1. Introduction

The maturity of the open source Linux® 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® operating system, with interfaces that are highly compliant to the IEEE’s POSIX® 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. In fact, the Linux Standard Base (LSB) Core Specification states that “It is the long term plan of the Free Standards Group to converge the LSB Core Specification with ISO/IEC 9945 POSIX.” This document aims to enhance this situation by permitting the application designer to consciously design around or otherwise handle the differences.

Many people in both the POSIX communities and the Linux communities are working to close this gap. For example, see Appendix A for a list of system interfaces and utilities whose incompatibilities have been removed since this document was begun in December 2004.

In discussing compatibility between the LSB and POSIX, it is important to understand that there are some fundamental differences between these standards. A key difference is that POSIX is a prescriptive standard, while the LSB is a descriptive standard. This means that POSIX was produced by a standards organization (i.e., the IEEE) and is specifically used by UNIX operating system developers to develop each interface, so compliance to POSIX is a result of careful design. If an incompatibility is found after implementation, it is considered a bug that must be fixed. On the other
hand, the LSB is a description of the current contents of actual GNU/Linux
distributions. The LSB is generally not used by Linux providers as a guide to
creating Linux distributions, but is more often used by application designers as a
description of what should be in a Linux distribution. If a discrepancy is found
between a Linux distribution and the LSB, it is most likely that the LSB will be
changed to remove the discrepancy. For this reason, Linux distributions frequently
describe their contents by describing the origin of each source version than by
claiming compliance with the LSB. For our purposes, however, the LSB still
provides a much more comprehensive description of the current state of Linux than
other existing documents, so it serves well as a point of comparison to POSIX.

It should also 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 document 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.

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 the specific needs of real-time systems, Section 6 discusses the
key issues involved in porting applications, and Section 7 provides a brief summary
and conclusion.

Acknowledgements

The author is grateful for review and commentary from Dave Marlow (NSWC),
Karen O’Donoghue (NSWC), Bruce McCormick (EG&G Inc), Doug Wells (The
Open Group), Leslie Madden (NSWC), and suggestions from Joe Bergmann (The
Open Group).

Trademarks

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 of these approaches: 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 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 *Conflicts between ISO/IEC 9945 (POSIX) and the Linux Standard Base* produced by The Open Group 11 November 2005 and submitted to ISO as a technical report IS 23660. This document describes the current conflicts between Linux and POSIX. That report was based on Linux Standard Base Core Specification 3.1 (LSB®) dated 31 October, 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, as well as from the minutes of Austin Group Core meetings.

### 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 to determine which options are available for the chosen OS. This is true whether the OS is POSIX compliant or is a Linux distribution. For example, general purpose POSIX-compliant OS’s such that are provided by many major server vendors may not provide the real-time options, but real-time POSIX-compliant OS’s generally
provided by independent software vendors specifically for real-time systems 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 maintenance team.

If OS version tests are used, or if major functionality not included in POSIX is needed, such as Access Control Lists (ACL’s) for higher security, it is strongly recommended that all application uses of non-POSIX API’s be collected together into localized code sections or libraries so that the effects of future 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.

- 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()**, **vfwscanf()**, **vscanf()**, **vsscanf()**, **vswscanf()**), LSB defines a length modifier “a” that, when used with the conversion specifier s (i.e., %as), 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. This is in conflict with ANSI C (included in the POSIX specification) that does not define this “a” length modifier. Thus, a conversion string such as “… %aseconds” would return a different value on a POSIX compliant system than on a Linux system. The
portable application shall not use “a” as a length modifier – it shall preallocate a string that is of sufficiently size and shall use %s for string conversion.

- 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.

### 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 using a pid value of -1 shall allow for the signal to be sent to the calling process, but it shall not depend on this for correct application behavior.

### 3.4 Threads

POSIX threads (called “pthreads”) were implemented in Linux prior to version 2.6 using a library called libthread. 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 POSIX threads. Threads implemented by libthread in Linux were called “LinuxThreads.”

Beginning with the Linux 2.6 release, a new version of libthread was used called nptl (i.e., “Native POSIX Threading Library”). This library makes a major change in the 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 libthread 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 avoiding unbounded priority inversion when mutexes are used in real-time programs. 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 – unbounded priority inversions will result. For further information about real-time application issues, see Section 5.

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. For further information about real-time application issues, see Section 5.

