

The Open Group Standard

Additional APIs for the Base Specifications Issue 8, Part 1



Copyright © 2021, The Open Group

The Open Group hereby authorizes you to use this document for any purpose, PROVIDED THAT any copy of this document, or any part thereof, which you make shall retain all copyright and other proprietary notices contained herein.

This document may contain other proprietary notices and copyright information.

Nothing contained herein shall be construed as conferring by implication, estoppel, or otherwise any license or right under any patent or trademark of The Open Group or any third party. Except as expressly provided above, nothing contained herein shall be construed as conferring any license or right under any copyright of The Open Group.

Note that any product, process, or technology in this document may be the subject of other intellectual property rights reserved by The Open Group, and may not be licensed hereunder.

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:

ogspeccs@opengroup.org

Contents

1	Introduction	1
1.1	Scope	1
1.2	Relationship to Other Formal Standards	1
2	Application Program Interfaces	2
2.1	Change Bars	2
2.2	Reference Pages	2

Sanity Review

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 Kjørby
- Curtis Smith
- Dennis Wölfing

Sanity Review

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:

<i>dladdr()</i>	<i>qsort_r()</i>
<i>getentropy()</i>	<i>reallocarray()</i>
<i>getlocalename_l()</i>	<i>sem_clockwait()</i>
<i>memmem()</i>	<i>sig2str()</i>
<i>posix_getdents()</i>	<i>str2sig()</i>
<i>ppoll()</i>	<i>strlcat()</i>
<i>pthread_cond_clockwait()</i>	<i>strlcpy()</i>
<i>pthread_mutex_clocklock()</i>	<i>wslcat()</i>
<i>pthread_rwlock_clockrdlock()</i>	<i>wslcpy()</i>
<i>pthread_rwlock_clockwrlock()</i>	

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.

2782 4.13 Memory Synchronization

2783 Applications shall ensure that access to any memory location by more than one thread of control
 2784 (threads or processes) is restricted such that no thread of control can read or modify a memory
 2785 location while another thread of control may be modifying it. Such access is restricted using
 2786 functions that synchronize thread execution and also synchronize memory with respect to other
 2787 threads. The following functions synchronize memory with respect to other threads:

2788	<code>fork()</code>	<code>pthread_mutex_trylock()</code>	<code>pthread_rwlock_unlock()</code>
2789	<code>pthread_barrier_wait()</code>	<code>pthread_mutex_unlock()</code>	<code>pthread_rwlock_wrlock()</code>
2790	<code>pthread_cond_broadcast()</code>	<code>pthread_spin_lock()</code>	<code>sem_clockwait()</code>
2791	<code>pthread_cond_clockwait()</code>	<code>pthread_spin_trylock()</code>	<code>sem_post()</code>
2792	<code>pthread_cond_signal()</code>	<code>pthread_spin_unlock()</code>	<code>sem_timedwait()</code>
2793	<code>pthread_cond_timedwait()</code>	<code>pthread_rwlock_clockrdlock()</code>	<code>sem_trywait()</code>
2794	<code>pthread_cond_wait()</code>	<code>pthread_rwlock_clockwrlock()</code>	<code>sem_wait()</code>
2795	<code>pthread_create()</code>	<code>pthread_rwlock_rdlock()</code>	<code>semctl()</code>
2796	<code>pthread_join()</code>	<code>pthread_rwlock_timedrdlock()</code>	<code>semop()</code>
2797	<code>pthread_mutex_clocklock()</code>	<code>pthread_rwlock_timedwrlock()</code>	<code>wait()</code>
2798	<code>pthread_mutex_lock()</code>	<code>pthread_rwlock_tryrdlock()</code>	<code>waitpid()</code>
2799	<code>pthread_mutex_timedlock()</code>	<code>pthread_rwlock_trywrlock()</code>	

2800 The `pthread_once()` function shall synchronize memory for the first call in each thread for a given
 2801 **pthread_once_t** object. If the `init_routine` called by `pthread_once()` is a cancellation point and is
 2802 canceled, a call to `pthread_once()` for the same **pthread_once_t** object made from a cancellation
 2803 cleanup handler shall also synchronize memory.

2804 The `pthread_mutex_lock()` function need not synchronize memory if the mutex type is
 2805 PTHREAD_MUTEX_RECURSIVE and the calling thread already owns the mutex. The
 2806 `pthread_mutex_unlock()` function need not synchronize memory if the mutex type is
 2807 PTHREAD_MUTEX_RECURSIVE and the mutex has a lock count greater than one.

2808 Unless explicitly stated otherwise, if one of the above functions returns an error, it is unspecified
 2809 whether the invocation causes memory to be synchronized.

2810 Applications may allow more than one thread of control to read a memory location
 2811 simultaneously.

2812 4.14 Pathname Resolution

2813 Pathname resolution is performed for a process to resolve a pathname to a particular directory
 2814 entry for a file in the file hierarchy. There may be multiple pathnames that resolve to the same
 2815 directory entry, and multiple directory entries for the same file. When a process resolves a
 2816 pathname of an existing directory entry, the entire pathname shall be resolved as described
 2817 below. When a process resolves a pathname of a directory entry that is to be created immediately
 2818 after the pathname is resolved, pathname resolution terminates when all components of the path
 2819 prefix of the last component have been resolved. It is then the responsibility of the process to
 2820 create the final component.

2821 Each filename in the pathname is located in the directory specified by its predecessor (for
 2822 example, in the pathname fragment **a/b**, file **b** is located in directory **a**). Pathname resolution
 2823 shall fail if this cannot be accomplished. If the pathname begins with a <slash>, the predecessor
 2824 of the first filename in the pathname shall be taken to be the root directory of the process (such
 2825 pathnames are referred to as “absolute pathnames”). If the pathname does not begin with a
 2826 <slash>, the predecessor of the first filename of the pathname shall be taken to be either the
 2827 current working directory of the process or for certain interfaces the directory identified by a file

7540 **NAME**

7541 dirent.h — format of directory entries

7542 **SYNOPSIS**

7543 #include <dirent.h>

7544 **DESCRIPTION**

7545 The internal format of directories is unspecified.

7546 The **<dirent.h>** header shall define the following type:7547 **DIR** A type representing a directory stream. The **DIR** type may be an incomplete type.7548 It shall also define the structure **dirent** which shall include the following members:

7549 ino_t d_ino File serial number. —

7550 char d_name[] Filename string of entry. —

7551 and the structure **posix_dent** which shall include the following members: |

7552 ino_t d_ino File serial number. |

7553 reflen_t d_reflen Length of this entry, including trailing |
7554 padding if necessary. See *posix_getdents()*. |

7555 unsigned char d_type File type or unknown-file-type indication. |

7556 char d_name[] Filename string of this entry. |

7557 The array *d_name* in each of these structures is of unspecified size, but shall contain a filename of |
7558 at most {NAME_MAX} bytes followed by a terminating null byte. |7559 The **<dirent.h>** header shall define the **ino_t**, **reflen_t**, **size_t**, and **ssize_t** types as described in +
7560 **<sys/types.h>**. +7561 The **<dirent.h>** header shall define the following symbolic constants for the file types and +
7562 unknown-file-type indicator returned in the *d_type* member of the **posix_dent** structure. The +
7563 values shall be distinct and shall be suitable for use in **#if** preprocessing directives: +

7564 DT_BLK Block special. +

7565 DT_CHR Character special. +

7566 DT_DIR Directory. +

7567 DT_FIFO FIFO special. +

7568 DT_LNK Symbolic link. +

7569 DT_REG Regular. +

7570 DT_SOCKET Socket. +

7571 DT_UNKNOWN +

7572 Unknown file type. +

7573 TYM The implementation may implement message queues, semaphores, shared memory objects or +
7574 typed memory objects as distinct file types. The following macros shall be provided to represent +
7575 these types. The values shall be distinct from each other and from the above symbolic constants +
7576 beginning with DT_, except when a distinct file type is not implemented, in which case the +
7577 corresponding constant shall have a value that is never returned in *d_type* by *posix_getdents()*. +
7578 The values shall be suitable for use in **#if** preprocessing directives: +

7579 DT_MQ Message queue. +

```

7580      DT_SEM    Semaphore.                                     +
7581      DT_SHM    Shared memory object.                       +
7582  TYM      DT_TMO    Typed memory object.                   +
7583      The following shall be declared as functions and may also be defined as macros. Function
7584      prototypes shall be provided.
7585      int        alphasort(const struct dirent **, const struct dirent **);
7586      int        closedir(DIR *);
7587      int        dirfd(DIR *);
7588      DIR        *fdopendir(int);
7589      DIR        *opendir(const char *);
7590      ssize_t    posix_getdents(int, void *, size_t, int);     +
7591      struct dirent *readdir(DIR *);
7592      int        readdir_r(DIR *restrict, struct dirent *restrict,
7593      struct dirent **restrict);
7594      void        rewinddir(DIR *);
7595      int        scandir(const char *, struct dirent ***,
7596      int (*)(const struct dirent *),
7597      int (*)(const struct dirent **),
7598      const struct dirent **));
7599  XSI      void        seekdir(DIR *, long);
7600      long        telldir(DIR *);

```

7601 **APPLICATION USAGE**

7602 None.

7603 **RATIONALE**

7604 Information similar to that in the <dirent.h> header is contained in a file <sys/dir.h> in 4.2 BSD
7605 and 4.3 BSD. The equivalent in these implementations of **struct dirent** from this volume of
7606 POSIX.1-202x is **struct direct**. The filename was changed because the name <sys/dir.h> was also
7607 used in earlier implementations to refer to definitions related to the older access method; this
7608 produced name conflicts. The name of the structure was changed because this volume of
7609 POSIX.1-202x does not completely define what is in the structure, so it could be different on
7610 some implementations from **struct direct**.

7611 The **posix_dent** structure was based on existing structures used by traditional *getdents()* +
7612 functions, but the name was changed because the existing structures differed in name and in +
7613 their members. Some used the **dirent** structure but this is not required to include a *d_type* +
7614 member, which is the main advantage of using *posix_getdents()* over *readdir()*. The *d_reclen* +
7615 member was included, even though some implementations return fixed-length entries and +
7616 therefore do not need it, as almost all existing code that used *getdents()* used *d_reclen* to iterate +
7617 through the returned entries. Implementations that return fixed-length entries can simply set +
7618 *d_reclen* to that length in *posix_getdents()*. The type **reclen_t** for *d_reclen* was introduced, instead +
7619 of using **unsigned short**, so as not to create a requirement that {NAME_MAX} cannot be greater +
7620 than (a value somewhat smaller than) {SHRT_MAX}. +

7621 Implementations are encouraged to define a DT_FORCE_TYPE symbolic constant for use in the +
7622 *flags* argument to *posix_getdents()*. See the RATIONALE for *posix_getdents()*. +

7623 The name of an array of **char** of an unspecified size should not be used as an lvalue. Use of:

7624 `sizeof(d_name)`

7625 is incorrect; use:

7626 `strlen(d_name)`

7627 instead.

7628 The array of **char** *d_name* cannot be assumed to have a fixed size. Implementations may define
7629 the *d_name* array in the **dirent** and **posix_dent** structures to have size 1, or size greater than
7630 {NAME_MAX}, or use a flexible array member, but in all cases the actual number of characters
7631 used for *d_name* is at least the length of the filename string including the terminating NUL byte.

7632 **FUTURE DIRECTIONS**

7633 A future version of this standard may add a DT_FORCE_TYPE symbolic constant for use as
7634 described in the RATIONALE for *posix_getdents()*.

7635 **SEE ALSO**

7636 [<sys/types.h>](#)

7637 XSH *alphasort()*, *closedir()*, *dirfd()*, *fdopendir()*, *posix_getdents()*, *readdir()*, *rewinddir()*, *seekdir()*,
7638 *telldir()* +

7639 **CHANGE HISTORY**

7640 First released in Issue 2.

7641 **Issue 5**

7642 The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

7643 **Issue 6**

7644 The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r()*.

7645 The **restrict** keyword is added to the prototype for *readdir_r()*.

7646 **Issue 7**

7647 The *alphasort()*, *dirfd()*, and *scandir()* functions are added from The Open Group Technical
7648 Standard, 2006, Extended API Set Part 1.

7649 The *fdopendir()* function is added from The Open Group Technical Standard, 2006, Extended API
7650 Set Part 2.

7651 Austin Group Interpretation 1003.1-2001 #110 is applied, clarifying the definition of the **DIR**
7652 type.

7653 POSIX.1-2008, Technical Corrigendum 1, XBD/TC1-2008/0039 [291], XBD/TC1-2008/0040 [291],
7654 XBD/TC1-2008/0041 [291], and XBD/TC1-2008/0042 [206] are applied. +

7655 **Issue 8** +

7656 Austin Group Defect 697 is applied, adding *posix_getdents()*.

7657 **NAME**

7658 dlfcn.h — dynamic linking

7659 **SYNOPSIS**

7660 #include <dlfcn.h>

7661 **DESCRIPTION**

7662 The <dlfcn.h> header shall define the **Dl_info_t** structure type, which shall include at least the +
7663 following members: +

7664	const char *	dli_fname	Pathname of mapped object file.	+
7665	void *	dli_fbase	Base of mapped address range.	+
7666	const char *	dli_sname	Symbol name or null pointer.	+
7667	void *	dli_saddr	Symbol address or null pointer.	+

7668 The <dlfcn.h> header shall define at least the following symbolic constants for use in the +
7669 construction of a *dlopen()* *mode* argument:

7670	RTLD_LAZY	Relocations are performed at an implementation-defined time.
7671	RTLD_NOW	Relocations are performed when the object is loaded.
7672	RTLD_GLOBAL	All symbols are available for relocation processing of other modules.
7673	RTLD_LOCAL	All symbols are not made available for relocation processing by other + 7674 modules.

7675 The following shall be declared as functions and may also be defined as macros. Function +
7676 prototypes shall be provided.

7677	int	dladdr(const void *restrict, Dl_info_t *restrict);	+
7678	int	dlclose(void *);	
7679	char *	dLError(void);	
7680	void *	dlopen(const char *, int);	
7681	void *	dlsym(void *restrict, const char *restrict);	

7682 **APPLICATION USAGE**

7683 None.

7684 **RATIONALE**

7685 None.

7686 **FUTURE DIRECTIONS**

7687 None.

7688 **SEE ALSO**

7689 XSH *dladdr()*, *dlclose()*, *dLError()*, *dlopen()*, *dlsym()* +

7690 **CHANGE HISTORY**

7691 First released in Issue 5.

7692 **Issue 6**

7693 The **restrict** keyword is added to the prototype for *dlsym()*.

7694 **Issue 7**

7695 The <dlfcn.h> header is moved from the XSI option to the Base.

7696 This reference page is clarified with respect to macros and symbolic constants. +

9402 **Other Invariant Values**

9403 The <limits.h> header shall define the following symbolic constants:

- 9404 {GETENTROPY_MAX} +
- 9405 The maximum value of the *length* argument in calls to the *getentropy()* function. +
- 9406 Minimum Acceptable Value: 256

- 9407 {NL_ARGMAX}
- 9408 Maximum value of *n* in conversion specifications using the "%n\$" sequence in calls to the
- 9409 *printf()* and *scanf()* families of functions.
- 9410 Minimum Acceptable Value: 9

- 9411 XSI {NL_LANGMAX}
- 9412 Maximum number of bytes in a *LANG* name.
- 9413 Minimum Acceptable Value: 14

- 9414 {NL_MSGMAX}
- 9415 Maximum message number.
- 9416 Minimum Acceptable Value: 32 767

- 9417 {NL_SETMAX}
- 9418 Maximum set number.
- 9419 Minimum Acceptable Value: 255

- 9420 {NL_TEXTMAX}
- 9421 Maximum number of bytes in a message string.
- 9422 Minimum Acceptable Value: {_POSIX2_LINE_MAX}

- 9423 {NSIG_MAX}
- 9424 Maximum possible return value of *sysconf*(_SC_NSIG). See XSH *sysconf()*. The value of
- 9425 {NSIG_MAX} shall be no greater than the number of signals that the **sigset_t** type (see
- 9426 <signal.h>) is capable of representing, ignoring any restrictions imposed by *sigfillset()* or
- 9427 *sigaddset()*.

- 9428 XSI {NZERO}
- 9429 Default process priority.
- 9430 Minimum Acceptable Value: 20

9431 **APPLICATION USAGE**

9432 None.

9433 **RATIONALE**

9434 A request was made to reduce the value of {_POSIX_LINK_MAX} from the value of 8 specified
 9435 for it in the POSIX.1-1990 standard to 2. The standard developers decided to deny this request
 9436 for several reasons:

- 9437 • They wanted to avoid making any changes to the standard that could break conforming
- 9438 applications, and the requested change could have that effect.

- 9439 • The use of multiple hard links to a file cannot always be replaced with use of symbolic
- 9440 links. Symbolic links are semantically different from hard links in that they associate a
- 9441 pathname with another pathname rather than a pathname with a file. This has
- 9442 implications for access control, file permanence, and transparency.

- 9443 • The original standard developers had considered the issue of allowing for
- 9444 implementations that did not in general support hard links, and decided that this would
- 9445 reduce consensus on the standard.

9612 CX The <locale.h> header shall contain at least the following macros representing bitmasks for use
9613 with the *newlocale()* function for each supported locale category:

- 9614 LC_COLLATE_MASK
- 9615 LC_CTYPE_MASK
- 9616 LC_MESSAGES_MASK
- 9617 LC_MONETARY_MASK
- 9618 LC_NUMERIC_MASK
- 9619 LC_TIME_MASK

9620 In addition, a macro to set the bits for all categories set shall be defined:

9621 LC_ALL_MASK

9622 The <locale.h> header shall define LC_GLOBAL_LOCALE, a special locale object descriptor
9623 used by the *duplocale()* and *uselocale()* functions.

9624 The <locale.h> header shall define the **locale_t** type, representing a locale object.

9625 The following shall be declared as functions and may also be defined as macros. Function
9626 prototypes shall be provided for use with ISO C standard compilers.

```

9627 CX locale_t      duplocale(locale_t);
9628 void          freelocale(locale_t);
9629 const char    *getlocalename_l(int, locale_t);
9630 struct lconv  *localeconv(void);
9631 CX locale_t    newlocale(int, const char *, locale_t);
9632 char          *setlocale(int, const char *);
9633 CX locale_t    uselocale (locale_t);

```

9634 **APPLICATION USAGE**

9635 None.

9636 **RATIONALE**

9637 It is suggested that each category macro name for use in *setlocale()* have a corresponding macro
9638 name ending in *_MASK* for use in *newlocale()*.

9639 **FUTURE DIRECTIONS**

9640 None.

9641 **SEE ALSO**

9642 [Chapter 8](#) (on page 153), [<stddef.h>](#)
9643 XSH *duplocale()*, *freelocale()*, *getlocalename_l()*, *localeconv()*, *newlocale()*, *setlocale()*, *uselocale()* +

9644 **CHANGE HISTORY**

9645 First released in Issue 3.

9646 Included for alignment with the ISO C standard.

9647 **Issue 6**

9648 The **lconv** structure is expanded with new members (**int_n_cs_precedes**, **int_n_sep_by_space**,
9649 **int_n_sign_posn**, **int_p_cs_precedes**, **int_p_sep_by_space**, and **int_p_sign_posn**) for alignment
9650 with the ISO/IEC 9899:1999 standard.

9651 Extensions beyond the ISO C standard are marked.

10565 **NAME**

10566 poll.h — definitions for the poll() function

10567 **SYNOPSIS**

10568 #include <poll.h>

10569 **DESCRIPTION**

10570 The <poll.h> header shall define the **pollfd** structure, which shall include at least the following
 10571 members:

10572 int fd The following descriptor being polled.
 10573 short events The input event flags (see below).
 10574 short revents The output event flags (see below).

10575 The <poll.h> header shall define the following type through **typedef**:

10576 **nfds_t** An unsigned integer type used for the number of file descriptors.

10577 The implementation shall support one or more programming environments in which the width
 10578 of **nfds_t** is no greater than the width of type **long**. The names of these programming
 10579 environments can be obtained using the *confstr()* function or the *getconf* utility.

10580 The <poll.h> header shall define the **sigset_t** type as described in <signal.h>. +

10581 The <poll.h> header shall define the **timespec** structure as described in <time.h>. +

10582 The <poll.h> header shall define the following symbolic constants, zero or more of which may
 10583 be OR'ed together to form the *events* or *revents* members in the **pollfd** structure:

- 10584 POLLIN Data other than high-priority data may be read without blocking.
- 10585 POLLRDNORM Normal data may be read without blocking.
- 10586 POLLRDBAND Priority data may be read without blocking.
- 10587 POLLPRI High priority data may be read without blocking.
- 10588 POLLOUT Normal data may be written without blocking.
- 10589 POLLWRNORM Equivalent to POLLOUT.
- 10590 POLLWRBAND Priority data may be written.
- 10591 POLLERR An error has occurred (*revents* only).
- 10592 POLLHUP Device has been disconnected (*revents* only).
- 10593 POLLNVAL Invalid *fd* member (*revents* only).

10594 The significance and semantics of normal, priority, and high-priority data are file and device-
 10595 specific.

10596 The following shall be declared as functions and may also be defined as macros. Function |
 10597 prototypes shall be provided.

```
10598 int poll(struct pollfd [], nfds_t, int);
10599 int ppoll(struct pollfd [], nfds_t, const struct timespec *restrict, +
10600           const sigset_t *restrict); +
```

10601 Inclusion of the <poll.h> header may make visible all symbols from the headers <signal.h> and +
 10602 <time.h>.


```

10671     int    pthread_atfork(void (*) (void), void (*) (void),
10672                          void(*) (void));
10673     int    pthread_attr_destroy(pthread_attr_t *);
10674     int    pthread_attr_getdetachstate(const pthread_attr_t *, int *);
10675     int    pthread_attr_getguardsize(const pthread_attr_t *restrict,
10676                                     size_t *restrict);
10677     TPS   int    pthread_attr_getinheritsched(const pthread_attr_t *restrict,
10678                                             int *restrict);
10679     int    pthread_attr_getschedparam(const pthread_attr_t *restrict,
10680                                     struct sched_param *restrict);
10681     TPS   int    pthread_attr_getschedpolicy(const pthread_attr_t *restrict,
10682                                             int *restrict);
10683     int    pthread_attr_getscope(const pthread_attr_t *restrict,
10684                                 int *restrict);
10685     TSA   TSS int    pthread_attr_getstack(const pthread_attr_t *restrict,
10686                                           void **restrict, size_t *restrict);
10687     TSS   int    pthread_attr_getstacksize(const pthread_attr_t *restrict,
10688                                           size_t *restrict);
10689     int    pthread_attr_init(pthread_attr_t *);
10690     int    pthread_attr_setdetachstate(pthread_attr_t *, int);
10691     int    pthread_attr_setguardsize(pthread_attr_t *, size_t);
10692     TPS   int    pthread_attr_setinheritsched(pthread_attr_t *, int);
10693     int    pthread_attr_setschedparam(pthread_attr_t *restrict,
10694                                     const struct sched_param *restrict);
10695     TPS   int    pthread_attr_setschedpolicy(pthread_attr_t *, int);
10696     int    pthread_attr_setscope(pthread_attr_t *, int);
10697     TSA   TSS int    pthread_attr_setstack(pthread_attr_t *, void *, size_t);
10698     TSS   int    pthread_attr_setstacksize(pthread_attr_t *, size_t);
10699     int    pthread_barrier_destroy(pthread_barrier_t *);
10700     int    pthread_barrier_init(pthread_barrier_t *restrict,
10701                               const pthread_barrierattr_t *restrict, unsigned);
10702     int    pthread_barrier_wait(pthread_barrier_t *);
10703     int    pthread_barrierattr_destroy(pthread_barrierattr_t *);
10704     TSH   int    pthread_barrierattr_getpshared(
10705         const pthread_barrierattr_t *restrict, int *restrict);
10706     int    pthread_barrierattr_init(pthread_barrierattr_t *);
10707     TSH   int    pthread_barrierattr_setpshared(pthread_barrierattr_t *, int);
10708     int    pthread_cancel(pthread_t);
10709     int    pthread_cond_broadcast(pthread_cond_t *);
10710     int    pthread_cond_clockwait(pthread_cond_t *restrict,
10711                                  pthread_mutex_t *restrict, clockid_t,
10712                                  const struct timespec *restrict);
10713     int    pthread_cond_destroy(pthread_cond_t *);
10714     int    pthread_cond_init(pthread_cond_t *restrict,
10715                             const pthread_condattr_t *restrict);
10716     int    pthread_cond_signal(pthread_cond_t *);
10717     int    pthread_cond_timedwait(pthread_cond_t *restrict,
10718                                  pthread_mutex_t *restrict, const struct timespec *restrict);
10719     int    pthread_cond_wait(pthread_cond_t *restrict,
10720                              pthread_mutex_t *restrict);
10721     int    pthread_condattr_destroy(pthread_condattr_t *);
10722     int    pthread_condattr_getclock(const pthread_condattr_t *restrict,

```

```

10723         clockid_t *restrict);
10724 TSH int pthread_condattr_getpshared(const pthread_condattr_t *restrict,
10725         int *restrict);
10726 int pthread_condattr_init(pthread_condattr_t *);
10727 int pthread_condattr_setclock(pthread_condattr_t *, clockid_t);
10728 TSH int pthread_condattr_setpshared(pthread_condattr_t *, int);
10729 int pthread_create(pthread_t *restrict, const pthread_attr_t *restrict,
10730         void *(*)(void*), void *restrict);
10731 int pthread_detach(pthread_t);
10732 int pthread_equal(pthread_t, pthread_t);
10733 void pthread_exit(void *);
10734 TCT int pthread_getcpuclockid(pthread_t, clockid_t *);
10735 TPS int pthread_getschedparam(pthread_t, int *restrict,
10736         struct sched_param *restrict);
10737 void *pthread_getspecific(pthread_key_t);
10738 int pthread_join(pthread_t, void **);
10739 int pthread_key_create(pthread_key_t *, void (*)(void*));
10740 int pthread_key_delete(pthread_key_t);
10741 int pthread_mutex_clocklock(pthread_mutex_t *restrict, clockid_t,
10742         const struct timespec *restrict);
10743 int pthread_mutex_consistent(pthread_mutex_t *);
10744 int pthread_mutex_destroy(pthread_mutex_t *);
10745 RPP|TPP int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict,
10746         int *restrict);
10747 int pthread_mutex_init(pthread_mutex_t *restrict,
10748         const pthread_mutexattr_t *restrict);
10749 int pthread_mutex_lock(pthread_mutex_t *);
10750 RPP|TPP int pthread_mutex_setprioceiling(pthread_mutex_t *restrict, int,
10751         int *restrict);
10752 int pthread_mutex_timedlock(pthread_mutex_t *restrict,
10753         const struct timespec *restrict);
10754 int pthread_mutex_trylock(pthread_mutex_t *);
10755 int pthread_mutex_unlock(pthread_mutex_t *);
10756 int pthread_mutexattr_destroy(pthread_mutexattr_t *);
10757 RPP|TPP int pthread_mutexattr_getprioceiling(
10758         const pthread_mutexattr_t *restrict, int *restrict);
10759 MC1 int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict,
10760         int *restrict);
10761 TSH int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict,
10762         int *restrict);
10763 int pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict,
10764         int *restrict);
10765 int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict,
10766         int *restrict);
10767 int pthread_mutexattr_init(pthread_mutexattr_t *);
10768 RPP|TPP int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *, int);
10769 MC1 int pthread_mutexattr_setprotocol(pthread_mutexattr_t *, int);
10770 TSH int pthread_mutexattr_setpshared(pthread_mutexattr_t *, int);
10771 int pthread_mutexattr_setrobust(pthread_mutexattr_t *, int);
10772 int pthread_mutexattr_settype(pthread_mutexattr_t *, int);
10773 int pthread_once(pthread_once_t *, void (*)(void));
10774 int pthread_rwlock_destroy(pthread_rwlock_t *);

```

```

10775     int    pthread_rwlock_init(pthread_rwlock_t *restrict,
10776         const pthread_rwlockattr_t *restrict);
10777     int    pthread_rwlock_clockrdlock(pthread_rwlock_t *restrict,          +
10778         clockid_t, const struct timespec *restrict);                      +
10779     int    pthread_rwlock_clockwrlock(pthread_rwlock_t *restrict,          +
10780         clockid_t, const struct timespec *restrict);                      +
10781     int    pthread_rwlock_rdlock(pthread_rwlock_t *);
10782     int    pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict,
10783         const struct timespec *restrict);
10784     int    pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict,
10785         const struct timespec *restrict);
10786     int    pthread_rwlock_tryrdlock(pthread_rwlock_t *);
10787     int    pthread_rwlock_trywrlock(pthread_rwlock_t *);
10788     int    pthread_rwlock_unlock(pthread_rwlock_t *);
10789     int    pthread_rwlock_wrlock(pthread_rwlock_t *);
10790     int    pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
10791     TSH    int    pthread_rwlockattr_getpshared(
10792         const pthread_rwlockattr_t *restrict, int *restrict);
10793     int    pthread_rwlockattr_init(pthread_rwlockattr_t *);
10794     TSH    int    pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);
10795     pthread_t
10796         pthread_self(void);
10797     int    pthread_setcancelstate(int, int *);
10798     int    pthread_setcanceltype(int, int *);
10799     TPS    int    pthread_setschedparam(pthread_t, int,
10800         const struct sched_param *);
10801     int    pthread_setschedprio(pthread_t, int);
10802     int    pthread_setspecific(pthread_key_t, const void *);
10803     int    pthread_spin_destroy(pthread_spinlock_t *);
10804     int    pthread_spin_init(pthread_spinlock_t *, int);
10805     int    pthread_spin_lock(pthread_spinlock_t *);
10806     int    pthread_spin_trylock(pthread_spinlock_t *);
10807     int    pthread_spin_unlock(pthread_spinlock_t *);
10808     void    pthread_testcancel(void);

```

10809 The following may be declared as functions, or defined as macros, or both. If functions are
10810 declared, function prototypes shall be provided.

```

10811     pthread_cleanup_pop()
10812     pthread_cleanup_push()

```

10813 Inclusion of the **<pthread.h>** header shall make symbols defined in the headers **<sched.h>** and
10814 **<time.h>** visible.

11165 **NAME**

11166 semaphore.h — semaphores

11167 **SYNOPSIS**

11168 #include <semaphore.h>

11169 **DESCRIPTION**

11170 The <semaphore.h> header shall define the **sem_t** type, used in performing semaphore
 11171 operations. The semaphore may be implemented using a file descriptor, in which case
 11172 applications are able to open up at least a total of {OPEN_MAX} files and semaphores.

