Table of Contents

NAME

__iconv_get_list, **__iconv_free_list** - retrieving a list of character encodings supported by iconv(3)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <iconv.h>

int
__iconv_get_list(*char ***names*, *size_t count*, *bool paired*);

void
__iconv_free_list(*char **names*, *size_t count*);

DESCRIPTION

The **__iconv_get_list**() function obtains a list of character encodings that are supported by the iconv(3) call. The list of the encoding names will be stored in *names* and the number of the entries is stored in *count*. If the *paired* variable is true, the list will be arranged into canonical/alias name pairs.

The **__iconv_free_list**() function is to free the allocated memory during the call of **__iconv_get_list**().

RETURN VALUES

Upon successful completion **__iconv_get_list**() returns 0 and set the *names* and *count* arguments. Otherwise, -1 is returned and errno is set to indicate the error.

SEE ALSO

iconv(3), iconvlist(3)

STANDARDS

The **__iconv_get_list** and **__iconv_free_list** functions are non-standard interfaces, which appeared in the implementation of the Citrus Project. The iconv implementation of the Citrus Project was adopted in FreeBSD 9.0.

AUTHORS

This manual page was written by Gabor Kovesdan <*gabor@FreeBSD.org*>.

NAME

syscall, **__syscall** - indirect system call

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/syscall.h>
#include <unistd.h>

int
syscall(*int number*, ...);

off_t
__syscall(*quad_t number*, ...);

DESCRIPTION

The **syscall**() function performs the system call whose assembly language interface has the specified *number* with the specified arguments. Symbolic constants for system calls can be found in the header file <*sys/syscall.h*>. The **__syscall**() form should be used when one or more of the arguments is a 64-bit argument to ensure that argument alignment is correct. This system call is useful for testing new system calls that do not have entries in the C library.

RETURN VALUES

The return values are defined by the system call being invoked. In general, a 0 return value indicates success. A -1 return value indicates an error, and an error code is stored in *errno*.

HISTORY

The **syscall**() function appeared in 4.0BSD.

BUGS

There is no way to simulate system calls that have multiple return values such as pipe(2).

NAME

_exit - terminate the calling process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

void
_exit(*int status*);

DESCRIPTION

The **_exit**() system call terminates a process with the following consequences:

- All of the descriptors open in the calling process are closed. This may entail delays, for example, waiting for output to drain; a process in this state may not be killed, as it is already dying.

- If the parent process of the calling process has an outstanding wait(2) call or catches the SIGCHLD signal, it is notified of the calling process's termination and the *status* is set as defined by wait(2).

- The parent process-ID of all of the calling process's existing child processes are set to the process-ID of the calling process's reaper; the reaper (normally the initialization process) inherits each of these processes (see procctl(2), init(8) and the *DEFINITIONS* section of intro(2)).

- If the termination of the process causes any process group to become orphaned (usually because the parents of all members of the group have now exited; see "orphaned process group" in intro(2)), and if any member of the orphaned group is stopped, the SIGHUP signal and the SIGCONT signal are sent to all members of the newly-orphaned process group.

- If the process is a controlling process (see intro(2)), the SIGHUP signal is sent to the foreground process group of the controlling terminal, and all current access to the controlling terminal is revoked.

Most C programs call the library routine exit(3), which flushes buffers, closes streams, unlinks temporary files, etc., before calling **_exit**().

RETURN VALUES

The **_exit**() system call can never return.

SEE ALSO

fork(2), sigaction(2), wait(2), exit(3), init(8)

STANDARDS

The **_exit**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **_exit**() function appeared in Version 7 AT&T UNIX.

NAME

sigsetjmp, **siglongjmp**, **setjmp**, **longjmp**, **_setjmp**, **_longjmp**, **longjmperror** - non-local jumps

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <setjmp.h>

int
sigsetjmp(*sigjmp_buf env*, *int savemask*);

void
siglongjmp(*sigjmp_buf env*, *int val*);

int
setjmp(*jmp_buf env*);

void
longjmp(*jmp_buf env*, *int val*);

int
_setjmp(*jmp_buf env*);

void
_longjmp(*jmp_buf env*, *int val*);

void
longjmperror(*void*);

DESCRIPTION

The **sigsetjmp**(), **setjmp**(), and **_setjmp**() functions save their calling environment in *env*. Each of these functions returns 0.

The corresponding **longjmp**() functions restore the environment saved by their most recent respective invocations of the **setjmp**() function. They then return so that program execution continues as if the corresponding invocation of the **setjmp**() call had just returned the value specified by *val*, instead of 0.

Pairs of calls may be intermixed, i.e., both **sigsetjmp**() and **siglongjmp**() and **setjmp**() and **longjmp**() combinations may be used in the same program, however, individual calls may not, e.g. the *env*

argument to **setjmp**() may not be passed to **siglongjmp**().

The **longjmp**() routines may not be called after the routine which called the **setjmp**() routines returns.

All accessible objects have values as of the time **longjmp**() routine was called, except that the values of objects of automatic storage invocation duration that do not have the *volatile* type and have been changed between the **setjmp**() invocation and **longjmp**() call are indeterminate.

The **setjmp**()/**longjmp**() pairs save and restore the signal mask while **_setjmp**()/**_longjmp**() pairs save and restore only the register set and the stack. (See **sigprocmask**(*2*).)

The **sigsetjmp**()/**siglongjmp**() function pairs save and restore the signal mask if the argument *savemask* is non-zero, otherwise only the register set and the stack are saved.

ERRORS

If the contents of the *env* are corrupted, or correspond to an environment that has already returned, the **longjmp**() routine calls the routine **longjmperror**(*3*). If **longjmperror**() returns the program is aborted (see abort(3)). The default version of **longjmperror**() prints the message "longjmp botch" to standard error and returns. User programs wishing to exit more gracefully should write their own versions of **longjmperror**().

SEE ALSO

sigaction(2), sigaltstack(2), signal(3)

STANDARDS

The **setjmp**() and **longjmp**() functions conform to ISO/IEC 9899:1990 ("ISO C90"). The **sigsetjmp**() and **siglongjmp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1").

NAME

_umtx_op - interface for implementation of userspace threading synchronization primitives

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/umtx.h>

int
_umtx_op(*void *obj*, *int op*, *u_long val*, *void *uaddr*, *void *uaddr2*);

DESCRIPTION

The **_umtx_op**() system call provides kernel support for userspace implementation of the threading synchronization primitives. The 1:1 Threading Library (libthr, -lthr) uses the syscall to implement IEEE Std 1003.1-2001 ("POSIX.1") pthread locks, like mutexes, condition variables and so on.

STRUCTURES

The operations, performed by the **_umtx_op**() syscall, operate on userspace objects which are described by the following structures. Reserved fields and paddings are omitted. All objects require ABI-mandated alignment, but this is not currently enforced consistently on all architectures.

The following flags are defined for flag fields of all structures:

USYNC_PROCESS_SHARED
: Allow selection of the process-shared sleep queue for the thread sleep container, when the lock ownership cannot be granted immediately, and the operation must sleep. The process-shared or process-private sleep queue is selected based on the attributes of the memory mapping which contains the first byte of the structure, see mmap(2). Otherwise, if the flag is not specified, the process-private sleep queue is selected regardless of the memory mapping attributes, as an optimization.

  See the *SLEEP QUEUES* subsection below for more details on sleep queues.

Mutex

```
struct umutex {
        volatile lwpid_t m_owner;
        uint32_t         m_flags;
```

```
        uint32_t        m_ceilings[2];
        uintptr_t       m_rb_lnk;
};
```

The m_owner field is the actual lock. It contains either the thread identifier of the lock owner in the locked state, or zero when the lock is unowned. The highest bit set indicates that there is contention on the lock. The constants are defined for special values:

UMUTEX_UNOWNED
: Zero, the value stored in the unowned lock.

UMUTEX_CONTESTED
: The contenion indicator.

UMUTEX_RB_OWNERDEAD
: A thread owning the robust mutex terminated. The mutex is in unlocked state.

UMUTEX_RB_NOTRECOV
: The robust mutex is in a non-recoverable state. It cannot be locked until reinitialized.

The m_flags field may contain the following umutex-specific flags, in addition to the common flags:

UMUTEX_PRIO_INHERIT
: Mutex implements *Priority Inheritance* protocol.

UMUTEX_PRIO_PROTECT
: Mutex implements *Priority Protection* protocol.

UMUTEX_ROBUST
: Mutex is robust, as described in the *ROBUST UMUTEXES* section below.

UMUTEX_NONCONSISTENT
: Robust mutex is in a transient non-consistent state. Not used by kernel.

In the manual page, mutexes not having UMUTEX_PRIO_INHERIT and UMUTEX_PRIO_PROTECT flags set, are called normal mutexes. Each type of mutex (normal, priority-inherited, and priority-protected) has a separate sleep queue associated with the given key.

For priority protected mutexes, the m_ceilings array contains priority ceiling values. The m_ceilings[0] is the ceiling value for the mutex, as specified by IEEE Std 1003.1-2008 ("POSIX.1") for the *Priority Protected* mutex protocol. The m_ceilings[1] is used only for the unlock of a priority protected mutex, when unlock is done in an order other than the reversed lock order. In this case, m_ceilings[1] must contain the ceiling value for the last locked priority protected mutex, for proper priority reassignment. If, instead, the unlocking mutex was the last priority propagated mutex locked by the thread, m_ceilings[1] should contain -1. This is required because kernel does not maintain the ordered lock list.

Condition variable

```
struct ucond {
        volatile uint32_t c_has_waiters;
        uint32_t        c_flags;
        uint32_t        c_clockid;
};
```

A non-zero c_has_waiters value indicates that there are in-kernel waiters for the condition, executing the UMTX_OP_CV_WAIT request.

The c_flags field contains flags. Only the common flags (USYNC_PROCESS_SHARED) are defined for ucond.

The c_clockid member provides the clock identifier to use for timeout, when the UMTX_OP_CV_WAIT request has both the CVWAIT_CLOCKID flag and the timeout specified. Valid clock identifiers are a subset of those for clock_gettime(2):

- CLOCK_MONOTONIC
- CLOCK_MONOTONIC_FAST
- CLOCK_MONOTONIC_PRECISE
- CLOCK_PROF
- CLOCK_REALTIME
- CLOCK_REALTIME_FAST
- CLOCK_REALTIME_PRECISE
- CLOCK_SECOND
- CLOCK_UPTIME
- CLOCK_UPTIME_FAST
- CLOCK_UPTIME_PRECISE
- CLOCK_VIRTUAL

Reader/writer lock

```
struct urwlock {
        volatile int32_t rw_state;
        uint32_t        rw_flags;
        uint32_t        rw_blocked_readers;
        uint32_t        rw_blocked_writers;
};
```

The rw_state field is the actual lock. It contains both the flags and counter of the read locks which were granted. Names of the rw_state bits are following:

URWLOCK_WRITE_OWNER
: Write lock was granted.

URWLOCK_WRITE_WAITERS
: There are write lock waiters.

URWLOCK_READ_WAITERS
: There are read lock waiters.

URWLOCK_READER_COUNT(c)
: Returns the count of currently granted read locks.

At any given time there may be only one thread to which the writer lock is granted on the *struct rwlock*, and no threads are granted read lock. Or, at the given time, up to URWLOCK_MAX_READERS threads may be granted the read lock simultaneously, but write lock is not granted to any thread.

The following flags for the rw_flags member of *struct urwlock* are defined, in addition to the common flags:

URWLOCK_PREFER_READER
: If specified, immediately grant read lock requests when urwlock is already read-locked, even in presence of unsatisfied write lock requests. By default, if there is a write lock waiter, further read requests are not granted, to prevent unfair write lock waiter starvation.

The rw_blocked_readers and rw_blocked_writers members contain the count of threads which are sleeping in kernel, waiting for the associated request type to be granted. The fields are used by kernel to update the URWLOCK_READ_WAITERS and URWLOCK_WRITE_WAITERS flags of the rw_state lock after requesting thread was

woken up.

Semaphore

```
struct _usem2 {
        volatile uint32_t _count;
        uint32_t          _flags;
};
```

The _count word represents a counting semaphore. A non-zero value indicates an unlocked (posted) semaphore, while zero represents the locked state. The maximal supported semaphore count is USEM_MAX_COUNT.

The _count word, besides the counter of posts (unlocks), also contains the USEM_HAS_WAITERS bit, which indicates that locked semaphore has waiting threads.

The USEM_COUNT() macro, applied to the _count word, returns the current semaphore counter, which is the number of posts issued on the semaphore.

The following bits for the _flags member of *struct _usem2* are defined, in addition to the common flags:

USEM_NAMED
Flag is ignored by kernel.

Timeout parameter

```
struct _umtx_time {
        struct timespec _timeout;
        uint32_t        _flags;
        uint32_t        _clockid;
};
```

Several **_umtx_op**() operations allow the blocking time to be limited, failing the request if it cannot be satisfied in the specified time period. The timeout is specified by passing either the address of *struct timespec*, or its extended variant, *struct _umtx_time*, as the *uaddr2* argument of **_umtx_op**(). They are distinguished by the *uaddr* value, which must be equal to the size of the structure pointed to by *uaddr2*, casted to *uintptr_t*.

The _timeout member specifies the time when the timeout should occur. Legal values for

clock identifier _clockid are shared with the *clock_id* argument to the clock_gettime(2) function, and use the same underlying clocks. The specified clock is used to obtain the current time value. Interval counting is always performed by the monotonic wall clock.

The _flags argument allows the following flags to further define the timeout behaviour:

UMTX_ABSTIME
: The _timeout value is the absolute time. The thread will be unblocked and the request failed when specified clock value is equal or exceeds the _timeout.

  If the flag is absent, the timeout value is relative, that is the amount of time, measured by the monotonic wall clock from the moment of the request start.

SLEEP QUEUES

When a locking request cannot be immediately satisfied, the thread is typically put to *sleep*, which is a non-runnable state terminated by the *wake* operation. Lock operations include a *try* variant which returns an error rather than sleeping if the lock cannot be obtained. Also, **_umtx_op**() provides requests which explicitly put the thread to sleep.

Wakes need to know which threads to make runnable, so sleeping threads are grouped into containers called *sleep queues*. A sleep queue is identified by a key, which for **_umtx_op**() is defined as the physical address of some variable. Note that the *physical* address is used, which means that same variable mapped multiple times will give one key value. This mechanism enables the construction of *process-shared* locks.

A related attribute of the key is shareability. Some requests always interpret keys as private for the current process, creating sleep queues with the scope of the current process even if the memory is shared. Others either select the shareability automatically from the mapping attributes, or take additional input as the USYNC_PROCESS_SHARED common flag. This is done as optimization, allowing the lock scope to be limited regardless of the kind of backing memory.

Only the address of the start byte of the variable specified as key is important for determining corresponding sleep queue. The size of the variable does not matter, so, for example, sleep on the same address interpeted as *uint32_t* and *long* on a little-endian 64-bit platform would collide.

The last attribute of the key is the object type. The sleep queue to which a sleeping thread is assigned is an individual one for simple wait requests, mutexes, rwlocks, condvars and other primitives, even when the physical address of the key is same.

When waking up a limited number of threads from a given sleep queue, the highest priority threads that

have been blocked for the longest on the queue are selected.

ROBUST UMUTEXES

The *robust umutexes* are provided as a substrate for a userspace library to implement POSIX robust mutexes. A robust umutex must have the UMUTEX_ROBUST flag set.

On thread termination, the kernel walks two lists of mutexes. The two lists head addresses must be provided by a prior call to UMTX_OP_ROBUST_LISTS request. The lists are singly-linked. The link to next element is provided by the m_rb_lnk member of the *struct umutex*.

Robust list processing is aborted if the kernel finds a mutex with any of the following conditions:

- the UMUTEX_ROBUST flag is not set
- not owned by the current thread, except when the mutex is pointed to by the robust_inactive member of the *struct umtx_robust_lists_params*, registered for the current thread
- the combination of mutex flags is invalid
- read of the umutex memory faults
- the list length limit described in libthr(3) is reached.

Every mutex in both lists is unlocked as if the UMTX_OP_MUTEX_UNLOCK request is performed on it, but instead of the UMUTEX_UNOWNED value, the m_owner field is written with the UMUTEX_RB_OWNERDEAD value. When a mutex in the UMUTEX_RB_OWNERDEAD state is locked by kernel due to the UMTX_OP_MUTEX_TRYLOCK and UMTX_OP_MUTEX_LOCK requests, the lock is granted and EOWNERDEAD error is returned.

Also, the kernel handles the UMUTEX_RB_NOTRECOV value of the m_owner field specially, always returning the ENOTRECOVERABLE error for lock attempts, without granting the lock.

OPERATIONS

The following operations, requested by the *op* argument to the function, are implemented:

UMTX_OP_WAIT

Wait. The arguments for the request are:

obj Pointer to a variable of type *long*.

val Current value of the *obj.

The current value of the variable pointed to by the *obj* argument is compared with the *val*. If they are equal, the requesting thread is put to interruptible sleep until woken up or the optionally specified timeout expires.

The comparison and sleep are atomic. In other words, if another thread writes a new value to *obj and then issues UMTX_OP_WAKE, the request is guaranteed to not miss the wakeup, which might otherwise happen between comparison and blocking.

The physical address of memory where the **obj* variable is located, is used as a key to index sleeping threads.

The read of the current value of the *obj variable is not guarded by barriers. In particular, it is the user's duty to ensure the lock acquire and release memory semantics, if the UMTX_OP_WAIT and UMTX_OP_WAKE requests are used as a substrate for implementing a simple lock.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and EINTR error.

Optionally, a timeout for the request may be specified.

UMTX_OP_WAKE
Wake the threads possibly sleeping due to UMTX_OP_WAIT. The arguments for the request are:

obj Pointer to a variable, used as a key to find sleeping threads.

val Up to *val* threads are woken up by this request. Specify INT_MAX to wake up all waiters.

UMTX_OP_MUTEX_TRYLOCK
Try to lock umutex. The arguments to the request are:

obj Pointer to the umutex.

Operates same as the UMTX_OP_MUTEX_LOCK request, but returns EBUSY instead of sleeping if the lock cannot be obtained immediately.

UMTX_OP_MUTEX_LOCK
Lock umutex. The arguments to the request are:

obj Pointer to the umutex.

Locking is performed by writing the current thread id into the m_owner word of the *struct umutex*. The write is atomic, preserves the UMUTEX_CONTESTED contention indicator, and

provides the acquire barrier for lock entrance semantic.

If the lock cannot be obtained immediately because another thread owns the lock, the current thread is put to sleep, with UMUTEX_CONTESTED bit set before. Upon wake up, the lock conditions are re-tested.

The request adheres to the priority protection or inheritance protocol of the mutex, specified by the UMUTEX_PRIO_PROTECT or UMUTEX_PRIO_INHERIT flag, respectively.

Optionally, a timeout for the request may be specified.

A request with a timeout specified is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and EINTR error. A request without timeout specified is always restarted after return from a signal handler.

UMTX_OP_MUTEX_UNLOCK

Unlock umutex. The arguments to the request are:

obj Pointer to the umutex.

Unlocks the mutex, by writing UMUTEX_UNOWNED (zero) value into m_owner word of the *struct umutex*. The write is done with a release barrier, to provide lock leave semantic.

If there are threads sleeping in the sleep queue associated with the umutex, one thread is woken up. If more than one thread sleeps in the sleep queue, the UMUTEX_CONTESTED bit is set together with the write of the UMUTEX_UNOWNED value into m_owner.

The request adheres to the priority protection or inheritance protocol of the mutex, specified by the UMUTEX_PRIO_PROTECT or UMUTEX_PRIO_INHERIT flag, respectively. See description of the m_ceilings member of the *struct umutex* structure for additional details of the request operation on the priority protected protocol mutex.

UMTX_OP_SET_CEILING

Set ceiling for the priority protected umutex. The arguments to the request are:

obj Pointer to the umutex.

val New ceiling value.

uaddr Address of a variable of type *uint32_t*. If not NULL and the update was successful, the

previous ceiling value is written to the location pointed to by *uaddr*.