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 may 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.

However, a new implementation of Timers called “hrtimers” provides high resolution timers. This is available for supported platforms starting with the Linux 2.6.19 release. See Section 5.3 of this document for additional details.

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. The designer of a portable application shall check the timer resolution and its
implementation details with the Linux distribution source 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 `strerror_r()` function call** (created in POSIX as a thread-safe alternative to `strerror()` for use in threaded applications) in POSIX returns an integer. Linux uses a GNU version of this call that returns a `char*`. This potentially 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. Alternatively, there is an open source (non-GNU) Linux patch making a function `__xpg_strerror_r()` available that provides the POSIX compatible functionality – the portable application can ensure that `string.h` defines `strerror_r()` to refer to this function. This patch can be found at [http://sources.redhat.com/ml/libc-hacker/2004-03/msg00116.html](http://sources.redhat.com/ml/libc-hacker/2004-03/msg00116.html). In the event that this link changes, search on `__xpg_strerror_r` to find the patch.

2. POSIX requires that each of the values `ENOTSUP` and `EOPNOTSUPP` 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.

3. 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. POSIX defines the type `DIR` (in `dirent.h`) as a type representing a directory stream, but is ambiguous as to its actual contents. The LSB defines `DIR` as an opaque type, therefore preventing the application from processing its contents. The portable application shall not attempt to process the contents of a `DIR` value.

### 3.8 Network Interfaces

POSIX defines the `socket()` and related system interfaces to provide for TCP/IP communications among multiple nodes which can be used for connection to the Internet or Local Area Networks. Linux provides a POSIX-compatible implementation of the `socket()` and related system interface.

In response to the proliferation of IP addresses, and in preparation for new networked applications requiring extremely large numbers of IP addresses (e.g., perhaps a unique IP address for every vehicle or appliance), the POSIX standard has extended the 32-bit IP address currently in use (called IPV4) to a new format called IPV6.
128-bit IP addresses. The process of converting systems from IPV4 to IPV6 is underway.

The POSIX socket() interface permits an IPV6 implementation to communicate with both IPV6 and IPV4 addresses that have been “mapped” to IPV6 addresses, for backward compatibility.

For the interim period, Ronald Pashby, a contractor for NSWC, has performed a series of IPV4/IPV6 interoperability experiments with several POSIX-compliant operating systems, and a Linux distribution. The results, available from NSWC, indicate that a few implementation discrepancies result in some interoperability problems. This is expected to be a short-term problem while the IPV6 implementations mature. The portable application shall use the POSIX IPV4 and IPV6 API’s, but shall also check the chosen implementations for interoperability issues.

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.

• ISO/IEC 9945 requires that a number of commands be provided as built-in to the command language interpreter (as detailed in XCU section 1.13) and also accessible via the exec family of functions as defined in XSH and directly invocable by standard utilities as needed (env, find, nice, nohup, time and xargs). The LSB requires that the cd, getopts, read, umask and wait utilities be built-in to the command language interpreter and forbids standard utilities from invoking them. Because these utilities may not be available in Linux as executable binaries, the portable application shall not invoke them using the exec family of functions. Instead, the portable application shall use POSIX system interfaces such as chdir() and getopt().

• 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.

- The Linux **bc** utility, which performs arbitrary precision computations, contains a number of extensions beyond POSIX functionality. The portable application shall avoid using this extended functionality, such as arbitrary length variable names, and shall instead use the POSIX definitions of options and the POSIX supported language syntax and semantics. In addition, the portable application shall not depend on correct processing of the POSIX-defined LANG environment variable and POSIX environment variables starting with “LC_”. 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.

- 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 and the –d options. The portable application shall not use these options.

- The LSB does not require the Linux **fuser** utility to support the –c and the –f options. In addition, this utility does not correctly handle a block special file with no options. The portable application shall not use the –c or –f options, and shall not use it with block special files with no 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 the –l and –u options 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, –l, –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.

