11

SYSTEM LIMITS AND OPTIONS

Each UNIX implementation sets limits on various system features and resources, and provides—or chooses not to provide—options defined in various standards. Examples include the following:

- How many files can a process hold open at one time?
- Does the system support realtime signals?
- What is the largest value that can be stored in a variable of type *int*?
- How big an argument list can a program have?
- What is the maximum length of a pathname?

While we could hard-code assumed limits and options into an application, this reduces portability, since limits and options may vary:

- *Across UNIX implementations*: Although limits and options may be fixed on an individual implementation, they can vary from one UNIX implementation to another. The maximum value that can be stored in an *int* is an example of such a limit.
- *At run time on a particular implementation*: The kernel may have been reconfigured to change a limit, for example. Alternatively, the application may have been compiled on one system, but run on another system with different limits and options.

- *From one file system to another*: For example, traditional System V file systems allow a filename to be up to 14 bytes, while traditional BSD file systems and most native Linux file systems allow filenames of up to 255 bytes.

Since system limits and options affect what an application may do, a portable application needs ways of determining limit values and whether options are supported. The C programming language standards and SUSv3 provide two principal avenues for an application to obtain such information:

- Some limits and options can be determined at compile time. For example, the maximum value of an *int* is determined by the hardware architecture and compiler design choices. Such limits can be recorded in header files.
- Other limits and options may vary at run time. For such cases, SUSv3 defines three functions—*sysconf()*, *pathconf()*, and *fpathconf()*—that an application can call to check these implementation limits and options.

SUSv3 specifies a range of limits that a conforming implementation may enforce, as well as a set of options, each of which may or may not be provided by a particular system. We describe a few of these limits and options in this chapter, and describe others at relevant points in later chapters.

11.1 System Limits

For each limit that it specifies, SUSv3 requires that all implementations support a *minimum value* for the limit. In most cases, this minimum value is defined as a constant in `<limits.h>` with a name prefixed by the string `_POSIX_`, and (usually) containing the string `_MAX`; thus, the form of the name is `_POSIX_XXX_MAX`.

If an application restricts itself to the minimum values that SUSv3 requires for each limit, then it will be portable to all implementations of the standard. However, doing so prevents an application from taking advantage of implementations that provide higher limits. For this reason, it is usually preferable to determine the limit on a particular system using `<limits.h>`, *sysconf()*, or *pathconf()*.

> The use of the string `_MAX` in the limit names defined by SUSv3 can appear confusing, given their description as *minimum* values. The rationale for the names becomes clear when we consider that each of these constants defines an upper limit on some resource or feature, and the standards are saying that this upper limit must have a certain minimum value.
>
> In some cases, *maximum values* are provided for a limit, and these values have names containing the string `_MIN`. For these constants, the converse holds true: they represent a lower limit on some resource, and the standards are saying that, on a conforming implementation, this lower limit can be no greater than a certain value. For example, the `FLT_MIN` limit (`1E-37`) defines the largest value that an implementation may set for the smallest floating-point number that may be represented, and all conforming implementations will be able to represent floating-point numbers at least this small.

Each limit has a *name*, which corresponds to the *minimum value name* described above, but lacks the `_POSIX_` prefix. An implementation *may* define a constant with this name in `<limits.h>` to indicate the corresponding limit for this implementation.

If defined, this limit will always be at least the size of the minimum value described above (i.e., `XXX_MAX >= _POSIX_XXX_MAX`).

SUSv3 divides the limits that it specifies into three categories: *runtime invariant values*, *pathname variable values*, and *runtime increasable values*. In the following paragraphs, we describe these categories and provide some examples.

Runtime invariant values (possibly indeterminate)

A runtime invariant value is a limit whose value, if defined in `<limits.h>`, is fixed for the implementation. However, the value may be indeterminate (perhaps because it depends on available memory space), and hence omitted from `<limits.h>`. In this case (and even if the limit is also defined in `<limits.h>`), an application can use *sysconf()* to determine the value at run time.