The request locks the umutex pointed to by the *obj* parameter, waiting for the lock if not immediately available. After the lock is obtained, the new ceiling value *val* is written to the m_ceilings[0] member of the *struct umutex,* after which the umutex is unlocked.

The locking does not adhere to the priority protect protocol, to conform to the POSIX requirements for the pthread_mutex_setprioceiling(3) interface.

UMTX_OP_CV_WAIT

Wait for a condition. The arguments to the request are:

obj Pointer to the *struct ucond*.

val Request flags, see below.

uaddr Pointer to the umutex.

uaddr2 Optional pointer to a *struct timespec* for timeout specification.

The request must be issued by the thread owning the mutex pointed to by the *uaddr* argument. The c_hash_waiters member of the *struct ucond*, pointed to by the *obj* argument, is set to an arbitrary non-zero value, after which the *uaddr* mutex is unlocked (following the appropriate protocol), and the current thread is put to sleep on the sleep queue keyed by the *obj* argument. The operations are performed atomically. It is guaranteed to not miss a wakeup from UMTX_OP_CV_SIGNAL or UMTX_OP_CV_BROADCAST sent between mutex unlock and putting the current thread on the sleep queue.

Upon wakeup, if the timeout expired and no other threads are sleeping in the same sleep queue, the c_hash_waiters member is cleared. After wakeup, the *uaddr* umutex is not relocked.

The following flags are defined:

CVWAIT_ABSTIME Timeout is absolute.

CVWAIT_CLOCKID Clockid is provided.

Optionally, a timeout for the request may be specified. Unlike other requests, the timeout value is specified directly by a *struct timespec*, pointed to by the *uaddr2* argument. If the CVWAIT_CLOCKID flag is provided, the timeout uses the clock from the c_clockid member of

the *struct ucond*, pointed to by *obj* argument. Otherwise, CLOCK_REALTIME is used, regardless of the clock identifier possibly specified in the *struct _umtx_time*. If the CVWAIT_ABSTIME flag is supplied, the timeout specifies absolute time value, otherwise it denotes a relative time interval.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and EINTR error.

UMTX_OP_CV_SIGNAL
Wake up one condition waiter. The arguments to the request are:

obj Pointer to *struct ucond*.

The request wakes up at most one thread sleeping on the sleep queue keyed by the *obj* argument. If the woken up thread was the last on the sleep queue, the c_has_waiters member of the *struct ucond* is cleared.

UMTX_OP_CV_BROADCAST
Wake up all condition waiters. The arguments to the request are:

obj Pointer to *struct ucond*.

The request wakes up all threads sleeping on the sleep queue keyed by the *obj* argument. The c_has_waiters member of the *struct ucond* is cleared.

UMTX_OP_WAIT_UINT
Same as UMTX_OP_WAIT, but the type of the variable pointed to by *obj* is *u_int* (a 32-bit integer).

UMTX_OP_RW_RDLOCK
Read-lock a *struct rwlock* lock. The arguments to the request are:

obj Pointer to the lock (of type *struct rwlock*) to be read-locked.

val Additional flags to augment locking behaviour. The valid flags in the *val* argument are:

URWLOCK_PREFER_READER

The request obtains the read lock on the specified *struct rwlock* by incrementing the count of readers in the rw_state word of the structure. If the URWLOCK_WRITE_OWNER bit is set in

the word rw_state, the lock was granted to a writer which has not yet relinquished its ownership. In this case the current thread is put to sleep until it makes sense to retry.

If the URWLOCK_PREFER_READER flag is set either in the rw_flags word of the structure, or in the *val* argument of the request, the presence of the threads trying to obtain the write lock on the same structure does not prevent the current thread from trying to obtain the read lock. Otherwise, if the flag is not set, and the URWLOCK_WRITE_WAITERS flag is set in rw_state, the current thread does not attempt to obtain read-lock. Instead it sets the URWLOCK_READ_WAITERS in the rw_state word and puts itself to sleep on corresponding sleep queue. Upon wakeup, the locking conditions are re-evaluated.

Optionally, a timeout for the request may be specified.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and EINTR error.

UMTX_OP_RW_WRLOCK

Write-lock a *struct rwlock* lock. The arguments to the request are:

obj Pointer to the lock (of type *struct rwlock*) to be write-locked.

The request obtains a write lock on the specified *struct rwlock*, by setting the URWLOCK_WRITE_OWNER bit in the rw_state word of the structure. If there is already a write lock owner, as indicated by the URWLOCK_WRITE_OWNER bit being set, or there are read lock owners, as indicated by the read-lock counter, the current thread does not attempt to obtain the write-lock. Instead it sets the URWLOCK_WRITE_WAITERS in the rw_state word and puts itself to sleep on corresponding sleep queue. Upon wakeup, the locking conditions are re-evaluated.

Optionally, a timeout for the request may be specified.

The request is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and EINTR error.

UMTX_OP_RW_UNLOCK

Unlock rwlock. The arguments to the request are:

obj Pointer to the lock (of type *struct rwlock*) to be unlocked.

The unlock type (read or write) is determined by the current lock state. Note that the *struct*

rwlock does not save information about the identity of the thread which acquired the lock.

If there are pending writers after the unlock, and the URWLOCK_PREFER_READER flag is not set in the rw_flags member of the **obj* structure, one writer is woken up, selected as described in the *SLEEP QUEUES* subsection. If the URWLOCK_PREFER_READER flag is set, a pending writer is woken up only if there is no pending readers.

If there are no pending writers, or, in the case that the URWLOCK_PREFER_READER flag is set, then all pending readers are woken up by unlock.

UMTX_OP_WAIT_UINT_PRIVATE
Same as UMTX_OP_WAIT_UINT, but unconditionally select the process-private sleep queue.

UMTX_OP_WAKE_PRIVATE
Same as UMTX_OP_WAKE, but unconditionally select the process-private sleep queue.

UMTX_OP_MUTEX_WAIT
Wait for mutex availability. The arguments to the request are:

obj Address of the mutex.

Similarly to the UMTX_OP_MUTEX_LOCK, put the requesting thread to sleep if the mutex lock cannot be obtained immediately. The UMUTEX_CONTESTED bit is set in the m_owner word of the mutex to indicate that there is a waiter, before the thread is added to the sleep queue. Unlike the UMTX_OP_MUTEX_LOCK request, the lock is not obtained.

The operation is not implemented for priority protected and priority inherited protocol mutexes.

Optionally, a timeout for the request may be specified.

A request with a timeout specified is not restartable. An unblocked signal delivered during the wait always results in sleep interruption and EINTR error. A request without a timeout automatically restarts if the signal disposition requested restart via the SA_RESTART flag in *struct sigaction* member sa_flags.

UMTX_OP_NWAKE_PRIVATE
Wake up a batch of sleeping threads. The arguments to the request are:

obj Pointer to the array of pointers.

val Number of elements in the array pointed to by *obj*.

For each element in the array pointed to by *obj*, wakes up all threads waiting on the *private* sleep queue with the key being the byte addressed by the array element.

UMTX_OP_MUTEX_WAKE

Check if a normal umutex is unlocked and wake up a waiter. The arguments for the request are:

obj Pointer to the umutex.

If the m_owner word of the mutex pointed to by the *obj* argument indicates unowned mutex, which has its contention indicator bit UMUTEX_CONTESTED set, clear the bit and wake up one waiter in the sleep queue associated with the byte addressed by the *obj*, if any. Only normal mutexes are supported by the request. The sleep queue is always one for a normal mutex type.

This request is deprecated in favor of UMTX_OP_MUTEX_WAKE2 since mutexes using it cannot synchronize their own destruction. That is, the m_owner word has already been set to UMUTEX_UNOWNED when this request is made, so that another thread can lock, unlock and destroy the mutex (if no other thread uses the mutex afterwards). Clearing the UMUTEX_CONTESTED bit may then modify freed memory.

UMTX_OP_MUTEX_WAKE2

Check if a umutex is unlocked and wake up a waiter. The arguments for the request are:

obj Pointer to the umutex.

val The umutex flags.

The request does not read the m_flags member of the *struct umutex*; instead, the *val* argument supplies flag information, in particular, to determine the sleep queue where the waiters are found for wake up.

If the mutex is unowned, one waiter is woken up.

If the mutex memory cannot be accessed, all waiters are woken up.

If there is more than one waiter on the sleep queue, or there is only one waiter but the mutex is owned by a thread, the UMUTEX_CONTESTED bit is set in the m_owner word of the *struct umutex*.

UMTX_OP_SEM2_WAIT

Wait until semaphore is available. The arguments to the request are:

obj Pointer to the semaphore (of type *struct _usem2*).

Put the requesting thread onto a sleep queue if the semaphore counter is zero. If the thread is put to sleep, the USEM_HAS_WAITERS bit is set in the _count word to indicate waiters. The function returns either due to _count indicating the semaphore is available (non-zero count due to post), or due to a wakeup. The return does not guarantee that the semaphore is available, nor does it consume the semaphore lock on successful return.

Optionally, a timeout for the request may be specified.

A request with non-absolute timeout value is not restartable. An unblocked signal delivered during such wait results in sleep interruption and EINTR error.

UMTX_OP_SEM2_WAKE

Wake up waiters on semaphore lock. The arguments to the request are:

obj Pointer to the semaphore (of type *struct _usem2*).

The request wakes up one waiter for the semaphore lock. The function does not increment the semaphore lock count. If the USEM_HAS_WAITERS bit was set in the _count word, and the last sleeping thread was woken up, the bit is cleared.

UMTX_OP_SHM

Manage anonymous POSIX shared memory objects (see shm_open(2)), which can be attached to a byte of physical memory, mapped into the process address space. The objects are used to implement process-shared locks in libthr.

The *val* argument specifies the sub-request of the UMTX_OP_SHM request:

UMTX_SHM_CREAT

Creates the anonymous shared memory object, which can be looked up with the specified key *uaddr*. If the object associated with the *uaddr* key already exists, it is returned instead of creating a new object. The object's size is one page. On success, the file descriptor referencing the object is returned. The descriptor can be used for mapping the object using mmap(2), or for other shared memory operations.

UMTX_SHM_LOOKUP

Same as UMTX_SHM_CREATE request, but if there is no shared memory object associated with the specified key *uaddr*, an error is returned, and no new object is created.

UMTX_SHM_DESTROY
De-associate the shared object with the specified key *uaddr*. The object is destroyed after the last open file descriptor is closed and the last mapping for it is destroyed.

UMTX_SHM_ALIVE
Checks whether there is a live shared object associated with the supplied key *uaddr*. Returns zero if there is, and an error otherwise. This request is an optimization of the UMTX_SHM_LOOKUP request. It is cheaper when only the liveness of the associated object is asked for, since no file descriptor is installed in the process fd table on success.

The *uaddr* argument specifies the virtual address, which backing physical memory byte identity is used as a key for the anonymous shared object creation or lookup.

UMTX_OP_ROBUST_LISTS
Register the list heads for the current thread's robust mutex lists. The arguments to the request are:

val Size of the structure passed in the *uaddr* argument.

uaddr Pointer to the structure of type *struct umtx_robust_lists_params*.

The structure is defined as

```
struct umtx_robust_lists_params {
        uintptr_t  robust_list_offset;
        uintptr_t  robust_priv_list_offset;
        uintptr_t  robust_inact_offset;
};
```

The robust_list_offset member contains address of the first element in the list of locked robust shared mutexes. The robust_priv_list_offset member contains address of the first element in the list of locked robust private mutexes. The private and shared robust locked lists are split to allow fast termination of the shared list on fork, in the child.

The robust_inact_offset contains a pointer to the mutex which might be locked in nearby future, or might have been just unlocked. It is typically set by the lock or unlock mutex implementation code around the whole operation, since lists can be only changed race-free when the thread owns

the mutex. The kernel inspects the robust_inact_offset in addition to walking the shared and private lists. Also, the mutex pointed to by robust_inact_offset is handled more loosely at the thread termination time, than other mutexes on the list. That mutex is allowed to be not owned by the current thread, in which case list processing is continued. See *ROBUST UMUTEXES* subsection for details.

RETURN VALUES

If successful, all requests, except UMTX_SHM_CREAT and UMTX_SHM_LOOKUP sub-requests of the UMTX_OP_SHM request, will return zero. The UMTX_SHM_CREAT and UMTX_SHM_LOOKUP return a shared memory file descriptor on success. On error -1 is returned, and the *errno* variable is set to indicate the error.

ERRORS

The **_umtx_op**() operations can fail with the following errors:

[EFAULT] One of the arguments point to invalid memory.

[EINVAL] The clock identifier, specified for the *struct _umtx_time* timeout parameter, or in the c_clockid member of *struct ucond,* is invalid.

[EINVAL] The type of the mutex, encoded by the m_flags member of *struct umutex*, is invalid.

[EINVAL] The m_owner member of the *struct umutex* has changed the lock owner thread identifier during unlock.

[EINVAL] The timeout.tv_sec or timeout.tv_nsec member of *struct _umtx_time* is less than zero, or timeout.tv_nsec is greater than 1000000000.

[EINVAL] The *op* argument specifies invalid operation.

[EINVAL] The *uaddr* argument for the UMTX_OP_SHM request specifies invalid operation.

[EINVAL] The UMTX_OP_SET_CEILING request specifies non priority protected mutex.

[EINVAL] The new ceiling value for the UMTX_OP_SET_CEILING request, or one or more of the values read from the m_ceilings array during lock or unlock operations, is greater than RTP_PRIO_MAX.

[EPERM] Unlock attempted on an object not owned by the current thread.

[EOWNERDEAD]
The lock was requested on an umutex where the m_owner field was set to the UMUTEX_RB_OWNERDEAD value, indicating terminated robust mutex. The lock was granted to the caller, so this error in fact indicates success with additional conditions.

[ENOTRECOVERABLE]
The lock was requested on an umutex which m_owner field is equal to the UMUTEX_RB_NOTRECOV value, indicating abandoned robust mutex after termination. The lock was not granted to the caller.

[ENOTTY] The shared memory object, associated with the address passed to the UMTX_SHM_ALIVE sub-request of UMTX_OP_SHM request, was destroyed.

[ESRCH] For the UMTX_SHM_LOOKUP, UMTX_SHM_DESTROY, and UMTX_SHM_ALIVE sub-requests of the UMTX_OP_SHM request, there is no shared memory object associated with the provided key.

[ENOMEM] The UMTX_SHM_CREAT sub-request of the UMTX_OP_SHM request cannot be satisfied, because allocation of the shared memory object would exceed the RLIMIT_UMTXP resource limit, see setrlimit(2).

[EAGAIN] The maximum number of readers (URWLOCK_MAX_READERS) were already granted ownership of the given *struct rwlock* for read.

[EBUSY] A try mutex lock operation was not able to obtain the lock.

[ETIMEDOUT] The request specified a timeout in the *uaddr* and *uaddr2* arguments, and timed out before obtaining the lock or being woken up.

[EINTR] A signal was delivered during wait, for a non-restartable operation. Operations with timeouts are typically non-restartable, but timeouts specified in absolute time may be restartable.

[ERESTART] A signal was delivered during wait, for a restartable operation. Mutex lock requests without timeout specified are restartable. The error is not returned to userspace code since restart is handled by usual adjustment of the instruction counter.

SEE ALSO

clock_gettime(2), mmap(2), setrlimit(2), shm_open(2), sigaction(2), thr_exit(2), thr_kill(2), thr_kill2(2),

thr_new(2), thr_self(2), thr_set_name(2), signal(3)

STANDARDS

The **_umtx_op**() system call is non-standard and is used by the 1:1 Threading Library (libthr, -lthr) to implement IEEE Std 1003.1-2001 ("POSIX.1") pthread(3) functionality.

BUGS

A window between a unlocking robust mutex and resetting the pointer in the robust_inact_offset member of the registered *struct umtx_robust_lists_params* allows another thread to destroy the mutex, thus making the kernel inspect freed or reused memory. The libthr implementation is only vulnerable to this race when operating on a shared mutex. A possible fix for the current implementation is to strengthen the checks for shared mutexes before terminating them, in particular, verifying that the mutex memory is mapped from a shared memory object allocated by the UMTX_OP_SHM request. This is not done because it is believed that the race is adequately covered by other consistency checks, while adding the check would prevent alternative implementations of libpthread.

NAME

a64l, **l64a**, **l64a_r** - convert between a long integer and a base-64 ASCII string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

long
a64l(*const char *s*);

*char **
l64a(*long int l*);

int
l64a_r(*long int l*, *char *buffer*, *int buflen*);

DESCRIPTION

These functions are used to maintain numbers stored in radix-64 ASCII characters. This is a notation by which 32-bit integers can be represented by up to six characters; each character represents a digit in radix-64 notation. If the type long contains more than 32 bits, only the low-order 32 bits are used for these operations.

The characters used to represent "digits" are '.' for 0, '/' for 1, '0' - '9' for 2 - 11, 'A' - 'Z' for 12 - 37, and 'a' - 'z' for 38 - 63.

The **a64l**() function takes a pointer to a radix-64 representation, in which the first digit is the least significant, and returns a corresponding *long* value. If the string pointed to by *s* contains more than six characters, **a64l**() uses the first six. If the first six characters of the string contain a null terminator, **a64l**() uses only characters preceding the null terminator. The **a64l**() function scans the character string from left to right with the least significant digit on the left, decoding each character as a 6-bit radix-64 number. If the type long contains more than 32 bits, the resulting value is sign-extended. The behavior of **a64l**() is unspecified if *s* is a null pointer or the string pointed to by *s* was not generated by a previous call to **l64a**().

The **l64a**() function takes a *long* argument and returns a pointer to the corresponding radix-64 representation. The behavior of **l64a**() is unspecified if value is negative.

The value returned by **l64a**() is a pointer into a static buffer. Subsequent calls to **l64a**() may overwrite

the buffer.

The **l64a_r**() function performs a conversion identical to that of **l64a**() and stores the resulting representation in the memory area pointed to by *buffer*, consuming at most *buflen* characters including the terminating NUL character.

RETURN VALUES

On successful completion, **a64l**() returns the *long* value resulting from conversion of the input string. If a string pointed to by *s* is an empty string, **a64l**() returns 0.

The **l64a**() function returns a pointer to the radix-64 representation. If value is 0, **l64a**() returns a pointer to an empty string.

SEE ALSO

strtoul(3)

HISTORY

The **a64l**(), **l64a**(), and **l64a_r**() functions are derived from NetBSD with modifications. They appeared in FreeBSD 6.1.

AUTHORS

The **a64l**(), **l64a**(), and **l64a_r**() functions were added to FreeBSD by Tom Rhodes <*trhodes@FreeBSD.org*>. Almost all of this manual page came from the POSIX standard.

NAME

set_constraint_handler_s, **abort_handler_s**, **ignore_handler_s** - runtime-constraint violation handling

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#define __STDC_WANT_LIB_EXT1__ 1

#include <stdlib.h>

constraint_handler_t
set_constraint_handler_s(*constraint_handler_t handler*);

Handler Prototype

typedef void
(*constraint_handler_t)(*const char * restrict msg*, *void * restrict ptr*, *errno_t error*);

Predefined Handlers

void
abort_handler_s(*const char * restrict msg*, *void * restrict ptr*, *errno_t error*);

void
ignore_handler_s(*const char * restrict msg*, *void * restrict ptr*, *errno_t error*);

DESCRIPTION

The **set_constraint_handler_s**() function sets the runtime-constraint violation handler to be *handler*.

The runtime-constraint handler is the callback function invoked when a library function detects a runtime-constraint violation.

The arguments are as follows:

msg A pointer to a character string describing the runtime-constraint violation.

ptr A NULL pointer.

error If the function calling the handler has a return type declared as *errno_t*, the return value of the function is passed. Otherwise, a positive value of type *errno_t* is passed.

Only the most recent handler registered with **set_constraint_handler_s**() is called when a runtime-constraint violation occurs.

