

# POSIX and Linux Application Compatibility Design Rules

by C. Douglass Locke  
www.douglocke.com  
4/11/2005  
Draft Version 0.92

## 1. Introduction

The maturity of the open source Linux<sup>®</sup> operating system has generated a great deal of interest in application development for this inexpensive, highly flexible, powerful infrastructure for a wide variety of applications. Linux provides a set of capabilities that are modeled directly after the UNIX<sup>®</sup> operating system, with interfaces that are highly compliant to the IEEE's POSIX<sup>®</sup> 1003.1 application programming interface (API) standard.

It is, however, a matter of much concern that while Linux provides a large set of API's that are essentially the same as POSIX, there does exist a set of differences that can inhibit the production of applications that are compatible with both operating system environments.

Although the fact that Linux is not POSIX-compliant is well known, it is not as well understood that, from a practical standpoint, Linux exhibits a very high degree of POSIX compliance. For this reason, it has often been observed that arbitrary POSIX compliant applications can most often be ported to Linux with little or no change. This document aims to enhance this situation by permitting the application designer to consciously design around or otherwise handle the differences.

In the remainder of this document, Section 2 describes the scope of this document, identifying the approach to be taken. Section 3 covers the system interfaces (API's) that an application will use to obtain all required operating system functionality, while Section 4 covers the issues involved in using the POSIX shells and utilities. Section 5 discusses a few internationalization issues, Section 6 discusses the specific needs of real-time systems, and Section 7 provides a brief summary and conclusion.

It should be noted that the term "Linux" actually refers only to the operating system kernel supplied with a Linux distribution. Such a distribution must, of necessity, include a large number of other components, including libraries, tools, documentation, etc. These additional components generally come from the Free Software Foundation's GNU project, so the correct term for the complete set of API's is "GNU/Linux". This paper uses the term "Linux" to refer to this complete set, as is commonly done in most contexts, even though it is technically inaccurate to do so.

Draft – Soliciting Comments

This work was performed under contract number N00178-99-D-2034

1

**“Statement A: Document approved for Public Release: Distribution is unlimited”**

Trademark notes: Linux is a registered trademark of Linus Torvalds. LSB is a registered trademark of the Free Standards Group. POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers. UNIX is a registered trademark of The Open Group.

## **2. Scope**

This document describes the technical issues involved in creating applications that will be highly portable between POSIX and Linux environments. There are, conceptually, two approaches one can take to this:

1. Build a POSIX compliant application that avoids or works around Application Programming Interfaces (API's) that are in conflict with Linux.
2. Build a Linux application that avoids or works around API's that are in conflict with POSIX.

This document takes the first approach: it describes design requirements for building a fully POSIX compliant application that avoids or designs around API's that are in conflict with Linux. Taking the second approach would be more difficult, because Linux includes a number of new interfaces, as well as many BSD and other Unix interfaces that are not part of the POSIX standard. Thus, designing an application that will be readily portable between POSIX and Linux will be somewhat simpler starting with the POSIX API than starting with the GNU/Linux API.

In addition, this document does not specifically deal with some important operating system interfaces that are common to both UNIX and Linux implementations but are not standardized in POSIX. For example, the `ioctl()` function is not considered in this document because its interface has many varying, unique capabilities that differ among most UNIX and Linux distributions. Such interfaces, while they are necessary for many, if not most applications, shall be localized by the application to the greatest extent possible so that they can be readily modified as needed when the application is to be ported to new platforms.

A number of the details for this document are derived from an Open Group draft report[3] dated 1 April 2005 describing the conflicts between Linux and POSIX. That report was based on Linux Standard Base Core Specification Work in Progress 3.0Preview1 (LSB<sup>®</sup>) dated March 31, 2005[2], and the ISO/IEC 9945:2003 edition[1] dated 15th August 2003, with the addition of the first Corrigenda ISO/IEC 9945:2003/Cor.1:2004 (published 15 Sep 2004)[4]. Further information is derived from the "man pages" for both operating systems and extensive experience with projects using both operating system environments.

## **3. System Interfaces**

Designing software to run easily on both POSIX compliant operating systems and Linux is not really a particularly difficult thing to do. To start, the application shall be created using any mandatory POSIX API system calls, which also includes the

library functions in the ANSI C [5] standard. In general, such a POSIX compliant application will very likely be ready to be recompiled and run on Linux with few problems.