11173 The <semaphore.h> header shall define the **timespec** structure as described in <time.h>.

11174 The <semaphore.h> header shall define the symbolic constant SEM_FAILED which shall have
 11175 type **sem_t** *.

11176 The <semaphore.h> header shall define O_CREAT and O_EXCL as described in <fcntl.h>.

11177 The following shall be declared as functions and may also be defined as macros. Function
 11178 prototypes shall be provided.

```

11179 int      sem_clockwait(sem_t *restrict, clockid_t,                +
11180                    const struct timespec *restrict);           +
11181 int      sem_close(sem_t *);
11182 int      sem_destroy(sem_t *);
11183 int      sem_getvalue(sem_t *restrict, int *restrict);
11184 int      sem_init(sem_t *, int, unsigned);
11185 sem_t *sem_open(const char *, int, ...);
11186 int      sem_post(sem_t *);
11187 int      sem_timedwait(sem_t *restrict, const struct timespec *restrict);
11188 int      sem_trywait(sem_t *);
11189 int      sem_unlink(const char *);
11190 int      sem_wait(sem_t *);
    
```

11191 Inclusion of the <semaphore.h> header may make visible symbols defined in the <fcntl.h> and
 11192 <time.h> headers.

11193 **APPLICATION USAGE**

11194 None.

11195 **RATIONALE**

11196 None.

11197 **FUTURE DIRECTIONS**

11198 None.

11199 **SEE ALSO**

11200 <fcntl.h>, <sys/types.h>, <time.h>

11201 XSH *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*, *sem_open()*, *sem_post()*,
 11202 *sem_timedwait()*, *sem_trywait()*, *sem_unlink()*

11203 **CHANGE HISTORY**

11204 First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

11205 **Issue 6**

11206 The <semaphore.h> header is marked as part of the Semaphores option.

11207 The Open Group Corrigendum U021/3 is applied, adding a description of SEM_FAILED.

11208 The *sem_timedwait()* function is added for alignment with IEEE Std 1003.1d-1999.

11291 The **signal** union shall be defined as:

11292 `int sival_int` Integer signal value.

11293 `void *sival_ptr` Pointer signal value.

11294 The <signal.h> header shall declare the SIGRTMIN and SIGRTMAX macros, which shall expand
 11295 to positive integer expressions with type **int**, but which need not be constant expressions. These
 11296 macros specify a range of signal numbers that are reserved for application use and for which the
 11297 realtime signal behavior specified in this volume of POSIX.1-202x is supported. The signal
 11298 numbers in this range do not overlap any of the signals specified in the following table.

11299 The range SIGRTMIN through SIGRTMAX inclusive shall include at least {RTSIG_MAX} signal
 11300 numbers. The value of SIGRTMAX shall be less than the value returned by *sysconf*(_SC_NSIG).

11301 It is implementation-defined whether realtime signal behavior is supported for other signals. +

11302 The <signal.h> header shall define the following symbolic constant. The value shall be suitable +
 11303 for use in **#if** preprocessing directives: +

11304 SIG2STR_MAX Maximum size of a signal name returned by *sig2str*(), including the +
 11305 terminating null byte. +

11306 The <signal.h> header shall define the following macros that are used to refer to the signals that
 11307 occur in the system. Signals defined here begin with the letters SIG followed by an uppercase
 11308 letter. The macros shall expand to positive integer constant expressions with type **int** and
 11309 CX distinct values less than the value of {NSIG_MAX} defined in <limits.h>. The value 0 is
 11310 reserved for use as the null signal (see *kill*(())). Additional implementation-defined signals may
 11311 occur in the system.

11312 The ISO C standard only requires the signal names SIGABRT, SIGFPE, SIGILL, SIGINT,
 11313 SIGSEGV, and SIGTERM to be defined. An implementation need not generate any of these six
 11314 CX signals, except as a result of explicit use of interfaces that generate signals, such as *raise*(), *kill*(),
 11315 the General Terminal Interface (see Section 11.1.9, on page 185), and the *kill* utility, unless
 11316 otherwise stated (see, for example, XSH Section 2.8.3.3, on page 491).

11317 The following signals shall be supported on all implementations (default actions are explained
 11318 below the table):

11461 CX In addition, the following signal-specific information shall be available:

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

11475 For some implementations, the value of *si_addr* may be inaccurate.

11476 The following shall be declared as functions and may also be defined as macros. Function
11477 prototypes shall be provided.

```

11478 CX int kill(pid_t, int);
11479 XSI int killpg(pid_t, int);
11480 CX void psiginfo(const siginfo_t *, const char *);
11481 void psignal(int, const char *);
11482 int pthread_kill(pthread_t, int);
11483 int pthread_sigmask(int, const sigset_t *restrict,
11484 sigset_t *restrict);
11485 int raise(int);
11486 CX int sig2str(int, char *);
11487 int sigaction(int, const struct sigaction *restrict,
11488 struct sigaction *restrict);
11489 int sigaddset(sigset_t *, int);
11490 XSI int sigaltstack(const stack_t *restrict, stack_t *restrict);
11491 CX int sigdelset(sigset_t *, int);
11492 int sigemptyset(sigset_t *);
11493 int sigfillset(sigset_t *);
11494 int sigismember(const sigset_t *, int);
11495 void (*signal(int, void (*)(int)))(int);
11496 CX int sigpending(sigset_t *);
11497 int sigprocmask(int, const sigset_t *restrict, sigset_t *restrict);
11498 int sigqueue(pid_t, int, union sigval);
11499 int sigsuspend(const sigset_t *);
11500 int sigtimedwait(const sigset_t *restrict, siginfo_t *restrict,
11501 const struct timespec *restrict);
11502 int sigwait(const sigset_t *restrict, int *restrict);
11503 int sigwaitinfo(const sigset_t *restrict, siginfo_t *restrict);
11504 int str2sig(const char *restrict, int *restrict);

```

11505 CX Inclusion of the <signal.h> header may make visible all symbols from the <time.h> header.

```

12398     int          putenv(char *);
12399     void         qsort(void *, size_t, size_t, int (*)(const void *,
12400                    const void *));
12401 CX     void         qsort_r(void *, size_t, size_t, int (*)(const void *,
12402                    const void *, void *), void *);
12403     int          rand(void);
12404 XSI    long         random(void);
12405     void         *realloc(void *, size_t);
12406 CX     void         *reallocarray(void *, size_t, size_t);
12407 XSI    char         *realpath(const char *restrict, char *restrict);
12408     unsigned short *seed48(unsigned short [3]);
12409 CX     int          setenv(const char *, const char *, int);
12410 OB XSI void         setkey(const char *);
12411 XSI    char         *setstate(char *);
12412     void         srand(unsigned);
12413 XSI    void         srand48(long);
12414     void         srandom(unsigned);
12415     double       strtod(const char *restrict, char **restrict);
12416     float        strtof(const char *restrict, char **restrict);
12417     long         strtol(const char *restrict, char **restrict, int);
12418     long double  strtold(const char *restrict, char **restrict);
12419     long long    strtoll(const char *restrict, char **restrict, int);
12420     unsigned long strtoul(const char *restrict, char **restrict, int);
12421     unsigned long long
12422         strtoull(const char *restrict, char **restrict, int);
12423     int          system(const char *);
12424 XSI    int          unlockpt(int);
12425 CX     int          unsetenv(const char *);
12426     size_t       wcstombs(char *restrict, const wchar_t *restrict, size_t);
12427     int          wctomb(char *, wchar_t);

12428 CX     Inclusion of the <stdlib.h> header may also make visible all symbols from <fcntl.h>, <limits.h>,
12429     <math.h>, <stddef.h>, and <sys/wait.h>.

```

12430 APPLICATION USAGE

12431 None.

12432 RATIONALE

12433 The ISO C standard requires that `exit(EXIT_FAILURE)` returns “unsuccessful termination
12434 status” to the host environment. In a POSIX host environment this means that the lower 8 bits of
12435 `EXIT_FAILURE` must have at least one bit set. The standard developers decided to further
12436 restrict the allowed values for the following reasons:

- 12437 • Exit statuses of 126, 127, and greater than 128 are ambiguous in certain circumstances
12438 because they have special meanings in the shell (see XCU Section 2.8.2, on page 2321).
- 12439 • The *xargs* utility quits when a command execution exits with status 255 (see XCU *xargs*).
- 12440 • Calling *exit()* with a value greater than 255 or less than 0 is something that only programs
12441 which are specifically designed to have their exit status obtained by *waitid()* should do
12442 (since it does not truncate the exit status to 8 bits). “Pure ISO C” programs that call
12443 *exit(EXIT_FAILURE)* do not meet this design criterion.

12501 **NAME**

12502 string.h — string operations

12503 **SYNOPSIS**

12504 #include <string.h>

12505 **DESCRIPTION**

12506 CX Some of the functionality described on this reference page extends the ISO C standard.
 12507 Applications shall define the appropriate feature test macro (see XSH Section 2.2, on page 460) to
 12508 enable the visibility of these symbols in this header.

12509 The <string.h> header shall define NULL and size_t as described in <stddef.h>.

12510 CX The <string.h> header shall define the locale_t type as described in <locale.h>.

12511 The following shall be declared as functions and may also be defined as macros. Function
 12512 prototypes shall be provided for use with ISO C standard compilers.

```

12513 XSI void *memcpy(void *restrict, const void *restrict, int, size_t);
12514 void *memchr(const void *, int, size_t);
12515 int memcmp(const void *, const void *, size_t);
12516 void *memcpy(void *restrict, const void *restrict, size_t);
12517 CX void *memmem(const void *, size_t, const void *, size_t); +
12518 void *memmove(void *, const void *, size_t);
12519 void *memset(void *, int, size_t);
12520 CX char *strcpy(char *restrict, const char *restrict);
12521 char *stpncpy(char *restrict, const char *restrict, size_t);
12522 char *strcat(char *restrict, const char *restrict);
12523 char *strchr(const char *, int);
12524 int strcmp(const char *, const char *);
12525 int strcoll(const char *, const char *);
12526 CX int strcoll_l(const char *, const char *, locale_t);
12527 char *strcpy(char *restrict, const char *restrict);
12528 size_t strcspn(const char *, const char *);
12529 CX char *strdup(const char *);
12530 char *strerror(int);
12531 CX char *strerror_l(int, locale_t);
12532 int strerror_r(int, char *, size_t);
12533 size_t strlcat(char *restrict, const char *restrict, size_t); +
12534 size_t strlcpy(char *restrict, const char *restrict, size_t); +
12535 size_t strlen(const char *);
12536 char *strncat(char *restrict, const char *restrict, size_t);
12537 int strncmp(const char *, const char *, size_t);
12538 char *strncpy(char *restrict, const char *restrict, size_t);
12539 CX char *strndup(const char *, size_t);
12540 size_t strnlen(const char *, size_t);
12541 char *strpbrk(const char *, const char *);
12542 char *strrchr(const char *, int);
12543 CX char *strsignal(int);
12544 size_t strspn(const char *, const char *);
12545 char *strstr(const char *, const char *);
12546 char *strtok(char *restrict, const char *restrict);
12547 CX char *strtok_r(char *restrict, const char *restrict, char **restrict);
12548 size_t strxfrm(char *restrict, const char *restrict, size_t);
    
```


13725	pthread_t	Used to identify a thread.	+
13726	reclen_t	Used for directory entry lengths.	
13727	size_t	Used for sizes of objects.	
13728	ssize_t	Used for a count of bytes or an error indication.	
13729	suseconds_t	Used for time in microseconds.	
13730	time_t	Used for time in seconds.	
13731	timer_t	Used for timer ID returned by <i>timer_create()</i> .	
13732	uid_t	Used for user IDs.	
13733	All of the types shall be defined as arithmetic types of an appropriate length, with the following exceptions:		
13734			
13735	pthread_attr_t		
13736	pthread_barrier_t		
13737	pthread_barrierattr_t		
13738	pthread_cond_t		
13739	pthread_condattr_t		
13740	pthread_key_t		
13741	pthread_mutex_t		
13742	pthread_mutexattr_t		
13743	pthread_once_t		
13744	pthread_rwlock_t		
13745	pthread_rwlockattr_t		
13746	pthread_spinlock_t		
13747	pthread_t		
13748	timer_t		
13749	Additionally:		
13750		• mode_t shall be an integer type.	
13751		• dev_t shall be an integer type.	
13752		• nlink_t , uid_t , gid_t , and id_t shall be integer types.	
13753		• blkcnt_t and off_t shall be signed integer types.	
13754		• fsblkcnt_t , fsfilcnt_t , reclen_t , and ino_t shall be defined as unsigned integer types.	+
13755		• size_t shall be an unsigned integer type.	
13756		• blksize_t , pid_t , and ssize_t shall be signed integer types.	
13757	CX	• clock_t shall be an integer or real-floating type. time_t shall be an integer type.	
13758	The type ssize_t shall be capable of storing values at least in the range $[-1, \{SSIZE_MAX\}]$.		
13759	XSI	The type suseconds_t shall be a signed integer type capable of storing values at least in the range $[-1, 1\ 000\ 000]$.	
13760			
13761	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 .		
13762	The names of these programming environments can be obtained using the <i>confstr()</i> function or the <i>getconf</i> utility.		
13763			
13764			
13765	There are no defined comparison or assignment operators for the following types:		

```

15234     int          dup2(int, int);
15235     int          dup3(int, int, int);
15236     void         _exit(int);
15237  OB XSI void     encrypt(char [64], int);
15238     int          execl(const char *, const char *, ...);
15239     int          execl_e(const char *, const char *, ...);
15240     int          execlp(const char *, const char *, ...);
15241     int          execv(const char *, char *const []);
15242     int          execve(const char *, char *const [], char *const []);
15243     int          execvp(const char *, char *const []);
15244     int          faccessat(int, const char *, int, int);
15245     int          fchdir(int);
15246     int          fchown(int, uid_t, gid_t);
15247     int          fchownat(int, const char *, uid_t, gid_t, int);
15248  SIO  int          fdatsync(int);
15249     int          fexecve(int, char *const [], char *const []);
15250     pid_t        _Fork(void);
15251     pid_t        fork(void);
15252     long         fpathconf(int, int);
15253  FSC  int          fsync(int);
15254     int          ftruncate(int, off_t);
15255     char         *getcwd(char *, size_t);
15256     gid_t        getegid(void);
15257     int          getentropy(void *, size_t);
15258     uid_t        geteuid(void);
15259     gid_t        getgid(void);
15260     int          getgroups(int, gid_t []);
15261  XSI  long         gethostid(void);
15262     int          gethostname(char *, size_t);
15263     char         *getlogin(void);
15264     int          getlogin_r(char *, size_t);
15265     int          getopt(int, char * const [], const char *);
15266     pid_t        getpgid(pid_t);
15267     pid_t        getpgrp(void);
15268     pid_t        getpid(void);
15269     pid_t        getppid(void);
15270     pid_t        getsid(pid_t);
15271     uid_t        getuid(void);
15272     int          isatty(int);
15273     int          lchown(const char *, uid_t, gid_t);
15274     int          link(const char *, const char *);
15275     int          linkat(int, const char *, int, const char *, int);
15276  XSI  int          lockf(int, int, off_t);
15277     off_t        lseek(int, off_t, int);
15278  XSI  int          nice(int);
15279     long         pathconf(const char *, int);
15280     int          pause(void);
15281     int          pipe(int [2]);
15282     int          pipe2(int [2], int);
15283     int          posix_close(int, int);
15284     ssize_t      pread(int, void *, size_t, off_t);
15285     ssize_t      pwrite(int, const void *, size_t, off_t);

```

+

```

15664 int fputws(const wchar_t *restrict, FILE *restrict);
15665 int fwide(FILE *, int);
15666 int fwprintf(FILE *restrict, const wchar_t *restrict, ...);
15667 int fwscanf(FILE *restrict, const wchar_t *restrict, ...);
15668 wint_t getwc(FILE *);
15669 wint_t getwchar(void);
15670 size_t mbrlen(const char *restrict, size_t, mbstate_t *restrict);
15671 size_t mbrtowc(wchar_t *restrict, const char *restrict, size_t,
15672 mbstate_t *restrict);
15673 int mbsinit(const mbstate_t *);
15674 CX size_t mbsnrtowcs(wchar_t *restrict, const char **restrict,
15675 size_t, size_t, mbstate_t *restrict);
15676 size_t mbsrtowcs(wchar_t *restrict, const char **restrict, size_t,
15677 mbstate_t *restrict);
15678 CX FILE *open_wmemstream(wchar_t **, size_t *);
15679 wint_t putwc(wchar_t, FILE *);
15680 wint_t putwchar(wchar_t);
15681 int swprintf(wchar_t *restrict, size_t,
15682 const wchar_t *restrict, ...);
15683 int swscanf(const wchar_t *restrict,
15684 const wchar_t *restrict, ...);
15685 wint_t ungetwc(wint_t, FILE *);
15686 int vfwprintf(FILE *restrict, const wchar_t *restrict, va_list);
15687 int vfwscanf(FILE *restrict, const wchar_t *restrict, va_list);
15688 int vswprintf(wchar_t *restrict, size_t,
15689 const wchar_t *restrict, va_list);
15690 int vswscanf(const wchar_t *restrict, const wchar_t *restrict,
15691 va_list);
15692 int vwprintf(const wchar_t *restrict, va_list);
15693 int vwscanf(const wchar_t *restrict, va_list);
15694 CX wchar_t *wcpcpy(wchar_t *restrict, const wchar_t *restrict);
15695 wchar_t *wcpncpy(wchar_t *restrict, const wchar_t *restrict, size_t);
15696 size_t wcrtoomb(char *restrict, wchar_t, mbstate_t *restrict);
15697 CX int wcscasecmp(const wchar_t *, const wchar_t *);
15698 int wcscasecmp_l(const wchar_t *, const wchar_t *, locale_t);
15699 wchar_t *wscat(wchar_t *restrict, const wchar_t *restrict);
15700 wchar_t *wcschr(const wchar_t *, wchar_t);
15701 int wcscmp(const wchar_t *, const wchar_t *);
15702 int wcscoll(const wchar_t *, const wchar_t *);
15703 CX int wcscoll_l(const wchar_t *, const wchar_t *, locale_t);
15704 wchar_t *wcscpy(wchar_t *restrict, const wchar_t *restrict);
15705 size_t wcscspn(const wchar_t *, const wchar_t *);
15706 CX wchar_t *wcsdup(const wchar_t *);
15707 size_t wcsftime(wchar_t *restrict, size_t,
15708 const wchar_t *restrict, const struct tm *restrict);
15709 CX size_t wcslcat(wchar_t *restrict, const wchar_t *restrict,
15710 size_t);
15711 size_t wcsncpy(wchar_t *restrict, const wchar_t *restrict,
15712 size_t);
15713 size_t wcslen(const wchar_t *);
15714 CX int wcsncasecmp(const wchar_t *, const wchar_t *, size_t);
15715 int wcsncasecmp_l(const wchar_t *, const wchar_t *, size_t,

```

	Header	Prefix	Suffix	Complete Name
16135	<aio.h>	aio_, lio_, AIO_, LIO_		
16136	<arpa/inet.h>	inet_		
16137	<ctype.h>	to[a-z], is[a-z]		
16138	<dlfcn.h>	RTLD_, dli_		
16139	<dirent.h>	d_, DT_		
16140	<fcntl.h>	l_		
16141	<fmtmsg.h>	MM_		
16142	<fnmatch.h>	FNM_		
16143	<ftw.h>	FTW		
16144	<glob.h>	gl_, GLOB_		
16145	<grp.h>	gr_		
16146	<limits.h>		_MAX, _MIN	
16147	<math.h>	M_		
16148	<mqueue.h>	mq_, MQ_		
16149	<ndbm.h>	dbm_, DBM_		
16150	<netdb.h>	ai_, h_, n_, p_, s_		
16151	<net/if.h>	if_, IF_		
16152	<netinet/in.h>	in_, ip_, s_, sin_, INADDR_, IPPROTO_		
16153		in6_, in6addr_, s6_, sin6_, IPV6_		
16154	<netinet/tcp.h>	TCP_		
16155	<nl_types.h>	NL_		
16156	<poll.h>	pd_, ph_, ps_, POLL		
16157	<pthread.h>	pthread_, PTHREAD_		
16158	<pwd.h>	pw_		
16159	<regex.h>	re_, rm_, REG_		
16160	<sched.h>	sched_, SCHED_		
16161	<semaphore.h>	sem_, SEM_		
16162	<signal.h>	sa_, si_, sigev_, sival_, uc_, BUS_, CLD_, FPE_, ILL_, SA_, SEGV_, SI_, SIGEV_, ss_, sv_, SS_, TRAP_		
16163	<stdlib.h>	str[a-z]		
16164	<string.h>	str[a-z], mem[a-z], wcs[a-z]		
16165	<sys/ipc.h>	ipc_, IPC_		key, pad, seq
16166	<sys/mman.h>	shm_, MAP_, MCL_, MS_, PROT_		
16167	<sys/msg.h>	msg, MSG_[A-Z]		msg
16168	<sys/resource.h>	rlim_, ru_, PRIO_, RLIMIT_, RUSAGE_		
16169	<sys/select.h>	fd_, fds_, FD_		
16170				
16171				
16172				
16173				
16174				

16894	<i>_Exit()</i>	<i>getpgrp()</i>	<i>read()</i>	<i>strncmp()</i>
16895	<i>_Fork()</i>	<i>getpid()</i>	<i>readlink()</i>	<i>strncpy()</i>
16896	<i>_exit()</i>	<i>getppid()</i>	<i>readlinkat()</i>	<i>strlen()</i>
16897	<i>abort()</i>	<i>getsockname()</i>	<i>recv()</i>	<i>strpbrk()</i>
16898	<i>accept()</i>	<i>getsockopt()</i>	<i>recvfrom()</i>	<i>strrchr()</i>
16899	<i>accept4()</i>	<i>getuid()</i>	<i>recvmsg()</i>	<i>strspn()</i>
16900	<i>access()</i>	<i>htobe16()</i>	<i>rename()</i>	<i>strstr()</i>
16901	<i>aio_error()</i>	<i>htobe32()</i>	<i>renameat()</i>	<i>strtok_r()</i>
16902	<i>aio_return()</i>	<i>htobe64()</i>	<i>rmdir()</i>	<i>symlink()</i>
16903	<i>aio_suspend()</i>	<i>htole16()</i>	<i>select()</i>	<i>symlinkat()</i>
16904	<i>alarm()</i>	<i>htole32()</i>	<i>sem_post()</i>	<i>tcdrain()</i>
16905	<i>be16toh()</i>	<i>htole64()</i>	<i>send()</i>	<i>tcflow()</i>
16906	<i>be32toh()</i>	<i>htonl()</i>	<i>sendmsg()</i>	<i>tcflush()</i>
16907	<i>be64toh()</i>	<i>htons()</i>	<i>sendto()</i>	<i>tcgetattr()</i>
16908	<i>bind()</i>	<i>kill()</i>	<i>setegid()</i>	<i>tcgetpgrp()</i>
16909	<i>cfgetispeed()</i>	<i>le16toh()</i>	<i>seteuid()</i>	<i>tcgetwinsize()</i>
16910	<i>cfgetospeed()</i>	<i>le32toh()</i>	<i>setgid()</i>	<i>tcsendbreak()</i>
16911	<i>cfsetispeed()</i>	<i>le64toh()</i>	<i>setpgid()</i>	<i>tcsetattr()</i>
16912	<i>cfsetospeed()</i>	<i>link()</i>	<i>setregid()</i>	<i>tcsetpgrp()</i>
16913	<i>chdir()</i>	<i>linkat()</i>	<i>setreuid()</i>	<i>tcsetwinsize()</i>
16914	<i>chmod()</i>	<i>listen()</i>	<i>setsid()</i>	<i>time()</i>
16915	<i>chown()</i>	<i>longjmp()</i>	<i>setsockopt()</i>	<i>timer_getoverrun()</i>
16916	<i>clock_gettime()</i>	<i>lseek()</i>	<i>setuid()</i>	<i>timer_gettime()</i>
16917	<i>close()</i>	<i>lstat()</i>	<i>shutdown()</i>	<i>timer_settime()</i>
16918	<i>connect()</i>	<i>memccpy()</i>	<i>sig2str()</i>	<i>times()</i>
16919	<i>creat()</i>	<i>memchr()</i>	<i>sigaction()</i>	<i>umask()</i>
16920	<i>dup()</i>	<i>memcmp()</i>	<i>sigaddset()</i>	<i>uname()</i>
16921	<i>dup2()</i>	<i>memcpy()</i>	<i>sigdelset()</i>	<i>unlink()</i>
16922	<i>dup3()</i>	<i>memmove()</i>	<i>sigemptyset()</i>	<i>unlinkat()</i>
16923	<i>execl()</i>	<i>memset()</i>	<i>sigfillset()</i>	<i>utimensat()</i>
16924	<i>execle()</i>	<i>mkdir()</i>	<i>sigismember()</i>	<i>utimes()</i>
16925	<i>execv()</i>	<i>mkdirat()</i>	<i>siglongjmp()</i>	<i>va_arg()</i>
16926	<i>execve()</i>	<i>mkfifo()</i>	<i>signal()</i>	<i>va_copy()</i>
16927	<i>faccessat()</i>	<i>mkfifoat()</i>	<i>sigpending()</i>	<i>va_end()</i>
16928	<i>fchdir()</i>	<i>mknod()</i>	<i>sigprocmask()</i>	<i>va_start()</i>
16929	<i>fchmod()</i>	<i>mknodat()</i>	<i>sigqueue()</i>	<i>wait()</i>
16930	<i>fchmodat()</i>	<i>ntohl()</i>	<i>sigsuspend()</i>	<i>waitpid()</i>
16931	<i>fchown()</i>	<i>ntohs()</i>	<i>sleep()</i>	<i>wcpcpy()</i>
16932	<i>fchownat()</i>	<i>open()</i>	<i>socketatmark()</i>	<i>wcpncpy()</i>
16933	<i>fcntl()</i>	<i>openat()</i>	<i>socket()</i>	<i>wcscat()</i>
16934	<i>fdatasync()</i>	<i>pause()</i>	<i>socketpair()</i>	<i>wcschr()</i>
16935	<i>fexecve()</i>	<i>pipe()</i>	<i>stat()</i>	<i>wcscmp()</i>
16936	<i>ffs()</i>	<i>pipe2()</i>	<i>stpncpy()</i>	<i>wcscpy()</i>
16937	<i>fstat()</i>	<i>poll()</i>	<i>stpncpy()</i>	<i>wcscspn()</i>
16938	<i>fstatat()</i>	<i>ppoll()</i>	<i>strcat()</i>	<i>wcslcat()</i>
16939	<i>fsync()</i>	<i>pread()</i>	<i>strchr()</i>	<i>wcslcpy()</i>
16940	<i>ftruncate()</i>	<i>pselect()</i>	<i>strcmp()</i>	<i>wcslen()</i>
16941	<i>futimens()</i>	<i>pthread_kill()</i>	<i>strcpy()</i>	<i>wcsncat()</i>
16942	<i>getegid()</i>	<i>pthread_self()</i>	<i>strcspn()</i>	<i>wcsncmp()</i>
16943	<i>geteuid()</i>	<i>pthread_setcancelstate()</i>	<i>strlcat()</i>	<i>wcsncpy()</i>
16944	<i>getgid()</i>	<i>pthread_sigmask()</i>	<i>strncpy()</i>	<i>wcsnlen()</i>
16945	<i>getgroups()</i>	<i>pwrite()</i>	<i>strlen()</i>	<i>wcspbrk()</i>
16946	<i>getpeername()</i>	<i>raise()</i>	<i>strncat()</i>	<i>wcsrchr()</i>

17706 If a thread is detached, its thread ID is invalid for use as an argument in a call to *pthread_detach()*
 17707 or *pthread_join()*.

17708 2.9.3 Thread Mutexes

17709 A thread that has blocked shall not prevent any unblocked thread that is eligible to use the same
 17710 processing resources from eventually making forward progress in its execution. Eligibility for
 17711 processing resources is determined by the scheduling policy.

17712 A thread shall become the owner of a mutex, *m*, when one of the following occurs:

- 17713 • It calls *pthread_mutex_clocklock()*, *pthread_mutex_lock()*, *pthread_mutex_timedlock()*, or |
 17714 *pthread_mutex_trylock()* with *m* as the *mutex* argument and the call returns zero or
 17715 [EOWNERDEAD].
- 17716 • It calls *pthread_mutex_setprioceiling()* with *m* as the *mutex* argument and the call returns -
 17717 [EOWNERDEAD].
- 17718 • It calls *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or *pthread_cond_wait()* with *m* as +
 17719 the *mutex* argument and the call returns zero or certain error numbers (see
 17720 *pthread_cond_timedwait()*). -

17721 The thread shall remain the owner of *m* until one of the following occurs:

- 17722 • It executes *pthread_mutex_unlock()* with *m* as the *mutex* argument
- 17723 • It blocks in a call to *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or +
 17724 *pthread_cond_wait()* with *m* as the *mutex* argument. -

17725 The implementation shall behave as if at all times there is at most one owner of any mutex.

17726 A thread that becomes the owner of a mutex is said to have “acquired” the mutex and the mutex
 17727 is said to have become “locked”; when a thread gives up ownership of a mutex it is said to have
 17728 “released” the mutex and the mutex is said to have become “unlocked”.

17729 A problem can occur if a process terminates while one of its threads holds a mutex lock.
 17730 Depending on the mutex type, it might be possible for another thread to unlock the mutex and
 17731 recover the state of the mutex. However, it is difficult to perform this recovery reliably.

17732 Robust mutexes provide a means to enable the implementation to notify other threads in the
 17733 event of a process terminating while one of its threads holds a mutex lock. The next thread that
 17734 acquires the mutex is notified about the termination by the return value [EOWNERDEAD] from
 17735 the locking function. The notified thread can then attempt to recover the state protected by the
 17736 mutex, and if successful mark the state protected by the mutex as consistent by a call to
 17737 *pthread_mutex_consistent()*. If the notified thread is unable to recover the state, it can declare the
 17738 state as not recoverable by a call to *pthread_mutex_unlock()* without a prior call to
 17739 *pthread_mutex_consistent()*.

17740 Whether or not the state protected by a mutex can be recovered is dependent solely on the
 17741 application using robust mutexes. The robust mutex support provided in the implementation
 17742 provides notification only that a mutex owner has terminated while holding a lock, or that the
 17743 state of the mutex is not recoverable.