The implementation has a default constraint handler that is used if no calls to the **set_constraint_handler_s**() function have been made. If the *handler* argument to **set_constraint_handler_s**() is a NULL pointer, the default handler becomes the current constraint handler.

The **abort_handler_s**() and **ignore_handler_s**() are the standard-defined runtime-constraint handlers provided by the C library.

The **abort_handler_s**() function writes the error message including the *msg* to stderr and calls the abort(3) function. The **abort_handler_s**() is currently the default runtime-constraint handler.

The **ignore_handler_s**() simply returns to its caller.

RETURN VALUES

The **set_constraint_handler_s**() function returns a pointer to the previously registered handler, or NULL if none was previously registered.

The **abort_handler_s**() function does not return to its caller.

The **ignore_handler_s**() function returns no value.

STANDARDS

The **set_constraint_handler_s**() function conforms to ISO/IEC 9899:2011 ("ISO C11") K.3.6.1.1.

AUTHORS

This manual page was written by Yuri Pankov <*yuripv@yuripv.net*>.

NAME

abort - cause abnormal program termination

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

void
abort(*void*);

DESCRIPTION

The **abort**() function causes abnormal program termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return.

Any open streams are flushed and closed.

IMPLEMENTATION NOTES

The **abort**() function is thread-safe. It is unknown if it is async-cancel-safe.

RETURN VALUES

The **abort**() function never returns.

SEE ALSO

abort2(2), sigaction(2), exit(3)

STANDARDS

The **abort**() function conforms to IEEE Std 1003.1-1990 ("POSIX.1"). The **abort**() function also conforms to ISO/IEC 9899:1999 ("ISO C99") with the implementation specific details as noted above.

NAME

abort2 - abort process with diagnostics

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

void
abort2(*const char *why*, *int nargs*, *void **args*);

DESCRIPTION

The **abort2**() system call causes the process to be killed and the specified diagnostic message (with arguments) to be delivered by the kernel to the syslogd(8) daemon.

The *why* argument points to a NUL-terminated string specifying a reason of the program's termination (maximum 128 characters long). The *args* array contains pointers which will be logged numerically (with the kernel's '%p' printf(9) format). The *nargs* argument specifies the number of pointers in *args* (maximum 16).

The **abort2**() system call is intended for use in situations where continuation of a process is impossible or for other definitive reasons is unwanted, and normal diagnostic channels cannot be trusted to deliver the message.

RETURN VALUES

The **abort2**() function never returns.

The process is killed with SIGABRT unless the arguments to **abort2**() are invalid, in which case SIGKILL is used.

EXAMPLES

```
#include <stdlib.h>

if (weight_kg > max_load) {
        void *ptrs[3];

        ptrs[0] = (void *)(intptr_t)weight_kg;
        ptrs[1] = (void *)(intptr_t)max_load;
        ptrs[2] = haystack;
```

```
        abort2("Camel overloaded", 3, ptrs);
}
```

SEE ALSO

abort(3), exit(3)

HISTORY

The **abort2**() system call first appeared in FreeBSD 7.0.

AUTHORS

The **abort2**() system call was designed by Poul-Henning Kamp <*phk@FreeBSD.org*>. It was implemented by Wojciech A. Koszek <*dunstan@freebsd.czest.pl*>.

NAME

abs - integer absolute value function

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
abs(*int j*);

DESCRIPTION

The **abs**() function computes the absolute value of the integer *j*.

RETURN VALUES

The **abs**() function returns the absolute value.

SEE ALSO

cabs(3), fabs(3), floor(3), hypot(3), imaxabs(3), labs(3), llabs(3), math(3)

STANDARDS

The **abs**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

BUGS

The absolute value of the most negative integer remains negative.

NAME

accept, **accept4** - accept a connection on a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

int
accept(*int s*, *struct sockaddr * restrict addr*, *socklen_t * restrict addrlen*);

int
accept4(*int s*, *struct sockaddr * restrict addr*, *socklen_t * restrict addrlen*, *int flags*);

DESCRIPTION

The argument *s* is a socket that has been created with socket(2), bound to an address with bind(2), and is listening for connections after a listen(2). The **accept**() system call extracts the first connection request on the queue of pending connections, creates a new socket, and allocates a new file descriptor for the socket which inherits the state of the O_NONBLOCK and O_ASYNC properties and the destination of SIGIO and SIGURG signals from the original socket *s*.

The **accept4**() system call is similar, but the O_NONBLOCK property of the new socket is instead determined by the SOCK_NONBLOCK flag in the *flags* argument, the O_ASYNC property is cleared, the signal destination is cleared and the close-on-exec flag on the new file descriptor can be set via the SOCK_CLOEXEC flag in the *flags* argument.

If no pending connections are present on the queue, and the original socket is not marked as non-blocking, **accept**() blocks the caller until a connection is present. If the original socket is marked non-blocking and no pending connections are present on the queue, **accept**() returns an error as described below. The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result argument that is filled-in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* argument is determined by the domain in which the communication is occurring. A null pointer may be specified for *addr* if the address information is not desired; in this case, *addrlen* is not used and should also be null. Otherwise, the *addrlen* argument is a value-result argument; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used

with connection-based socket types, currently with SOCK_STREAM.

It is possible to select(2) a socket for the purposes of doing an **accept**() by selecting it for read.

For certain protocols which require an explicit confirmation, such as ISO or DATAKIT, **accept**() can be thought of as merely dequeueing the next connection request and not implying confirmation. Confirmation can be implied by a normal read or write on the new file descriptor, and rejection can be implied by closing the new socket.

For some applications, performance may be enhanced by using an accept_filter(9) to pre-process incoming connections.

When using **accept**(), portable programs should not rely on the O_NONBLOCK and O_ASYNC properties and the signal destination being inherited, but should set them explicitly using fcntl(2); **accept4**() sets these properties consistently, but may not be fully portable across UNIX platforms.

RETURN VALUES

These calls return -1 on error. If they succeed, they return a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The **accept**() and **accept4**() system calls will fail if:

[EBADF] The descriptor is invalid.

[EINTR] The **accept**() operation was interrupted.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[ENOTSOCK] The descriptor references a file, not a socket.

[EINVAL] listen(2) has not been called on the socket descriptor.

[EFAULT] The *addr* argument is not in a writable part of the user address space.

[EWOULDBLOCK] or [EAGAIN]
The socket is marked non-blocking and no connections are present to be accepted.

[ECONNABORTED]
A connection arrived, but it was closed while waiting on the listen queue.

The **accept4**() system call will also fail if:

[EINVAL] The *flags* argument is invalid.

SEE ALSO

bind(2), connect(2), getpeername(2), getsockname(2), listen(2), select(2), socket(2), accept_filter(9)

HISTORY

The **accept**() system call appeared in 4.2BSD.

The **accept4**() system call appeared in FreeBSD 10.0.

NAME

access, **eaccess**, **faccessat** - check accessibility of a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
access(*const char *path*, *int mode*);

int
eaccess(*const char *path*, *int mode*);

int
faccessat(*int fd*, *const char *path*, *int mode*, *int flag*);

DESCRIPTION

The **access**() and **eaccess**() system calls check the accessibility of the file named by the *path* argument for the access permissions indicated by the *mode* argument. The value of *mode* is either the bitwise-inclusive OR of the access permissions to be checked (R_OK for read permission, W_OK for write permission, and X_OK for execute/search permission), or the existence test (F_OK).

For additional information, see the *File Access Permission* section of intro(2).

The **eaccess**() system call uses the effective user ID and the group access list to authorize the request; the **access**() system call uses the real user ID in place of the effective user ID, the real group ID in place of the effective group ID, and the rest of the group access list.

The **faccessat**() system call is equivalent to **access**() except in the case where *path* specifies a relative path. In this case the file whose accessibility is to be determined is located relative to the directory associated with the file descriptor *fd* instead of the current working directory. If **faccessat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **access**(). Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

AT_EACCESS
: The checks for accessibility are performed using the effective user and group IDs instead of the real user and group ID as required in a call to **access**().

Even if a process's real or effective user has appropriate privileges and indicates success for X_OK, the file may not actually have execute permission bits set. Likewise for R_OK and W_OK.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

access(), **eaccess**(), or **faccessat**() will fail if:

[EINVAL] The value of the *mode* argument is invalid.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EROFS] Write access is requested for a file on a read-only file system.

[ETXTBSY] Write access is requested for a pure procedure (shared text) file presently being executed.

[EACCES] Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

Also, the **faccessat**() system call may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor.

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chmod(2), intro(2), stat(2)

STANDARDS

The **access**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **faccessat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **access**() function appeared in Version 7 AT&T UNIX. The **faccessat**() system call appeared in FreeBSD 8.0.

SECURITY CONSIDERATIONS

The **access**() system call is a potential security hole due to race conditions and should never be used. Set-user-ID and set-group-ID applications should restore the effective user or group ID, and perform actions directly rather than use **access**() to simulate access checks for the real user or group ID. The **eaccess**() system call likewise may be subject to races if used inappropriately.

access() remains useful for providing clues to users as to whether operations make sense for particular filesystem objects (e.g. 'delete' menu item only highlighted in a writable folder ... avoiding interpretation of the st_mode bits that the application might not understand -- e.g. in the case of AFS). It also allows a cheaper file existence test than stat(2).

NAME

acct - enable or disable process accounting

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
acct(*const char *file*);

DESCRIPTION

The **acct**() system call enables or disables the collection of system accounting records. If the argument *file* is a null pointer, accounting is disabled. If *file* is an *existing* pathname (null-terminated), record collection is enabled and for every process initiated which terminates under normal conditions an accounting record is appended to *file*. Abnormal conditions of termination are reboots or other fatal system problems. Records for processes which never terminate cannot be produced by **acct**().

For more information on the record structure used by **acct**(), see <*sys/acct.h*> and acct(5).

This call is permitted only to the super-user.

NOTES

Accounting is automatically disabled when the file system the accounting file resides on runs out of space; it is enabled when space once again becomes available. The values controlling this behaviour can be modified using the following sysctl(8) variables:

kern.acct_chkfreq	Specifies the frequency (in seconds) with which free disk space should be checked.
kern.acct_resume	The percentage of free disk space above which process accounting will resume.
kern.acct_suspend	The percentage of free disk space below which process accounting will suspend.

RETURN VALUES

On error -1 is returned. The file must exist and the call may be exercised only by the super-user.

ERRORS

The **acct**() system call will fail if one of the following is true:

[EPERM] The caller is not the super-user.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix, or the path name is not a regular file.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EROFS] The named file resides on a read-only file system.

[EFAULT] The *file* argument points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

acct(5), accton(8), sa(8)

HISTORY

The **acct**() function appeared in Version 7 AT&T UNIX.

NAME

acl_add_flag_np - add flags to a flagset

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_add_flag_np(*acl_flagset_t flagset_d*, *acl_flag_t flag*);

DESCRIPTION

The **acl_add_flag_np**() function is a non-portable call that adds the NFSv4 ACL flags contained in *flags* to the flagset *flagset_d*.

Note: it is not considered an error to attempt to add flags that already exist in the flagset.

Valid values are:

ACL_ENTRY_FILE_INHERIT	Will be inherited by files.
ACL_ENTRY_DIRECTORY_INHERIT	Will be inherited by directories.
ACL_ENTRY_NO_PROPAGATE_INHERIT	Will not propagate.
ACL_ENTRY_INHERIT_ONLY	Inherit-only.
ACL_ENTRY_INHERITED	Inherited from parent

RETURN VALUES

The **acl_add_flag_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_add_flag_np**() function fails if:

[EINVAL] Argument *flagset_d* is not a valid descriptor for a flagset within an ACL entry. Argument *flag* does not contain a valid *acl_flag_t* value.

SEE ALSO

acl(3), acl_clear_flags_np(3), acl_delete_flag_np(3), acl_get_flagset_np(3), acl_set_flagset_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_add_flag_np**() function was added in FreeBSD 8.0.

AUTHORS

The **acl_add_flag_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_add_perm - add permissions to a permission set

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_add_perm(*acl_permset_t permset_d*, *acl_perm_t perm*);

DESCRIPTION

The **acl_add_perm**() function is a POSIX.1e call that adds the permission contained in *perm* to the permission set *permset_d*.

Note: it is not considered an error to attempt to add permissions that already exist in the permission set.

For POSIX.1e ACLs, valid values are:

ACL_EXECUTE	Execute permission
ACL_WRITE	Write permission
ACL_READ	Read permission

For NFSv4 ACLs, valid values are:

ACL_READ_DATA	Read permission
ACL_LIST_DIRECTORY	Same as ACL_READ_DATA
ACL_WRITE_DATA	Write permission, or permission to create files
ACL_ADD_FILE	Same as ACL_READ_DATA
ACL_APPEND_DATA	Permission to create directories. Ignored for files
ACL_ADD_SUBDIRECTORY	Same as ACL_APPEND_DATA
ACL_READ_NAMED_ATTRS	Ignored
ACL_WRITE_NAMED_ATTRS	Ignored
ACL_EXECUTE	Execute permission
ACL_DELETE_CHILD	Permission to delete files and subdirectories
ACL_READ_ATTRIBUTES	Permission to read basic attributes
ACL_WRITE_ATTRIBUTES	Permission to change basic attributes
ACL_DELETE	Permission to delete the object this ACL is placed on

ACL_READ_ACL	Permission to read ACL
ACL_WRITE_ACL	Permission to change the ACL and file mode
ACL_SYNCHRONIZE	Ignored

Calling **acl_add_perm**() with *perm* equal to ACL_WRITE or ACL_READ brands the ACL as POSIX. Calling it with ACL_READ_DATA, ACL_LIST_DIRECTORY, ACL_WRITE_DATA, ACL_ADD_FILE, ACL_APPEND_DATA, ACL_ADD_SUBDIRECTORY, ACL_READ_NAMED_ATTRS, ACL_WRITE_NAMED_ATTRS, ACL_DELETE_CHILD, ACL_READ_ATTRIBUTES, ACL_WRITE_ATTRIBUTES, ACL_DELETE, ACL_READ_ACL, ACL_WRITE_ACL or ACL_SYNCHRONIZE brands the ACL as NFSv4.

RETURN VALUES

The **acl_add_perm**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_add_perm**() function fails if:

[EINVAL]	Argument *permset_d* is not a valid descriptor for a permission set within an ACL entry. Argument *perm* does not contain a valid *acl_perm_t* value. ACL is already branded differently.

SEE ALSO

acl(3), acl_clear_perms(3), acl_delete_perm(3), acl_get_brand_np(3), acl_get_permset(3), acl_set_permset(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_add_perm**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_add_perm**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_calc_mask - calculate and set ACL mask permissions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_calc_mask(*acl_t *acl_p*);

DESCRIPTION

The **acl_calc_mask**() function is a POSIX.1e call that calculates and set the permissions associated with the ACL_MASK ACL entry of the ACL referred to by *acl_p*.

The value of new permissions are the union of the permissions granted by the ACL_GROUP, ACL_GROUP_OBJ, ACL_USER tag types which match processes in the file group class contained in the ACL referred to by *acl_p*.

If the ACL referred to by *acl_p* already contains an ACL_MASK entry, its permissions shall be overwritten; if it does not contain an ACL_MASK entry, one shall be added.

RETURN VALUES

The **acl_calc_mask**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_calc_mask**() function fails if:

[EINVAL] Argument *acl_p* does not point to a pointer to a valid ACL.

SEE ALSO

acl(3), acl_get_entry(3), acl_valid(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_calc_mask**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_calc_mask**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_clear_flags_np - clear flags from a flagset

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_clear_flags_np(*acl_flagset_t flagset_d*);

DESCRIPTION

The **acl_clear_flags_np**() function is a non-portable call that clears all NFSv4 ACL flags from flagset *flagset_d*.

RETURN VALUES

The **acl_clear_flags_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_clear_flags_np**() function fails if:

[EINVAL] Argument *flagset_d* is not a valid descriptor for a flagset.

SEE ALSO

acl(3), acl_add_flag_np(3), acl_delete_flag_np(3), acl_get_flagset_np(3), acl_set_flagset_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_clear_flags_np**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_clear_flags_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_clear_perms - clear permissions from a permission set

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_clear_perms(*acl_permset_t permset_d*);

DESCRIPTION

The **acl_clear_perms**() function is a POSIX.1e call that clears all permissions from permissions set *permset_d*.

RETURN VALUES

The **acl_clear_perms**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_clear_perms**() function fails if:

[EINVAL] Argument *permset_d* is not a valid descriptor for a permission set.

SEE ALSO

acl(3), acl_add_perm(3), acl_delete_perm(3), acl_get_permset(3), acl_set_permset(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_clear_perms**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_clear_perms**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_copy_entry - copy an ACL entry to another ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_copy_entry(*acl_entry_t dest_d*, *acl_entry_t src_d*);

DESCRIPTION

The **acl_copy_entry**() function is a POSIX.1e call that copies the contents of ACL entry *src_d* to ACL entry *dest_d*.

RETURN VALUES

The **acl_copy_entry**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_copy_entry**() function fails if:

[EINVAL] Argument *src_d* or *dest_d* is not a valid descriptor for an ACL entry, or arguments *src_d* and *dest_d* reference the same ACL entry.

SEE ALSO

acl(3), acl_get_entry(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_copy_entry**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_copy_entry**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_create_entry, **acl_create_entry_np** - create a new ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_create_entry(*acl_t *acl_p*, *acl_entry_t *entry_p*);

int
acl_create_entry_np(*acl_t *acl_p*, *acl_entry_t *entry_p*, *int index*);

DESCRIPTION

The **acl_create_entry**() function is a POSIX.1e call that creates a new ACL entry in the ACL pointed to by *acl_p*. The **acl_create_entry_np**() function is a non-portable version that creates the ACL entry at position *index*. Positions are numbered starting from zero, i.e. calling **acl_create_entry_np**() with *index* argument equal to zero will prepend the entry to the ACL.

RETURN VALUES

The **acl_create_entry**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_create_entry**() function fails if:

[EINVAL] Argument *acl_p* does not point to a pointer to a valid ACL. Argument *index* is out of bounds.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_delete_entry(3), acl_get_entry(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_create_entry**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_create_entry**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_delete_def_file, **acl_delete_def_link_np**, **acl_delete_fd_np**, **acl_delete_file_np**, **acl_delete_link_np** - delete an ACL from a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_delete_def_file(*const char *path_p*);

int
acl_delete_def_link_np(*const char *path_p*);

int
acl_delete_fd_np(*int filedes*, *acl_type_t type*);

int
acl_delete_file_np(*const char *path_p*, *acl_type_t type*);

int
acl_delete_link_np(*const char *path_p*, *acl_type_t type*);

DESCRIPTION

The **acl_delete_def_file**(), **acl_delete_def_link_np**(), **acl_delete_fd_np**(), **acl_delete_file_np**(), and **acl_delete_link_np**() each allow the deletion of an ACL from a file. The **acl_delete_def_file**() function is a POSIX.1e call that deletes the default ACL from a file (normally a directory) by name; the remainder of the calls are non-portable extensions that permit the deletion of arbitrary ACL types from a file/directory either by path name or file descriptor. The **_file**() variations follow a symlink if it occurs in the last segment of the path name; the **_link**() variations operate on the symlink itself.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, these functions shall return -1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix, or the object exists and the process does not have appropriate access rights.

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

[EINVAL] The ACL type passed is invalid for this file object.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named object does not exist, or the *path_p* argument points to an empty string.

[ENOMEM] Insufficient memory available to fulfill request.

[ENOTDIR] A component of the path prefix is not a directory.

Argument *path_p* must be a directory, and is not.

[EOPNOTSUPP] The file system does not support ACL deletion.

[EPERM] The process does not have appropriate privilege to perform the operation to delete an ACL.

[EROFS] The file system is read-only.