If optional POSIX API's are needed, such as the execution scheduling functions needed by real-time applications, the designer will need to check the specific OS vendor for which options are available for the chosen OS, whether the OS is POSIX compliant or a Linux distribution. For example, general purpose POSIX-compliant OS's such as are provided by many major server vendors may not provide the real-time options, but real-time POSIX-compliant OS's generally provided specifically for real-time systems by independent software vendors are likely to provide most or all such options. Similarly for Linux distributions, general purpose distributions such as those from major Linux vendors may not provide many of the real-time options, but others from vendors specifically targeting real-time and embedded systems are likely to provide most or all such options. Some implementations may support profiles such as the POSIX 1003.13 PSE54 profile that define a specific set of POSIX options that must be supported.

As the application design progresses, each relevant section of this document should be checked to ensure that the few POSIX API's known to create portability issues with Linux are avoided or designed around. The purpose of this section is to identify these problematic system interfaces so that the port can be performed trouble-free.

For interfaces with known portability issues, this document generally assumes that the application designer will avoid them by using alternative API's. However, for some portability issues, this document mentions the use of an OS version test.

An OS version test consists of compile-time (e.g., `#define / #ifdef / #else / #endif`) options permitting alternative code snippets to be compiled that use the appropriate API for each chosen Linux or POSIX target. This approach to portability is highly flexible, but it results in the design of multiple versions of the application code, each of which must be thoroughly tested – the programmer may easily have created a bug in one version that is not present in the other version(s) at each compile-time switched point. In practice, this approach often leads to the need to check multiple conditions, and even nested conditional code snippets; this can make verification of application correctness very difficult, especially over an extended application lifetime when the original implementers may no longer be part of the implementation team. If OS version tests are used, it is strongly recommended that they be collected together into localized code sections or libraries where the effect of platform changes can be handled in one place.

This section is divided into several major functional areas to make it easy to consider each application service as the design progresses:

1. File Management
2. Process Management
3. Signals

4. Threads
5. Synchronization
6. Clocks and Timers
7. Other System Interfaces

As we deal with each functional area, note that there will be very few functions that cannot be used due to the requirement for POSIX / Linux compatibility, but there will be a need to use some functions very carefully.

Application code designs for operating system portability will generally take one of 3 forms when a portability issue is encountered:

1. Checking an alternative error code or checking more than one error code for some system interface functions that return different error codes among POSIX and Linux systems.
2. Avoiding certain options or limiting the code to certain options where there are inconsistencies among different target OS's.
3. Using the compile-time flags to create an OS version test as described above to select slightly different options or API's for the different targets, ensuring that the application can successfully compile and execute in each environment.

### 3.1 File Management

The Linux file management API's are almost entirely compatible with the corresponding API's from POSIX, so the application designer can use virtually all of the same system interfaces in either system. There are only a few things to watch out for:

- **fcntl()** – under the LSB, if the file system's default behavior matches the **O\_LARGEFILE** behavior, the file descriptor flags returned by **fcntl()** when specified with the command **F\_GETFL** are allowed to include the **O\_LARGEFILE** flag, even if the application had not set it previously. POSIX permits this flag to be set only by overt action of the application. The portable application shall therefore tolerate the possibility that this flag may appear “magically,” even if the flag was never set by the application.
- Linux permits several file system functions to return either **ENODEV** or **ENXIO** for a non-existent device, while POSIX requires an **ENXIO** return in this case. The portable application shall tolerate either return value from calls to **fopen()**, **freopen()**, **open()**, and **creat()**.
- If a call is made to **unlink()** with a directory pathname, a POSIX system will return **EPERM** if it determines that the directory cannot be unlinked with this function. A Linux system may return **EISDIR** in this case. The portable application shall tolerate either return value for this case.
- The **ioctl()** function is defined independently in Linux from POSIX. However, the **ioctl()** function is also used inconsistently among many Unix implementations, so it should generally be avoided when possible. If **ioctl()** is needed, the portable application will probably find it necessary to use an OS version test for each target operating system to portably use **ioctl()** to control devices.