17869 2.9.5.2 Cancellation Points

17870 Cancellation points shall occur when a thread is executing the following functions:

17871	<i>accept()</i>	<i>nanosleep()</i>	<i>recvmsg()</i>
17872	<i>accept4()</i>	<i>open()</i>	<i>select()</i>
17873	<i>aio_suspend()</i>	<i>openat()</i>	<i>send()</i>
17874	<i>clock_nanosleep()</i>	<i>pause()</i>	<i>sendmsg()</i>
17875	<i>close()</i>	<i>poll()</i>	<i>sendto()</i>
17876	<i>connect()</i>	<i>ppoll()</i>	<i>sigsuspend()</i>
17877	<i>creat()</i>	<i>pread()</i>	<i>sigtimedwait()</i>
17878	<i>fcntl()†</i>	<i>pselect()</i>	<i>sigwait()</i>
17879	<i>fdatasync()</i>	<i>pthread_cond_clockwait()</i>	<i>sigwaitinfo()</i>
17880	<i>fsync()</i>	<i>pthread_cond_timedwait()</i>	<i>sleep()</i>
17881	<i>lockf()††</i>	<i>pthread_cond_wait()</i>	<i>tcdrain()</i>
17882	<i>mq_receive()</i>	<i>pthread_join()</i>	<i>wait()</i>
17883	<i>mq_send()</i>	<i>pthread_testcancel()</i>	<i>waitid()</i>
17884	<i>mq_timedreceive()</i>	<i>pwrite()</i>	<i>waitpid()</i>
17885	<i>mq_timedsend()</i>	<i>read()</i>	<i>write()</i>
17886	<i>msgrcv()</i>	<i>readv()</i>	<i>writew()</i>
17887	<i>msgsnd()</i>	<i>recv()</i>	
17888	<i>msync()</i>	<i>recvfrom()</i>	

17889 A cancellation point may also occur when a thread is executing the following functions: -

17890	<i>access()</i>	<i>fchownat()</i>	<i>fseeko()</i>
17891	<i>asctime_r()</i>	<i>fclose()</i>	<i>fsetpos()</i>
17892	<i>catclose()</i>	<i>fcntl()†††</i>	<i>fstat()</i>
17893	<i>catopen()</i>	<i>fflush()</i>	<i>fstatat()</i>
17894	<i>chmod()</i>	<i>fgetc()</i>	<i>ftell()</i>
17895	<i>chown()</i>	<i>fgetpos()</i>	<i>ftello()</i>
17896	<i>closedir()</i>	<i>fgets()</i>	<i>futimens()</i>
17897	<i>closelog()</i>	<i>fgetwc()</i>	<i>fwprintf()</i>
17898	<i>ctermid()</i>	<i>fgetws()</i>	<i>fwrite()</i>
17899	<i>ctime_r()</i>	<i>fntmsg()</i>	<i>fwscanf()</i>
17900	<i>dlclose()</i>	<i>fopen()</i>	<i>getaddrinfo()</i>
17901	<i>dlopen()</i>	<i>fpathconf()</i>	<i>getc()</i>
17902	<i>dprintf()</i>	<i>fprintf()</i>	<i>getc_unlocked()</i>
17903	<i>endhostent()</i>	<i>fputc()</i>	<i>getchar()</i>
17904	<i>endnetent()</i>	<i>fputs()</i>	<i>getchar_unlocked()</i>
17905	<i>endprotoent()</i>	<i>fputwc()</i>	<i>getcwd()</i>
17906	<i>endservent()</i>	<i>fputws()</i>	<i>getdelim()</i>
17907	<i>faccessat()</i>	<i>fread()</i>	<i>getgrgid_r()</i>
17908	<i>fchmod()</i>	<i>freopen()</i>	<i>getgrnam_r()</i>
17909	<i>fchmodat()</i>	<i>fscanf()</i>	<i>gethostid()</i>
17910	<i>fchown()</i>	<i>fseek()</i>	<i>gethostname()</i>

17911 † When the *cmd* argument is F_SETLKW.

17912 †† When the *function* argument is F_LOCK.

17913 ††† For any value of the *cmd* argument.

17914	<code>getline()</code>	<code>posix_openpt()</code>	<code>sem_wait()</code>
17915	<code>getlogin_r()</code>	<code>posix_spawn()</code>	<code>semop()</code>
17916	<code>getnameinfo()</code>	<code>posix_spawnnp()</code>	<code>sethostent()</code>
17917	<code>getpwnam_r()</code>	<code>posix_typed_mem_open()</code>	<code>setnetent()</code>
17918	<code>getpwuid_r()</code>	<code>printf()</code>	<code>setprotoent()</code>
17919	<code>getwc()</code>	<code>psiginfo()</code>	<code>setserverent()</code>
17920	<code>getwchar()</code>	<code>psignal()</code>	<code>stat()</code>
17921	<code>glob()</code>	<code>pthread_rwlock_clockrdlock()</code>	<code>strerror_l()</code>
17922	<code>iconv_close()</code>	<code>pthread_rwlock_clockwrlock()</code>	<code>strerror_r()</code>
17923	<code>iconv_open()</code>	<code>pthread_rwlock_rdlock()</code>	<code>strftime()</code>
17924	<code>link()</code>	<code>pthread_rwlock_timedrdlock()</code>	<code>strftime_l()</code>
17925	<code>linkat()</code>	<code>pthread_rwlock_timedwrlock()</code>	<code>symlink()</code>
17926	<code>lio_listio()</code>	<code>pthread_rwlock_wrlock()</code>	<code>symlinkat()</code>
17927	<code>localtime_r()</code>	<code>ptsname()</code>	<code>sync()</code>
17928	<code>lockf()</code>	<code>ptsname_r()</code>	<code>syslog()</code>
17929	<code>lseek()</code>	<code>putc()</code>	<code>tmpfile()</code>
17930	<code>lstat()</code>	<code>putc_unlocked()</code>	<code>tmpnam()</code>
17931	<code>mkdir()</code>	<code>putchar()</code>	<code>ttyname_r()</code>
17932	<code>mkdirat()</code>	<code>putchar_unlocked()</code>	<code>tzset()</code>
17933	<code>mkdtemp()</code>	<code>puts()</code>	<code>ungetc()</code>
17934	<code>mkfifo()</code>	<code>putwc()</code>	<code>ungetwc()</code>
17935	<code>mkfifoat()</code>	<code>putwchar()</code>	<code>unlink()</code>
17936	<code>mknod()</code>	<code>readdir_r()</code>	<code>unlinkat()</code>
17937	<code>mknodat()</code>	<code>readlink()</code>	<code>utimensat()</code>
17938	<code>mkstemp()</code>	<code>readlinkat()</code>	<code>utimes()</code>
17939	<code>mktime()</code>	<code>remove()</code>	<code>vdprintf()</code>
17940	<code>opendir()</code>	<code>rename()</code>	<code>vfprintf()</code>
17941	<code>openlog()</code>	<code>renameat()</code>	<code>vfwprintf()</code>
17942	<code>pathconf()</code>	<code>rewind()</code>	<code>vprintf()</code>
17943	<code>perror()</code>	<code>rewinddir()</code>	<code>vwprintf()</code>
17944	<code>popen()</code>	<code>scandir()</code>	<code>wcsftime()</code>
17945	<code>posix_fadvise()</code>	<code>scanf()</code>	<code>wordexp()</code>
17946	<code>posix_fallocate()</code>	<code>seekdir()</code>	<code>wprintf()</code>
17947	<code>posix_getdents()</code>	<code>sem_clockwait()</code>	<code>wscanf()</code>
17948	<code>posix_madvise()</code>	<code>sem_timedwait()</code>	

17949 In addition, a cancellation point may occur when a thread is executing any function that this
 17950 standard does not require to be thread-safe but the implementation documents as being thread-
 17951 safe. If a thread is cancelled while executing a non-thread-safe function, the behavior is
 17952 undefined.

17953 An implementation shall not introduce cancellation points into any other functions specified in
 17954 this volume of POSIX.1-202x.

17955 The side-effects of acting upon a cancellation request while suspended during a call of a function
 17956 are the same as the side-effects that may be seen in a single-threaded program when a call to a
 17957 function is interrupted by a signal and the given function returns [EINTR]. Any such side-
 17958 effects occur before any cancellation cleanup handlers are called. For functions that are explicitly
 17959 required not to return when interrupted (for example, `pclose()`), if a thread is canceled while
 17960 executing the function, the behavior is undefined.

17961 Whenever a thread has cancelability enabled and a cancellation request has been made with that
 17962 thread as the target, and the thread then calls any function that is a cancellation point (such as
 17963 `pthread_testcancel()` or `read()`), the cancellation request shall be acted upon before the function
 17964 returns. If a thread has cancelability enabled and a cancellation request is made with the thread

18595 **2.11.1 Defined Types**

18596 All of the data types used by various functions are defined by the implementation. The
 18597 following table describes some of these types. Other types referenced in the description of a
 18598 function, not mentioned here, can be found in the appropriate header for that function.

18599	Defined Type	Description
18600	cc_t	Type used for terminal special characters.
18601	clock_t	Integer or real-floating type used for processor times, as defined in the ISO C standard.
18602		
18603	clockid_t	Used for clock ID type in some timer functions.
18604	dev_t	Integer type used for device numbers.
18605	DIR	Type representing a directory stream.
18606	div_t	Structure type returned by the <i>div()</i> function.
18607	FILE	Structure containing information about a file.
18608	glob_t	Structure type used in pathname pattern matching.
18609	fpos_t	Type containing all information needed to specify uniquely every position within a file.
18610		
18611	gid_t	Integer type used for group IDs.
18612	iconv_t	Type used for conversion descriptors.
18613	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 .
18614		
18615	ino_t	Unsigned integer type used for file serial numbers.
18616	key_t	Arithmetic type used for XSI interprocess communication.
18617	ldiv_t	Structure type returned by the <i>ldiv()</i> function.
18618	mode_t	Integer type used for file attributes.
18619	mqd_t	Used for message queue descriptors.
18620	nfds_t	Integer type used for the number of file descriptors.
18621	nlink_t	Integer type used for link counts.
18622	off_t	Signed integer type used for file sizes.
18623	pid_t	Signed integer type used for process and process group IDs.
18624	pthread_attr_t	Used to identify a thread attribute object.
18625	pthread_cond_t	Used for condition variables.
18626	pthread_condattr_t	Used to identify a condition attribute object.
18627	pthread_key_t	Used for thread-specific data keys.
18628	pthread_mutex_t	Used for mutexes.
18629	pthread_mutexattr_t	Used to identify a mutex attribute object.
18630	pthread_once_t	Used for dynamic package initialization.
18631	pthread_rwlock_t	Used for read-write locks.
18632	pthread_rwlockattr_t	Used for read-write lock attributes.
18633	pthread_t	Used to identify a thread.
18634	ptrdiff_t	Signed integer type of the result of subtracting two pointers.
18635	reclen_t	Unsigned integer type used for directory entry lengths.
18636	regex_t	Structure type used in regular expression matching.
18637	regmatch_t	Structure type used in regular expression matching.
18638	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.
18639		
18640	sem_t	Type used in performing semaphore operations.
18641	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.
18642		
18643		
18644	sigset_t	Integer or structure type of an object used to represent sets

21037 **NAME**

21038 bind — bind a name to a socket

21039 **SYNOPSIS**

```
21040 #include <sys/socket.h>
21041 int bind(int socket, const struct sockaddr *address,
21042         socklen_t address_len);
```

21043 **DESCRIPTION**

21044 The *bind()* function shall assign a local socket address *address* to a socket identified by descriptor
 21045 *socket* that has no local socket address assigned. Sockets created with the *socket()* function are
 21046 initially unnamed; they are identified only by their address family.

21047 The *bind()* function takes the following arguments:

21048	<i>socket</i>	Specifies the file descriptor of the socket to be bound.
21049	<i>address</i>	Points to a sockaddr structure containing the address to be bound to the 21050 socket. The length and format of the address depend on the address family of 21051 the socket.
21052	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 21053 argument.

21054 The socket specified by *socket* may require the process to have appropriate privileges to use the
 21055 *bind()* function.

21056 If the address family of the socket is AF_UNIX and the pathname in *address* names a symbolic
 21057 link, *bind()* shall fail and set *errno* to [EADDRINUSE].

21058 If the socket address cannot be assigned immediately and O_NONBLOCK is set for the file
 21059 descriptor for the socket, *bind()* shall fail and set *errno* to [EINPROGRESS], but the assignment
 21060 request shall not be aborted, and the assignment shall be completed asynchronously. Subsequent
 21061 calls to *bind()* for the same socket, before the assignment is completed, shall fail and set *errno* to
 21062 [EALREADY].

21063 When the assignment has been performed asynchronously, *pselect()*, *select()*, *poll()*, and *ppoll()* |
 21064 shall indicate that the file descriptor for the socket is ready for reading and writing.

21065 **RETURN VALUE**

21066 Upon successful completion, *bind()* shall return 0; otherwise, -1 shall be returned and *errno* set
 21067 to indicate the error.

21068 **ERRORS**

21069 The *bind()* function shall fail if:

21070	[EADDRINUSE]	The specified address is already in use.
21071	[EADDRNOTAVAIL]	The specified address is not available from the local machine.
21072	[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the 21073 specified socket.
21074	[EALREADY]	An assignment request is already in progress for the specified socket.
21075	[EBADF]	The <i>socket</i> argument is not a valid file descriptor.
21076		
21077		

23556 **NAME**

23557 connect — connect a socket

23558 **SYNOPSIS**

23559 #include <sys/socket.h>

```
23560 int connect(int socket, const struct sockaddr *address,
23561            socklen_t address_len);
```

23562 **DESCRIPTION**

23563 The *connect()* function shall attempt to make a connection on a connection-mode socket or to set
 23564 or reset the peer address of a connectionless-mode socket. The function takes the following
 23565 arguments:

23566	<i>socket</i>	Specifies the file descriptor associated with the socket.
23567	<i>address</i>	Points to a sockaddr structure containing the peer address. The length and 23568 format of the address depend on the address family of the socket.
23569	<i>address_len</i>	Specifies the length of the sockaddr structure pointed to by the <i>address</i> 23570 argument.

23571 If the socket has not already been bound to a local address, *connect()* shall bind it to an address
 23572 which, unless the socket's address family is AF_UNIX, is an unused local address.

23573 If the initiating socket is not connection-mode, then *connect()* shall set the socket's peer address,
 23574 and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all
 23575 datagrams are sent on subsequent *send()* functions, and limits the remote sender for subsequent
 23576 *recv()* functions. If the *sa_family* member of *address* is AF_UNSPEC, the socket's peer address
 23577 shall be reset. Note that despite no connection being made, the term "connected" is used to
 23578 describe a connectionless-mode socket for which a peer address has been set.

23579 If the initiating socket is connection-mode, then *connect()* shall attempt to establish a connection
 23580 to the address specified by the *address* argument. If the connection cannot be established
 23581 immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect()* shall
 23582 block for up to an unspecified timeout interval until the connection is established. If the timeout
 23583 interval expires before the connection is established, *connect()* shall fail and the connection
 23584 attempt shall be aborted. If *connect()* is interrupted by a signal that is caught while blocked
 23585 waiting to establish a connection, *connect()* shall fail and set *errno* to [EINTR], but the connection
 23586 request shall not be aborted, and the connection shall be established asynchronously.

23587 If the connection cannot be established immediately and O_NONBLOCK is set for the file
 23588 descriptor for the socket, *connect()* shall fail and set *errno* to [EINPROGRESS], but the connection
 23589 request shall not be aborted, and the connection shall be established asynchronously. Subsequent
 23590 calls to *connect()* for the same socket, before the connection is established, shall fail and set *errno*
 23591 to [EALREADY].

23592 When the connection has been established asynchronously, *pselect()*, *select()*, *poll()*, and *ppoll()* |
 23593 shall indicate that the file descriptor for the socket is ready for writing.

23594 The socket in use may require the process to have appropriate privileges to use the *connect()*
 23595 function.

23596 **RETURN VALUE**

23597 Upon successful completion, *connect()* shall return 0; otherwise, -1 shall be returned and *errno*
 23598 set to indicate the error.

		+
24808	NAME	+
24809	dladdr — get information relating to an address	+
24810	SYNOPSIS	+
24811	#include <dlfcn.h>	+
24812	int dladdr(const void *restrict addr, Dl_info_t *restrict dlip);	+
24813	DESCRIPTION	+
24814	The <i>dladdr()</i> function shall determine whether the address specified by <i>addr</i> is located within the	+
24815	address range occupied by a mapped object. The mapped objects examined shall include any	+
24816	executable object files that have previously been loaded by a call to <i>dlopen()</i> and for which	+
24817	<i>dlclose()</i> has not subsequently been called, and any shared library files that were loaded as	+
24818	dependencies of the executable file from which the current process image was loaded; they may	+
24819	also include any executable object files that have previously been loaded by a call to <i>dlopen()</i>	+
24820	and for which <i>dlclose()</i> has subsequently been called, the executable file from which the current	+
24821	process image was loaded, and implementation-defined additional mapped objects (for	+
24822	example, all regular files mapped using <i>mmap()</i> might be included). If the specified address is	+
24823	within the mapped address range of one of these mapped objects and the object contains a	+
24824	symbol table, the symbol table shall be searched for a symbol (a function identifier or a data	+
24825	object identifier) that has the largest address less than or equal to the specified address.	+
24826	If the address specified by <i>addr</i> is within the mapped address range of one of the examined	+
24827	mapped objects, the structure pointed to by <i>dlip</i> shall be populated as follows:	+
24828	• The value of the <i>dli_fname</i> member shall be set to point to the pathname of the mapped	+
24829	object. (This might no longer resolve to the file that was mapped, for example if it was a	+
24830	link that has subsequently been removed or renamed.)	+
24831	• The value of the <i>dli_fbase</i> member shall be set to the base of the address range occupied by	+
24832	the mapped object.	+
24833	• The value of the <i>dli_sname</i> member shall be set to point to the name of the symbol that has	+
24834	the largest address less than or equal to the specified address, or to a null pointer if no such	+
24835	symbol was found.	+
24836	• If <i>dli_sname</i> is set to a null pointer, the value of the <i>dli_saddr</i> member shall also be set to a	+
24837	null pointer. Otherwise, if <i>dli_sname</i> names a function identifier, <i>dli_saddr</i> shall be set to the	+
24838	address of the function converted from type pointer to function to type pointer to void ;	+
24839	otherwise, <i>dli_saddr</i> shall be set to the address of the data object named by <i>dli_sname</i>	+
24840	converted from a pointer to the type of the data object to a pointer to void .	+
24841	RETURN VALUE	+
24842	Upon successful completion, a non-zero value shall be returned. If the specified address is not	+
24843	located within the address range occupied by an examined mapped object, or if an error occurs,	+
24844	zero shall be returned. More detailed diagnostic information shall be available through <i>dlderror()</i> .	+
24845	ERRORS	+
24846	No errors are defined.	+

24847	EXAMPLES	+
24848	None.	+
24849	APPLICATION USAGE	+
24850	The DI_info_t members may point to addresses within the mapped object. These pointers can	+
24851	become invalid if the object is unmapped (for example, loaded executable objects may be	+
24852	unloaded by <i>dlclose()</i>).	+
24853	If <i>dli_sname</i> names a function identifier, the value of <i>dli_saddr</i> can be converted back to type	+
24854	pointer to function using a cast in the manner shown in the <i>dlsym()</i> EXAMPLES section. Note	+
24855	that this conversion is not defined by the ISO C standard. This standard requires this conversion	+
24856	to work correctly on conforming implementations.	+
24857	RATIONALE	+
24858	None.	+
24859	FUTURE DIRECTIONS	+
24860	None.	+
24861	SEE ALSO	+
24862	<i>dlclose()</i> , <i>dLError()</i> , <i>dlopen()</i> , <i>dlsym()</i>	+
24863	XBD <dlfcn.h>	+
24864	CHANGE HISTORY	+
24865	First released in Issue 8.	+
24866		+

```

25353         assert(xsubi[1] == 10728);
25354         assert(xsubi[2] == 27921);
25355         assert(nrand48(xsubi) == 754104482);
25356         assert(xsubi[0] == 6828);
25357         assert(xsubi[1] == 28997);
25358         assert(xsubi[2] == 23013);
25359         assert(nrand48(xsubi) == 609453945);
25360         assert(xsubi[0] == 58183);
25361         assert(xsubi[1] == 3826);
25362         assert(xsubi[2] == 18599);
25363         assert(nrand48(xsubi) == 1878644360);
25364         assert(xsubi[0] == 36678);
25365         assert(xsubi[1] == 44304);
25366         assert(xsubi[2] == 57331);
25367         assert(nrand48(xsubi) == 2114923686);
25368         assert(xsubi[0] == 58585);
25369         assert(xsubi[1] == 22861);
25370         assert(xsubi[2] == 64542);
25371     }
25372 }

```

25373 APPLICATION USAGE

25374 These functions should be avoided whenever non-trivial requirements (including safety) have to
 25375 be fulfilled, unless seeded using *getentropy()*. |

25376 RATIONALE

25377 None.

25378 FUTURE DIRECTIONS

25379 None.

25380 SEE ALSO

25381 *getentropy()*, *initstate()*, *rand()* +

25382 XBD <stdlib.h>

25383 CHANGE HISTORY

25384 First released in Issue 1. Derived from Issue 1 of the SVID.

25385 Issue 5

25386 A note indicating that the *drand48()*, *lrand48()*, and *mrnd48()* functions need not be reentrant is
 25387 added to the DESCRIPTION.

25388 Issue 6

25389 The normative text is updated to avoid use of the term “must” for application requirements.

25390 Issue 7

25391 Austin Group Interpretation 1003.1-2001 #156 is applied.

25392 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0083 [743] is applied.

25393 Issue 8

25394 Austin Group Defect 1107 is applied, clarifying how the return value is calculated from X_i for
 25395 each function. +

25396 Austin Group Defect 1134 is applied, adding *getentropy()*. |

30202 possible for the system to conform to the intent of this volume of POSIX.1-202x.

30203 The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it
30204 occurs or not is not in any practical sense under the control of the application because the
30205 condition is usually a consequence of the user's use of the system, not of the application's code.
30206 Thus, no application can or should rely upon its occurrence under any circumstances, nor
30207 should the exact semantics of what concept of "user" is used be of concern to the application
30208 developer. Validation writers should be cognizant of this limitation.

30209 There are two reasons why POSIX programmers call *fork()*. One reason is to create a new thread
30210 of control within the same program (which was originally only possible in POSIX by creating a
30211 new process); the other is to create a new process running a different program. In the latter case,
30212 the call to *fork()* is soon followed by a call to one of the *exec* functions.

30213 The general problem with making *fork()* work in a multi-threaded world is what to do with all
30214 of the threads. There are two alternatives. One is to copy all of the threads into the new process.
30215 This causes the programmer or implementation to deal with threads that are suspended on
30216 system calls or that might be about to execute system calls that should not be executed in the
30217 new process. The other alternative is to copy only the thread that calls *fork()*. This creates the
30218 difficulty that the state of process-local resources is usually held in process memory. If a thread
30219 that is not calling *fork()* holds a resource, that resource is never released in the child process
30220 because the thread whose job it is to release the resource does not exist in the child process.

30221 When a programmer is writing a multi-threaded program, the first described use of *fork()*,
30222 creating new threads in the same program, is provided by the *pthread_create()* function. The
30223 *fork()* function is thus used only to run new programs, and the effects of calling functions that
30224 require certain resources between the call to *fork()* and the call to an *exec* function are undefined.

30225 The addition of the *forkall()* function to the standard was considered and rejected. The *forkall()*
30226 function lets all the threads in the parent be duplicated in the child. This essentially duplicates
30227 the state of the parent in the child. This allows threads in the child to continue processing and
30228 allows locks and the state to be preserved without explicit *pthread_atfork()* code. The calling
30229 process has to ensure that the threads processing state that is shared between the parent and
30230 child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall()*. For
30231 example, if a thread is reading a file descriptor in the parent when *forkall()* is called, then two
30232 threads (one in the parent and one in the child) are reading the file descriptor after the *forkall()*.
30233 If this is not desired behavior, the parent process has to synchronize with such threads before
30234 calling *forkall()*.

30235 When *forkall()* is called, threads, other than the calling thread, that are in functions that can
30236 return with an [EINTR] error may have those functions return [EINTR] if the implementation
30237 cannot ensure that the function behaves correctly in the parent and child. In particular,
30238 *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, and *pthread_cond_wait()* need to return in
30239 order to ensure that the condition has not changed. These functions can be awakened by a
30240 spurious condition wakeup rather than returning [EINTR].

30241 **FUTURE DIRECTIONS**

30242 None.

30243 **SEE ALSO**

30244 *alarm()*, *exec*, *fcntl()*, *pthread_atfork()*, *semop()*, *signal()*, *times()*

30245 XBD Section 4.13 (on page 91), [<sys/types.h>](#), [<unistd.h>](#)

31485 **NAME**

31486 free — free allocated memory

31487 **SYNOPSIS**

```
31488 #include <stdlib.h>
31489 void free(void *ptr);
```

31490 **DESCRIPTION**

31491 CX The functionality described on this reference page is aligned with the ISO C standard. Any
 31492 conflict between the requirements described here and the ISO C standard is unintentional. This
 31493 volume of POSIX.1-202x defers to the ISO C standard.

31494 The *free()* function shall cause the space pointed to by *ptr* to be deallocated; that is, made
 31495 available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the
 31496 argument does not match a pointer earlier returned by a function in POSIX.1-202x that allocates
 31497 CX memory as if by *malloc()*, or if the space has been deallocated by a call to *free()*, *realloc()*, or
 31498 *reallocarray()*, the behavior is undefined.

31499 Any use of a pointer that refers to freed space results in undefined behavior.

31500 CX The *free()* function shall not modify *errno* if *ptr* is a null pointer or a pointer previously returned
 31501 as if by *malloc()* and not yet deallocated.

31502 **RETURN VALUE**

31503 The *free()* function shall not return a value.

31504 **ERRORS**

31505 No errors are defined.

31506 **EXAMPLES**

31507 None.

31508 **APPLICATION USAGE**

31509 There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

31510 Because the *free()* function does not modify *errno* for valid pointers, it is safe to use it in cleanup
 31511 code without corrupting earlier errors, such as in this example code:

```
31512 // buf was obtained by malloc(buflen)
31513 ret = write(fd, buf, buflen);
31514 if (ret < 0) {
31515     free(buf);
31516     return ret;
31517 }
```

31518 However, earlier versions of this standard did not require this, and the same example had to be
 31519 written as:

```
31520 // buf was obtained by malloc(buflen)
31521 ret = write(fd, buf, buflen);
31522 if (ret < 0) {
31523     int save = errno;
31524     free(buf);
31525     errno = save;
31526     return ret;
31527 }
```


NAME

34913 getentropy — fill a buffer with random bytes
34914

SYNOPSIS

34915 #include <unistd.h>
34916
34917 int getentropy(void *buffer, size_t length);

DESCRIPTION

34918 The *getentropy()* function shall write *length* bytes of data starting at the location pointed to by
34919 *buffer*. The output shall be unpredictable high quality random data, generated by a
34920 cryptographically secure pseudo-random number generator. The maximum permitted value for
34921 the *length* argument is given by the {GETENTROPY_MAX} symbolic constant defined in
34922 <limits.h>.
34923

34924 A successful call to *getentropy()* shall always provide the requested number of bytes of entropy.

RETURN VALUE

34925 Upon successful completion, *getentropy()* shall return 0; otherwise, -1 shall be returned and
34926 *errno* set to indicate the error.
34927

ERRORS

34928 The *getentropy()* function shall fail if:

34929 [EINVAL] The value of *length* is greater than {GETENTROPY_MAX}.

34930 The *getentropy()* function may fail if:

34931 [ENOSYS] The system does not provide the necessary source of entropy.

EXAMPLES

34932 None.
34933

APPLICATION USAGE

34934 The intended use of this function is to create a seed for other pseudo-random number
34935 generators.
34936

RATIONALE

34937 The *getentropy()* function is not a cancellation point. (See [Section 2.9.5.2](#) (on page 504).)
34938

FUTURE DIRECTIONS

34939 None.
34940

SEE ALSO

34941 [drand48\(\)](#), [initstate\(\)](#), [rand\(\)](#)
34942

34943 XBD [<limits.h>](#), [<unistd.h>](#)
34944

CHANGE HISTORY

34945 First released in Issue 8.
34946
34947

NAME

35541 getlocalename_l — get a locale name from a locale object
 35542

SYNOPSIS

```
35544 CX #include <locale.h>
35545     const char *getlocalename_l(int category, locale_t locobj);
35546
```

DESCRIPTION

35547 The *getlocalename_l()* function shall return the locale name for the given locale category of the
 35548 locale object *locobj*, or of the global locale if *locobj* is the special locale object
 35549 LC_GLOBAL_LOCALE.
 35550

35551 The *category* argument specifies the locale category to be queried. If the value is LC_ALL or is
 35552 not a supported locale category value (see *setlocale()*), *getlocalename_l()* shall fail.

35553 The behavior is undefined if the *locobj* argument is neither the special locale object
 35554 LC_GLOBAL_LOCALE nor a valid locale object handle.

RETURN VALUE

35555 Upon successful completion, *getlocalename_l()* shall return a pointer to a string containing the
 35556 locale name; otherwise, a null pointer shall be returned.
 35557

35558 If *locobj* is LC_GLOBAL_LOCALE, the returned string pointer might be invalidated or the string
 35559 content might be overwritten by a subsequent call in the same thread to *getlocalename_l()* with
 35560 LC_GLOBAL_LOCALE; the returned string pointer might also be invalidated if the calling
 35561 thread is terminated. Otherwise, the returned string pointer and content shall remain valid until
 35562 the locale object *locobj* is used in a call to *freelocale()* or as the *base* argument in a successful call to
 35563 *newlocale()*.

