The Open Group Standard

Additional APIs for the Base Specifications Issue 8, Part 1

This document is provided "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

Any publication of The Open Group may include technical inaccuracies or typographical errors. Changes may be periodically made to these publications; these changes will be incorporated in new editions of these publications. The Open Group may make improvements and/or changes in the products and/or the programs described in these publications at any time without notice.

Should any viewer of this document respond with information including feedback data, such as questions, comments, suggestions, or the like regarding the content of this document, such information shall be deemed to be non-confidential and The Open Group shall have no obligation of any kind with respect to such information and shall be free to reproduce, use, disclose, and distribute the information to others without limitation. Further, The Open Group shall be free to use any ideas, concepts, know-how, or techniques contained in such information for any purpose whatsoever including but not limited to developing, manufacturing, and marketing products incorporating such information.

If you did not obtain this copy through The Open Group, it may not be the latest version. For your convenience, the latest version of this publication may be downloaded at www.opengroup.org/library.

The Open Group Standard

Additional APIs for the Base Specifications Issue 8, Part 1

ISBN: TBA

Document Number: TBA

Published by The Open Group, <Month Year>.

Comments relating to the material contained in this document may be submitted to:

The Open Group, Apex Plaza, Forbury Road, Reading, Berkshire, RG1 1AX, United Kingdom

or by electronic mail to:

ogspecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through technology standards. Our diverse membership of more than 800 organizations includes customers, systems and solutions suppliers, tools vendors, integrators, academics, and consultants across multiple industries.

The mission of The Open Group is to drive the creation of Boundaryless Information Flow™ achieved by:

- Working with customers to capture, understand, and address current and emerging requirements, establish policies, and share best practices
- Working with suppliers, consortia, and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies
- Offering a comprehensive set of services to enhance the operational efficiency of consortia
- Developing and operating the industry's premier certification service and encouraging procurement of certified products

Further information on The Open Group is available at www.opengroup.org.

The Open Group publishes a wide range of technical documentation, most of which is focused on development of Standards and Guides, but which also includes white papers, technical studies, certification and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/library.

This Document

This document has been prepared by The Open Group Base Working Group. The Open Group Base Working Group is considering submitting a number of additional APIs to the Austin Group as input to the Issue 8 revision of the Base Specifications.

This document contains the first set of these APIs.

Trademarks

ArchiMate, DirecNet, Making Standards Work, Open O logo, Open O and Check Certification logo, Platform 3.0, The Open Group, TOGAF, UNIX, UNIXWARE, and the Open Brand X logo are registered trademarks and Boundaryless Information Flow, Build with Integrity Buy with Confidence, Commercial Aviation Reference Architecture, Dependability Through Assuredness, Digital Practitioner Body of Knowledge, DPBoK, EMMM, FACE, the FACE logo, FHIM Profile Builder, the FHIM logo, FPB, Future Airborne Capability Environment, IT4IT, the IT4IT logo, O-AA, O-DEF, O-HERA, O-PAS, Open Agile Architecture, Open FAIR, Open Footprint, Open Process Automation, Open Subsurface Data Universe, Open Trusted Technology Provider, OSDU, Sensor Integration Simplified, SOSA, and the SOSA logo are trademarks of The Open Group.

All other brands, company, and product names are used for identification purposes only and may be trademarks that are the sole property of their respective owners.

Acknowledgements

The Open Group gratefully acknowledges the contribution of the following in the development of this document:

- The Open Group Base Working Group
- The Austin Group

The Open Group gratefully acknowledges the following reviewers who participated in the Company Review of this document:

- Eric Blake
- Geoff Clare
- Mike Crowe
- Chris Frost
- Jens Kjærby
- Curtis Smith
- Dennis Wölfing

1 Introduction

1.1 Scope

The purpose of this document is to define a set of additional APIs for inclusion in the Issue 8 revision of the Base Specifications of the Single UNIX Specification.

The additional APIs proposed by participants in the Austin Group that The Open Group has agreed to sponsor are as follows:

dladdr()
getentropy()
getlocalename_l()
memmem()
posix_getdents()
ppoll()
pthread_cond_clockwait()
pthread_mutex_clocklock()
pthread_rwlock_clockrdlock()
pthread_rwlock_clockwrlock()
qsort_r()
reallocarray()
sem_clockwait()
sig2str()
str2sig()
strlcat()
strlcpy()
wcslcat()
wcslcpy()

1.2 Relationship to Other Formal Standards

This Standard is being forwarded to the Austin Group for consideration as input to the Issue 8 revision of the Base Specifications.

2 Application Program Interfaces

The following pages are extracted from a complete draft of the Base Specifications in which the proposed changes have been applied, with change bars showing the differences from Issue 8 draft 1.1. Only pages with technical changes are included – editorial changes such as additions to SEE ALSO and CHANGE HISTORY sections have been omitted (unless they appear on the same page as a technical change). The complete draft is also being made available for reference.

2.1 Change Bars

Changed lines are marked with a '|' in the right-hand margin, new lines with a '+', and deleted lines with a '-'.

Note that sometimes the placement of change bars is slightly inaccurate. In particular, changes may extend into a line following a set of change-barred lines. Also, changes within tables do not have change bars.

2.2 Reference Pages

The reference pages for the new functions and related header additions follow.

4.13 Memory Synchronization

Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads. The following functions synchronize memory with respect to other threads:

fork( )
pthread_barrier_wait( )
pthread_cond_broadcast( )
pthread_cond_clockwait( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_create( )
pthread_join( )
pthread_mutex_clocklock( )
pthread_mutex_lock( )
pthread_mutex_timedlock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
pthread_spin_lock( )
pthread_spin_trylock( )
pthread_spin_unlock( )
pthread_rwlock_clockrdlock( )
pthread_rwlock_clockwrlock( )
pthread_rwlock_rdlock( )
pthread_rwlock_timedrdlock( )
pthread_rwlock_timedwrlock( )
pthread_rwlock_tryrdlock( )
pthread_rwlock_trywrlock( )
pthread_rwlock_unlock( )
pthread_rwlock_wrlock( )
sem_clockwait( )
sem_post( )
sem_timedwait( )
sem_trywait( )
sem_wait( )
semctl( )
semop( )
wait( )
waitpid( )

The *pthread_once*( ) function shall synchronize memory for the first call in each thread for a given **pthread_once_t** object. If the *init_routine* called by *pthread_once*( ) is a cancellation point and is canceled, a call to *pthread_once*( ) for the same **pthread_once_t** object made from a cancellation cleanup handler shall also synchronize memory.

The *pthread_mutex_lock*( ) function need not synchronize memory if the mutex type is PTHREAD_MUTEX_RECURSIVE and the calling thread already owns the mutex. The *pthread_mutex_unlock*( ) function need not synchronize memory if the mutex type is PTHREAD_MUTEX_RECURSIVE and the mutex has a lock count greater than one.

Unless explicitly stated otherwise, if one of the above functions returns an error, it is unspecified whether the invocation causes memory to be synchronized.

Applications may allow more than one thread of control to read a memory location simultaneously.

4.14 Pathname Resolution

Pathname resolution is performed for a process to resolve a pathname to a particular directory entry for a file in the file hierarchy. There may be multiple pathnames that resolve to the same directory entry, and multiple directory entries for the same file. When a process resolves a pathname of an existing directory entry, the entire pathname shall be resolved as described below. When a process resolves a pathname of a directory entry that is to be created immediately after the pathname is resolved, pathname resolution terminates when all components of the path prefix of the last component have been resolved. It is then the responsibility of the process to create the final component.

Each filename in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment **a/b**, file **b** is located in directory **a**). Pathname resolution shall fail if this cannot be accomplished. If the pathname begins with a <slash>, the predecessor of the first filename in the pathname shall be taken to be the root directory of the process (such pathnames are referred to as ``absolute pathnames''). If the pathname does not begin with a <slash>, the predecessor of the first filename of the pathname shall be taken to be either the current working directory of the process or for certain interfaces the directory identified by a file

NAME

dirent.h — format of directory entries

SYNOPSIS

```
#include <dirent.h>
```

DESCRIPTION

The internal format of directories is unspecified.

The **<dirent.h>** header shall define the following type:

DIR A type representing a directory stream. The **DIR** type may be an incomplete type.

It shall also define the structure **dirent** which shall include the following members:

`ino_t`	`d_ino`	File serial number.
`char`	`d_name[]`	Filename string of entry.

and the structure **posix_dent** which shall include the following members:

`ino_t`	`d_ino`	File serial number.
`reclen_t`	`d_reclen`	Length of this entry, including trailing padding if necessary. See *posix_getdents*( ).
`unsigned char`	`d_type`	File type or unknown-file-type indication.
`char`	`d_name[]`	Filename string of this entry.

The array *d_name* in each of these structures is of unspecified size, but shall contain a filename of at most {NAME_MAX} bytes followed by a terminating null byte.

The **<dirent.h>** header shall define the **ino_t**, **reclen_t**, **size_t**, and **ssize_t** types as described in **<sys/types.h>**.

The **<dirent.h>** header shall define the following symbolic constants for the file types and unknown-file-type indicator returned in the *d_type* member of the **posix_dent** structure. The values shall be distinct and shall be suitable for use in **#if** preprocessing directives:

DT_BLK	Block special.
DT_CHR	Character special.
DT_DIR	Directory.
DT_FIFO	FIFO special.
DT_LNK	Symbolic link.
DT_REG	Regular.
DT_SOCK	Socket.
DT_UNKNOWN	Unknown file type.

TYM The implementation may implement message queues, semaphores, shared memory objects or typed memory objects as distinct file types. The following macros shall be provided to represent these types. The values shall be distinct from each other and from the above symbolic constants beginning with DT_, except when a distinct file type is not implemented, in which case the corresponding constant shall have a value that is never returned in *d_type* by *posix_getdents*( ). The values shall be suitable for use in **#if** preprocessing directives:

DT_MQ	Message queue.

DT_SEM	Semaphore.
DT_SHM	Shared memory object.
DT_TMO	Typed memory object.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int            alphasort(const struct dirent **, const struct dirent **);
int            closedir(DIR *);
int            dirfd(DIR *);
DIR           *fdopendir(int);
DIR           *opendir(const char *);
ssize_t        posix_getdents(int, void *, size_t, int);
struct dirent *readdir(DIR *);
int            readdir_r(DIR *restrict, struct dirent *restrict,
                   struct dirent **restrict);
void           rewinddir(DIR *);
int            scandir(const char *, struct dirent ***,
                   int (*)(const struct dirent *),
                   int (*)(const struct dirent **,
                   const struct dirent **));
void           seekdir(DIR *, long);
long           telldir(DIR *);
```

APPLICATION USAGE

None.

RATIONALE

Information similar to that in the **<dirent.h>** header is contained in a file **<sys/dir.h>** in 4.2 BSD and 4.3 BSD. The equivalent in these implementations of **struct dirent** from this volume of POSIX.1-202x is **struct direct**. The filename was changed because the name **<sys/dir.h>** was also used in earlier implementations to refer to definitions related to the older access method; this produced name conflicts. The name of the structure was changed because this volume of POSIX.1-202x does not completely define what is in the structure, so it could be different on some implementations from **struct direct**.

The **posix_dent** structure was based on existing structures used by traditional *getdents*() functions, but the name was changed because the existing structures differed in name and in their members. Some used the **dirent** structure but this is not required to include a *d_type* member, which is the main advantage of using *posix_getdents*() over *readdir*(). The *d_reclen* member was included, even though some implementations return fixed-length entries and therefore do not need it, as almost all existing code that used *getdents*() used *d_reclen* to iterate through the returned entries. Implementations that return fixed-length entries can simply set *d_reclen* to that length in *posix_getdents*(). The type **reclen_t** for *d_reclen* was introduced, instead of using **unsigned short**, so as not to create a requirement that {NAME_MAX} cannot be greater than (a value somewhat smaller than) {SHRT_MAX}.

Implementations are encouraged to define a DT_FORCE_TYPE symbolic constant for use in the *flags* argument to *posix_getdents*(). See the RATIONALE for *posix_getdents*().

The name of an array of **char** of an unspecified size should not be used as an lvalue. Use of:

```
sizeof(d_name)
```

is incorrect; use:

```
strlen(d_name)
```

instead.

The array of **char** *d_name* cannot be assumed to have a fixed size. Implementations may define the *d_name* array in the **dirent** and **posix_dent** structures to have size 1, or size greater than {NAME_MAX}, or use a flexible array member, but in all cases the actual number of characters used for *d_name* is at least the length of the filename string including the terminating NUL byte.

FUTURE DIRECTIONS

A future version of this standard may add a DT_FORCE_TYPE symbolic constant for use as described in the RATIONALE for *posix_getdents*( ).

SEE ALSO

<sys/types.h>

XSH *alphasort*( ), *closedir*( ), *dirfd*( ), *fdopendir*( ), *posix_getdents*( ), *readdir*( ), *rewinddir*( ), *seekdir*( ), *telldir*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r*( ).

The **restrict** keyword is added to the prototype for *readdir_r*( ).

Issue 7

The *alphasort*( ), *dirfd*( ), and *scandir*( ) functions are added from The Open Group Technical Standard, 2006, Extended API Set Part 1.

The *fdopendir*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

Austin Group Interpretation 1003.1-2001 #110 is applied, clarifying the definition of the **DIR** type.

POSIX.1-2008, Technical Corrigendum 1, XBD/TC1-2008/0039 [291], XBD/TC1-2008/0040 [291], XBD/TC1-2008/0041 [291], and XBD/TC1-2008/0042 [206] are applied.

Issue 8

Austin Group Defect 697 is applied, adding *posix_getdents*( ).

NAME

dlfcn.h — dynamic linking

SYNOPSIS

```
#include <dlfcn.h>
```

DESCRIPTION

The **<dlfcn.h>** header shall define the **Dl_info_t** structure type, which shall include at least the following members:

`const char *dli_fname`	Pathname of mapped object file.
`void       *dli_fbase`	Base of mapped address range.
`const char *dli_sname`	Symbol name or null pointer.
`void       *dli_saddr`	Symbol address or null pointer.

The **<dlfcn.h>** header shall define at least the following symbolic constants for use in the construction of a *dlopen*( ) *mode* argument:

RTLD_LAZY	Relocations are performed at an implementation-defined time.
RTLD_NOW	Relocations are performed when the object is loaded.
RTLD_GLOBAL	All symbols are available for relocation processing of other modules.
RTLD_LOCAL	All symbols are not made available for relocation processing by other modules.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    dladdr(const void *restrict, Dl_info_t *restrict);
int    dlclose(void *);
char  *dlerror(void);
void  *dlopen(const char *, int);
void  *dlsym(void *restrict, const char *restrict);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

XSH *dladdr*( ), *dlclose*( ), *dlerror*( ), *dlopen*( ), *dlsym*( )

CHANGE HISTORY

First released in Issue 5.

Issue 6

The **restrict** keyword is added to the prototype for *dlsym*( ).

Issue 7

The **<dlfcn.h>** header is moved from the XSI option to the Base.

This reference page is clarified with respect to macros and symbolic constants.

Other Invariant Values

The **<limits.h>** header shall define the following symbolic constants:

{GETENTROPY_MAX}
The maximum value of the *length* argument in calls to the *getentropy*( ) function.
Minimum Acceptable Value: 256

{NL_ARGMAX}
Maximum value of *n* in conversion specifications using the `"%n$"` sequence in calls to the *printf*( ) and *scanf*( ) families of functions.
Minimum Acceptable Value: 9

XSI {NL_LANGMAX}
Maximum number of bytes in a *LANG* name.
Minimum Acceptable Value: 14

{NL_MSGMAX}
Maximum message number.
Minimum Acceptable Value: 32 767

{NL_SETMAX}
Maximum set number.
Minimum Acceptable Value: 255

{NL_TEXTMAX}
Maximum number of bytes in a message string.
Minimum Acceptable Value: {_POSIX2_LINE_MAX}

{NSIG_MAX}
Maximum possible return value of *sysconf*(_SC_NSIG). See XSH *sysconf*( ). The value of {NSIG_MAX} shall be no greater than the number of signals that the **sigset_t** type (see **<signal.h>**) is capable of representing, ignoring any restrictions imposed by *sigfillset*( ) or *sigaddset*( ).

XSI {NZERO}
Default process priority.
Minimum Acceptable Value: 20

APPLICATION USAGE

None.

RATIONALE

A request was made to reduce the value of {_POSIX_LINK_MAX} from the value of 8 specified for it in the POSIX.1-1990 standard to 2. The standard developers decided to deny this request for several reasons:

- They wanted to avoid making any changes to the standard that could break conforming applications, and the requested change could have that effect.
- The use of multiple hard links to a file cannot always be replaced with use of symbolic links. Symbolic links are semantically different from hard links in that they associate a pathname with another pathname rather than a pathname with a file. This has implications for access control, file permanence, and transparency.
- The original standard developers had considered the issue of allowing for implementations that did not in general support hard links, and decided that this would reduce consensus on the standard.

CX The **<locale.h>** header shall contain at least the following macros representing bitmasks for use with the *newlocale*( ) function for each supported locale category:

LC_COLLATE_MASK
LC_CTYPE_MASK
LC_MESSAGES_MASK
LC_MONETARY_MASK
LC_NUMERIC_MASK
LC_TIME_MASK

In addition, a macro to set the bits for all categories set shall be defined:

LC_ALL_MASK

The **<locale.h>** header shall define LC_GLOBAL_LOCALE, a special locale object descriptor used by the *duplocale*( ) and *uselocale*( ) functions.

The **<locale.h>** header shall define the **locale_t** type, representing a locale object.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
CX  locale_t      duplocale(locale_t);
    void          freelocale(locale_t);
    const char   *getlocalename_l(int, locale_t);
    struct lconv *localeconv(void);
CX  locale_t      newlocale(int, const char *, locale_t);
    char         *setlocale(int, const char *);
CX  locale_t      uselocale (locale_t);
```

APPLICATION USAGE

None.

RATIONALE

It is suggested that each category macro name for use in *setlocale*( ) have a corresponding macro name ending in *_MASK* for use in *newlocale*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

Chapter 8 (on page 153), **<stddef.h>**

XSH *duplocale*( ), *freelocale*( ), *getlocalename_l*( ), *localeconv*( ), *newlocale*( ), *setlocale*( ), *uselocale*( )

CHANGE HISTORY

First released in Issue 3.

Included for alignment with the ISO C standard.

Issue 6

The **lconv** structure is expanded with new members (**int_n_cs_precedes**, **int_n_sep_by_space**, **int_n_sign_posn**, **int_p_cs_precedes**, **int_p_sep_by_space**, and **int_p_sign_posn**) for alignment with the ISO/IEC 9899: 1999 standard.

Extensions beyond the ISO C standard are marked.

NAME

poll.h — definitions for the poll( ) function

SYNOPSIS

```
#include <poll.h>
```

DESCRIPTION

The **<poll.h>** header shall define the **pollfd** structure, which shall include at least the following members:

`int`	`fd`	The following descriptor being polled.
`short`	`events`	The input event flags (see below).
`short`	`revents`	The output event flags (see below).

The **<poll.h>** header shall define the following type through **typedef**:

nfds_t	An unsigned integer type used for the number of file descriptors.

The implementation shall support one or more programming environments in which the width of **nfds_t** is no greater than the width of type **long**. The names of these programming environments can be obtained using the *confstr*( ) function or the *getconf* utility.

The **<poll.h>** header shall define the **sigset_t** type as described in **<signal.h>**.

The **<poll.h>** header shall define the **timespec** structure as described in **<time.h>**.

The **<poll.h>** header shall define the following symbolic constants, zero or more of which may be OR'ed together to form the *events* or *revents* members in the **pollfd** structure:

POLLIN	Data other than high-priority data may be read without blocking.
POLLRDNORM	Normal data may be read without blocking.
POLLRDBAND	Priority data may be read without blocking.
POLLPRI	High priority data may be read without blocking.
POLLOUT	Normal data may be written without blocking.
POLLWRNORM	Equivalent to POLLOUT.
POLLWRBAND	Priority data may be written.
POLLERR	An error has occurred (*revents* only).
POLLHUP	Device has been disconnected (*revents* only).
POLLNVAL	Invalid *fd* member (*revents* only).

The significance and semantics of normal, priority, and high-priority data are file and device-specific.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int   poll(struct pollfd [], nfds_t, int);
int   ppoll(struct pollfd [], nfds_t, const struct timespec *restrict,
          const sigset_t *restrict);
```

Inclusion of the **<poll.h>** header may make visible all symbols from the headers **<signal.h>** and **<time.h>**.

