POSIX, a developer's view of standards

Heiko Eißfeldt, *Unifix Software GmbH*

November 20, 1996

Abstract: First I give an introduction to the POSIX.1 set of standards. I am going to continue with an survey of POSIX.1, POSIX.1b and POSIX.1c. Standards as treated here are seen as an application programming interface (API), defined by some kind of protocol, between application and system programming. In the following I am going to examine what functionality is provided and what is available under Linux. Mostly I take the perspective of an application programmer, however sometimes some additional remarks on POSIX.1 certification issues are given. Included topics are POSIX.1, its extensions, and a short review of XPG4 features.

Introduction

In order to produce a high quality, portable code nowadays a developer is expected to pay attention to applicable standards. They allow the use of common interfaces and pose a set of rules to the implementor as well as to the application developer. Prior to standard definitions a vendor often has to maintain several interfaces in parallel. Standards reduce these different treatments to one general accepted method. Having a code at one place instead of multiples enhances the consistency for later maintenance. Nowadays the value of standards is seldom being questioned. In the long run the gains of reusable code, clearly defined interfaces and functionality are obvious. This is valid especially for Linux, as proven by the rapid ports to other platforms and the amazing amount of applications becoming available in a short time.

Scope: While there are also "industry standards" and "de facto standards", I am dealing here exclusively with official (inter)national standards. These standards released by authoritative independent organizations are commonly regarded as final. The other so-called standards are not independent of companies and often lack an authoritative forum. Use of the covered functionality and/or interface is generally coupled with license fees. International standards on the other hand are designed to give the most generic and fundamental definitions for worldwide use. Often the drafting process of a standard is influenced by the public. The consideration of highly different interests and the unrestricted usage of the standard makes them so important and wide applicable. This is crucial to wide-spread distribution and success. Standards as treated here can be thought of as an abstract model of the interface between user applications and system internals. The language used in the standard is a formalized English and may be difficult to read at times. But the advantages of an unambiguous and definitive description outweigh the sometimes difficult wording by far. The important concepts of portably internationalization/localization of applications are too complex to treat here. So they are outside of the scope of this talk.

History

The development in the computer software field has accelerated enormously over the last decades. Everything seemed possible limited only by imagination. This fast growth has created a lot of new applications. With more powerful hardware applications got bigger and bigger. As complexity increased maintenance of software became more expensive. Especially upgrading slightly incompatible hardware became harder. Hardware dependencies, weak error handling, and static limits throughout the software caused the main problems. Strict structuring, breaking big functions/modules into manageable pieces and modular design were considered essential. Abstractions like high level programming languages and structured programming were used and became new programming virtues which diminished these problems.

Portability: A difficult problem to tackle was that of portability. Here a more global approach was required. At first the need for a C language standard was recognized, but this task took some time. The original specification from Kernighan and Ritchie lacked the required rigor and completeness.

Preprocessors for example were not fully specified and thus implemented with different semantics.

For most obscure examples of misuses of C have a look at the fascinating winner entries of early International Obfuscated C Contests [1].

```
main(){printf(&unix["\021%six\012\0"],
(unix)["have"]+"fun"-0x60);}
```

©David Korn, AT&T Bell Labs
Best one-liner in the International Obfuscated C Contest 1987

ISO-C: The proposed standard had to be an evolutionary step (otherwise it would have been rejected), since high compatibility to large quantities of existing C source code was mandatory. Nevertheless newer powerful features from the development of C++ (i.e. function prototypes, 'const' modifiers) were included to support a more precise declaration. Thus the compiler could check for matching function parameters. To ensure acceptance as an international standard one of the goals was extended support for non American character code sets (internationalization (also called i18n) and localization (short l10n)). This seems to be one of the most complex tasks, since it has not been finished yet. Finally C as defined by ANSI (American National Standards Institute) was ready. It was later adopted by ISO (International Standards Organization) as in international standard [2]. This was considered a major breakthrough however the need for it has been there for quite a while.

POSIX.1: The next step was to enhance the definition of the interface to the operating system, which needed to be unified and specified. Due to the free availability and powerful flexible underlying concepts of early Unix versions, a lot of different versions evolved. The two main variants were AT&T's System V and Berkeley's BSD. Trouble spots were for example different semantics in terminal handling and signal handling.

POSIX.1 (Portable Operating System Interface) was based on classic C (K&R) and ISO-C, but added extra requirements for better localization and extended the C library functions to multitasking needs. ISO adopted POSIX.1 in 1990 [3]. The standard was created with the aim to allow implementations on most platforms. In order to adjust to differently powerful platforms, POSIX.1 contains a lot of options. Backward compatibility was an issue. So instead of fixing semantics of old interfaces, a group of new functions with clearly defined semantics were introduced and recommended for future use (i.e. new functions for signal handling). The American FIPS-151-2 standard by NIST (National Institute for Standards and Technology) used POSIX.1 as a base and demanded stricter minimum requirements (features like job control and certain system limits). NIST provided test suites for conformance testing, which had and have to be passed to claim FIPS-151-2 conformance. All Unix-like operating systems purchased by the government have to be certified FIPS-151 systems.