ERRORS

35564 No errors are defined.
 35565

EXAMPLES**Determining the locale name for a category of the current locale**

35567 The following example shows how to obtain the locale name for the LC_NUMERIC category of
 35568 the current thread-local locale, or of the global locale if no thread-local locale is in use.
 35569

```
35570 #include <locale.h>
35571 ...
35572 const char *name;
35573 locale_t loc = uselocale(NULL);
35574 name = getlocalename_l(LC_NUMERIC, loc);
```

APPLICATION USAGE

35575 None.
 35576

RATIONALE

35577 Historical versions of *getlocalename_l()* did not handle the special locale object
 35578 LC_GLOBAL_LOCALE, requiring that applications used *setlocale(category, NULL)* to query the
 35579 global locale if *uselocale(NULL)* returned LC_GLOBAL_LOCALE. However, since *setlocale()* is
 35580 not required to be thread-safe (even when the only concurrent calls are ones that query the
 35581 locale), this method was problematic for multi-threaded processes. This standard requires that
 35582 *getlocalename_l(category, LC_GLOBAL_LOCALE)* queries the global locale in a thread-safe
 35583 manner, for example by returning a pointer to a thread-local internal buffer instead of a process-
 35584 wide internal buffer.
 35585

35586 **FUTURE DIRECTIONS**

35587 None.

35588 **SEE ALSO**35589 *freelocale()*, *newlocale()*, *setlocale()*, *uselocale()*35590 XBD Chapter 7 (on page 113), **<locale.h>**35591 **CHANGE HISTORY**

35592 First released in Issue 8.

35593

Sanity Review

38372 **ERRORS**

38373 No errors are defined.

38374 **EXAMPLES**

38375 None.

38376 **APPLICATION USAGE**

38377 After initialization, a state array can be restarted at a different point in one of two ways:

- 38378 1. The *initstate()* function can be used, with the desired seed, state array, and size of the
38379 array.
- 38380 2. The *setstate()* function, with the desired state, can be used, followed by *srandom()* with
38381 the desired seed. The advantage of using both of these functions is that the size of the
38382 state array does not have to be saved once it is initialized.

38383 Although some implementations of *random()* have written messages to standard error, such
38384 implementations do not conform to POSIX.1-202x.

38385 Issue 5 restored the historical behavior of this function.

38386 Threaded applications should use *erand48()*, *nrand48()*, or *jrand48()* instead of *random()* when
38387 an independent random number sequence in multiple threads is required.38388 These functions should be avoided whenever non-trivial requirements (including safety) have to
38389 be fulfilled, unless seeded using *getentropy()*.38390 **RATIONALE**

38391 None.

38392 **FUTURE DIRECTIONS**

38393 None.

38394 **SEE ALSO**38395 [drand48\(\)](#), [getentropy\(\)](#), [rand\(\)](#) +38396 XBD [<stdlib.h>](#)38397 **CHANGE HISTORY**

38398 First released in Issue 4, Version 2.

38399 **Issue 5**

38400 Moved from X/OPEN UNIX extension to BASE.

38401 In the DESCRIPTION, the phrase “values smaller than 8” is replaced with “values greater than
38402 or equal to 8, or less than 32”, “size<8” is replaced with “8≤size <32”, and a new first paragraph
38403 is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE
38404 indicating that these changes restore the historical behavior of the function.38405 **Issue 6**

38406 In the DESCRIPTION, duplicate text “For values greater than or equal to 8 . . .” is removed.

38407 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/30 is applied, removing *rand_r()* from the
38408 list of suggested functions in the APPLICATION USAGE section.38409 **Issue 7**38410 The type of the first argument to *setstate()* is changed from **const char *** to **char ***.

38411 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0179 [743] is applied. +

NAME

43075 memmem — find a byte subsequence in a byte sequence
43076

SYNOPSIS

```
43078 CX #include <string.h>  
43079 void *memmem(const void *haystack, size_t haystacklen,  
43080             const void *needle, size_t needlelen);  
43081
```

DESCRIPTION

43082 The *memmem()* function shall locate the first occurrence of byte sequence *needle* of length
43083 *needlelen* in byte sequence *haystack* of length *haystacklen*.
43084

RETURN VALUE

43085 Upon successful completion, *memmem()* shall return a pointer to the the first byte of the located
43086 byte sequence in *haystack*, or a null pointer if the byte sequence is not found.
43087

43088 If *needlelen* is zero, the function shall return *haystack*.

43089 If *haystacklen* is less than *needlelen*, the function shall return a null pointer.

ERRORS

43090 No errors are defined.
43091

EXAMPLES

43092 None.
43093

APPLICATION USAGE

43094 None.
43095

RATIONALE

43096 This function is similar to *strstr()*, except that NUL bytes may be included in either *needle* or
43097 *haystack*.
43098

FUTURE DIRECTIONS

43099 None.
43100

SEE ALSO

43101 *memchr()*, *strstr()*

43102 XBD <[string.h](#)>

CHANGE HISTORY

43104 First released in Issue 8.
43105
43106

47102 **NAME**

47103 poll, ppoll — input/output multiplexing +

47104 **SYNOPSIS**

```
47105 #include <poll.h>
47106 int poll(struct pollfd fds[], nfds_t nfds, int timeout);
47107 int ppoll(struct pollfd fds[], nfds_t nfds, +
47108           const struct timespec *restrict timeout, +
47109           const sigset_t *restrict sigmask); +
```

47110 **DESCRIPTION**

47111 The *ppoll()* function provides applications with a mechanism for multiplexing input/output |
 47112 over a set of file descriptors. For each member of the array pointed to by *fds*, *ppoll()* shall |
 47113 examine the given file descriptor for the event(s) specified in *events*. The number of **pollfd** |
 47114 structures in the *fds* array is specified by *nfds*. The *ppoll()* function shall identify those file |
 47115 descriptors on which an application can read or write data, or on which certain events have |
 47116 occurred.

47117 The *poll()* function shall be equivalent to the *ppoll()* function, except as follows: +

- 47118 • For the *poll()* function, the timeout period is given in milliseconds in an argument of type +
 47119 **int**, whereas for the *ppoll()* function the timeout period is given in seconds and +
 47120 nanoseconds via an argument of type pointer to **struct timespec**. A *timeout* of -1 for *poll()* +
 47121 shall be equivalent to passing a null pointer for the *timeout* for *ppoll()*. +
- 47122 • The *poll()* function has no *sigmask* argument; it shall behave as *ppoll()* does when *sigmask* is +
 47123 a null pointer. +

47124 The *fds* argument specifies the file descriptors to be examined and the events of interest for each
 47125 file descriptor. It is a pointer to an array with one member for each open file descriptor of
 47126 interest. The array's members are **pollfd** structures within which *fd* specifies an open file
 47127 descriptor and *events* and *revents* are bitmasks constructed by OR'ing a combination of the
 47128 following event flags:

- | | | |
|-------|------------|--|
| 47129 | POLLIN | Data other than high-priority data may be read without blocking. |
| 47130 | POLLRDNORM | Normal data may be read without blocking. |
| 47131 | POLLRDBAND | Priority data may be read without blocking. |
| 47132 | POLLPRI | High-priority data may be read without blocking. |
| 47133 | POLLOUT | Normal data may be written without blocking. |
| 47134 | POLLWRNORM | Equivalent to POLLOUT. |
| 47135 | POLLWRBAND | Priority data may be written. |
| 47136 | POLLERR | An error has occurred on the device or stream. This flag is only valid in the |
| 47137 | | <i>revents</i> bitmask; it shall be ignored in the <i>events</i> member. |
| 47138 | POLLHUP | A device has been disconnected, or a pipe or FIFO has been closed by the last |
| 47139 | | process that had it open for writing. Once set, the hangup state of a FIFO shall |
| 47140 | | persist until some process opens the FIFO for writing or until all read-only file |
| 47141 | | descriptors for the FIFO are closed. This event and POLLOUT are mutually- |
| 47142 | | exclusive; a stream can never be writable if a hangup has occurred. However, |
| 47143 | | this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are |
| 47144 | | not mutually-exclusive. This flag is only valid in the <i>revents</i> bitmask; it shall be |
| 47145 | | ignored in the <i>events</i> member. |

47146 POLLNVAL The specified *fd* value is invalid. This flag is only valid in the *revents* member;
47147 it shall ignored in the *events* member.

47148 The significance and semantics of normal, priority, and high-priority data are file and device-
47149 specific.

47150 If the value of *fd* is less than 0, *events* shall be ignored, and *revents* shall be set to 0 in that entry on
47151 return from *poll()* or *ppoll()*.

47152 In each **pollfd** structure, *poll()* or *ppoll()* shall clear the *revents* member, except that where the +
47153 application requested a report on a condition by setting one of the bits of *events* listed above, +
47154 *poll()* or *ppoll()* shall set the corresponding bit in *revents* if the requested condition is true. In +
47155 addition, *poll()* or *ppoll()* shall set the POLLHUP, POLLERR, and POLLNVAL flag in *revents* if +
47156 the condition is true, even if the application did not set the corresponding bit in *events*.

47157 The *timeout* argument controls how long the *poll()* or *ppoll()* function shall wait before timing |
47158 out. If the *timeout* argument is positive for *poll()* or not a null pointer for *ppoll()*, it specifies a |
47159 maximum interval to wait for the poll to complete. If the specified time interval expires without |
47160 any of the defined events having occurred, the function shall return. If the *timeout* argument is |
47161 -1 for *poll()* or a null pointer for *ppoll()*, then the call shall block indefinitely until at least one |
47162 descriptor meets the specified criteria or until the call is interrupted. To effect a poll, the |
47163 application shall ensure that the *timeout* argument for *poll()* is 0, or for *ppoll()* is not a null |
47164 pointer and points to a zero-valued **timespec** structure.

47165 Implementations may place limitations on the maximum timeout interval supported. All |
47166 implementations shall support a maximum timeout interval of at least 31 days for *ppoll()*. If the |
47167 *timeout* argument specifies a timeout interval greater than the implementation-defined |
47168 maximum value, the maximum value shall be used as the actual timeout value. Implementations |
47169 may also place limitations on the granularity of timeout intervals. If the requested timeout |
47170 interval requires a finer granularity than the implementation supports, the actual timeout |
47171 interval shall be rounded up to the next supported value.

47172 The *poll()* and *ppoll()* functions shall not be affected by the O_NONBLOCK flag. |

47173 The *poll()* and *ppoll()* functions shall support regular files, terminal and pseudo-terminal |
47174 devices, FIFOs, pipes, and sockets. The behavior of *poll()* and *ppoll()* on elements of *fds* that refer +
47175 to other types of file is unspecified.

47176 Regular files shall always poll TRUE for reading and writing.

47177 A file descriptor for a socket that is listening for connections shall indicate that it is ready for
47178 reading, once connections are available. A file descriptor for a socket that is connecting
47179 asynchronously shall indicate that it is ready for writing, once a connection has been established.

47180 Provided the application does not perform any action that results in unspecified or undefined
47181 behavior, the value of the *fd* and *events* members of each element of *fds* shall not be modified by -
47182 *poll()* or *ppoll()*.

47183 If *sigmask* is not a null pointer, the *ppoll()* function shall replace the signal mask of the caller by |
47184 the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the |
47185 signal mask of the calling thread before returning. If a signal is unmasked as a result of the |
47186 signal mask being altered by *ppoll()*, and a signal-catching function is called for that signal |
47187 during the execution of the *ppoll()* function, and SA_RESTART is clear for the interrupting |
47188 signal, then

- 47189 • If none of the defined events have occurred on any selected file descriptor, *ppoll()* shall |
47190 immediately fail with the [EINTR] error after the signal-catching function returns. |

47191 • If one or more of the defined events have occurred, it is unspecified whether *ppoll()* |
 47192 behaves the same as if none of the events had occurred (failing with [EINTR] as above) or |
 47193 behaves the same as if it was not interrupted (returning the total number of **pollfd** |
 47194 structures that have selected events). |

47195 If a thread is canceled during a *ppoll()* call, it is unspecified whether the signal mask in effect |
 47196 when executing the registered cleanup functions is the original signal mask or the signal mask |
 47197 installed as part of the *ppoll()* call. |

47198 RETURN VALUE

47199 Upon successful completion, a non-negative value shall be returned. A positive value shall |
 47200 indicate the total number of **pollfd** structures that have selected events (that is, those for which |
 47201 the *revents* member is non-zero). A value of 0 shall indicate that the call timed out and no file |
 47202 descriptors have been selected. Upon failure, -1 shall be returned and *errno* set to indicate the |
 47203 error. |

47204 ERRORS

47205 The *poll()* and *ppoll()* functions shall fail if: |

47206 [EAGAIN] The allocation of internal data structures failed but a subsequent request may |
 47207 succeed. |

47208 [EINTR] A signal was caught during *poll()* or *ppoll()*. |

47209 [EINVAL] The *nfds* argument is greater than {OPEN_MAX}. |

47210 The *ppoll()* function shall fail if: +

47211 [EINVAL] An invalid timeout interval was specified. +

47212 EXAMPLES

47213 None.

47214 APPLICATION USAGE

47215 Other than the difference in the precision of the requested timeout, the following *ppoll()* call: |

```
47216 ready = ppoll(&fds, nfds, tmo_p, &sigmask); |
```

47217 is equivalent to atomically executing the following calls: |

```
47218 sigset_t origmask; |
```

```
47219 int timeout; |
```

```
47220 timeout = (tmo_p == NULL) ? -1 : |
```

```
47221 (tmo_p->tv_sec * 1000 + tmo_p->tv_nsec / 1000000); |
```

```
47222 pthread_sigmask(SIG_SETMASK, &sigmask, &origmask); |
```

```
47223 ready = poll(&fds, nfds, timeout); |
```

```
47224 pthread_sigmask(SIG_SETMASK, &origmask, NULL); |
```

47225 RATIONALE

47226 The POLLHUP event does not occur for FIFOs just because the FIFO is not open for writing. It |
 47227 only occurs when the FIFO is closed by the last writer and persists until some process opens the |
 47228 FIFO for writing or until all read-only file descriptors for the FIFO are closed. +

47229 Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers +
 47230 can install an additional cancellation handler which resets the signal mask to the expected value: +

```
47231 void cleanup(void *arg) +
```

```
47232 { +
```

```
47233     sigset_t *ss = (sigset_t *) arg; +
```



```

47234     pthread_sigmask(SIG_SETMASK, ss, NULL);           +
47235     }                                                  +
47236     int call_ppoll(struct pollfd fds[], nfds_t nfds,    +
47237     const struct timespec *restrict timeout,          +
47238     const sigset_t *restrict sigmask)                +
47239     {                                                  +
47240     sigset_t oldmask;                                  +
47241     int result;                                        +
47242     pthread_sigmask(SIG_SETMASK, NULL, &oldmask);     +
47243     pthread_cleanup_push(cleanup, &oldmask);         +
47244     result = ppoll(fds, nfds, timeout, sigmask);      +
47245     pthread_cleanup_pop(0);                           +
47246     return result;                                    +
47247     }                                                  +

```

47248 FUTURE DIRECTIONS

47249 None.

47250 SEE ALSO

47251 *pselect()*, *read()*, *write()*

47252 XBD <poll.h>

47253 CHANGE HISTORY

47254 First released in Issue 4, Version 2.

47255 Issue 5

47256 Moved from X/OPEN UNIX extension to BASE.

47257 The description of POLLWRBAND is updated.

47258 Issue 6

47259 Text referring to sockets is added to the DESCRIPTION.

47260 Functionality relating to the XSI STREAMS Option Group is marked.

47261 The Open Group Corrigendum U055/3 is applied, updating the DESCRIPTION of
47262 POLLWRBAND.

47263 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/66 is applied, correcting the spacing in
47264 the EXAMPLES section.

47265 Issue 7

47266 Austin Group Interpretation 1003.1-2001 #209 is applied, clarifying the POLLHUP event.

47267 The *poll()* function is moved from the XSI option to the Base.

47268 Functionality relating to the XSI STREAMS option is marked obsolescent.

47269 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0249 [623] and XSH/TC2-2008/0250
47270 [683] are applied.

47271 Issue 8

47272 Austin Group Defect 1263 is applied, adding *ppoll()*. +

47273 Austin Group Defect 1330 is applied, removing obsolescent interfaces. |

NAME

47780 posix_getdents — read directory entries

47781 SYNOPSIS

47782 #include <dirent.h>
47783
47784 ssize_t posix_getdents(int *fildes*, void **buf*, size_t *nbyte*, int *flags*);

47785 DESCRIPTION

47786 The *posix_getdents()* function shall attempt to read directory entries from the directory associated
47787 with the open file descriptor *fildes* and shall place information about the directory entries and the
47788 files they refer to in **posix_dent** structures in the buffer pointed to by *buf*, up to a maximum of
47789 *nbyte* bytes. The number of **posix_dent** structures populated in *buf* may be fewer than the
47790 number that will fit in *nbyte* bytes, but shall be at least one if *nbyte* is greater than the size of the
47791 **posix_dent** structure plus [NAME_MAX] and *fildes* is not currently at end-of-file.

47792 The application shall ensure that *buf* is aligned suitably to point to a **posix_dent** structure. The
47793 alignment needed shall not be more restrictive than the alignment provided by *malloc()*. Strictly
47794 conforming applications shall ensure that the value of *flags* is zero; other applications can set it to
47795 a value constructed by a bitwise-inclusive OR of implementation-defined bitwise-distinct flag
47796 values.

47797 Each **posix_dent** structure returned in *buf* shall be located at an address that satisfies the
47798 implementation's alignment requirements for the **posix_dent** structure and shall be populated
47799 as follows:

- 47800 • The value of the *d_ino* member shall be set to the file serial number of the file named by the
47801 *d_name* member.
- 47802 • The value of the *d_reclen* member shall be set to the number of bytes occupied by this entry
47803 in *buf*, including any padding bytes needed before the next entry, if any. If this is the last
47804 entry in *buf*, *d_reclen* shall include any padding bytes needed to make the address of this
47805 entry plus *d_reclen* bytes satisfy the alignment requirements for the **posix_dent** structure.
- 47806 • The value of the *d_type* member shall be set to indicate the file type of the named file, if the
47807 file type can be determined without needing to use the file serial number to obtain the
47808 file's metadata; otherwise it may be set to DT_UNKNOWN. If the file type is determined
47809 and it is one of the file types defined in this standard, the value of *d_type* shall be DT_BLK,
47810 DT_CHR, DT_DIR, DT_FIFO, DT_LNK, DT_REG, DT_SOCKET, DT_MQ, DT_SEM,
47811 TYP DT_SHM, or DT_TMO (see <dirent.h>). If it is determined but is not a standard file type,
47812 the value of *d_type* shall not equal any of those listed here.
- 47813 • The *d_name* member shall be a filename string, and (if not dot or dot-dot) shall contain the
47814 same byte sequence as the last pathname component of the string used to create the
47815 directory entry, plus the terminating NUL byte.

47816 If the *d_name* member names a symbolic link, the values of the *d_ino* and *d_type* members shall
47817 be set to the values for the symbolic link itself.

47818 The *posix_getdents()* function shall start reading at the current file offset in the open file
47819 description associated with *fildes*. On successful return, the file offset shall be incremented to
47820 point to the directory entry immediately following the last entry whose information was
47821 returned in *buf*, or to point to end-of-file if there are no more directory entries. On failure, the
47822 value of the file offset is unspecified. The current file offset can be set and retrieved using *lseek()*
47823 on the open file description associated with *fildes*. The behavior is unspecified if *lseek()* is used
47824 to set the file offset to a value other than zero or a value returned by a previous call to *lseek()* on
47825 the same open file description.

47826 The `posix_getdents()` function shall not return directory entries containing empty names. If
 47827 entries for dot or dot-dot exist, a sequence of calls that reads from offset zero to end-of-file shall
 47828 return one entry for dot and one entry for dot-dot; otherwise, they shall not be returned.

47829 Upon successful completion, `posix_getdents()` shall mark for update the last data access
 47830 timestamp of the directory.

47831 If `fildev` is a file descriptor associated with a directory stream opened using `fdopendir()` or
 47832 `opendir()`, the behavior is unspecified.

47833 If `posix_getdents()` is called concurrently with an operation that adds, deletes, or modifies a
 47834 directory entry, the results from `posix_getdents()` shall reflect either all of the effects of the
 47835 concurrent operation or none of them. If a sequence of calls to `posix_getdents()` is made that reads
 47836 from offset zero to end-of-file and a file is removed from or added to the directory between the
 47837 first and last of those calls, whether the sequence of calls returns an entry for that file is
 47838 unspecified.

47839 RETURN VALUE

47840 Upon successful completion, either a non-negative integer shall be returned indicating the
 47841 number of bytes occupied by the `posix_dent` structures placed in `buf` or 0 shall be returned
 47842 indicating the end of the directory was reached without any directory entries being placed in `buf`.
 47843 Otherwise, `-1` shall be returned and `errno` shall be set to indicate the error.

47844 ERRORS

47845 The `posix_getdents()` function shall fail if:

- | | | |
|-------|-------------|---|
| 47846 | [EBADF] | The <code>fildev</code> argument is not a valid file descriptor open for reading. |
| 47847 | [EINVAL] | The <code>nbyte</code> argument is not large enough to contain the information to be returned about the directory entry located at the current file offset. |
| 47848 | | |
| 47849 | [ENOENT] | The current file offset is not located at a valid directory entry. |
| 47850 | [ENOTDIR] | The <code>fildev</code> argument is associated with a non-directory file. |
| 47851 | [EOVERFLOW] | One of the values in a structure to be placed in <code>buf</code> cannot be represented correctly. |
| 47852 | | |

47853 The `posix_getdents()` function may fail if:

- | | | |
|-------|----------|---|
| 47854 | [EIO] | A physical I/O error has occurred. |
| 47855 | [ENOMEM] | Insufficient memory was available to fulfill the request. |

47856 EXAMPLES

47857 This example function lists the files in a specified directory with their file serial number and file
 47858 type. If the file type is not available from `posix_getdents()`, it is obtained using `fstatat()`.

```
47859 #include <dirent.h>
47860 #include <fcntl.h>
47861 #include <stdio.h>
47862 #include <stdlib.h>
47863 #include <sys/stat.h>
47864 #include <unistd.h>
47865 #define ENTBUFFSIZ 10240
47866 int list_dir(const char *dirnam)
47867 {
47868     int fd = open(dirnam, O_RDONLY | O_DIRECTORY);
```

```

47869     if (fd == -1)
47870         return -1;

47871     char *buf = malloc(ENTBUFSIZ);
47872     if (buf == NULL)
47873     {
47874         close(fd);
47875         return -1;
47876     }

47877     ssize_t bytesinbuf;
47878     for(;;)
47879     {
47880         ssize_t nextent = 0;

47881         bytesinbuf = posix_getdents(fd, buf, ENTBUFSIZ, 0);
47882         if (bytesinbuf <= 0)
47883             break;

47884         do {
47885             const char *ftype;
47886             struct posix_dent *entp = (void *)&buf[nextent];
47887             if (entp->d_type == DT_UNKNOWN)
47888             {
47889                 struct stat stbuf;
47890                 if (fstatat(fd, entp->d_name, &stbuf,
47891                     AT_SYMLINK_NOFOLLOW) == -1)
47892                     ftype = "?";
47893                 else
47894                     ftype = S_ISBLK(stbuf.st_mode) ? "b" :
47895                             S_ISCHR(stbuf.st_mode) ? "c" :
47896                             S_ISDIR(stbuf.st_mode) ? "d" :
47897                             S_ISFIFO(stbuf.st_mode) ? "p" :
47898                             S_ISLNK(stbuf.st_mode) ? "l" :
47899                             S_ISREG(stbuf.st_mode) ? "r" :
47900                             S_ISSOCK(stbuf.st_mode) ? "s" :
47901                             S_TYPEISMQ(&stbuf) ? "mq" :
47902                             S_TYPEISSEM(&stbuf) ? "sem" :
47903                             S_TYPEISSHM(&stbuf) ? "shm" :
47904                             #ifdef S_TYPEISTMO
47905                                 S_TYPEISTMO(&stbuf) ? "tmo" :
47906                             #endif
47907                                 "?";
47908             }
47909             else
47910             {
47911                 ftype = entp->d_type == DT_BLK ? "b" :
47912                         entp->d_type == DT_CHR ? "c" :
47913                         entp->d_type == DT_DIR ? "d" :
47914                         entp->d_type == DT_FIFO ? "p" :
47915                         entp->d_type == DT_LNK ? "l" :
47916                         entp->d_type == DT_REG ? "r" :
47917                         entp->d_type == DT_SOCKET ? "s" :
47918                         entp->d_type == DT_MQ ? "mq" :

```

```

47919             entp->d_type == DT_SEM ? "sem" :
47920             entp->d_type == DT_SHM ? "shm" :
47921     #ifdef DT_TMO
47922             entp->d_type == DT_TMO ? "tmo" :
47923     #endif
47924             "?";
47925     }
47926     printf("%ld\t%s\t%s\n", (long)entp->d_ino, ftype,
47927           entp->d_name);
47928     nextent += entp->d_reclen;
47929     } while (nextent < bytesinbuf);
47930 }
47931 close(fd);
47932 free(buf);
47933 return bytesinbuf;
47934 }

```

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:

```

47942 #define DESIREDSIZE 10240
47943 struct posix_dent buf[DESIREDSIZE / sizeof(struct posix_dent) + 1];
47944 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.

47964 **RATIONALE**

47965 The *posix_getdents()* function was derived from existing *getdents()* functions but the name was
47966 changed because the existing *getdents()* functions differed in various ways, in particular the type
47967 of the second argument (structure pointer or **void ***), the members of the populated structures,
47968 and the error numbers used for some conditions. The name change also provided an
47969 opportunity to add a *flags* argument to provide for future extensibility.

47970 Implementations are encouraged to include support for a DT_FORCE_TYPE flag which, when
47971 that bit is set in *flags*, causes *posix_getdents()* to look up the file type if it can not be obtained from
47972 the directory entry. This will allow applications that need to know the file type of every directory
47973 entry to keep the cost of these lookups to the minimum needed to obtain the type at the file
47974 system level, without the additional overhead of making a call to *fstatat()* for every file (that has
47975 *d_type* equal to DT_UNKNOWN).

47976 Some existing *getdents()* or similar functions return directory entry structures for deleted
47977 directory entries in *buf*, marked with a special value of one of the structure members to
47978 distinguish them from non-deleted entries. This behavior is not allowed for *posix_getdents()*,
47979 although the data from a deleted directory entry may be present in *buf* in the form of extra
47980 padding on the end of the previous entry.

47981 **FUTURE DIRECTIONS**

47982 A future version of this standard may add a DT_FORCE_TYPE flag as described in
47983 RATIONALE.

47984 **SEE ALSO**

47985 *fdopendir()*, *fstatat()*, *lseek()*, *readdir()*

47986 XBD <**dirent.h**>

47987 **CHANGE HISTORY**

47988 First released in Issue 8.

47989

NAME

ppoll — input/output multiplexing

49675

SYNOPSIS

49676

#include <poll.h>

49678

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

49679

49680

DESCRIPTION

49681

Refer to *poll()*.

49682

Sanity Review

49836 of the *pselect()* call.

49837 RETURN VALUE

49838 Upon successful completion, the *pselect()* and *select()* functions shall return the total number of
49839 bits set in the bit masks. Otherwise, -1 shall be returned, and *errno* shall be set to indicate the
49840 error.

49841 *FD_CLR()*, *FD_SET()*, and *FD_ZERO()* do not return a value. *FD_ISSET()* shall return a non-
49842 zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and
49843 0 otherwise.

49844 ERRORS

49845 Under the following conditions, *pselect()* and *select()* shall fail and set *errno* to:

49846 [EBADF] One or more of the file descriptor sets specified a file descriptor that is not a
49847 valid open file descriptor.

49848 [EINTR] The function was interrupted by a signal.

49849 If SA_RESTART has been set for the interrupting signal, it is implementation-
49850 defined whether the function restarts or returns with [EINTR].

49851 [EINVAL] An invalid timeout interval was specified.

49852 [EINVAL] The *nfds* argument is less than 0 or greater than FD_SETSIZE.

49853 EXAMPLES

49854 None.

49855 APPLICATION USAGE

49856 The use of *select()* and *pselect()* requires that the application construct the set of file descriptors
49857 to work on each time through a polling loop, and is inherently limited from operating on file
49858 descriptors larger than FD_SETSIZE. Also, the amount of work to perform scales as *nfds*
49859 increases, even if the number of file descriptors selected within the larger set remains the same.
49860 Thus, applications may wish to consider using *poll()* and *ppoll()* instead, for better scaling.

49861 RATIONALE

49862 In earlier versions of the Single UNIX Specification, the *select()* function was defined in the
49863 `<sys/time.h>` header. This is now changed to `<sys/select.h>`. The rationale for this change was
49864 as follows: the introduction of the *pselect()* function included the `<sys/select.h>` header and the
49865 `<sys/select.h>` header defines all the related definitions for the *pselect()* and *select()* functions.
49866 Backwards-compatibility to existing XSI implementations is handled by allowing `<sys/time.h>`
49867 to include `<sys/select.h>`.

49868 Code which wants to avoid the ambiguity of the signal mask for thread cancellation handlers
49869 can install an additional cancellation handler which resets the signal mask to the expected value.