- For **fscanf()**, and its related functions (**fwscanf()**, **scanf()**, **vfscanf()**, **vwscanf()**, **vscanf()**, **vsscanf()**, **vswscanf()**), LSB defines a conversion specifier **%a** that defines a conversion exactly the same as that for **%s**, but that allocates (using **malloc()**) the required memory for the converted string and assigns its address to the next variable, treating it as a **char\*** variable. In contrast, ANSI C (included in the POSIX specification) defines the **%a** conversion specifier for these functions to convert the corresponding input field as an optionally signed floating point number. The portable application shall not use the **%a** conversion specifier for an input character string, but shall instead use the **%s** with a preallocated string of sufficient size, and shall use the **%f** conversion specifier to read optionally signed floating point numbers.
- When symbolic links are referenced, the **link()** function has inconsistent semantics between Linux and some POSIX compliant implementations, and even among multiple POSIX implementations. While symbolic links are generally dereferenced when their paths are given to normal file operations (e.g., **open()**, **close()**, **read()**, **write()**) by following the link and acting directly on the target file, some implementations may provide the capability for the **link()** function to reference the symbolic link special file itself. The portable application shall not use the **link()** function to operate on symbolic links.

### 3.2 Process Management

Process management in Linux is nearly identical to POSIX. Some optional POSIX interfaces (e.g., **spawn()**) are not included, but all the mandatory POSIX process functionality is there, including **fork()**, **exec()**, **wait()**, etc. There are, however a few issues to be considered:

- Linux does not require implementation of the **WCONTINUED** functionality of the **waitpid()** call. This also implies that the dependent **WIFCONTINUED** macro is not required to be available. The portable application shall not use the **WCONTINUED** functionality.
- The POSIX **waitid()** function is not required by the LSB. The portable application shall not use the **waitid()** function.

### 3.3 Signals

Signals and signal handling are almost completely portable between a POSIX operating system and a Linux operating system for processes. Signals will also work correctly for threads as long as the Linux distribution uses the **nptl** library. See Section 3.4 for details of Linux vs. POSIX thread handling.

The only signal portability issue is found in the **kill()** function when the **pid** value is set to **-1**. In this case, POSIX specifies that the signal is to be sent to all processes (excluding an unspecified set of system processes). The **kill()** function in Linux with the **pid** value set to **-1** is the same except that the signal will not be sent to the calling process. The portable application shall allow for the signal to be sent to the calling process, but it shall not depend on this for correct application behavior.

Draft – Soliciting Comments

### 3.4 Threads

POSIX threads (called “pthreads”) were implemented in Linux prior to version 2.6 using a library called **libpthread**. This library used Linux’s underlying concurrency mechanism called “tasks” to implement both processes and threads. However, the semantics of Linux tasks had a number of significant differences from those of POSIX threads; these differences had no significant effect on its support of POSIX processes, but there were a number of compatibility issues with the support of Linux threads. Threads implemented by **libpthread** in Linux were called “LinuxThreads.”

Beginning with the Linux 2.6 release, a new version of **libpthread** was used called **nptl** (i.e., “Native POSIX Threading Library”). This library makes a major change in Linux thread functions, and is highly POSIX pthread compatible. Linux 2.6 uses the **nptl** library by default, but Linux 2.6 distributions can also be configured to support the original **libpthread** rather than the **nptl** by setting the shell environment variable **LD\_ASSUME\_KERNEL**.

Note that at present, most Linux distributions, whether they are using **nptl** or not, do not support the options **\_POSIX\_THREAD\_PRIO\_INHERIT** and **\_POSIX\_THREAD\_PRIO\_PROTECT** for real-time mutexes. This means that real-time applications will not be able to obtain predictable response when critical sections are protected by mutexes. However, a set of real-time patches have been created by the Linux community and are now available for Linux 2.6. These patches implement these POSIX options and provide other critical improvements to the preemptibility of the Linux kernel specifically for real-time systems. For additional information, see <http://developer.osdl.org/dev/robustmutexes/>. A few Linux distributions are now available that incorporate these patches fully tested in the kernel.

The portable application using pthreads shall ensure that it is using **nptl**, and shall not expect real-time mutex behavior unless the kernel has been patched. Note that there is no effective work-around for real-time applications if the real-time patches are not installed.

### 3.5 Synchronization

The POSIX semaphore calls are portable between POSIX and Linux operating systems. The mutex and condition variable functionality is defined for POSIX threads; for portability information, see Section 3.4. It should be noted that the POSIX semaphore are not suitable for mutual exclusion in real-time applications because there is no way to avoid unbounded priority inversion with the POSIX semaphore.