The `MQ_PRIO_MAX` limit is an example of a runtime invariant value. As noted in Section 52.5.1, there is a limit on the priority for messages in POSIX message queues. SUSv3 defines the constant `_POSIX_MQ_PRIO_MAX`, with the value 32, as the minimum value that all conforming implementations must provide for this limit. This means that we can be sure that all conforming implementations will allow priorities from 0 up to at least 31 for message priorities. A UNIX implementation can set a higher limit than this, defining the constant `MQ_PRIO_MAX` in `<limits.h>` with the value of its limit. For example, on Linux, `MQ_PRIO_MAX` is defined with the value 32,768. This value can also be determined at run time using the following call:

```
lim = sysconf(_SC_MQ_PRIO_MAX);
```

Pathname variable values

Pathname variable values are limits that relate to pathnames (files, directories, terminals, and so on). Each limit may be constant for the implementation or may vary from one file system to another. In cases where a limit can vary depending on the pathname, an application can determine its value using *pathconf()* or *fpathconf()*.

The `NAME_MAX` limit is an example of a pathname variable value. This limit defines the maximum size for a filename on a particular file system. SUSv3 defines the constant `_POSIX_NAME_MAX`, with the value 14 (the old System V file-system limit), as the minimum value that an implementation must allow. An implementation may define `NAME_MAX` with a limit higher than this and/or make information about a specific file system available via a call of the following form:

```
lim = pathconf(directory_path, _PC_NAME_MAX)
```

The *directory_path* is a pathname for a directory on the file system of interest.

Runtime increasable values

A runtime increasable value is a limit that has a fixed minimum value for a particular implementation, and all systems running the implementation will provide at least this minimum value. However, a specific system may increase this limit at run time, and an application can find the actual value supported on the system using *sysconf()*.

An example of a runtime increasable value is `NGROUPS_MAX`, which defines the maximum number of simultaneous supplementary group IDs for a process

(Section 9.6). SUSv3 defines the corresponding minimum value, `_POSIX_NGROUPS_MAX`, with the value 8. At run time, an application can retrieve the limit using the call *sysconf(_SC_NGROUPS_MAX)*.

Summary of selected SUSv3 limits

Table 11-1 lists a few of the SUSv3-defined limits that are relevant to this book (other limits are introduced in later chapters).

Table 11-1: Selected SUSv3 limits

Name of limit (`<limits.h>`)	Min. value	*sysconf()/pathconf()* name (`<unistd.h>`)	Description
`ARG_MAX`	4096	`_SC_ARG_MAX`	Maximum bytes for arguments (*argv*) plus environment (*environ*) that can be supplied to an *exec()* (Sections 6.7 and 27.2.3)
none	none	`_SC_CLK_TCK`	Unit of measurement for *times()*
`LOGIN_NAME_MAX`	9	`_SC_LOGIN_NAME_MAX`	Maximum size of a login name (including terminating null byte)
`OPEN_MAX`	20	`_SC_OPEN_MAX`	Maximum number of file descriptors that a process can have open at one time, and one greater than maximum usable descriptor number (Section 36.2)
`NGROUPS_MAX`	8	`_SC_NGROUPS_MAX`	Maximum number of supplementary groups of which a process can be a member (Section 9.7.3)
none	1	`_SC_PAGESIZE`	Size of a virtual memory page (`_SC_PAGE_SIZE` is a synonym)
`RTSIG_MAX`	8	`_SC_RTSIG_MAX`	Maximum number of distinct realtime signals (Section 22.8)
`SIGQUEUE_MAX`	32	`_SC_SIGQUEUE_MAX`	Maximum number of queued realtime signals (Section 22.8)
`STREAM_MAX`	8	`_SC_STREAM_MAX`	Maximum number of *stdio* streams that can be open at one time
`NAME_MAX`	14	`_PC_NAME_MAX`	Maximum number of bytes in a filename, *excluding* terminating null byte
`PATH_MAX`	256	`_PC_PATH_MAX`	Maximum number of bytes in a pathname, *including* terminating null byte
`PIPE_BUF`	512	`_PC_PIPE_BUF`	Maximum number of bytes that can be written atomically to a pipe or FIFO (Section 44.1)

The first column of Table 11-1 gives the name of the limit, which may be defined as a constant in `<limits.h>` to indicate the limit for a particular implementation. The second column is the SUSv3-defined minimum for the limit (also defined in `<limits.h>`). In most cases, each of the minimum values is defined as a constant prefixed with the string `_POSIX_`. For example, the constant `_POSIX_RTSIG_MAX` (defined with the value 8) specifies the SUSv3-required minimum corresponding to the RTSIG_MAX implementation constant. The third column specifies the constant name that can be given at run time to *sysconf()* or *pathconf()* in order to retrieve the implementation limit. The constants beginning with _SC_ are for use with *sysconf()*; those beginning with _PC_ are for use with *pathconf()* and *fpathconf()*.