SEE ALSO

acl(3), acl_get(3), acl_set(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

NAME

acl_delete_entry, **acl_delete_entry_np** - delete an ACL entry from an ACL

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_delete_entry(*acl_t acl*, *acl_entry_t entry_d*);

int
acl_delete_entry_np(*acl_t acl*, *int index*);

DESCRIPTION

The **acl_delete_entry**() function is a POSIX.1e call that removes the ACL entry *entry_d* from ACL *acl*. The **acl_delete_entry_np**() function is a non-portable version that removes the ACL entry at position *index* from ACL *acl*. Positions are numbered starting from zero, i.e. calling **acl_delete_entry_np**() with *index* argument equal to zero will remove the first ACL entry.

RETURN VALUES

The **acl_delete_entry**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_delete_entry**() function fails if:

[EINVAL] Argument *acl* does not point to a valid ACL. Argument *entry_d* is not a valid descriptor for an ACL entry in *acl*. Argument *index* is out of bounds.

SEE ALSO

acl(3), acl_copy_entry(3), acl_get_entry(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_delete_entry**() function was added in

FreeBSD 5.0.

AUTHORS

The **acl_delete_entry**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_delete_flag_np - delete flags from a flagset

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_delete_flag_np(*acl_flagset_t flagset_d*, *acl_flag_t flag*);

DESCRIPTION

The **acl_delete_flag_np**() function is a non-portable call that removes specific NFSv4 ACL flags from flagset *flags*.

RETURN VALUES

The **acl_delete_flag_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_delete_flag_np**() function fails if:

[EINVAL] Argument *flagset_d* is not a valid descriptor for a flagset. Argument *flag* does not contain a valid *acl_flag_t* value.

SEE ALSO

acl(3), acl_add_flag_np(3), acl_clear_flags_np(3), acl_get_flagset_np(3), acl_set_flagset_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_delete_flag_np**() function was added in FreeBSD 8.0.

AUTHORS

The **acl_delete_flag_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_delete_perm - delete permissions from a permission set

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_delete_perm(*acl_permset_t permset_d*, *acl_perm_t perm*);

DESCRIPTION

The **acl_delete_perm**() function is a POSIX.1e call that removes specific permissions from permissions set *perm*.

RETURN VALUES

The **acl_delete_perm**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_delete_perm**() function fails if:

[EINVAL] Argument *permset_d* is not a valid descriptor for a permission set. Argument *perm* does not contain a valid *acl_perm_t* value.

SEE ALSO

acl(3), acl_add_perm(3), acl_clear_perms(3), acl_get_permset(3), acl_set_permset(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_delete_perm**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_delete_perm**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_dup - duplicate an ACL

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

acl_t
acl_dup(*acl_t acl*);

DESCRIPTION

The **acl_dup**() function returns a pointer to a copy of the ACL pointed to by the argument *acl*.

This function may cause memory to be allocated. The caller should free any releasable memory, when the new ACL is no longer required, by calling acl_free(3) with the *(void*)acl_t* as an argument.

Any existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, this function shall return a pointer to the duplicate ACL. Otherwise, a value of *(acl_t)NULL* shall be returned, and *errno* shall be set to indicate the error.

ERRORS

If any of the following conditions occur, the **acl_init**() function shall return a value of *(acl_t)NULL* and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL.

[ENOMEM] The *acl_t* to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_free(3), acl_get(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

NAME

acl_free - free ACL working state

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_free(*void *obj_p*);

DESCRIPTION

The **acl_free**() call allows the freeing of ACL working space, such as is allocated by acl_dup(3), or acl_from_text(3).

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

The **acl_free**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, the **acl_free**() function shall return -1 and set *errno* to the corresponding value:

[EINVAL] The value of the *obj_p* argument is invalid.

SEE ALSO

acl(3), acl_dup(3), acl_from_text(3), acl_get(3), acl_init(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

NAME

acl_from_text - create an ACL from text

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

acl_t
acl_from_text(*const char *buf_p*);

DESCRIPTION

The **acl_from_text**() function converts the text form of an ACL referred to by *buf_p* into the internal working structure for ACLs, appropriate for applying to files or manipulating.

This function may cause memory to be allocated. The caller should free any releasable memory, when the new ACL is no longer required, by calling acl_free(3) with the *(void *)acl_t* as an argument.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, the function shall return a pointer to the internal representation of the ACL in working storage. Otherwise, a value of *(acl_t)NULL* shall be returned, and *errno* shall be set to indicate the error.

ERRORS

If any of the following conditions occur, the **acl_from_text**() function shall return a value of *(acl_t)NULL* and set *errno* to the corresponding value:

[EINVAL] Argument *buf_p* cannot be translated into an ACL.

[ENOMEM] The ACL working storage requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_free(3), acl_get(3), acl_to_text(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

BUGS

The **acl_from_text**() and **acl_to_text**() functions rely on the getpwent(3) library calls to manage username and uid mapping, as well as the getgrent(3) library calls to manage groupname and gid mapping. These calls are not thread safe, and so transitively, neither are **acl_from_text**() and **acl_to_text**(). These functions may also interfere with stateful calls associated with the **getpwent**() and **getgrent**() calls.

NAME

acl_get_brand_np - retrieve the ACL brand from an ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_brand_np(*acl_t acl*, *int *brand_p*);

DESCRIPTION

The **acl_get_brand_np**() function is a non-portable call that returns the ACL brand for the ACL *acl*. Upon successful completion, the location referred to by the argument *brand_p* will be set to the ACL brand of the ACL *acl*.

Branding is an internal mechanism intended to prevent mixing POSIX.1e and NFSv4 entries by mistake. It's also used by the libc to determine how to print out the ACL. The first call to function that is specific for one particular brand - POSIX.1e or NFSv4 - "brands" the ACL. After that, calling function specific to another brand will result in error.

RETURN VALUES

The **acl_get_brand_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_get_brand_np**() function fails if:

[EINVAL] Argument *acl* does not point to a valid ACL.

SEE ALSO

acl(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_brand_np**() function was added in

FreeBSD 8.0.

AUTHORS

The **acl_get_brand_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_get_entry_type_np - retrieve the ACL type from an NFSv4 ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_entry_type_np(*acl_entry_t entry_d*, *acl_entry_type_t *entry_type_p*);

DESCRIPTION

The **acl_get_entry_type_np**() function is a non-portable call that returns the ACL type for the NFSv4 ACL entry *entry_d*. Upon successful completion, the location referred to by the argument *entry_type_p* will be set to the ACL type of the ACL entry *entry_d*.

RETURN VALUES

The **acl_get_entry_type_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_get_entry_type_np**() function fails if:

[EINVAL] Argument *entry_d* is not a valid descriptor for an NFSv4 ACL entry;

SEE ALSO

acl(3), acl_set_entry_type_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_entry_type_np**() function was added in FreeBSD 8.0.

AUTHORS

The **acl_get_entry_type_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_get_entry - retrieve an ACL entry from an ACL

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_entry(*acl_t acl*, *int entry_id*, *acl_entry_t *entry_p*);

DESCRIPTION

The **acl_get_entry**() function is a POSIX.1e call that retrieves a descriptor for an ACL entry specified by the argument *entry_d* within the ACL indicated by the argument *acl*.

If the value of *entry_id* is ACL_FIRST_ENTRY, then the function will return in *entry_p* a descriptor for the first ACL entry within *acl*. If a call is made to **acl_get_entry**() with *entry_id* set to ACL_NEXT_ENTRY when there has not been either an initial successful call to **acl_get_entry**(), or a previous successful call to **acl_create_entry**(), **acl_delete_entry**(), **acl_dup**(), **acl_from_text**(), **acl_get_fd**(), **acl_get_file**(), **acl_set_fd**(), **acl_set_file**(), or **acl_valid**(), then the result is unspecified.

RETURN VALUES

If the **acl_get_entry**() function successfully obtains an ACL entry, a value of 1 is returned. If the ACL has no ACL entries, the **acl_get_entry**() returns a value of 0. If the value of *entry_id* is ACL_NEXT_ENTRY and the last ACL entry in the ACL has already been returned by a previous call to **acl_get_entry**(), a value of 0 will be returned until a successful call with *entry_id* of ACL_FIRST_ENTRY is made. Otherwise, a value of -1 will be returned and the global variable *errno* will be set to indicate the error.

ERRORS

The **acl_get_entry**() fails if:

[EINVAL] Argument *acl* does not point to a valid ACL. Argument *entry_id* is neither ACL_FIRST_ENTRY nor ACL_NEXT_ENTRY.

SEE ALSO

acl(3), acl_calc_mask(3), acl_create_entry(3), acl_delete_entry(3), acl_dup(3), acl_from_text(3), acl_get_fd(3), acl_get_file(3), acl_init(3), acl_set_fd(3), acl_set_file(3), acl_valid(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_entry**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_get_entry**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_get_fd, **acl_get_fd_np**, **acl_get_file**, **acl_get_link_np** - get an ACL for a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

acl_t
acl_get_fd(*int fd*);

acl_t
acl_get_fd_np(*int fd*, *acl_type_t type*);

acl_t
acl_get_file(*const char *path_p*, *acl_type_t type*);

acl_t
acl_get_link_np(*const char *path_p*, *acl_type_t type*);

DESCRIPTION

The **acl_get_fd**(), **acl_get_file**(), **acl_get_link_np**(), and **acl_get_fd_np**() each allow the retrieval of an ACL from a file. The **acl_get_fd**() is a POSIX.1e call that allows the retrieval of an ACL of type ACL_TYPE_ACCESS from a file descriptor. The **acl_get_fd_np**() function is a non-portable form of **acl_get_fd**() that allows the retrieval of any type of ACL from a file descriptor. The **acl_get_file**() function is a POSIX.1e call that allows the retrieval of a specified type of ACL from a file by name; **acl_get_link_np**() is a non-portable variation on **acl_get_file**() which does not follow a symlink if the target of the call is a symlink.

These functions may cause memory to be allocated. The caller should free any releasable memory, when the new ACL is no longer required, by calling acl_free(3) with the *(void *)acl_t* as an argument.

The ACL in the working storage is an independent copy of the ACL associated with the object referred to by *fd*. The ACL in the working storage shall not participate in any access control decisions.

Valid values for the *type* argument are:

ACL_TYPE_ACCESS POSIX.1e access ACL

ACL_TYPE_DEFAULT	POSIX.1e default ACL
ACL_TYPE_NFS4	NFSv4 ACL

The ACL returned will be branded accordingly.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, the function shall return a pointer to the ACL that was retrieved. Otherwise, a value of *(acl_t)NULL* shall be returned, and *errno* shall be set to indicate the error.

ERRORS

If any of the following conditions occur, the **acl_get_fd**() function shall return a value of *(acl_t)NULL* and set *errno* to the corresponding value:

[EACCES]	Search permission is denied for a component of the path prefix, or the object exists and the process does not have appropriate access rights.
[EBADF]	The *fd* argument is not a valid file descriptor.
[EINVAL]	The ACL type passed is invalid for this file object.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named object does not exist, or the *path_p* argument points to an empty string.
[ENOMEM]	Insufficient memory available to fulfill request.
[EOPNOTSUPP]	The file system does not support ACL retrieval.

SEE ALSO

acl(3), acl_free(3), acl_get(3), acl_get_brand_np(3), acl_set(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e

implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

NAME

acl_get_flag_np - check if a flag is set in a flagset

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_flag_np(*acl_flagset_t flagset_d*, *acl_flag_t flag*);

DESCRIPTION

The **acl_get_flag_np**() function is a non-portable function that checks if a NFSv4 ACL flag is set in a flagset.

RETURN VALUES

If the flag in *flag* is set in the flagset *flagset_d*, a value of 1 is returned, otherwise a value of 0 is returned.

ERRORS

If any of the following conditions occur, the **acl_get_flag_np**() function will return a value of -1 and set global variable *errno* to the corresponding value:

[EINVAL] Argument *flag* does not contain a valid ACL flag or argument *flagset_d* is not a valid ACL flagset.

SEE ALSO

acl(3), acl_add_flag_np(3), acl_clear_flags_np(3), acl_delete_flag_np(3), acl_get_flagset_np(3), acl_set_flagset_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_flag_np**() function was added in FreeBSD 8.0.

AUTHORS

The **acl_get_flag_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_get_flagset_np - retrieve flagset from an NFSv4 ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_flagset_np(*acl_entry_t entry_d*, *acl_flagset_t *flagset_p*);

DESCRIPTION

The **acl_get_flagset_np**() function is a non-portable call that returns via *flagset_np_p* a descriptor to the flagset in the NFSv4 ACL entry *entry_d*. Subsequent operations using the returned flagset operate on the flagset within the ACL entry.

RETURN VALUES

The **acl_get_flagset_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_get_flagset_np**() function fails if:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

SEE ALSO

acl(3), acl_add_flag_np(3), acl_clear_flags_np(3), acl_delete_flag_np(3), acl_set_flagset_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_flagset_np**() function was added in FreeBSD 8.0.

AUTHORS

The **acl_get_flagset_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_get_perm_np - check if a permission is set in a permission set

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_perm_np(*acl_permset_t permset_d*, *acl_perm_t perm*);

DESCRIPTION

The **acl_get_perm_np**() function is a non-portable function that checks if a permission is set in a permission set.

RETURN VALUES

If the permission in *perm* is set in the permission set *permset_d*, a value of 1 is returned, otherwise a value of 0 is returned.

ERRORS

If any of the following conditions occur, the **acl_get_perm_np**() function will return a value of -1 and set global variable *errno* to the corresponding value:

[EINVAL] Argument *perm* does not contain a valid ACL permission or argument *permset_d* is not a valid ACL permset.

SEE ALSO

acl(3), acl_add_perm(3), acl_clear_perms(3), acl_delete_perm(3), acl_get_permset(3), acl_set_permset(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_perm_np**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_get_perm_np**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_get_permset - retrieve permission set from an ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_permset(*acl_entry_t entry_d*, *acl_permset_t *permset_p*);

DESCRIPTION

The **acl_get_permset**() function is a POSIX.1e call that returns via *permset_p* a descriptor to the permission set in the ACL entry *entry_d*. Subsequent operations using the returned permission set operate on the permission set within the ACL entry.

RETURN VALUES

The **acl_get_permset**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_get_permset**() function fails if:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

SEE ALSO

acl(3), acl_add_perm(3), acl_clear_perms(3), acl_delete_perm(3), acl_set_permset(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_permset**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_get_permset**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_get_qualifier - retrieve the qualifier from an ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

*void **
acl_get_qualifier(*acl_entry_t entry_d*);

DESCRIPTION

The **acl_get_qualifier**() function is a POSIX.1e call that retrieves the qualifier of the tag for the ACL entry indicated by the argument *entry_d* into working storage and returns a pointer to that storage.

If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_USER, then the value returned by **acl_get_qualifier**() will be a pointer to type *uid_t*.

If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_GROUP, then the value returned by **acl_get_qualifier**() will be a pointer to type *gid_t*.

If the value of the tag type in the ACL entry referred to by *entry_d* is ACL_UNDEFINED_TAG, ACL_USER_OBJ, ACL_GROUP_OBJ, ACL_OTHER, ACL_MASK, or an implementation-defined value for which a qualifier is not supported, then **acl_get_qualifier**() will return a value of (*void* *)NULL and the function will fail.

This function may cause memory to be allocated. The caller should free any releasable memory, when the new qualifier is no longer required, by calling **acl_free**() with *void* * as the argument.

RETURN VALUES

The **acl_get_qualifier**() function returns a pointer to the allocated storage if successful; otherwise a NULL pointer is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_get_qualifier**() fails if:

[EINVAL] Argument *entry_d* does not point to a valid descriptor for an ACL entry. The value of the tag type in the ACL entry referenced by argument *entry_d* is not

ACL_USER or ACL_GROUP.

[ENOMEM] The value to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_create_entry(3), acl_free(3), acl_get_entry(3), acl_get_tag_type(3), acl_set_qualifier(3), acl_set_tag_type(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_qualifier**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_get_qualifier**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_get_tag_type - retrieve the tag type from an ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_get_tag_type(*acl_entry_t entry_d*, *acl_tag_t *tag_type_p*);

DESCRIPTION

The **acl_get_tag_type**() function is a POSIX.1e call that returns the tag type for the ACL entry *entry_d*. Upon successful completion, the location referred to by the argument *tag_type_p* will be set to the tag type of the ACL entry *entry_d*.

RETURN VALUES

The **acl_get_tag_type**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_get_tag_type**() function fails if:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry;

SEE ALSO

acl(3), acl_create_entry(3), acl_get_entry(3), acl_get_qualifier(3), acl_init(3), acl_set_qualifier(3), acl_set_tag_type(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_tag_type**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_get_tag_type**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_init - initialize ACL working storage

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

acl_t
acl_init(*int count*);

DESCRIPTION

The **acl_init**() function allocates and initializes the working storage for an ACL of at least *count* ACL entries. A pointer to the working storage is returned. The working storage allocated to contain the ACL is freed by a call to acl_free(3). When the area is first allocated, it shall contain an ACL that contains no ACL entries.

This function may cause memory to be allocated. The caller should free any releasable memory, when the new ACL is no longer required, by calling acl_free(3) with the *(void*)acl_t* as an argument.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, this function shall return a pointer to the working storage. Otherwise, a value of *(acl_t)NULL* shall be returned, and *errno* shall be set to indicate the error.

ERRORS

If any of the following conditions occur, the **acl_init**() function shall return a value of *(acl_t)NULL* and set *errno* to the corresponding value:

[EINVAL] The value of count is less than zero.

[ENOMEM] The *acl_t* to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_free(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

NAME

acl_is_trivial_np - determine whether ACL is trivial

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_is_trivial_np(*const acl_t aclp*, *int *trivialp*);

DESCRIPTION

The **acl_is_trivial**() function determines whether the ACL pointed to by the argument *acl* is trivial. Upon successful completion, the location referred to by the argument *trivialp* will be set to 1, if the ACL *aclp* points to is trivial, or 0 if it's not.

ACL is trivial if it can be fully expressed as a file mode without losing any access rules. For POSIX.1e ACLs, ACL is trivial if it has the three required entries, one for owner, one for owning group, and one for other. For NFSv4 ACLs, ACL is trivial if it is identical to the ACL generated by **acl_strip_np**(*3*). Files that have non-trivial ACL have a plus sign appended after mode bits in "ls -l" output.

RETURN VALUES

The **acl_get_tag_type**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

SEE ALSO

acl(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_is_trivial_np**() function was added in FreeBSD 8.0.

AUTHORS

Edward Tomasz Napierala <*trasz@FreeBSD.org*>

NAME

acl_set_entry_type_np - set NFSv4 ACL entry type

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_set_entry_type_np(*acl_entry_t entry_d*, *acl_entry_type_t entry_type*);

DESCRIPTION

The **acl_set_entry_type_np**() function is a non-portable call that sets the type of the NFSv4 ACL entry *entry_d* to the value referred to by *entry_type*.

Valid values are:

ACL_ENTRY_TYPE_ALLOW	allow type entry
ACL_ENTRY_TYPE_DENY	deny type entry

This call brands the ACL as NFSv4.

RETURN VALUES

The **acl_set_entry_type_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_set_entry_type_np**() function fails if:

[EINVAL]	Argument *entry_d* is not a valid descriptor for an ACL entry. The value pointed to by *entry_type* is not valid. ACL is already branded as POSIX.1e.
[ENOMEM]	The value to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_get_brand_np(3), acl_get_entry_type_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_entry_type_np**() function was added in FreeBSD 8.0.

AUTHORS

The **acl_get_entry_type_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_set_fd, **acl_set_fd_np**, **acl_set_file**, **acl_set_link_np** - set an ACL for a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_set_fd(*int fd*, *acl_t acl*);

int
acl_set_fd_np(*int fd*, *acl_t acl*, *acl_type_t type*);

int
acl_set_file(*const char *path_p*, *acl_type_t type*, *acl_t acl*);

int
acl_set_link_np(*const char *path_p*, *acl_type_t type*, *acl_t acl*);

DESCRIPTION

The **acl_set_fd**(), **acl_set_fd_np**(), **acl_set_file**(), and **acl_set_link_np**() each associate an ACL with an object referred to by *fd* or *path_p*. The **acl_set_fd_np**() and **acl_set_link_np**() functions are not POSIX.1e calls. The **acl_set_fd**() function allows only the setting of ACLs of type ACL_TYPE_ACCESS where as **acl_set_fd_np**() allows the setting of ACLs of any type. The **acl_set_link_np**() function acts on a symlink rather than its target, if the target of the path is a symlink.