### 3.6 Clocks and Timers

The POSIX clocks and timers calls are portable between POSIX and Linux, but it is highly likely that real-time application designers will find the standard Linux timer resolution to be inadequate. Linux measures timer values in “jiffies” which are set to 10 milliseconds by default. This means that Linux timers, used for example by **nanosleep()**, are settable only in 10 millisecond increments. In addition, the timeout

value actually used is set to the next higher jiffy to ensure that the timeout will never be less than the set value. Thus for example, a call to **nanosleep()** with a time value of 5 milliseconds will actually awaken 20 milliseconds later. Note that Linux also has a potentially severe anomaly; a timer value of 2 milliseconds or less is implemented with a “busy wait” which uses 100% of the CPU until the requested sleep time has elapsed.

There is an open source project (see <http://sourceforge.net/projects/high-res-timers>) creating a Linux patch to provide high-resolution timers. This will require recompiling the Linux kernel.

Some “real-time Linux” distributions simply set the “jiffy” value to 1 millisecond or smaller to provide “high resolution” timers. This greatly increases the OS overhead. Other “real-time Linux” distributions use the patch described above that provides much higher resolution without significantly increasing the overhead. The designer of a portable application shall check the timer resolution and its implementation details with the Linux distribution vendor to determine whether the application can obtain satisfactory timer resolution with acceptable overhead.

### 3.7 Other System Interfaces

There is a group of miscellaneous system interfaces in which POSIX and Linux have differing functionality. This section discusses each of them.

1. The **getopt()** function call iteratively parses the argument string passed to the application’s **main()** function. Linux permits several differences in its interface under certain circumstances relative to a POSIX system. The portable application shall use **getopt()** only to make a single pass through the argument string, with a single set of options, and avoid using the **-W** option which is handled differently between Linux and POSIX. An environment variable **POSIXLY\_CORRECT** is defined for Linux that forces **getopt()** to produce POSIX behavior, but this is likely to cause problems in other Linux utilities that are not expecting this behavior, and shall not be used in portable applications.
2. The **strerror\_r()** function call (created only for use in threaded applications) in POSIX returns an integer. Linux uses a GNU version of this call that returns a **char\***. This completely breaks the purpose of this function which was to create a thread-safe version of **strerror()** that wouldn’t return a pointer to a static string. The portable application shall not use this function. Its functionality shall be provided by using **strerror()** during application startup-up before multiple threads are started to retrieve and store all of the error strings that will be needed; the application shall then use these stored strings during all normal (threaded) execution.
3. POSIX requires that the constant **\_XOPEN\_VERSION** be defined with a value of 600 for conforming implementations. Linux defines this value as 500. The portable application shall therefore not depend on **\_XOPEN\_VERSION** having a particular one of these values.
4. POSIX requires that each of the values **ENOTSUP** and **EOPNOTSUP** be defined to have unique values. Linux sets these two error codes to the same

value. The portable application shall therefore not depend on being able to distinguish between these two values.

5. The function **strptime()** is generally compatible between POSIX and Linux. The only issue that might be encountered is in the case where strings with leading zeros are presented to **strptime()**. The portable application shall ensure that “excess leading zeros” are removed before calling **strptime()** to avoid a potential anomalous error return. Excess leading zeros means leading zeros that result in more digits than is logically possible for a field. For example, ‘1’ and ‘01’ are ok for months, but ‘001’ is not.

#### 4. Shells and Utilities

The POSIX shells and utilities provide a very large amount of basic functionality that can greatly improve an application’s cost and significantly reduce its risk. These utilities can be used either within the application code, for example, by using the **system()** call (or **fork()** and **exec()**), or in a shell script executing cooperatively with the application. As with the system calls, Linux supports almost all of the POSIX shells and utilities, but the application designer must consider a few issues with respect to some POSIX-defined functionality. This section identifies the known issues.

- The utility **ar** is deprecated in the LSB. Its functionality is generally provided by **tar** and **cpio** in Linux systems, and by **pax** in POSIX systems. POSIX identifies **tar** and **cpio** as legacy utilities that are optional (although they are very commonly available in most POSIX implementations), while it makes **pax**, which combines the functionality of **tar** and **cpio**, mandatory. The LSB makes **tar** and **cpio** mandatory, but makes **pax** optional. If a specific Linux distribution does not include **pax**, a Linux **pax** utility can be obtained from an open source project on sourceforge.org. To maintain strict compatibility, the portable application shall make their use dependent on an OS version test, or avoid them. However, a high degree of portability can be achieved by using **pax**, **tar** and **cpio** because of their widespread availability in POSIX compatible implementations and Linux distributions.
- The POSIX definition for the **at -r** functionality (removal of prior **at** jobs) is provided in Linux using the **-d** option. The **-r** and the **-t** options may not be supported on Linux distributions. The portable application shall use the timespec operand rather than the **-t** functionality, and shall either avoid using the **-r** and **-d** functionality, or shall make their use dependent on an OS version test.
- In the Linux **at**, **batch**, and **crontab** utilities, the files **at.allow** and **at.deny** reside in **/etc** rather than in **/usr/lib/cron** as specified in POSIX. The portable application shall allow these files to be in either location.
- Certain aspects of internationalized regular expressions may differ from POSIX in the Linux **awk** utility. The portable application shall not use internationalized regular expressions unless they are specifically required for the application’s intended user base. If they cannot be avoided, see Section 5 for further information.