Note the following information supplementary to that shown in Table 11-1:

- The *getdtablesize()* function is an obsolete alternative for determining the process file descriptor limit (OPEN_MAX). This function was specified in SUSv2 (marked LEGACY), but was removed in SUSv3.
- The *getpagesize()* function is an obsolete alternative for determining the system page size (_SC_PAGESIZE). This function was specified in SUSv2 (marked LEGACY), but was removed in SUSv3.
- The constant FOPEN_MAX, defined in <stdio.h>, is synonymous with STREAM_MAX.
- NAME_MAX excludes the terminating null byte, while PATH_MAX includes it. This inconsistency repairs an earlier inconsistency in the POSIX.1 standard that left it unclear whether PATH_MAX included the terminating null byte. Defining PATH_MAX to include the terminator means that applications that allocated just PATH_MAX bytes for a pathname will still conform to the standard.

Determining limits and options from the shell: *getconf*

From the shell, we can use the *getconf* command to obtain the limits and options implemented by a particular UNIX implementation. The general form of this command is as follows:

```
$ getconf variable-name [ pathname ]
```

The *variable-name* identifies the limit of interest and is one of the SUSV3 standard limit names, such as ARG_MAX or NAME_MAX. Where the limit relates to a pathname, we must specify a pathname as the second argument to the command, as in the second of the following examples.

```
$ getconf ARG_MAX
131072
$ getconf NAME_MAX /boot
255
```

11.2 Retrieving System Limits (and Options) at Run Time

The *sysconf()* function allows an application to obtain the values of system limits at run time.

```
#include <unistd.h>

long sysconf(int name);
```

Returns value of limit specified by *name*, or –1 if limit is indeterminate or an error occurred

The *name* argument is one of the `_SC_*` constants defined in `<unistd.h>`, some of which are listed in Table 11-1. The value of the limit is returned as the function result.

If a limit can't be determined, *sysconf()* returns –1. It may also return –1 if an error occurred. (The only specified error is `EINVAL`, indicating that *name* is not valid.) To distinguish the case of an indeterminate limit from an error, we must set *errno* to 0 before the call; if the call returns –1 and *errno* is set after the call, then an error occurred.

> The limit values returned by *sysconf()* (as well as *pathconf()* and *fpathconf()*) are always (*long*) integers. In the rationale text for *sysconf()*, SUSv3 notes that strings were considered as possible return values, but were rejected because of the complexity of implementation and use.

Listing 11-1 demonstrates the use of *sysconf()* to display various system limits. Running this program on one Linux 2.6.31/x86-32 system yields the following:

```
$ ./t_sysconf
_SC_ARG_MAX:        2097152
_SC_LOGIN_NAME_MAX: 256
_SC_OPEN_MAX:       1024
_SC_NGROUPS_MAX:    65536
_SC_PAGESIZE:       4096
_SC_RTSIG_MAX:      32
```

Listing 11-1: Using *sysconf()*

syslim/t_sysconf.c

```
#include "tlpi_hdr.h"

static void             /* Print 'msg' plus sysconf() value for 'name' */
sysconfPrint(const char *msg, int name)
{
    long lim;

    errno = 0;
    lim = sysconf(name);
    if (lim != -1) {        /* Call succeeded, limit determinate */
        printf("%s %ld\n", msg, lim);
    } else {
        if (errno == 0)     /* Call succeeded, limit indeterminate */
            printf("%s (indeterminate)\n", msg);
        else                /* Call failed */
            errExit("sysconf %s", msg);
    }
}

int
main(int argc, char *argv[])
{
    sysconfPrint("_SC_ARG_MAX:        ", _SC_ARG_MAX);
    sysconfPrint("_SC_LOGIN_NAME_MAX: ", _SC_LOGIN_NAME_MAX);
    sysconfPrint("_SC_OPEN_MAX:       ", _SC_OPEN_MAX);
    sysconfPrint("_SC_NGROUPS_MAX:    ", _SC_NGROUPS_MAX);
    sysconfPrint("_SC_PAGESIZE:       ", _SC_PAGESIZE);
    sysconfPrint("_SC_RTSIG_MAX:      ", _SC_RTSIG_MAX);
    exit(EXIT_SUCCESS);
}
```

syslim/t_sysconf.c

SUSv3 requires that the value returned by *sysconf()* for a particular limit be constant for the lifetime of the calling process. For example, we can assume that the value returned for `_SC_PAGESIZE` won't change while a process is running.