```
          int   pthread_atfork(void (*)(void), void (*)(void),
                    void(*)(void));
          int   pthread_attr_destroy(pthread_attr_t *);
          int   pthread_attr_getdetachstate(const pthread_attr_t *, int *);
          int   pthread_attr_getguardsize(const pthread_attr_t *restrict,
                    size_t *restrict);
TPS       int   pthread_attr_getinheritsched(const pthread_attr_t *restrict,
                    int *restrict);
          int   pthread_attr_getschedparam(const pthread_attr_t *restrict,
                    struct sched_param *restrict);
TPS       int   pthread_attr_getschedpolicy(const pthread_attr_t *restrict,
                    int *restrict);
          int   pthread_attr_getscope(const pthread_attr_t *restrict,
                    int *restrict);
TSA TSS   int   pthread_attr_getstack(const pthread_attr_t *restrict,
                    void **restrict, size_t *restrict);
TSS       int   pthread_attr_getstacksize(const pthread_attr_t *restrict,
                    size_t *restrict);
          int   pthread_attr_init(pthread_attr_t *);
          int   pthread_attr_setdetachstate(pthread_attr_t *, int);
          int   pthread_attr_setguardsize(pthread_attr_t *, size_t);
TPS       int   pthread_attr_setinheritsched(pthread_attr_t *, int);
          int   pthread_attr_setschedparam(pthread_attr_t *restrict,
                    const struct sched_param *restrict);
TPS       int   pthread_attr_setschedpolicy(pthread_attr_t *, int);
          int   pthread_attr_setscope(pthread_attr_t *, int);
TSA TSS   int   pthread_attr_setstack(pthread_attr_t *, void *, size_t);
TSS       int   pthread_attr_setstacksize(pthread_attr_t *, size_t);
          int   pthread_barrier_destroy(pthread_barrier_t *);
          int   pthread_barrier_init(pthread_barrier_t *restrict,
                    const pthread_barrierattr_t *restrict, unsigned);
          int   pthread_barrier_wait(pthread_barrier_t *);
          int   pthread_barrierattr_destroy(pthread_barrierattr_t *);
TSH       int   pthread_barrierattr_getpshared(
                    const pthread_barrierattr_t *restrict, int *restrict);
          int   pthread_barrierattr_init(pthread_barrierattr_t *);
TSH       int   pthread_barrierattr_setpshared(pthread_barrierattr_t *, int);
          int   pthread_cancel(pthread_t);
          int   pthread_cond_broadcast(pthread_cond_t *);
          int   pthread_cond_clockwait(pthread_cond_t *restrict,                 +
                    pthread_mutex_t *restrict, clockid_t,                        +
                    const struct timespec *restrict);                            +
          int   pthread_cond_destroy(pthread_cond_t *);
          int   pthread_cond_init(pthread_cond_t *restrict,
                    const pthread_condattr_t *restrict);
          int   pthread_cond_signal(pthread_cond_t *);
          int   pthread_cond_timedwait(pthread_cond_t *restrict,
                    pthread_mutex_t *restrict, const struct timespec *restrict);
          int   pthread_cond_wait(pthread_cond_t *restrict,
                    pthread_mutex_t *restrict);
          int   pthread_condattr_destroy(pthread_condattr_t *);
          int   pthread_condattr_getclock(const pthread_condattr_t *restrict,
                    clockid_t *restrict);
TSH      int    pthread_condattr_getpshared(const pthread_condattr_t *restrict,
                    int *restrict);
         int    pthread_condattr_init(pthread_condattr_t *);
         int    pthread_condattr_setclock(pthread_condattr_t *, clockid_t);
TSH      int    pthread_condattr_setpshared(pthread_condattr_t *, int);
         int    pthread_create(pthread_t *restrict, const pthread_attr_t *restrict,
                    void *(*)(void*), void *restrict);
         int    pthread_detach(pthread_t);
         int    pthread_equal(pthread_t, pthread_t);
         void   pthread_exit(void *);
TCT      int    pthread_getcpuclockid(pthread_t, clockid_t *);
TPS      int    pthread_getschedparam(pthread_t, int *restrict,
                    struct sched_param *restrict);
         void *pthread_getspecific(pthread_key_t);
         int    pthread_join(pthread_t, void **);
         int    pthread_key_create(pthread_key_t *, void (*)(void*));
         int    pthread_key_delete(pthread_key_t);
         int    pthread_mutex_clocklock(pthread_mutex_t *restrict, clockid_t,      +
                    const struct timespec *restrict);                              +
         int    pthread_mutex_consistent(pthread_mutex_t *);
         int    pthread_mutex_destroy(pthread_mutex_t *);
RPP|TPP  int    pthread_mutex_getprioceiling(const pthread_mutex_t *restrict,
                    int *restrict);
         int    pthread_mutex_init(pthread_mutex_t *restrict,
                    const pthread_mutexattr_t *restrict);
         int    pthread_mutex_lock(pthread_mutex_t *);
RPP|TPP  int    pthread_mutex_setprioceiling(pthread_mutex_t *restrict, int,
                    int *restrict);
         int    pthread_mutex_timedlock(pthread_mutex_t *restrict,
                    const struct timespec *restrict);
         int    pthread_mutex_trylock(pthread_mutex_t *);
         int    pthread_mutex_unlock(pthread_mutex_t *);
         int    pthread_mutexattr_destroy(pthread_mutexattr_t *);
RPP|TPP  int    pthread_mutexattr_getprioceiling(
                    const pthread_mutexattr_t *restrict, int *restrict);
MC1      int    pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict,
                    int *restrict);
TSH      int    pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict,
                    int *restrict);
         int    pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict,
                    int *restrict);
         int    pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict,
                    int *restrict);
         int    pthread_mutexattr_init(pthread_mutexattr_t *);
RPP|TPP  int    pthread_mutexattr_setprioceiling(pthread_mutexattr_t *, int);
MC1      int    pthread_mutexattr_setprotocol(pthread_mutexattr_t *, int);
TSH      int    pthread_mutexattr_setpshared(pthread_mutexattr_t *, int);
         int    pthread_mutexattr_setrobust(pthread_mutexattr_t *, int);
         int    pthread_mutexattr_settype(pthread_mutexattr_t *, int);
         int    pthread_once(pthread_once_t *, void (*)(void));
         int    pthread_rwlock_destroy(pthread_rwlock_t *);
         int   pthread_rwlock_init(pthread_rwlock_t *restrict,
                   const pthread_rwlockattr_t *restrict);
         int   pthread_rwlock_clockrdlock(pthread_rwlock_t *restrict,           +
                   clockid_t, const struct timespec *restrict);                 +
         int   pthread_rwlock_clockwrlock(pthread_rwlock_t *restrict,           +
                   clockid_t, const struct timespec *restrict);                 +
         int   pthread_rwlock_rdlock(pthread_rwlock_t *);
         int   pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict,
                   const struct timespec *restrict);
         int   pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict,
                   const struct timespec *restrict);
         int   pthread_rwlock_tryrdlock(pthread_rwlock_t *);
         int   pthread_rwlock_trywrlock(pthread_rwlock_t *);
         int   pthread_rwlock_unlock(pthread_rwlock_t *);
         int   pthread_rwlock_wrlock(pthread_rwlock_t *);
         int   pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
TSH      int   pthread_rwlockattr_getpshared(
                   const pthread_rwlockattr_t *restrict, int *restrict);
         int   pthread_rwlockattr_init(pthread_rwlockattr_t *);
TSH      int   pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);
         pthread_t
               pthread_self(void);
         int   pthread_setcancelstate(int, int *);
         int   pthread_setcanceltype(int, int *);
TPS      int   pthread_setschedparam(pthread_t, int,
                   const struct sched_param *);
         int   pthread_setschedprio(pthread_t, int);
         int   pthread_setspecific(pthread_key_t, const void *);
         int   pthread_spin_destroy(pthread_spinlock_t *);
         int   pthread_spin_init(pthread_spinlock_t *, int);
         int   pthread_spin_lock(pthread_spinlock_t *);
         int   pthread_spin_trylock(pthread_spinlock_t *);
         int   pthread_spin_unlock(pthread_spinlock_t *);
         void  pthread_testcancel(void);
```

The following may be declared as functions, or defined as macros, or both. If functions are declared, function prototypes shall be provided.

pthread_cleanup_pop()
pthread_cleanup_push()

Inclusion of the **<pthread.h>** header shall make symbols defined in the headers **<sched.h>** and **<time.h>** visible.

NAME

semaphore.h — semaphores

SYNOPSIS

```
#include <semaphore.h>
```

DESCRIPTION

The **<semaphore.h>** header shall define the **sem_t** type, used in performing semaphore operations. The semaphore may be implemented using a file descriptor, in which case applications are able to open up at least a total of {OPEN_MAX} files and semaphores.

The **<semaphore.h>** header shall define the **timespec** structure as described in **<time.h>**.

The **<semaphore.h>** header shall define the symbolic constant SEM_FAILED which shall have type **sem_t ***.

The **<semaphore.h>** header shall define O_CREAT and O_EXCL as described in **<fcntl.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    sem_clockwait(sem_t *restrict, clockid_t,                +
           const struct timespec *restrict);                    +
int    sem_close(sem_t *);
int    sem_destroy(sem_t *);
int    sem_getvalue(sem_t *restrict, int *restrict);
int    sem_init(sem_t *, int, unsigned);
sem_t *sem_open(const char *, int, ...);
int    sem_post(sem_t *);
int    sem_timedwait(sem_t *restrict, const struct timespec *restrict);
int    sem_trywait(sem_t *);
int    sem_unlink(const char *);
int    sem_wait(sem_t *);
```

Inclusion of the **<semaphore.h>** header may make visible symbols defined in the **<fcntl.h>** and **<time.h>** headers.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<fcntl.h>, **<sys/types.h>**, **<time.h>**

XSH *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_unlink*( )

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The **<semaphore.h>** header is marked as part of the Semaphores option.

The Open Group Corrigendum U021/3 is applied, adding a description of SEM_FAILED.

The *sem_timedwait*( ) function is added for alignment with IEEE Std 1003.1d-1999.

The **sigval** union shall be defined as:

```
int     sival_int      Integer signal value.
void   *sival_ptr      Pointer signal value.
```

The **<signal.h>** header shall declare the SIGRTMIN and SIGRTMAX macros, which shall expand to positive integer expressions with type **int**, but which need not be constant expressions. These macros specify a range of signal numbers that are reserved for application use and for which the realtime signal behavior specified in this volume of POSIX.1-202x is supported. The signal numbers in this range do not overlap any of the signals specified in the following table.

The range SIGRTMIN through SIGRTMAX inclusive shall include at least {RTSIG_MAX} signal numbers. The value of SIGRTMAX shall be less than the value returned by *sysconf*(_SC_NSIG).

It is implementation-defined whether realtime signal behavior is supported for other signals. +

The **<signal.h>** header shall define the following symbolic constant. The value shall be suitable for use in **#if** preprocessing directives: +

SIG2STR_MAX Maximum size of a signal name returned by *sig2str*( ), including the terminating null byte. +

The **<signal.h>** header shall define the following macros that are used to refer to the signals that occur in the system. Signals defined here begin with the letters SIG followed by an uppercase letter. The macros shall expand to positive integer constant expressions with type **int** and distinct values CX less than the value of {NSIG_MAX} defined in **<limits.h>**. The value 0 is reserved for use as the null signal (see *kill*( )). Additional implementation-defined signals may occur in the system.

The ISO C standard only requires the signal names SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM to be defined. An implementation need not generate any of these six signals, except as a result of explicit use of interfaces that generate signals, such as *raise*( ), CX *kill*( ), the General Terminal Interface (see Section 11.1.9, on page 185), and the *kill* utility, unless otherwise stated (see, for example, XSH Section 2.8.3.3, on page 491).

The following signals shall be supported on all implementations (default actions are explained below the table):

CX In addition, the following signal-specific information shall be available:

Signal	Member	Value
SIGILL SIGFPE	**void** * *si_addr*	Address of faulting instruction.
SIGSEGV SIGBUS	**void** * *si_addr*	Address of faulting memory reference.
SIGCHLD	**pid_t** *si_pid* **int** *si_status*	Child process ID. If *si_code* is equal to CLD_EXITED, then *si_status* holds the exit value of the process; otherwise, it is equal to the signal that caused the process to change state. The exit value in *si_status* shall be equal to the full exit value (that is, the value passed to *_exit*( ), *_Exit*( ), or *exit*( ), or returned from *main*( )); it shall not be limited to the least significant eight bits of the value.
	uid_t *si_uid*	Real user ID of the process that sent the signal.

For some implementations, the value of *si_addr* may be inaccurate.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
CX   int    kill(pid_t, int);
XSI  int    killpg(pid_t, int);
CX   void   psiginfo(const siginfo_t *, const char *);
     void   psignal(int, const char *);
     int    pthread_kill(pthread_t, int);
     int    pthread_sigmask(int, const sigset_t *restrict,
                sigset_t *restrict);
     int    raise(int);
CX   int    sig2str(int, char *);                                      +
     int    sigaction(int, const struct sigaction *restrict,
                struct sigaction *restrict);
     int    sigaddset(sigset_t *, int);
XSI  int    sigaltstack(const stack_t *restrict, stack_t *restrict);
CX   int    sigdelset(sigset_t *, int);
     int    sigemptyset(sigset_t *);
     int    sigfillset(sigset_t *);
     int    sigismember(const sigset_t *, int);
     void (*signal(int, void (*)(int)))(int);
CX   int    sigpending(sigset_t *);
     int    sigprocmask(int, const sigset_t *restrict, sigset_t *restrict);
     int    sigqueue(pid_t, int, union sigval);
     int    sigsuspend(const sigset_t *);
     int    sigtimedwait(const sigset_t *restrict, siginfo_t *restrict,
                const struct timespec *restrict);
     int    sigwait(const sigset_t *restrict, int *restrict);
     int    sigwaitinfo(const sigset_t *restrict, siginfo_t *restrict);
     int    str2sig(const char *restrict, int *restrict);              +
```

CX Inclusion of the **<signal.h>** header may make visible all symbols from the **<time.h>** header.

```
        int           putenv(char *);
        void          qsort(void *, size_t, size_t, int (*)(const void *,
                          const void *));
CX      void          qsort_r(void *, size_t, size_t, int (*)(const void *,      +
                          const void *, void *), void *);                      +
        int           rand(void);
XSI     long          random(void);
        void         *realloc(void *, size_t);
CX      void         *reallocarray(void *, size_t, size_t);                    +
XSI     char         *realpath(const char *restrict, char *restrict);
        unsigned short *seed48(unsigned short [3]);
CX      int           setenv(const char *, const char *, int);
OB XSI  void          setkey(const char *);
XSI     char         *setstate(char *);
        void          srand(unsigned);
XSI     void          srand48(long);
        void          srandom(unsigned);
        double        strtod(const char *restrict, char **restrict);
        float         strtof(const char *restrict, char **restrict);
        long          strtol(const char *restrict, char **restrict, int);
        long double   strtold(const char *restrict, char **restrict);
        long long     strtoll(const char *restrict, char **restrict, int);
        unsigned long strtoul(const char *restrict, char **restrict, int);
        unsigned long long
                      strtoull(const char *restrict, char **restrict, int);
        int           system(const char *);
XSI     int           unlockpt(int);
CX      int           unsetenv(const char *);
        size_t        wcstombs(char *restrict, const wchar_t *restrict, size_t);
        int           wctomb(char *, wchar_t);
```

CX Inclusion of the **<stdlib.h>** header may also make visible all symbols from **<fcntl.h>**, **<limits.h>**, **<math.h>**, **<stddef.h>**, and **<sys/wait.h>**.

APPLICATION USAGE

None.

RATIONALE

The ISO C standard requires that `exit(EXIT_FAILURE)` returns ``unsuccessful termination status'' to the host environment. In a POSIX host environment this means that the lower 8 bits of EXIT_FAILURE must have at least one bit set. The standard developers decided to further restrict the allowed values for the following reasons:

- Exit statuses of 126, 127, and greater than 128 are ambiguous in certain circumstances because they have special meanings in the shell (see XCU Section 2.8.2, on page 2321).
- The *xargs* utility quits when a command execution exits with status 255 (see XCU *xargs*).
- Calling *exit*( ) with a value greater than 255 or less than 0 is something that only programs which are specifically designed to have their exit status obtained by *waitid*( ) should do (since it does not truncate the exit status to 8 bits). ``Pure ISO C'' programs that call `exit(EXIT_FAILURE)` do not meet this design criterion.

NAME

string.h — string operations

SYNOPSIS

```
#include <string.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see XSH Section 2.2, on page 460) to enable the visibility of these symbols in this header.

The **<string.h>** header shall define NULL and **size_t** as described in **<stddef.h>**.

CX The **<string.h>** header shall define the **locale_t** type as described in **<locale.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided for use with ISO C standard compilers.

```
XSI  void     *memccpy(void *restrict, const void *restrict, int, size_t);
     void     *memchr(const void *, int, size_t);
     int       memcmp(const void *, const void *, size_t);
     void     *memcpy(void *restrict, const void *restrict, size_t);
CX   void     *memmem(const void *, size_t, const void *, size_t);
     void     *memmove(void *, const void *, size_t);
     void     *memset(void *, int, size_t);
CX   char     *stpcpy(char *restrict, const char *restrict);
     char     *stpncpy(char *restrict, const char *restrict, size_t);
     char     *strcat(char *restrict, const char *restrict);
     char     *strchr(const char *, int);
     int       strcmp(const char *, const char *);
     int       strcoll(const char *, const char *);
CX   int       strcoll_l(const char *, const char *, locale_t);
     char     *strcpy(char *restrict, const char *restrict);
     size_t    strcspn(const char *, const char *);
CX   char     *strdup(const char *);
     char     *strerror(int);
CX   char     *strerror_l(int, locale_t);
     int       strerror_r(int, char *, size_t);
     size_t    strlcat(char *restrict, const char *restrict, size_t);
     size_t    strlcpy(char *restrict, const char *restrict, size_t);
     size_t    strlen(const char *);
     char     *strncat(char *restrict, const char *restrict, size_t);
     int       strncmp(const char *, const char *, size_t);
     char     *strncpy(char *restrict, const char *restrict, size_t);
CX   char     *strndup(const char *, size_t);
     size_t    strnlen(const char *, size_t);
     char     *strpbrk(const char *, const char *);
     char     *strrchr(const char *, int);
CX   char     *strsignal(int);
     size_t    strspn(const char *, const char *);
     char     *strstr(const char *, const char *);
     char     *strtok(char *restrict, const char *restrict);
CX   char     *strtok_r(char *restrict, const char *restrict, char **restrict);
     size_t    strxfrm(char *restrict, const char *restrict, size_t);
```

pthread_t	Used to identify a thread.
reclen_t	Used for directory entry lengths.
size_t	Used for sizes of objects.
ssize_t	Used for a count of bytes or an error indication.
suseconds_t	Used for time in microseconds.
time_t	Used for time in seconds.
timer_t	Used for timer ID returned by *timer_create*( ).
uid_t	Used for user IDs.

All of the types shall be defined as arithmetic types of an appropriate length, with the following exceptions:

pthread_attr_t
pthread_barrier_t
pthread_barrierattr_t
pthread_cond_t
pthread_condattr_t
pthread_key_t
pthread_mutex_t
pthread_mutexattr_t
pthread_once_t
pthread_rwlock_t
pthread_rwlockattr_t
pthread_spinlock_t
pthread_t
timer_t

Additionally:

- **mode_t** shall be an integer type.
- **dev_t** shall be an integer type.
- **nlink_t**, **uid_t**, **gid_t**, and **id_t** shall be integer types.
- **blkcnt_t** and **off_t** shall be signed integer types.
- **fsblkcnt_t**, **fsfilcnt_t**, **reclen_t**, and **ino_t** shall be defined as unsigned integer types.
- **size_t** shall be an unsigned integer type.
- **blksize_t**, **pid_t**, and **ssize_t** shall be signed integer types.
- CX **clock_t** shall be an integer or real-floating type. **time_t** shall be an integer type.

The type **ssize_t** shall be capable of storing values at least in the range [−1, {SSIZE_MAX}].

XSI The type **suseconds_t** shall be a signed integer type capable of storing values at least in the range [−1, 1 000 000].

The implementation shall support one or more programming environments in which the widths of **blksize_t**, **pid_t**, **size_t**, **ssize_t**, and **suseconds_t** are no greater than the width of type **long**. The names of these programming environments can be obtained using the *confstr*( ) function or the *getconf* utility.

There are no defined comparison or assignment operators for the following types:

```
        int          dup2(int, int);
        int          dup3(int, int, int);
        void         _exit(int);
OB XSI  void         encrypt(char [64], int);
        int          execl(const char *, const char *, ...);
        int          execle(const char *, const char *, ...);
        int          execlp(const char *, const char *, ...);
        int          execv(const char *, char *const []);
        int          execve(const char *, char *const [], char *const []);
        int          execvp(const char *, char *const []);
        int          faccessat(int, const char *, int, int);
        int          fchdir(int);
        int          fchown(int, uid_t, gid_t);
        int          fchownat(int, const char *, uid_t, gid_t, int);
SIO     int          fdatasync(int);
        int          fexecve(int, char *const [], char *const []);
        pid_t        _Fork(void);
        pid_t        fork(void);
        long         fpathconf(int, int);
FSC     int          fsync(int);
        int          ftruncate(int, off_t);
        char        *getcwd(char *, size_t);
        gid_t        getegid(void);
        int          getentropy(void *, size_t);                          +
        uid_t        geteuid(void);
        gid_t        getgid(void);
        int          getgroups(int, gid_t []);
XSI     long         gethostid(void);
        int          gethostname(char *, size_t);
        char        *getlogin(void);
        int          getlogin_r(char *, size_t);
        int          getopt(int, char * const [], const char *);
        pid_t        getpgid(pid_t);
        pid_t        getpgrp(void);
        pid_t        getpid(void);
        pid_t        getppid(void);
        pid_t        getsid(pid_t);
        uid_t        getuid(void);
        int          isatty(int);
        int          lchown(const char *, uid_t, gid_t);
        int          link(const char *, const char *);
        int          linkat(int, const char *, int, const char *, int);
XSI     int          lockf(int, int, off_t);
        off_t        lseek(int, off_t, int);
XSI     int          nice(int);
        long         pathconf(const char *, int);
        int          pause(void);
        int          pipe(int [2]);
        int          pipe2(int [2], int);
        int          posix_close(int, int);
        ssize_t      pread(int, void *, size_t, off_t);
        ssize_t      pwrite(int, const void *, size_t, off_t);
```

```
      int          fputws(const wchar_t *restrict, FILE *restrict);
      int          fwide(FILE *, int);
      int          fwprintf(FILE *restrict, const wchar_t *restrict, ...);
      int          fwscanf(FILE *restrict, const wchar_t *restrict, ...);
      wint_t       getwc(FILE *);
      wint_t       getwchar(void);
      size_t       mbrlen(const char *restrict, size_t, mbstate_t *restrict);
      size_t       mbrtowc(wchar_t *restrict, const char *restrict, size_t,
                       mbstate_t *restrict);
      int          mbsinit(const mbstate_t *);
CX    size_t       mbsnrtowcs(wchar_t *restrict, const char **restrict,
                       size_t, size_t, mbstate_t *restrict);
      size_t       mbsrtowcs(wchar_t *restrict, const char **restrict, size_t,
                       mbstate_t *restrict);
CX    FILE        *open_wmemstream(wchar_t **, size_t *);
      wint_t       putwc(wchar_t, FILE *);
      wint_t       putwchar(wchar_t);
      int          swprintf(wchar_t *restrict, size_t,
                       const wchar_t *restrict, ...);
      int          swscanf(const wchar_t *restrict,
                       const wchar_t *restrict, ...);
      wint_t       ungetwc(wint_t, FILE *);
      int          vfwprintf(FILE *restrict, const wchar_t *restrict, va_list);
      int          vfwscanf(FILE *restrict, const wchar_t *restrict, va_list);
      int          vswprintf(wchar_t *restrict, size_t,
                       const wchar_t *restrict, va_list);
      int          vswscanf(const wchar_t *restrict, const wchar_t *restrict,
                       va_list);
      int          vwprintf(const wchar_t *restrict, va_list);
      int          vwscanf(const wchar_t *restrict, va_list);
CX    wchar_t     *wcpcpy(wchar_t *restrict, const wchar_t *restrict);
      wchar_t     *wcpncpy(wchar_t *restrict, const wchar_t *restrict, size_t);
      size_t       wcrtomb(char *restrict, wchar_t, mbstate_t *restrict);
CX    int          wcscasecmp(const wchar_t *, const wchar_t *);
      int          wcscasecmp_l(const wchar_t *, const wchar_t *, locale_t);
      wchar_t     *wcscat(wchar_t *restrict, const wchar_t *restrict);
      wchar_t     *wcschr(const wchar_t *, wchar_t);
      int          wcscmp(const wchar_t *, const wchar_t *);
      int          wcscoll(const wchar_t *, const wchar_t *);
CX    int          wcscoll_l(const wchar_t *, const wchar_t *, locale_t);
      wchar_t     *wcscpy(wchar_t *restrict, const wchar_t *restrict);
      size_t       wcscspn(const wchar_t *, const wchar_t *);
CX    wchar_t     *wcsdup(const wchar_t *);
      size_t       wcsftime(wchar_t *restrict, size_t,
                       const wchar_t *restrict, const struct tm *restrict);
CX    size_t       wcslcat(wchar_t *restrict, const wchar_t *restrict,           +
                       size_t);                                                  +
      size_t       wcslcpy(wchar_t *restrict, const wchar_t *restrict,           +
                       size_t);                                                  +
      size_t       wcslen(const wchar_t *);
CX    int          wcsncasecmp(const wchar_t *, const wchar_t *, size_t);
      int          wcsncasecmp_l(const wchar_t *, const wchar_t *, size_t,
```