X/Open Company: X/Open was founded by major Unix vendors to specify commonly needed extensions based on POSIX.1 (the latest release is called X/Open Portability Guide Release 4 or XPG4). Compared to other organizations X/Open is more open to European influences due to its European members. Their pioneering standardization efforts includes topics like i18n, streams, NFS, and curses. The required minimum functionality takes considerable implementation efforts and can be achieved mostly for Unix-based systems.

Summary: In order to reduce chaos rising from fast growth in the computer field, standard bodies (like IEEE, ANSI, ISO etc.) have constituted work groups to supply suitable abstractions for interchange of information, functionality and source codes.

Standards should:

- Define a sound, consistent interface and functionality.
- Allow re-usage of code across different platforms.
- Allow for reduction of code complexity.

What constitutes a standard: Before creating the layout of a new standard working groups from standard bodies like IEEE, ISO/IEC, ECMA, ITU analyze the existing practice and problems. National committees often perform the role of pioneers and are faster, while international organizations do often integrate the results of the connected national bodies.

The creation of a standard is usually an iterative process taking years from early drafts through balloting up to the final approval. Later the standard bodies are the authoritative forum for interpretation questions of the standard. To ensure portability limitations and differences, like national and hardware specifics, have to be overcome and unified. The underlying system and environment should hide these differences to applications as much as possible.

Often this leads to more complex interfaces, but of all abstractions the most elegant and simple solutions should be preferred.

The result addresses two major groups, system implementators and application developers. Both have to satisfy the requirements of the standard to ensure the expected portability and functionality. The abstraction model can be seen as a "gateway" between the system or language environment (supplying services) and the applications (requesting services).

The POSIX family

There are a lot of different standardization efforts in the POSIX set. The full titles are given in the reference section. [1]

Here are some of the relevant POSIX standards for developers:

- POSIX.1, basic OS interface (C language) 1988, 1990, 1993, 1996 [3]
- POSIX.1a, extensions (symbolic links) [4]
- POSIX.1b, real-time and I/O extensions, included in 1003.1-1996 [5]
- POSIX.1c, threads, included in 1003.1-1996 [6]
- POSIX.1d, more real-time extensions
- POSIX.1e, security extensions (access control lists) [7]
- POSIX.1f, transparent file access
- POSIX.1g, socket interface [8]
- POSIX.1h, real-time features in distributed systems
- POSIX.1i, corrections to POSIX.1b, included in 1003.1-1996 [9]
- all of POSIX.2, shell and utility programs [10], [11], [12], [13],
- all of POSIX.3, test methods [14], [15], [16], [17], [18],

The current "all-embracing" specification is probably XPG4 UNIX ([19], [20], [21], [22], [23]), which adds a lot of useful interfaces. Together with POSIX.1:1996 this can be seen as state of the art.

[1] In 1993 the numbering of most POSIX parts were changed to give a better ordered scheme.

In the following I concentrate on POSIX.1 (ISO/IEC 9945-1:1990, IEEE Std.1003.1) and POSIX.1b/c. In this text I pay attention only to C language bindings, although other language bindings are available or being developed (ADA and FORTRAN).

Overview of POSIX.1: POSIX.1 covers the basic functionality of an operating system. It constitutes the foundation for higher POSIX standards like POSIX.1a (symbolic links). Affected parts of the system are the kernel, the C and math library and system header files. A prerequisite is K&R C or ISO-C conformance. Linux' C library consists mainly of GNU/FSF code which is extremely stable and well tested. Linux has been designed with POSIX.1 in mind, so all functionality is basically available. Based on that commercial FIPS-151-2/POSIX.1 certified Linux versions have become available [24].

The chapters of POSIX.1-1990 are

- **General Requirements.** Header files contents, feature test macros, system limits.
- **Process primitives.** Process creation, process termination, reliable signal functions.
- **Process environment.** Process identification, user identification, process groups and sessions.
- **Files and directories.** Directory functions, working directory, file creation, special file creation, file removal, file characteristics, configurable pathname variables.
- **Input and output primitives.** Pipes, file descriptor manipulation, input and output, control operations.
- **Device and class-specific functions.** Baud rate functions, get and set state, line control, foreground process group id.
- **Language specific services for C programming.** Function extensions over ISO-C.
- **System databases.** Group and user database access.
- **Data interchange format.** Extended tar and cpio format.

Topics not considered: Some important topics are not included in the POSIX.1 standard. The most relevant are networking, graphics services and system administration functions. Often programmers cannot restrict their programs to pure POSIX.1

functionality because advanced features are not yet standardized. However, it is sensible to use as many POSIX functions as possible in order to keep the portable part at its' maximum.