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

<table>
<thead>
<tr>
<th>Utilities</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>484</td>
</tr>
<tr>
<td>bg</td>
<td>485</td>
</tr>
<tr>
<td>ctags</td>
<td>486</td>
</tr>
<tr>
<td>ex</td>
<td>487</td>
</tr>
<tr>
<td>fc</td>
<td>488</td>
</tr>
<tr>
<td>fg</td>
<td>489</td>
</tr>
<tr>
<td>jobs</td>
<td>490</td>
</tr>
<tr>
<td>mesg</td>
<td>491</td>
</tr>
<tr>
<td>nm</td>
<td>492</td>
</tr>
<tr>
<td>printf</td>
<td>492</td>
</tr>
<tr>
<td>strings</td>
<td>493</td>
</tr>
<tr>
<td>tabs</td>
<td>494</td>
</tr>
<tr>
<td>talk</td>
<td>495</td>
</tr>
<tr>
<td>tput</td>
<td>496</td>
</tr>
<tr>
<td>uudecode</td>
<td>497</td>
</tr>
<tr>
<td>uuencode</td>
<td>498</td>
</tr>
<tr>
<td>vi</td>
<td>499</td>
</tr>
<tr>
<td>who</td>
<td>500</td>
</tr>
<tr>
<td>write</td>
<td>501</td>
</tr>
<tr>
<td>unalias</td>
<td>502</td>
</tr>
</tbody>
</table>

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. 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 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. Today, however, with the 2.6.19 and later kernels, Linux does address many real-time
requirements. For earlier distributions, a number of open source projects within the Linux developer community specifically addressed 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 the following sections, we briefly discuss each of them.

### 5.1 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 mechanisms 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?

5.2 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 they use 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. Beginning with Linux 2.6.18 distributions, Linux generally supports the PTHREAD_PRIO_INHERIT (but not PTHREAD_PRIO_PROTECT) attribute. For previous Linux distributions, there exists an open source patch providing “robust mutexes” that support these attributes (see http://developer.osdl.org/dev/robustmutexes/).

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.

5.3 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, prior to 2.6.16 Linux distributions provided a timer granularity that was limited to Linux’s software-controlled “jiffy” timer, which defaults to 10 milliseconds. The “jiffy” value could be changed, and for distributions intended for real-time use, was usually set to either 1 millisecond or 1/1024 second.
This greatly improved the timer granularity, but increased the overhead for all timer processing by a factor of about 10.

A Linux kernel mechanism called “hrtimer” was developed that has been incorporated into the Linux 2.6.16 and later distributions that, for some CPU’s, provides a mechanism to greatly increase timer resolution when using the POSIX timer interfaces. The hrtimer implementation is currently (as of 2.6.19) available on X86 CPU’s and others if a suitable clock driver is available. For older Linux distributions, an alternative Linux patch is available from a sourceforge.org project that leaves the “jiffy” timer set to 10 milliseconds, but, similarly to the hrtimer mechanism, 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 (see http://high-resolution-timers.sourceforge.net/).

5.4 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, which is where the majority of message passing is used. Linux (and some UNIX operating systems) support another message passing mechanism that is derived from UNIX System V that works across multiple nodes, but this mechanism does not permit prioritized messages, and is therefore difficult to use in real-time applications.

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 and permits the application to limit the resulting priority inversion.

POSIX message passing is supported in Linux starting with version 2.6.6, so a portable real-time application that does not require sending messages across multiple nodes in a distributed system can, and should, use the POSIX message queues.

Beyond POSIX, the Object Management Group has developed two standards that provide communication across multiple nodes in a distributed system. The first of these standards is Real-Time CORBA (Common Object Request Broker Architecture), designed for Object Oriented systems making language and OS-independent method invocations among distributed objects (using the Remote Procedure Call (RPC) paradigm). The second OMG standard is the Data Distribution Service (DDS) that permits language and OS-independent messages to be handled among multiple nodes, and includes parameters defining Quality of Service (QoS), including priority. Commercial implementations of both of these standards are available, and should definitely be considered for portable real-time distributed applications.
6. Application Porting Between Linux and POSIX

This document has been prepared primarily to discuss the possibility of creating applications that can be easily ported between POSIX and Linux platforms. However, most application development is not targeted to create all-new applications, but rather is updating or extending an existing application. Such existing applications are not likely to have followed the rules defined in this document.

In this section, we discuss some of the critical aspects of application porting from a POSIX or UNIX platform to a Linux platform, or vice-versa.

If the application to be ported followed the rules in this document, the porting process consists of working through the steps of:

1. Setting up the compilation environment in the target environment,
2. Checking any platform-dependent module(s) for platform-dependent issues,
3. Rebuilding the application, and
4. Testing the application, including its performance requirements.