	Header	Prefix	Suffix	Complete Name
	<aio.h>	aio_, lio_, AIO_, LIO_		
	<arpa/inet.h>	inet_		
	<ctype.h>	to[a-z], is[a-z]		
	<dlfcn.h>	RTLD_, dli_		
	<dirent.h>	d_, DT_		
	<fcntl.h>	l_		
XSI	**<fmtmsg.h>**	MM_		
	<fnmatch.h>	FNM_		
XSI	**<ftw.h>**	FTW		
	<glob.h>	gl_, GLOB_		
	<grp.h>	gr_		
	<limits.h>		_MAX, _MIN	
XSI	**<math.h>**	M_		
MSG	**<mqueue.h>**	mq_, MQ_		
XSI	**<ndbm.h>**	dbm_, DBM_		
	<netdb.h>	ai_, h_, n_, p_, s_		
	<net/if.h>	if_, IF_		
	<netinet/in.h>	in_, ip_, s_, sin_, INADDR_, IPPROTO_		
IP6		in6_, in6addr_, s6_, sin6_, IPV6_		
	<netinet/tcp.h>	TCP_		
	<nl_types.h>	NL_		
	<poll.h>	pd_, ph_, ps_, POLL		
	<pthread.h>	pthread_, PTHREAD_		
	<pwd.h>	pw_		
	<regex.h>	re_, rm_, REG_		
	<sched.h>	sched_, SCHED_		
	<semaphore.h>	sem_, SEM_		
CX	**<signal.h>**	sa_, si_, sigev_, sival_, uc_, BUS_, CLD_,		
		FPE_, ILL_, SA_, SEGV_, SI_, SIGEV_,		
XSI		ss_, sv_, SS_, TRAP_		
	<stdlib.h>	str[a-z]		
	<string.h>	str[a-z], mem[a-z], wcs[a-z]		
XSI	**<sys/ipc.h>**	ipc_, IPC_		key, pad, seq
	<sys/mman.h>	shm_, MAP_, MCL_, MS_, PROT_		
XSI	**<sys/msg.h>**	msg, MSG_[A-Z]		msg
XSI	**<sys/resource.h>**	rlim_, ru_, PRIO_, RLIMIT_, RUSAGE_		
	<sys/select.h>	fd_, fds_, FD_		

_Exit( )
_Fork( )
_exit( )
abort( )
accept( )
accept4( )
access( )
aio_error( )
aio_return( )
aio_suspend( )
alarm( )
be16toh( )
be32toh( )
be64toh( )
bind( )
cfgetispeed( )
cfgetospeed( )
cfsetispeed( )
cfsetospeed( )
chdir( )
chmod( )
chown( )
clock_gettime( )
close( )
connect( )
creat( )
dup( )
dup2( )
dup3( )
execl( )
execle( )
execv( )
execve( )
faccessat( )
fchdir( )
fchmod( )
fchmodat( )
fchown( )
fchownat( )
fcntl( )
fdatasync( )
fexecve( )
ffs( )
fstat( )
fstatat( )
fsync( )
ftruncate( )
futimens( )
getegid( )
geteuid( )
getgid( )
getgroups( )
getpeername( )
getpgrp( )
getpid( )
getppid( )
getsockname( )
getsockopt( )
getuid( )
htobe16( )
htobe32( )
htobe64( )
htole16( )
htole32( )
htole64( )
htonl( )
htons( )
kill( )
le16toh( )
le32toh( )
le64toh( )
link( )
linkat( )
listen( )
longjmp( )
lseek( )
lstat( )
memccpy( )
memchr( )
memcmp( )
memcpy( )
memmove( )
memset( )
mkdir( )
mkdirat( )
mkfifo( )
mkfifoat( )
mknod( )
mknodat( )
ntohl( )
ntohs( )
open( )
openat( )
pause( )
pipe( )
pipe2( )
poll( )
ppoll( )
pread( )
pselect( )
pthread_kill( )
pthread_self( )
pthread_setcancelstate( )
pthread_sigmask( )
pwrite( )
raise( )
read( )
readlink( )
readlinkat( )
recv( )
recvfrom( )
recvmsg( )
rename( )
renameat( )
rmdir( )
select( )
sem_post( )
send( )
sendmsg( )
sendto( )
setegid( )
seteuid( )
setgid( )
setpgid( )
setregid( )
setreuid( )
setsid( )
setsockopt( )
setuid( )
shutdown( )
sig2str( )
sigaction( )
sigaddset( )
sigdelset( )
sigemptyset( )
sigfillset( )
sigismember( )
siglongjmp( )
signal( )
sigpending( )
sigprocmask( )
sigqueue( )
sigsuspend( )
sleep( )
sockatmark( )
socket( )
socketpair( )
stat( )
stpcpy( )
stpncpy( )
strcat( )
strchr( )
strcmp( )
strcpy( )
strcspn( )
strlcat( )
strlcpy( )
strlen( )
strncat( )
strncmp( )
strncpy( )
strnlen( )
strpbrk( )
strrchr( )
strspn( )
strstr( )
strtok_r( )
symlink( )
symlinkat( )
tcdrain( )
tcflow( )
tcflush( )
tcgetattr( )
tcgetpgrp( )
tcgetwinsize( )
tcsendbreak( )
tcsetattr( )
tcsetpgrp( )
tcsetwinsize( )
time( )
timer_getoverrun( )
timer_gettime( )
timer_settime( )
times( )
umask( )
uname( )
unlink( )
unlinkat( )
utimensat( )
utimes( )
va_arg( )
va_copy( )
va_end( )
va_start( )
wait( )
waitpid( )
wcpcpy( )
wcpncpy( )
wcscat( )
wcschr( )
wcscmp( )
wcscpy( )
wcscspn( )
wcslcat( )
wcslcpy( )
wcslen( )
wcsncat( )
wcsncmp( )
wcsncpy( )
wcsnlen( )
wcspbrk( )
wcsrchr( )

If a thread is detached, its thread ID is invalid for use as an argument in a call to *pthread_detach*( ) or *pthread_join*( ).

2.9.3 Thread Mutexes

A thread that has blocked shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources is determined by the scheduling policy.

A thread shall become the owner of a mutex, *m*, when one of the following occurs:

- It calls *pthread_mutex_clocklock*( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), or *pthread_mutex_trylock*( ) with *m* as the *mutex* argument and the call returns zero or [EOWNERDEAD].
- It calls *pthread_mutex_setprioceiling*( ) with *m* as the *mutex* argument and the call returns [EOWNERDEAD].
- It calls *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) with *m* as the *mutex* argument and the call returns zero or certain error numbers (see *pthread_cond_timedwait*( )).

The thread shall remain the owner of *m* until one of the following occurs:

- It executes *pthread_mutex_unlock*( ) with *m* as the *mutex* argument
- It blocks in a call to *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) with *m* as the *mutex* argument.

The implementation shall behave as if at all times there is at most one owner of any mutex.

A thread that becomes the owner of a mutex is said to have ``acquired'' the mutex and the mutex is said to have become ``locked''; when a thread gives up ownership of a mutex it is said to have ``released'' the mutex and the mutex is said to have become ``unlocked''.

A problem can occur if a process terminates while one of its threads holds a mutex lock. Depending on the mutex type, it might be possible for another thread to unlock the mutex and recover the state of the mutex. However, it is difficult to perform this recovery reliably.

Robust mutexes provide a means to enable the implementation to notify other threads in the event of a process terminating while one of its threads holds a mutex lock. The next thread that acquires the mutex is notified about the termination by the return value [EOWNERDEAD] from the locking function. The notified thread can then attempt to recover the state protected by the mutex, and if successful mark the state protected by the mutex as consistent by a call to *pthread_mutex_consistent*( ). If the notified thread is unable to recover the state, it can declare the state as not recoverable by a call to *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ).

Whether or not the state protected by a mutex can be recovered is dependent solely on the application using robust mutexes. The robust mutex support provided in the implementation provides notification only that a mutex owner has terminated while holding a lock, or that the state of the mutex is not recoverable.

2.9.5.2 *Cancellation Points*

Cancellation points shall occur when a thread is executing the following functions:

accept( )
accept4( )
aio_suspend( )
clock_nanosleep( )
close( )
connect( )
creat( )
fcntl( )†
fdatasync( )
fsync( )
lockf( )††
mq_receive( )
mq_send( )
mq_timedreceive( )
mq_timedsend( )
msgrcv( )
msgsnd( )
msync( )
nanosleep( )
open( )
openat( )
pause( )
poll( )
ppoll( )
pread( )
pselect( )
pthread_cond_clockwait( )
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_join( )
pthread_testcancel( )
pwrite( )
read( )
readv( )
recv( )
recvfrom( )
recvmsg( )
select( )
send( )
sendmsg( )
sendto( )
sigsuspend( )
sigtimedwait( )
sigwait( )
sigwaitinfo( )
sleep( )
tcdrain( )
wait( )
waitid( )
waitpid( )
write( )
writev( )

A cancellation point may also occur when a thread is executing the following functions:

access( )
asctime_r( )
catclose( )
catopen( )
chmod( )
chown( )
closedir( )
closelog( )
ctermid( )
ctime_r( )
dlclose( )
dlopen( )
dprintf( )
endhostent( )
endnetent( )
endprotoent( )
endservent( )
faccessat( )
fchmod( )
fchmodat( )
fchown( )
fchownat( )
fclose( )
fcntl( )†††
fflush( )
fgetc( )
fgetpos( )
fgets( )
fgetwc( )
fgetws( )
fmtmsg( )
fopen( )
fpathconf( )
fprintf( )
fputc( )
fputs( )
fputwc( )
fputws( )
fread( )
freopen( )
fscanf( )
fseek( )
fseeko( )
fsetpos( )
fstat( )
fstatat( )
ftell( )
ftello( )
futimens( )
fwprintf( )
fwrite( )
fwscanf( )
getaddrinfo( )
getc( )
getc_unlocked( )
getchar( )
getchar_unlocked( )
getcwd( )
getdelim( )
getgrgid_r( )
getgrnam_r( )
gethostid( )
gethostname( )

† When the *cmd* argument is F_SETLKW.

†† When the *function* argument is F_LOCK.

††† For any value of the *cmd* argument.

getline()
getlogin_r()
getnameinfo()
getpwnam_r()
getpwuid_r()
getwc()
getwchar()
glob()
iconv_close()
iconv_open()
link()
linkat()
lio_listio()
localtime_r()
lockf()
lseek()
lstat()
mkdir()
mkdirat()
mkdtemp()
mkfifo()
mkfifoat()
mknod()
mknodat()
mkstemp()
mktime()
opendir()
openlog()
pathconf()
perror()
popen()
posix_fadvise()
posix_fallocate()
posix_getdents()
posix_madvise()
posix_openpt()
posix_spawn()
posix_spawnp()
posix_typed_mem_open()
printf()
psiginfo()
psignal()
pthread_rwlock_clockrdlock()
pthread_rwlock_clockwrlock()
pthread_rwlock_rdlock()
pthread_rwlock_timedrdlock()
pthread_rwlock_timedwrlock()
pthread_rwlock_wrlock()
ptsname()
ptsname_r()
putc()
putc_unlocked()
putchar()
putchar_unlocked()
puts()
putwc()
putwchar()
readdir_r()
readlink()
readlinkat()
remove()
rename()
renameat()
rewind()
rewinddir()
scandir()
scanf()
seekdir()
sem_clockwait()
sem_timedwait()
sem_wait()
semop()
sethostent()
setnetent()
setprotoent()
setservent()
stat()
strerror_l()
strerror_r()
strftime()
strftime_l()
symlink()
symlinkat()
sync()
syslog()
tmpfile()
tmpnam()
ttyname_r()
tzset()
ungetc()
ungetwc()
unlink()
unlinkat()
utimensat()
utimes()
vdprintf()
vfprintf()
vfwprintf()
vprintf()
vwprintf()
wcsftime()
wordexp()
wprintf()
wscanf()

In addition, a cancellation point may occur when a thread is executing any function that this standard does not require to be thread-safe but the implementation documents as being thread-safe. If a thread is cancelled while executing a non-thread-safe function, the behavior is undefined.

An implementation shall not introduce cancellation points into any other functions specified in this volume of POSIX.1-202x.

The side-effects of acting upon a cancellation request while suspended during a call of a function are the same as the side-effects that may be seen in a single-threaded program when a call to a function is interrupted by a signal and the given function returns [EINTR]. Any such side-effects occur before any cancellation cleanup handlers are called. For functions that are explicitly required not to return when interrupted (for example, *pclose*()), if a thread is canceled while executing the function, the behavior is undefined.

Whenever a thread has cancelability enabled and a cancellation request has been made with that thread as the target, and the thread then calls any function that is a cancellation point (such as *pthread_testcancel*() or *read*()), the cancellation request shall be acted upon before the function returns. If a thread has cancelability enabled and a cancellation request is made with the thread

2.11.1 Defined Types

All of the data types used by various functions are defined by the implementation. The following table describes some of these types. Other types referenced in the description of a function, not mentioned here, can be found in the appropriate header for that function.

Defined Type	Description
cc_t	Type used for terminal special characters.
clock_t	Integer or real-floating type used for processor times, as defined in the ISO C standard.
clockid_t	Used for clock ID type in some timer functions.
dev_t	Integer type used for device numbers.
DIR	Type representing a directory stream.
div_t	Structure type returned by the *div*( ) function.
FILE	Structure containing information about a file.
glob_t	Structure type used in pathname pattern matching.
fpos_t	Type containing all information needed to specify uniquely every position within a file.
gid_t	Integer type used for group IDs.
iconv_t	Type used for conversion descriptors.
id_t	Integer type used as a general identifier; can be used to contain at least the largest of a **pid_t**, **uid_t**, or **gid_t**.
ino_t	Unsigned integer type used for file serial numbers.
key_t	Arithmetic type used for XSI interprocess communication.
ldiv_t	Structure type returned by the *ldiv*( ) function.
mode_t	Integer type used for file attributes.
mqd_t	Used for message queue descriptors.
nfds_t	Integer type used for the number of file descriptors.
nlink_t	Integer type used for link counts.
off_t	Signed integer type used for file sizes.
pid_t	Signed integer type used for process and process group IDs.
pthread_attr_t	Used to identify a thread attribute object.
pthread_cond_t	Used for condition variables.
pthread_condattr_t	Used to identify a condition attribute object.
pthread_key_t	Used for thread-specific data keys.
pthread_mutex_t	Used for mutexes.
pthread_mutexattr_t	Used to identify a mutex attribute object.
pthread_once_t	Used for dynamic package initialization.
pthread_rwlock_t	Used for read-write locks.
pthread_rwlockattr_t	Used for read-write lock attributes.
pthread_t	Used to identify a thread.
ptrdiff_t	Signed integer type of the result of subtracting two pointers.
reclen_t	Unsigned integer type used for directory entry lengths.
regex_t	Structure type used in regular expression matching.
regmatch_t	Structure type used in regular expression matching.
rlim_t	Unsigned integer type used for limit values, to which objects of type **int** and **off_t** can be cast without loss of value.
sem_t	Type used in performing semaphore operations.
sig_atomic_t	Possibly volatile-qualified integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.
sigset_t	Integer or structure type of an object used to represent sets

NAME

bind — bind a name to a socket

SYNOPSIS

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

int bind(int socket, const struct sockaddr *address,
    socklen_t address_len);
```

DESCRIPTION

The *bind*( ) function shall assign a local socket address *address* to a socket identified by descriptor *socket* that has no local socket address assigned. Sockets created with the *socket*( ) function are initially unnamed; they are identified only by their address family.

The *bind*( ) function takes the following arguments:

socket Specifies the file descriptor of the socket to be bound.

address Points to a **sockaddr** structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.

address_len Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

The socket specified by *socket* may require the process to have appropriate privileges to use the *bind*( ) function.

If the address family of the socket is AF_UNIX and the pathname in *address* names a symbolic link, *bind*( ) shall fail and set *errno* to [EADDRINUSE].

If the socket address cannot be assigned immediately and O_NONBLOCK is set for the file descriptor for the socket, *bind*( ) shall fail and set *errno* to [EINPROGRESS], but the assignment request shall not be aborted, and the assignment shall be completed asynchronously. Subsequent calls to *bind*( ) for the same socket, before the assignment is completed, shall fail and set *errno* to [EALREADY].

When the assignment has been performed asynchronously, *pselect*( ), *select*( ), *poll*( ), and *ppoll*( ) shall indicate that the file descriptor for the socket is ready for reading and writing. |

RETURN VALUE

Upon successful completion, *bind*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *bind*( ) function shall fail if:

[EADDRINUSE] The specified address is already in use.

[EADDRNOTAVAIL]
The specified address is not available from the local machine.

[EAFNOSUPPORT]
The specified address is not a valid address for the address family of the specified socket.

[EALREADY] An assignment request is already in progress for the specified socket.

[EBADF] The *socket* argument is not a valid file descriptor.

NAME

connect — connect a socket

SYNOPSIS

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

int connect(int socket, const struct sockaddr *address,
    socklen_t address_len);
```

DESCRIPTION

The *connect*( ) function shall attempt to make a connection on a connection-mode socket or to set or reset the peer address of a connectionless-mode socket. The function takes the following arguments:

socket	Specifies the file descriptor associated with the socket.
address	Points to a **sockaddr** structure containing the peer address. The length and format of the address depend on the address family of the socket.
address_len	Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

If the socket has not already been bound to a local address, *connect*( ) shall bind it to an address which, unless the socket's address family is AF_UNIX, is an unused local address.

If the initiating socket is not connection-mode, then *connect*( ) shall set the socket's peer address, and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all datagrams are sent on subsequent *send*( ) functions, and limits the remote sender for subsequent *recv*( ) functions. If the *sa_family* member of *address* is AF_UNSPEC, the socket's peer address shall be reset. Note that despite no connection being made, the term ``connected'' is used to describe a connectionless-mode socket for which a peer address has been set.

If the initiating socket is connection-mode, then *connect*( ) shall attempt to establish a connection to the address specified by the *address* argument. If the connection cannot be established immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect*( ) shall block for up to an unspecified timeout interval until the connection is established. If the timeout interval expires before the connection is established, *connect*( ) shall fail and the connection attempt shall be aborted. If *connect*( ) is interrupted by a signal that is caught while blocked waiting to establish a connection, *connect*( ) shall fail and set *errno* to [EINTR], but the connection request shall not be aborted, and the connection shall be established asynchronously.

If the connection cannot be established immediately and O_NONBLOCK is set for the file descriptor for the socket, *connect*( ) shall fail and set *errno* to [EINPROGRESS], but the connection request shall not be aborted, and the connection shall be established asynchronously. Subsequent calls to *connect*( ) for the same socket, before the connection is established, shall fail and set *errno* to [EALREADY].

When the connection has been established asynchronously, *pselect*( ), *select*( ), *poll*( ), and *ppoll*( ) shall indicate that the file descriptor for the socket is ready for writing.

The socket in use may require the process to have appropriate privileges to use the *connect*( ) function.

RETURN VALUE

Upon successful completion, *connect*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

NAME

dladdr — get information relating to an address

SYNOPSIS

```
#include <dlfcn.h>

int dladdr(const void *restrict addr, Dl_info_t *restrict dlip);
```

DESCRIPTION

The *dladdr*( ) function shall determine whether the address specified by *addr* is located within the address range occupied by a mapped object. The mapped objects examined shall include any executable object files that have previously been loaded by a call to *dlopen*( ) and for which *dlclose*( ) has not subsequently been called, and any shared library files that were loaded as dependencies of the executable file from which the current process image was loaded; they may also include any executable object files that have previously been loaded by a call to *dlopen*( ) and for which *dlclose*( ) has subsequently been called, the executable file from which the current process image was loaded, and implementation-defined additional mapped objects (for example, all regular files mapped using *mmap*( ) might be included). If the specified address is within the mapped address range of one of these mapped objects and the object contains a symbol table, the symbol table shall be searched for a symbol (a function identifier or a data object identifier) that has the largest address less than or equal to the specified address.

If the address specified by *addr* is within the mapped address range of one of the examined mapped objects, the structure pointed to by *dlip* shall be populated as follows:

- The value of the *dli_fname* member shall be set to point to the pathname of the mapped object. (This might no longer resolve to the file that was mapped, for example if it was a link that has subsequently been removed or renamed.)

- The value of the *dli_fbase* member shall be set to the base of the address range occupied by the mapped object.

- The value of the *dli_sname* member shall be set to point to the name of the symbol that has the largest address less than or equal to the specified address, or to a null pointer if no such symbol was found.