Application developer's view: Now I will address some topics relevant for application developers. In an application one might want to check whether a conforming POSIX.1 system at all is running, if so which version there is and finally what optional parts are supported by the implementation. Although these questions could be answered by the accompanying POSIX.1 Conformance Document, it is useful to request this information from the compilation environment and at runtime.

Identifying a POSIX.1 system: POSIX.1 provides some means to check which version is available.

When the symbolic constants `_POSIX_SOURCE` and `_POSIX_C_SOURCE` are defined prior to inclusion of the header file `unistd.h`, the symbol `_POSIX_VERSION` is defined afterwards on POSIX systems. Moreover, its value defines the POSIX.1 version.

Example:

```
#define _POSIX_SOURCE 1  /* old style */
#define _POSIX_C_SOURCE 199009
#include <unistd.h>
#ifdef _POSIX_VERSION
  /* It is a POSIX.1 system */
...
#endif
```

Depending on the value of the symbol `_POSIX_VERSION`, functionality introduced by a certain release of POSIX.1 can be expected. Possible values of `_POSIX_VERSION` are

- **symbol is not defined** → POSIX.1 is not supported on this system
- **198808L** → POSIX.1 as of 1988 is supported (IEEE 1003.1-1988) (possibly FIPS-151-1)
- **199009L** → POSIX.1 as of 1990 is supported (ISO/IEC 9945-1:1990) (possibly FIPS-151-2)
- **199309L** → POSIX.1 and POSIX.1b as of 1993 are supported (IEEE 1003.1-1993)
- **199606L** → POSIX.1, POSIX.1b, POSIX.1c as of 1996 are supported (IEEE 1003.1-1996)

What optional parts are included?: Certain advanced features of POSIX.1 are optional and thus may not be present. For instance Windows NT and small embedded systems do not support any options while most other systems offer the most useful of them. [2] Portable applications are required to check for optional parts before using them. POSIX.1 supplied "feature test macros" for this purpose. Special symbols can be checked after the inclusion of the `unistd.h` header file.

Compile time option symbols: These are

Symbol	is defined, when
_POSIX_JOB_CONTROL	process groups can be suspended and resumed.
_POSIX_SAVED_IDS	saved SETUIDs and SETGIDs are supported.

Real-time extensions compile time option symbols: These are

Symbol	is defined, when there is support for
_POSIX_ASYNCHRONOUS_IO	asynchronous I/O functions (`aio_*()`)
_POSIX_PRIORITIZED_IO	prioritized asynchronous I/O (see below)
_POSIX_SYNCHRONIZED_IO	explicit synchronization
_POSIX_FSYNC	the `fsync()` function

[2] To find out what POSIX.1 certified product supports what major options, a list from POSIX@nist.gov can be obtained. An email with body "send 151-2reg" to this address will make the list via email available.

Symbol	is defined, when there is support for
_POSIX_MAPPED_FILES	
	memory mapped functions (`mmap()`, `munmap()` and `ftruncate()`)
_POSIX_MEMLOCK	
	locking functions `mlockall()` and `munlockall()`
_POSIX_MEMLOCK_RANGE	
	_POSIX_MEMLOCK and ranges with `mlock()` and `munlock()`
_POSIX_MEMORY_PROTECTION	
	function `mprotect()`
_POSIX_SHARED_MEMORY_OBJECTS	
	shared memory (functions `shm_*()` and `mmap()`, `munmap()` and `ftruncate()`)

Symbol	is defined, when there is support for
_POSIX_PRIORITY_SCHEDULING	
	process scheduling control (functions `sched_*()`)
_POSIX_REALTIME_SIGNALING	
	real-time signal functions `sigwaitinfo()`, `sigtimedwait()` and `sigqueue()`
_POSIX_SEMAPHORES	
	semaphore functions `sem_*()`
_POSIX_MESSAGE_PASSING	
	message queues (`mq_*()` functions)
_POSIX_TIMERS	
	clocks and timers (functions `clock_*()` and `timer_*()`)

Mandatory parts of POSIX.1b: POSIX.1b introduces a new set of signals (see below), which is mandatory. Guaranteed are at least 32 user signals in the range SIGRTMIN up to SIGRTMAX.

Header files: The visible parts of the interface are the system header files. Since standard C full type checking is performed by the C compiler through function prototypes. The header files provide a single location for the declaration of each function. Thus the library or kernel will use the same declaration as the application and consistency is achieved. By use of header files all explicit external declarations outside of header files can be eliminated.

Header files allow for macro and/or function implementations of the requested routines. If both implementations are present, the macro definition will take precedence.

Name space issues: At compilation time symbols as well as global data and code definitions of system and application components are combined. A similar thing happens at the linking stage when names are resolved from different object files and libraries. In both cases there is a potential for name conflicts (also called name space pollution). To avoid conflicts at compile time ISO-C, POSIX.1 and XPG4 reserve special namespaces for exclusive system usage. For example all identifier names with leading underscores are reserved for system use. At linking time there is no type checking available. So an integer variable could be linked against a function pointer reference with the same name.