> On Linux, there are some (sensible) exceptions to the statement that limit values are constant for the life of a process. A process can use *setrlimit()* (Section 36.2) to change various process resource limits that affect limit values reported by *sysconf()*: `RLIMIT_NOFILE`, which determines the number of files the process may open (`_SC_OPEN_MAX`); `RLIMIT_NPROC` (a resource limit not actually specified in SUSv3), which is the per-user limit on the number of processes that may created by this process (`_SC_CHILD_MAX`); and `RLIMIT_STACK`, which, since Linux 2.6.23, determines the limit on the space allowed for the process's command-line arguments and environment (`_SC_ARG_MAX`; see the *execve(2)* manual page for details).

11.3 Retrieving File-Related Limits (and Options) at Run Time

The *pathconf()* and *fpathconf()* functions allow an application to obtain the values of file-related limits at run time.

```
#include <unistd.h>

long pathconf(const char *pathname, int name);
long fpathconf(int fd, int name);
```

Both return value of limit specified by *name*,
or –1 if limit is indeterminate or an error occurred

The only difference between *pathconf()* and *fpathconf()* is the manner in which a file or directory is specified. For *pathconf()*, specification is by pathname; for *fpathconf()*, specification is via a (previously opened) file descriptor.

The *name* argument is one of the `_PC_*` constants defined in `<unistd.h>`, some of which are listed in Table 11-1. Table 11-2 provides some further details about the `_PC_*` constants that were shown in Table 11-1.

The value of the limit is returned as the function result. We can distinguish between an indeterminate return and an error return in the same manner as for *sysconf()*.

Unlike *sysconf()*, SUSv3 doesn't require that the values returned by *pathconf()* and *fpathconf()* remain constant over the lifetime of a process, since, for example, a file system may be dismounted and remounted with different characteristics while a process is running.

Table 11-2: Details of selected *pathconf()* _PC_* names

Constant	Notes
_PC_NAME_MAX	For a directory, this yields a value for files in the directory. Behavior for other file types is unspecified.
_PC_PATH_MAX	For a directory, this yields the maximum length for a relative pathname from this directory. Behavior for other file types is unspecified.
_PC_PIPE_BUF	For a FIFO or a pipe, this yields a value that applies to the referenced file. For a directory, the value applies to a FIFO created in that directory. Behavior for other file types is unspecified.

Listing 11-2 shows the use of *fpathconf()* to retrieve various limits for the file referred to by its standard input. When we run this program specifying standard input as a directory on an *ext2* file system, we see the following:

```
$ ./t_fpathconf < .
_PC_NAME_MAX:  255
_PC_PATH_MAX:  4096
_PC_PIPE_BUF:  4096
```

Listing 11-2: Using *fpathconf()*

syslim/t_fpathconf.c

```
#include "tlpi_hdr.h"

static void             /* Print 'msg' plus value of fpathconf(fd, name) */
fpathconfPrint(const char *msg, int fd, int name)
{
    long lim;

    errno = 0;
    lim = fpathconf(fd, name);
    if (lim != -1) {        /* Call succeeded, limit determinate */
        printf("%s %ld\n", msg, lim);
    } else {
        if (errno == 0)     /* Call succeeded, limit indeterminate */
            printf("%s (indeterminate)\n", msg);
        else                /* Call failed */
            errExit("fpathconf %s", msg);
    }
}

int
main(int argc, char *argv[])
{
    fpathconfPrint("_PC_NAME_MAX: ", STDIN_FILENO, _PC_NAME_MAX);
    fpathconfPrint("_PC_PATH_MAX: ", STDIN_FILENO, _PC_PATH_MAX);
    fpathconfPrint("_PC_PIPE_BUF: ", STDIN_FILENO, _PC_PIPE_BUF);
    exit(EXIT_SUCCESS);
}
```

syslim/t_fpathconf.c

11.4 Indeterminate Limits

On occasion, we may find that some system limit is not defined by an implementation limit constant (e.g., PATH_MAX), and that *sysconf()* or *pathconf()* informs us that the limit (e.g., _PC_PATH_MAX) is indeterminate. In this case, we can employ one of the following strategies:

- When writing an application to be portable across multiple UNIX implementations, we could elect to use the minimum limit value specified by SUSv3. These are the constants with names of the form _POSIX_*_MAX, described in Section 11.1. Sometimes, this approach may not be viable because the limit is unrealistically low, as in the cases of _POSIX_PATH_MAX and _POSIX_OPEN_MAX.
- In some cases, a practical solution may be to ignore the checking of limits, and instead perform the relevant system or library function call. (Similar arguments can also apply with respect to some of the SUSv3 options described in Section 11.5.) If the call fails and *errno* indicates that the error occurred because some system limit was exceeded, then we can try again, modifying the application behavior as necessary. For example, most UNIX implementations impose a limit on the number of realtime signals that can be queued to a process. Once this limit is reached, attempts to send further signals (using *sigqueue()*) fail with the error EAGAIN. In this case, the sending process could simply retry, perhaps after some delay interval. Similarly, attempting to open a file whose name is too long yields the error ENAMETOOLONG, and an application could deal with this by trying again with a shorter name.
- We can write our own program or function to either deduce or estimate the limit. In each case, the relevant *sysconf()* or *pathconf()* call is made, and if this limit is indeterminate, the function returns a "good guess" value. While not perfect, such a solution is often viable in practice.
- We can employ a tool such as GNU *Autoconf*, an extensible tool that can determine the existence and settings of various system features and limits. The Autoconf program produces header files based on the information it determines, and these files can then be included in C programs. Further information about Autoconf can be found at *http://www.gnu.org/software/autoconf/*.

11.5 System Options

As well as specifying limits for various system resources, SUSv3 also specifies various options that a UNIX implementation may support. These include support for features such as realtime signals, POSIX shared memory, job control, and POSIX threads. With a few exceptions, implementations are not required to support these options. Instead, SUSv3 allows an implementation to advise—at both compile time and run time—whether it supports a particular feature.

An implementation can advertise support for a particular SUSv3 option at compile time by defining a corresponding constant in <unistd.h>. Each such constant starts with a prefix that indicates the standard from which it originates (e.g., _POSIX_ or _XOPEN_).

Each option constant, if defined, has one of the following values:

- A value of –1 means that *the option is not supported*. In this case, header files, data types, and function interfaces associated with the option need not be defined by the implementation. We may need to handle this possibility by conditional compilation using `#if` preprocessor directives.
- A value of 0 means that *the option may be supported*. An application must check at run time to see whether the option is supported.
- A value greater than 0 means that *the option is supported*. All header files, data types, and function interfaces associated with this option are defined and behave as specified. In many cases, SUSv3 requires that this positive value be `200112L`, a constant corresponding to the year and month number in which SUSv3 was approved as a standard. (The analogous value for SUSv4 is `200809L`.)

Where a constant is defined with the value 0, an application can use the *sysconf()* and *pathconf()* (or *fpathconf()*) functions to check at run time whether the option is supported. The *name* arguments passed to these functions generally have the same form as the corresponding compile-time constants, but with the prefix replaced by `_SC_` or `_PC_`. The implementation must provide at least the header files, constants, and function interfaces necessary to perform the run-time check.

> SUSv3 is unclear on whether an undefined option constant has the same meaning as defining the constant with the value 0 ("the option may be supported") or with the value –1 ("the option is not supported"). The standards committee subsequently resolved that this case should mean the same as defining the constant with the value –1, and SUSv4 states this explicitly.

Table 11-3 lists some of the options specified in SUSv3. The first column of the table gives the name of the associated compile-time constant for the option (defined in `<unistd.h>`), as well as the corresponding *sysconf()* (`_SC_*`) or *pathconf()* (`_PC_*`) *name* argument. Note the following points regarding specific options:

- Certain options are required by SUSv3; that is, the compile-time constant always evaluates to a value greater than 0. Historically, these options were truly optional, but nowadays they are not. These options are marked with the character + in the *Notes* column. (In SUSv4, a range of options that were optional in SUSv3 become mandatory.)