- The Linux **bc** utility contains extensions beyond POSIX functionality. The portable application shall avoid using this functionality. An application could use the Linux **-s** and **-w** options to ensure that only POSIX functionality is used, but these options may not be recognized in a POSIX implementation, so they do not increase portability. The portable application shall use only the common syntax and semantics between the Linux and POSIX **bc** implementations, which includes most common uses of **bc**.
- The LSB does not require Linux distribution **chgrp** and **chown** utilities to support the **-L**, **-H**, or **-P** options that determine some of the behavior of the **-R** option on symbolic links. The portable application shall not be dependent on the **-R** option to perform the correct actions on certain symbolic links specified in the command line or found in recursively following the directory hierarchy.
- The **cut** utility has an option used with multi-byte character strings to ensure that multi-byte characters are not split. POSIX defines specific actions for the **-n** option on byte selections specified by the **-b** option; the **-n** option in Linux has the same purpose, but the specification is not as precise. The portable application shall take care when using the **-n** option with multi-byte character strings that the byte selections from the **-b** option do not result in incorrect (split) character selections.
- Under POSIX, the **df** and **du** utilities return disk space in 512-byte blocks unless the **-k** option is used; in this case the space is reported in 1024-byte blocks. Under Linux, the **df** and **du** utilities return disk space in 1024-byte blocks by default as well as when the **-k** option is used. The portable application shall always use the **-k** option and shall accept the result only in terms of 1024-byte blocks. In addition, with **df** there is a conflict between Linux and POSIX on the use of the **-t** option. The portable application shall not use **df**'s **-t** option.
- The **echo** utility in a Linux distribution may be derived from any of several sources, some of which may ignore some options or support non-POSIX options. The portable application shall use **echo** with no options, and shall ensure that the first argument doesn't begin with a hyphen or contain a backslash. If the portable application requires the use of arguments beginning with a hyphen or containing a backslash, it shall use the **printf** utility instead.
- The Linux **file** utility is not required to support the **-M**, **-h**, **-d**, and the **-i** options. The portable application shall not use these options.
- The **find** utility in a Linux distribution may be derived from any of several sources, some of which may ignore some options or support non-POSIX options. In particular, **find** may not support the **-H** and **-L** option required by POSIX. The portable application shall not use the **-H** or **-L** options with **find**. Additionally, the Linux **find** utility has some minor differences between POSIX and Linux with respect to Internationalization and Pattern Matching. The portable application shall not use Internationalization and Pattern Matching unless they are specifically required for the application's intended user base. If they cannot be avoided, see Section 5 for further information.
- The Linux **grep** utility has some minor differences between POSIX and Linux with respect to Internationalization and Pattern Matching. The portable application shall not use Internationalization and Pattern Matching unless they are

specifically required for the application's intended user base. If they cannot be avoided, see Section 5 for further information.