- If *dli_sname* is set to a null pointer, the value of the *dli_saddr* member shall also be set to a null pointer. Otherwise, if *dli_sname* names a function identifier, *dli_saddr* shall be set to the address of the function converted from type pointer to function to type pointer to **void**; otherwise, *dli_saddr* shall be set to the address of the data object named by *dli_sname* converted from a pointer to the type of the data object to a pointer to **void**.

RETURN VALUE

Upon successful completion, a non-zero value shall be returned. If the specified address is not located within the address range occupied by an examined mapped object, or if an error occurs, zero shall be returned. More detailed diagnostic information shall be available through *dlerror*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The **Dl_info_t** members may point to addresses within the mapped object. These pointers can become invalid if the object is unmapped (for example, loaded executable objects may be unloaded by *dlclose*( )).

If *dli_sname* names a function identifier, the value of *dli_saddr* can be converted back to type pointer to function using a cast in the manner shown in the *dlsym*( ) EXAMPLES section. Note that this conversion is not defined by the ISO C standard. This standard requires this conversion to work correctly on conforming implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dlclose( ), *dlerror*( ), *dlopen*( ), *dlsym*( )

XBD **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 8.

```
        assert(xsubi[1] == 10728);
        assert(xsubi[2] == 27921);
        assert(nrand48(xsubi) == 754104482);
        assert(xsubi[0] == 6828);
        assert(xsubi[1] == 28997);
        assert(xsubi[2] == 23013);
        assert(nrand48(xsubi) == 609453945);
        assert(xsubi[0] == 58183);
        assert(xsubi[1] == 3826);
        assert(xsubi[2] == 18599);
        assert(nrand48(xsubi) == 1878644360);
        assert(xsubi[0] == 36678);
        assert(xsubi[1] == 44304);
        assert(xsubi[2] == 57331);
        assert(nrand48(xsubi) == 2114923686);
        assert(xsubi[0] == 58585);
        assert(xsubi[1] == 22861);
        assert(xsubi[2] == 64542);
      }
    }
```

APPLICATION USAGE

These functions should be avoided whenever non-trivial requirements (including safety) have to be fulfilled, unless seeded using *getentropy*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getentropy( ), *initstate*( ), *rand*( )

XBD **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that the *drand48*( ), *lrand48*( ), and *mrand48*( ) functions need not be reentrant is added to the DESCRIPTION.

Issue 6

The normative text is updated to avoid use of the term ``must'' for application requirements.

Issue 7

Austin Group Interpretation 1003.1-2001 #156 is applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0083 [743] is applied.

Issue 8

Austin Group Defect 1107 is applied, clarifying how the return value is calculated from X_i for each function.

Austin Group Defect 1134 is applied, adding *getentropy*( ).

possible for the system to conform to the intent of this volume of POSIX.1-202x.

The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it occurs or not is not in any practical sense under the control of the application because the condition is usually a consequence of the user's use of the system, not of the application's code. Thus, no application can or should rely upon its occurrence under any circumstances, nor should the exact semantics of what concept of ``user'' is used be of concern to the application developer. Validation writers should be cognizant of this limitation.

There are two reasons why POSIX programmers call *fork*( ). One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to *fork*( ) is soon followed by a call to one of the *exec* functions.

The general problem with making *fork*( ) work in a multi-threaded world is what to do with all of the threads. There are two alternatives. One is to copy all of the threads into the new process. This causes the programmer or implementation to deal with threads that are suspended on system calls or that might be about to execute system calls that should not be executed in the new process. The other alternative is to copy only the thread that calls *fork*( ). This creates the difficulty that the state of process-local resources is usually held in process memory. If a thread that is not calling *fork*( ) holds a resource, that resource is never released in the child process because the thread whose job it is to release the resource does not exist in the child process.

When a programmer is writing a multi-threaded program, the first described use of *fork*( ), creating new threads in the same program, is provided by the *pthread_create*( ) function. The *fork*( ) function is thus used only to run new programs, and the effects of calling functions that require certain resources between the call to *fork*( ) and the call to an *exec* function are undefined.

The addition of the *forkall*( ) function to the standard was considered and rejected. The *forkall*( ) function lets all the threads in the parent be duplicated in the child. This essentially duplicates the state of the parent in the child. This allows threads in the child to continue processing and allows locks and the state to be preserved without explicit *pthread_atfork*( ) code. The calling process has to ensure that the threads processing state that is shared between the parent and child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall*( ). For example, if a thread is reading a file descriptor in the parent when *forkall*( ) is called, then two threads (one in the parent and one in the child) are reading the file descriptor after the *forkall*( ). If this is not desired behavior, the parent process has to synchronize with such threads before calling *forkall*( ).

When *forkall*( ) is called, threads, other than the calling thread, that are in functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function behaves correctly in the parent and child. In particular, *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), and *pthread_cond_wait*( ) need to return in order to ensure that the condition has not changed. These functions can be awakened by a spurious condition wakeup rather than returning [EINTR].

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *exec*, *fcntl*( ), *pthread_atfork*( ), *semop*( ), *signal*( ), *times*( )

XBD Section 4.13 (on page 91), **<sys/types.h>**, **<unistd.h>**

NAME

free — free allocated memory

SYNOPSIS

```
#include <stdlib.h>

void free(void *ptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-202x defers to the ISO C standard.

The *free*( ) function shall cause the space pointed to by *ptr* to be deallocated; that is, made available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by a function in POSIX.1-202x that allocates memory as if by *malloc*( ), or if the space has been deallocated by a call to *free*( ), *realloc*( ), CX or *reallocarray*( ), the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

CX The *free*( ) function shall not modify *errno* if *ptr* is a null pointer or a pointer previously returned as if by *malloc*( ) and not yet deallocated.

RETURN VALUE

The *free*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

Because the *free*( ) function does not modify *errno* for valid pointers, it is safe to use it in cleanup code without corrupting earlier errors, such as in this example code:

```
// buf was obtained by malloc(buflen)
ret = write(fd, buf, buflen);
if (ret < 0) {
    free(buf);
    return ret;
}
```

However, earlier versions of this standard did not require this, and the same example had to be written as:

```
// buf was obtained by malloc(buflen)
ret = write(fd, buf, buflen);
if (ret < 0) {
    int save = errno;
    free(buf);
    errno = save;
    return ret;
}
```

NAME

getentropy — fill a buffer with random bytes

SYNOPSIS

```
#include <unistd.h>

int getentropy(void *buffer, size_t length);
```

DESCRIPTION

The *getentropy*( ) function shall write *length* bytes of data starting at the location pointed to by *buffer*. The output shall be unpredictable high quality random data, generated by a cryptographically secure pseudo-random number generator. The maximum permitted value for the *length* argument is given by the {GETENTROPY_MAX} symbolic constant defined in **<limits.h>**.

A successful call to *getentropy*( ) shall always provide the requested number of bytes of entropy.

RETURN VALUE

Upon successful completion, *getentropy*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getentropy*( ) function shall fail if:

[EINVAL] The value of *length* is greater than {GETENTROPY_MAX}.

The *getentropy*( ) function may fail if:

[ENOSYS] The system does not provide the necessary source of entropy.

EXAMPLES

None.

APPLICATION USAGE

The intended use of this function is to create a seed for other pseudo-random number generators.

RATIONALE

The *getentropy*( ) function is not a cancellation point. (See Section 2.9.5.2 (on page 504).)

FUTURE DIRECTIONS

None.

SEE ALSO

drand48( ), *initstate*( ), *rand*( )

XBD **<limits.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 8.

NAME

getlocalename_l — get a locale name from a locale object

SYNOPSIS

CX
```
#include <locale.h>

const char *getlocalename_l(int category, locale_t locobj);
```

DESCRIPTION

The *getlocalename_l*( ) function shall return the locale name for the given locale category of the locale object *locobj*, or of the global locale if *locobj* is the special locale object LC_GLOBAL_LOCALE.

The *category* argument specifies the locale category to be queried. If the value is LC_ALL or is not a supported locale category value (see *setlocale*( )), *getlocalename_l*( ) shall fail.

The behavior is undefined if the *locobj* argument is neither the special locale object LC_GLOBAL_LOCALE nor a valid locale object handle.

RETURN VALUE

Upon successful completion, *getlocalename_l*( ) shall return a pointer to a string containing the locale name; otherwise, a null pointer shall be returned.

If *locobj* is LC_GLOBAL_LOCALE, the returned string pointer might be invalidated or the string content might be overwritten by a subsequent call in the same thread to *getlocalename_l*( ) with LC_GLOBAL_LOCALE; the returned string pointer might also be invalidated if the calling thread is terminated. Otherwise, the returned string pointer and content shall remain valid until the locale object *locobj* is used in a call to *freelocale*( ) or as the *base* argument in a successful call to *newlocale*( ).

ERRORS

No errors are defined.

EXAMPLES

Determining the locale name for a category of the current locale

The following example shows how to obtain the locale name for the LC_NUMERIC category of the current thread-local locale, or of the global locale if no thread-local locale is in use.

```
#include <locale.h>
...
const char *name;
locale_t loc = uselocale(NULL);
name = getlocalename_l(LC_NUMERIC, loc);
```

APPLICATION USAGE

None.

RATIONALE

Historical versions of *getlocalename_l*( ) did not handle the special locale object LC_GLOBAL_LOCALE, requiring that applications used *setlocale*(*category*, NULL) to query the global locale if *uselocale*(NULL) returned LC_GLOBAL_LOCALE. However, since *setlocale*( ) is not required to be thread-safe (even when the only concurrent calls are ones that query the locale), this method was problematic for multi-threaded processes. This standard requires that *getlocalename_l*(*category*, LC_GLOBAL_LOCALE) queries the global locale in a thread-safe manner, for example by returning a pointer to a thread-local internal buffer instead of a process-wide internal buffer.

FUTURE DIRECTIONS

None.

SEE ALSO

freelocale( ), *newlocale( )*, *setlocale( )*, *uselocale( )*

XBD Chapter 7 (on page 113), **<locale.h>**

CHANGE HISTORY

First released in Issue 8.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

After initialization, a state array can be restarted at a different point in one of two ways:

1. The *initstate*( ) function can be used, with the desired seed, state array, and size of the array.

2. The *setstate*( ) function, with the desired state, can be used, followed by *srandom*( ) with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

Although some implementations of *random*( ) have written messages to standard error, such implementations do not conform to POSIX.1-202x.

Issue 5 restored the historical behavior of this function.

Threaded applications should use *erand48*( ), *nrand48*( ), or *jrand48*( ) instead of *random*( ) when an independent random number sequence in multiple threads is required.

These functions should be avoided whenever non-trivial requirements (including safety) have to be fulfilled, unless seeded using *getentropy*( ). |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

drand48( ), *getentropy*( ), *rand*( ) +

XBD **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the phrase ``values smaller than 8'' is replaced with ``values greater than or equal to 8, or less than 32'', ``*size*<8'' is replaced with ``8≤*size* <32'', and a new first paragraph is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE indicating that these changes restore the historical behavior of the function.

Issue 6

In the DESCRIPTION, duplicate text ``For values greater than or equal to 8 . . .'' is removed.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/30 is applied, removing *rand_r*( ) from the list of suggested functions in the APPLICATION USAGE section.

Issue 7

The type of the first argument to *setstate*( ) is changed from **const char *** to **char ***.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0179 [743] is applied. +

NAME

memmem — find a byte subsequence in a byte sequence

SYNOPSIS

CX

```
#include <string.h>

void *memmem(const void *haystack, size_t haystacklen,
             const void *needle, size_t needlelen);
```

DESCRIPTION

The *memmem*( ) function shall locate the first occurrence of byte sequence *needle* of length *needlelen* in byte sequence *haystack* of length *haystacklen*.

RETURN VALUE

Upon successful completion, *memmem*( ) shall return a pointer to the the first byte of the located byte sequence in *haystack*, or a null pointer if the byte sequence is not found.

If *needlelen* is zero, the function shall return *haystack*.

If *haystacklen* is less than *needlelen*, the function shall return a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

This function is similar to *strstr*( ), except that NUL bytes may be included in either *needle* or *haystack*.

FUTURE DIRECTIONS

None.

SEE ALSO

memchr( ), *strstr*( )

XBD **<string.h>**

CHANGE HISTORY

First released in Issue 8.

NAME

poll, ppoll — input/output multiplexing

SYNOPSIS

```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
int ppoll(struct pollfd fds[], nfds_t nfds,
    const struct timespec *restrict timeout,
    const sigset_t *restrict sigmask);
```

DESCRIPTION

The *ppoll*( ) function provides applications with a mechanism for multiplexing input/output over a set of file descriptors. For each member of the array pointed to by *fds*, *ppoll*( ) shall examine the given file descriptor for the event(s) specified in *events*. The number of **pollfd** structures in the *fds* array is specified by *nfds*. The *ppoll*( ) function shall identify those file descriptors on which an application can read or write data, or on which certain events have occurred.

The *poll*( ) function shall be equivalent to the *ppoll*( ) function, except as follows:

- For the *poll*( ) function, the timeout period is given in milliseconds in an argument of type **int**, whereas for the *ppoll*( ) function the timeout period is given in seconds and nanoseconds via an argument of type pointer to **struct timespec**. A *timeout* of −1 for *poll*( ) shall be equivalent to passing a null pointer for the *timeout* for *ppoll*( ).
- The *poll*( ) function has no *sigmask* argument; it shall behave as *ppoll*( ) does when *sigmask* is a null pointer.

The *fds* argument specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one member for each open file descriptor of interest. The array's members are **pollfd** structures within which *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by OR'ing a combination of the following event flags:

POLLIN Data other than high-priority data may be read without blocking.

POLLRDNORM Normal data may be read without blocking.

POLLRDBAND Priority data may be read without blocking.

POLLPRI High-priority data may be read without blocking.

POLLOUT Normal data may be written without blocking.

POLLWRNORM Equivalent to POLLOUT.

POLLWRBAND Priority data may be written.

POLLERR An error has occurred on the device or stream. This flag is only valid in the *revents* bitmask; it shall be ignored in the *events* member.

POLLHUP A device has been disconnected, or a pipe or FIFO has been closed by the last process that had it open for writing. Once set, the hangup state of a FIFO shall persist until some process opens the FIFO for writing or until all read-only file descriptors for the FIFO are closed. This event and POLLOUT are mutually-exclusive; a stream can never be writable if a hangup has occurred. However, this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are not mutually-exclusive. This flag is only valid in the *revents* bitmask; it shall be ignored in the *events* member.

POLLNVAL The specified *fd* value is invalid. This flag is only valid in the *revents* member; it shall ignored in the *events* member.

The significance and semantics of normal, priority, and high-priority data are file and device-specific.

If the value of *fd* is less than 0, *events* shall be ignored, and *revents* shall be set to 0 in that entry on return from *poll*( ) or *ppoll*( ).

In each **pollfd** structure, *poll*( ) or *ppoll*( ) shall clear the *revents* member, except that where the application requested a report on a condition by setting one of the bits of *events* listed above, *poll*( ) or *ppoll*( ) shall set the corresponding bit in *revents* if the requested condition is true. In addition, *poll*( ) or *ppoll*( ) shall set the POLLHUP, POLLERR, and POLLNVAL flag in *revents* if the condition is true, even if the application did not set the corresponding bit in *events*.

The *timeout* argument controls how long the *poll*( ) or *ppoll*( ) function shall wait before timing out. If the *timeout* argument is positive for *poll*( ) or not a null pointer for *ppoll*( ), it specifies a maximum interval to wait for the poll to complete. If the specified time interval expires without any of the defined events having occurred, the function shall return. If the *timeout* argument is −1 for *poll*( ) or a null pointer for *ppoll*( ), then the call shall block indefinitely until at least one descriptor meets the specified criteria or until the call is interrupted. To effect a poll, the application shall ensure that the *timeout* argument for *poll*( ) is 0, or for *ppoll*( ) is not a null pointer and points to a zero-valued **timespec** structure.

Implementations may place limitations on the maximum timeout interval supported. All implementations shall support a maximum timeout interval of at least 31 days for *ppoll*( ). If the *timeout* argument specifies a timeout interval greater than the implementation-defined maximum value, the maximum value shall be used as the actual timeout value. Implementations may also place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.

The *poll*( ) and *ppoll*( ) functions shall not be affected by the O_NONBLOCK flag.

The *poll*( ) and *ppoll*( ) functions shall support regular files, terminal and pseudo-terminal devices, FIFOs, pipes, and sockets. The behavior of *poll*( ) and *ppoll*( ) on elements of *fds* that refer to other types of file is unspecified.

Regular files shall always poll TRUE for reading and writing.

A file descriptor for a socket that is listening for connections shall indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously shall indicate that it is ready for writing, once a connection has been established.

Provided the application does not perform any action that results in unspecified or undefined behavior, the value of the *fd* and *events* members of each element of *fds* shall not be modified by *poll*( ) or *ppoll*( ).

If *sigmask* is not a null pointer, the *ppoll*( ) function shall replace the signal mask of the caller by the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the signal mask of the calling thread before returning. If a signal is unmasked as a result of the signal mask being altered by *ppoll*( ), and a signal-catching function is called for that signal during the execution of the *ppoll*( ) function, and SA_RESTART is clear for the interrupting signal, then

- If none of the defined events have occurred on any selected file descriptor, *ppoll*( ) shall immediately fail with the [EINTR] error after the signal-catching function returns.

- If one or more of the defined events have occurred, it is unspecified whether *ppoll*( ) behaves the same as if none of the events had occurred (failing with [EINTR] as above) or behaves the same as if it was not interrupted (returning the total number of **pollfd** structures that have selected events).

If a thread is canceled during a *ppoll*( ) call, it is unspecified whether the signal mask in effect when executing the registered cleanup functions is the original signal mask or the signal mask installed as part of the *ppoll*( ) call.

RETURN VALUE

Upon successful completion, a non-negative value shall be returned. A positive value shall indicate the total number of **pollfd** structures that have selected events (that is, those for which the *revents* member is non-zero). A value of 0 shall indicate that the call timed out and no file descriptors have been selected. Upon failure, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *poll*( ) and *ppoll*( ) functions shall fail if:

[EAGAIN] The allocation of internal data structures failed but a subsequent request may succeed.

[EINTR] A signal was caught during *poll*( ) or *ppoll*( ).

[EINVAL] The *nfds* argument is greater than {OPEN_MAX}.

The *ppoll*( ) function shall fail if:

[EINVAL] An invalid timeout interval was specified.

EXAMPLES

None.

APPLICATION USAGE

Other than the difference in the precision of the requested timeout, the following *ppoll*( ) call:

```
ready = ppoll(&fds, nfds, tmo_p, &sigmask);
```

is equivalent to atomically executing the following calls:

```
sigset_t origmask;
int timeout;

timeout = (tmo_p == NULL) ? -1 :
    (tmo_p->tv_sec * 1000 + tmo_p->tv_nsec / 1000000);
pthread_sigmask(SIG_SETMASK, &sigmask, &origmask);
ready = poll(&fds, nfds, timeout);
pthread_sigmask(SIG_SETMASK, &origmask, NULL);
```

RATIONALE

The POLLHUP event does not occur for FIFOs just because the FIFO is not open for writing. It only occurs when the FIFO is closed by the last writer and persists until some process opens the FIFO for writing or until all read-only file descriptors for the FIFO are closed.

Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers can install an additional cancellation handler which resets the signal mask to the expected value:

```
void cleanup(void *arg)
{
    sigset_t *ss = (sigset_t *) arg;
    pthread_sigmask(SIG_SETMASK, ss, NULL);
}
int call_ppoll(struct pollfd fds[], nfds_t nfds,
    const struct timespec *restrict timeout,
    const sigset_t *restrict sigmask)
{
    sigset_t oldmask;
    int result;
    pthread_sigmask(SIG_SETMASK, NULL, &oldmask);
    pthread_cleanup_push(cleanup, &oldmask);
    result = ppoll(fds, nfds, timeout, sigmask);
    pthread_cleanup_pop(0);
    return result;
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

pselect( ), *read*( ), *write*( )

XBD **<poll.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The description of POLLWRBAND is updated.

Issue 6

Text referring to sockets is added to the DESCRIPTION.

Functionality relating to the XSI STREAMS Option Group is marked.

The Open Group Corrigendum U055/3 is applied, updating the DESCRIPTION of POLLWRBAND.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/66 is applied, correcting the spacing in the EXAMPLES section.

Issue 7

Austin Group Interpretation 1003.1-2001 #209 is applied, clarifying the POLLHUP event.

The *poll*( ) function is moved from the XSI option to the Base.

Functionality relating to the XSI STREAMS option is marked obsolescent.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0249 [623] and XSH/TC2-2008/0250 [683] are applied.

Issue 8

Austin Group Defect 1263 is applied, adding *ppoll*( ).

Austin Group Defect 1330 is applied, removing obsolescent interfaces.

NAME

posix_getdents — read directory entries

SYNOPSIS

```
#include <dirent.h>

ssize_t posix_getdents(int fildes, void *buf, size_t nbyte, int flags);
```

DESCRIPTION

The *posix_getdents*( ) function shall attempt to read directory entries from the directory associated with the open file descriptor *fildes* and shall place information about the directory entries and the files they refer to in **posix_dent** structures in the buffer pointed to by *buf*, up to a maximum of *nbyte* bytes. The number of **posix_dent** structures populated in *buf* may be fewer than the number that will fit in *nbyte* bytes, but shall be at least one if *nbyte* is greater than the size of the **posix_dent** structure plus {NAME_MAX} and *fildes* is not currently at end-of-file.

The application shall ensure that *buf* is aligned suitably to point to a **posix_dent** structure. The alignment needed shall not be more restrictive than the alignment provided by *malloc*( ). Strictly conforming applications shall ensure that the value of *flags* is zero; other applications can set it to a value constructed by a bitwise-inclusive OR of implementation-defined bitwise-distinct flag values.

Each **posix_dent** structure returned in *buf* shall be located at an address that satisfies the implementation's alignment requirements for the **posix_dent** structure and shall be populated as follows:

- The value of the *d_ino* member shall be set to the file serial number of the file named by the *d_name* member.

- The value of the *d_reclen* member shall be set to the number of bytes occupied by this entry in *buf*, including any padding bytes needed before the next entry, if any. If this is the last entry in *buf*, *d_reclen* shall include any padding bytes needed to make the address of this entry plus *d_reclen* bytes satisfy the alignment requirements for the **posix_dent** structure.