  > Although such options are required by SUSv3, some UNIX systems may nevertheless be installed in a nonconforming configuration. Thus, for portable applications, it may be worth checking whether an option that affects the application is supported, regardless of whether the standard requires that option.

- For certain options, the compile-time constant must have a value other than –1. In other words, either the option must be supported or support at run time must be checkable. These options are marked with the character * in the *Notes* column.

Table 11-3: Selected SUSv3 options

Option (constant) name (***sysconf() / pathconf()*** **name**)	**Description**	**Notes**
`_POSIX_ASYNCHRONOUS_IO` (`_SC_ASYNCHRONOUS_IO`)	*Asynchronous I/O*	
`_POSIX_CHOWN_RESTRICTED` (`_PC_CHOWN_RESTRICTED`)	Only privileged processes can use *chown()* and *fchown()* to change the user ID and group ID of a file to arbitrary values (Section 15.3.2)	*
`_POSIX_JOB_CONTROL` (`_SC_JOB_CONTROL`)	*Job Control* (Section 34.7)	+
`_POSIX_MESSAGE_PASSING` (`_SC_MESSAGE_PASSING`)	*POSIX Message Queues* (Chapter 52)	
`_POSIX_PRIORITY_SCHEDULING` (`_SC_PRIORITY_SCHEDULING`)	*Process Scheduling* (Section 35.3)	
`_POSIX_REALTIME_SIGNALS` (`_SC_REALTIME_SIGNALS`)	*Realtime Signals Extension* (Section 22.8)	
`_POSIX_SAVED_IDS` (none)	Processes have saved set-user-IDs and saved set-group-IDs (Section 9.4)	+
`_POSIX_SEMAPHORES` (`_SC_SEMAPHORES`)	*POSIX Semaphores* (Chapter 53)	
`_POSIX_SHARED_MEMORY_OBJECTS` (`_SC_SHARED_MEMORY_OBJECTS`)	*POSIX Shared Memory Objects* (Chapter 54)	
`_POSIX_THREADS` (`_SC_THREADS`)	*POSIX Threads*	
`_XOPEN_UNIX` (`_SC_XOPEN_UNIX`)	The XSI extension is supported (Section 1.3.4)	

11.6 Summary

SUSv3 specifies limits that an implementation may enforce and system options that an implementation may support.

Often, it is desirable not to hard-code assumptions about system limits and options into a program, since these may vary across implementations and also on a single implementation, either at run time or across file systems. Therefore, SUSv3 specifies methods by which an implementation can advertise the limits and options it supports. For most limits, SUSv3 specifies a minimum value that all implementations must support. Additionally, each implementation can advertise its implementation-specific limits and options at compile time (via a constant definition in `<limits.h>` or `<unistd.h>`) and/or run time (via a call to *sysconf()*, *pathconf()*, or *fpathconf()*). These techniques may similarly be used to find out which SUSv3 options an implementation supports. In some cases, it may not be possible to determine a particular limit using either of these methods. For such indeterminate limits, we must resort to ad hoc techniques to determine the limit to which an application should adhere.

Further information

Chapter 2 of [Stevens & Rago, 2005] and Chapter 2 of [Gallmeister, 1995] cover similar ground to this chapter. [Lewine, 1991] also provides much useful (although now slightly outdated) background. Some information about POSIX options with notes on *glibc* and Linux details can be found at *http://people.redhat.com/drepper/posix-option-groups.html*. The following Linux manual pages are also relevant: *sysconf(3)*, *pathconf(3)*, *feature_test_macros(7)*, *posixoptions(7)*, and *standards(7)*.

The best sources of information (though sometimes difficult reading) are the relevant parts of SUSv3, particularly Chapter 2 from the Base Definitions (XBD), and the specifications for *<unistd.h>*, *<limits.h>*, *sysconf()*, and *fpathconf()*. [Josey, 2004] provides guidance on the use of SUSv3.

11.7 Exercises

11-1. Try running the program in Listing 11-1 on other UNIX implementations if you have access to them.

11-2. Try running the program in Listing 11-2 on other file systems.