LOADAVG_5MIN]);
11     printf ("Load averaged over 15 minutes: %.2f\n", load_av [LOADAVG_15MIN]);

12     return (0);
13 }
```

sys_info/loadav.c

Program 8.13 Displaying the system's load averages.

Running Program 8.13 on a lightly loaded Ultra 60 gave the following results:

```
$ ./loadav
Load averaged over 1 minute: 0.10
Load averaged over 5 minutes: 0.06
Load averaged over 15 minutes: 0.04
```

We can verify these results by examining the output of the uptime command:

```
$ uptime
  9:49pm  up 2 day(s),  3:21,  1 user,  load average: 0.10, 0.06, 0.04
```

8.10 Summary

In this chapter, we discussed the various ways a process can discover information about the machine it is running on. This information includes (but is not limited to): the system's hostname, the OS version, the system's CPU architecture, etc.

We then talked about system and per-process resource limits, showing how a process can portably determine the limits of its environment. We also discussed the resource control facility that was introduced with Solaris 9, and showed a couple of examples of how programs can use this facility.

Finally, we discussed how a process can determine its own resource usage by using the `getrusage` function and the newer `procfs` methods, and how to retrieve the system load averaged over 1, 5, and 15 minutes.

Exercises

8.1 Compare the contents of the `uname` section of Program 8.1 and the output from the command `uname -a`.

8.2 If we call `sethostname` to change the system's name, does this new name persist across a reboot?

8.3 Assuming a `CLK_TCK` value of 100 and that `times` returns the number of clock ticks since the system last booted, how long after the system boots will the value returned by `times` first wrap around in a 32-bit process?