```
49870 void cleanup(void *arg)
49871 {
49872     sigset_t *ss = (sigset_t *) arg;
49873     pthread_sigmask(SIG_SETMASK, ss, NULL);
49874 }
49875 int call_pselect(int nfds, fd_set *readfds, fd_set *writefds,
49876                 fd_set errorfds, const struct timespec *timeout,
49877                 const sigset_t *sigmask)
49878 {
49879     sigset_t oldmask;
49880     int result;
```


51397 **NAME**

51398 pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

51399 **SYNOPSIS**

51400 #include <pthread.h>

51401 int pthread_cond_broadcast(pthread_cond_t *cond);

51402 int pthread_cond_signal(pthread_cond_t *cond);

51403 **DESCRIPTION**

51404 These functions shall unblock threads blocked on a condition variable.

51405 The *pthread_cond_broadcast()* function shall unblock all threads currently blocked on the
51406 specified condition variable *cond*.51407 The *pthread_cond_signal()* function shall unblock at least one of the threads that are blocked on
51408 the specified condition variable *cond* (if any threads are blocked on *cond*).51409 If more than one thread is blocked on a condition variable, the scheduling policy shall determine
51410 the order in which threads are unblocked. When each thread unblocked as a result of a
51411 *pthread_cond_broadcast()* or *pthread_cond_signal()* returns from its call to *pthread_cond_clockwait()*,
51412 *pthread_cond_timedwait()*, or *pthread_cond_wait()*, the thread shall own the mutex with which it
51413 called *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or *pthread_cond_wait()*. The thread(s)
51414 that are unblocked shall contend for the mutex according to the scheduling policy (if applicable),
51415 and as if each had called *pthread_mutex_lock()*.51416 The *pthread_cond_broadcast()* or *pthread_cond_signal()* functions may be called by a thread
51417 whether or not it currently owns the mutex that threads calling *pthread_cond_clockwait()*,
51418 *pthread_cond_timedwait()*, or *pthread_cond_wait()* have associated with the condition variable
51419 during their waits; however, if predictable scheduling behavior is required, then that mutex shall
51420 be locked by the thread calling *pthread_cond_broadcast()* or *pthread_cond_signal()*.51421 The *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall have no effect if there are
51422 no threads currently blocked on *cond*.51423 The behavior is undefined if the value specified by the *cond* argument to *pthread_cond_broadcast()*
51424 or *pthread_cond_signal()* does not refer to an initialized condition variable.51425 **RETURN VALUE**51426 If successful, the *pthread_cond_broadcast()* and *pthread_cond_signal()* functions shall return zero;
51427 otherwise, an error number shall be returned to indicate the error.51428 **ERRORS**

51429 These functions shall not return an error code of [EINTR].

51430 **EXAMPLES**

51431 None.

51432 **APPLICATION USAGE**51433 The *pthread_cond_broadcast()* function is used whenever the shared-variable state has been
51434 changed in a way that more than one thread can proceed with its task. Consider a single
51435 producer/multiple consumer problem, where the producer can insert multiple items on a list
51436 that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast()*
51437 function, the producer would notify all consumers that might be waiting, and thereby the
51438 application would receive more throughput on a multi-processor. In addition,
51439 *pthread_cond_broadcast()* makes it easier to implement a read-write lock. The
51440 *pthread_cond_broadcast()* function is needed in order to wake up all waiting readers when a
51441 writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function
51442 to notify all clients of an impending transaction commit.

51443 It is not safe to use the *pthread_cond_signal()* function in a signal handler that is invoked
 51444 asynchronously. Even if it were safe, there would still be a race between the test of the Boolean
 51445 *pthread_cond_wait()* that could not be efficiently eliminated.

51446 Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling
 51447 from code running in a signal handler.

51448 RATIONALE

51449 If an implementation detects that the value specified by the *cond* argument to
 51450 *pthread_cond_broadcast()* or *pthread_cond_signal()* does not refer to an initialized condition
 51451 variable, it is recommended that the function should fail and report an [EINVAL] error.

51452 Multiple Awakenings by Condition Signal

51453 On a multi-processor, it may be impossible for an implementation of *pthread_cond_signal()* to
 51454 avoid the unblocking of more than one thread blocked on a condition variable. For example,
 51455 consider the following partial implementation of *pthread_cond_wait()* and *pthread_cond_signal()*,
 51456 executed by two threads in the order given. One thread is trying to wait on the condition
 51457 variable, another is concurrently executing *pthread_cond_signal()*, while a third thread is already
 51458 waiting.

```
51459 pthread_cond_wait(mutex, cond):
51460     value = cond->value; /* 1 */
51461     pthread_mutex_unlock(mutex); /* 2 */
51462     pthread_mutex_lock(cond->mutex); /* 10 */
51463     if (value == cond->value) { /* 11 */
51464         me->next_cond = cond->waiter;
51465         cond->waiter = me;
51466         pthread_mutex_unlock(cond->mutex);
51467         unable_to_run(me);
51468     } else
51469         pthread_mutex_unlock(cond->mutex); /* 12 */
51470     pthread_mutex_lock(mutex); /* 13 */

51471 pthread_cond_signal(cond):
51472     pthread_mutex_lock(cond->mutex); /* 3 */
51473     cond->value++; /* 4 */
51474     if (cond->waiter) { /* 5 */
51475         sleeper = cond->waiter; /* 6 */
51476         cond->waiter = sleeper->next_cond; /* 7 */
51477         able_to_run(sleeper); /* 8 */
51478     }
51479     pthread_mutex_unlock(cond->mutex); /* 9 */
```

51480 The effect is that more than one thread can return from its call to *pthread_cond_clockwait()*,
 51481 *pthread_cond_timedwait()*, or *pthread_cond_wait()* as a result of one call to *pthread_cond_signal()*.
 51482 This effect is called “spurious wakeup”. Note that the situation is self-correcting in that the
 51483 number of threads that are so awakened is finite; for example, the next thread to call
 51484 *pthread_cond_wait()* after the sequence of events above blocks.

51485 While this problem could be resolved, the loss of efficiency for a fringe condition that occurs
 51486 only rarely is unacceptable, especially given that one has to check the predicate associated with a
 51487 condition variable anyway. Correcting this problem would unnecessarily reduce the degree of
 51488 concurrency in this basic building block for all higher-level synchronization operations.

51489 An added benefit of allowing spurious wakeups is that applications are forced to code a

51637 **NAME**

51638 pthread_cond_clockwait, pthread_cond_timedwait, pthread_cond_wait — wait on a condition +

51639 **SYNOPSIS**

51640 #include <pthread.h>

51641 int pthread_cond_clockwait(pthread_cond_t *restrict cond, +

51642 pthread_mutex_t *restrict mutex, clockid_t clock_id, +

51643 const struct timespec *restrict abstime); +

51644 int pthread_cond_timedwait(pthread_cond_t *restrict cond,

51645 pthread_mutex_t *restrict mutex,

51646 const struct timespec *restrict abstime);

51647 int pthread_cond_wait(pthread_cond_t *restrict cond,

51648 pthread_mutex_t *restrict mutex);

51649 **DESCRIPTION**

51650 The *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, and *pthread_cond_wait()* functions shall |
 51651 block on a condition variable. The application shall ensure that these functions are called with |
 51652 *mutex* locked by the calling thread; otherwise, an error (for PTHREAD_MUTEX_ERRORCHECK |
 51653 and robust mutexes) or undefined behavior (for other mutexes) results.

51654 These functions atomically release *mutex* and cause the calling thread to block on the condition |
 51655 variable *cond*; atomically here means “atomically with respect to access by another thread to the |
 51656 mutex and then the condition variable”. That is, if another thread is able to acquire the mutex |
 51657 after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast()* |
 51658 or *pthread_cond_signal()* in that thread shall behave as if it were issued after the about-to-block |
 51659 thread has blocked.

51660 Upon successful return, the mutex shall have been locked and shall be owned by the calling |
 51661 thread.

51662 If *mutex* is a robust mutex where an owner terminated while holding the lock and the state is |
 51663 recoverable, the mutex shall be acquired even though the function returns [EOWNERDEAD].

51664 When using condition variables there is always a Boolean predicate involving shared variables |
 51665 associated with each condition wait that is true if the thread should proceed. Spurious wakeups |
 51666 from the *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or *pthread_cond_wait()* functions |
 51667 may occur. Since the return from *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or |
 51668 *pthread_cond_wait()* does not imply anything about the value of this predicate, the predicate |
 51669 should be re-evaluated upon such return.

51670 When a thread waits on a condition variable, having specified a particular mutex to the |
 51671 *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or *pthread_cond_wait()* operation, a dynamic |
 51672 binding is formed between that mutex and condition variable that remains in effect as long as at |
 51673 least one thread is blocked on the condition variable. During this time, the effect of an attempt |
 51674 by any thread to wait on that condition variable using a different mutex is undefined. Once all |
 51675 waiting threads have been unblocked (as by the *pthread_cond_broadcast()* operation), the next |
 51676 wait operation on that condition variable shall form a new dynamic binding with the mutex |
 51677 specified by that wait operation. Even though the dynamic binding between condition variable |
 51678 and mutex may be removed or replaced between the time a thread is unblocked from a wait on |
 51679 the condition variable and the time that it returns to the caller or begins cancellation cleanup, |
 51680 the unblocked thread shall always re-acquire the mutex specified in the condition wait operation |
 51681 call from which it is returning.

51682 A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a |
 51683 thread is set to PTHREAD_CANCEL_DEFERRED, a side-effect of acting upon a cancellation |
 51684 request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first

51685 cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up
 51686 to the point of returning from the call to *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or
 51687 *pthread_cond_wait()*, but at that point notices the cancellation request and, instead of returning to
 51688 the caller, starts the thread cancellation activities, which includes calling cancellation cleanup
 51689 handlers.

51690 A thread that has been unblocked because it has been canceled while blocked in a call to
 51691 *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or *pthread_cond_wait()* shall not consume any
 51692 condition signal that may be directed concurrently at the condition variable if there are other
 51693 threads blocked on the condition variable.

51694 The *pthread_cond_timedwait()* function shall be equivalent to *pthread_cond_wait()*, except that an
 51695 error is returned if the absolute time specified by *abstime* passes (that is, system time equals or
 51696 exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time
 51697 specified by *abstime* has already been passed at the time of the call. When such timeouts occur,
 51698 *pthread_cond_timedwait()* shall nonetheless release and re-acquire the mutex referenced by *mutex*,
 51699 and may consume a condition signal directed concurrently at the condition variable.

51700 The condition variable shall have a clock attribute which specifies the clock that shall be used by
 51701 *pthread_cond_timedwait()* to measure the time specified by the *abstime* argument. The
 51702 *pthread_cond_timedwait()* function is also a cancellation point.

51703 The *pthread_cond_clockwait()* function shall be equivalent to *pthread_cond_timedwait()*, except that +
 51704 the absolute time specified by *abstime* is measured against the clock indicated by *clock_id* rather +
 51705 than the clock specified in the condition variable's clock attribute. Implementations shall +
 51706 support passing CLOCK_REALTIME and CLOCK_MONOTONIC to *pthread_cond_clockwait()* as +
 51707 the *clock_id* argument. +

51708 If a signal is delivered to a thread waiting for a condition variable, upon return from the signal
 51709 handler the thread resumes waiting for the condition variable as if it was not interrupted, or it
 51710 shall return zero due to spurious wakeup.

51711 The behavior is undefined if the value specified by the *cond* or *mutex* argument to these
 51712 functions does not refer to an initialized condition variable or an initialized mutex object,
 51713 respectively.

51714 RETURN VALUE

51715 Except for [ETIMEDOUT], [ENOTRECOVERABLE], and [EOWNERDEAD], all these error
 51716 checks shall act as if they were performed immediately at the beginning of processing for the
 51717 function and shall cause an error return, in effect, prior to modifying the state of the mutex
 51718 specified by *mutex* or the condition variable specified by *cond*.

51719 Upon successful completion, a value of zero shall be returned; otherwise, an error number shall
 51720 be returned to indicate the error.

51721 ERRORS

51722 These functions shall fail if:

51723 [EAGAIN] The mutex is a robust mutex and the system resources available for robust
 51724 mutexes owned would be exceeded.

51725 [ENOTRECOVERABLE]
 51726 The state protected by the mutex is not recoverable.

51727 [EOWNERDEAD]
 51728 The mutex is a robust mutex and the process containing the previous owning
 51729 thread terminated while holding the mutex lock. The mutex lock shall be
 51730 acquired by the calling thread and it is up to the new owner to make the state

51731 consistent.

51732 [EPERM] The mutex type is PTHREAD_MUTEX_ERRORCHECK or the mutex is a
51733 robust mutex, and the current thread does not own the mutex.

51734 The `pthread_cond_clockwait()` and `pthread_cond_timedwait()` functions shall fail if:

51735 [ETIMEDOUT] The time specified by `abstime` has passed.

51736 [EINVAL] The `abstime` argument specified a nanosecond value less than zero or greater
51737 than or equal to 1000 million, or the `clock_id` argument passed to
51738 `pthread_cond_clockwait()` is invalid or not supported.

51739 These functions may fail if:

51740 [EOWNERDEAD]

51741 The mutex is a robust mutex and the previous owning thread terminated
51742 while holding the mutex lock. The mutex lock shall be acquired by the calling
51743 thread and it is up to the new owner to make the state consistent.

51744 These functions shall not return an error code of [EINTR].

51745 EXAMPLES

51746 None.

51747 APPLICATION USAGE

51748 Applications that have assumed that non-zero return values are errors will need updating for
51749 use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting
51750 a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error
51751 returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If
51752 an application is supposed to work with normal and robust mutexes, it should check all return
51753 values for error conditions and if necessary take appropriate action.

51754 RATIONALE

51755 If an implementation detects that the value specified by the `cond` argument to
51756 `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()` does not refer to an
51757 initialized condition variable, or detects that the value specified by the `mutex` argument does not
51758 refer to an initialized mutex object, it is recommended that the function should fail and report an
51759 [EINVAL] error.

51760 Condition Wait Semantics

51761 It is important to note that when `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and
51762 `pthread_cond_wait()` return without error, the associated predicate may still be false. Similarly,
51763 when `pthread_cond_clockwait()` or `pthread_cond_timedwait()` returns with the timeout error, the
51764 associated predicate may be true due to an unavoidable race between the expiration of the
51765 timeout and the predicate state change.

51766 The application needs to recheck the predicate on any return because it cannot be sure there is
51767 another thread waiting on the thread to handle the signal, and if there is not then the signal is
51768 lost. The burden is on the application to check the predicate.

51769 Some implementations, particularly on a multi-processor, may sometimes cause multiple
51770 threads to wake up when the condition variable is signaled simultaneously on different
51771 processors.

51772 In general, whenever a condition wait returns, the thread has to re-evaluate the predicate
51773 associated with the condition wait to determine whether it can safely proceed, should wait
51774 again, or should declare a timeout. A return from the wait does not imply that the associated

51775 predicate is either true or false.

51776 It is thus recommended that a condition wait be enclosed in the equivalent of a “while loop”
51777 that checks the predicate.

51778 **Timed Wait Semantics**

51779 An absolute time measure was chosen for specifying the timeout parameter for two reasons.
51780 First, a relative time measure can be easily implemented on top of a function that specifies
51781 absolute time, but there is a race condition associated with specifying an absolute timeout on top
51782 of a function that specifies relative timeouts. For example, assume that `clock_gettime()` returns
51783 the current time and `cond_relative_timed_wait()` uses relative timeouts:

```
51784 clock_gettime(CLOCK_REALTIME, &now)
51785 reltime = sleep_til_this_absolute_time -now;
51786 cond_relative_timed_wait(c, m, &reltime);
```

51787 If the thread is preempted between the first statement and the last statement, the thread blocks
51788 for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout
51789 also need not be recomputed if it is used multiple times in a loop, such as that enclosing a
51790 condition wait.

51791 For cases when the system clock is advanced discontinuously by an operator, it is expected that
51792 implementations process any timed wait expiring at an intervening time as if that time had
51793 actually occurred. +

51794 **Choice of Clock** +

51795 Care should be taken to decide which clock is most appropriate when waiting with a timeout. +
51796 The system clock `CLOCK_REALTIME`, as used by default with `pthread_cond_timedwait()`, may be +
51797 subject to jumps forwards and backwards in order to correct it against actual time. +
51798 `CLOCK_MONOTONIC` is guaranteed not to jump backwards and must also advance in real +
51799 time, so using it via `pthread_cond_clockwait()` or `pthread_condattr_setclock()` may be more +
51800 appropriate. +

51801 **Cancellation and Condition Wait**

51802 A condition wait, whether timed or not, is a cancellation point. That is, the functions +
51803 `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, and `pthread_cond_wait()` are points where a -
51804 pending (or concurrent) cancellation request is noticed. The reason for this is that an indefinite
51805 wait is possible at these points—whatever event is being waited for, even if the program is
51806 totally correct, might never occur; for example, some input data being awaited might never be
51807 sent. By making condition wait a cancellation point, the thread can be canceled and perform its
51808 cancellation cleanup handler even though it may be stuck in some indefinite wait.

51809 A side-effect of acting on a cancellation request while a thread is blocked on a condition variable
51810 is to re-acquire the mutex before calling any of the cancellation cleanup handlers. This is done in
51811 order to ensure that the cancellation cleanup handler is executed in the same state as the critical
51812 code that lies both before and after the call to the condition wait function. This rule is also
51813 required when interfacing to POSIX threads from languages, such as Ada or C++, which may
51814 choose to map cancellation onto a language exception; this rule ensures that each exception
51815 handler guarding a critical section can always safely depend upon the fact that the associated
51816 mutex has already been locked regardless of exactly where within the critical section the
51817 exception was raised. Without this rule, there would not be a uniform rule that exception
51818 handlers could follow regarding the lock, and so coding would become very cumbersome.

51859 **Timed Condition Wait**

51860 The `pthread_cond_clockwait()` and `pthread_cond_timedwait()` functions allow an application to give
 51861 up waiting for a particular condition after a given amount of time. An example follows:

```
51862 (void) pthread_mutex_lock(&t.mn);
51863     t.waiters++;
51864     clock_gettime(CLOCK_MONOTONIC, &ts);
51865     ts.tv_sec += 5;
51866     rc = 0;
51867     while (! mypredicate(&t) && rc == 0)
51868         rc = pthread_cond_clockwait(&t.cond, &t.mn,
51869             CLOCK_MONOTONIC, &ts);
51870     t.waiters--;
51871     if (rc == 0 || mypredicate(&t))
51872         setmystate(&t);
51873 (void) pthread_mutex_unlock(&t.mn);
```

51874 By making the timeout parameter absolute, it does not need to be recomputed each time the
 51875 program checks its blocking predicate. If the timeout was relative, it would have to be
 51876 recomputed before each call. This would be especially difficult since such code would need to
 51877 take into account the possibility of extra wakeups that result from extra broadcasts or signals on
 51878 the condition variable that occur before either the predicate is true or the timeout is due. Using
 51879 `CLOCK_MONOTONIC` rather than `CLOCK_REALTIME` means that the timeout is not
 51880 influenced by the system clock being changed.

51881 **FUTURE DIRECTIONS**

51882 None.

51883 **SEE ALSO**

51884 [pthread_cond_broadcast\(\)](#)

51885 XBD [Section 4.13](#) (on page 91), [<pthread.h>](#)

51886 **CHANGE HISTORY**

51887 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

51888 **Issue 6**

51889 The `pthread_cond_timedwait()` and `pthread_cond_wait()` functions are marked as part of the
 51890 Threads option.

51891 The Open Group Corrigendum U021/9 is applied, correcting the prototype for the
 51892 `pthread_cond_wait()` function.

51893 The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics
 51894 for the Clock Selection option.

51895 The ERRORS section has an additional case for [EPERM] in response to IEEE PASC
 51896 Interpretation 1003.1c #28.

51897 The **restrict** keyword is added to the `pthread_cond_timedwait()` and `pthread_cond_wait()`
 51898 prototypes for alignment with the ISO/IEC 9899:1999 standard.

51899 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/89 is applied, updating the
 51900 DESCRIPTION for consistency with the `pthread_cond_destroy()` function that states it is safe to
 51901 destroy an initialized condition variable upon which no threads are currently blocked.

51902 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/90 is applied, updating words in the
 51903 DESCRIPTION from “the cancelability enable state” to “the cancelability type”.

51993 **NAME**

51994 pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection
51995 condition variable attribute

51996 **SYNOPSIS**

```
51997 #include <pthread.h>
51998 int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
51999     clockid_t *restrict clock_id);
52000 int pthread_condattr_setclock(pthread_condattr_t *attr,
52001     clockid_t clock_id);
```

52002 **DESCRIPTION**

52003 The *pthread_condattr_getclock()* function shall obtain the value of the *clock* attribute from the
52004 attributes object referenced by *attr*.

52005 The *pthread_condattr_setclock()* function shall set the *clock* attribute in an initialized attributes
52006 object referenced by *attr*. If *pthread_condattr_setclock()* is called with a *clock_id* argument that
52007 refers to a CPU-time clock, the call shall fail.

52008 The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of
52009 *pthread_cond_timedwait()*. The default value of the *clock* attribute shall refer to the system clock.
52010 The *clock* attribute shall have no effect on the *pthread_cond_clockwait()* function.

52011 The behavior is undefined if the value specified by the *attr* argument to
52012 *pthread_condattr_getclock()* or *pthread_condattr_setclock()* does not refer to an initialized condition
52013 variable attributes object.

52014 **RETURN VALUE**

52015 If successful, the *pthread_condattr_getclock()* function shall return zero and store the value of the
52016 clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error
52017 number shall be returned to indicate the error.

52018 If successful, the *pthread_condattr_setclock()* function shall return zero; otherwise, an error
52019 number shall be returned to indicate the error.

52020 **ERRORS**

52021 The *pthread_condattr_setclock()* function may fail if:

52022 [EINVAL] The value specified by *clock_id* does not refer to a known clock, or is a CPU-
52023 time clock.

52024 These functions shall not return an error code of [EINTR].

52025 **EXAMPLES**

52026 None.

52027 **APPLICATION USAGE**

52028 None.

52029 **RATIONALE**

52030 If an implementation detects that the value specified by the *attr* argument to
52031 *pthread_condattr_getclock()* or *pthread_condattr_setclock()* does not refer to an initialized condition
52032 variable attributes object, it is recommended that the function should fail and report an
52033 [EINVAL] error.

53021 **NAME**

53022 pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

53023 **SYNOPSIS**

```
53024 #include <pthread.h>
53025 int pthread_mutex_destroy(pthread_mutex_t *mutex);
53026 int pthread_mutex_init(pthread_mutex_t *restrict mutex,
53027     const pthread_mutexattr_t *restrict attr);
53028 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

53029 **DESCRIPTION**

53030 The *pthread_mutex_destroy()* function shall destroy the mutex object referenced by *mutex*; the
 53031 mutex object becomes, in effect, uninitialized. An implementation may cause
 53032 *pthread_mutex_destroy()* to set the object referenced by *mutex* to an invalid value.

53033 A destroyed mutex object can be reinitialized using *pthread_mutex_init()*; the results of otherwise
 53034 referencing the object after it has been destroyed are undefined.

53035 It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked
 53036 mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a
 53037 *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or *pthread_cond_wait()* call by another thread,
 53038 results in undefined behavior.

53039 The *pthread_mutex_init()* function shall initialize the mutex referenced by *mutex* with attributes
 53040 specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the
 53041 same as passing the address of a default mutex attributes object. Upon successful initialization,
 53042 the state of the mutex becomes initialized and unlocked.

53043 See [Section 2.9.9](#) (on page 508) for further requirements.

53044 Attempting to initialize an already initialized mutex results in undefined behavior.

53045 In cases where default mutex attributes are appropriate, the macro
 53046 PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes. The effect shall be
 53047 equivalent to dynamic initialization by a call to *pthread_mutex_init()* with parameter *attr*
 53048 specified as NULL, except that no error checks are performed.

53049 The behavior is undefined if the value specified by the *mutex* argument to
 53050 *pthread_mutex_destroy()* does not refer to an initialized mutex.

53051 The behavior is undefined if the value specified by the *attr* argument to *pthread_mutex_init()*
 53052 does not refer to an initialized mutex attributes object.

53053 **RETURN VALUE**

53054 If successful, the *pthread_mutex_destroy()* and *pthread_mutex_init()* functions shall return zero;
 53055 otherwise, an error number shall be returned to indicate the error.

53056 **ERRORS**

53057 The *pthread_mutex_init()* function shall fail if:

53058 [EAGAIN] The system lacked the necessary resources (other than memory) to initialize
 53059 another mutex.

53060 [ENOMEM] Insufficient memory exists to initialize the mutex.

53061 [EPERM] The caller does not have the privilege to perform the operation.

53062 The *pthread_mutex_init()* function may fail if:
 53063 [EINVAL] The attributes object referenced by *attr* has the robust mutex attribute set
 53064 without the process-shared attribute being set.
 53065 These functions shall not return an error code of [EINTR].

53066 EXAMPLES

53067 None.

53068 APPLICATION USAGE

53069 None.

53070 RATIONALE

53071 If an implementation detects that the value specified by the *mutex* argument to
 53072 *pthread_mutex_destroy()* does not refer to an initialized mutex, it is recommended that the
 53073 function should fail and report an [EINVAL] error.

53074 If an implementation detects that the value specified by the *mutex* argument to
 53075 *pthread_mutex_destroy()* or *pthread_mutex_init()* refers to a locked mutex or a mutex that is
 53076 referenced (for example, while being used in a *pthread_cond_clockwait()*,
 53077 *pthread_cond_timedwait()*, or *pthread_cond_wait()* call) by another thread, or detects that the value
 53078 specified by the *mutex* argument to *pthread_mutex_init()* refers to an already initialized mutex, it
 53079 is recommended that the function should fail and report an [EBUSY] error.

53080 If an implementation detects that the value specified by the *attr* argument to
 53081 *pthread_mutex_init()* does not refer to an initialized mutex attributes object, it is recommended
 53082 that the function should fail and report an [EINVAL] error.

53083 Alternate Implementations Possible

53084 This volume of POSIX.1-202x supports several alternative implementations of mutexes. An
 53085 implementation may store the lock directly in the object of type **pthread_mutex_t**. Alternatively,
 53086 an implementation may store the lock in the heap and merely store a pointer, handle, or unique
 53087 ID in the mutex object. Either implementation has advantages or may be required on certain
 53088 hardware configurations. So that portable code can be written that is invariant to this choice, this
 53089 volume of POSIX.1-202x does not define assignment or equality for this type, and it uses the
 53090 term “initialize” to reinforce the (more restrictive) notion that the lock may actually reside in the
 53091 mutex object itself.

53092 Note that this precludes an over-specification of the type of the mutex or condition variable and
 53093 motivates the opaqueness of the type.

53094 An implementation is permitted, but not required, to have *pthread_mutex_destroy()* store an
 53095 illegal value into the mutex. This may help detect erroneous programs that try to lock (or
 53096 otherwise reference) a mutex that has already been destroyed.

53097 Tradeoff Between Error Checks and Performance Supported

53098 Many error conditions that can occur are not required to be detected by the implementation in
 53099 order to let implementations trade off performance *versus* degree of error checking according to
 53100 the needs of their specific applications and execution environment. As a general rule, conditions
 53101 caused by the system (such as insufficient memory) are required to be detected, but conditions
 53102 caused by an erroneously coded application (such as failing to provide adequate
 53103 synchronization to prevent a mutex from being deleted while in use) are specified to result in
 53104 undefined behavior.

53105 A wide range of implementations is thus made possible. For example, an implementation

53199 particular, it can happen at most as many times as there are statically allocated synchronization
 53200 objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init()* or
 53201 *pthread_cond_init()*.

53202 Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient
 53203 performance for an application on some implementation, the application can avoid static
 53204 initialization altogether by explicitly initializing all synchronization objects with the
 53205 corresponding *pthread*_init()* functions, which are supported by all implementations. An
 53206 implementation can also document the tradeoffs and advise which initialization technique is
 53207 more efficient for that particular implementation.

53208 **Destroying Mutexes**

53209 A mutex can be destroyed immediately after it is unlocked. However, since attempting to
 53210 destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is
 53211 being used in a *pthread_cond_clockwait()*, *pthread_cond_timedwait()*, or *pthread_cond_wait()* call by
 53212 another thread, results in undefined behavior, care must be taken to ensure that no other thread
 53213 may be referencing the mutex.

53214 **Robust Mutexes**

53215 Implementations are required to provide robust mutexes for mutexes with the process-shared
 53216 attribute set to PTHREAD_PROCESS_SHARED. Implementations are allowed, but not required,
 53217 to provide robust mutexes when the process-shared attribute is set to
 53218 PTHREAD_PROCESS_PRIVATE.

53219 **FUTURE DIRECTIONS**

53220 None.

53221 **SEE ALSO**

53222 *pthread_mutex_getprioceiling()*, *pthread_mutexattr_getrobust()*, *pthread_mutex_lock()*,
 53223 *pthread_mutex_timedlock()*, *pthread_mutexattr_getpshared()*

53224 XBD <pthread.h>

53225 **CHANGE HISTORY**

53226 First released in Issue 5. Included for alignment with the POSIX Threads Extension.

53227 **Issue 6**

53228 The *pthread_mutex_destroy()* and *pthread_mutex_init()* functions are marked as part of the
 53229 Threads option.

53230 The *pthread_mutex_timedlock()* function is added to the SEE ALSO section for alignment with
 53231 IEEE Std 1003.1d-1999.

53232 IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

53233 The **restrict** keyword is added to the *pthread_mutex_init()* prototype for alignment with the
 53234 ISO/IEC 9899:1999 standard.

53235 **Issue 7**

53236 Changes are made from The Open Group Technical Standard, 2006, Extended API Set Part 3.

53237 The *pthread_mutex_destroy()* and *pthread_mutex_init()* functions are moved from the Threads
 53238 option to the Base.

53239 The [EINVAL] error for an uninitialized mutex or an uninitialized mutex attributes object is
 53240 removed; this condition results in undefined behavior.

53518 **NAME**

53519 pthread_mutex_clocklock, pthread_mutex_timedlock — lock a mutex +

53520 **SYNOPSIS**

53521 #include <pthread.h>

53522 int pthread_mutex_clocklock(pthread_mutex_t *restrict mutex, +

53523 clockid_t clock_id, const struct timespec *restrict abstime); +

53524 int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,

53525 const struct timespec *restrict abstime);

53526 **DESCRIPTION**

53527 The *pthread_mutex_clocklock()* and *pthread_mutex_timedlock()* functions shall lock the mutex |
 53528 object referenced by *mutex*. If the mutex is already locked, the calling thread shall block until the |
 53529 mutex becomes available as in the *pthread_mutex_lock()* function. If the mutex cannot be locked |
 53530 without waiting for another thread to unlock the mutex, this wait shall be terminated when the |
 53531 specified timeout expires.

53532 The timeout shall expire when the absolute time specified by *abstime* passes, as measured by the |
 53533 clock on which timeouts are based (that is, when the value of that clock equals or exceeds |
 53534 *abstime*), or if the absolute time specified by *abstime* has already been passed at the time of the |
 53535 call.

53536 For *pthread_mutex_timedlock()*, the timeout shall be based on the CLOCK_REALTIME clock. For |
 53537 *pthread_mutex_clocklock()*, the timeout shall be based on the clock specified by the *clock_id* |
 53538 argument. The resolution of the timeout shall be the resolution of the clock on which it is based. |
 53539 Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to |
 53540 *pthread_mutex_clocklock()* as the *clock_id* argument.

53541 Under no circumstance shall the function fail with a timeout if the mutex can be locked |
 53542 immediately. The validity of the *abstime* parameter need not be checked if the mutex can be |
 53543 locked immediately.

53544 RPI|TPI As a consequence of the priority inheritance rules (for mutexes initialized with the |
 53545 PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the |
 53546 priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this |
 53547 thread is no longer among the threads waiting for the mutex.