- The value of the *d_type* member shall be set to indicate the file type of the named file, if the file type can be determined without needing to use the file serial number to obtain the file's metadata; otherwise it may be set to DT_UNKNOWN. If the file type is determined and it is one of the file types defined in this standard, the value of *d_type* shall be DT_BLK, DT_CHR, DT_DIR, DT_FIFO, DT_LNK, DT_REG, DT_SOCK, DT_MQ, DT_SEM, TYM DT_SHM, or DT_TMO (see **<dirent.h>**). If it is determined but is not a standard file type, the value of *d_type* shall not equal any of those listed here.

- The *d_name* member shall be a filename string, and (if not dot or dot-dot) shall contain the same byte sequence as the last pathname component of the string used to create the directory entry, plus the terminating NUL byte.

If the *d_name* member names a symbolic link, the values of the *d_ino* and *d_type* members shall be set to the values for the symbolic link itself.

The *posix_getdents*( ) function shall start reading at the current file offset in the open file description associated with *fildes*. On successful return, the file offset shall be incremented to point to the directory entry immediately following the last entry whose information was returned in *buf*, or to point to end-of-file if there are no more directory entries. On failure, the value of the file offset is unspecified. The current file offset can be set and retrieved using *lseek*( ) on the open file description associated with *fildes*. The behavior is unspecified if *lseek*( ) is used to set the file offset to a value other than zero or a value returned by a previous call to *lseek*( ) on the same open file description.

The *posix_getdents*( ) function shall not return directory entries containing empty names. If entries for dot or dot-dot exist, a sequence of calls that reads from offset zero to end-of-file shall return one entry for dot and one entry for dot-dot; otherwise, they shall not be returned.

Upon successful completion, *posix_getdents*( ) shall mark for update the last data access timestamp of the directory.

If *fildes* is a file descriptor associated with a directory stream opened using *fdopendir*( ) or *opendir*( ), the behavior is unspecified.

If *posix_getdents*( ) is called concurrently with an operation that adds, deletes, or modifies a directory entry, the results from *posix_getdents*( ) shall reflect either all of the effects of the concurrent operation or none of them. If a sequence of calls to *posix_getdents*( ) is made that reads from offset zero to end-of-file and a file is removed from or added to the directory between the first and last of those calls, whether the sequence of calls returns an entry for that file is unspecified.

RETURN VALUE

Upon successful completion, either a non-negative integer shall be returned indicating the number of bytes occupied by the **posix_dent** structures placed in *buf* or 0 shall be returned indicating the end of the directory was reached without any directory entries being placed in *buf*. Otherwise, −1 shall be returned and *errno* shall be set to indicate the error.

ERRORS

The *posix_getdents*( ) function shall fail if:

[EBADF] The *fildes* argument is not a valid file descriptor open for reading.

[EINVAL] The *nbyte* argument is not large enough to contain the information to be returned about the directory entry located at the current file offset.

[ENOENT] The current file offset is not located at a valid directory entry.

[ENOTDIR] The *fildes* argument is associated with a non-directory file.

[EOVERFLOW] One of the values in a structure to be placed in *buf* cannot be represented correctly.

The *posix_getdents*( ) function may fail if:

[EIO] A physical I/O error has occurred.

[ENOMEM] Insufficient memory was available to fulfill the request.

EXAMPLES

This example function lists the files in a specified directory with their file serial number and file type. If the file type is not available from *posix_getdents*( ), it is obtained using *fstatat*( ).

```
#include <dirent.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#define ENTBUFSIZ 10240

int list_dir(const char *dirnam)
{
    int fd = open(dirnam, O_RDONLY | O_DIRECTORY);
    if (fd == -1)
        return -1;

    char *buf = malloc(ENTBUFSIZ);
    if (buf == NULL)
    {
        close(fd);
        return -1;
    }

    ssize_t bytesinbuf;
    for(;;)
    {
        ssize_t nextent = 0;

        bytesinbuf = posix_getdents(fd, buf, ENTBUFSIZ, 0);
        if (bytesinbuf <= 0)
            break;

        do {
            const char *ftype;
            struct posix_dent *entp = (void *)&buf[nextent];
            if (entp->d_type == DT_UNKNOWN)
            {
                struct stat stbuf;
                if (fstatat(fd, entp->d_name, &stbuf,
                            AT_SYMLINK_NOFOLLOW) == -1)
                    ftype = "?";
                else
                    ftype = S_ISBLK(stbuf.st_mode) ? "b" :
                         S_ISCHR(stbuf.st_mode) ? "c" :
                         S_ISDIR(stbuf.st_mode) ? "d" :
                         S_ISFIFO(stbuf.st_mode) ? "p" :
                         S_ISLNK(stbuf.st_mode) ? "l" :
                         S_ISREG(stbuf.st_mode) ? "r" :
                         S_ISSOCK(stbuf.st_mode) ? "s" :
                         S_TYPEISMQ(&stbuf) ? "mq" :
                         S_TYPEISSEM(&stbuf) ? "sem" :
                         S_TYPEISSHM(&stbuf) ? "shm" :
#ifdef S_TYPEISTMO
                         S_TYPEISTMO(&stbuf) ? "tmo" :
#endif
                         "?";
            }
            else
            {
                ftype = entp->d_type == DT_BLK ? "b" :
                        entp->d_type == DT_CHR ? "c" :
                        entp->d_type == DT_DIR ? "d" :
                        entp->d_type == DT_FIFO ? "p" :
                        entp->d_type == DT_LNK ? "l" :
                        entp->d_type == DT_REG ? "r" :
                        entp->d_type == DT_SOCK ? "s" :
                        entp->d_type == DT_MQ ? "mq" :
```

```
                        entp->d_type == DT_SEM ? "sem" :
                        entp->d_type == DT_SHM ? "shm" :
#ifdef DT_TMO
                        entp->d_type == DT_TMO ? "tmo" :
#endif
                        "?";
            }

            printf("%ld\t%s\t%s\n", (long)entp->d_ino, ftype,
                entp->d_name);

            nextent += entp->d_reclen;

        } while (nextent < bytesinbuf);
    }

    close(fd);
    free(buf);
    return bytesinbuf;
}
```

APPLICATION USAGE

If an array of **posix_dent** structures (which is only possible on implementations where *d_name* is not a flexible array member) is used to provide the storage for *buf* in order to satisfy the alignment requirement, it should be noted that the number of array elements used to size the array may bear little or no relation to the number of directory entries that can be stored in it. It is recommended that the number of elements is calculated from the desired size in bytes, for example:

```
#define DESIREDSIZE 10240
struct posix_dent buf[DESIREDSIZE / sizeof(struct posix_dent) + 1];
size_t nbyte = sizeof buf;
```

When *posix_getdents*( ) is called with a *buf* that is not type **char** *, it is important to note that *d_reclen* is a byte count and therefore any pointer arithmetic involved in calculating the start of the next entry needs to use a **char** * pointer.

On implementations where directory entries in a directory take up more space than the corresponding **posix_dent** structures in *buf*, a call to *posix_getdents*( ) may read *nbyte* bytes from the directory, resulting (in most cases) in the actual number of bytes placed in *buf* being less than *nbyte*.

One advantage of *posix_getdents*( ) is that it provides the file type of each directory entry (if available), whereas *readdir*( ) only does so on implementations that have the file type as a non-standard additional member of the **dirent** structure. Knowing the file type can greatly reduce the number of *fstatat*( ) calls that need to be made when traversing the file hierarchy.

Whether or not a file's type can be determined without needing to use the file serial number to obtain the file's metadata may vary across the different file system types supported by an implementation. Therefore applications should not assume that if *d_type* contains known file types (i.e. not DT_UNKNOWN) for entries in a given directory then it will also contain known file types for entries in subdirectories of that directory or in its parent.

Since the *d_reclen* value for the last entry in *buf* includes padding to satisfy alignment requirements, applications can grow the buffer and call *posix_getdents*( ) again to append to it without needing to perform an alignment calculation.

RATIONALE

The *posix_getdents*( ) function was derived from existing *getdents*( ) functions but the name was changed because the existing *getdents*( ) functions differed in various ways, in particular the type of the second argument (structure pointer or **void** *), the members of the populated structures, and the error numbers used for some conditions. The name change also provided an opportunity to add a *flags* argument to provide for future extensibility.

Implementations are encouraged to include support for a DT_FORCE_TYPE flag which, when that bit is set in *flags*, causes *posix_getdents*( ) to look up the file type if it can not be obtained from the directory entry. This will allow applications that need to know the file type of every directory entry to keep the cost of these lookups to the minimum needed to obtain the type at the file system level, without the additional overhead of making a call to *fstatat*( ) for every file (that has *d_type* equal to DT_UNKNOWN).

Some existing *getdents*( ) or similar functions return directory entry structures for deleted directory entries in *buf*, marked with a special value of one of the structure members to distinguish them from non-deleted entries. This behavior is not allowed for *posix_getdents*( ), although the data from a deleted directory entry may be present in *buf* in the form of extra padding on the end of the previous entry.

FUTURE DIRECTIONS

A future version of this standard may add a DT_FORCE_TYPE flag as described in RATIONALE.

SEE ALSO

fdopendir( ), *fstatat*( ), *lseek*( ), *readdir*( )

XBD **<dirent.h>**

CHANGE HISTORY

First released in Issue 8.

NAME

ppoll — input/output multiplexing

SYNOPSIS

```
#include <poll.h>

int ppoll(struct pollfd fds[], nfds_t nfds,
        const struct timespec *restrict timeout,
        const sigset_t *restrict sigmask);
```

DESCRIPTION

Refer to *poll*( ).

of the *pselect*( ) call.

RETURN VALUE

Upon successful completion, the *pselect*( ) and *select*( ) functions shall return the total number of bits set in the bit masks. Otherwise, −1 shall be returned, and *errno* shall be set to indicate the error.

FD_CLR( ), *FD_SET*( ), and *FD_ZERO*( ) do not return a value. *FD_ISSET*( ) shall return a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

ERRORS

Under the following conditions, *pselect*( ) and *select*( ) shall fail and set *errno* to:

[EBADF] One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

[EINTR] The function was interrupted by a signal.

If SA_RESTART has been set for the interrupting signal, it is implementation-defined whether the function restarts or returns with [EINTR].

[EINVAL] An invalid timeout interval was specified.

[EINVAL] The *nfds* argument is less than 0 or greater than FD_SETSIZE.

EXAMPLES

None.

APPLICATION USAGE

The use of *select*( ) and *pselect*( ) requires that the application construct the set of file descriptors to work on each time through a polling loop, and is inherently limited from operating on file descriptors larger than FD_SETSIZE. Also, the amount of work to perform scales as *nfds* increases, even if the number of file descriptors selected within the larger set remains the same. Thus, applications may wish to consider using *poll*( ) and *ppoll*( ) instead, for better scaling.

RATIONALE

In earlier versions of the Single UNIX Specification, the *select*( ) function was defined in the **<sys/time.h>** header. This is now changed to **<sys/select.h>**. The rationale for this change was as follows: the introduction of the *pselect*( ) function included the **<sys/select.h>** header and the **<sys/select.h>** header defines all the related definitions for the *pselect*( ) and *select*( ) functions. Backwards-compatibility to existing XSI implementations is handled by allowing **<sys/time.h>** to include **<sys/select.h>**.

Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers can install an additional cancellation handler which resets the signal mask to the expected value.

```
void cleanup(void *arg)
{
    sigset_t *ss = (sigset_t *) arg;
    pthread_sigmask(SIG_SETMASK, ss, NULL);
}

int call_pselect(int nfds, fd_set *readfds, fd_set *writefds,
    fd_set errorfds, const struct timespec *timeout,
    const sigset_t *sigmask)
{
    sigset_t oldmask;
    int result;
```

NAME

pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The *pthread_cond_broadcast*( ) function shall unblock all threads currently blocked on the specified condition variable *cond*.

The *pthread_cond_signal*( ) function shall unblock at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) returns from its call to *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ), the thread shall own the mutex with which it called *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ). The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable), and as if each had called *pthread_mutex_lock*( ).

The *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) functions may be called by a thread whether or not it currently owns the mutex that threads calling *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ).

The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions shall have no effect if there are no threads currently blocked on *cond*.

The behavior is undefined if the value specified by the *cond* argument to *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) does not refer to an initialized condition variable.

RETURN VALUE

If successful, the *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_cond_broadcast*( ) function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task. Consider a single producer/multiple consumer problem, where the producer can insert multiple items on a list that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast*( ) function, the producer would notify all consumers that might be waiting, and thereby the application would receive more throughput on a multi-processor. In addition, *pthread_cond_broadcast*( ) makes it easier to implement a read-write lock. The *pthread_cond_broadcast*( ) function is needed in order to wake up all waiting readers when a writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function to notify all clients of an impending transaction commit.

It is not safe to use the *pthread_cond_signal*( ) function in a signal handler that is invoked asynchronously. Even if it were safe, there would still be a race between the test of the Boolean *pthread_cond_wait*( ) that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling from code running in a signal handler.

RATIONALE

If an implementation detects that the value specified by the *cond* argument to *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) does not refer to an initialized condition variable, it is recommended that the function should fail and report an [EINVAL] error.

Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of *pthread_cond_signal*( ) to avoid the unblocking of more than one thread blocked on a condition variable. For example, consider the following partial implementation of *pthread_cond_wait*( ) and *pthread_cond_signal*( ), executed by two threads in the order given. One thread is trying to wait on the condition variable, another is concurrently executing *pthread_cond_signal*( ), while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++; /* 4 */
    if (cond->waiter) { /* 5 */
        sleeper = cond->waiter; /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper); /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) as a result of one call to *pthread_cond_signal*( ). This effect is called ``spurious wakeup''. Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call *pthread_cond_wait*( ) after the sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a

NAME

pthread_cond_clockwait, pthread_cond_timedwait, pthread_cond_wait — wait on a condition

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_clockwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex, clockid_t clock_id,
    const struct timespec *restrict abstime);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

DESCRIPTION

The *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), and *pthread_cond_wait*( ) functions shall block on a condition variable. The application shall ensure that these functions are called with *mutex* locked by the calling thread; otherwise, an error (for PTHREAD_MUTEX_ERRORCHECK and robust mutexes) or undefined behavior (for other mutexes) results.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means ``atomically with respect to access by another thread to the mutex and then the condition variable''. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

If *mutex* is a robust mutex where an owner terminated while holding the lock and the state is recoverable, the mutex shall be acquired even though the function returns [EOWNERDEAD].

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) functions may occur. Since the return from *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

When a thread waits on a condition variable, having specified a particular mutex to the *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) operation, a dynamic binding is formed between that mutex and condition variable that remains in effect as long as at least one thread is blocked on the condition variable. During this time, the effect of an attempt by any thread to wait on that condition variable using a different mutex is undefined. Once all waiting threads have been unblocked (as by the *pthread_cond_broadcast*( ) operation), the next wait operation on that condition variable shall form a new dynamic binding with the mutex specified by that wait operation. Even though the dynamic binding between condition variable and mutex may be removed or replaced between the time a thread is unblocked from a wait on the condition variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked thread shall always re-acquire the mutex specified in the condition wait operation call from which it is returning.

A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a thread is set to PTHREAD_CANCEL_DEFERRED, a side-effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ), but at that point notices the cancellation request and, instead of returning to the caller, starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The *pthread_cond_timedwait*( ) function shall be equivalent to *pthread_cond_wait*( ), except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time specified by *abstime* has already been passed at the time of the call. When such timeouts occur, *pthread_cond_timedwait*( ) shall nonetheless release and re-acquire the mutex referenced by *mutex*, and may consume a condition signal directed concurrently at the condition variable.

The condition variable shall have a clock attribute which specifies the clock that shall be used by *pthread_cond_timedwait*( ) to measure the time specified by the *abstime* argument. The *pthread_cond_timedwait*( ) function is also a cancellation point.

The *pthread_cond_clockwait*( ) function shall be equivalent to *pthread_cond_timedwait*( ), except that the absolute time specified by *abstime* is measured against the clock indicated by *clock_id* rather than the clock specified in the condition variable's clock attribute. Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to *pthread_cond_clockwait*( ) as the *clock_id* argument.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

The behavior is undefined if the value specified by the *cond* or *mutex* argument to these functions does not refer to an initialized condition variable or an initialized mutex object, respectively.

RETURN VALUE

Except for [ETIMEDOUT], [ENOTRECOVERABLE], and [EOWNERDEAD], all these error checks shall act as if they were performed immediately at the beginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable specified by *cond*.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions shall fail if:

[EAGAIN] The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.

[ENOTRECOVERABLE]
The state protected by the mutex is not recoverable.

[EOWNERDEAD]
The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state

consistent.

[EPERM] The mutex type is PTHREAD_MUTEX_ERRORCHECK or the mutex is a robust mutex, and the current thread does not own the mutex.

The *pthread_cond_clockwait*( ) and *pthread_cond_timedwait*( ) functions shall fail if:

[ETIMEDOUT] The time specified by *abstime* has passed.

[EINVAL] The *abstime* argument specified a nanosecond value less than zero or greater than or equal to 1000 million, or the *clock_id* argument passed to *pthread_cond_clockwait*( ) is invalid or not supported.

These functions may fail if:

[EOWNERDEAD]
The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

RATIONALE

If an implementation detects that the value specified by the *cond* argument to *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) does not refer to an initialized condition variable, or detects that the value specified by the *mutex* argument does not refer to an initialized mutex object, it is recommended that the function should fail and report an [EINVAL] error.

Condition Wait Semantics

It is important to note that when *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), and *pthread_cond_wait*( ) return without error, the associated predicate may still be false. Similarly, when *pthread_cond_clockwait*( ) or *pthread_cond_timedwait*( ) returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

The application needs to recheck the predicate on any return because it cannot be sure there is another thread waiting on the thread to handle the signal, and if there is not then the signal is lost. The burden is on the application to check the predicate.

Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equivalent of a ``while loop'' that checks the predicate.

Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons. First, a relative time measure can be easily implemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts. For example, assume that *clock_gettime*( ) returns the current time and *cond_relative_timed_wait*( ) uses relative timeouts:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time -now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that implementations process any timed wait expiring at an intervening time as if that time had actually occurred.

Choice of Clock

Care should be taken to decide which clock is most appropriate when waiting with a timeout. The system clock CLOCK_REALTIME, as used by default with *pthread_cond_timedwait*( ), may be subject to jumps forwards and backwards in order to correct it against actual time. CLOCK_MONOTONIC is guaranteed not to jump backwards and must also advance in real time, so using it via *pthread_cond_clockwait*( ) or *pthread_condattr_setclock*( ) may be more appropriate.

Cancellation and Condition Wait

A condition wait, whether timed or not, is a cancellation point. That is, the functions *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), and *pthread_cond_wait*( ) are points where a pending (or concurrent) cancellation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancellation point, the thread can be canceled and perform its cancellation cleanup handler even though it may be stuck in some indefinite wait.

A side-effect of acting on a cancellation request while a thread is blocked on a condition variable is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in order to ensure that the cancellation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to POSIX threads from languages, such as Ada or C++, which may choose to map cancellation onto a language exception; this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

Timed Condition Wait

The *pthread_cond_clockwait*( ) and *pthread_cond_timedwait*( ) functions allow an application to give up waiting for a particular condition after a given amount of time. An example follows:

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_clockwait(&t.cond, &t.mn,
            CLOCK_MONOTONIC, &ts);
    t.waiters--;
    if (rc == 0 || mypredicate(&t))
        setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

By making the timeout parameter absolute, it does not need to be recomputed each time the program checks its blocking predicate. If the timeout was relative, it would have to be recomputed before each call. This would be especially difficult since such code would need to take into account the possibility of extra wakeups that result from extra broadcasts or signals on the condition variable that occur before either the predicate is true or the timeout is due. Using CLOCK_MONOTONIC rather than CLOCK_REALTIME means that the timeout is not influenced by the system clock being changed.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_broadcast( )

XBD Section 4.13 (on page 91), **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions are marked as part of the Threads option.

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the *pthread_cond_wait*( ) function.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for the Clock Selection option.

The ERRORS section has an additional case for [EPERM] in response to IEEE PASC Interpretation 1003.1c #28.

The **restrict** keyword is added to the *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) prototypes for alignment with the ISO/IEC 9899: 1999 standard.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/89 is applied, updating the DESCRIPTION for consistency with the *pthread_cond_destroy*( ) function that states it is safe to destroy an initialized condition variable upon which no threads are currently blocked.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/90 is applied, updating words in the DESCRIPTION from ``the cancelability enable state'' to ``the cancelability type''.

NAME

pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection condition variable attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
    clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

DESCRIPTION

The *pthread_condattr_getclock*( ) function shall obtain the value of the *clock* attribute from the attributes object referenced by *attr*.

The *pthread_condattr_setclock*( ) function shall set the *clock* attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock*( ) is called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of *pthread_cond_timedwait*( ). The default value of the *clock* attribute shall refer to the system clock. The *clock* attribute shall have no effect on the *pthread_cond_clockwait*( ) function.

The behavior is undefined if the value specified by the *attr* argument to *pthread_condattr_getclock*( ) or *pthread_condattr_setclock*( ) does not refer to an initialized condition variable attributes object.

RETURN VALUE

If successful, the *pthread_condattr_getclock*( ) function shall return zero and store the value of the clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_condattr_setclock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_condattr_setclock*( ) function may fail if:

[EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-time clock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

If an implementation detects that the value specified by the *attr* argument to *pthread_condattr_getclock*( ) or *pthread_condattr_setclock*( ) does not refer to an initialized condition variable attributes object, it is recommended that the function should fail and report an [EINVAL] error.

NAME

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

The *pthread_mutex_destroy*( ) function shall destroy the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy*( ) to set the object referenced by *mutex* to an invalid value.

A destroyed mutex object can be reinitialized using *pthread_mutex_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) call by another thread, results in undefined behavior.

The *pthread_mutex_init*( ) function shall initialize the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

See Section 2.9.9 (on page 508) for further requirements.

Attempting to initialize an already initialized mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes. The effect shall be equivalent to dynamic initialization by a call to *pthread_mutex_init*( ) with parameter *attr* specified as NULL, except that no error checks are performed.

The behavior is undefined if the value specified by the *mutex* argument to *pthread_mutex_destroy*( ) does not refer to an initialized mutex.