Valid values for the *type* argument are:

ACL_TYPE_ACCESS	POSIX.1e access ACL
ACL_TYPE_DEFAULT	POSIX.1e default ACL
ACL_TYPE_NFS4	NFSv4 ACL

Trying to set ACL_TYPE_NFS4 with *acl* branded as POSIX.1e, or ACL_TYPE_ACCESS or ACL_TYPE_DEFAULT with ACL branded as NFSv4, will result in error.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, these functions shall return -1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix, or the object exists and the process does not have appropriate access rights.

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

[EINVAL] Argument *acl* does not point to a valid ACL for this object, or the ACL type specified in *type* is invalid for this object, or there is branding mismatch.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named object does not exist, or the *path_p* argument points to an empty string.

[ENOMEM] Insufficient memory available to fulfill request.

[ENOSPC] The directory or file system that would contain the new ACL cannot be extended, or the file system is out of file allocation resources.

[EOPNOTSUPP] The file system does not support ACL retrieval.

[EROFS] This function requires modification of a file system which is currently read-only.

SEE ALSO

acl(3), acl_delete(3), acl_get(3), acl_get_brand_np(3), acl_valid(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

NAME

acl_set_flagset_np - set the flags of an NFSv4 ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_set_flagset_np(*acl_entry_t entry_d*, *acl_flagset_t flagset_d*);

DESCRIPTION

The **acl_set_flagset_np**() function is a non-portable call that sets the flags of NFSv4 ACL entry *entry_d* with the flags contained in *flagset_d*.

This call brands the ACL as NFSv4.

RETURN VALUES

The **acl_set_flagset_np**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_set_flagset_np**() function fails if:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry. ACL is already branded as POSIX.1e.

SEE ALSO

acl(3), acl_add_flag_np(3), acl_clear_flags_np(3), acl_delete_flag_np(3), acl_get_brand_np(3), acl_get_flagset_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_set_flagset_np**() function was added in FreeBSD 8.0.

AUTHORS

The **acl_set_flagset_np**() function was written by Edward Tomasz Napierala <*trasz@FreeBSD.org*>.

NAME

acl_set_permset - set the permissions of an ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_set_permset(*acl_entry_t entry_d*, *acl_permset_t permset_d*);

DESCRIPTION

The **acl_set_permset**() function is a POSIX.1e call that sets the permissions of ACL entry *entry_d* with the permissions contained in *permset_d*.

RETURN VALUES

The **acl_set_permset**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_set_permset**() function fails if:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry.

SEE ALSO

acl(3), acl_add_perm(3), acl_clear_perms(3), acl_delete_perm(3), acl_get_permset(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_set_permset**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_set_permset**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_set_qualifier - set ACL tag qualifier

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_set_qualifier(*acl_entry_t entry_d*, *const void *tag_qualifier_p*);

DESCRIPTION

The **acl_set_qualifier**() function is a POSIX.1e call that sets the qualifier of the tag for the ACL entry *entry_d* to the value referred to by *tag_qualifier_p*.

RETURN VALUES

The **acl_set_qualifier**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_set_qualifier**() function fails if:

[EINVAL] Argument *entry_d* is not a valid descriptor for an ACL entry. The tag type of the ACL entry *entry_d* is not ACL_USER or ACL_GROUP. The value pointed to by *tag_qualifier_p* is not valid.

[ENOMEM] The value to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_get_qualifier(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_get_qualifier**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_get_qualifier**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_set_tag_type - set the tag type of an ACL entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_set_tag_type(*acl_entry_t entry_d*, *acl_tag_t tag_type*);

DESCRIPTION

The **acl_set_tag_type**() function is a POSIX.1e call that sets the ACL tag type of ACL entry *entry_d* to the value of *tag_type*.

Valid values are:

ACL_USER_OBJ	Permissions apply to file owner
ACL_USER	Permissions apply to additional user specified by qualifier
ACL_GROUP_OBJ	Permissions apply to file group
ACL_GROUP	Permissions apply to additional group specified by qualifier
ACL_MASK	Permissions specify mask
ACL_OTHER	Permissions apply to other
ACL_OTHER_OBJ	Same as ACL_OTHER
ACL_EVERYONE	Permissions apply to everyone@

Calling **acl_set_tag_type**() with *tag_type* equal to ACL_MASK, ACL_OTHER or ACL_OTHER_OBJ brands the ACL as POSIX.1e. Calling it with ACL_EVERYONE brands the ACL as NFSv4.

RETURN VALUES

The **acl_set_tag_type**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **acl_set_tag_type**() function fails if:

[EINVAL]	Argument *entry_d* is not a valid descriptor for an ACL entry. Argument *tag_type* is not a valid ACL tag type. ACL is already branded differently.

SEE ALSO

acl(3), acl_get_brand_np(3), acl_get_tag_type(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_set_tag_type**() function was added in FreeBSD 5.0.

AUTHORS

The **acl_set_tag_type**() function was written by Chris D. Faulhaber <*jedgar@fxp.org*>.

NAME

acl_strip_np - strip extended entries from an ACL

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

acl_t
acl_strip_np(*const acl_t acl*, *int recalculate_mask*);

DESCRIPTION

The **acl_strip_np**() function returns a pointer to a trivial ACL computed from the ACL pointed to by the argument *acl*.

This function may cause memory to be allocated. The caller should free any releasable memory, when the new ACL is no longer required, by calling acl_free(3) with the *(void*)acl_t* as an argument.

Any existing ACL pointers that refer to the ACL referred to by *acl* shall continue to refer to the ACL.

RETURN VALUES

Upon successful completion, this function shall return a pointer to the newly allocated ACL. Otherwise, a value of *(acl_t)NULL* shall be returned, and *errno* shall be set to indicate the error.

ERRORS

If any of the following conditions occur, the **acl_init**() function shall return a value of *(acl_t)NULL* and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL.

[ENOMEM] The *acl_t* to be returned requires more memory than is allowed by the hardware or system-imposed memory management constraints.

SEE ALSO

acl(3), acl_is_trivial_np(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-

platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0. The **acl_strip_np**() function was added in FreeBSD 8.0.

AUTHORS

Edward Tomasz Napierala <*trasz@FreeBSD.org*>

NAME

acl_to_text, **acl_to_text_np** - convert an ACL to text

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

*char **
acl_to_text(*acl_t acl*, *ssize_t *len_p*);

*char **
acl_to_text_np(*acl_t acl*, *ssize_t *len_p*, *int flags*);

DESCRIPTION

The **acl_to_text**() and **acl_to_text_np**() functions translate the ACL pointed to by argument *acl* into a NULL terminated character string. If the pointer *len_p* is not NULL, then the function shall return the length of the string (not including the NULL terminator) in the location pointed to by *len_p*. If the ACL is POSIX.1e, the format of the text string returned by **acl_to_text**() shall be the POSIX.1e long ACL form. If the ACL is NFSv4, the format of the text string shall be the compact form, unless the *ACL_TEXT_VERBOSE* flag is given.

The flags specified are formed by *or*'ing the following values

ACL_TEXT_VERBOSE	Format ACL using verbose form
ACL_TEXT_NUMERIC_IDS	Do not resolve IDs into user or group names
ACL_TEXT_APPEND_ID	In addition to user and group names, append numeric IDs

This function allocates any memory necessary to contain the string and returns a pointer to the string. The caller should free any releasable memory, when the new string is no longer required, by calling acl_free(3) with the *(void*)char* as an argument.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, the function shall return a pointer to the long text form of an ACL. Otherwise, a value of *(char*)NULL* shall be returned and *errno* shall be set to indicate the error.

ERRORS

If any of the following conditions occur, the **acl_to_text**() function shall return a value of *(acl_t)NULL* and set *errno* to the corresponding value:

[EINVAL] Argument *acl* does not point to a valid ACL.

The ACL denoted by *acl* contains one or more improperly formed ACL entries, or for some other reason cannot be translated into a text form of an ACL.

[ENOMEM] The character string to be returned requires more memory than is allowed by the hardware or software-imposed memory management constraints.

SEE ALSO

acl(3), acl_free(3), acl_from_text(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

BUGS

The **acl_from_text**() and **acl_to_text**() functions rely on the getpwent(3) library calls to manage username and uid mapping, as well as the getgrent(3) library calls to manage groupname and gid mapping. These calls are not thread safe, and so transitively, neither are **acl_from_text**() and **acl_to_text**(). These functions may also interfere with stateful calls associated with the **getpwent**() and **getgrent**() calls.

NAME

acl_valid, **acl_valid_fd_np**, **acl_valid_file_np**, **acl_valid_link_np** - validate an ACL

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

int
acl_valid(*acl_t acl*);

int
acl_valid_fd_np(*int fd*, *acl_type_t type*, *acl_t acl*);

int
acl_valid_file_np(*const char *path_p*, *acl_type_t type*, *acl_t acl*);

int
acl_valid_link_np(*const char *path_p*, *acl_type_t type*, *acl_t acl*);

DESCRIPTION

These functions check that the ACL referred to by the argument *acl* is valid. The POSIX.1e routine, **acl_valid**(), checks this validity only with POSIX.1e ACL semantics, and irrespective of the context in which the ACL is to be used. The non-portable forms, **acl_valid_fd_np**(), **acl_valid_file_np**(), and **acl_valid_link_np**() allow an ACL to be checked in the context of a specific acl type, *type*, and file system object. In environments where additional ACL types are supported than just POSIX.1e, this makes more sense. Whereas **acl_valid_file_np**() will follow the symlink if the specified path is to a symlink, **acl_valid_link_np**() will not.

For POSIX.1e semantics, the checks include:

- The three required entries (ACL_USER_OBJ, ACL_GROUP_OBJ, and ACL_OTHER) shall exist exactly once in the ACL. If the ACL contains any ACL_USER, ACL_GROUP, or any other implementation-defined entries in the file group class then one ACL_MASK entry shall also be required. The ACL shall contain at most one ACL_MASK entry.

- The qualifier field shall be unique among all entries of the same POSIX.1e ACL facility defined tag type. The tag type field shall contain valid values including any implementation-defined values.

Validation of the values of the qualifier field is implementation-defined.

The POSIX.1e **acl_valid**() function may reorder the ACL for the purposes of verification; the non-portable validation functions will not.

IMPLEMENTATION NOTES

FreeBSD's support for POSIX.1e interfaces and features is still under development at this time.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, these functions shall return -1 and set *errno* to the corresponding value:

[EACCES] Search permission is denied for a component of the path prefix, or the object exists and the process does not have appropriate access rights.

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

[EINVAL] Argument *acl* does not point to a valid ACL.

One or more of the required ACL entries is not present in *acl*.

The ACL contains entries that are not unique.

The file system rejects the ACL based on fs-specific semantics issues.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named object does not exist, or the *path_p* argument points to an empty string.

[ENOMEM] Insufficient memory available to fulfill request.

[EOPNOTSUPP] The file system does not support ACL retrieval.

SEE ALSO

acl(3), acl_get(3), acl_init(3), acl_set(3), posix1e(3)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0, and development continues.

AUTHORS

Robert N M Watson

NAME

acl - introduction to the POSIX.1e/NFSv4 ACL security API

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>

DESCRIPTION

FreeBSD permits file systems to export Access Control Lists via the VFS, and provides a library for userland access to and manipulation of these ACLs. FreeBSD supports POSIX.1e and NFSv4 ACLs, but not all file systems provide support for ACLs, and some may require that ACL support be explicitly enabled by the administrator. The library calls include routines to allocate, duplicate, retrieve, set, and validate ACLs associated with file objects. As well as the POSIX.1e routines, there are a number of non-portable extensions defined that allow for ACL semantics alternative to POSIX.1e, such as NFSv4. Where routines are non-standard, they are suffixed with _np to indicate that they are not portable.

POSIX.1e describes a set of ACL manipulation routines to manage the contents of ACLs, as well as their relationships with files; almost all of these support routines are implemented in FreeBSD.

Available functions, sorted by behavior, include:

acl_add_flag_np()
: This function is described in acl_add_flag_np(3), and may be used to add flags to a flagset.

acl_add_perm()
: This function is described in acl_add_perm(3), and may be used to add permissions to a permission set.

acl_calc_mask()
: This function is described in acl_calc_mask(3), and may be used to calculate and set the permissions associated with the ACL_MASK entry.

acl_clear_flags_np()
: This function is described in acl_clear_flags_np(3), and may be used to clear all flags from a flagset.

acl_clear_perms()

This function is described in acl_clear_perms(3), and may be used to clear all permissions from a permission set.

acl_copy_entry()
This function is described in acl_copy_entry(3), and may be used to copy the contents of an ACL entry.

acl_create_entry(), **acl_create_entry_np**()
These functions are described in acl_create_entry(3), and may be used to create an empty entry in an ACL.

acl_delete_def_file(), **acl_delete_def_link_np**(), **acl_delete_fd_np**(), **acl_delete_file_np**(), **acl_delete_link_np**()
These functions are described in acl_delete(3), and may be used to delete ACLs from file system objects.

acl_delete_entry(), **acl_delete_entry_np**(),
This functions are described in acl_delete_entry(3), and may be used to delete an entry from an ACL.

acl_delete_flag_np()
This function is described in acl_delete_flag_np(3), and may be used to delete flags from a flagset.

acl_delete_perm()
This function is described in acl_delete_perm(3), and may be used to delete permissions from a permset.

acl_dup()
This function is described in acl_dup(3), and may be used to duplicate an ACL structure.

acl_free()
This function is described in acl_free(3), and may be used to free userland working ACL storage.

acl_from_text()
This function is described in acl_from_text(3), and may be used to convert a text-form ACL into working ACL state, if the ACL has POSIX.1e or NFSv4 semantics.

acl_get_brand_np()
This function is described in acl_get_brand_np(3) and may be used to determine whether the

ACL has POSIX.1e or NFSv4 semantics.

acl_get_entry()
This function is described in acl_get_entry(3), and may be used to retrieve a designated ACL entry from an ACL.

acl_get_fd(), **acl_get_fd_np**(), **acl_get_file**(), **acl_get_link_np**()
These functions are described in acl_get(3), and may be used to retrieve ACLs from file system objects.

acl_get_entry_type_np()
This function is described in acl_get_entry_type_np(3), and may be used to retrieve an ACL type from an ACL entry.

acl_get_flagset_np()
This function is described in acl_get_flagset_np(3), and may be used to retrieve a flagset from an ACL entry.

acl_get_permset()
This function is described in acl_get_permset(3), and may be used to retrieve a permset from an ACL entry.

acl_get_qualifier()
This function is described in acl_get_qualifier(3), and may be used to retrieve the qualifier from an ACL entry.

acl_get_tag_type()
This function is described in acl_get_tag_type(3), and may be used to retrieve the tag type from an ACL entry.

acl_init()
This function is described in acl_init(3), and may be used to allocate a fresh (empty) ACL structure.

acl_is_trivial_np()
This function is described in acl_is_trivial_np(3), and may be used to find out whether ACL is trivial.

acl_set_fd(), **acl_set_fd_np**(), **acl_set_file**(), **acl_set_link_np**()
These functions are described in acl_set(3), and may be used to assign an ACL to a file system

object.

acl_set_entry_type_np()
This function is described in acl_set_entry_type_np(3), and may be used to set the ACL type of an ACL entry.

acl_set_flagset_np()
This function is described in acl_set_flagset_np(3), and may be used to set the flags of an ACL entry from a flagset.

acl_set_permset()
This function is described in acl_set_permset(3), and may be used to set the permissions of an ACL entry from a permset.

acl_set_qualifier()
This function is described in acl_set_qualifier(3), and may be used to set the qualifier of an ACL.

acl_set_tag_type()
This function is described in acl_set_tag_type(3), and may be used to set the tag type of an ACL.

acl_strip_np()
This function is described in acl_strip_np(3), and may be used to remove extended entries from an ACL.

acl_to_text(), **acl_to_text_np**()
These functions are described in acl_to_text(3), and may be used to generate a text-form of a POSIX.1e or NFSv4 semantics ACL.

acl_valid(), **acl_valid_fd_np**(), **acl_valid_file_np**(), **acl_valid_link_np**()
These functions are described in acl_valid(3), and may be used to validate an ACL as correct POSIX.1e-semantics, or as appropriate for a particular file system object regardless of semantics.

Documentation of the internal kernel interfaces backing these calls may be found in acl(9). The syscalls between the internal interfaces and the public library routines may change over time, and as such are not documented. They are not intended to be called directly without going through the library.

SEE ALSO

getfacl(1), setfacl(1), acl_add_flag_np(3), acl_add_perm(3), acl_calc_mask(3), acl_clear_flags_np(3), acl_clear_perms(3), acl_copy_entry(3), acl_create_entry(3), acl_delete_entry(3), acl_delete_flag_np(3), acl_delete_perm(3), acl_dup(3), acl_free(3), acl_from_text(3), acl_get(3), acl_get_brand_np(3),

acl_get_entry_type_np(3), acl_get_flagset_np(3), acl_get_permset(3), acl_get_qualifier(3), acl_get_tag_type(3), acl_init(3), acl_is_trivial_np(3), acl_set(3), acl_set_entry_type_np(3), acl_set_flagset_np(3), acl_set_permset(3), acl_set_qualifier(3), acl_set_tag_type(3), acl_strip_np(3), acl_to_text(3), acl_valid(3), posix1e(3), acl(9)

STANDARDS

POSIX.1e assigns security labels to all objects, extending the security functionality described in POSIX.1. These additional labels provide fine-grained discretionary access control, fine-grained capabilities, and labels necessary for mandatory access control. POSIX.2c describes a set of userland utilities for manipulating these labels.

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0; FreeBSD 5.0 was the first version to include a complete ACL implementation based on extended attributes for the UFS and UFS2 file systems. NFSv4 ACL support was introduced in FreeBSD 8.0.

The getfacl(1) and setfacl(1) utilities describe the user tools that permit direct manipulation of complete file ACLs.

AUTHORS

Robert N M Watson

NAME

adjtime - correct the time to allow synchronization of the system clock

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/time.h>

int
adjtime(*const struct timeval *delta*, *struct timeval *olddelta*);

DESCRIPTION

The **adjtime**() system call makes small adjustments to the system time, as returned by gettimeofday(2), advancing or retarding it by the time specified by the timeval *delta*. If *delta* is negative, the clock is slowed down by incrementing it more slowly than normal until the correction is complete. If *delta* is positive, a larger increment than normal is used. The skew used to perform the correction is generally a fraction of one percent. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to **adjtime**() may not be finished when **adjtime**() is called again. If *olddelta* is not a null pointer, the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The **adjtime**() system call is restricted to the super-user.

RETURN VALUES

The **adjtime**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **adjtime**() system call will fail if:

[EFAULT] An argument points outside the process's allocated address space.

[EPERM] The process's effective user ID is not that of the super-user.

SEE ALSO

date(1), gettimeofday(2), timed(8), timedc(8)

R. Gusella and S. Zatti, *TSP: The Time Synchronization Protocol for UNIX 4.3BSD*.

HISTORY

The **adjtime**() system call appeared in 4.3BSD.

NAME

aio_cancel - cancel an outstanding asynchronous I/O operation (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
aio_cancel(*int fildes*, *struct aiocb *iocb*);

DESCRIPTION

The **aio_cancel**() system call cancels the outstanding asynchronous I/O request for the file descriptor specified in *fildes*. If *iocb* is specified, only that specific asynchronous I/O request is cancelled.

Normal asynchronous notification occurs for cancelled requests. Requests complete with an error result of ECANCELED.

RESTRICTIONS

The **aio_cancel**() system call does not cancel asynchronous I/O requests for raw disk devices. The **aio_cancel**() system call will always return AIO_NOTCANCELED for file descriptors associated with raw disk devices.