53548 If *mutex* is a robust mutex and the process containing the owning thread terminated while |
 53549 holding the mutex lock, a call to *pthread_mutex_clocklock()* or *pthread_mutex_timedlock()* shall +
 53550 return the error value [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread |
 53551 terminated while holding the mutex lock, a call to *pthread_mutex_clocklock()* or +
 53552 *pthread_mutex_timedlock()* may return the error value [EOWNERDEAD] even if the process in |
 53553 which the owning thread resides has not terminated. In these cases, the mutex is locked by the |
 53554 thread but the state it protects is marked as inconsistent. The application should ensure that the |
 53555 state is made consistent for reuse and when that is complete call *pthread_mutex_consistent()*. If |
 53556 the application is unable to recover the state, it should unlock the mutex without a prior call to |
 53557 *pthread_mutex_consistent()*, after which the mutex is marked permanently unusable.

53558 If *mutex* does not refer to an initialized mutex object, the behavior is undefined.

53559 **RETURN VALUE**

53560 If successful, the *pthread_mutex_clocklock()* and *pthread_mutex_timedlock()* functions shall return |
 53561 zero; otherwise, an error number shall be returned to indicate the error.

53562 **ERRORS**

53563 The `pthread_mutex_clocklock()` and `pthread_mutex_timedlock()` functions shall fail if:

53564 [EAGAIN] The mutex could not be acquired because the maximum number of recursive
53565 locks for *mutex* has been exceeded.

53566 [EAGAIN] The mutex is a robust mutex and the system resources available for robust
53567 mutexes owned would be exceeded.

53568 [EDEADLK] The mutex type is `PTHREAD_MUTEX_ERRORCHECK` and the current
53569 thread already owns the mutex.

53570 [EINVAL] The mutex was created with the protocol attribute having the value
53571 `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than
53572 the mutex' current priority ceiling.

53573 [EINVAL] The process or thread would have blocked, and either the *abstime* parameter
53574 specified a nanoseconds field value less than zero or greater than or equal to
53575 1 000 million, or the `pthread_mutex_clocklock()` function was passed an invalid
53576 or unsupported *clock_id* value.

53577 [ENOTRECOVERABLE]
53578 The state protected by the mutex is not recoverable.

53579 [EOWNERDEAD]
53580 The mutex is a robust mutex and the process containing the previous owning
53581 thread terminated while holding the mutex lock. The mutex lock shall be
53582 acquired by the calling thread and it is up to the new owner to make the state
53583 consistent.

53584 [ETIMEDOUT] The mutex could not be locked before the specified timeout expired.

53585 The `pthread_mutex_clocklock()` and `pthread_mutex_timedlock()` functions may fail if:

53586 [EDEADLK] A deadlock condition was detected.

53587 [EOWNERDEAD]
53588 The mutex is a robust mutex and the previous owning thread terminated
53589 while holding the mutex lock. The mutex lock shall be acquired by the calling
53590 thread and it is up to the new owner to make the state consistent.

53591 This function shall not return an error code of [EINTR].

53592 **EXAMPLES**

53593 None.

53594 **APPLICATION USAGE**

53595 Applications that have assumed that non-zero return values are errors will need updating for
53596 use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting
53597 a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error
53598 returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If
53599 an application is supposed to work with normal and robust mutexes, it should check all return
53600 values for error conditions and if necessary take appropriate action.

53601 **RATIONALE**

53602 Refer to `pthread_mutex_lock()`.

54163 **NAME**

54164 pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set the mutex type attribute

54165 **SYNOPSIS**

```
54166 #include <pthread.h>
54167 int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
54168 int *restrict type);
54169 int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

54170 **DESCRIPTION**

54171 The *pthread_mutexattr_gettype()* and *pthread_mutexattr_settype()* functions, respectively, shall get
54172 and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The
54173 default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

54174 The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types
54175 include:

```
54176 PTHREAD_MUTEX_NORMAL
54177 PTHREAD_MUTEX_ERRORCHECK
54178 PTHREAD_MUTEX_RECURSIVE
54179 PTHREAD_MUTEX_DEFAULT
```

54180 The mutex type affects the behavior of calls which lock and unlock the mutex. See
54181 *pthread_mutex_lock()* for details. An implementation may map PTHREAD_MUTEX_DEFAULT
54182 to one of the other mutex types.

54183 The behavior is undefined if the value specified by the *attr* argument to
54184 *pthread_mutexattr_gettype()* or *pthread_mutexattr_settype()* does not refer to an initialized mutex
54185 attributes object.

54186 **RETURN VALUE**

54187 Upon successful completion, the *pthread_mutexattr_gettype()* function shall return zero and store
54188 the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise,
54189 an error shall be returned to indicate the error.

54190 If successful, the *pthread_mutexattr_settype()* function shall return zero; otherwise, an error
54191 number shall be returned to indicate the error.

54192 **ERRORS**

54193 The *pthread_mutexattr_settype()* function shall fail if:

54194 [EINVAL] The value *type* is invalid.

54195 These functions shall not return an error code of [EINTR].

54196 **EXAMPLES**

54197 None.

54198 **APPLICATION USAGE**

54199 It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with
54200 condition variables because the implicit unlock performed in a *pthread_cond_clockwait()*,
54201 *pthread_cond_timedwait()*, or *pthread_cond_wait()* call may not actually release the mutex (if it had
54202 been locked multiple times). If this happens, no other thread can satisfy the condition of the
54203 predicate.

54555 **NAME**

54556 pthread_rwlock_clockrdlock, pthread_rwlock_timedrdlock — lock a read-write lock for reading +

54557 **SYNOPSIS**

54558 #include <pthread.h>

54559 int pthread_rwlock_clockrdlock(pthread_rwlock_t *restrict *rwlock*, +54560 clockid_t *clock_id*, const struct timespec *restrict *abstime*); +54561 int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict *rwlock*,54562 const struct timespec *restrict *abstime*);54563 **DESCRIPTION**

54564 The *pthread_rwlock_clockrdlock()* and *pthread_rwlock_timedrdlock()* functions shall apply a read |
 54565 lock to the read-write lock referenced by *rwlock* as in the *pthread_rwlock_rdlock()* function. |
 54566 However, if the lock cannot be acquired without waiting for other threads to unlock the lock, |
 54567 this wait shall be terminated when the specified timeout expires. The timeout shall expire when |
 54568 the absolute time specified by *abstime* passes, as measured by the clock on which timeouts are |
 54569 based (that is, when the value of that clock equals or exceeds *abstime*), or if the absolute time |
 54570 specified by *abstime* has already been passed at the time of the call.

54571 For *pthread_rwlock_timedrdlock()*, the timeout shall be based on the CLOCK_REALTIME clock. |
 54572 For *pthread_rwlock_clockrdlock()*, the timeout shall be based on the clock specified by the *clock_id* |
 54573 argument. The resolution of the timeout shall be the resolution of the clock on which it is based. |
 54574 Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to |
 54575 *pthread_rwlock_clockrdlock()* as the *clock_id* argument. |

54576 Under no circumstances shall the function fail with a timeout if the lock can be acquired |
 54577 immediately. The validity of the *abstime* parameter need not be checked if the lock can be |
 54578 immediately acquired.

54579 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read- |
 54580 write lock via a call to *pthread_rwlock_clockrdlock()* or *pthread_rwlock_timedrdlock()*, upon return +
 54581 from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

54582 The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*.
 54583 The results are undefined if this function is called with an uninitialized read-write lock.

54584 **RETURN VALUE**

54585 The *pthread_rwlock_clockrdlock()* and *pthread_rwlock_timedrdlock()* functions shall return zero if |
 54586 the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an |
 54587 error number shall be returned to indicate the error.

54588 **ERRORS**54589 The *pthread_rwlock_clockrdlock()* and *pthread_rwlock_timedrdlock()* functions shall fail if: |

54590 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired. |

54591 The *pthread_rwlock_clockrdlock()* and *pthread_rwlock_timedrdlock()* functions may fail if: |54592 [EAGAIN] The read lock could not be acquired because the maximum number of read |
54593 locks for lock would be exceeded.54594 [EDEADLK] A deadlock condition was detected or the calling thread already holds a write |
54595 lock on *rwlock*.54596 [EINVAL] The *abstime* nanosecond value is less than zero or greater than or equal to 1 000 |
54597 million, or the *pthread_rwlock_clockrdlock()* function was passed an invalid or |
54598 unsupported *clock_id* value.

54599 This function shall not return an error code of [EINTR].

54600 **EXAMPLES**

54601 None.

54602 **APPLICATION USAGE**

54603 Applications using this function may be subject to priority inversion, as discussed in XBD
54604 [Section 3.260](#) (on page 66).

54605 **RATIONALE**

54606 If an implementation detects that the value specified by the *rwlock* argument to +
54607 *pthread_rwlock_clockrdlock()* or *pthread_rwlock_timedrdlock()* does not refer to an initialized read-
54608 write lock object, it is recommended that the function should fail and report an [EINVAL] error.

54609 **FUTURE DIRECTIONS**

54610 None.

54611 **SEE ALSO**

54612 [pthread_rwlock_destroy\(\)](#), [pthread_rwlock_rdlock\(\)](#), [pthread_rwlock_timedwrlock\(\)](#),
54613 [pthread_rwlock_trywrlock\(\)](#), [pthread_rwlock_unlock\(\)](#)

54614 XBD [Section 3.260](#) (on page 66), [Section 4.13](#) (on page 91), [<pthread.h>](#), [<time.h>](#)

54615 **CHANGE HISTORY**

54616 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

54617 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/102 is applied, updating the ERRORS
54618 section so that the [EDEADLK] error includes detection of a deadlock condition.

54619 **Issue 7**

54620 The *pthread_rwlock_timedrdlock()* function is moved from the Timeouts option to the Base.

54621 The [EINVAL] error for an uninitialized read-write lock object is removed; this condition results
54622 in undefined behavior.

54623 **Issue 8**

54624 Austin Group Defect 592 is applied, removing text relating to [<time.h>](#) from the SYNOPSIS and
54625 DESCRIPTION sections. +

54626 Austin Group Defect 1216 is applied, adding *pthread_rwlock_clockrdlock()*. |

54627 **NAME**

54628 pthread_rwlock_clockwrlock, pthread_rwlock_timedwrlock — lock a read-write lock for writing +

54629 **SYNOPSIS**

54630 #include <pthread.h>

54631 int pthread_rwlock_clockwrlock(pthread_rwlock_t *restrict *rwlock*, +54632 clockid_t *clock_id*, const struct timespec *restrict *abstime*); +54633 int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict *rwlock*,54634 const struct timespec *restrict *abstime*);54635 **DESCRIPTION**

54636 The *pthread_rwlock_clockwrlock()* and *pthread_rwlock_timedwrlock()* functions shall apply a write
 54637 lock to the read-write lock referenced by *rwlock* as in the *pthread_rwlock_wrlock()* function.
 54638 However, if the lock cannot be acquired without waiting for other threads to unlock the lock,
 54639 this wait shall be terminated when the specified timeout expires. The timeout shall expire when
 54640 the absolute time specified by *abstime* passes, as measured by the clock on which timeouts are
 54641 based (that is, when the value of that clock equals or exceeds *abstime*), or if the absolute time
 54642 specified by *abstime* has already been passed at the time of the call.

54643 For *pthread_rwlock_timedwrlock()*, the timeout shall be based on the CLOCK_REALTIME clock.
 54644 For *pthread_rwlock_clockwrlock()*, the timeout shall be based on the clock specified by the *clock_id*
 54645 argument. The resolution of the timeout shall be the resolution of the clock on which it is based.
 54646 Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to
 54647 *pthread_rwlock_clockwrlock()* as the *clock_id* argument.

54648 Under no circumstances shall the function fail with a timeout if the lock can be acquired
 54649 immediately. The validity of the *abstime* parameter need not be checked if the lock can be
 54650 immediately acquired.

54651 If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-
 54652 write lock via a call to *pthread_rwlock_clockwrlock()* or *pthread_rwlock_timedwrlock()*, upon return
 54653 from the signal handler the thread shall resume waiting for the lock as if it was not interrupted. +

54654 The calling thread may deadlock if at the time the call is made it holds the read-write lock. The
 54655 results are undefined if this function is called with an uninitialized read-write lock.

54656 **RETURN VALUE**

54657 The *pthread_rwlock_clockwrlock()* and *pthread_rwlock_timedwrlock()* functions shall return zero if
 54658 the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an
 54659 error number shall be returned to indicate the error.

54660 **ERRORS**

54661 The *pthread_rwlock_clockwrlock()* and *pthread_rwlock_timedwrlock()* functions shall fail if:

54662 [ETIMEDOUT] The lock could not be acquired before the specified timeout expired.

54663 The *pthread_rwlock_clockwrlock()* and *pthread_rwlock_timedwrlock()* functions may fail if:

54664 [EDEADLK] A deadlock condition was detected or the calling thread already holds the
 54665 *rwlock*.

54666 [EINVAL] The *abstime* nanosecond value is less than zero or greater than or equal to 1 000
 54667 million, or the *pthread_rwlock_clockwrlock()* function was passed an invalid or
 54668 unsupported *clock_id* value.

54669 This function shall not return an error code of [EINTR].

54670 **EXAMPLES**

54671 None.

54672 **APPLICATION USAGE**

54673 Applications using this function may be subject to priority inversion, as discussed in XBD
 54674 [Section 3.260](#) (on page 66).

54675 **RATIONALE**

54676 If an implementation detects that the value specified by the *rwlock* argument to +
 54677 *pthread_rwlock_clockwrlock()* or *pthread_rwlock_timedwrlock()* does not refer to an initialized read-
 54678 write lock object, it is recommended that the function should fail and report an [EINVAL] error.

54679 **FUTURE DIRECTIONS**

54680 None.

54681 **SEE ALSO**

54682 [pthread_rwlock_destroy\(\)](#), [pthread_rwlock_rdlock\(\)](#), [pthread_rwlock_timedrdlock\(\)](#),
 54683 [pthread_rwlock_trywrlock\(\)](#), [pthread_rwlock_unlock\(\)](#)

54684 XBD [Section 3.260](#) (on page 66), [Section 4.13](#) (on page 91), [<pthread.h>](#), [<time.h>](#)

54685 **CHANGE HISTORY**

54686 First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

54687 IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/103 is applied, updating the ERRORS
 54688 section so that the [EDEADLK] error includes detection of a deadlock condition.

54689 **Issue 7**54690 The *pthread_rwlock_timedwrlock()* function is moved from the Timeouts option to the Base.

54691 The [EINVAL] error for an uninitialized read-write lock object is removed; this condition results
 54692 in undefined behavior.

54693 **Issue 8**

54694 Austin Group Defect 592 is applied, removing text relating to [<time.h>](#) from the SYNOPSIS and
 54695 DESCRIPTION sections. +

54696 Austin Group Defect 1216 is applied, adding *pthread_rwlock_clockwrlock()*. |

55818 **NAME**

55819 qsort, qsort_r — sort a table of data

+

55820 **SYNOPSIS**

55821 #include <stdlib.h>

55822 void qsort(void *base, size_t nel, size_t width,
55823 int (*compar)(const void *, const void *));55824 CX void qsort_r(void *base, size_t nel, size_t width,
55825 int (*compar)(const void *, const void *, void *), void *arg);

+

+

55826 **DESCRIPTION**55827 CX For *qsort()*: The functionality described on this reference page is aligned with the ISO C
55828 standard. Any conflict between the requirements described here and the ISO C standard is
55829 unintentional. This volume of POSIX.1-202x defers to the ISO C standard.

+

55830 The *qsort()* function shall sort an array of *nel* objects, the initial element of which is pointed to by
55831 *base*. The size of each object, in bytes, is specified by the *width* argument. If the *nel* argument has
55832 the value zero, the comparison function pointed to by *compar* shall not be called and no
55833 rearrangement shall take place.55834 The application shall ensure that the comparison function pointed to by *compar* does not alter the
55835 contents of the array. The implementation may reorder elements of the array between calls to the
55836 comparison function, but shall not alter the contents of any individual element.55837 When the same objects (consisting of *width* bytes, irrespective of their current positions in the
55838 array) are passed more than once to the comparison function, the results shall be consistent with
55839 one another. That is, they shall define a total ordering on the array.55840 The contents of the array shall be sorted in ascending order according to a comparison function.
55841 The *compar* argument is a pointer to the comparison function, which is called with two
55842 arguments that point to the elements being compared. The application shall ensure that the
55843 function returns an integer less than, equal to, or greater than 0, if the first argument is
55844 considered respectively less than, equal to, or greater than the second. If two members compare
55845 as equal, their order in the sorted array is unspecified.55846 CX The *qsort_r()* function shall be identical to *qsort()* except that the comparison function *compar*
55847 takes a third argument. The *arg* opaque pointer passed to *qsort_r()* shall in turn be passed as the
55848 third argument to the comparison function.

|

|

|

55849 **RETURN VALUE**

55850 These functions shall not return a value.

55851 **ERRORS**

55852 No errors are defined.

55853 **EXAMPLES**

55854 None.

55855 **APPLICATION USAGE**55856 The comparison function need not compare every byte, so arbitrary data may be contained in
55857 the elements in addition to the values being compared.

+

55858 If the *compar* callback function requires any additional state outside of the items being sorted, it
55859 can only access this state through global variables, making it potentially unsafe to use *qsort()*
55860 with the same *compar* function from separate threads at the same time. The *qsort_r()* function
55861 was added with the ability to pass through arbitrary arguments to the comparator, which avoids
55862 the need to access global variables and thus making it possible to safely share a stateful

+

+

+

+

+

55863 comparator across threads.

55864 **RATIONALE**

55865 The requirement that each argument (hereafter referred to as *p*) to the comparison function is a
55866 pointer to elements of the array implies that for every call, for each argument separately, all of
55867 the following expressions are non-zero:

```
55868 ((char *)p - (char *)base) % width == 0
55869 (char *)p >= (char *)base
55870 (char *)p < (char *)base + nel * width
```

55871 **FUTURE DIRECTIONS**

55872 None.

55873 **SEE ALSO**

55874 [alphasort\(\)](#)

55875 XBD <stdlib.h>

55876 **CHANGE HISTORY**

55877 First released in Issue 1. Derived from Issue 1 of the SVID.

55878 **Issue 6**

55879 The normative text is updated to avoid use of the term “must” for application requirements.

55880 IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/49 is applied, adding the last sentence to
55881 the first non-shaded paragraph in the DESCRIPTION, and the following two paragraphs. The
55882 RATIONALE is also updated. These changes are for alignment with the ISO C standard. +

55883 **Issue 8** +

55884 Austin Group Defect 900 is applied, adding the *qsort_r()* function. |

```

55971         keystr[len++] = c;
55972     }
55973     keystr[len] = '\0';
55974     printf("%s Element%0*ld\n", keystr, elementlen, i);
55975     len = 0;
55976 }

```

55977 **Generating the Same Sequence on Different Machines**

55978 The following code defines a pair of functions that could be incorporated into applications
55979 wishing to ensure that the same sequence of numbers is generated across different machines.

```

55980 static unsigned long next = 1;
55981 int myrand(void) /* RAND_MAX assumed to be 32767. */
55982 {
55983     next = next * 1103515245 + 12345;
55984     return((unsigned)(next/65536) % 32768);
55985 }
55986 void mysrand(unsigned seed)
55987 {
55988     next = seed;
55989 }

```

55990 **APPLICATION USAGE**

55991 These functions should be avoided whenever non-trivial requirements (including safety) have to
55992 be fulfilled, unless seeded using *getentropy()*.

55993 The *drand48()* and *random()* functions provide much more elaborate pseudo-random number
55994 generators.

55995 **RATIONALE**

55996 The ISO C standard *rand()* and *srand()* functions allow per-process pseudo-random streams
55997 shared by all threads. Those two functions need not change, but there has to be mutual-
55998 exclusion that prevents interference between two threads concurrently accessing the random
55999 number generator.

56000 With regard to *rand()*, there are two different behaviors that may be wanted in a multi-threaded
56001 program:

- 56002 1. A single per-process sequence of pseudo-random numbers that is shared by all threads
56003 that call *rand()*
- 56004 2. A different sequence of pseudo-random numbers for each thread that calls *rand()*

56005 This is provided by the modified thread-safe function based on whether the seed value is global
56006 to the entire process or local to each thread.

56007 This does not address the known deficiencies of the *rand()* function implementations, which
56008 have been approached by maintaining more state. In effect, this specifies new thread-safe forms
56009 of a deficient function.

56010 **FUTURE DIRECTIONS**

56011 None.

56680 **NAME**

56681 realloc, reallocarray — memory reallocators |

56682 **SYNOPSIS**

56683 #include <stdlib.h>

56684 void *realloc(void *ptr, size_t size);

56685 CX void *reallocarray(void *ptr, size_t nelem, size_t elsize); +

56686 **DESCRIPTION**

56687 CX For *realloc()*: The functionality described on this reference page is aligned with the ISO C +
 56688 standard. Any conflict between the requirements described here and the ISO C standard is
 56689 unintentional. This volume of POSIX.1-202x defers to the ISO C standard.

56690 The *realloc()* function shall deallocate the old object pointed to by *ptr* and return a pointer to a
 56691 new object that has the size specified by *size*. The contents of the new object shall be the same as
 56692 that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in
 56693 the new object beyond the size of the old object have indeterminate values. If the size of the
 56694 space requested is zero, the behavior shall be implementation-defined: either a null pointer is
 56695 returned, or the behavior shall be as if the size were some non-zero value, except that the
 56696 behavior is undefined if the returned pointer is used to access an object. If the space cannot be
 56697 allocated, the object shall remain unchanged.

56698 CX The *reallocarray()* function shall be equivalent to the call *realloc(ptr, nelem * elsize)* +
 56699 except that overflow in the multiplication shall be an error. +

56700 CX If *ptr* is a null pointer, *realloc()* or *reallocarray()* shall be equivalent to *malloc()* for the specified |
 56701 size.

56702 If *ptr* does not match a pointer earlier returned by a function in POSIX.1-202x that allocates |
 56703 memory as if by *malloc()*, or if the space has previously been deallocated by a call to *free()*, |
 56704 CX *realloc()*, or *reallocarray()*, the behavior is undefined. |

56705 CX The order and contiguity of storage allocated by successive calls to *realloc()* or *reallocarray()* is |
 56706 unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it
 56707 may be assigned to a pointer to any type of object and then used to access such an object in the
 56708 space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield
 56709 a pointer to an object disjoint from any other object. The pointer returned shall point to the start
 56710 (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall
 56711 be returned.

56712 **RETURN VALUE**

56713 CX Upon successful completion, *realloc()* and *reallocarray()* shall return a pointer to the (possibly |
 56714 CX moved) allocated space. If *size* is 0, or either *nelem* or *elsize* is 0, then either:

- 56715 CX • A null pointer shall be returned and, if *ptr* is not a null pointer, *errno* shall be set to an
 56716 implementation-defined value.
- 56717 • A pointer to the allocated space shall be returned, and the memory object pointed to by *ptr*
 56718 shall be freed. The application shall ensure that the pointer is not used to access an object.

56719 CX If there is not enough available memory, *realloc()* and *reallocarray()* shall return a null pointer |
 56720 CX and set *errno* to [ENOMEM]. If *realloc()* or *reallocarray()* returns a null pointer and *errno* has |
 56721 been set to [ENOMEM], the memory referenced by *ptr* shall not be changed.

56722 **ERRORS**

- 56723 CX The `realloc()` and `reallocarray()` functions shall fail if: |
- 56724 CX [ENOMEM] Insufficient memory is available.
- 56725 CX The `reallocarray()` function shall fail if: +
- 56726 [ENOMEM] The calculation $nelem * elsize$ would overflow. +

56727 **EXAMPLES**

- 56728 None.

56729 **APPLICATION USAGE**

- 56730 The description of `realloc()` has been modified from previous versions of this standard to align
 56731 with the ISO/IEC 9899:1999 standard. Previous versions explicitly permitted a call to `realloc(p, 0)` |
 56732 to free the space pointed to by `p` and return a null pointer. While this behavior could be |
 56733 interpreted as permitted by this version of the standard, the C language committee has indicated |
 56734 that this interpretation is incorrect. Applications should assume that if `realloc()` returns a null
 56735 pointer, the space pointed to by `p` has not been freed. Since this could lead to double-frees,
 56736 implementations should also set `errno` if a null pointer actually indicates a failure, and
 56737 applications should only free the space if `errno` was changed.

56738 **RATIONALE**

- 56739 None.

56740 **FUTURE DIRECTIONS**

- 56741 This standard defers to the ISO C standard. While that standard currently has language that
 56742 might permit `realloc(p, 0)`, where `p` is not a null pointer, to free `p` while still returning a null |
 56743 pointer, the committee responsible for that standard is considering clarifying the language to
 56744 explicitly prohibit that alternative.

56745 **SEE ALSO**

- 56746 `calloc()`, `free()`, `malloc()`
- 56747 XBD `<stdlib.h>`

56748 **CHANGE HISTORY**

- 56749 First released in Issue 1. Derived from Issue 1 of the SVID.

56750 **Issue 6**

- 56751 Extensions beyond the ISO C standard are marked.

56752 The following new requirements on POSIX implementations derive from alignment with the
 56753 Single UNIX Specification:

- 56754 • In the RETURN VALUE section, if there is not enough available memory, the setting of
 56755 `errno` to [ENOMEM] is added.
- 56756 • The [ENOMEM] error condition is added.

56757 **Issue 7**

56758 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0495 [400], XSH/TC1-2008/0496 [400],
 56759 XSH/TC1-2008/0497 [400], and XSH/TC1-2008/0498 [400] are applied.

56760 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0309 [526] and XSH/TC2-2008/0310
 56761 [526,688] are applied. +

58787 **NAME**

58788 sem_init — initialize an unnamed semaphore

58789 **SYNOPSIS**

58790 #include <semaphore.h>

58791 int sem_init(sem_t *sem, int pshared, unsigned value);

58792 **DESCRIPTION**

58793 The `sem_init()` function shall initialize the unnamed semaphore referred to by `sem`. The value of
 58794 the initialized semaphore shall be `value`. Following a successful call to `sem_init()`, the semaphore
 58795 can be used in subsequent calls to `sem_clockwait()`, `sem_destroy()`, `sem_post()`, `sem_timedwait()`,
 58796 `sem_trywait()`, and `sem_wait()`. This semaphore shall remain usable until the semaphore is
 58797 destroyed. An unnamed semaphore may be implemented using a file descriptor.

58798 If the `pshared` argument has a non-zero value, then the semaphore is shared between processes;
 58799 in this case, any process that can access the semaphore `sem` can use `sem` for performing
 58800 `sem_clockwait()`, `sem_destroy()`, `sem_post()`, `sem_timedwait()`, `sem_trywait()`, and `sem_wait()`
 58801 operations.

58802 If the `pshared` argument is zero, then the semaphore is shared between threads of the process; any
 58803 thread in this process can use `sem` for performing `sem_clockwait()`, `sem_destroy()`, `sem_post()`,
 58804 `sem_timedwait()`, `sem_trywait()`, and `sem_wait()` operations.

58805 See [Section 2.9.9](#) (on page 508) for further requirements.

58806 Attempting to initialize an already initialized semaphore results in undefined behavior.

58807 **RETURN VALUE**

58808 Upon successful completion, the `sem_init()` function shall initialize the semaphore in `sem` and
 58809 return 0. Otherwise, it shall return `-1` and set `errno` to indicate the error.

58810 **ERRORS**

58811 The `sem_init()` function shall fail if:

58812 [EINVAL] The `value` argument exceeds {SEM_VALUE_MAX}.

58813 [ENOSPC] A resource required to initialize the semaphore has been exhausted, or the
 58814 limit on semaphores ({SEM_NSEMS_MAX}) has been reached.

58815 [EPERM] The process lacks appropriate privileges to initialize the semaphore.

58816 The `sem_init()` function may fail if:

58817 [EMFILE] All file descriptors available to the process are currently open.

58818 [ENFILE] The maximum allowable number of files is currently open in the system.

58819 **EXAMPLES**

58820 None.

58821 **APPLICATION USAGE**

58822 None.

58823 **RATIONALE**

58824 None.

58825 **FUTURE DIRECTIONS**

58826 None.

58849 **NAME**

58850 sem_open — initialize and open a named semaphore

58851 **SYNOPSIS**

58852 #include <semaphore.h>

58853 sem_t *sem_open(const char *name, int oflag, ...);

58854 **DESCRIPTION**

58855 The *sem_open()* function shall establish a connection between a named semaphore and a process.
 58856 A named semaphore may be implemented using a file descriptor. Following a call to *sem_open()*
 58857 with semaphore name *name*, the process may reference the semaphore associated with *name*
 58858 using the address returned from the call. This semaphore can be used in subsequent calls to
 58859 *sem_clockwait()*, *sem_close()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, and *sem_wait()*. The
 58860 semaphore remains usable by this process until the semaphore is closed by a successful call to
 58861 *sem_close()*, *_exit()*, or one of the *exec* functions.

58862 The *oflag* argument controls whether the semaphore is created or merely accessed by the call to
 58863 *sem_open()*. The following flag bits may be set in *oflag*:

58864 **O_CREAT** This flag is used to create a semaphore if it does not already exist. If **O_CREAT** is
 58865 set and the semaphore already exists, then **O_CREAT** has no effect, except as noted
 58866 under **O_EXCL**. Otherwise, *sem_open()* creates a named semaphore. The **O_CREAT**
 58867 flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and
 58868 *value*, which is of type **unsigned**. The semaphore is created with an initial value of
 58869 *value*. Valid initial values for semaphores are less than or equal to
 58870 {SEM_VALUE_MAX}.

58871 The user ID of the semaphore shall be set to the effective user ID of the process.
 58872 The group ID of the semaphore shall be set to the effective group ID of the process;
 58873 however, if the *name* argument is visible in the file system, the group ID may be set
 58874 to the group ID of the containing directory. The permission bits of the semaphore
 58875 are set to the value of the *mode* argument except those set in the file mode creation
 58876 mask of the process. When bits in *mode* other than the file permission bits are
 58877 specified, the effect is unspecified.

58878 After the semaphore named *name* has been created by *sem_open()* with the
 58879 **O_CREAT** flag, other processes can connect to the semaphore by calling
 58880 *sem_open()* with the same value of *name*.

58881 **O_EXCL** If **O_EXCL** and **O_CREAT** are set, *sem_open()* fails if the semaphore *name* exists.
 58882 The check for the existence of the semaphore and the creation of the semaphore if it
 58883 does not exist are atomic with respect to other processes executing *sem_open()* with
 58884 **O_EXCL** and **O_CREAT** set. If **O_EXCL** is set and **O_CREAT** is not set, the effect is
 58885 undefined.

58886 If flags other than **O_CREAT** and **O_EXCL** are specified in the *oflag* parameter, the
 58887 effect is unspecified.

58888 The *name* argument points to a string naming a semaphore object. It is unspecified whether the
 58889 name appears in the file system and is visible to functions that take pathnames as arguments.
 58890 The *name* argument conforms to the construction rules for a pathname, except that the
 58891 interpretation of <slash> characters other than the leading <slash> character in *name* is
 58892 implementation-defined, and that the length limits for the *name* argument are implementation-
 58893 defined and need not be the same as the pathname limits {PATH_MAX} and {NAME_MAX}. If
 58894 *name* begins with the <slash> character, then processes calling *sem_open()* with the same value of
 58895 *name* shall refer to the same semaphore object, as long as that name has not been removed. If

59036 **NAME**

59037 sem_clockwait, sem_timedwait — lock a semaphore +

59038 **SYNOPSIS**

```
59039 #include <semaphore.h>
59040 int sem_clockwait(sem_t *restrict sem, clockid_t clock_id, +
59041                 const struct timespec *restrict abstime); +
59042 int sem_timedwait(sem_t *restrict sem,
59043                 const struct timespec *restrict abstime);
```

59044 **DESCRIPTION**

59045 The *sem_clockwait()* and *sem_timedwait()* functions shall lock the semaphore referenced by *sem* as |
 59046 in the *sem_wait()* function. However, if the semaphore cannot be locked without waiting for |
 59047 another process or thread to unlock the semaphore by performing a *sem_post()* function, this |
 59048 wait shall be terminated when the specified timeout expires.

59049 The timeout shall expire when the absolute time specified by *abstime* passes, as measured by the |
 59050 clock on which timeouts are based (that is, when the value of that clock equals or exceeds |
 59051 *abstime*), or if the absolute time specified by *abstime* has already been passed at the time of the |
 59052 call.

59053 For *sem_timedwait()*, the timeout shall be based on the CLOCK_REALTIME clock. For |
 59054 *sem_clockwait()*, the timeout shall be based on the clock specified by the *clock_id* argument. The |
 59055 resolution of the timeout shall be the resolution of the clock on which it is based. |
 59056 Implementations shall support passing CLOCK_REALTIME and CLOCK_MONOTONIC to |
 59057 *sem_clockwait()* as the *clock_id* argument.

59058 Under no circumstance shall the function fail with a timeout if the semaphore can be locked |
 59059 immediately. The validity of the *abstime* need not be checked if the semaphore can be locked |
 59060 immediately.

59061 **RETURN VALUE**

59062 The *sem_clockwait()* and *sem_timedwait()* functions shall return zero if the calling process |
 59063 successfully performed the semaphore lock operation on the semaphore designated by *sem*. If |
 59064 the call was unsuccessful, the state of the semaphore shall be unchanged, and the functions shall |
 59065 return a value of -1 and set *errno* to indicate the error.

59066 **ERRORS**

59067 The *sem_clockwait()* and *sem_timedwait()* functions shall fail if: |

59068 [EINVAL] The process or thread would have blocked, and either the *abstime* parameter |
 59069 specified a nanoseconds field value less than zero or greater than or equal to |
 59070 1 000 million, or the *sem_clockwait()* function was passed an invalid or |
 59071 unsupported *clock_id* value.

59072 [ETIMEDOUT] The semaphore could not be locked before the specified timeout expired.

59073 The *sem_clockwait()* and *sem_timedwait()* functions may fail if: |

59074 [EDEADLK] A deadlock condition was detected.

59075 [EINTR] A signal interrupted the function. |

59076 [EINVAL] The *sem* argument does not refer to a valid semaphore.

59077 **EXAMPLES**

59078 The program shown below operates on an unnamed semaphore. The program expects two
 59079 command-line arguments. The first argument specifies a seconds value that is used to set an
 59080 alarm timer to generate a SIGALRM signal. This handler performs a *sem_post()* to increment the
 59081 semaphore that is being waited on in *main()* using *sem_clockwait()*. The second command-line
 59082 argument specifies the length of the timeout, in seconds, for *sem_clockwait()*.