The behavior is undefined if the value specified by the *attr* argument to *pthread_mutex_init*( ) does not refer to an initialized mutex attributes object.

RETURN VALUE

If successful, the *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_init*( ) function shall fail if:

[EAGAIN] The system lacked the necessary resources (other than memory) to initialize another mutex.

[ENOMEM] Insufficient memory exists to initialize the mutex.

[EPERM] The caller does not have the privilege to perform the operation.

The *pthread_mutex_init*( ) function may fail if:

[EINVAL] The attributes object referenced by *attr* has the robust mutex attribute set without the process-shared attribute being set.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

If an implementation detects that the value specified by the *mutex* argument to *pthread_mutex_destroy*( ) does not refer to an initialized mutex, it is recommended that the function should fail and report an [EINVAL] error.

If an implementation detects that the value specified by the *mutex* argument to *pthread_mutex_destroy*( ) or *pthread_mutex_init*( ) refers to a locked mutex or a mutex that is referenced (for example, while being used in a *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) call) by another thread, or detects that the value specified by the *mutex* argument to *pthread_mutex_init*( ) refers to an already initialized mutex, it is recommended that the function should fail and report an [EBUSY] error.

If an implementation detects that the value specified by the *attr* argument to *pthread_mutex_init*( ) does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an [EINVAL] error.

Alternate Implementations Possible

This volume of POSIX.1-202x supports several alternative implementations of mutexes. An implementation may store the lock directly in the object of type **pthread_mutex_t**. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice, this volume of POSIX.1-202x does not define assignment or equality for this type, and it uses the term ``initialize'' to reinforce the (more restrictive) notion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opaqueness of the type.

An implementation is permitted, but not required, to have *pthread_mutex_destroy*( ) store an illegal value into the mutex. This may help detect erroneous programs that try to lock (or otherwise reference) a mutex that has already been destroyed.

Tradeoff Between Error Checks and Performance Supported

Many error conditions that can occur are not required to be detected by the implementation in order to let implementations trade off performance *versus* degree of error checking according to the needs of their specific applications and execution environment. As a general rule, conditions caused by the system (such as insufficient memory) are required to be detected, but conditions caused by an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are specified to result in undefined behavior.

A wide range of implementations is thus made possible. For example, an implementation particular, it can happen at most as many times as there are statically allocated synchronization objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init*( ) or *pthread_cond_init*( ).

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient performance for an application on some implementation, the application can avoid static initialization altogether by explicitly initializing all synchronization objects with the corresponding *pthread_*_init*( ) functions, which are supported by all implementations. An implementation can also document the tradeoffs and advise which initialization technique is more efficient for that particular implementation.

Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. However, since attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) call by another thread, results in undefined behavior, care must be taken to ensure that no other thread may be referencing the mutex.

Robust Mutexes

Implementations are required to provide robust mutexes for mutexes with the process-shared attribute set to PTHREAD_PROCESS_SHARED. Implementations are allowed, but not required, to provide robust mutexes when the process-shared attribute is set to PTHREAD_PROCESS_PRIVATE.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_getprioceiling( ), *pthread_mutexattr_getrobust*( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), *pthread_mutexattr_getpshared*( )

XBD **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions are marked as part of the Threads option.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The **restrict** keyword is added to the *pthread_mutex_init*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

Issue 7

Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

The *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions are moved from the Threads option to the Base.

The [EINVAL] error for an uninitialized mutex or an uninitialized mutex attributes object is removed; this condition results in undefined behavior.

NAME

pthread_mutex_clocklock, pthread_mutex_timedlock — lock a mutex

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_clocklock(pthread_mutex_t *restrict mutex,
    clockid_t clock_id, const struct timespec *restrict abstime);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
```

DESCRIPTION

The *pthread_mutex_clocklock*( ) and *pthread_mutex_timedlock*( ) functions shall lock the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread shall block until the mutex becomes available as in the *pthread_mutex_lock*( ) function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.

The timeout shall expire when the absolute time specified by *abstime* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abstime*), or if the absolute time specified by *abstime* has already been passed at the time of the call.

For *pthread_mutex_timedlock*( ), the timeout shall be based on the CLOCK_REALTIME clock. For *pthread_mutex_clocklock*( ), the timeout shall be based on the clock specified by the *clock_id* argument. The resolution of the timeout shall be the resolution of the clock on which it is based. Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to *pthread_mutex_clocklock*( ) as the *clock_id* argument.

Under no circumstance shall the function fail with a timeout if the mutex can be locked immediately. The validity of the *abstime* parameter need not be checked if the mutex can be locked immediately.

RPI|TPI As a consequence of the priority inheritance rules (for mutexes initialized with the PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

If *mutex* is a robust mutex and the process containing the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_clocklock*( ) or *pthread_mutex_timedlock*( ) shall return the error value [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_clocklock*( ) or *pthread_mutex_timedlock*( ) may return the error value [EOWNERDEAD] even if the process in which the owning thread resides has not terminated. In these cases, the mutex is locked by the thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call *pthread_mutex_consistent*( ). If the application is unable to recover the state, it should unlock the mutex without a prior call to *pthread_mutex_consistent*( ), after which the mutex is marked permanently unusable.

If *mutex* does not refer to an initialized mutex object, the behavior is undefined.

RETURN VALUE

If successful, the *pthread_mutex_clocklock*( ) and *pthread_mutex_timedlock*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_clocklock*( ) and *pthread_mutex_timedlock*( ) functions shall fail if:

[EAGAIN] The mutex could not be acquired because the maximum number of recursive locks for *mutex* has been exceeded.

[EAGAIN] The mutex is a robust mutex and the system resources available for robust mutexes owned would be exceeded.

[EDEADLK] The mutex type is PTHREAD_MUTEX_ERRORCHECK and the current thread already owns the mutex.

[EINVAL] The mutex was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex' current priority ceiling.

[EINVAL] The process or thread would have blocked, and either the *abstime* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million, or the *pthread_mutex_clocklock*( ) function was passed an invalid or unsupported *clock_id* value.

[ENOTRECOVERABLE]
The state protected by the mutex is not recoverable.

[EOWNERDEAD]
The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

[ETIMEDOUT] The mutex could not be locked before the specified timeout expired.

The *pthread_mutex_clocklock*( ) and *pthread_mutex_timedlock*( ) functions may fail if:

[EDEADLK] A deadlock condition was detected.

[EOWNERDEAD]
The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

RATIONALE

Refer to *pthread_mutex_lock*( ).

NAME

pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set the mutex type attribute

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
    int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

DESCRIPTION

The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions, respectively, shall get and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types include:

PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_RECURSIVE
PTHREAD_MUTEX_DEFAULT

The mutex type affects the behavior of calls which lock and unlock the mutex. See *pthread_mutex_lock*( ) for details. An implementation may map PTHREAD_MUTEX_DEFAULT to one of the other mutex types.

The behavior is undefined if the value specified by the *attr* argument to *pthread_mutexattr_gettype*( ) or *pthread_mutexattr_settype*( ) does not refer to an initialized mutex attributes object.

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_gettype*( ) function shall return zero and store the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise, an error shall be returned to indicate the error.

If successful, the *pthread_mutexattr_settype*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_settype*( ) function shall fail if:

[EINVAL] The value *type* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed in a *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) call may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

NAME

pthread_rwlock_clockrdlock, pthread_rwlock_timedrdlock — lock a read-write lock for reading

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_clockrdlock(pthread_rwlock_t *restrict rwlock,
    clockid_t clock_id, const struct timespec *restrict abstime);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abstime);
```

DESCRIPTION

The *pthread_rwlock_clockrdlock*( ) and *pthread_rwlock_timedrdlock*( ) functions shall apply a read lock to the read-write lock referenced by *rwlock* as in the *pthread_rwlock_rdlock*( ) function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait shall be terminated when the specified timeout expires. The timeout shall expire when the absolute time specified by *abstime* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abstime*), or if the absolute time specified by *abstime* has already been passed at the time of the call.

For *pthread_rwlock_timedrdlock*( ), the timeout shall be based on the CLOCK_REALTIME clock. For *pthread_rwlock_clockrdlock*( ), the timeout shall be based on the clock specified by the *clock_id* argument. The resolution of the timeout shall be the resolution of the clock on which it is based. Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to *pthread_rwlock_clockrdlock*( ) as the *clock_id* argument.

Under no circumstances shall the function fail with a timeout if the lock can be acquired immediately. The validity of the *abstime* parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to *pthread_rwlock_clockrdlock*( ) or *pthread_rwlock_timedrdlock*( ), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*. The results are undefined if this function is called with an uninitialized read-write lock.

RETURN VALUE

The *pthread_rwlock_clockrdlock*( ) and *pthread_rwlock_timedrdlock*( ) functions shall return zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_clockrdlock*( ) and *pthread_rwlock_timedrdlock*( ) functions shall fail if:

[ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

The *pthread_rwlock_clockrdlock*( ) and *pthread_rwlock_timedrdlock*( ) functions may fail if:

[EAGAIN] The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.

[EDEADLK] A deadlock condition was detected or the calling thread already holds a write lock on *rwlock*.

[EINVAL] The *abstime* nanosecond value is less than zero or greater than or equal to 1 000 million, or the *pthread_rwlock_clockrdlock*( ) function was passed an invalid or unsupported *clock_id* value.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in XBD Section 3.260 (on page 66).

RATIONALE

If an implementation detects that the value specified by the *rwlock* argument to *pthread_rwlock_clockrdlock*( ) or *pthread_rwlock_timedrdlock*( ) does not refer to an initialized read-write lock object, it is recommended that the function should fail and report an [EINVAL] error. +

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( )

XBD Section 3.260 (on page 66), Section 4.13 (on page 91), **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/102 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

The *pthread_rwlock_timedrdlock*( ) function is moved from the Timeouts option to the Base.

The [EINVAL] error for an uninitialized read-write lock object is removed; this condition results in undefined behavior.

Issue 8

Austin Group Defect 592 is applied, removing text relating to **<time.h>** from the SYNOPSIS and DESCRIPTION sections. +

Austin Group Defect 1216 is applied, adding *pthread_rwlock_clockrdlock*( ). |

NAME

pthread_rwlock_clockwrlock, pthread_rwlock_timedwrlock — lock a read-write lock for writing

SYNOPSIS

```
#include <pthread.h>

int pthread_rwlock_clockwrlock(pthread_rwlock_t *restrict rwlock,
    clockid_t clock_id, const struct timespec *restrict abstime);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abstime);
```

DESCRIPTION

The *pthread_rwlock_clockwrlock*( ) and *pthread_rwlock_timedwrlock*( ) functions shall apply a write lock to the read-write lock referenced by *rwlock* as in the *pthread_rwlock_wrlock*( ) function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait shall be terminated when the specified timeout expires. The timeout shall expire when the absolute time specified by *abstime* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abstime*), or if the absolute time specified by *abstime* has already been passed at the time of the call.

For *pthread_rwlock_timedwrlock*( ), the timeout shall be based on the CLOCK_REALTIME clock. For *pthread_rwlock_clockwrlock*( ), the timeout shall be based on the clock specified by the *clock_id* argument. The resolution of the timeout shall be the resolution of the clock on which it is based. Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to *pthread_rwlock_clockwrlock*( ) as the *clock_id* argument.

Under no circumstances shall the function fail with a timeout if the lock can be acquired immediately. The validity of the *abstime* parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to *pthread_rwlock_clockwrlock*( ) or *pthread_rwlock_timedwrlock*( ), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

The calling thread may deadlock if at the time the call is made it holds the read-write lock. The results are undefined if this function is called with an uninitialized read-write lock.

RETURN VALUE

The *pthread_rwlock_clockwrlock*( ) and *pthread_rwlock_timedwrlock*( ) functions shall return zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_clockwrlock*( ) and *pthread_rwlock_timedwrlock*( ) functions shall fail if:

[ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

The *pthread_rwlock_clockwrlock*( ) and *pthread_rwlock_timedwrlock*( ) functions may fail if:

[EDEADLK] A deadlock condition was detected or the calling thread already holds the *rwlock*.

[EINVAL] The *abstime* nanosecond value is less than zero or greater than or equal to 1 000 million, or the *pthread_rwlock_clockwrlock*( ) function was passed an invalid or unsupported *clock_id* value.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in XBD Section 3.260 (on page 66).

RATIONALE

If an implementation detects that the value specified by the *rwlock* argument to *pthread_rwlock_clockwrlock*( ) or *pthread_rwlock_timedwrlock*( ) does not refer to an initialized read-write lock object, it is recommended that the function should fail and report an [EINVAL] error. +

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( )

XBD Section 3.260 (on page 66), Section 4.13 (on page 91), **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/103 is applied, updating the ERRORS section so that the [EDEADLK] error includes detection of a deadlock condition.

Issue 7

The *pthread_rwlock_timedwrlock*( ) function is moved from the Timeouts option to the Base.

The [EINVAL] error for an uninitialized read-write lock object is removed; this condition results in undefined behavior.

Issue 8

Austin Group Defect 592 is applied, removing text relating to **<time.h>** from the SYNOPSIS and DESCRIPTION sections. +

Austin Group Defect 1216 is applied, adding *pthread_rwlock_clockwrlock*( ). |

NAME

qsort, qsort_r — sort a table of data

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
    int (*compar)(const void *, const void *));
void qsort_r(void *base, size_t nel, size_t width,
    int (*compar)(const void *, const void *, void *), void *arg);
```

DESCRIPTION

For *qsort*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-202x defers to the ISO C standard.

The *qsort*( ) function shall sort an array of *nel* objects, the initial element of which is pointed to by *base*. The size of each object, in bytes, is specified by the *width* argument. If the *nel* argument has the value zero, the comparison function pointed to by *compar* shall not be called and no rearrangement shall take place.

The application shall ensure that the comparison function pointed to by *compar* does not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

When the same objects (consisting of width bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, they shall define a total ordering on the array.

The contents of the array shall be sorted in ascending order according to a comparison function. The *compar* argument is a pointer to the comparison function, which is called with two arguments that point to the elements being compared. The application shall ensure that the function returns an integer less than, equal to, or greater than 0, if the first argument is considered respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

The *qsort_r*( ) function shall be identical to *qsort*( ) except that the comparison function *compar* takes a third argument. The *arg* opaque pointer passed to *qsort_r*( ) shall in turn be passed as the third argument to the comparison function.

RETURN VALUE

These functions shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

If the *compar* callback function requires any additional state outside of the items being sorted, it can only access this state through global variables, making it potentially unsafe to use *qsort*( ) with the same *compar* function from separate threads at the same time. The *qsort_r*( ) function was added with the ability to pass through arbitrary arguments to the comparator, which avoids the need to access global variables and thus making it possible to safely share a stateful comparator across threads.

RATIONALE

The requirement that each argument (hereafter referred to as *p)* to the comparison function is a pointer to elements of the array implies that for every call, for each argument separately, all of the following expressions are non-zero:

```
((char *)p - (char *)base) % width == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nel * width
```

FUTURE DIRECTIONS

None.

SEE ALSO

alphasort( )

XBD **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The normative text is updated to avoid use of the term ``must'' for application requirements.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/49 is applied, adding the last sentence to the first non-shaded paragraph in the DESCRIPTION, and the following two paragraphs. The RATIONALE is also updated. These changes are for alignment with the ISO C standard. +

Issue 8 +

Austin Group Defect 900 is applied, adding the *qsort_r*( ) function. |

```
                keystr[len++] = c;
        }
        keystr[len] = '\0';
        printf("%s Element%0*ld\n", keystr, elementlen, i);
        len = 0;
    }
```

Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```
static unsigned long next = 1;
int myrand(void)  /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
void mysrand(unsigned seed)
{
    next = seed;
}
```

APPLICATION USAGE

These functions should be avoided whenever non-trivial requirements (including safety) have to be fulfilled, unless seeded using *getentropy*( ).

The *drand48*( ) and *random*( ) functions provide much more elaborate pseudo-random number generators.

RATIONALE

The ISO C standard *rand*( ) and *srand*( ) functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to *rand*( ), there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call *rand*( )

2. A different sequence of pseudo-random numbers for each thread that calls *rand*( )

This is provided by the modified thread-safe function based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the *rand*( ) function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient function.

FUTURE DIRECTIONS

None.

NAME

realloc, reallocarray — memory reallocators

SYNOPSIS

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
CX void *reallocarray(void *ptr, size_t nelem, size_t elsize);
```

DESCRIPTION

CX For *realloc*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-202x defers to the ISO C standard.

The *realloc*( ) function shall deallocate the old object pointed to by *ptr* and return a pointer to a new object that has the size specified by *size*. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values. If the size of the space requested is zero, the behavior shall be implementation-defined: either a null pointer is returned, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object. If the space cannot be allocated, the object shall remain unchanged.

CX The *reallocarray*( ) function shall be equivalent to the call `realloc(`*`ptr`*`, `*`nelem`*` * `*`elsize`*`)` except that overflow in the multiplication shall be an error.

CX If *ptr* is a null pointer, *realloc*( ) or *reallocarray*( ) shall be equivalent to *malloc*( ) for the specified size.

If *ptr* does not match a pointer earlier returned by a function in POSIX.1-202x that allocates memory as if by *malloc*( ), or if the space has previously been deallocated by a call to *free*( ), CX *realloc*( ), or *reallocarray*( ), the behavior is undefined.

CX The order and contiguity of storage allocated by successive calls to *realloc*( ) or *reallocarray*( ) is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

RETURN VALUE

CX Upon successful completion, *realloc*( ) and *reallocarray*( ) shall return a pointer to the (possibly CX moved) allocated space. If *size* is 0, or either *nelem* or *elsize* is 0, then either:

- CX A null pointer shall be returned and, if *ptr* is not a null pointer, *errno* shall be set to an implementation-defined value.
- A pointer to the allocated space shall be returned, and the memory object pointed to by *ptr* shall be freed. The application shall ensure that the pointer is not used to access an object.

CX If there is not enough available memory, *realloc*( ) and *reallocarray*( ) shall return a null pointer CX and set *errno* to [ENOMEM]. If *realloc*( ) or *reallocarray*( ) returns a null pointer and *errno* has been set to [ENOMEM], the memory referenced by *ptr* shall not be changed.

ERRORS

CX The *realloc*( ) and *reallocarray*( ) functions shall fail if:

CX [ENOMEM] Insufficient memory is available.

CX The *reallocarray*( ) function shall fail if:

[ENOMEM] The calculation *nelem* * *elsize* would overflow.

EXAMPLES

None.

APPLICATION USAGE

The description of *realloc*( ) has been modified from previous versions of this standard to align with the ISO/IEC 9899:1999 standard. Previous versions explicitly permitted a call to *realloc*(*p*, 0) to free the space pointed to by *p* and return a null pointer. While this behavior could be interpreted as permitted by this version of the standard, the C language committee has indicated that this interpretation is incorrect. Applications should assume that if *realloc*( ) returns a null pointer, the space pointed to by *p* has not been freed. Since this could lead to double-frees, implementations should also set *errno* if a null pointer actually indicates a failure, and applications should only free the space if *errno* was changed.

RATIONALE

None.

FUTURE DIRECTIONS

This standard defers to the ISO C standard. While that standard currently has language that might permit *realloc*(*p*, 0), where *p* is not a null pointer, to free *p* while still returning a null pointer, the committee responsible for that standard is considering clarifying the language to explicitly prohibit that alternative.

SEE ALSO

calloc( ), *free*( ), *malloc*( )

XBD **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, if there is not enough available memory, the setting of *errno* to [ENOMEM] is added.
- The [ENOMEM] error condition is added.

Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0495 [400], XSH/TC1-2008/0496 [400], XSH/TC1-2008/0497 [400], and XSH/TC1-2008/0498 [400] are applied.

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0309 [526] and XSH/TC2-2008/0310 [526,688] are applied.

NAME

sem_init — initialize an unnamed semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

DESCRIPTION

The *sem_init*( ) function shall initialize the unnamed semaphore referred to by *sem*. The value of the initialized semaphore shall be *value*. Following a successful call to *sem_init*( ), the semaphore can be used in subsequent calls to *sem_clockwait*( ), *sem_destroy*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), and *sem_wait*( ). This semaphore shall remain usable until the semaphore is destroyed. An unnamed semaphore may be implemented using a file descriptor.

If the *pshared* argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore *sem* can use *sem* for performing *sem_clockwait*( ), *sem_destroy*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), and *sem_wait*( ) operations.

If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing *sem_clockwait*( ), *sem_destroy*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), and *sem_wait*( ) operations.

See Section 2.9.9 (on page 508) for further requirements.

Attempting to initialize an already initialized semaphore results in undefined behavior.

RETURN VALUE

Upon successful completion, the *sem_init*( ) function shall initialize the semaphore in *sem* and return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *sem_init*( ) function shall fail if:

[EINVAL] The *value* argument exceeds {SEM_VALUE_MAX}.

[ENOSPC] A resource required to initialize the semaphore has been exhausted, or the limit on semaphores ({SEM_NSEMS_MAX}) has been reached.

[EPERM] The process lacks appropriate privileges to initialize the semaphore.

The *sem_init*( ) function may fail if:

[EMFILE] All file descriptors available to the process are currently open.

[ENFILE] The maximum allowable number of files is currently open in the system.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

NAME

sem_open — initialize and open a named semaphore

SYNOPSIS

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);
```

DESCRIPTION

The *sem_open*( ) function shall establish a connection between a named semaphore and a process. A named semaphore may be implemented using a file descriptor. Following a call to *sem_open*( ) with semaphore name *name*, the process may reference the semaphore associated with *name* using the address returned from the call. This semaphore can be used in subsequent calls to *sem_clockwait*( ), *sem_close*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), and *sem_wait*( ). The semaphore remains usable by this process until the semaphore is closed by a successful call to *sem_close*( ), *_exit*( ), or one of the *exec* functions.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to *sem_open*( ). The following flag bits may be set in *oflag*:

O_CREAT This flag is used to create a semaphore if it does not already exist. If O_CREAT is set and the semaphore already exists, then O_CREAT has no effect, except as noted under O_EXCL. Otherwise, *sem_open*( ) creates a named semaphore. The O_CREAT flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and *value*, which is of type **unsigned**. The semaphore is created with an initial value of *value*. Valid initial values for semaphores are less than or equal to {SEM_VALUE_MAX}.