RETURN VALUES

The **aio_cancel**() system call returns -1 to indicate an error, or one of the following:

[AIO_CANCELED]
All outstanding requests meeting the criteria specified were cancelled.

[AIO_NOTCANCELED]
Some requests were not cancelled, status for the requests should be checked with aio_error(2).

[AIO_ALLDONE]
All of the requests meeting the criteria have finished.

ERRORS

An error return from **aio_cancel**() indicates:

[EBADF] The *fildes* argument is an invalid file descriptor.

SEE ALSO

aio_error(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), aio(4)

STANDARDS

The **aio_cancel**() system call is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

HISTORY

The **aio_cancel**() system call first appeared in FreeBSD 3.0. The first functional implementation of **aio_cancel**() appeared in FreeBSD 4.0.

AUTHORS

This manual page was originally written by Wes Peters <*wes@softweyr.com*>. Christopher M Sedore <*cmsedore@maxwell.syr.edu*> updated it when **aio_cancel**() was implemented for FreeBSD 4.0.

NAME

aio_error - retrieve error status of asynchronous I/O operation (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
aio_error(*const struct aiocb *iocb*);

DESCRIPTION

The **aio_error**() system call returns the error status of the asynchronous I/O request associated with the structure pointed to by *iocb*.

RETURN VALUES

If the asynchronous I/O request has completed successfully, **aio_error**() returns 0. If the request has not yet completed, EINPROGRESS is returned. If the request has completed unsuccessfully the error status is returned as described in read(2), write(2), or fsync(2). On failure, **aio_error**() returns -1 and sets errno to indicate the error condition.

ERRORS

The **aio_error**() system call will fail if:

[EINVAL] The *iocb* argument does not reference an outstanding asynchronous I/O request.

SEE ALSO

aio_cancel(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), fsync(2), read(2), write(2), aio(4)

STANDARDS

The **aio_error**() system call is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

HISTORY

The **aio_error**() system call first appeared in FreeBSD 3.0.

AUTHORS

This manual page was written by Wes Peters <*wes@softweyr.com*>.

NAME

aio_fsync - asynchronous file synchronization (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
aio_fsync(*int op*, *struct aiocb *iocb*);

DESCRIPTION

The **aio_fsync**() system call allows the calling process to move all modified data associated with the descriptor *iocb->aio_fildes* to a permanent storage device. The call returns immediately after the synchronization request has been enqueued to the descriptor; the synchronization may or may not have completed at the time the call returns.

The *op* argument can only be set to O_SYNC to cause all currently queued I/O operations to be completed as if by a call to fsync(2).

If _POSIX_PRIORITIZED_IO is defined, and the descriptor supports it, then the enqueued operation is submitted at a priority equal to that of the calling process minus *iocb->aio_reqprio*.

The *iocb* pointer may be subsequently used as an argument to **aio_return**() and **aio_error**() in order to determine return or error status for the enqueued operation while it is in progress.

If the request could not be enqueued (generally due to invalid arguments), the call returns without having enqueued the request.

The *iocb->aio_sigevent* structure can be used to request notification of the operation's completion as described in aio(4).

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *iocb* must remain valid until the operation has completed.

The asynchronous I/O control buffer *iocb* should be zeroed before the **aio_fsync**() call to avoid passing bogus context information to the kernel.

Modification of the Asynchronous I/O Control Block structure is not allowed while the request is queued.

RETURN VALUES

The **aio_fsync**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **aio_fsync**() system call will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

[EINVAL] The asynchronous notification method in *iocb->aio_sigevent.sigev_notify* is invalid or not supported.

[EOPNOTSUPP] Asynchronous file synchronization operations on the file descriptor *iocb->aio_fildes* are unsafe and unsafe asynchronous I/O operations are disabled.

[EINVAL] A value of the *op* argument is not set to O_SYNC.

The following conditions may be synchronously detected when the **aio_fsync**() system call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_fsync**() returns -1 and sets *errno* appropriately; otherwise the **aio_return**() system call must be called, and will return -1, and **aio_error**() must be called to determine the actual value that would have been returned in *errno*.

[EBADF] The *iocb->aio_fildes* argument is not a valid descriptor.

[EINVAL] This implementation does not support synchronized I/O for this file.

If the request is successfully enqueued, but subsequently cancelled or an error occurs, the value returned by the **aio_return**() system call is per the read(2) and write(2) system calls, and the value returned by the **aio_error**() system call is one of the error returns from the read(2) or write(2) system calls.

SEE ALSO

aio_cancel(2), aio_error(2), aio_read(2), aio_return(2), aio_suspend(2), aio_waitcomplete(2), aio_write(2), fsync(2), sigevent(3), siginfo(3), aio(4)

STANDARDS

The **aio_fsync**() system call is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

HISTORY

The **aio_fsync**() system call first appeared in FreeBSD 7.0.

NAME

aio_mlock - asynchronous mlock(2) operation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
aio_mlock(*struct aiocb *iocb*);

DESCRIPTION

The **aio_mlock**() system call allows the calling process to lock into memory the physical pages associated with the virtual address range starting at *iocb->aio_buf* for *iocb->aio_nbytes* bytes. The call returns immediately after the locking request has been enqueued; the operation may or may not have completed at the time the call returns.

The *iocb* pointer may be subsequently used as an argument to **aio_return**() and **aio_error**() in order to determine return or error status for the enqueued operation while it is in progress.

If the request could not be enqueued (generally due to aio(4) limits), then the call returns without having enqueued the request.

The *iocb->aio_sigevent* structure can be used to request notification of the operation's completion as described in aio(4).

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *iocb* and the buffer that the *iocb->aio_buf* member of that structure references must remain valid until the operation has completed.

The asynchronous I/O control buffer *iocb* should be zeroed before the **aio_mlock**() call to avoid passing bogus context information to the kernel.

Modifications of the Asynchronous I/O Control Block structure or the memory mapping described by the virtual address range are not allowed while the request is queued.

RETURN VALUES

The **aio_mlock**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **aio_mlock**() system call will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

[EINVAL] The asynchronous notification method in *iocb->aio_sigevent.sigev_notify* is invalid or not supported.

If the request is successfully enqueued, but subsequently cancelled or an error occurs, the value returned by the **aio_return**() system call is per the mlock(2) system call, and the value returned by the **aio_error**() system call is one of the error returns from the mlock(2) system call, or ECANCELED if the request was explicitly cancelled via a call to **aio_cancel**().

SEE ALSO

aio_cancel(2), aio_error(2), aio_return(2), mlock(2), sigevent(3), aio(4)

PORTABILITY

The **aio_mlock**() system call is a FreeBSD extension, and should not be used in portable code.

HISTORY

The **aio_mlock**() system call first appeared in FreeBSD 10.0.

AUTHORS

The system call was introduced by Gleb Smirnoff <*glebius@FreeBSD.org*>.

NAME

aio_read - asynchronous read from a file (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
aio_read(*struct aiocb *iocb*);

DESCRIPTION

The **aio_read**() system call allows the calling process to read *iocb->aio_nbytes* from the descriptor *iocb->aio_fildes* beginning at the offset *iocb->aio_offset* into the buffer pointed to by *iocb->aio_buf*. The call returns immediately after the read request has been enqueued to the descriptor; the read may or may not have completed at the time the call returns.

If _POSIX_PRIORITIZED_IO is defined, and the descriptor supports it, then the enqueued operation is submitted at a priority equal to that of the calling process minus *iocb->aio_reqprio*.

The *iocb->aio_lio_opcode* argument is ignored by the **aio_read**() system call.

The *iocb* pointer may be subsequently used as an argument to **aio_return**() and **aio_error**() in order to determine return or error status for the enqueued operation while it is in progress.

If the request could not be enqueued (generally due to invalid arguments), then the call returns without having enqueued the request.

If the request is successfully enqueued, the value of *iocb->aio_offset* can be modified during the request as context, so this value must not be referenced after the request is enqueued.

The *iocb->aio_sigevent* structure can be used to request notification of the operation's completion as described in aio(4).

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *iocb* and the buffer that the *iocb->aio_buf* member of that structure references must remain valid until the operation has completed.

The asynchronous I/O control buffer *iocb* should be zeroed before the **aio_read**() call to avoid passing

bogus context information to the kernel.

Modifications of the Asynchronous I/O Control Block structure or the buffer contents are not allowed while the request is queued.

If the file offset in *iocb->aio_offset* is past the offset maximum for *iocb->aio_fildes*, no I/O will occur.

RETURN VALUES

The **aio_read**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

DIAGNOSTICS

None.

ERRORS

The **aio_read**() system call will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

[EINVAL] The asynchronous notification method in *iocb->aio_sigevent.sigev_notify* is invalid or not supported.

[EOPNOTSUPP] Asynchronous read operations on the file descriptor *iocb->aio_fildes* are unsafe and unsafe asynchronous I/O operations are disabled.

The following conditions may be synchronously detected when the **aio_read**() system call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_read**() returns -1 and sets *errno* appropriately; otherwise the **aio_return**() system call must be called, and will return -1, and **aio_error**() must be called to determine the actual value that would have been returned in *errno*.

[EBADF] The *iocb->aio_fildes* argument is invalid.

[EINVAL] The offset *iocb->aio_offset* is not valid, the priority specified by *iocb->aio_reqprio* is not a valid priority, or the number of bytes specified by *iocb->aio_nbytes* is not valid.

[EOVERFLOW] The file is a regular file, *iocb->aio_nbytes* is greater than zero, the starting offset in *iocb->aio_offset* is before the end of the file, but is at or beyond the *iocb->aio_fildes* offset maximum.

If the request is successfully enqueued, but subsequently cancelled or an error occurs, the value returned by the **aio_return**() system call is per the read(2) system call, and the value returned by the **aio_error**() system call is either one of the error returns from the read(2) system call, or one of:

[EBADF] The *iocb->aio_fildes* argument is invalid for reading.

[ECANCELED] The request was explicitly cancelled via a call to **aio_cancel**().

[EINVAL] The offset *iocb->aio_offset* would be invalid.

SEE ALSO

aio_cancel(2), aio_error(2), aio_return(2), aio_suspend(2), aio_waitcomplete(2), aio_write(2), sigevent(3), siginfo(3), aio(4)

STANDARDS

The **aio_read**() system call is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

HISTORY

The **aio_read**() system call first appeared in FreeBSD 3.0.

AUTHORS

This manual page was written by Terry Lambert <*terry@whistle.com*>.

BUGS

Invalid information in *iocb->_aiocb_private* may confuse the kernel.

NAME

aio_return - retrieve return status of asynchronous I/O operation (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

ssize_t
aio_return(*struct aiocb *iocb*);

DESCRIPTION

The **aio_return**() system call returns the final status of the asynchronous I/O request associated with the structure pointed to by *iocb*.

The **aio_return**() system call should only be called once, to obtain the final status of an asynchronous I/O operation once it has completed (aio_error(2) returns something other than EINPROGRESS).

RETURN VALUES

If the asynchronous I/O request has completed, the status is returned as described in read(2), write(2), or fsync(2). Otherwise, **aio_return**() returns -1 and sets *errno* to indicate the error condition.

ERRORS

The **aio_return**() system call will fail if:

[EINVAL] The *iocb* argument does not reference a completed asynchronous I/O request.

SEE ALSO

aio_cancel(2), aio_error(2), aio_suspend(2), aio_waitcomplete(2), aio_write(2), fsync(2), read(2), write(2), aio(4)

STANDARDS

The **aio_return**() system call is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

HISTORY

The **aio_return**() system call first appeared in FreeBSD 3.0.

AUTHORS

This manual page was written by Wes Peters <*wes@softweyr.com*>.

NAME

aio_suspend - suspend until asynchronous I/O operations or timeout complete (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
aio_suspend(*const struct aiocb *const iocbs[]*, *int niocb*, *const struct timespec *timeout*);

DESCRIPTION

The **aio_suspend**() system call suspends the calling process until at least one of the specified asynchronous I/O requests have completed, a signal is delivered, or the *timeout* has passed.

The *iocbs* argument is an array of *niocb* pointers to asynchronous I/O requests. Array members containing null pointers will be silently ignored.

If *timeout* is not a null pointer, it specifies a maximum interval to suspend. If *timeout* is a null pointer, the suspend blocks indefinitely. To effect a poll, the *timeout* should point to a zero-value timespec structure.

RETURN VALUES

If one or more of the specified asynchronous I/O requests have completed, **aio_suspend**() returns 0. Otherwise it returns -1 and sets *errno* to indicate the error, as enumerated below.

ERRORS

The **aio_suspend**() system call will fail if:

[EAGAIN] the *timeout* expired before any I/O requests completed.

[EINVAL] The *iocbs* argument contains more asynchronous I/O requests than the *vfs.aio.max_aio_queue_per_proc* sysctl(8) variable, or at least one of the requests is not valid.

[EINTR] the suspend was interrupted by a signal.

SEE ALSO

aio_cancel(2), aio_error(2), aio_return(2), aio_waitcomplete(2), aio_write(2), aio(4)

STANDARDS

The **aio_suspend**() system call is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

HISTORY

The **aio_suspend**() system call first appeared in FreeBSD 3.0.

AUTHORS

This manual page was written by Wes Peters <*wes@softweyr.com*>.

NAME

aio_waitcomplete - wait for the next completion of an aio request

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

ssize_t
aio_waitcomplete(*struct aiocb **iocbp*, *struct timespec *timeout*);

DESCRIPTION

The **aio_waitcomplete**() system call waits for completion of an asynchronous I/O request. Upon completion, **aio_waitcomplete**() returns the result of the function and sets *iocbp* to point to the structure associated with the original request. If an asynchronous I/O request is completed before **aio_waitcomplete**() is called, it returns immediately with the completed request.

If *timeout* is a non-NULL pointer, it specifies a maximum interval to wait for a asynchronous I/O request to complete. If *timeout* is a NULL pointer, **aio_waitcomplete**() waits indefinitely. To effect a poll, the *timeout* argument should be non-NULL, pointing to a zero-valued timeval structure.

The **aio_waitcomplete**() system call also serves the function of **aio_return**(), thus **aio_return**() should not be called for the control block returned in *iocbp*.

RETURN VALUES

If an asynchronous I/O request has completed, *iocbp* is set to point to the control block passed with the original request, and the status is returned as described in read(2), write(2), or fsync(2). On failure, **aio_waitcomplete**() returns -1, sets iocbp to NULL and sets *errno* to indicate the error condition.

ERRORS

The **aio_waitcomplete**() system call fails if:

[EINVAL]	The specified time limit is invalid.
[EAGAIN]	The process has not yet called **aio_read**() or **aio_write**().
[EINTR]	A signal was delivered before the timeout expired and before any asynchronous I/O requests completed.

[EWOULDBLOCK]

[EINPROGRESS] The specified time limit expired before any asynchronous I/O requests completed.

SEE ALSO

aio_cancel(2), aio_error(2), aio_read(2), aio_return(2), aio_suspend(2), aio_write(2), fsync(2), read(2), write(2), aio(4)

STANDARDS

The **aio_waitcomplete**() system call is a FreeBSD-specific extension.

HISTORY

The **aio_waitcomplete**() system call first appeared in FreeBSD 4.0.

AUTHORS

The **aio_waitcomplete**() system call and this manual page were written by Christopher M Sedore <*cmsedore@maxwell.syr.edu*>.

NAME

aio_write - asynchronous write to a file (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
aio_write(*struct aiocb *iocb*);

DESCRIPTION

The **aio_write**() system call allows the calling process to write *iocb->aio_nbytes* from the buffer pointed to by *iocb->aio_buf* to the descriptor *iocb->aio_fildes*. The call returns immediately after the write request has been enqueued to the descriptor; the write may or may not have completed at the time the call returns. If the request could not be enqueued, generally due to invalid arguments, the call returns without having enqueued the request.

If O_APPEND is set for *iocb->aio_fildes*, **aio_write**() operations append to the file in the same order as the calls were made. If O_APPEND is not set for the file descriptor, the write operation will occur at the absolute position from the beginning of the file plus *iocb->aio_offset*.

If _POSIX_PRIORITIZED_IO is defined, and the descriptor supports it, then the enqueued operation is submitted at a priority equal to that of the calling process minus *iocb->aio_reqprio*.

The *iocb* pointer may be subsequently used as an argument to **aio_return**() and **aio_error**() in order to determine return or error status for the enqueued operation while it is in progress.

If the request is successfully enqueued, the value of *iocb->aio_offset* can be modified during the request as context, so this value must not be referenced after the request is enqueued.

The *iocb->aio_sigevent* structure can be used to request notification of the operation's completion as described in aio(4).

RESTRICTIONS

The Asynchronous I/O Control Block structure pointed to by *iocb* and the buffer that the *iocb->aio_buf* member of that structure references must remain valid until the operation has completed.

The asynchronous I/O control buffer *iocb* should be zeroed before the **aio_write**() system call to avoid

passing bogus context information to the kernel.

Modifications of the Asynchronous I/O Control Block structure or the buffer contents are not allowed while the request is queued.

If the file offset in *iocb->aio_offset* is past the offset maximum for *iocb->aio_fildes*, no I/O will occur.

RETURN VALUES

The **aio_write**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **aio_write**() system call will fail if:

[EAGAIN] The request was not queued because of system resource limitations.

[EINVAL] The asynchronous notification method in *iocb->aio_sigevent.sigev_notify* is invalid or not supported.

[EOPNOTSUPP] Asynchronous write operations on the file descriptor *iocb->aio_fildes* are unsafe and unsafe asynchronous I/O operations are disabled.

The following conditions may be synchronously detected when the **aio_write**() system call is made, or asynchronously, at any time thereafter. If they are detected at call time, **aio_write**() returns -1 and sets *errno* appropriately; otherwise the **aio_return**() system call must be called, and will return -1, and **aio_error**() must be called to determine the actual value that would have been returned in *errno*.

[EBADF] The *iocb->aio_fildes* argument is invalid, or is not opened for writing.

[EINVAL] The offset *iocb->aio_offset* is not valid, the priority specified by *iocb->aio_reqprio* is not a valid priority, or the number of bytes specified by *iocb->aio_nbytes* is not valid.

If the request is successfully enqueued, but subsequently canceled or an error occurs, the value returned by the **aio_return**() system call is per the write(2) system call, and the value returned by the **aio_error**() system call is either one of the error returns from the write(2) system call, or one of:

[EBADF] The *iocb->aio_fildes* argument is invalid for writing.

[ECANCELED] The request was explicitly canceled via a call to **aio_cancel**().

[EINVAL] The offset *iocb->aio_offset* would be invalid.

SEE ALSO

aio_cancel(2), aio_error(2), aio_return(2), aio_suspend(2), aio_waitcomplete(2), sigevent(3), siginfo(3), aio(4)

STANDARDS

The **aio_write**() system call is expected to conform to the IEEE Std 1003.1 ("POSIX.1") standard.

HISTORY

The **aio_write**() system call first appeared in FreeBSD 3.0.

AUTHORS

This manual page was written by Wes Peters <*wes@softweyr.com*>.

BUGS

Invalid information in *iocb->_aiocb_private* may confuse the kernel.

NAME

alarm - set signal timer alarm

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

unsigned int
alarm(*unsigned int seconds*);

DESCRIPTION

This interface is made obsolete by setitimer(2).

The **alarm**() function sets a timer to deliver the signal SIGALRM to the calling process after the specified number of *seconds*. If an alarm has already been set with **alarm**() but has not been delivered, another call to **alarm**() will supersede the prior call. The request **alarm**(*0*) voids the current alarm and the signal SIGALRM will not be delivered.

Due to setitimer(2) restriction the maximum number of *seconds* allowed is 100000000.

RETURN VALUES

The return value of **alarm**() is the amount of time left on the timer from a previous call to **alarm**(). If no alarm is currently set, the return value is 0.

SEE ALSO

setitimer(2), sigaction(2), sigsuspend(2), signal(3), sleep(3), ualarm(3), usleep(3)

HISTORY

An **alarm**() function appeared in Version 7 AT&T UNIX.