```

59083 #include <unistd.h>
59084 #include <stdio.h>
59085 #include <stdlib.h>
59086 #include <semaphore.h>
59087 #include <time.h>
59088 #include <assert.h>
59089 #include <errno.h>
59090 #include <signal.h>

59091 sem_t sem;

59092 static void
59093 handler(int sig)
59094 {
59095     int sav_errno = errno;
59096     static const char info_msg[] = "sem_post() from handler\n";
59097     write(STDOUT_FILENO, info_msg, sizeof info_msg - 1);
59098     if (sem_post(&sem) == -1) {
59099         static const char err_msg[] = "sem_post() failed\n";
59100         write(STDERR_FILENO, err_msg, sizeof err_msg - 1);
59101         _exit(EXIT_FAILURE);
59102     }
59103     errno = sav_errno;
59104 }

59105 int
59106 main(int argc, char *argv[])
59107 {
59108     struct sigaction sa;
59109     struct timespec ts;
59110     int s;

59111     if (argc != 3) {
59112         fprintf(stderr, "Usage: %s <alarm-secs> <wait-secs>\n",
59113             argv[0]);
59114         exit(EXIT_FAILURE);
59115     }

59116     if (sem_init(&sem, 0, 0) == -1) {
59117         perror("sem_init");
59118         exit(EXIT_FAILURE);
59119     }

59120     /* Establish SIGALRM handler; set alarm timer using argv[1] */

59121     sa.sa_handler = handler;
59122     sigemptyset(&sa.sa_mask);
59123     sa.sa_flags = 0;
59124     if (sigaction(SIGALRM, &sa, NULL) == -1) {

```

```

59125         perror("sigaction");
59126         exit(EXIT_FAILURE);
59127     }
59128     alarm(atoi(argv[1]));
59129     /* Calculate relative interval as current time plus
59130        number of seconds given argv[2] */
59131     if (clock_gettime(CLOCK_MONOTONIC, &ts) == -1) {
59132         perror("clock_gettime");
59133         exit(EXIT_FAILURE);
59134     }
59135     ts.tv_sec += atoi(argv[2]);
59136     printf("main() about to call sem_clockwait()\n");
59137     while ((s = sem_clockwait(&sem, CLOCK_MONOTONIC, &ts)) == -1 &&
59138         errno == EINTR)
59139         continue; /* Restart if interrupted by handler */
59140     /* Check what happened */
59141     if (s == -1) {
59142         if (errno == ETIMEDOUT)
59143             printf("sem_clockwait() timed out\n");
59144         else
59145             perror("sem_clockwait");
59146     } else
59147         printf("sem_clockwait() succeeded\n");
59148     exit((s == 0) ? EXIT_SUCCESS : EXIT_FAILURE);
59149 }

```

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

61660 sig2str, str2sig — translate between signal names and numbers
61661

SYNOPSIS

```
61663 CX #include <signal.h>
61664 int sig2str(int signum, char *str);
61665 int str2sig(const char *restrict str, int *restrict pnum);
61666
```

DESCRIPTION

61667 The *sig2str()* function shall translate the signal number specified by *signum* to a signal name and
61668 shall store this string in the location specified by *str*. The application shall ensure that *str* points
61669 to a location that can store the string including the terminating null byte. The symbolic constant
61670 SIG2STR_MAX defined in **<signal.h>** gives the maximum number of bytes required.
61671

61672 If *signum* is equal to 0, the behavior is unspecified.

61673 If *signum* is equal to one of the symbolic constants listed in the table of signal numbers in
61674 **<signal.h>**, the stored signal name shall be the name of the symbolic constant without the **SIG**
61675 prefix.

61676 If *signum* is equal to SIGRTMIN or SIGRTMAX, the stored string shall be "RTMIN" or "RTMAX",
61677 respectively.

61678 If *signum* is between SIGRTMIN+1 and (SIGRTMIN+SIGRTMAX)/2 inclusive, the stored string
61679 shall be of the form "RTMIN+n", where *n* is the shortest decimal representation of the value of
61680 *signum*-SIGRTMIN.

61681 If *signum* is between (SIGRTMIN+SIGRTMAX)/2 + 1 and SIGRTMAX-1 inclusive, the stored
61682 string shall be either of the form "RTMIN+n" or of the form "RTMAX-m", where *n* is the shortest
61683 decimal representation of the value of *signum*-SIGRTMIN and *m* is the shortest decimal
61684 representation of the value of SIGRTMAX-*signum*.

61685 If *signum* is a valid, supported signal number, is either less than SIGRTMIN or greater than
61686 SIGRTMAX, and is not equal to one of the symbolic constants listed in the table of signal
61687 numbers in **<signal.h>**, the stored string shall uniquely identify the signal number *signum* in an
61688 unspecified manner.

61689 The *str2sig()* function shall translate the signal name in the string pointed to by *str* to a signal
61690 number and shall store this value in the location specified by *pnum*.

61691 If *str* points to a string containing the name of one of the symbolic constants listed in the table of
61692 signal numbers in **<signal.h>**, without the **SIG** prefix, the stored signal number shall be equal to
61693 the value of the symbolic constant.

61694 If *str* points to the string "RTMIN" or "RTMAX", the stored value shall be equal to SIGRTMIN or
61695 SIGRTMAX, respectively.

61696 If *str* points to a string of the form "RTMIN+n", where *n* is a decimal representation of a number
61697 between 1 and SIGRTMAX-SIGRTMIN-1 inclusive, the stored value shall be equal to
61698 SIGRTMIN+n.

61699 If *str* points to a string of the form "RTMAX-n", where *n* is a decimal representation of a number
61700 between 1 and SIGRTMAX-SIGRTMIN-1 inclusive, the stored value shall be equal to
61701 SIGRTMAX-n.

61702 If *str* points to a string containing a decimal representation of a valid, supported signal number,
61703 the value stored in the location pointed to by *pnum* shall be equal to that number.

61704 If *str* points to a string containing a decimal representation of the value 0 and the string was not
61705 returned by a previous successful call to *sig2str()* with a *signum* argument of 0, the behavior is
61706 unspecified.

61707 If *str* points to a string returned by a previous successful call to *sig2str(signum, str)*, the value
61708 stored in the location pointed to by *pnum* shall be equal to *signum*.

61709 If *str* points to a string that does not meet any of the above criteria, *str2sig()* shall store a value in
61710 the location pointed to by *pnum* if and only if it recognizes the string as an additional
61711 implementation-dependent form of signal name.

61712 RETURN VALUE

61713 If *signum* is a valid, supported signal number (that is, one for which *kill()* does not return -1
61714 with *errno* set to [EINVAL]), the *sig2str()* function shall return 0; otherwise, if *signum* is not equal
61715 to 0, it shall return -1.

61716 If *str2sig()* stores a value in the location pointed to by *pnum*, it shall return 0; otherwise, it shall
61717 return -1.

61718 ERRORS

61719 No errors are defined.

61720 EXAMPLES

61721 None.

61722 APPLICATION USAGE

61723 None.

61724 RATIONALE

61725 Historical versions of these functions translated a *signum* value 0 to "EXIT" (and vice versa), so
61726 that they could be used by the shell for the *trap* utility. When adding the functions to this
61727 standard, the standard developers felt that they should be aimed at more general-purpose use,
61728 and consequently requiring this behavior did not seem appropriate and so the behavior in this
61729 case has been made unspecified.

61730 FUTURE DIRECTIONS

61731 None.

61732 SEE ALSO

61733 *kill()*, *sigaction()*, *strsignal()*

61734 XBD <signal.h>

61735 CHANGE HISTORY

61736 First released in Issue 8.

61737

NAME

str2sig — translate between signal names and numbers

63653

SYNOPSIS

63654

63655 CX `#include <signal.h>`63656 `int str2sig(const char *restrict str, int *restrict pnum);`

63657

DESCRIPTION

63658

63659 Refer to *sig2str()*.

Sanity Review

NAME

64871 strlcat, strlcpy — size-bounded string concatenation and copying
64872

SYNOPSIS

```
64874 CX #include <string.h>
64875 size_t strlcat(char *restrict dst, const char *restrict src,
64876 size_t dstsize);
64877 size_t strlcpy(char *restrict dst, const char *restrict src,
64878 size_t dstsize);
64879
```

DESCRIPTION

64880 The *strlcpy()* and *strlcat()* functions copy and concatenate strings, stopping when either a NUL
64881 terminator in the source string is encountered or the specified full size of the destination buffer is
64882 reached. They NUL terminate the result if there is room. The application should ensure that
64883 room for the NUL terminator is included in *dstsize*.
64884

64885 The *strlcpy()* function shall copy not more than *dstsize* – 1 bytes from the string pointed to by *src*
64886 to the array pointed to by *dst*; a NUL byte in *src* and bytes that follow it shall not be copied. A
64887 terminating NUL byte shall be appended to the result, unless *dstsize* is 0. If copying takes place
64888 between objects that overlap, the behavior is undefined.

64889 The *strlcat()* function shall append not more than *dstsize* – *strlen(dst)* – 1 bytes from the string
64890 pointed to by *src* to the end of the string pointed to by *dst*; a NUL byte in *src* and bytes that
64891 follow it shall not be appended. The initial byte of *src* shall overwrite the NUL byte at the end of
64892 *dst*. A terminating NUL byte shall be appended to the result, unless its location would be at or
64893 beyond *dst* + *dstsize*. If copying takes place between objects that overlap, the behavior is
64894 undefined.

64895 The *strlcpy()* and *strlcat()* functions shall not change the setting of *errno* on valid input.

RETURN VALUE

64896 Upon successful completion, the *strlcpy()* function shall return the length of the string pointed to
64897 by *src*; that is, the number of bytes in the string, not including the terminating NUL byte.
64898

64899 Upon successful completion, the *strlcat()* function shall return the initial length of the string
64900 pointed to by *dst* plus the length of the string pointed to by *src*.

64901 No return values are reserved to indicate an error.

ERRORS

64902 No errors are defined.
64903

EXAMPLES

64904 The following example detects truncation while combining a path prefix (including trailing
64905 <slash>) and a filename to produce a portable pathname:
64906

```
64907 char *prefix, *filenam, pathnam[_POSIX_PATH_MAX];
64908 if (strlcpy(pathnam, prefix, sizeof pathnam) >= sizeof pathnam ||
64909 strlcat(pathnam, filenam, sizeof pathnam) >= sizeof pathnam)
64910 {
64911 // truncation occurred
64912 ...
64913 }
```

64914 This code ensures there is room for the NUL terminator by:

- 64915 • Calling *strncpy()* with a non-zero *dstsize* argument.
- 64916 • Only calling *strlcat()* if the return value of *strncpy()* indicated that truncation did not occur.

64917 APPLICATION USAGE

64918 The return value of the *strncpy()* and *strlcat()* functions follows the same convention as
64919 *snprintf()*; that is, they return the total length of the string they tried to create. If the return value
64920 is greater than or equal to *dstsize*, the output string has been truncated.

64921 RATIONALE

64922 None.

64923 FUTURE DIRECTIONS

64924 None.

64925 SEE ALSO

64926 *fprintf()*, *strlen()*, *strncat()*, *strncpy()*, *wcslcat()*

64927 XBD <**string.h**>

64928 CHANGE HISTORY

64929 First released in Issue 8.

64930

NAME

71081 `wcslcat`, `wcsncpy` — size-bounded wide string concatenation and copying
 71082

SYNOPSIS