System limits: The header file `limits.h` contains the implementation defined limits. An application can use these limits at compile time, check them at runtime with the `sysconf()` function and adjust itself accordingly. Other limits may depend on the file system being used. In these cases the functions `pathconf()`, `fpathconf()` are to be used to get the specific limit at runtime (see example below).

`Pathconf()` operates on pathnames while `fpathconf()` needs an opened file descriptor as an argument.

Example:

```
#define _POSIX_SOURCE 1  /* old style */
#define _POSIX_C_SOURCE 199009
#include <unistd.h>
int main (void)
{
  long tmp_namemax;
  /* what is the allowed file name
     length in the /tmp directory? */
  tmp_namemax = pathconf("/tmp",
                         _PC_NAME_MAX);
  ...
```

Runtime symbols for use with the `pathconf()` and `fpathconf()` functions: Some functionality is dependent on file systems. Thus runtime checks are needed to test for the presence of a feature for a particular file. `Pathconf()` and `fpathconf()` can be used to return what options are present for a particular pathname. The options are

Symbol	checks, if
_POSIX_CHOWN_RESTRICTED	`chown()` is restricted on a particular file.
_POSIX_NO_TRUNC	an error is generated by filenames that are too long.
_POSIX_VDISABLE	special-character processing on terminals can be disabled.

Real-time extensions runtime symbols for use with the `pathconf()` and `fpathconf()` functions: These are

Symbol	requests if these operations are allowed on a particular file
_POSIX_ASYNC_IO	asynchronous I/O
_POSIX_PRIO_IO	prioritized I/O
_POSIX_SYNC_IO	synchronous I/O

Application conformance classes: A "strictly conforming application" may only use POSIX.1 specified routines and must be prepared for any unspecified or implementation defined behavior. For example programs from test suites are supposed to belong to the strictly conforming class. "ISO/IEC conforming applications" may require optional parts and other standards as long as these requirements are documented. "Conforming applications using extensions" may require any POSIX.1 compatible extension, but all of them shall be documented.

Some experiences from conformance testing: In fall 1995 Unifix did conformance testing with Linux 1.2.13 and the NIST PCTS test suite. What were the typical errors found? One prerequisite for POSIX.1 was ISO-C conformance. Linux did not master it. This required a revision of the C library header files and fixing the math library. Only then we could compile the test programs without warnings and errors. All unsuccessful test results could be divided into misconfigurations, wrong header files, kernel bugs, C library bugs and test suite bugs. All kinds of bugs were present. Common kernel bugs were errors when reaching resource limits, wrong error return codes and incorrect handling of file names that are too long. Signal propagation into process groups had flaws and the `execlp()`/`execvp()` functions could not start shell scripts. On the other hand Linux' `verify_area()` function revealed two bugs in the test suite where illegal arguments were passed to the system. The test suite fixes have been sent to NIST but are not incorporated since now. The kernel patches for POSIX.1 conformance have been sent to Linus Torvalds and are mostly integrated by now.

POSIX.1b features

POSIX.1b was created to add real-time related functionality. The real-time extensions found their way into the latest POSIX.1 standard from IEEE (IEEE Std.1003.1:1996) now and are becoming important. Linux does not have a complete set of real-time features yet. There is an interesting real-time extension from Yodaiken et al. (who also gives a talk here on USELINUX), but it does not follow POSIX.1b semantics. Linux' current status of POSIX.1b work and pointers to several related projects are very good described in [25].

In future Linux developers will probably implement more of these interfaces. Real-time capabilities and high performance I/O will greatly enable Linux for time critical applications like industrial process control systems, real-time database systems, multi media programs, sampling/recording programs, embedded systems, EPROM burners etc. Without real-time support these kind of applications are currently more easily done in single user, single tasking environments, where the whole system can be dedicated to the application.

The following description of POSIX.1b features will hopefully arouse your curiosity and wet your appetite for the implementation of these features.

Real-time signals: POSIX.1b introduced a new variant of signal: real-time signals. They are now lined up in different queues according to priority. So there are no losses anymore (POSIX.1 left this detail implementation defined). The delivery order is the same as the queuing order. The lowest signal number (highest priority) is delivered first. The number of user signals is increased and signals carry a data item (an integer or pointer). There is a faster response mechanism for blocked signals, which does not invoke the signal handler (functions `sigwaitinfo()` and `sigtimedwait()`). When the SA_SIGINFO flag is set, the signal handler is invoked with three (new real-time style) instead of one parameter (old style). Extra data can be passed with the `sigqueue()` function instead of `kill()`. New functions are `sigwaitinfo()`,`sigtimedwait()`, `sigqueue()`.

Interprocess communication (IPC): POSIX.1 offers pipes and FIFOs for process communication purposes. System V introduced more efficient mechanisms (which are present in Linux). Compared to System V POSIX.1b shared resources are identified by strings instead of numeric keys. Shared memory and memory mapped files are handled through the `mmap()` function. Message queues can be prioritized. Highly efficient semaphores are possible through a new flavor of memory based semaphores, when they are placed in shared memory. As noted in [25] database servers could benefit greatly from this feature as less context switches have to occur.