- The LSB does not require the Linux **fuser** utility to support the `-c` and the `-f` options. The portable application shall not use the `-c` or `-f` options.
- The Linux **ipcrm** utility is POSIX compliant when used with any of its non-deprecated options `-q`, `-Q`, `-s`, `-S`, `-m`, and/or `-M`. The portable application shall always use one or more of these options, and shall not use any other options.
- The Linux **ipcs** utility is similar to the POSIX **ipcs** utility, but its option definition is not described in similar precision to the POSIX version, so there may be differences. For example, the `-a` option is described differently between POSIX and Linux. In addition, the Linux version does not support the `-b` and `-o` options, but does support a `-l` and `-u` option not defined by POSIX. The portable application shall not use the `-a`, `-b`, `-o`, `-l`, and `-u` options.
- The Linux **ls** utility using the `-l` option will substitute major and minor device numbers for the file size for character special files or block special files. In addition, the Linux **ls** utility using the `-p` option may display additional characters for some file types beyond those defined by POSIX. The portable application shall not depend on the `-l` option of **ls** for file size information for character special and block special files, and shall accept some additional characters for some file types when the `-p` option is used.
- The POSIX **more** utility respects the `LINES` and `COLUMNS` environment variables, while the Linux version does not. In addition, the Linux version supports a number of different interactive commands from the terminal operator. Also, the Linux version has different behavior for options `-num`, `-e`, `-I`, `-n`, `-p`, and `-t`. The portable application shall use **more** without these options, and shall accept its default `LINES` and `COLUMNS` settings.
- The POSIX **newgrp** utility is fully supported by Linux except for the `-l` option which is used to start a shell with the user's login settings. The portable application shall not depend on the `-l` option.
- The Linux **od** utility is POSIX compliant, but offers some extensions including the `-w` (same as `-width`) option to control output line width, and the `-traditional` option that causes **od** to accept a set of pre-POSIX and XSI options. The portable application shall use only the POSIX defined options.
- The POSIX **renice** utility is supported by Linux with the exception that the `-n` option has unspecified behavior in Linux distributions. Unfortunately, this option is the basic mechanism POSIX provides to increment (or decrement) the nice value, so this means that the **renice** utility is difficult to use portably. The portable application shall use **renice** without the `-n` option, shall make its use dependent on an OS version test, or shall avoid the **renice** utility.
- Certain aspects of internationalized regular expressions may differ from POSIX in the Linux **sed** utility. The portable application shall not use internationalized regular expressions unless they are specifically required for the application's intended user base. If they cannot be avoided, see Section 5 for further information.
- The **xargs** utility has compatible functionality, but uses lower case for the POSIX options `-E`, `-I`, and `-L`. This means that these utility options cannot be used

Draft – Soliciting Comments

portably. The portable application shall avoid the use of `-E`, `-e`, `-I`, `-i`, `-L`, and `-l`, or shall make their use dependent on an OS version test.

- The LSB does not require the following POSIX utilities to be included in a Linux distribution, but they are widely available for Linux:

<b>alias</b>	<b>jobs</b>	<b>tput</b>
<b>bg</b>	<b>mesg</b>	<b>unalias</b>
<b>ctags</b>	<b>nm</b>	<b>uudecode</b>
<b>ex</b>	<b>strings</b>	<b>uuencode</b>
<b>fc</b>	<b>tabs</b>	<b>vi</b>
<b>fg</b>	<b>talk</b>	<b>who</b>
		<b>write</b>

The portable application designer shall not depend on the presence of the utilities listed here without ensuring that they are present in the distribution being used.

## 5. Internationalization Issues

There are two areas in which Linux and POSIX treat internationalization (a.k.a. localization) issues differently. These may affect some of the utilities (e.g., **awk**, **grep**, and **sed**) described in Section 4. These issues are:

1. **Regular Expressions** – Linux distributions follow the POSIX standard on internationalization with three exceptions:
  - a. Range expressions in Linux (such as `[a-z]`) may be ordered by code point (i.e., ordered by the binary character representation) instead of being ordered by collating element (language-dependent character ordering unique to each locale) as defined by POSIX.
  - b. Equivalence class expressions (such as `[=a=]`) and multi-character collating element expressions (such as `[.ch.]`) may not be supported by Linux.
  - c. In Linux, multi-character collating elements may not be supported.
2. **Pattern Matching Notation** – Linux distributions follow the POSIX standard except for the same three exceptions noted for regular expressions.

## 6. Considerations for Real-Time Applications

This overall document is intended to make it possible for all applications, both real-time and non-real-time, to have a high degree of portability between POSIX compliant and Linux implementations. However, real-time applications have special needs that differ from other kinds of applications.

The key operating system requirement for a real-time application is that the OS is expected to manage the system resources in such a way that the application's response time constraints can be met. This is most commonly reflected in several critical areas of operating system design:

1. Thread priorities must always be honored; the operating system must ensure that the highest priority real-time thread that is ready to run is always running.
2. When a high priority thread is waiting for some activity (e.g., a mutex or an I/O operation) attributable to a lower priority thread, the waiting time must be

Draft – Soliciting Comments

rigorously limited for that specific lower priority thread activity. This must be true within the operating system as well as at the application level. The most common mechanisms used for this are “priority inheritance” and “priority ceiling emulation.”