```
71084 CX #include <wchar.h>
71085
71085     size_t wcslcat(wchar_t *restrict dst, const wchar_t *restrict src,
71086                   size_t dstsize);
71087     size_t wcsncpy(wchar_t *restrict dst, const wchar_t *restrict src,
71088                   size_t dstsize);
71089
```

DESCRIPTION

71090 The `wcsncpy()` and `wcslcat()` functions copy and concatenate wide strings, stopping when either a
 71091 terminating null wide-character code in the source wide string is encountered or the specified
 71092 full size (in wide-character codes) of the destination buffer is reached. They null terminate the
 71093 result if there is room. The application should ensure that room for the terminating null wide-
 71094 character code is included in `dstsize`.
 71095

71096 The `wcsncpy()` function shall copy not more than `dstsize - 1` wide-character codes from the wide
 71097 string pointed to by `src` to the array pointed to by `dst`; a terminating null wide-character code in
 71098 `src` and wide-character codes that follow it shall not be copied. A terminating null wide-
 71099 character code shall be appended to the result, unless `dstsize` is 0. If copying takes place between
 71100 objects that overlap, the behavior is undefined.

71101 The `wcslcat()` function shall append not more than `dstsize - wcslen(dst) - 1` wide-character codes
 71102 from the wide string pointed to by `src` to the end of the wide string pointed to by `dst`; a
 71103 terminating null wide-character code in `src` and wide-character codes that follow it shall not be
 71104 appended. The initial wide-character code of `src` shall overwrite the null wide-character code at
 71105 the end of `dst`. A terminating null wide-character code shall be appended to the result, unless its
 71106 location would be at or beyond `dst + dstsize`. If copying takes place between objects that overlap,
 71107 the behavior is undefined.

71108 The `wcsncpy()` and `wcslcat()` functions shall not change the setting of `errno` on valid input.

RETURN VALUE

71109 Upon successful completion, the `wcsncpy()` function shall return the length of the wide string
 71110 pointed to by `src`; that is, the number of wide-character codes in the wide string, not including
 71111 the terminating null wide-character code.
 71112

71113 Upon successful completion, the `wcslcat()` function shall return the initial length of the wide
 71114 string pointed to by `dst` plus the length of the wide string pointed to by `src`.

71115 No return values are reserved to indicate an error.

ERRORS

71116 No errors are defined.
 71117

EXAMPLES

71118 None.
 71119

APPLICATION USAGE

71120 The return value of the `wcsncpy()` and `wcslcat()` functions follows the same convention as
 71121 `snprintf()`; that is, they return the total length (in wide-character codes) of the wide string they
 71122 tried to create. If the return value is greater than or equal to `dstsize`, the output wide string has
 71123 been truncated.
 71124

71125 **RATIONALE**

71126 None.

71127 **FUTURE DIRECTIONS**

71128 None.

71129 **SEE ALSO**

71130 *fprintf()*, *strlcat()*, *wcslen()*, *wcsncat()*, *wcsncpy()*

71131 XBD <**wchar.h**>

71132 **CHANGE HISTORY**

71133 First released in Issue 8.

71134

Sanity Review

121068 POSIX.1b is a software, source-level standard and most of the benefits of the alternate
121069 representation are enjoyed by hardware implementations of clocks and algorithms. It was
121070 felt that mandating this format for POSIX.1b clocks and timers would unnecessarily
121071 burden the application developer with writing, possibly non-portable, multiple precision
121072 arithmetic packages to perform conversion between binary fractions and integral units
121073 such as nanoseconds, milliseconds, and so on.

121074 **Rationale for the Monotonic Clock**

121075 For those applications that use time services to achieve realtime behavior, changing the value of
121076 the clock on which these services rely may cause erroneous timing behavior. For these
121077 applications, it is necessary to have a monotonic clock which cannot run backwards, and which
121078 has a maximum clock jump that is required to be documented by the implementation.
121079 Additionally, it is desirable (but not required by POSIX.1-202x) that the monotonic clock
121080 increases its value uniformly. This clock should not be affected by changes to the system time;
121081 for example, to synchronize the clock with an external source or to account for leap seconds.
121082 Such changes would cause errors in the measurement of time intervals for those time services
121083 that use the absolute value of the clock.

121084 One could argue that by defining the behavior of time services when the value of a clock is
121085 changed, deterministic realtime behavior can be achieved. For example, one could specify that
121086 relative time services should be unaffected by changes in the value of a clock. However, there are
121087 time services that are based upon an absolute time, but that are essentially intended as relative
121088 time services. For example, `pthread_cond_timedwait()` uses an absolute time to allow it to wake
121089 up after the required interval despite spurious wakeups. Although sometimes the
121090 `pthread_cond_timedwait()` timeouts are absolute in nature, there are many occasions in which they
121091 are relative, and their absolute value is determined from the current time plus a relative time
121092 interval. In this latter case, if the clock changes while the thread is waiting, the wait interval will
121093 not be the expected length. If a `pthread_cond_timedwait()` function were created that would take a
121094 relative time, it would not solve the problem because to retain the intended “deadline” a thread
121095 would need to compensate for latency due to the spurious wakeup, and preemption between
121096 wakeup and the next wait.

121097 The solution is to create a new monotonic clock, whose value does not change except for the
121098 regular ticking of the clock, and use this clock for implementing the various relative timeouts
121099 that appear in the different POSIX interfaces, as well as allow `pthread_cond_timedwait()` to choose
121100 this new clock for its timeout. A new `clock_nanosleep()` function is created to allow an application
121101 to take advantage of this newly defined clock. Notice that the monotonic clock may be
121102 implemented using the same hardware clock as the system clock.

121103 Relative timeouts for `sigtimedwait()` and `aio_suspend()` have been redefined to use the monotonic
121104 clock, if present. The `alarm()` function has not been redefined, because the same effect but with
121105 better resolution can be achieved by creating a timer (for which the appropriate clock may be
121106 chosen).

121107 The `pthread_cond_timedwait()` function has been treated in a different way, compared to other
121108 functions with absolute timeouts, because it is used to wait for an event, and thus it may have a
121109 deadline, while the other timeouts are generally used as an error recovery mechanism, and for
121110 them the use of the monotonic clock is not so important. Since the desired timeout for the
121111 `pthread_cond_timedwait()` function may either be a relative interval or an absolute time of day
121112 deadline, a new initialization attribute has been created for condition variables to specify the
121113 clock that is used for measuring the timeout in a call to `pthread_cond_timedwait()`. In this way, if
121114 a relative timeout is desired, the monotonic clock will be used; if an absolute deadline is
121115 required instead, the `CLOCK_REALTIME` or another appropriate clock may be used. For
121116 condition variables, this capability is also available by passing `CLOCK_MONOTONIC` to the

121117 *pthread_cond_clockwait()* function. Similarly, CLOCK_MONOTONIC can be specified when
 121118 calling *pthread_mutex_clocklock()*, *pthread_rwlock_clockrdlock()*, *pthread_rwlock_clockwrlock()*, and
 121119 *sem_clockwait()*.

121120 It was later found necessary to add variants of almost all interfaces that accept absolute timeouts
 121121 that allow the clock to be specified. This is because, despite the claim in the previous paragraph,
 121122 it is not possible to safely use a CLOCK_REALTIME absolute timeout even to prevent errors
 121123 when the system clock is warped by a potentially large amount. A “safety timeout” of a minute
 121124 on a call to *pthread_mutex_timedlock()* could actually mean that the call would return
 121125 ETIMEDOUT early without acquiring the lock if the system clock is warped forwards
 121126 immediately prior to or during the call. On the other hand, a short timeout could end up being
 121127 arbitrarily long if the system clock is warped backwards immediately prior to or during the call.
 121128 These problems are solved by the new *clockwait* and *clocklock* variants of the existing *timedwait*
 121129 and *timedlock* functions. These variants accept an extra **clockid_t** parameter to indicate the clock
 121130 to be used for the wait. The clock ID is passed rather than using attributes as previously for
 121131 *pthread_cond_timedwait()* in order to allow the ISO/IEC 14882:2011 standard (C++11) and later to
 121132 be implemented correctly. C++ requires that the clock to use for the wait is not known until the
 121133 time of the wait call, so it cannot be supplied during creation. The new functions are
 121134 *pthread_cond_clockwait()*, *pthread_mutex_clocklock()*, *pthread_mutex_clockrdlock()*,
 121135 *pthread_mutex_clockwrlock()*, and *sem_clockwait()*. It is expected that *mq_clockreceive()* and
 121136 *mq_clocksend()* functions will be added in a future version of this standard.

121137 The *nanosleep()* function has not been modified with the introduction of the monotonic clock.
 121138 Instead, a new *clock_nanosleep()* function has been created, in which the desired clock may be
 121139 specified in the function call.

121140 • History of Resolution Issues

121141 Due to the shift from relative to absolute timeouts in IEEE Std 1003.1d-1999, the
 121142 amendments to the *sem_timedwait()*, *pthread_mutex_timedlock()*, *mq_timedreceive()*, and
 121143 *mq_timedsend()* functions of that standard have been removed. Those amendments
 121144 specified that CLOCK_MONOTONIC would be used for the (relative) timeouts if the
 121145 Monotonic Clock option was supported.

121146 Having these functions continue to be tied solely to CLOCK_MONOTONIC would not
 121147 work. Since the absolute value of a time value obtained from CLOCK_MONOTONIC is
 121148 unspecified, under the absolute timeouts interface, applications would behave differently
 121149 depending on whether the Monotonic Clock option was supported or not (because the
 121150 absolute value of the clock would have different meanings in either case).

121151 Two options were considered:

- 121152 1. Leave the current behavior unchanged, which specifies the CLOCK_REALTIME
 121153 clock for these (absolute) timeouts, to allow portability of applications between
 121154 implementations supporting or not the Monotonic Clock option.
- 121155 2. Modify these functions in the way that *pthread_cond_timedwait()* was modified to
 121156 allow a choice of clock, so that an application could use CLOCK_REALTIME when
 121157 it is trying to achieve an absolute timeout and CLOCK_MONOTONIC when it is
 121158 trying to achieve a relative timeout.

121159 It was decided that the features of CLOCK_MONOTONIC are not as critical to these
 121160 functions as they are to *pthread_cond_timedwait()*. The *pthread_cond_timedwait()* function is
 121161 given a relative timeout; the timeout may represent a deadline for an event. When these
 121162 functions are given relative timeouts, the timeouts are typically for error recovery
 121163 purposes and need not be so precise.

121164 Therefore, it was decided that these functions should be tied to CLOCK_REALTIME and

122004 **Supported Threads Functions**

122005 On POSIX-conforming systems, the following symbolic constants are always conforming:

122006 `_POSIX_READER_WRITER_LOCKS`
122007 `_POSIX_THREADS`

122008 Therefore, the following threads functions are always supported:

122009	<code>pthread_atfork()</code>	<code>pthread_kill()</code>
122010	<code>pthread_attr_destroy()</code>	<code>pthread_mutex_destroy()</code>
122011	<code>pthread_attr_getdetachstate()</code>	<code>pthread_mutex_init()</code>
122012	<code>pthread_attr_getguardsize()</code>	<code>pthread_mutex_lock()</code>
122013	<code>pthread_attr_getschedparam()</code>	<code>pthread_mutex_trylock()</code>
122014	<code>pthread_attr_init()</code>	<code>pthread_mutex_unlock()</code>
122015	<code>pthread_attr_setdetachstate()</code>	<code>pthread_mutexattr_destroy()</code>
122016	<code>pthread_attr_setguardsize()</code>	<code>pthread_mutexattr_getpshared()</code>
122017	<code>pthread_attr_setschedparam()</code>	<code>pthread_mutexattr_gettype()</code>
122018	<code>pthread_cancel()</code>	<code>pthread_mutexattr_init()</code>
122019	<code>pthread_cleanup_pop()</code>	<code>pthread_mutexattr_setpshared()</code>
122020	<code>pthread_cleanup_push()</code>	<code>pthread_mutexattr_settype()</code>
122021	<code>pthread_cond_broadcast()</code>	<code>pthread_once()</code>
122022	<code>pthread_cond_clockwait()</code>	<code>pthread_rwlock_destroy()</code>
122023	<code>pthread_cond_destroy()</code>	<code>pthread_rwlock_init()</code>
122024	<code>pthread_cond_init()</code>	<code>pthread_rwlock_rdlock()</code>
122025	<code>pthread_cond_signal()</code>	<code>pthread_rwlock_tryrdlock()</code>
122026	<code>pthread_cond_timedwait()</code>	<code>pthread_rwlock_trywrlock()</code>
122027	<code>pthread_cond_wait()</code>	<code>pthread_rwlock_unlock()</code>
122028	<code>pthread_condattr_destroy()</code>	<code>pthread_rwlock_wrlock()</code>
122029	<code>pthread_condattr_getpshared()</code>	<code>pthread_rwlockattr_destroy()</code>
122030	<code>pthread_condattr_init()</code>	<code>pthread_rwlockattr_getpshared()</code>
122031	<code>pthread_condattr_setpshared()</code>	<code>pthread_rwlockattr_init()</code>
122032	<code>pthread_create()</code>	<code>pthread_rwlockattr_setpshared()</code>
122033	<code>pthread_detach()</code>	<code>pthread_self()</code>
122034	<code>pthread_equal()</code>	<code>pthread_setcancelstate()</code>
122035	<code>pthread_exit()</code>	<code>pthread_setcanceltype()</code>
122036	<code>pthread_getspecific()</code>	<code>pthread_setspecific()</code>
122037	<code>pthread_join()</code>	<code>pthread_sigmask()</code>
122038	<code>pthread_key_create()</code>	<code>pthread_testcancel()</code>
122039	<code>pthread_key_delete()</code>	<code>sigwait()</code>

122084 `pthread_mutex_lock()`
 122085 `pthread_mutex_trylock()`
 122086 `pthread_mutex_unlock()`

122087 to take account of the new mutex attribute type and to specify behavior which was
 122088 declared as undefined in POSIX.1c. How a calling thread acquires or releases a mutex now
 122089 depends upon the mutex *type* attribute.

122090 The *type* attribute can have the following values:

122091 PTHREAD_MUTEX_NORMAL

122092 Basic mutex with no specific error checking built in. Does not report a deadlock error.

122093 PTHREAD_MUTEX_RECURSIVE

122094 Allows any thread to recursively lock a mutex. The mutex must be unlocked an equal
 122095 number of times to release the mutex.

122096 PTHREAD_MUTEX_ERRORCHECK

122097 Detects and reports simple usage errors; that is, an attempt to unlock a mutex that is
 122098 not locked by the calling thread or that is not locked at all, or an attempt to relock a
 122099 mutex the thread already owns.

122100 PTHREAD_MUTEX_DEFAULT

122101 The default mutex type. May be mapped to any of the above mutex types or may be
 122102 an implementation-defined type.

122103 *Normal* mutexes do not detect deadlock conditions; for example, a thread will hang if it
 122104 tries to relock a normal mutex that it already owns. Attempting to unlock a mutex locked
 122105 by another thread, or unlocking an unlocked mutex, results in undefined behavior. Normal
 122106 mutexes will usually be the fastest type of mutex available on a platform but provide the
 122107 least error checking.

122108 *Recursive* mutexes are useful for converting old code where it is difficult to establish clear
 122109 boundaries of synchronization. A thread can relock a recursive mutex without first
 122110 unlocking it. The relocking deadlock which can occur with normal mutexes cannot occur
 122111 with this type of mutex. However, multiple locks of a recursive mutex require the same
 122112 number of unlocks to release the mutex before another thread can acquire the mutex.
 122113 Furthermore, this type of mutex maintains the concept of an owner. Thus, a thread
 122114 attempting to unlock a recursive mutex which another thread has locked returns with an
 122115 error. A thread attempting to unlock a recursive mutex that is not locked returns with an
 122116 error. Never use a recursive mutex with condition variables because the implicit unlock
 122117 performed by `pthread_cond_clockwait()`, `pthread_cond_timedwait()`, or `pthread_cond_wait()`
 122118 will not actually release the mutex if it had been locked multiple times.

122119 *Errorcheck* mutexes provide error checking and are useful primarily as a debugging aid. A
 122120 thread attempting to relock an errorcheck mutex without first unlocking it returns with an
 122121 error. Again, this type of mutex maintains the concept of an owner. Thus, a thread
 122122 attempting to unlock an errorcheck mutex which another thread has locked returns with
 122123 an error. A thread attempting to unlock an errorcheck mutex that is not locked also returns
 122124 with an error. It should be noted that errorcheck mutexes will almost always be much
 122125 slower than normal mutexes due to the extra state checks performed.

122126 The default mutex type provides implementation-defined error checking. The default
 122127 mutex may be mapped to one of the other defined types or may be something entirely
 122128 different. This enables each vendor to provide the mutex semantics which the vendor feels
 122129 will be most useful to their target users. Most vendors will probably choose to make
 122130 normal mutexes the default so as to give applications the benefit of the fastest type of

126192 **Unsatisfied Requirements**

126193 Detailed control of common device classes, specifically magnetic tape, is not provided.

126194 **D.2.5 Bounded (Realtime) Response**

126195 The realtime signal functions *sigqueue()*, *sigtimedwait()*, and *sigwaitinfo()* provide queued signals
126196 and the prioritization of the handling of signals.

126197 The SCHED_FIFO, SCHED_SPORADIC, and SCHED_RR scheduling policies provide control
126198 over processor allocation.

126199 The semaphore functions *sem_clockwait()*, *sem_close()*, *sem_destroy()*, *sem_getvalue()*, *sem_init()*,
126200 *sem_open()*, *sem_post()*, *sem_timedwait()*, *sem_trywait()*, *sem_unlink()*, and *sem_wait()* provide
126201 high-performance synchronization.

126202 The memory management functions provide memory locking for control of memory allocation,
126203 file mapping for high performance, and shared memory for high-performance interprocess
126204 communication. The Message Passing option provides for interprocess communication without
126205 being dependent on shared memory.

126206 The timers functions *clock_getres()*, *clock_gettime()*, *clock_settime()*, *nanosleep()*, *timer_create()*,
126207 *timer_delete()*, *timer_getoverrun()*, *timer_gettime()*, and *timer_settime()* provide functionality to
126208 manipulate clocks and timers and include a high resolution function called *nanosleep()* with a
126209 finer resolution than the *sleep()* function.

126210 The timeout functions — *pthread_mutex_clocklock()*, *pthread_mutex_timedlock()*,
126211 *pthread_rwlock_clockrdlock()*, *pthread_rwlock_clockwrlock()*, *pthread_rwlock_timedrdlock()*,
126212 *pthread_rwlock_timedwrlock()*, *sem_clockwait()*, and *sem_timedwait()* — the Typed Memory
126213 Objects option and the Monotonic Clock option provide further facilities for applications to use
126214 to obtain predictable bounded response.

126215 **D.2.6 Operating System-Dependent Profile**

126216 POSIX.1-202x makes no distinction between text and binary files. The values of EXIT_SUCCESS
126217 and EXIT_FAILURE are further defined.

126218 **Unsatisfied Requirements**

126219 None known, but the ISO C standard may contain some additional options that could be
126220 specified.

126221 **D.2.7 I/O Interaction**

126222 POSIX.1-202x defines how each of the ISO C standard *stdio* functions interact with the POSIX.1
126223 operations, typically specifying the behavior in terms of POSIX.1 operations.

126793	<i>rintl()</i> , <i>round()</i> , <i>roundf()</i> , <i>roundl()</i> , <i>scalbln()</i> , <i>scalblnf()</i> , <i>scalblnl()</i> , <i>scalbn()</i> , <i>scalbnf()</i> ,	
126794	<i>scalbnl()</i> , <i>signbit()</i> , <i>sin()</i> , <i>sinf()</i> , <i>sinh()</i> , <i>sinhf()</i> , <i>sinhl()</i> , <i>sinl()</i> , <i>sqrt()</i> , <i>sqrtf()</i> , <i>sqrtl()</i> , <i>tan()</i> ,	
126795	<i>tanf()</i> , <i>tanh()</i> , <i>tanhf()</i> , <i>tanhL()</i> , <i>tanl()</i> , <i>tgamma()</i> , <i>tgammaf()</i> , <i>tgammaL()</i> , <i>trunc()</i> , <i>truncf()</i> ,	
126796	<i>truncl()</i>	
126797	POSIX_C_LANG_SUPPORT: General ISO C Library	
126798	<i>abs()</i> , <i>asctime()</i> , <i>atof()</i> , <i>atoi()</i> , <i>atol()</i> , <i>atoll()</i> , <i>bsearch()</i> , <i>calloc()</i> , <i>ctime()</i> , <i>difftime()</i> , <i>div()</i> ,	
126799	<i>feclearexcept()</i> , <i>fegetenv()</i> , <i>fegetexceptflag()</i> , <i>fegetround()</i> , <i>fehldexcept()</i> , <i>feraiseexcept()</i> ,	
126800	<i>fesetenv()</i> , <i>fesetexceptflag()</i> , <i>fesetround()</i> , <i>fetestexcept()</i> , <i>feupdateenv()</i> , <i>free()</i> , <i>gmtime()</i> ,	
126801	<i>imaxabs()</i> , <i>imaxdiv()</i> , <i>isalnum()</i> , <i>isalpha()</i> , <i>isblank()</i> , <i>iscntrl()</i> , <i>isdigit()</i> , <i>isgraph()</i> , <i>islower()</i> ,	
126802	<i>isprint()</i> , <i>ispunct()</i> , <i>isspace()</i> , <i>isupper()</i> , <i>isxdigit()</i> , <i>labs()</i> , <i>ldiv()</i> , <i>llabs()</i> , <i>lldiv()</i> , <i>localeconv()</i> ,	
126803	<i>localtime()</i> , <i>malloc()</i> , <i>memchr()</i> , <i>memcmp()</i> , <i>memcpy()</i> , <i>memmove()</i> , <i>memset()</i> , <i>mktime()</i> ,	
126804	<i>qsort()</i> , <i>rand()</i> , <i>realloc()</i> , <i>setlocale()</i> , <i>snprintf()</i> , <i>sprintf()</i> , <i>srand()</i> , <i>sscanf()</i> , <i>strcat()</i> , <i>strchr()</i> ,	
126805	<i>strcmp()</i> , <i>strcoll()</i> , <i>strcpy()</i> , <i>strcspn()</i> , <i>strerror()</i> , <i>strftime()</i> , <i>strlen()</i> , <i>strncat()</i> , <i>strncmp()</i> ,	
126806	<i>strncpy()</i> , <i>strpbrk()</i> , <i>strrchr()</i> , <i>strspn()</i> , <i>strstr()</i> , <i>strtod()</i> , <i>strtof()</i> , <i>strtoimax()</i> , <i>strtok()</i> , <i>strtol()</i> ,	
126807	<i>strtold()</i> , <i>strtoll()</i> , <i>strtoul()</i> , <i>strtoull()</i> , <i>strtoumax()</i> , <i>strxfrm()</i> , <i>time()</i> , <i>tolower()</i> , <i>toupper()</i> ,	
126808	<i>tzname</i> , <i>tzset()</i> , <i>va_arg()</i> , <i>va_copy()</i> , <i>va_end()</i> , <i>va_start()</i> , <i>vsprintf()</i> , <i>vsscanf()</i>	
126809	POSIX_C_LANG_SUPPORT_R: Thread-Safe General ISO C Library	
126810	<i>asctime_r()</i> , <i>ctime_r()</i> , <i>gmtime_r()</i> , <i>localtime_r()</i> , <i>qsort_r()</i> , <i>strerror_r()</i> , <i>strtok_r()</i>	+
126811	POSIX_C_LANG_WIDE_CHAR: Wide-Character ISO C Library	
126812	<i>btowc()</i> , <i>iswalnum()</i> , <i>iswalphabeta()</i> , <i>iswblank()</i> , <i>iswcntrl()</i> , <i>iswctype()</i> , <i>iswdigit()</i> , <i>iswgraph()</i> ,	
126813	<i>iswlower()</i> , <i>iswprint()</i> , <i>iswpunct()</i> , <i>iswspace()</i> , <i>iswupper()</i> , <i>iswxdigit()</i> , <i>mblen()</i> , <i>mbrlen()</i> ,	
126814	<i>mbrtowc()</i> , <i>mbsinit()</i> , <i>mbsrtowcs()</i> , <i>mbstowcs()</i> , <i>mbtowc()</i> , <i>swprintf()</i> , <i>swscanf()</i> , <i>towctrans()</i> ,	
126815	<i>towlower()</i> , <i>towupper()</i> , <i>vwprintf()</i> , <i>vwscanf()</i> , <i>wcrtomb()</i> , <i>wcscat()</i> , <i>wcschr()</i> , <i>wcscmp()</i> ,	
126816	<i>wcscoll()</i> , <i>wcscpy()</i> , <i>wcscspn()</i> , <i>wcsftime()</i> , <i>wcslen()</i> , <i>wcsncat()</i> , <i>wcsncmp()</i> , <i>wcsncpy()</i> ,	
126817	<i>wcspbrk()</i> , <i>wcsrchr()</i> , <i>wcrtombs()</i> , <i>wcsspn()</i> , <i>wcsstr()</i> , <i>wcstod()</i> , <i>wcstof()</i> , <i>wcstoimax()</i> ,	
126818	<i>wcstok()</i> , <i>wcstol()</i> , <i>wcstold()</i> , <i>wcstoll()</i> , <i>wcstombs()</i> , <i>wcstoul()</i> , <i>wcstoull()</i> , <i>wcstoumax()</i> ,	
126819	<i>wcsxfrm()</i> , <i>wctob()</i> , <i>wctomb()</i> , <i>wctrans()</i> , <i>wctype()</i> , <i>wmemchr()</i> , <i>wmemcmp()</i> , <i>wmemcpy()</i> ,	
126820	<i>wmemmove()</i> , <i>wmemset()</i>	
126821	POSIX_C_LANG_WIDE_CHAR_EXT: Extended Wide-Character ISO C Library	
126822	<i>mbsnrtowcs()</i> , <i>wcpcpy()</i> , <i>wcpncpy()</i> , <i>wcscasecmp()</i> , <i>wcsdup()</i> , <i>wcslcat()</i> , <i>wcslcpy()</i> ,	+
126823	<i>wcsncasecmp()</i> , <i>wcsnlen()</i> , <i>wcsnrtombs()</i>	
126824	POSIX_C_LIB_EXT: General C Library Extension	
126825	<i>fnmatch()</i> , <i>getentropy()</i> , <i>getopt()</i> , <i>getsubopt()</i> , <i>memmem()</i> , <i>optarg</i> , <i>opterr</i> , <i>optind</i> , <i>optopt</i> ,	+
126826	<i>reallocarray()</i> , <i>stpncpy()</i> , <i>strncasecmp()</i> , <i>strdup()</i> , <i>strfmon()</i> , <i>strlcat()</i> , <i>strlcpy()</i> ,	+
126827	<i>strncasecmp()</i> , <i>strndup()</i> , <i>strnlen()</i>	
126828	POSIX_CLOCK_SELECTION: Clock Selection	
126829	<i>clock_nanosleep()</i> , <i>pthread_condattr_getclock()</i> , <i>pthread_condattr_setclock()</i>	
126830	POSIX_DEVICE_IO: Device Input and Output	
126831	<i>FD_CLR()</i> , <i>FD_ISSET()</i> , <i>FD_SET()</i> , <i>FD_ZERO()</i> , <i>clearerr()</i> , <i>close()</i> , <i>fclose()</i> , <i>fdopen()</i> , <i>feof()</i> ,	
126832	<i>ferror()</i> , <i>fflush()</i> , <i>fgetc()</i> , <i>fgets()</i> , <i>fileno()</i> , <i>fopen()</i> , <i>fprintf()</i> , <i>fputc()</i> , <i>fputs()</i> , <i>fread()</i> , <i>freopen()</i> ,	
126833	<i>fscanf()</i> , <i>fwrite()</i> , <i>getc()</i> , <i>getchar()</i> , <i>open()</i> , <i>perror()</i> , <i>poll()</i> , <i>ppoll()</i> , <i>printf()</i> , <i>pread()</i> , <i>pselect()</i> ,	+
126834	<i>putc()</i> , <i>putchar()</i> , <i>puts()</i> , <i>pwrite()</i> , <i>read()</i> , <i>scanf()</i> , <i>select()</i> , <i>setbuf()</i> , <i>setvbuf()</i> , <i>stderr</i> , <i>stdin</i> ,	
126835	<i>stdout</i> , <i>ungetc()</i> , <i>vfprintf()</i> , <i>vfsprintf()</i> , <i>vprintf()</i> , <i>vscanf()</i> , <i>write()</i>	
126836	POSIX_DEVICE_IO_EXT: Extended Device Input and Output	
126837	<i>dprintf()</i> , <i>fmemopen()</i> , <i>open_memstream()</i> , <i>vdprintf()</i>	
126838	POSIX_DEVICE_SPECIFIC: General Terminal	
126839	<i>cfgetispeed()</i> , <i>cfgetospeed()</i> , <i>cfsetispeed()</i> , <i>cfsetospeed()</i> , <i>ctermid()</i> , <i>isatty()</i> , <i>tcdrain()</i> , <i>tcflow()</i> ,	
126840	<i>tcflush()</i> , <i>tcgetattr()</i> , <i>tcgetwinsize()</i> , <i>tcsendbreak()</i> , <i>tcsetattr()</i> , <i>tcsetwinsize()</i> , <i>ttyname()</i>	

126841	POSIX_DEVICE_SPECIFIC_R: Thread-Safe General Terminal	
126842	<i>ttyname_r()</i>	
126843	POSIX_DYNAMIC_LINKING: Dynamic Linking	
126844	<i>dladdr(), dlclose(), dlerror(), dlopen(), dlsym()</i>	+
126845	POSIX_FD_MGMT: File Descriptor Management	
126846	<i>dup(), dup2(), dup3(), fcntl(), fgetpos(), fseek(), fseeko(), fsetpos(), ftell(), ftello(), ftruncate(),</i>	
126847	<i>lseek(), rewind()</i>	
126848	POSIX_FIFO: FIFO	
126849	<i>mkfifo()</i>	
126850	POSIX_FIFO_FD: FIFO File Descriptor Routines	
126851	<i>mkfifoat(), mknodat()</i>	
126852	POSIX_FILE_ATTRIBUTES: File Attributes	
126853	<i>chmod(), chown(), fchmod(), fchown(), umask()</i>	
126854	POSIX_FILE_ATTRIBUTES_FD: File Attributes File Descriptor Routines	
126855	<i>fchmodat(), fchownat()</i>	
126856	POSIX_FILE_LOCKING: Thread-Safe Stdio Locking	
126857	<i>flockfile(), frylockfile(), funlockfile(), getc_unlocked(), getchar_unlocked(), putc_unlocked(),</i>	
126858	<i>putchar_unlocked()</i>	
126859	POSIX_FILE_SYSTEM: File System	
126860	<i>access(), chdir(), closedir(), creat(), fchdir(), fpathconf(), fstat(), fstatvfs(), getcwd(), link(),</i>	
126861	<i>mkdir(), mkostemp(), mkstemp(), opendir(), pathconf(), posix_getdents(), readdir(), remove(),</i>	+
126862	<i>rename(), rewinddir(), rmdir(), stat(), statvfs(), tmpfile(), tmpnam(), truncate(), unlink()</i>	
126863	POSIX_FILE_SYSTEM_EXT: File System Extensions	
126864	<i>alphasort(), dirfd(), getdelim(), getline(), mkdtemp(), scandir()</i>	
126865	POSIX_FILE_SYSTEM_FD: File System File Descriptor Routines	
126866	<i>faccessat(), fdopendir(), fstatat(), linkat(), mkdirat(), openat(), renameat(), unlinkat(),</i>	
126867	<i>utimensat()</i>	
126868	POSIX_FILE_SYSTEM_GLOB: File System Glob Expansion	
126869	<i>glob(), globfree()</i>	
126870	POSIX_FILE_SYSTEM_R: Thread-Safe File System	
126871	<i>readdir_r()</i>	
126872	POSIX_I18N: Internationalization	
126873	<i>catclose(), catgets(), catopen(), iconv(), iconv_close(), iconv_open(), nl_langinfo()</i>	
126874	POSIX_JOB_CONTROL: Job Control	
126875	<i>setpgid(), tcgetpgrp(), tcsetpgrp(), tcgetsid()</i>	
126876	POSIX_MAPPED_FILES: Memory Mapped Files	
126877	<i>mmap(), munmap()</i>	
126878	POSIX_MEMORY_PROTECTION: Memory Protection	
126879	<i>mprotect()</i>	
126880	POSIX_MULTI_CONCURRENT_LOCALES: Multiple Concurrent Locales	
126881	<i>duplocale(), freelocale(), getlocalename_l(), isalnum_l(), isalpha_l(), isblank_l(), iscntrl_l(),</i>	+
126882	<i>isdigit_l(), isgraph_l(), islower_l(), isprint_l(), ispunct_l(), isspace_l(), isupper_l(),</i>	
126883	<i>iswalnum_l(), iswalpha_l(), iswblank_l(), iswcntrl_l(), iswctype_l(), iswdigit_l(), iswgraph_l(),</i>	
126884	<i>iswlower_l(), iswprint_l(), iswpunct_l(), iswspace_l(), iswupper_l(), iswxdigit_l(), isxdigit_l(),</i>	

126885	<i>newlocale()</i> , <i>strcasemp_l()</i> , <i>strcoll_l()</i> , <i>strfmon_l()</i> , <i>strncasemp_l()</i> , <i>strxfrm_l()</i> , <i>tolower_l()</i> ,	
126886	<i>toupper_l()</i> , <i>towctrans_l()</i> , <i>towlower()</i> , <i>towupper()</i> , <i>uselocale()</i> , <i>wscasemp_l()</i> , <i>wscoll_l()</i> ,	
126887	<i>wscncasemp_l()</i> , <i>wcsxfrm_l()</i> , <i>wctrans_l()</i> , <i>wctype_l()</i>	
126888	POSIX_MULTI_PROCESS: Multiple Processes	
126889	<i>_Exit()</i> , <i>_exit()</i> , <i>assert()</i> , <i>atexit()</i> , <i>clock()</i> , <i>execl()</i> , <i>execle()</i> , <i>execlp()</i> , <i>execv()</i> , <i>execve()</i> , <i>execvp()</i> ,	
126890	<i>exit()</i> , <i>fork()</i> , <i>getpgrp()</i> , <i>getpgid()</i> , <i>getpid()</i> , <i>getppid()</i> , <i>getsid()</i> , <i>setsid()</i> , <i>sleep()</i> , <i>times()</i> , <i>wait()</i> ,	
126891	<i>waitid()</i> , <i>waitpid()</i>	
126892	POSIX_MULTI_PROCESS_FD: Multiple Processes File Descriptor Routines	
126893	<i>fexecve()</i>	
126894	POSIX_NETWORKING: Networking	
126895	<i>accept()</i> , <i>accept4()</i> , <i>bind()</i> , <i>connect()</i> , <i>endhostent()</i> , <i>endnetent()</i> , <i>endprotoent()</i> , <i>endservent()</i> ,	
126896	<i>freaddrinfo()</i> , <i>gai_strerror()</i> , <i>getaddrinfo()</i> , <i>gethostent()</i> , <i>gethostname()</i> , <i>getnameinfo()</i> ,	
126897	<i>getnetbyaddr()</i> , <i>getnetbyname()</i> , <i>getnetent()</i> , <i>getpeername()</i> , <i>getprotobyname()</i> ,	
126898	<i>getprotobynumber()</i> , <i>getprotoent()</i> , <i>getserbyname()</i> , <i>getservbyport()</i> , <i>getservent()</i> ,	
126899	<i>getsockname()</i> , <i>getsockopt()</i> , <i>htonl()</i> , <i>htons()</i> , <i>if_freenameindex()</i> , <i>if_indextoname()</i> ,	
126900	<i>if_nameindex()</i> , <i>if_nametoindex()</i> , <i>inet_addr()</i> , <i>inet_ntoa()</i> , <i>inet_ntop()</i> , <i>inet_pton()</i> , <i>listen()</i> ,	
126901	<i>ntohl()</i> , <i>ntohs()</i> , <i>recv()</i> , <i>recvfrom()</i> , <i>recvmsg()</i> , <i>send()</i> , <i>sendmsg()</i> , <i>sendto()</i> , <i>sethostent()</i> ,	
126902	<i>setnetent()</i> , <i>setprotoent()</i> , <i>setservent()</i> , <i>setsockopt()</i> , <i>shutdown()</i> , <i>socket()</i> , <i>socketatmark()</i> ,	
126903	<i>socketpair()</i>	
126904	POSIX_PIPE: Pipe	
126905	<i>pipe()</i> , <i>pipe2()</i>	
126906	POSIX_ROBUST_MUTEXES: Robust Mutexes	
126907	<i>pthread_mutex_consistent()</i> , <i>pthread_mutexattr_getrobust()</i> , <i>pthread_mutexattr_setrobust()</i>	
126908	POSIX_REALTIME_SIGNALS: Realtime Signals	
126909	<i>sigqueue()</i> , <i>sigtimedwait()</i> , <i>sigwaitinfo()</i>	
126910	POSIX_REGEX: Regular Expressions	
126911	<i>regcomp()</i> , <i>regerror()</i> , <i>regexec()</i> , <i>regfree()</i>	
126912	POSIX_RW_LOCKS: Reader Writer Locks	
126913	<i>pthread_rwlock_clockrdlock()</i> , <i>pthread_rwlock_clockwrlock()</i> , <i>pthread_rwlock_destroy()</i> ,	+
126914	<i>pthread_rwlock_init()</i> , <i>pthread_rwlock_rdlock()</i> , <i>pthread_rwlock_timedrdlock()</i> ,	
126915	<i>pthread_rwlock_timedwrlock()</i> , <i>pthread_rwlock_tryrdlock()</i> , <i>pthread_rwlock_trywrlock()</i> ,	
126916	<i>pthread_rwlock_unlock()</i> , <i>pthread_rwlock_wrlock()</i> , <i>pthread_rwlockattr_destroy()</i> ,	
126917	<i>pthread_rwlockattr_init()</i> , <i>pthread_rwlockattr_getpshared()</i> , <i>pthread_rwlockattr_setpshared()</i>	
126918	POSIX_SEMAPHORES: Semaphores	
126919	<i>sem_clockwait()</i> , <i>sem_close()</i> , <i>sem_destroy()</i> , <i>sem_getvalue()</i> , <i>sem_init()</i> , <i>sem_open()</i> ,	+
126920	<i>sem_post()</i> , <i>sem_timedwait()</i> , <i>sem_trywait()</i> , <i>sem_unlink()</i> , <i>sem_wait()</i>	
126921	POSIX_SHELL_FUNC: Shell and Utilities	
126922	<i>pclose()</i> , <i>popen()</i> , <i>system()</i> , <i>wordexp()</i> , <i>wordfree()</i>	
126923	POSIX_SIGNAL_JUMP: Signal Jump Functions	
126924	<i>siglongjmp()</i> , <i>sigsetjmp()</i>	
126925	POSIX_SIGNALS: Signals	
126926	<i>abort()</i> , <i>alarm()</i> , <i>kill()</i> , <i>pause()</i> , <i>raise()</i> , <i>sigaction()</i> , <i>sigaddset()</i> , <i>sigdelset()</i> , <i>sigemptyset()</i> ,	
126927	<i>sigfillset()</i> , <i>sigismember()</i> , <i>signal()</i> , <i>sigpending()</i> , <i>sigprocmask()</i> , <i>sigsuspend()</i> , <i>sigwait()</i>	
126928	POSIX_SIGNALS_EXT: Extended Signals	
126929	<i>psignal()</i> , <i>psiginfo()</i> , <i>sig2str()</i> , <i>str2sig()</i> , <i>strsignal()</i>	+

126930	POSIX_SINGLE_PROCESS: Single Process
126931	<i>confstr()</i> , <i>environ</i> , <i>errno</i> , <i>getenv()</i> , <i>setenv()</i> , <i>sysconf()</i> , <i>uname()</i> , <i>unsetenv()</i>
126932	POSIX_SPIN_LOCKS: Spin Locks
126933	<i>pthread_spin_destroy()</i> , <i>pthread_spin_init()</i> , <i>pthread_spin_lock()</i> , <i>pthread_spin_trylock()</i> ,
126934	<i>pthread_spin_unlock()</i>
126935	POSIX_SYMBOLIC_LINKS: Symbolic Links
126936	<i>lchown()</i> , ¹¹ <i>lstat()</i> , <i>readlink()</i> , <i>symlink()</i>
126937	POSIX_SYMBOLIC_LINKS_FD: Symbolic Links File Descriptor Routines
126938	<i>readlinkat()</i> , <i>symlinkat()</i>
126939	POSIX_SYSTEM_DATABASE: System Database
126940	<i>getgrgid()</i> , <i>getgrnam()</i> , <i>getpwnam()</i> , <i>getpwuid()</i>
126941	POSIX_SYSTEM_DATABASE_R: Thread-Safe System Database
126942	<i>getgrgid_r()</i> , <i>getgrnam_r()</i> , <i>getpwnam_r()</i> , <i>getpwuid_r()</i>
126943	POSIX_THREADS_BASE: Base Threads
126944	<i>pthread_atfork()</i> , <i>pthread_attr_destroy()</i> , <i>pthread_attr_getdetachstate()</i> ,
126945	<i>pthread_attr_getschedparam()</i> , <i>pthread_attr_init()</i> , <i>pthread_attr_setdetachstate()</i> ,
126946	<i>pthread_attr_setschedparam()</i> , <i>pthread_cancel()</i> , <i>pthread_cleanup_pop()</i> , <i>pthread_cleanup_push()</i> ,
126947	<i>pthread_cond_broadcast()</i> , <i>pthread_cond_clockwait()</i> , <i>pthread_cond_destroy()</i> , +
126948	<i>pthread_cond_init()</i> , <i>pthread_cond_signal()</i> , <i>pthread_cond_timedwait()</i> , <i>pthread_cond_wait()</i> ,
126949	<i>pthread_condattr_destroy()</i> , <i>pthread_condattr_init()</i> , <i>pthread_create()</i> , <i>pthread_detach()</i> ,
126950	<i>pthread_equal()</i> , <i>pthread_exit()</i> , <i>pthread_getspecific()</i> , <i>pthread_join()</i> , <i>pthread_key_create()</i> ,
126951	<i>pthread_key_delete()</i> , <i>pthread_kill()</i> , <i>pthread_mutex_clocklock()</i> , <i>pthread_mutex_destroy()</i> , +
126952	<i>pthread_mutex_init()</i> , <i>pthread_mutex_lock()</i> , <i>pthread_mutex_timedlock()</i> ,
126953	<i>pthread_mutex_trylock()</i> , <i>pthread_mutex_unlock()</i> , <i>pthread_mutexattr_destroy()</i> ,
126954	<i>pthread_mutexattr_init()</i> , <i>pthread_once()</i> , <i>pthread_self()</i> , <i>pthread_setcancelstate()</i> ,
126955	<i>pthread_setcanceltype()</i> , <i>pthread_setspecific()</i> , <i>pthread_sigmask()</i> , <i>pthread_testcancel()</i>
126956	POSIX_THREADS_EXT: Extended Threads
126957	<i>pthread_attr_getguardsize()</i> , <i>pthread_attr_setguardsize()</i> , <i>pthread_mutexattr_gettype()</i> ,
126958	<i>pthread_mutexattr_settype()</i>
126959	POSIX_TIMERS: Timers
126960	<i>clock_getres()</i> , <i>clock_gettime()</i> , <i>clock_settime()</i> , <i>nanosleep()</i> , <i>timer_create()</i> , <i>timer_delete()</i> ,
126961	<i>timer_getoverrun()</i> , <i>timer_gettime()</i> , <i>timer_settime()</i>
126962	POSIX_USER_GROUPS: User and Group
126963	<i>getegid()</i> , <i>geteuid()</i> , <i>getgid()</i> , <i>getgroups()</i> , <i>getlogin()</i> , <i>getuid()</i> , <i>setegid()</i> , <i>seteuid()</i> , <i>setgid()</i> ,
126964	<i>setuid()</i>
126965	POSIX_USER_GROUPS_R: Thread-Safe User and Group
126966	<i>getlogin_r()</i>
126967	POSIX_WIDE_CHAR_DEVICE_IO: Device Input and Output
126968	<i>fgetwc()</i> , <i>fgetwts()</i> , <i>fputwc()</i> , <i>fputwts()</i> , <i>fwide()</i> , <i>fwprintf()</i> , <i>fwscanf()</i> , <i>getwc()</i> , <i>getwchar()</i> ,
126969	<i>putwc()</i> , <i>putwchar()</i> , <i>ungetwc()</i> , <i>vwprintf()</i> , <i>vwscanf()</i> , <i>vwprintf()</i> , <i>vwscanf()</i> , <i>wprintf()</i> ,
126970	<i>wscanf()</i>
126971	XSI_C_LANG_SUPPORT: XSI General C Library
126972	<i>a64l()</i> , <i>daylight</i> , <i>drand48()</i> , <i>erand48()</i> , <i>ffs()</i> , <i>ffsl()</i> , <i>ffsll()</i> , <i>getdate()</i> , <i>hcreate()</i> , <i>hdestroy()</i> ,
126973	<i>hsearch()</i> , <i>initstate()</i> , <i>insque()</i> , <i>rand48()</i> , <i>l64a()</i> , <i>lcong48()</i> , <i>lfind()</i> , <i>lrand48()</i> , <i>lsearch()</i> ,
126974	<i>memccpy()</i> , <i>mrnd48()</i> , <i>nrnd48()</i> , <i>random()</i> , <i>remque()</i> , <i>seed48()</i> , <i>setstate()</i> , <i>siggam</i> ,

126975 11. The *lchown()* function also depends on POSIX_FILE_ATTRIBUTES.