Memory (un)locking: Locking of critical memory portions against swapping can be handled through these functions: `mlockall()`, `mlock()`, `munlockall()`, `munlock()`. `Mlockall()` locks all of the current program code, data, heap, stack memory as well as all used shared libraries and shared memory. Unlocking is done via `munlockall()`. Optionally all future allocations can be locked too. Parts can be locked with `mlock()`, which is supplied with an address and the range of the memory area to be locked. The granularity is typically the page size used by the implementation.

Scheduling policies: Hard real-time processes (which demand a maximum response time) can switch the scheduling policies they run under. Three schemes are supported, the first two of them being portable. Selectable scheduler algorithms are

- **SCHED_FIFO** (First in first out) This is the priority-based preemptive scheduler. No time slicing is done. Processes of each priority level are queued independently. Lower prioritized processes are run only when there are no higher prioritized runnable processes. Highest priority processes run until they are finished or are blocked. Lower priority processes are interrupted by higher priority ones. The FIFO behavior applies for processes of the same priority level.

- **SCHED_RR** (Round robin) This policy has priority levels as in SCHED_FIFO, but processes with the same level can preempt one another after a certain time. This can yield a more balanced timesharing of processes of the same level. After a certain time quantum the running process is preempted and queued at the end of its priority level.

- **SCHED_OTHER** (under Linux `nice` based) Variable priorities are assigned by the kernel through some sort of heuristic. This can be influenced through the `nice()` function. The nice value has a statistical (averaged) effect. Relying on characteristics of this variant makes a program non portable.

These system calls and scheduler algorithms are implemented in newer Linux kernels.

Asynchronous I/O (AIO): POSIX.1 offered the O_NONBLOCK flag for a limited use of nonblocking I/O. Other systems had additional functions `select()` and `poll()`. POSIX.1b on the other hand can offer asynchronous I/O with signaling at completion. Multiple outstanding requests are queued and maintain their private error return status. Of course they can be prioritized. List I/O allows for atomic reads or writes of lists of buffers. Other systems had similar but synchronous functions `writev()` and `readv()`. New functions are: `aio_cancel()` (cancel uncompleted AIO), `aio_error()` (retrieve the error return value for an AIO), `aio_read()` (perform an AIO read), `aio_return()` (retrieve the return value of an AIO), `aio_suspend()` (wait for an AIO to complete), `aio_write()` (perform an AIO write), `lio_listio()` (launch many AIO operations at once).

Synchronized I/O: To force buffered data to be written to the output device, synchronous I/O is needed. `fsync()` will flush explicitly, files opened with the O_SYNC flag will have all writes flushed automatically. The new function `fdatasync()` flushes only data but not necessarily meta data (the system's bookkeeping data, e.g. inodes). Memory mapped files can be flushed with the `msync()` function. This functionality has been implemented in Linux with `fdatasync()` currently mapped to `fsync()`.

Timers: Interval timers and one-shot timers are provided. At least 32 interval timers per process with an resolution of at least 20 milliseconds are guaranteed. Timers are dynamically allocated and atomic operations are provided for absolute time setting. New functions are `nanosleep()`, `clock_gettimer()`, `clock_settimer()`, `clock_getres()`.

Thread extensions

POSIX.1c standardized threads (sometimes also called light weight processes). Basically they can be seen as children of a parent process with shared resources. In order to distinguish this variant from others they are termed "pthreads". Advantages of pthreads are transparent utilization of multiprocessors, simplified programming and efficient usage of memory resources and of course portability. The scope of this talk allows me to only briefly sketch out the new functionality.

What is provided: Thread creation/deletion is done via `pthread_create()` and `pthread_exit()`. Upon termination of a thread another one waits for that event with `pthread_join()`. If this is to be avoided a thread can call `pthread_detach()` (thread resources are freed on termination).

Attribute functions: They allow access to thread characteristics such as scheduling policies, detach state and scheduling contention scope.

Function	Description
pthread_attr_...	
init()	Create an attribute
destroy()	Delete an attribute
setdetachstate()	Set detachment state
getdetachstate()	Get detachment state
setscope()	Set sched. contention scope
getscope()	Get sched. contention scope
setinheritsched()	Set sched. inherit.
getinheritsched()	Get sched. inherit.
setschedpolicy()	Set sched. policy
getschedpolicy()	Get sched. policy
setstacksize()	Set stacksize setting
getstacksize()	Get stacksize setting
setschedparam()	Set sched. parameters
getschedparam()	Get sched. parameters

Synchronization: Multiple threads need means for synchronization to avoid race conditions on their shared resources. Mutexes and condition variables are provided.

Mutexes: These are simple binary locking switches.