The user ID of the semaphore shall be set to the effective user ID of the process. The group ID of the semaphore shall be set to the effective group ID of the process; however, if the *name* argument is visible in the file system, the group ID may be set to the group ID of the containing directory. The permission bits of the semaphore are set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are specified, the effect is unspecified.

After the semaphore named *name* has been created by *sem_open*( ) with the O_CREAT flag, other processes can connect to the semaphore by calling *sem_open*( ) with the same value of *name*.

O_EXCL If O_EXCL and O_CREAT are set, *sem_open*( ) fails if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing *sem_open*( ) with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the effect is undefined.

If flags other than O_CREAT and O_EXCL are specified in the *oflag* parameter, the effect is unspecified.

The *name* argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname, except that the interpretation of <slash> characters other than the leading <slash> character in *name* is implementation-defined, and that the length limits for the *name* argument are implementation-defined and need not be the same as the pathname limits {PATH_MAX} and {NAME_MAX}. If *name* begins with the <slash> character, then processes calling *sem_open*( ) with the same value of *name* shall refer to the same semaphore object, as long as that name has not been removed. If

NAME

sem_clockwait, sem_timedwait — lock a semaphore

SYNOPSIS

```
#include <semaphore.h>

int sem_clockwait(sem_t *restrict sem, clockid_t clock_id,
    const struct timespec *restrict abstime);
int sem_timedwait(sem_t *restrict sem,
    const struct timespec *restrict abstime);
```

DESCRIPTION

The *sem_clockwait*( ) and *sem_timedwait*( ) functions shall lock the semaphore referenced by *sem* as in the *sem_wait*( ) function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a *sem_post*( ) function, this wait shall be terminated when the specified timeout expires.

The timeout shall expire when the absolute time specified by *abstime* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abstime*), or if the absolute time specified by *abstime* has already been passed at the time of the call.

For *sem_timedwait*( ), the timeout shall be based on the CLOCK_REALTIME clock. For *sem_clockwait*( ), the timeout shall be based on the clock specified by the *clock_id* argument. The resolution of the timeout shall be the resolution of the clock on which it is based. Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to *sem_clockwait*( ) as the *clock_id* argument.

Under no circumstance shall the function fail with a timeout if the semaphore can be locked immediately. The validity of the *abstime* need not be checked if the semaphore can be locked immediately.

RETURN VALUE

The *sem_clockwait*( ) and *sem_timedwait*( ) functions shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the functions shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sem_clockwait*( ) and *sem_timedwait*( ) functions shall fail if:

[EINVAL] The process or thread would have blocked, and either the *abstime* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million, or the *sem_clockwait*( ) function was passed an invalid or unsupported *clock_id* value.

[ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

The *sem_clockwait*( ) and *sem_timedwait*( ) functions may fail if:

[EDEADLK] A deadlock condition was detected.

[EINTR] A signal interrupted the function.

[EINVAL] The *sem* argument does not refer to a valid semaphore.

EXAMPLES

The program shown below operates on an unnamed semaphore. The program expects two command-line arguments. The first argument specifies a seconds value that is used to set an alarm timer to generate a SIGALRM signal. This handler performs a *sem_post*( ) to increment the semaphore that is being waited on in *main*( ) using *sem_clockwait*( ). The second command-line argument specifies the length of the timeout, in seconds, for *sem_clockwait*( ).

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <time.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>

sem_t sem;

static void
handler(int sig)
{
    int sav_errno = errno;
    static const char info_msg[] = "sem_post() from handler\n";
    write(STDOUT_FILENO, info_msg, sizeof info_msg - 1);
    if (sem_post(&sem) == -1) {
        static const char err_msg[] = "sem_post() failed\n";
        write(STDERR_FILENO, err_msg, sizeof err_msg - 1);
        _exit(EXIT_FAILURE);
    }
    errno = sav_errno;
}

int
main(int argc, char *argv[])
{
    struct sigaction sa;
    struct timespec ts;
    int s;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <alarm-secs> <wait-secs>\n",
            argv[0]);
        exit(EXIT_FAILURE);
    }

    if (sem_init(&sem, 0, 0) == -1) {
        perror("sem_init");
        exit(EXIT_FAILURE);
    }

    /* Establish SIGALRM handler; set alarm timer using argv[1] */

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGALRM, &sa, NULL) == -1) {
            perror("sigaction");
            exit(EXIT_FAILURE);
        }

    alarm(atoi(argv[1]));

    /* Calculate relative interval as current time plus
       number of seconds given argv[2] */

    if (clock_gettime(CLOCK_MONOTONIC, &ts) == -1) {
        perror("clock_gettime");
        exit(EXIT_FAILURE);
    }
    ts.tv_sec += atoi(argv[2]);

    printf("main() about to call sem_clockwait()\n");
    while ((s = sem_clockwait(&sem, CLOCK_MONOTONIC, &ts)) == -1 &&
          errno == EINTR)
        continue;       /* Restart if interrupted by handler */

    /* Check what happened */

    if (s == -1) {
        if (errno == ETIMEDOUT)
            printf("sem_clockwait() timed out\n");
        else
            perror("sem_clockwait");
    } else
        printf("sem_clockwait() succeeded\n");

    exit((s == 0) ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in XBD Section 3.260 (on page 66).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sem_post( ), *sem_trywait*( ), *semctl*( ), *semget*( ), *semop*( ), *time*( )

XBD Section 3.260 (on page 66), **<semaphore.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/120 is applied, updating the ERRORS section so that the [EINVAL] error becomes optional.

Issue 7

The *sem_timedwait*( ) function is moved from the Semaphores option to the Base.

Functionality relating to the Timers option is moved to the Base.

An example is added.

NAME

sig2str, str2sig — translate between signal names and numbers

SYNOPSIS

CX

```
#include <signal.h>

int sig2str(int signum, char *str);
int str2sig(const char *restrict str, int *restrict pnum);
```

DESCRIPTION

The *sig2str*( ) function shall translate the signal number specified by *signum* to a signal name and shall store this string in the location specified by *str*. The application shall ensure that *str* points to a location that can store the string including the terminating null byte. The symbolic constant SIG2STR_MAX defined in **<signal.h>** gives the maximum number of bytes required.

If *signum* is equal to 0, the behavior is unspecified.

If *signum* is equal to one of the symbolic constants listed in the table of signal numbers in **<signal.h>**, the stored signal name shall be the name of the symbolic constant without the **SIG** prefix.

If *signum* is equal to SIGRTMIN or SIGRTMAX, the stored string shall be `"RTMIN"` or `"RTMAX"`, respectively.

If *signum* is between SIGRTMIN+1 and (SIGRTMIN+SIGRTMAX)/2 inclusive, the stored string shall be of the form `"RTMIN+`*n*`"`, where *n* is the shortest decimal representation of the value of *signum*−SIGRTMIN.

If *signum* is between (SIGRTMIN+SIGRTMAX)/2 + 1 and SIGRTMAX−1 inclusive, the stored string shall be either of the form `"RTMIN+`*n*`"` or of the form `"RTMAX−`*m*`"`, where *n* is the shortest decimal representation of the value of *signum*−SIGRTMIN and *m* is the shortest decimal representation of the value of SIGRTMAX−*signum*.

If *signum* is a valid, supported signal number, is either less than SIGRTMIN or greater than SIGRTMAX, and is not equal to one of the symbolic constants listed in the table of signal numbers in **<signal.h>**, the stored string shall uniquely identify the signal number *signum* in an unspecified manner.

The *str2sig*( ) function shall translate the signal name in the string pointed to by *str* to a signal number and shall store this value in the location specified by *pnum*.

If *str* points to a string containing the name of one of the symbolic constants listed in the table of signal numbers in **<signal.h>**, without the **SIG** prefix, the stored signal number shall be equal to the value of the symbolic constant.

If *str* points to the string `"RTMIN"` or `"RTMAX"`, the stored value shall be equal to SIGRTMIN or SIGRTMAX, respectively.

If *str* points to a string of the form `"RTMIN+`*n*`"`, where *n* is a decimal representation of a number between 1 and SIGRTMAX−SIGRTMIN−1 inclusive, the stored value shall be equal to SIGRTMIN+n.

If *str* points to a string of the form `"RTMAX−`*n*`"`, where *n* is a decimal representation of a number between 1 and SIGRTMAX−SIGRTMIN−1 inclusive, the stored value shall be equal to SIGRTMAX−*n*.

If *str* points to a string containing a decimal representation of a valid, supported signal number, the value stored in the location pointed to by *pnum* shall be equal to that number.

If *str* points to a string containing a decimal representation of the value 0 and the string was not returned by a previous successful call to *sig2str*( ) with a *signum* argument of 0, the behavior is unspecified.

If *str* points to a string returned by a previous successful call to *sig2str*(*signum*,*str*), the value stored in the location pointed to by *pnum* shall be equal to *signum*.

If *str* points to a string that does not meet any of the above criteria, *str2sig*( ) shall store a value in the location pointed to by *pnum* if and only if it recognizes the string as an additional implementation-dependent form of signal name.

RETURN VALUE

If *signum* is a valid, supported signal number (that is, one for which *kill*( ) does not return −1 with *errno* set to [EINVAL]), the *sig2str*( ) function shall return 0; otherwise, if *signum* is not equal to 0, it shall return −1.

If *str2sig*( ) stores a value in the location pointed to by *pnum*, it shall return 0; otherwise, it shall return −1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Historical versions of these functions translated a *signum* value 0 to `"EXIT"` (and vice versa), so that they could be used by the shell for the *trap* utility. When adding the functions to this standard, the standard developers felt that they should be aimed at more general-purpose use, and consequently requiring this behavior did not seem appropriate and so the behavior in this case has been made unspecified.

FUTURE DIRECTIONS

None.

SEE ALSO

kill( ), *sigaction*( ), *strsignal*( )

XBD **<signal.h>**

CHANGE HISTORY

First released in Issue 8.

NAME

str2sig — translate between signal names and numbers

SYNOPSIS

CX

```
#include <signal.h>

int str2sig(const char *restrict str, int *restrict pnum);
```

DESCRIPTION

Refer to *sig2str*( ).

NAME

strlcat, strlcpy — size-bounded string concatenation and copying

SYNOPSIS

CX

```
#include <string.h>

size_t strlcat(char *restrict dst, const char *restrict src,
    size_t dstsize);
size_t strlcpy(char *restrict dst, const char *restrict src,
    size_t dstsize);
```

DESCRIPTION

The *strlcpy*( ) and *strlcat*( ) functions copy and concatenate strings, stopping when either a NUL terminator in the source string is encountered or the specified full size of the destination buffer is reached. They NUL terminate the result if there is room. The application should ensure that room for the NUL terminator is included in *dstsize*.

The *strlcpy*( ) function shall copy not more than *dstsize* − 1 bytes from the string pointed to by *src* to the array pointed to by *dst*; a NUL byte in *src* and bytes that follow it shall not be copied. A terminating NUL byte shall be appended to the result, unless *dstsize* is 0. If copying takes place between objects that overlap, the behavior is undefined.

The *strlcat*( ) function shall append not more than *dstsize* − *strlen*(*dst*) − 1 bytes from the string pointed to by *src* to the end of the string pointed to by *dst*; a NUL byte in *src* and bytes that follow it shall not be appended. The initial byte of *src* shall overwrite the NUL byte at the end of *dst*. A terminating NUL byte shall be appended to the result, unless its location would be at or beyond *dst* + *dstsize*. If copying takes place between objects that overlap, the behavior is undefined.

The *strlcpy*( ) and *strlcat*( ) functions shall not change the setting of *errno* on valid input.

RETURN VALUE

Upon successful completion, the *strlcpy*( ) function shall return the length of the string pointed to by *src*; that is, the number of bytes in the string, not including the terminating NUL byte.

Upon successful completion, the *strlcat*( ) function shall return the initial length of the string pointed to by *dst* plus the length of the string pointed to by *src*.

No return values are reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

The following example detects truncation while combining a path prefix (including trailing <slash>) and a filename to produce a portable pathname:

```
char *prefix, *filenam, pathnam[_POSIX_PATH_MAX];

if (strlcpy(pathnam, prefix, sizeof pathnam) >= sizeof pathnam ||
    strlcat(pathnam, filenam, sizeof pathnam) >= sizeof pathnam)
{
    // truncation occurred
    ...
}
```

This code ensures there is room for the NUL terminator by:

- Calling *strlcpy*( ) with a non-zero *dstsize* argument.
- Only calling *strlcat*( ) if the return value of *strlcpy*( ) indicated that truncation did not occur.

APPLICATION USAGE

The return value of the *strlcpy*( ) and *strlcat*( ) functions follows the same convention as *snprintf*( ); that is, they return the total length of the string they tried to create. If the return value is greater than or equal to *dstsize*, the output string has been truncated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fprintf( ), *strlen*( ), *strncat*( ), *strncpy*( ), *wcslcat*( )

XBD **<string.h>**

CHANGE HISTORY

First released in Issue 8.

NAME

wcslcat, wcslcpy — size-bounded wide string concatenation and copying

SYNOPSIS

CX

```
#include <wchar.h>

size_t wcslcat(wchar_t *restrict dst, const wchar_t *restrict src,
    size_t dstsize);
size_t wcslcpy(wchar_t *restrict dst, const wchar_t *restrict src,
    size_t dstsize);
```

DESCRIPTION

The *wcslcpy*( ) and *wcslcat*( ) functions copy and concatenate wide strings, stopping when either a terminating null wide-character code in the source wide string is encountered or the specified full size (in wide-character codes) of the destination buffer is reached. They null terminate the result if there is room. The application should ensure that room for the terminating null wide-character code is included in *dstsize*.

The *wcslcpy*( ) function shall copy not more than *dstsize* − 1 wide-character codes from the wide string pointed to by *src* to the array pointed to by *dst*; a terminating null wide-character code in *src* and wide-character codes that follow it shall not be copied. A terminating null wide-character code shall be appended to the result, unless *dstsize* is 0. If copying takes place between objects that overlap, the behavior is undefined.

The *wcslcat*( ) function shall append not more than *dstsize* − *wcslen*(*dst*) − 1 wide-character codes from the wide string pointed to by *src* to the end of the wide string pointed to by *dst*; a terminating null wide-character code in *src* and wide-character codes that follow it shall not be appended. The initial wide-character code of *src* shall overwrite the null wide-character code at the end of *dst*. A terminating null wide-character code shall be appended to the result, unless its location would be at or beyond *dst* + *dstsize*. If copying takes place between objects that overlap, the behavior is undefined.

The *wcslcpy*( ) and *wcslcat*( ) functions shall not change the setting of *errno* on valid input.

RETURN VALUE

Upon successful completion, the *wcslcpy*( ) function shall return the length of the wide string pointed to by *src*; that is, the number of wide-character codes in the wide string, not including the terminating null wide-character code.

Upon successful completion, the *wcslcat*( ) function shall return the initial length of the wide string pointed to by *dst* plus the length of the wide string pointed to by *src*.

No return values are reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The return value of the *wcslcpy*( ) and *wcslcat*( ) functions follows the same convention as *snprintf*( ); that is, they return the total length (in wide-character codes) of the wide string they tried to create. If the return value is greater than or equal to *dstsize*, the output wide string has been truncated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fprintf( ), *strlcat*( ), *wcslen*( ), *wcsncat*( ), *wcsncpy*( )

XBD **<wchar.h>**

CHANGE HISTORY

First released in Issue 8.

POSIX.1b is a software, source-level standard and most of the benefits of the alternate representation are enjoyed by hardware implementations of clocks and algorithms. It was felt that mandating this format for POSIX.1b clocks and timers would unnecessarily burden the application developer with writing, possibly non-portable, multiple precision arithmetic packages to perform conversion between binary fractions and integral units such as nanoseconds, milliseconds, and so on.

Rationale for the Monotonic Clock

For those applications that use time services to achieve realtime behavior, changing the value of the clock on which these services rely may cause erroneous timing behavior. For these applications, it is necessary to have a monotonic clock which cannot run backwards, and which has a maximum clock jump that is required to be documented by the implementation. Additionally, it is desirable (but not required by POSIX.1-202x) that the monotonic clock increases its value uniformly. This clock should not be affected by changes to the system time; for example, to synchronize the clock with an external source or to account for leap seconds. Such changes would cause errors in the measurement of time intervals for those time services that use the absolute value of the clock.

One could argue that by defining the behavior of time services when the value of a clock is changed, deterministic realtime behavior can be achieved. For example, one could specify that relative time services should be unaffected by changes in the value of a clock. However, there are time services that are based upon an absolute time, but that are essentially intended as relative time services. For example, *pthread_cond_timedwait*( ) uses an absolute time to allow it to wake up after the required interval despite spurious wakeups. Although sometimes the *pthread_cond_timedwait*( ) timeouts are absolute in nature, there are many occasions in which they are relative, and their absolute value is determined from the current time plus a relative time interval. In this latter case, if the clock changes while the thread is waiting, the wait interval will not be the expected length. If a *pthread_cond_timedwait*( ) function were created that would take a relative time, it would not solve the problem because to retain the intended ``deadline'' a thread would need to compensate for latency due to the spurious wakeup, and preemption between wakeup and the next wait.

The solution is to create a new monotonic clock, whose value does not change except for the regular ticking of the clock, and use this clock for implementing the various relative timeouts that appear in the different POSIX interfaces, as well as allow *pthread_cond_timedwait*( ) to choose this new clock for its timeout. A new *clock_nanosleep*( ) function is created to allow an application to take advantage of this newly defined clock. Notice that the monotonic clock may be implemented using the same hardware clock as the system clock.

Relative timeouts for *sigtimedwait*( ) and *aio_suspend*( ) have been redefined to use the monotonic clock, if present. The *alarm*( ) function has not been redefined, because the same effect but with better resolution can be achieved by creating a timer (for which the appropriate clock may be chosen).

The *pthread_cond_timedwait*( ) function has been treated in a different way, compared to other functions with absolute timeouts, because it is used to wait for an event, and thus it may have a deadline, while the other timeouts are generally used as an error recovery mechanism, and for them the use of the monotonic clock is not so important. Since the desired timeout for the *pthread_cond_timedwait*( ) function may either be a relative interval or an absolute time of day deadline, a new initialization attribute has been created for condition variables to specify the clock that is used for measuring the timeout in a call to *pthread_cond_timedwait*( ). In this way, if a relative timeout is desired, the monotonic clock will be used; if an absolute deadline is required instead, the CLOCK_REALTIME or another appropriate clock may be used. For condition variables, this capability is also available by passing CLOCK_MONOTONIC to the *pthread_cond_clockwait*( ) function. Similarly, CLOCK_MONOTONIC can be specified when calling *pthread_mutex_clocklock*( ), *pthread_rwlock_clockrdlock*( ), *pthread_rwlock_clockwrlock*( ), and *sem_clockwait*( ).

It was later found necessary to add variants of almost all interfaces that accept absolute timeouts that allow the clock to be specified. This is because, despite the claim in the previous paragraph, it is not possible to safely use a CLOCK_REALTIME absolute timeout even to prevent errors when the system clock is warped by a potentially large amount. A ``safety timeout'' of a minute on a call to *pthread_mutex_timedlock*( ) could actually mean that the call would return ETIMEDOUT early without acquiring the lock if the system clock is warped forwards immediately prior to or during the call. On the other hand, a short timeout could end up being arbitrarily long if the system clock is warped backwards immediately prior to or during the call. These problems are solved by the new *clockwait* and *clocklock* variants of the existing *timedwait* and *timedlock* functions. These variants accept an extra **clockid_t** parameter to indicate the clock to be used for the wait. The clock ID is passed rather than using attributes as previously for *pthread_cond_timedwait*( ) in order to allow the ISO/IEC 14882: 2011 standard (C++11) and later to be implemented correctly. C++ requires that the clock to use for the wait is not known until the time of the wait call, so it cannot be supplied during creation. The new functions are *pthread_cond_clockwait*( ), *pthread_mutex_clocklock*( ), *pthread_mutex_clockrdlock*( ), *pthread_mutex_clockwrlock*( ), and *sem_clockwait*( ). It is expected that *mq_clockreceive( )* and *mq_clocksend( )* functions will be added in a future version of this standard.

The *nanosleep*( ) function has not been modified with the introduction of the monotonic clock. Instead, a new *clock_nanosleep*( ) function has been created, in which the desired clock may be specified in the function call.

- History of Resolution Issues

  Due to the shift from relative to absolute timeouts in IEEE Std 1003.1d-1999, the amendments to the *sem_timedwait*( ), *pthread_mutex_timedlock*( ), *mq_timedreceive*( ), and *mq_timedsend*( ) functions of that standard have been removed. Those amendments specified that CLOCK_MONOTONIC would be used for the (relative) timeouts if the Monotonic Clock option was supported.

  Having these functions continue to be tied solely to CLOCK_MONOTONIC would not work. Since the absolute value of a time value obtained from CLOCK_MONOTONIC is unspecified, under the absolute timeouts interface, applications would behave differently depending on whether the Monotonic Clock option was supported or not (because the absolute value of the clock would have different meanings in either case).

  Two options were considered:

  1. Leave the current behavior unchanged, which specifies the CLOCK_REALTIME clock for these (absolute) timeouts, to allow portability of applications between implementations supporting or not the Monotonic Clock option.

  2. Modify these functions in the way that *pthread_cond_timedwait*( ) was modified to allow a choice of clock, so that an application could use CLOCK_REALTIME when it is trying to achieve an absolute timeout and CLOCK_MONOTONIC when it is trying to achieve a relative timeout.

  It was decided that the features of CLOCK_MONOTONIC are not as critical to these functions as they are to *pthread_cond_timedwait*( ). The *pthread_cond_timedwait*( ) function is given a relative timeout; the timeout may represent a deadline for an event. When these functions are given relative timeouts, the timeouts are typically for error recovery purposes and need not be so precise.