NAME

alloca - memory allocator

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*void **
alloca(*size_t size*);

DESCRIPTION

The **alloca**() function allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

RETURN VALUES

The **alloca**() function returns a pointer to the beginning of the allocated space.

SEE ALSO

brk(2), calloc(3), getpagesize(3), malloc(3), realloc(3)

HISTORY

The **alloca**() function appeared in Version 32V AT&T UNIX.

BUGS

The **alloca**() function is machine and compiler dependent; its use is discouraged.

The **alloca**() function is slightly unsafe because it cannot ensure that the pointer returned points to a valid and usable block of memory. The allocation made may exceed the bounds of the stack, or even go further into other objects in memory, and **alloca**() cannot determine such an error. Avoid **alloca**() with large unbounded allocations.

NAME

scandir, **alphasort** - scan a directory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <dirent.h>

int
scandir(*const char *dirname*, *struct dirent ***namelist*, *int (*select)(const struct dirent *)*,
*int (*compar)(const struct dirent **, const struct dirent **)*);

int
scandir_b(*const char *dirname*, *struct dirent ***namelist*, *int (*select(rp(const struct dirent *)*,
*int (^compar)(const struct dirent **, const struct dirent **)*);

int
alphasort(*const struct dirent **d1*, *const struct dirent **d2*);

DESCRIPTION

The **scandir**() function reads the directory *dirname* and builds an array of pointers to directory entries using malloc(3). It returns the number of entries in the array. A pointer to the array of directory entries is stored in the location referenced by *namelist*.

The *select* argument is a pointer to a user supplied subroutine which is called by **scandir**() to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* argument is a pointer to a user supplied subroutine which is passed to qsort(3) to sort the completed array. If this pointer is null, the array is not sorted.

The **alphasort**() function is a routine which can be used for the *compar* argument to sort the array alphabetically using strcoll(3).

The memory allocated for the array can be deallocated with free(3), by freeing each pointer in the array and then the array itself.

The **scandir_b**() function behaves in the same way as **scandir**(), but takes blocks as arguments instead of

function pointers and calls **qsort_b**() rather than **qsort**().

DIAGNOSTICS

Returns -1 if the directory cannot be opened for reading or if malloc(3) cannot allocate enough memory to hold all the data structures.

SEE ALSO

directory(3), malloc(3), qsort(3), strcoll(3), dir(5)

HISTORY

The **scandir**() and **alphasort**() functions appeared in 4.2BSD.

NAME

arc4random, **arc4random_buf**, **arc4random_uniform** - random number generator

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

uint32_t
arc4random(*void*);

void
arc4random_buf(*void *buf*, *size_t nbytes*);

uint32_t
arc4random_uniform(*uint32_t upper_bound*);

DESCRIPTION

This family of functions provides higher quality data than those described in rand(3), random(3), and rand48(3).

Use of these functions is encouraged for almost all random number consumption because the other interfaces are deficient in either quality, portability, standardization, or availability. These functions can be called in almost all coding environments, including pthreads(3) and chroot(2).

High quality 32-bit pseudo-random numbers are generated very quickly. On each call, a cryptographic pseudo-random number generator is used to generate a new result. One data pool is used for all consumers in a process, so that consumption under program flow can act as additional stirring. The subsystem is re-seeded from the kernel random number subsystem using getentropy(2) on a regular basis, and also upon fork(2).

The **arc4random**() function returns a single 32-bit value. The **arc4random**() function returns pseudo-random numbers in the range of 0 to (2**32)-1, and therefore has twice the range of rand(3) and random(3).

arc4random_buf() fills the region *buf* of length *nbytes* with random data.

arc4random_uniform() will return a single 32-bit value, uniformly distributed but less than *upper_bound*. This is recommended over constructions like "arc4random() % upper_bound" as it avoids

"modulo bias" when the upper bound is not a power of two. In the worst case, this function may consume multiple iterations to ensure uniformity; see the source code to understand the problem and solution.

RETURN VALUES

These functions are always successful, and no return value is reserved to indicate an error.

EXAMPLES

The following produces a drop-in replacement for the traditional **rand**() and **random**() functions using **arc4random**():

```
#define foo4random() (arc4random() % ((unsigned)RAND_MAX + 1))
```

SEE ALSO

rand(3), rand48(3), random(3)

HISTORY

These functions first appeared in OpenBSD 2.1.

The original version of this random number generator used the RC4 (also known as ARC4) algorithm. In OpenBSD 5.5 it was replaced with the ChaCha20 cipher, and it may be replaced again in the future as cryptographic techniques advance. A good mnemonic is "A Replacement Call for Random".

The **arc4random**() random number generator was first introduced in FreeBSD 2.2.6. The ChaCha20 based implementation was introduced in FreeBSD 12.0, with obsolete stir and addrandom interfaces removed at the same time.

NAME

asctime, **asctime_r**, **ctime**, **ctime_r**, **difftime**, **gmtime**, **gmtime_r**, **localtime**, **localtime_r**, **mktime**, **timegm** - transform binary date and time values

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

*extern char *tzname[2]*;

*char **
ctime(*const time_t *clock*);

double
difftime(*time_t time1*, *time_t time0*);

*char **
asctime(*const struct tm *tm*);

*struct tm **
localtime(*const time_t *clock*);

*struct tm **
gmtime(*const time_t *clock*);

time_t
mktime(*struct tm *tm*);

time_t
timegm(*struct tm *tm*);

*char **
ctime_r(*const time_t *clock*, *char *buf*);

*struct tm **
localtime_r(*const time_t *clock*, *struct tm *result*);

*struct tm **

gmtime_r(*const time_t *clock*, *struct tm *result*);

*char **
asctime_r(*const struct tm *tm*, *char *buf*);

DESCRIPTION

The functions **ctime**(), **gmtime**() and **localtime**() all take as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970; see time(3)).

The function **localtime**() converts the time value pointed at by *clock*, and returns a pointer to a "*struct tm*" (described below) which contains the broken-out time information for the value after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the TZ environment variable (see tzset(3)). The function **localtime**() uses tzset(3) to initialize time conversion information if tzset(3) has not already been called by the process.

After filling in the tm structure, **localtime**() sets the *tm_isdst*'th element of *tzname* to a pointer to an ASCII string that is the time zone abbreviation to be used with **localtime**()'s return value.

The function **gmtime**() similarly converts the time value, but without any time zone adjustment, and returns a pointer to a tm structure (described below).

The **ctime**() function adjusts the time value for the current time zone in the same manner as **localtime**(), and returns a pointer to a 26-character string of the form:

Thu Nov 24 18:22:48 1986\n\0

All the fields have constant width.

The **ctime_r**() function provides the same functionality as **ctime**() except the caller must provide the output buffer *buf* to store the result, which must be at least 26 characters long. The **localtime_r**() and **gmtime_r**() functions provide the same functionality as **localtime**() and **gmtime**() respectively, except the caller must provide the output buffer *result*.

The **asctime**() function converts the broken down time in the structure *tm* pointed at by **tm* to the form shown in the example above.

The **asctime_r**() function provides the same functionality as **asctime**() except the caller provide the output buffer *buf* to store the result, which must be at least 26 characters long.

The functions **mktime**() and **timegm**() convert the broken-down time in the structure pointed to by tm

into a time value with the same encoding as that of the values returned by the time(3) function (that is, seconds from the Epoch, UTC). The **mktime**() function interprets the input structure according to the current timezone setting (see tzset(3)). The **timegm**() function interprets the input structure as representing Universal Coordinated Time (UTC).

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges, and will be normalized if needed. For example, October 40 is changed into November 9, a *tm_hour* of -1 means 1 hour before midnight, *tm_mday* of 0 means the day preceding the current month, and *tm_mon* of -2 means 2 months before January of *tm_year*. (A positive or zero value for *tm_isdst* causes **mktime**() to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the specified time, respectively. A negative value for *tm_isdst* causes the **mktime**() function to attempt to divine whether summer time is in effect for the specified time. The *tm_isdst* and *tm_gmtoff* members are forced to zero by **timegm**().)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. The **mktime**() function returns the specified calendar time; if the calendar time cannot be represented, it returns -1;

The **difftime**() function returns the difference between two calendar times, (*time1 - time0*), expressed in seconds.

External declarations as well as the tm structure definition are in the <*time.h*> include file. The tm structure includes at least the following fields:

```
int tm_sec;     /* seconds (0 - 60) */
int tm_min;     /* minutes (0 - 59) */
int tm_hour;    /* hours (0 - 23) */
int tm_mday;    /* day of month (1 - 31) */
int tm_mon;     /* month of year (0 - 11) */
int tm_year;    /* year - 1900 */
int tm_wday;    /* day of week (Sunday = 0) */
int tm_yday;    /* day of year (0 - 365) */
int tm_isdst;   /* is summer time in effect? */
char *tm_zone;  /* abbreviation of timezone name */
long tm_gmtoff; /* offset from UTC in seconds */
```

The field *tm_isdst* is non-zero if summer time is in effect.

The field *tm_gmtoff* is the offset (in seconds) of the time represented from UTC, with positive values indicating east of the Prime Meridian.

SEE ALSO

date(1), gettimeofday(2), getenv(3), time(3), tzset(3), tzfile(5)

STANDARDS

The **asctime**(), **ctime**(), **difftime**(), **gmtime**(), **localtime**(), and **mktime**() functions conform to ISO/IEC 9899:1990 ("ISO C90"), and conform to ISO/IEC 9945-1:1996 ("POSIX.1") provided the selected local timezone does not contain a leap-second table (see zic(8)).

The **asctime_r**(), **ctime_r**(), **gmtime_r**(), and **localtime_r**() functions are expected to conform to ISO/IEC 9945-1:1996 ("POSIX.1") (again provided the selected local timezone does not contain a leap-second table).

The **timegm**() function is not specified by any standard; its function cannot be completely emulated using the standard functions described above.

HISTORY

This manual page is derived from the time package contributed to Berkeley by Arthur Olson and which appeared in 4.3BSD.

BUGS

Except for **difftime**(), **mktime**(), and the **_r**() variants of the other functions, these functions leave their result in an internal static object and return a pointer to that object. Subsequent calls to these function will modify the same object.

The C Standard provides no mechanism for a program to modify its current local timezone setting, and the POSIX-standard method is not reentrant. (However, thread-safe implementations are provided in the POSIX threaded environment.)

The *tm_zone* field of a returned *tm* structure points to a static array of characters, which will also be overwritten by any subsequent calls (as well as by subsequent calls to tzset(3) and tzsetwall(3)).

Use of the external variable *tzname* is discouraged; the *tm_zone* entry in the tm structure is preferred.

NAME

printf_l, **asprintf_l**, **fprintf_l**, **snprintf_l**, **sprintf_l**, **vasprintf_l**, **vfprintf_l**, **vprintf_l**, **vsnprintf_l**, **vsprintf_l** - formatted output conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <xlocale.h>

int
printf_l(*locale_t loc*, *const char * restrict format*, ...);

int
asprintf_l(*char **ret*, *locale_t loc*, *const char * format*, ...);

int
fprintf_l(*FILE * restrict stream*, *locale_t loc*, *const char * restrict format*, ...);

int
snprintf_l(*char * restrict str*, *size_t size*, *locale_t loc*, *const char * restrict format*, ...);

int
sprintf_l(*char * restrict str*, *locale_t loc*, *const char * restrict format*, ...);

int
vasprintf_l(*char **ret*, *locale_t loc*, *const char *format*, *va_list ap*);

int
vfprintf_l(*FILE * restrict stream*, *locale_t loc*, *const char * restrict format*, *va_list ap*);

int
vprintf_l(*locale_t loc*, *const char * restrict format*, *va_list ap*);

int
vsnprintf_l(*char * restrict str*, *size_t size*, *locale_t loc*, *const char * restrict format*, *va_list ap*);

int
vsprintf_l(*char * restrict str*, *locale_t loc*, *const char * restrict format*, *va_list ap*);

DESCRIPTION

The above functions are used to convert formatted output in the locale *loc*. They behave in the same way as the versions without the _l suffix, but use the specified locale rather than the global or per-thread locale. See the specific manual pages for more information.

SEE ALSO

printf(3), xlocale(3)

STANDARDS

These functions do not conform to any specific standard so they should be considered as non-portable local extensions.

HISTORY

These functions first appeared in Darwin and were first implemented in FreeBSD 9.1.

NAME

printf, **fprintf**, **sprintf**, **snprintf**, **asprintf**, **dprintf**, **vprintf**, **vfprintf**, **vsprintf**, **vsnprintf**, **vasprintf**, **vdprintf** - formatted output conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
printf(*const char * restrict format*, ...);

int
fprintf(*FILE * restrict stream*, *const char * restrict format*, ...);

int
sprintf(*char * restrict str*, *const char * restrict format*, ...);

int
snprintf(*char * restrict str*, *size_t size*, *const char * restrict format*, ...);

int
asprintf(*char **ret*, *const char *format*, ...);

int
dprintf(*int fd*, *const char * restrict format*, ...);

#include <stdarg.h>

int
vprintf(*const char * restrict format*, *va_list ap*);

int
vfprintf(*FILE * restrict stream*, *const char * restrict format*, *va_list ap*);

int
vsprintf(*char * restrict str*, *const char * restrict format*, *va_list ap*);

int

vsnprintf(*char * restrict str*, *size_t size*, *const char * restrict format*, *va_list ap*);

int
vasprintf(*char **ret*, *const char *format*, *va_list ap*);

int
vdprintf(*int fd*, *const char * restrict format*, *va_list ap*);

DESCRIPTION

The **printf**() family of functions produces output according to a *format* as described below. The **printf**() and **vprintf**() functions write output to stdout, the standard output stream; **fprintf**() and **vfprintf**() write output to the given output *stream*; **dprintf**() and **vdprintf**() write output to the given file descriptor; **sprintf**(), **snprintf**(), **vsprintf**(), and **vsnprintf**() write to the character string *str*; and **asprintf**() and **vasprintf**() dynamically allocate a new string with malloc(3).

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of stdarg(3)) are converted for output.

The **asprintf**() and **vasprintf**() functions set **ret* to be a pointer to a buffer sufficiently large to hold the formatted string. This pointer should be passed to free(3) to release the allocated storage when it is no longer needed. If sufficient space cannot be allocated, **asprintf**() and **vasprintf**() will return -1 and set *ret* to be a NULL pointer.

The **snprintf**() and **vsnprintf**() functions will write at most *size*-1 of the characters printed into the output string (the *size*'th character then gets the terminating '\0'); if the return value is greater than or equal to the *size* argument, the string was too short and some of the printed characters were discarded. The output is always null-terminated, unless *size* is 0.

The **sprintf**() and **vsprintf**() functions effectively assume a *size* of INT_MAX + 1.

The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. The arguments must correspond properly (after type promotion) with the conversion specifier. After the **%**, the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a **$**, specifying the next argument to access. If this field is not provided, the argument following the last argument accessed will be used. Arguments are numbered starting at **1**. If unaccessed arguments in the format string are

interspersed with ones that are accessed the results will be indeterminate.

- Zero or more of the following flags:

  '**#**' The value should be converted to an "alternate form". For **c**, **d**, **i**, **n**, **p**, **s**, and **u** conversions, this option has no effect. For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero. For **x** and **X** conversions, a non-zero result has the string '0x' (or '0X' for **X** conversions) prepended to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be.

  '**0**' (zero) Zero padding. For all conversions except **n**, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (**d**, **i**, **o**, **u**, **i**, **x**, and **X**), the **0** flag is ignored.

  '**-**' A negative field width flag; the converted value is to be left adjusted on the field boundary. Except for **n** conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A **-** overrides a **0** if both are given.

  ' ' (space) A blank should be left before a positive number produced by a signed conversion (**a**, **A**, **d**, **e**, **E**, **f**, **F**, **g**, **G**, or **i**).

  '**+**' A sign must always be placed before a number produced by a signed conversion. A **+** overrides a space if both are used.

  '' (apostrophe)
  Decimal conversions (**d**, **u**, or **i**) or the integral portion of a floating point conversion (**f** or **F**) should be grouped and separated by thousands using the non-monetary separator returned by localeconv(3).

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.

- An optional precision, in the form of a period **.** followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear

for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** conversions.

- An optional length modifier, that specifies the size of the argument. The following length modifiers are valid for the **d**, **i**, **n**, **o**, **u**, **x**, or **X** conversion:

Modifier	**d**, **i**	**o**, **u**, **x**, **X**	**n**
hh	*signed char*	*unsigned char*	*signed char **
h	*short*	*unsigned short*	*short **
l (ell)	*long*	*unsigned long*	*long **
ll (ell ell)	*long long*	*unsigned long long*	*long long **
j	*intmax_t*	*uintmax_t*	*intmax_t **
t	*ptrdiff_t*	(see note)	*ptrdiff_t **
z	(see note)	*size_t*	(see note)
q *(deprecated)*	*quad_t*	*u_quad_t*	*quad_t **

Note: the **t** modifier, when applied to a **o**, **u**, **x**, or **X** conversion, indicates that the argument is of an unsigned type equivalent in size to a *ptrdiff_t*. The **z** modifier, when applied to a **d** or **i** conversion, indicates that the argument is of a signed type equivalent in size to a *size_t*. Similarly, when applied to an **n** conversion, it indicates that the argument is a pointer to a signed type equivalent in size to a *size_t*.

The following length modifier is valid for the **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion:

Modifier	**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**
l (ell)	*double* (ignored, same behavior as without it)
L	*long double*

The following length modifier is valid for the **c** or **s** conversion:

Modifier	**c**	**s**
l (ell)	*wint_t*	*wchar_t **

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '*' or an asterisk followed by one or more decimal digits and a '$' instead of a digit string. In this case, an *int* argument supplies the field width or precision. A negative field width is treated as a left adjustment flag followed by a positive field width; a negative precision is treated as though it were missing. If a single format directive mixes

positional (nn$) and non-positional arguments, the results are undefined.

The conversion specifiers and their meanings are:

diouxX The *int* (or appropriate variant) argument is converted to signed decimal (**d** and **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters "abcdef" are used for **x** conversions; the letters "ABCDEF" are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.

DOU The *long int* argument is converted to signed decimal, unsigned octal, or unsigned decimal, as if the format had been **ld**, **lo**, or **lu** respectively. These conversion characters are deprecated, and will eventually disappear.

eE The *double* argument is rounded and converted in the style [-]*d.ddd*e+-*dd* where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An **E** conversion uses the letter ‘E’ (rather than ‘e’) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, positive and negative infinity are represented as inf and -inf respectively when using the lowercase conversion character, and INF and -INF respectively when using the uppercase conversion character. Similarly, NaN is represented as nan when using the lowercase conversion, and NAN when using the uppercase conversion.

fF The *double* argument is rounded and converted to decimal notation in the style [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

gG The *double* argument is converted in style **f** or **e** (or **F** or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

aA The *double* argument is rounded and converted to hexadecimal notation in the style [-]0x*h.hhhp*[+-]*d*, where the number of digits after the hexadecimal-point character is equal to the precision specification. If the precision is missing, it is taken as enough to represent the floating-point number exactly, and no rounding occurs. If the precision is zero, no hexadecimal-point character appears. The **p** is a literal character ‘p’, and the exponent consists of a positive or negative sign followed by a decimal number representing an exponent of 2. The **A** conversion uses the prefix "0X" (rather than "0x"), the letters "ABCDEF" (rather than "abcdef") to represent the hex digits, and the letter ‘P’ (rather than ‘p’) to separate the mantissa and exponent.

Note that there may be multiple valid ways to represent floating-point numbers in this hexadecimal format. For example, 0x1.92p+1, 0x3.24p+0, 0x6.48p-1, and 0xc.9p-2 are all equivalent. FreeBSD 8.0 and later always prints finite non-zero numbers using ‘1’ as the digit before the hexadecimal point. Zeroes are always represented with a mantissa of 0 (preceded by a ‘-’ if appropriate) and an exponent of +0.