Function	Description
pthread_mutex_...	
init()	Initialize a mutex
lock()	Lock a mutex
unlock()	Free a mutex
trylock()	Check for locking a mutex
destroy()	Delete a mutex

Condition variables: These can be compared to semaphores. Threads use them typically to wait for a resource to become available. If the condition is reached threads are woken up (without the overhead of the signal mechanism).

Function	Description
pthread_cond_...	
init()	Initialization
destroy()	Deletion
wait()	Wait on termination
timedwait()	Wait with timeout
signal()	Signal a condition
broadcast()	Broadcast a condition

(Priority) queue handling functions:

Function	Description
pthread_...	
queue_init()	Initialization
queue_remove()	Deletion
queue_enq()	Enqueue
queue_deq()	Dequeue
queue_get()	
prio_queue_init()	Initialization
prio_queue_enq()	Enqueue
prio_queue_deq()	Dequeue

Miscellaneous functions:

Function	Description
pthread_...	
once()	Initialize a routine only by the first thread
self()	Return callers thread ID
equal()	Compare thread IDs
yield()	Yield the processor to other threads
cleanup_push()	Register cleanup function
cleanup_pop()	Delete cleanup function
setschedparam()	Set scheduling policy
getschedparam()	Get scheduling policy

Thread-specific data: Although visible to all threads, a thread can use his private slot in the key array and associate data with it.

Function	Description
pthread_...	
key_create()	Create a specific data key
key_delete()	Delete a specific data key
setspecific()	Associate a value with a key
getspecific()	Get the key-associated value

Linux' status: Linux has two flavors of pthread implementations. Newer C libraries already contain the user-level thread package from Chris Provenzano, whereas Xavier Leroy developed an optional kernel-level package based on Linux' special `clone()`

system call. The latter version has slight deviations from the POSIX.1c behavior. One is that different threads from one parent have different process IDs (until the CLONE_PID flag is implemented). The Linux kernel will probably add the functionality in the 2.1.x series. Higher bits of the process ID already contain a thread ID (TID). The user space implementation relies on cooperative multi threading. Meaning that a single thread can monopolize the CPU without being preempted by other threads.

Changes to libraries are necessary since libraries have to be thread safe. The code has to be reentrant and some traditional global variables have to be changed to one instance per thread. One example is `errno` which cannot be used anymore directly (since it would be shared between threads). Instead a thread-local `errno` has to be used.

X/Open's XPG4 specifications

X/Open has specified a very rich set of functions for their XPG4 (X/Open Portability Guide) Common Application Environment (CAE). These functions were chosen from a large set of representative applications (50, among them CAD, database systems, and 3500 utilities). The more often required extra functions were added to the functionality of POSIX.1/.2 and are based on System V, 4.3BSD and Open Software Foundations AES (Application Environment Specification) functionality. The most extended profile XPG4 UNIX (aka SPEC 1170) defines what a branded UNIX system has to have. The name UNIX is controlled by X/Open company. X/Open will try to keep XPG compatible or aligned with POSIX.1 and POSIX.2, thus its extensions are kept in synchronization. In case of conflicts there are changes to expect for XPG4. When ISO releases a new version of POSIX.1 a conflict might arise, although the close cooperation between standard bodies ought to avoid this.

One remarkable aspect is the pioneering effort of X/Open in the field of i18n. Library functions are greatly extended with wide character string functions (`wcs*()`). Other additions are network file systems (NFS) and curses terminal interfaces.

XPG4 profiles: There are two modern branding (XPG4's term for certification) profiles: **XPG4 Base 95** and **XPG4 UNIX** (also known as SPEC 1170).
XPG4 UNIX contains

- "Internationalized System Calls and Libraries Extended". POSIX.1, POSIX.2, ISO C Multi byte Support Extension Addendum, STREAMS, Sys V shared memory calls.
- "Commands and Utilities V2". POSIX.2 shell and utilities and a large number of additional utilities are included.
- "C Language".
- "Sockets".
- "Transport Service XTI".
- "Internationalized Terminal Interfaces". New extensions are color and multi-byte support.

XPG4 Base 95 contains a subset of the above.

XPG4 feature test macros: As for POSIX.1 XPG4 systems add their own symbols to `unistd.h`. The example for XPG4 identification follows:

```
#define _XOPEN_SOURCE 1      /* need Base 95 */
#define _XOPEN_SOURCE_EXTENDED 1 /* and UNIX */
#include <unistd.h>
#if _XOPEN_VERSION == 4
  /* It is a XPG4 system */
#ifdef _XOPEN_UNIX
  /* It is a branded XPG4 UNIX system */
...
#endif
#endif
```

Current status: Linux is currently far from being complete with respect to XPG4. Major missing components are extended (internationalized) regular expressions and STREAMS.

Furthermore XPG4 development is expensive. There are no freely available test suites (as for example the NIST PCTS POSIX.1 test suite, see in the resources section). As a prerequisite for getting a license of X/Open's VSX test suite an X/Open membership would be required to brand Linux.