  Therefore, it was decided that these functions should be tied to CLOCK_REALTIME and

Supported Threads Functions

On POSIX-conforming systems, the following symbolic constants are always conforming:

_POSIX_READER_WRITER_LOCKS
_POSIX_THREADS

Therefore, the following threads functions are always supported:

pthread_atfork( )
pthread_attr_destroy( )
pthread_attr_getdetachstate( )
pthread_attr_getguardsize( )
pthread_attr_getschedparam( )
pthread_attr_init( )
pthread_attr_setdetachstate( )
pthread_attr_setguardsize( )
pthread_attr_setschedparam( )
pthread_cancel( )
pthread_cleanup_pop( )
pthread_cleanup_push( )
pthread_cond_broadcast( )
pthread_cond_clockwait( )
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_condattr_destroy( )
pthread_condattr_getpshared( )
pthread_condattr_init( )
pthread_condattr_setpshared( )
pthread_create( )
pthread_detach( )
pthread_equal( )
pthread_exit( )
pthread_getspecific( )
pthread_join( )
pthread_key_create( )
pthread_key_delete( )
pthread_kill( )
pthread_mutex_destroy( )
pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
pthread_mutexattr_destroy( )
pthread_mutexattr_getpshared( )
pthread_mutexattr_gettype( )
pthread_mutexattr_init( )
pthread_mutexattr_setpshared( )
pthread_mutexattr_settype( )
pthread_once( )
pthread_rwlock_destroy( )
pthread_rwlock_init( )
pthread_rwlock_rdlock( )
pthread_rwlock_tryrdlock( )
pthread_rwlock_trywrlock( )
pthread_rwlock_unlock( )
pthread_rwlock_wrlock( )
pthread_rwlockattr_destroy( )
pthread_rwlockattr_getpshared( )
pthread_rwlockattr_init( )
pthread_rwlockattr_setpshared( )
pthread_self( )
pthread_setcancelstate( )
pthread_setcanceltype( )
pthread_setspecific( )
pthread_sigmask( )
pthread_testcancel( )
sigwait( )

pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )

to take account of the new mutex attribute type and to specify behavior which was declared as undefined in POSIX.1c. How a calling thread acquires or releases a mutex now depends upon the mutex *type* attribute.

The *type* attribute can have the following values:

PTHREAD_MUTEX_NORMAL
Basic mutex with no specific error checking built in. Does not report a deadlock error.

PTHREAD_MUTEX_RECURSIVE
Allows any thread to recursively lock a mutex. The mutex must be unlocked an equal number of times to release the mutex.

PTHREAD_MUTEX_ERRORCHECK
Detects and reports simple usage errors; that is, an attempt to unlock a mutex that is not locked by the calling thread or that is not locked at all, or an attempt to relock a mutex the thread already owns.

PTHREAD_MUTEX_DEFAULT
The default mutex type. May be mapped to any of the above mutex types or may be an implementation-defined type.

Normal mutexes do not detect deadlock conditions; for example, a thread will hang if it tries to relock a normal mutex that it already owns. Attempting to unlock a mutex locked by another thread, or unlocking an unlocked mutex, results in undefined behavior. Normal mutexes will usually be the fastest type of mutex available on a platform but provide the least error checking.

Recursive mutexes are useful for converting old code where it is difficult to establish clear boundaries of synchronization. A thread can relock a recursive mutex without first unlocking it. The relocking deadlock which can occur with normal mutexes cannot occur with this type of mutex. However, multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. Furthermore, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock a recursive mutex which another thread has locked returns with an error. A thread attempting to unlock a recursive mutex that is not locked returns with an error. Never use a recursive mutex with condition variables because the implicit unlock performed by *pthread_cond_clockwait*( ), *pthread_cond_timedwait*( ), or *pthread_cond_wait*( ) will not actually release the mutex if it had been locked multiple times. -

Errorcheck mutexes provide error checking and are useful primarily as a debugging aid. A thread attempting to relock an errorcheck mutex without first unlocking it returns with an error. Again, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock an errorcheck mutex which another thread has locked returns with an error. A thread attempting to unlock an errorcheck mutex that is not locked also returns with an error. It should be noted that errorcheck mutexes will almost always be much slower than normal mutexes due to the extra state checks performed.

The default mutex type provides implementation-defined error checking. The default mutex may be mapped to one of the other defined types or may be something entirely different. This enables each vendor to provide the mutex semantics which the vendor feels will be most useful to their target users. Most vendors will probably choose to make normal mutexes the default so as to give applications the benefit of the fastest type of

Unsatisfied Requirements

Detailed control of common device classes, specifically magnetic tape, is not provided.

D.2.5 Bounded (Realtime) Response

The realtime signal functions *sigqueue*( ), *sigtimedwait*( ), and *sigwaitinfo*( ) provide queued signals and the prioritization of the handling of signals.

The SCHED_FIFO, SCHED_SPORADIC, and SCHED_RR scheduling policies provide control over processor allocation.

The semaphore functions *sem_clockwait*( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_unlink*( ), and *sem_wait*( ) provide high-performance synchronization.

The memory management functions provide memory locking for control of memory allocation, file mapping for high performance, and shared memory for high-performance interprocess communication. The Message Passing option provides for interprocess communication without being dependent on shared memory.

The timers functions *clock_getres*( ), *clock_gettime*( ), *clock_settime*( ), *nanosleep*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) provide functionality to manipulate clocks and timers and include a high resolution function called *nanosleep*( ) with a finer resolution than the *sleep*( ) function.

The timeout functions — *pthread_mutex_clocklock*( ), *pthread_mutex_timedlock*( ), *pthread_rwlock_clockrdlock*( ), *pthread_rwlock_clockwrlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *sem_clockwait*( ), and *sem_timedwait*( ) — the Typed Memory Objects option and the Monotonic Clock option provide further facilities for applications to use to obtain predictable bounded response.

D.2.6 Operating System-Dependent Profile

POSIX.1-202x makes no distinction between text and binary files. The values of EXIT_SUCCESS and EXIT_FAILURE are further defined.

Unsatisfied Requirements

None known, but the ISO C standard may contain some additional options that could be specified.

D.2.7 I/O Interaction

POSIX.1-202x defines how each of the ISO C standard *stdio* functions interact with the POSIX.1 operations, typically specifying the behavior in terms of POSIX.1 operations.

rintl(), round(), roundf(), roundl(), scalbln(), scalblnf(), scalblnl(), scalbn(), scalbnf(), scalbnl(), signbit(), sin(), sinf(), sinh(), sinhf(), sinhl(), sinl(), sqrt(), sqrtf(), sqrtl(), tan(), tanf(), tanh(), tanhf(), tanhl(), tanl(), tgamma(), tgammaf(), tgammal(), trunc(), truncf(), truncl()

POSIX_C_LANG_SUPPORT: General ISO C Library

abs(), asctime(), atof(), atoi(), atol(), atoll(), bsearch(), calloc(), ctime(), difftime(), div(), feclearexcept(), fegetenv(), fegetexceptflag(), fegetround(), feholdexcept(), feraiseexcept(), fesetenv(), fesetexceptflag(), fesetround(), fetestexcept(), feupdateenv(), free(), gmtime(), imaxabs(), imaxdiv(), isalnum(), isalpha(), isblank(), iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit(), labs(), ldiv(), llabs(), lldiv(), localeconv(), localtime(), malloc(), memchr(), memcmp(), memcpy(), memmove(), memset(), mktime(), qsort(), rand(), realloc(), setlocale(), snprintf(), sprintf(), srand(), sscanf(), strcat(), strchr(), strcmp(), strcoll(), strcpy(), strcspn(), strerror(), strftime(), strlen(), strncat(), strncmp(), strncpy(), strpbrk(), strrchr(), strspn(), strstr(), strtod(), strtof(), strtoimax(), strtok(), strtol(), strtold(), strtoll(), strtoul(), strtoull(), strtoumax(), strxfrm(), time(), tolower(), toupper(), tzname, tzset(), va_arg(), va_copy(), va_end(), va_start(), vsnprintf(), vsprintf(), vsscanf()

POSIX_C_LANG_SUPPORT_R: Thread-Safe General ISO C Library

asctime_r(), ctime_r(), gmtime_r(), localtime_r(), qsort_r(), strerror_r(), strtok_r() +

POSIX_C_LANG_WIDE_CHAR: Wide-Character ISO C Library

btowc(), iswalnum(), iswalpha(), iswblank(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), mblen(), mbrlen(), mbrtowc(), mbsinit(), mbsrtowcs(), mbstowcs(), mbtowc(), swprintf(), swscanf(), towctrans(), towlower(), towupper(), vswprintf(), vswscanf(), wcrtomb(), wcscat(), wcschr(), wcscmp(), wcscoll(), wcscpy(), wcscspn(), wcsftime(), wcslen(), wcsncat(), wcsncmp(), wcsncpy(), wcspbrk(), wcsrchr(), wcsrtombs(), wcsspn(), wcsstr(), wcstod(), wcstof(), wcstoimax(), wcstok(), wcstol(), wcstold(), wcstoll(), wcstombs(), wcstoul(), wcstoull(), wcstoumax(), wcsxfrm(), wctob(), wctomb(), wctrans(), wctype(), wmemchr(), wmemcmp(), wmemcpy(), wmemmove(), wmemset()

POSIX_C_LANG_WIDE_CHAR_EXT: Extended Wide-Character ISO C Library

mbsnrtowcs(), wcpcpy(), wcpncpy(), wcscasecmp(), wcsdup(), wcslcat(), wcslcpy(), +
wcsncasecmp(), wcsnlen(), wcsnrtombs()

POSIX_C_LIB_EXT: General C Library Extension

fnmatch(), getentropy(), getopt(), getsubopt(), memmem(), optarg, opterr, optind, optopt, +
reallocarray(), stpcpy(), stpncpy(), strcasecmp(), strdup(), strfmon(), strlcat(), strlcpy(), +
strncasecmp(), strndup(), strnlen()

POSIX_CLOCK_SELECTION: Clock Selection

clock_nanosleep(), pthread_condattr_getclock(), pthread_condattr_setclock()

POSIX_DEVICE_IO: Device Input and Output

FD_CLR(), FD_ISSET(), FD_SET(), FD_ZERO(), clearerr(), close(), fclose(), fdopen(), feof(),
ferror(), fflush(), fgetc(), fgets(), fileno(), fopen(), fprintf(), fputc(), fputs(), fread(), freopen(),
fscanf(), fwrite(), getc(), getchar(), open(), perror(), poll(), ppoll(), printf(), pread(), pselect(), +
putc(), putchar(), puts(), pwrite(), read(), scanf(), select(), setbuf(), setvbuf(), stderr, stdin,
stdout, ungetc(), vfprintf(), vfscanf(), vprintf(), vscanf(), write()

POSIX_DEVICE_IO_EXT: Extended Device Input and Output

dprintf(), fmemopen(), open_memstream(), vdprintf()

POSIX_DEVICE_SPECIFIC: General Terminal

cfgetispeed(), cfgetospeed(), cfsetispeed(), cfsetospeed(), ctermid(), isatty(), tcdrain(), tcflow(), tcflush(), tcgetattr(), tcgetwinsize(), tcsendbreak(), tcsetattr(), tcsetwinsize(), ttyname()

POSIX_DEVICE_SPECIFIC_R: Thread-Safe General Terminal

ttyname_r( )

POSIX_DYNAMIC_LINKING: Dynamic Linking

dladdr( ), *dlclose*( ), *dlerror*( ), *dlopen*( ), *dlsym*( )

POSIX_FD_MGMT: File Descriptor Management

dup( ), *dup2*( ), *dup3*( ), *fcntl*( ), *fgetpos*( ), *fseek*( ), *fseeko*( ), *fsetpos*( ), *ftell*( ), *ftello*( ), *ftruncate*( ), *lseek*( ), *rewind*( )

POSIX_FIFO: FIFO

mkfifo( )

POSIX_FIFO_FD: FIFO File Descriptor Routines

mkfifoat( ), *mknodat*( )

POSIX_FILE_ATTRIBUTES: File Attributes

chmod( ), *chown*( ), *fchmod*( ), *fchown*( ), *umask*( )

POSIX_FILE_ATTRIBUTES_FD: File Attributes File Descriptor Routines

fchmodat( ), *fchownat*( )

POSIX_FILE_LOCKING: Thread-Safe Stdio Locking

flockfile( ), *ftrylockfile*( ), *funlockfile*( ), *getc_unlocked*( ), *getchar_unlocked*( ), *putc_unlocked*( ), *putchar_unlocked*( )

POSIX_FILE_SYSTEM: File System

access( ), *chdir*( ), *closedir*( ), *creat*( ), *fchdir*( ), *fpathconf*( ), *fstat*( ), *fstatvfs*( ), *getcwd*( ), *link*( ), *mkdir*( ), *mkostemp*( ), *mkstemp*( ), *opendir*( ), *pathconf*( ), *posix_getdents*( ), *readdir*( ), *remove*( ), *rename*( ), *rewinddir*( ), *rmdir*( ), *stat*( ), *statvfs*( ), *tmpfile*( ), *tmpnam*( ), *truncate*( ), *unlink*( )

POSIX_FILE_SYSTEM_EXT: File System Extensions

alphasort( ), *dirfd*( ), *getdelim*( ), *getline*( ), *mkdtemp*( ), *scandir*( )

POSIX_FILE_SYSTEM_FD: File System File Descriptor Routines

faccessat( ), *fdopendir*( ), *fstatat*( ), *linkat*( ), *mkdirat*( ), *openat*( ), *renameat*( ), *unlinkat*( ), *utimensat*( )

POSIX_FILE_SYSTEM_GLOB: File System Glob Expansion

glob( ), *globfree*( )

POSIX_FILE_SYSTEM_R: Thread-Safe File System

readdir_r( )

POSIX_I18N: Internationalization

catclose( ), *catgets*( ), *catopen*( ), *iconv*( ), *iconv_close*( ), *iconv_open*( ), *nl_langinfo*( )

POSIX_JOB_CONTROL: Job Control

setpgid( ), *tcgetpgrp*( ), *tcsetpgrp*( ), *tcgetsid*( )

POSIX_MAPPED_FILES: Memory Mapped Files

mmap( ), *munmap*( )

POSIX_MEMORY_PROTECTION: Memory Protection

mprotect( )

POSIX_MULTI_CONCURRENT_LOCALES: Multiple Concurrent Locales

duplocale( ), *freelocale*( ), *getlocalename_l*( ), *isalnum_l*( ), *isalpha_l*( ), *isblank_l*( ), *iscntrl_l*( ), *isdigit_l*( ), *isgraph_l*( ), *islower_l*( ), *isprint_l*( ), *ispunct_l*( ), *isspace_l*( ), *isupper_l*( ), *iswalnum_l*( ), *iswalpha_l*( ), *iswblank_l*( ), *iswcntrl_l*( ), *iswctype_l*( ), *iswdigit_l*( ), *iswgraph_l*( ), *iswlower_l*( ), *iswprint_l*( ), *iswpunct_l*( ), *iswspace_l*( ), *iswupper_l*( ), *iswxdigit_l*( ), *isxdigit_l*( ), *newlocale*( ), *strcasecmp_l*( ), *strcoll_l*( ), *strfmon_l*( ), *strncasecmp_l*( ), *strxfrm_l*( ), *tolower_l*( ), *toupper_l*( ), *towctrans_l*( ), *towlower*( ), *towupper*( ), *uselocale*( ), *wcscasecmp_l*( ), *wcscoll_l*( ), *wcsncasecmp_l*( ), *wcsxfrm_l*( ), *wctrans_l*( ), *wctype_l*( )

POSIX_MULTI_PROCESS: Multiple Processes
_Exit( ), *_exit*( ), *assert*( ), *atexit*( ), *clock*( ), *execl*( ), *execle*( ), *execlp*( ), *execv*( ), *execve*( ), *execvp*( ), *exit*( ), *fork*( ), *getpgrp*( ), *getpgid*( ), *getpid*( ), *getppid*( ), *getsid*( ), *setsid*( ), *sleep*( ), *times*( ), *wait*( ), *waitid*( ), *waitpid*( )

POSIX_MULTI_PROCESS_FD: Multiple Processes File Descriptor Routines
fexecve( )

POSIX_NETWORKING: Networking
accept( ), *accept4*( ), *bind*( ), *connect*( ), *endhostent*( ), *endnetent*( ), *endprotoent*( ), *endservent*( ), *freeaddrinfo*( ), *gai_strerror*( ), *getaddrinfo*( ), *gethostent*( ), *gethostname*( ), *getnameinfo*( ), *getnetbyaddr*( ), *getnetbyname*( ), *getnetent*( ), *getpeername*( ), *getprotobyname*( ), *getprotobynumber*( ), *getprotoent*( ), *getservbyname*( ), *getservbyport*( ), *getservent*( ), *getsockname*( ), *getsockopt*( ), *htonl*( ), *htons*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *if_nametoindex*( ), *inet_addr*( ), *inet_ntoa*( ), *inet_ntop*( ), *inet_pton*( ), *listen*( ), *ntohl*( ), *ntohs*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *sendto*( ), *sethostent*( ), *setnetent*( ), *setprotoent*( ), *setservent*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), *sockatmark*( ), *socketpair*( )

POSIX_PIPE: Pipe
pipe( ), *pipe2*( )

POSIX_ROBUST_MUTEXES: Robust Mutexes
pthread_mutex_consistent( ), *pthread_mutexattr_getrobust*( ), *pthread_mutexattr_setrobust*( )

POSIX_REALTIME_SIGNALS: Realtime Signals
sigqueue( ), *sigtimedwait*( ), *sigwaitinfo*( )

POSIX_REGEXP: Regular Expressions
regcomp( ), *regerror*( ), *regexec*( ), *regfree*( )

POSIX_RW_LOCKS: Reader Writer Locks
pthread_rwlock_clockrdlock( ), *pthread_rwlock_clockwrlock*( ), *pthread_rwlock_destroy*( ), + *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), *pthread_rwlockattr_destroy*( ), *pthread_rwlockattr_init*( ), *pthread_rwlockattr_getpshared*( ), *pthread_rwlockattr_setpshared*( )

POSIX_SEMAPHORES: Semaphores
sem_clockwait( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), + *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_unlink*( ), *sem_wait*( )

POSIX_SHELL_FUNC: Shell and Utilities
pclose( ), *popen*( ), *system*( ), *wordexp*( ), *wordfree*( )

POSIX_SIGNAL_JUMP: Signal Jump Functions
siglongjmp( ), *sigsetjmp*( )

POSIX_SIGNALS: Signals
abort( ), *alarm*( ), *kill*( ), *pause*( ), *raise*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *signal*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigwait*( )

POSIX_SIGNALS_EXT: Extended Signals
psignal( ), *psiginfo*( ), *sig2str*( ), *str2sig*( ), *strsignal*( ) +

POSIX_SINGLE_PROCESS: Single Process
confstr( ), *environ*, *errno*, *getenv*( ), *setenv*( ), *sysconf*( ), *uname*( ), *unsetenv*( )

POSIX_SPIN_LOCKS: Spin Locks
pthread_spin_destroy( ), *pthread_spin_init*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), *pthread_spin_unlock*( )

POSIX_SYMBOLIC_LINKS: Symbolic Links
lchown( ),[11] *lstat*( ), *readlink*( ), *symlink*( )

POSIX_SYMBOLIC_LINKS_FD: Symbolic Links File Descriptor Routines
readlinkat( ), *symlinkat*( )

POSIX_SYSTEM_DATABASE: System Database
getgrgid( ), *getgrnam*( ), *getpwnam*( ), *getpwuid*( )

POSIX_SYSTEM_DATABASE_R: Thread-Safe System Database
getgrgid_r( ), *getgrnam_r*( ), *getpwnam_r*( ), *getpwuid_r*( )

POSIX_THREADS_BASE: Base Threads
pthread_atfork( ), *pthread_attr_destroy*( ), *pthread_attr_getdetachstate*( ), *pthread_attr_getschedparam*( ), *pthread_attr_init*( ), *pthread_attr_setdetachstate*( ), *pthread_attr_setschedparam*( ), *pthread_cancel*( ), *pthread_cleanup_pop*( ), *pthread_cleanup_push*( ), *pthread_cond_broadcast*( ), *pthread_cond_clockwait*( ), *pthread_cond_destroy*( ), *pthread_cond_init*( ), *pthread_cond_signal*( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), *pthread_condattr_destroy*( ), *pthread_condattr_init*( ), *pthread_create*( ), *pthread_detach*( ), *pthread_equal*( ), *pthread_exit*( ), *pthread_getspecific*( ), *pthread_join*( ), *pthread_key_create*( ), *pthread_key_delete*( ), *pthread_kill*( ), *pthread_mutex_clocklock*( ), *pthread_mutex_destroy*( ), *pthread_mutex_init*( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), *pthread_mutex_trylock*( ), *pthread_mutex_unlock*( ), *pthread_mutexattr_destroy*( ), *pthread_mutexattr_init*( ), *pthread_once*( ), *pthread_self*( ), *pthread_setcancelstate*( ), *pthread_setcanceltype*( ), *pthread_setspecific*( ), *pthread_sigmask*( ), *pthread_testcancel*( )

POSIX_THREADS_EXT: Extended Threads
pthread_attr_getguardsize( ), *pthread_attr_setguardsize*( ), *pthread_mutexattr_gettype*( ), *pthread_mutexattr_settype*( )

POSIX_TIMERS: Timers
clock_getres( ), *clock_gettime*( ), *clock_settime*( ), *nanosleep*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), *timer_settime*( )

POSIX_USER_GROUPS: User and Group
getegid( ), *geteuid*( ), *getgid*( ), *getgroups*( ), *getlogin*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setuid*( )

POSIX_USER_GROUPS_R: Thread-Safe User and Group
getlogin_r( )

POSIX_WIDE_CHAR_DEVICE_IO: Device Input and Output
fgetwc( ), *fgetws*( ), *fputwc*( ), *fputws*( ), *fwide*( ), *fwprintf*( ), *fwscanf*( ), *getwc*( ), *getwchar*( ), *putwc*( ), *putwchar*( ), *ungetwc*( ), *vfwprintf*( ), *vfwscanf*( ), *vwprintf*( ), *vwscanf*( ), *wprintf*( ), *wscanf*( )

XSI_C_LANG_SUPPORT: XSI General C Library
a64l( ), *daylight*, *drand48*( ), *erand48*( ), *ffs*( ), *ffsl*( ), *ffsll*( ), *getdate*( ), *hcreate*( ), *hdestroy*( ), *hsearch*( ), *initstate*( ), *insque*( ), *jrand48*( ), *l64a*( ), *lcong48*( ), *lfind*( ), *lrand48*( ), *lsearch*( ), *memccpy*( ), *mrand48*( ), *nrand48*( ), *random*( ), *remque*( ), *seed48*( ), *setstate*( ), *signgam*,

[11] The *lchown*( ) function also depends on POSIX_FILE_ATTRIBUTES.