If the application being ported was designed to follow these rules, the effort to do this should be dominated by the first and third steps.

For legacy applications that were not designed according to these rules, these same steps will be followed, but the effort for the port will increase for the second and fourth steps. In fact, if many non-POSIX API’s were used, the second and fourth steps will probably dominate the porting effort.

There are three general approaches that could be applied to the process of porting a legacy application. The choice of which approach to use will have a significant effect on the cost and schedule required for the port. It should be remembered that the lifetime of a platform is typically much shorter than the lifetime of the application for the vast majority of embedded and real-time applications.

These general porting approaches are:

1. Directly port the application, making minimal changes to correct any interfaces that “break” the application. This is the least cost and schedule porting approach, but it is likely that the resulting application will still not satisfy the rules defined by this document, so a later port of the application to a new platform is likely to require a similar effort. For example, this can occur when the initial infrastructure contains non-POSIX BSD API’s or utilities that may be available in some Linux environments but not others, leading to incompatibilities in future infrastructure porting efforts.

2. Port the application as in the first goal, but search the application for non-POSIX API’s that can be replaced by POSIX API’s. For those API’s that are critical to the application, or are used frequently in the application, do the update so that further ports will be able to largely exploit the greatly reduced porting costs enjoyed by applications that follow these rules.
3. Search the application for all OS and middleware API’s, and replace all non-POSIX or non-standards-based middleware API’s. The resulting application should fully comply with the rules in this document, and further ports should require a much lower level of effort.

For each port, the porting agency should carefully evaluate the goals to be pursued, trading off the cost and schedule of the port against the likely future requirements of the application.

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 author 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 different or better than described.

References:


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.


Appendix A.

Visible Progress Reconciling Linux and POSIX

This paper was initially drafted in December 2004. At the time, it was clear that there was good momentum reconciling Linux to POSIX, but it was not clear how fast this process might show progress. The purpose of this appendix is to identify some of the changes made to this document due to progress in reconciling Linux and POSIX between that time and December 2006. The order of these changes is arbitrary—not necessarily chronological.

1. Formerly, Linux permitted several file system functions to return either **ENODEV** or **ENXIO** for a non-existent device, while POSIX requires an **ENXIO** return in this case. Linux now returns the correct error under this condition. This applied to **fopen()**, **freopen()**, **open()**, and **creat()**.

2. Linux did not require implementation of the WCONTINUED functionality of the **waitpid()** call. This also implied that the dependent WIFCONTINUED macro was not required to be available. Linux now supports this POSIX functionality.

3. The POSIX **waitid()** function was not available in Linux. This is now available in Linux.

4. The POSIX pthread functionality was implemented in Linux using “LinuxThreads” which, while very similar to pthreads, was not completely POSIX compliant. Linux now supports pthreads using the nptl library which is POSIX compliant.

5. Effective with the Linux 2.6.19 distribution, the “hrtimers” implementation provides high resolution timers, effectively completing its compatibility with POSIX Clocks and Timers.

6. Formerly, the Linux **getopt()** function (called iteratively to parse the argument string passed to the application’s main() function) permitted several differences in its interface under certain circumstances relative to a POSIX system. This function is now POSIX compliant.

7. POSIX requires that **_XOPEN_VERSION** should be defined with a value of 600 for conforming implementations. Formerly, Linux defined this value as 500. This has now been corrected.

8. Formerly, Linux did not support all of the POSIX Internationalization capabilities. Linux is now POSIX compliant in this area.

9. Formerly, the Linux **chgrp** and **chown** utilities did not need to support the –L, –H, or –P options that determine some of the behavior of the –R option. Linux is now POSIX compliant in this area.

10. The Linux **cut** utility had an option used with multi-byte character strings to ensure that multi-byte characters were 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 was not as precise. Linux is now POSIX compliant in this area.

11. The Linux **fuser** utility was not required to support the –c and the –f options. Linux is now POSIX compliant in this area.
12. The `xargs` utility had compatible functionality, but used lower case for the POSIX options –E, –I, and –L. This means that these utility options could not be used portably. Linux is now POSIX compliant in this area.

13. Formerly, the Linux mutex operations were generally POSIX compliant, but did not support `PTHREAD_PRIO_INHERIT`. This is now supported in Linux starting with the 2.6.18 release.

14. POSIX message passing is now supported in Linux starting with 2.6.6.