In May 1996 Caldera Inc. announced a very promising claim to brand Linux as XPG4 UNIX in the long term, but as far as I know there is no mentioning of it anymore.

Conclusion

I have been trying to line out the most relevant standards with respect to Linux. Linux' remarkable success is surely based on its commitment to standards and will continue as long as standards are being supported and integrated. New developments on the horizon are real-time extensions and threads as defined by the new POSIX.1 release. With these new features and the current strong POSIX.1 basis Linux will enable growth to more industrial and office applications. These kinds of applications are crucial in order to make Linux more acceptable. Linux so far has excelled in performance and portability. If the dynamic development succeeds in combining innovation and integration of standards, the future will be bright. As Linux continues to grow even a future XPG4 UNIX branding is likely to happen.

Resources

Standard C: The "Rationale for American National Standard for Information Systems- Programming Language C" is freely available via ftp:
`ftp://ftp.uu.net/doc/standards/ansi/X3.159-1989/rationale.latex.tar.Z`.

The rationale has also available in printed form[26].

Newsgroups for discussion of standard C and internationalization matters are `comp.std.c` rsp. `comp.std.internat`.

Standard C defect reports and technical corrigenda are available here:
`ftp://ftp.dmk.com/DMK/sc22wg14/rr`
`http://www.dknet.dk/JTC1/SC22/WG14/`.

Commercial compiler test suites are available from Plum Hall.
`http://www.plumhall.com/`.

The POSIX family: Markus Kuhn did a nice Linux status report regarding POSIX.1b (real-time) features [25]. It is available via ftp:
`ftp.informatik.uni-erlangen.de/local/cip/mskuhn/misc/linux-posix.1b`.

The FIPS-151-2 standard: NIST has specified the FIPS-151-2 Federal Information Processing Standard based on POSIX.1. The POSIX FIPS Conformance Test Suite (PCTS) was made freely available by NIST in fall 1996. It is available via
`ftp://tsx-11.mit.edu/pub/linux/sources/test_suites/NIST-PCTS.tar.gz`.

POSIX and other OS standards are discussed in the newsgroup `comp.std.unix`.

X/Open documents: The X/Open specifications are available in paper form in five volumes from X/Open company. X/Open also license their test suites for branding XPG4 to X/Open members. A non authoritative book with the complete XPG4 UNIX specifications on an accompanying CD-ROM is also available [27].

Kernel extensions: The real-time extension from Yodaiken et al. are available via
`http://luz.cs.nmt.edu/~rtlinux/`.

The POSIX.1c clone() based pthreads package from Xavier Leroy is available via
`http://pauillac.inria.fr/~xleroy/linuxthreads/`.

Documentation for Linux threads in html format is available via
`http://sunsite.unc.edu/pub/Linux/docs/faqs/Threads-FAQ/html/`

Trademarks: X/Open is a registered trade mark of X/Open Company Limited. UNIX is a registered trade mark of X/Open Company Limited. System V is a registered trade mark of AT&T Bell. BSD is a registered trade mark of Berkeley Software Distribution. Windows NT is a registered trade mark of Microsoft Corporation.

References

[1] Don Libes. *Obfuscated C and other Mysteries.* John Wiley & Sons, Inc., 1993, ISBN 0-471-57805-3.

[2] ISO/IEC. *ISO/IEC 9899:1990, Programming Languages — C.* ISO/IEC, 1990. Definition of standard C.

[3] IEEE/ISO/IEC. *ISO/IEC 9945-1:1990 IEEE Std. 1003.1-1996 Information Technology Portable Operating System Interface (POSIX) Part 1 System Application Program Interface (API) [C Language].* IEEE/ISO/IEC, 1996, ISBN 1-55937-573-0. POSIX.1 POSIX documents contact address: stds.info@ieee.org, http://stdsbbs.ieee.org, http://www.knosof.co.uk.

[4] *Draft POSIX.1a 1995 Draft revision to POSIX.1*, 1995. Contains symbolic links extensions.

[5] IEEE/ISO/IEC. *ISO/IEC IEEE Std. 1003.1b-1993 realtime extensions.* IEEE/ISO/IEC, 1993, ISBN 1-55937-375-X. Formerly POSIX.4, now included in 1003.1-1996.

[6] IEEE. *IEEE Std. 1003.1c-1995 thread extensions.* IEEE, 1995, ISBN 1-55937-375-X. Formerly POSIX.4a, now included in 1003.1-1996.

[7] *Draft 1003.1e 1995 Protection Audit & Control Interface*, 1995. Formerly POSIX.6, contains ACLs and audit features for TCSEC.

[8] *Draft 1003.1g 1995 Protocol Independent Interfaces*, 1995. Formerly called POSIX.12 contains socket interface.

[9] *Draft 1003.1i 1995 Amendment i Realtime Extensions*, 1995. Technical corrections to 1003.1b, now included in 1003.1-1996.