3. When interrupts occur, the vast majority of the resulting processing must occur at the priority of the application thread on whose behalf the interrupt occurred, not at a hardware-defined or arbitrarily high interrupt priority.
4. Clocks and timers must have sufficient effective resolution that all application timing measurements and clock/timer operations can be correctly handled.
5. If interrupts must occasionally be disabled, the time spent disabled must be so small that the resulting delay of high priority threads is essentially undetectable by the application.

Starting in 1987, a major effort was made for POSIX to include (as options) a set of resource management API's that can influence the OS internal resource management mechanisms to meet these requirements. The first, and most important set of these API's were included in the 1996 update to the POSIX specification; additional new real-time API's have also been subsequently added to the POSIX specification.

Because all of the real-time API functions are optional for POSIX compliant operating systems, claims of POSIX compliance do not indicate that an operating system is suitable for real-time applications. If an operating system vendor claims POSIX compliance, the user must ascertain which, if any, of the real-time API's are supported by the vendor.

Additionally, the support of the real-time API's does not necessarily mean that the implementation is suitable for real-time applications. It is quite possible to support every real-time POSIX API function and still not meet the resource management requirements mentioned above.

Until recently, Linux has not addressed most of the POSIX real-time requirements. Even today, the LSB does not address most real-time requirements, but a number of open source projects underway within the Linux developer community are specifically addressing these requirements. Some of these projects have been mentioned in previous sections of this document, but we bring them all together in this section.

Note that Linux-hosted applications will not find it sufficient to check the `_XOPEN_REALTIME` flag for which POSIX requires that an important set of real-time API's must be supported. The LSB requires `_XOPEN_REALTIME` to be defined as 1, even if some of the POSIX real-time API requirements are not supported. For example, POSIX requires that the asynchronous I/O API's be supported if the `_XOPEN_REALTIME` flag is defined, but the LSB does not. The portable real-time application shall separately check each of the POSIX-defined real-time option flags for each API group (e.g., `_POSIX_PRIORITY_SCHEDULING`) required by the application.

Although all of the POSIX real-time functions are optional, a POSIX real-time standard (IEEE 1003.13-2003) defines four real-time domain profiles (called PSE51, PSE52, PSE53, and PSE54 in order of increasing complexity), each specifying a complete set of both real-time and non-real-time API's that are required for that domain. An excellent approach to choosing an operating system for a real-time application is to consider which profile best fits the application, and to look for an operating system that is compliant with that profile. In this way, all of the required API's for that profile will be supported without the need for checking each individual option.

The primary POSIX real-time API's include those handling scheduling, synchronization, clocks & timers, and message passing. In this section, we briefly discuss each of them.

## 6.1 Scheduling

The basic POSIX real-time scheduling policies, which are all optional, are generally supported by all Linux distributions and most POSIX compliant operating systems. These policies, called `SCHED_FIFO` (for simple priority scheduled threads and processes), `SCHED_RR` (same as `SCHED_FIFO` except that the threads and processes are limited in their run time so other `SCHED_RR` threads at the same priority can be permitted to run in round-robin fashion), and `SCHED_OTHER` (which is undefined in POSIX, but is usually the normal UNIX time-sharing scheduling policy) provide the fundamental mechanism needed to produce predictable real-time application scheduling.

The principal problem facing the real-time application designer is determining whether the operating system's internal scheduling of its own threads and interrupt handlers follow compatible rules with the POSIX scheduling rules. Specifically, does the operating system disable interrupts (or permit device drivers to disable interrupts) for significant amounts of time? Are interrupts always run at hardware interrupt priorities (usually too high), or can their priorities be set by the application designer? Do kernel threads always run at a higher priority than the application threads? Does the operating system use the same priority range as the real-time application?

## 6.2 Synchronization

The POSIX synchronization mechanisms consist of semaphores, mutexes, and condition variables. Linux generally provides all of these mechanisms with POSIX compliant calls. However, for real-time use, the application designer should be aware of several things.

First, the POSIX semaphore is an extremely flexible mechanism that can be used for many purposes, but it is not a good choice for implementing mutual exclusion among processes or threads. This is because the semaphore does not identify the owner of the resulting locks, so the operating system has no way to avoid unbounded priority inversion in the application. This will also be true of operating system internal locks if it uses a semaphore mechanism.

The POSIX mutex is the proper way to perform mutual exclusion between threads or processes in real-time applications. For real-time applications, POSIX provides two optional mutex attributes called `PTHREAD_PRIO_PROTECT` and `PTHREAD_PRIO_INHERIT` that can prevent unbounded priority inversion for threads and processes. Current Linux distributions do not generally support these mutex attributes, but there does exist a sourceforge.org project providing “robust mutexes” that support these attributes.