[10] IEEE/ISO/IEC. *ISO/IEC 9945-2:1993 IEEE Std. 1003.2-1992 Information Technology Portable Operating System Interface (POSIX) Part 2 Shell and Utilities Volume 1/2.* IEEE/ISO/IEC, 1992, ISBN 1-55937-255-9. POSIX.2 includes POSIX.2a (User Portability Extension).

[11] *Draft 1003.2b 1995 Shell & Utilities Amendment 2*, 1995. Symbolic link extensions.

[12] *Draft 1003.2c 1995 Protection & Control Interface*, 1995. Security extensions.

[13] IEEE. *IEEE Std. 1003.2d-1994 Shell and Utilities Amendment 1: Batch Environment.* IEEE, 1994, ISBN 1-55937-512-4. Batch queueing extensions.

[14] IEEE/ISO/IEC. *ISO/IEC 13210:1994 IEEE Std. 1003.3-1991 Test methods.* IEEE/ISO/IEC, 1991, ISBN 1-55937-494-2. Testing and verification requirements.

[15] IEEE/ISO/IEC. *ISO/IEC 14515-1 IEEE Std. 2003.1-1992 Test methods for part 1.* IEEE/ISO/IEC, 1992, ISBN 1-55937-275-3.

[16] *Draft 2003-1996 Test Methods*, 1996.

[17] *Draft 2003.1b-1996 Test Methods Part 1: System application*, 1996.

[18] *Approved Draft 2003.2-1996 Test Methods Part 2 Shell and Utilities*, 1996.

[19] X/Open. *X/Open System and Branded Products XPG4, Vol. 1 System Interface Definition.* X/Open Company, 1992, ISBN 1-85912-036-9. Contact: XoPubs@xopen.org
WWW http://www.xopen.org.

[20] X/Open. *X/Open System and Branded Products XPG4, Vol. 2 System Interfaces and Headers.* X/Open Company, 1992, ISBN 1-85912-037-7.

[21] X/Open. *X/Open System and Branded Products XPG4, Vol. 3 Command and Utilities.* X/Open Company, 1992, ISBN 1-85912-034-2.

[22] X/Open. *X/Open System and Branded Products XPG4, Vol. 4 X/Open Curses.* X/Open Company, 1992, ISBN 1-85912-077-6.

[23] X/Open. *X/Open System and Branded Products XPG4, Vol. 5 Networking Services.* X/Open Company, 1992, ISBN 1-85912-049-0.

[24] Unifix Software GmbH. *Unifix Linux 2.0.* Unifix Software GmbH, 1996, ISBN to be determined. Contact: http://www.unifix.de.

[25] Markus Kuhn. A Vision for Linux 2.2 — POSIX.1b Compatibility and Real-Time Support, 1996. Linux status report regarding POSIX.1b available via ftp:
ftp.informatik.uni-erlangen.de/local
/cip/mskuhn/misc/linux-posix.1b.

[26] ANSI. *Rationale for American National Standard for Information Systems— Programming Language C.* Silicon Press, 1989, ISBN 0-929306-07-4.

[27] Stephen Walli. *Go Solo, How to Implement and Go Solo with the Single UNIX Specification.* Prentice Hall, 1995, ISBN 0-13-439381-3. Includes a CD-ROM with SPEC 1170 specifications in a proprietary SGML format and a Win 3.11 browser, but alas none for Linux yet.

[28] ISO/IEC. *ISO/IEC 9899:1995/Amendment 1, C Integrity.* ISO/IEC, 1995.

[29] P.J. Plauger. *The Standard C library.* Prentice Hall, 1992, ISBN 0-13-131509-9. Discusses the standard C library and contains a sample implementation of a conforming standard C library.

[30] *Draft 1003.1j 1995 Amendment j Additional Realtime Extensions*, 1995.

[31] IEEE/ISO/IEC. *ISO/IEC DIS 14519-1 IEEE Std. 1003.5-1992 ADA language binding.* IEEE/ISO/IEC, 1992, ISBN 1-55937-229-X.

[32] *Approved Draft 1003.5b-1996 ADA Realtime extensions*, 1996.

[33] IEEE. *IEEE Std. 1003.9-1992 FORTRAN language binding.* IEEE, 1992, ISBN 1-55937-230-3.

[34] IEEE. *IEEE Std. 1003.10-1995 Supercomputing AEP.* IEEE, 1995, ISBN 1-55937-546-9.

[35] IEEE. *IEEE Std. 1224.2-1993 network directory services interface.* IEEE, 1993. P1224.2 (formerly POSIX.17).

[36] W. Richard Stevens. *Advanced Programming in the Unix Environment.* Addison Wesley, 1992, ISBN 0-201-56317-7.

[37] Donald Lewine. *POSIX Programmer Guide.* O'Reilly & Associates, Inc., 1991, ISBN 0-937175-73-0.

[38] Bill O. Gallmeister. *POSIX.4, Programming for the Real World.* O'Reilly & Associates, Inc., 1995, ISBN 1-56592-074-0.