The POSIX priority inversion avoidance mutex options are available in many POSIX compliant operating systems, but not all, so it is important to check their availability in both POSIX compliant and Linux systems.

### **6.3 Clocks & Timers**

POSIX clocks and timers, as with all POSIX’s other interfaces for real-time systems, are defined as POSIX options. As part of its clocks and timers interfaces, POSIX defines a C structure called `timespec` that can hold a time value (either absolute or relative) with nanosecond precision. However, the actual granularity of all the resulting clock and/or timer operations is dependent on the underlying OS implementation and the specific hardware capabilities.

Clock and timer granularity is an important issue to be evaluated for any implementation of either POSIX or Linux operating systems. The current 2.6 Linux distributions provide clock granularity that maps directly to the granularity of the underlying hardware. However, the current 2.6 Linux distributions provide a timer granularity that is limited to Linux’s software-controlled “jiffy” timer, which defaults to 10 milliseconds. The “jiffy” value can be changed, and for distributions intended for real-time use, is usually set to either 1 millisecond or 1/1024 second. This greatly improves the timer granularity, but increases the overhead for all timer processing by a factor of about 10.

An alternative Linux patch is available from a sourceforge.org project that leaves the “jiffy” timer set to 10 milliseconds, but changes the underlying timer handling to use a time event queue rather than a periodic software clock mechanism. This means that all timer requests are simply queued, and the resulting timer granularity is the same as that of the underlying hardware clock – generally less than 1 microsecond.

### **6.4 Message Passing**

Message passing between real-time threads and/or processes is a weak area in both POSIX and Linux. POSIX supports a message queue mechanism that is highly suitable for real-time applications because it supports prioritized messages with blocking and non-blocking send and receive, but the mechanism is not designed to handle messages across multiple nodes in a distributed system. Linux and some UNIX operating systems support a message passing mechanism that is derived from UNIX System V that works across distributed systems, but does not permit prioritized messages.

Prioritized message passing is an important requirement for real-time distributed applications because it permits the message passing mechanism to provide correct message ordering in the message queues.

For real-time applications not running in distributed environments, there is a patch for Linux that supports POSIX message passing that will work well for non-distributed applications.

Beyond POSIX, the Object Management Group's *Real-Time CORBA* (Common Object Request Broker Architecture) includes a message passing mechanism that can be run on both Linux and POSIX systems. CORBA communication is generally oriented around the Remote Procedure Call (RPC) paradigm, but one-way messages are also supported. CORBA implementations, including Real-Time CORBA, are available for both POSIX compliant and Linux operating systems from multiple vendors.

## **7. Summary and Conclusions**

As we have seen, there is an extremely high degree of compliance between Linux and POSIX, and portable applications can be readily created. It is expected that the level of compliance will continue to increase, although it is not yet clear when or whether complete compliance will be achieved.

It should be noted that this document is not complete, and probably cannot be complete, but it does represent a significant effort to pull together multiple sources of information. Users of this document are strongly encouraged to contact its authors if a need for corrections or updates is identified – either to add additional areas of incompatibility that have been encountered, or to correct interfaces described here whose compatibility has been found to differ or better than described.

### **References:**

1. Portable Operating Systems Interface Standard (POSIX) 1003.1-2003 (also called ISO/IEC 9945:2003), IEEE Computer Society's Portable Application Standards Committee, August, 2003.
2. Linux Standard Base Core Specification 3.0Preview1, Free Standards Group, March 31, 2005, available at [www.linuxbase.org/spec/specs.php](http://www.linuxbase.org/spec/specs.php).
3. Technical Report: Conflicts between ISO/IEC 9945 (POSIX) and the Linux Standard Base (unapproved draft 1.2.8), Andrew Josey, The Open Group, 1 April, 2005.
4. POSIX 1003.1-2003 Technical Corrigenda 2 IEEE Std 1003.1-2001/Cor 2-2004 (identical to ISO/IEC 9945:2003/Cor1:2004), IEEE Computer Society's Portable Application Standards Committee, February, 2004.
5. Programming Language C – ISO C Standard (ISO 9899:1990[1992]), also called ANSI C, ANSI Accredited Standards Committee, X3 Information Processing Systems, American National Standard for Information Systems, Spring, 2000.