C Treated as **c** with the **l** (ell) modifier.

c The *int* argument is converted to an *unsigned char*, and the resulting character is written.

If the **l** (ell) modifier is used, the *wint_t* argument shall be converted to a *wchar_t*, and the (potentially multi-byte) sequence representing the single wide character is written, including any shift sequences. If a shift sequence is used, the shift state is also restored to the original state after the character.

S Treated as **s** with the **l** (ell) modifier.

s The *char* * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.

If the **l** (ell) modifier is used, the *wchar_t* * argument is expected to be a pointer to an array of wide characters (pointer to a wide string). For each wide character in the string, the (potentially multi-byte) sequence representing the wide character is written, including any shift sequences. If any shift sequence is used, the shift state is also restored to the original state after the string. Wide characters from the array are written up to (but not including) a terminating wide NUL character; if a precision is specified, no more than the number of bytes specified are written (including shift sequences). Partial characters are never written.

If a precision is given, no null character need be present; if the precision is not specified, or is greater than the number of bytes required to render the multibyte representation of the string, the array must contain a terminating wide NUL character.

p The *void* * pointer argument is printed in hexadecimal (as if by '%#x' or '%#lx').

n The number of characters written so far is stored into the integer indicated by the *int* * (or variant) pointer argument. No argument is converted.

m Print the string representation of the error code stored in the errno variable at the beginning of the call, as returned by strerror(3). No argument is taken.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

The decimal point character is defined in the program's locale (category LC_NUMERIC).

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

RETURN VALUES

These functions return the number of characters printed (not including the trailing '\0' used to end output to strings), except for **snprintf**() and **vsnprintf**(), which return the number of characters that would have been printed if the *size* were unlimited (again, not including the final '\0'). These functions return a negative value if an error occurs.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

To print pi to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

To allocate a 128 byte string and print into it:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *newfmt(const char *fmt, ...)
{
        char *p;
        va_list ap;
        if ((p = malloc(128)) == NULL)
                return (NULL);
        va_start(ap, fmt);
        (void) vsnprintf(p, 128, fmt, ap);
        va_end(ap);
        return (p);
}
```

COMPATIBILITY

The conversion formats **%D**, **%O**, and **%U** are not standard and are provided only for backward compatibility. The conversion format **%m** is also not standard and provides the popular extension from the GNU C library.

The effect of padding the **%p** format with zeros (either by the **0** flag or by specifying a precision), and the benign effect (i.e., none) of the **#** flag on **%n** and **%p** conversions, as well as other nonsensical combinations such as **%Ld**, are not standard; such combinations should be avoided.

ERRORS

In addition to the errors documented for the write(2) system call, the **printf**() family of functions may fail if:

[EILSEQ] An invalid wide character code was encountered.

[ENOMEM] Insufficient storage space is available.

[EOVERFLOW] The *size* argument exceeds INT_MAX + 1, or the return value would be too large to be represented by an *int*.

SEE ALSO

printf(1), errno(2), fmtcheck(3), scanf(3), setlocale(3), strerror(3), wprintf(3)

STANDARDS

Subject to the caveats noted in the *BUGS* section below, the **fprintf**(), **printf**(), **sprintf**(), **vprintf**(), **vfprintf**(), and **vsprintf**() functions conform to ANSI X3.159-1989 ("ANSI C89") and ISO/IEC 9899:1999 ("ISO C99"). With the same reservation, the **snprintf**() and **vsnprintf**() functions conform to ISO/IEC 9899:1999 ("ISO C99"), while **dprintf**() and **vdprintf**() conform to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The functions **asprintf**() and **vasprintf**() first appeared in the GNU C library. These were implemented by Peter Wemm <*peter@FreeBSD.org*> in FreeBSD 2.2, but were later replaced with a different implementation from OpenBSD 2.3 by Todd C. Miller <*Todd.Miller@courtesan.com*>. The **dprintf**() and **vdprintf**() functions were added in FreeBSD 8.0. The **%m** format extension first appeared in the GNU C library, and was implemented in FreeBSD 12.0.

BUGS

The **printf** family of functions do not correctly handle multibyte characters in the *format* argument.

SECURITY CONSIDERATIONS

The **sprintf**() and **vsprintf**() functions are easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. Because **sprintf**() and **vsprintf**() assume an infinitely long string, callers must be careful not to overflow the actual space; this is often hard to assure. For safety, programmers should use the **snprintf**() interface instead. For example:

```
void
foo(const char *arbitrary_string, const char *and_another)
{
        char onstack[8];

#ifdef BAD
        /*
         * This first sprintf is bad behavior.  Do not use sprintf!
         */
        sprintf(onstack, "%s, %s", arbitrary_string, and_another);
#else
        /*
         * The following two lines demonstrate better use of
         * snprintf().
         */
        snprintf(onstack, sizeof(onstack), "%s, %s", arbitrary_string,
```

```
            and_another);
#endif
}
```

The **printf**() and **sprintf**() family of functions are also easily misused in a manner allowing malicious users to arbitrarily change a running program's functionality by either causing the program to print potentially sensitive data "left on the stack", or causing it to generate a memory fault or bus error by dereferencing an invalid pointer.

%n can be used to write arbitrary data to potentially carefully-selected addresses. Programmers are therefore strongly advised to never pass untrusted strings as the *format* argument, as an attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if the string was built using a function like **snprintf**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by **printf**().

Always use the proper secure idiom:

```
snprintf(buffer, sizeof(buffer), "%s", string);
```

NAME

at_quick_exit - registers a cleanup function to run on quick exit

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
at_quick_exit(*void (*func)(void)*);

DESCRIPTION

The **at_quick_exit**() function registers a cleanup function to be called when the program exits as a result of calling quick_exit(3). The cleanup functions are called in the reverse order and will not be called if the program exits by calling exit(3), _Exit(3), or abort(3).

RETURN VALUES

The **at_quick_exit**() function returns the value 0 if successful and a non-zero value on failure.

SEE ALSO

exit(3), quick_exit(3)

STANDARDS

The **at_quick_exit**() function conforms to ISO/IEC 9899:2011 ("ISO C11").

NAME

atexit - register a function to be called on exit

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
atexit(*void (*function)(void)*);

int
atexit_b(*void (^function)(void)*);

DESCRIPTION

The **atexit**() function registers the given *function* to be called at program exit, whether via exit(3) or via return from the program's **main**(). Functions so registered are called in reverse order; no arguments are passed.

These functions must not call **exit**(); if it should be necessary to terminate the process while in such a function, the _exit(2) function should be used. (Alternatively, the function may cause abnormal process termination, for example by calling abort(3).)

At least 32 functions can always be registered, and more are allowed as long as sufficient memory can be allocated.

The **atexit_b**() function behaves identically to **atexit**(), except that it takes a block, rather than a function pointer.

RETURN VALUES

The **atexit**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[ENOMEM] No memory was available to add the function to the list. The existing list of functions is unmodified.

[ENOSYS] The **atexit_b**() function was called by a program that did not supply a **_Block_copy**() implementation.

SEE ALSO

at_quick_exit(3), exit(3)

STANDARDS

The **atexit**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

NAME

atof - convert ASCII string to double

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

double
atof(*const char *nptr*);

DESCRIPTION

The **atof**() function converts the initial portion of the string pointed to by *nptr* to *double* representation.

It is equivalent to:

```
strtod(nptr, (char **)NULL);
```

The decimal point character is defined in the program's locale (category LC_NUMERIC).

IMPLEMENTATION NOTES

The **atof**() function is not thread-safe and also not async-cancel-safe.

The **atof**() function has been deprecated by **strtod**() and should not be used in new code.

ERRORS

The function **atof**() need not affect the value of *errno* on an error.

SEE ALSO

atoi(3), atol(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atof**() function conforms to IEEE Std 1003.1-1990 ("POSIX.1"), ISO/IEC 9899:1990 ("ISO C90"), and ISO/IEC 9899:1999 ("ISO C99").

NAME

atoi - convert ASCII string to integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
atoi(*const char *nptr*);

DESCRIPTION

The **atoi**() function converts the initial portion of the string pointed to by *nptr* to *int* representation.

It is equivalent to:

```
(int)strtol(nptr, NULL, 10);
```

The **atoi**() function has been deprecated by **strtol**() and should not be used in new code.

ERRORS

The function **atoi**() need not affect the value of *errno* on an error.

SEE ALSO

atof(3), atol(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atoi**() function conforms to IEEE Std 1003.1-1990 ("POSIX.1"), ISO/IEC 9899:1990 ("ISO C90"), and ISO/IEC 9899:1999 ("ISO C99").

NAME

atol, **atoll** - convert ASCII string to *long* or *long long* integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

long
atol(*const char *nptr*);

long long
atoll(*const char *nptr*);

DESCRIPTION

The **atol**() function converts the initial portion of the string pointed to by *nptr* to *long* integer representation.

It is equivalent to:

```
strtol(nptr, (char **)NULL, 10);
```

The **atoll**() function converts the initial portion of the string pointed to by *nptr* to *long long* integer representation.

It is equivalent to:

```
strtoll(nptr, (char **)NULL, 10);
```

COMPATIBILITY

The FreeBSD implementations of the **atol**() and **atoll**() functions are thin wrappers around **strtol**() and **strtoll**() respectively, so these functions will affect the value of *errno* in the same way that the **strtol**() and **strtoll**() functions are able to. This behavior of **atol**() and **atoll**() is not required by ISO/IEC 9899:1990 ("ISO C90") or ISO/IEC 9899:1999 ("ISO C99"), but it is allowed by all of ISO/IEC 9899:1990 ("ISO C90"), ISO/IEC 9899:1999 ("ISO C99") and IEEE Std 1003.1-2001 ("POSIX.1").

ERRORS

The functions **atol**() and **atoll**() may affect the value of *errno* on an error.

SEE ALSO

atof(3), atoi(3), strtod(3), strtol(3), strtoul(3)

STANDARDS

The **atol**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **atoll**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

auth_destroy, **authnone_create**, **authsys_create**, **authsys_create_default** - library routines for client side remote procedure call authentication

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

void
auth_destroy(*AUTH *auth*);

*AUTH **
authnone_create(*void*);

*AUTH **
authsys_create(*const char *host*, *const uid_t uid*, *const gid_t gid*, *const int len*, *const gid_t *aup_gids*);

*AUTH **
authsys_create_default(*void*);

DESCRIPTION

These routines are part of the RPC library that allows C language programs to make procedure calls on other machines across the network, with desired authentication.

These routines are normally called after creating the *CLIENT* handle. The *cl_auth* field of the *CLIENT* structure should be initialized by the *AUTH* structure returned by some of the following routines. The client's authentication information is passed to the server when the RPC call is made.

Only the NULL and the SYS style of authentication is discussed here.

Routines

auth_destroy() A function macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy**().

authnone_create() Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

authsys_create() Create and return an RPC authentication handle that contains AUTH_SYS authentication information. The *host* argument is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs.

authsys_create_default() Call **authsys_create**() with the appropriate arguments.

SEE ALSO

rpc(3), rpc_clnt_calls(3), rpc_clnt_create(3)

NAME

rpc_soc, **auth_destroy**, **authnone_create**, **authunix_create**, **authunix_create_default**, **callrpc**, **clnt_broadcast**, **clnt_call**, **clnt_control**, **clnt_create**, **clnt_destroy**, **clnt_freeres**, **clnt_geterr**, **clnt_pcreateerror**, **clnt_perrno**, **clnt_perror**, **clnt_spcreateerror**, **clnt_sperrno**, **clnt_sperror**, **clntraw_create**, **clnttcp_create**, **clntudp_bufcreate**, **clntudp_create**, **clntunix_create**, **get_myaddress**, **pmap_getmaps**, **pmap_getport**, **pmap_rmtcall**, **pmap_set**, **pmap_unset**, **registerrpc**, **rpc_createerr**, **svc_destroy**, **svc_fds**, **svc_fdset**, **svc_getargs**, **svc_getcaller**, **svc_getreq**, **svc_getreqset**, **svc_register**, **svc_run**, **svc_sendreply**, **svc_unregister**, **svcerr_auth**, **svcerr_decode**, **svcerr_noproc**, **svcerr_noprog**, **svcerr_progvers**, **svcerr_systemerr**, **svcerr_weakauth**, **svcfd_create**, **svcunixfd_create**, **svcraw_create**, **svcunix_create**, **xdr_accepted_reply**, **xdr_authunix_parms**, **xdr_callhdr**, **xdr_callmsg**, **xdr_opaque_auth**, **xdr_pmap**, **xdr_pmaplist**, **xdr_rejected_reply**, **xdr_replymsg**, **xprt_register**, **xprt_unregister** - library routines for remote procedure calls

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

See *DESCRIPTION* for function declarations.

DESCRIPTION

The svc_*() and clnt_*() functions described in this page are the old, TS-RPC interface to the XDR and RPC library, and exist for backward compatibility. The new interface is described in the pages referenced from rpc(3).

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

Routines that are used for Secure RPC (DES authentication) are described in rpc_secure(3). Secure RPC can be used only if DES encryption is available.

void
auth_destroy(*AUTH *auth*)

A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy**().

*AUTH **
authnone_create()

Create and return an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

*AUTH **
authunix_create(*char *host*, *u_int uid*, *u_int gid*, *int len*, *u_int *aup_gids*)

Create and return an RPC authentication handle that contains UNIX authentication information. The *host* argument is the name of the machine on which the information was created; *uid* is the user's user ID; *gid* is the user's current group ID; *len* and *aup_gids* refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

*AUTH **
authunix_create_default()

Calls **authunix_create**() with the appropriate arguments.

int **callrpc**(*char *host*, *u_long prognum*, *u_long versnum*, *u_long procnum*, *xdrproc_t inproc*, *void *in*, *xdrproc_t outproc*, *void *out*)

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine *host*. The *in* argument is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's arguments, and *outproc* is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of *enum clnt_stat* cast to an integer if it fails. The routine **clnt_perrno**() is handy for translating failure statuses into messages.

Warning: calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp_create**() for restrictions. You do not have control of timeouts or authentication using this routine.

enum clnt_stat
clnt_broadcast(*u_long prognum*, *u_long versnum*, *u_long procnum*, *xdrproc_t inproc*, *char *in*, *xdrproc_t outproc*, *char *out*, *bool_t (*eachresult)(caddr_t, struct sockaddr_in *)*)

Like **callrpc**(), except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls **eachresult**(), whose form is:

bool_t **eachresult**(*caddr_t out*, *struct sockaddr_in *addr*)

where *out* is the same as *out* passed to **clnt_broadcast**(), except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If **eachresult**() returns zero, **clnt_broadcast**() waits for more replies; otherwise it returns with appropriate status.

Warning: broadcast sockets are limited in size to the maximum transfer unit of the data link. For ethernet, this value is 1500 bytes.

enum clnt_stat
clnt_call(*CLIENT *clnt*, *u_long procnum*, *xdrproc_t inproc*, *char *in*, *xdrproc_t outproc*, *char *out*, *struct timeval tout*)

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create**(). The *in* argument is the address of the procedure's argument(s), and *out* is the address of where to place the result(s); *inproc* is used to encode the procedure's arguments, and *outproc* is used to decode the procedure's results; *tout* is the time allowed for results to come back.

void **clnt_destroy**(*CLIENT *clnt*)

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy**(). If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

*CLIENT **
clnt_create(*char *host*, *u_long prog*, *u_long vers*, *char *proto*)

Generic client creation routine. The *host* argument identifies the name of the remote host where the server is located. The *proto* argument indicates which kind of transport protocol to use. The currently supported values for this field are "udp" and "tcp". Default timeouts are set, but can be modified using **clnt_control**().

Warning: Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

bool_t

clnt_control(*CLIENT *cl*, *u_int req*, *char *info*)

A macro used to change or retrieve various information about a client object. The *req* argument indicates the type of operation, and *info* is a pointer to the information. For both UDP and TCP, the supported values of *req* and their argument types and what they do are:

CLSET_TIMEOUT	*struct timeval*	set total timeout
CLGET_TIMEOUT	*struct timeval*	get total timeout

Note: if you set the timeout using **clnt_control**(), the timeout argument passed to **clnt_call**() will be ignored in all future calls.

CLGET_SERVER_ADDR	*struct sockaddr_in*	get server's address

The following operations are valid for UDP only:

CLSET_RETRY_TIMEOUT	*struct timeval*	set the retry timeout
CLGET_RETRY_TIMEOUT	*struct timeval*	get the retry timeout

The retry timeout is the time that UDP RPC waits for the server to reply before retransmitting the request.

bool_t **clnt_freeres**(*CLIENT *clnt*, *xdrproc_t outproc*, *char *out*)

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The *out* argument is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns one if the results were successfully freed, and zero otherwise.

void
clnt_geterr(*CLIENT *clnt*, *struct rpc_err *errp*)

A macro that copies the error structure out of the client handle to the structure at address *errp*.

void
clnt_pcreateerror(*char *s*)

prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string *s* and a colon. A newline is appended at the end of the message. Used when a **clnt_create**(), **clntraw_create**(), **clnttcp_create**(), or **clntudp_create**() call

fails.

void
clnt_perrno(*enum clnt_stat stat*)

Print a message to standard error corresponding to the condition indicated by *stat*. A newline is appended at the end of the message. Used after **callrpc**().

void **clnt_perror**(*CLIENT *clnt*, *char *s*)

Print a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A newline is appended at the end of the message. Used after **clnt_call**().

*char **
clnt_spcreateerror(*char *s*)

Like **clnt_pcreateerror**(), except that it returns a string instead of printing to the standard error.

Bugs: returns pointer to static data that is overwritten on each call.

*char **
clnt_sperrno(*enum clnt_stat stat*)

Take the same arguments as **clnt_perrno**(), but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message.

The **clnt_sperrno**() function is used instead of **clnt_perrno**() if the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with **printf**(), or if a message format different from that supported by **clnt_perrno**() is to be used.

Note: unlike **clnt_sperror**() and **clnt_spcreateerror**(), **clnt_sperrno**() returns pointer to static data, but the result will not get overwritten on each call.

*char **
clnt_sperror(*CLIENT *rpch*, *char *s*)

Like **clnt_perror**(), except that (like **clnt_sperrno**()) it returns a string instead of printing to standard error.

Bugs: returns pointer to static data that is overwritten on each call.

*CLIENT **
clntraw_create(*u_long prognum*, *u_long versnum*)

This routine creates a toy RPC client for the remote program *prognum*, version *versnum*. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see **svcraw_create**(). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

*CLIENT **
clnttcp_create(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*, *int *sockp*, *u_int sendsz*, *u_int recvsz*)

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is zero, then it is set to the actual port that the remote program is listening on (the remote rpcbind(8) service is consulted for this information). The *sockp* argument is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the *sendsz* and *recvsz* arguments; values of zero choose suitable defaults. This routine returns NULL if it fails.

*CLIENT **
clntudp_create(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*, *struct timeval wait*, *int *sockp*)

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote rpcbind(8) service is consulted for this information). The *sockp* argument is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**().

Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

*CLIENT **

clntudp_bufcreate(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*, *struct timeval wait*, *int *sockp*, *unsigned int sendsize*, *unsigned int recosize*)

This routine creates an RPC client for the remote program *prognum*, on *versnum*; the client uses UDP/IP as a transport. The remote program is located at Internet address *addr*. If *addr->sin_port* is zero, then it is set to actual port that the remote program is listening on (the remote rpcbind(8) service is consulted for this information). The *sockp* argument is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**().

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

*CLIENT **
clntunix_create(*struct sockaddr_un *raddr*, *u_long prognum*, *u_long versnum*, *int *sockp*, *u_int sendsz*, *u_int recvsz*)

This routine creates an RPC client for the local program *prognum*, version *versnum*; the client uses UNIX-domain sockets as a transport. The local program is located at the **raddr*. The *sockp* argument is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets *sockp*. Since UNIX-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the *sendsz* and *recvsz* arguments; values of zero choose suitable defaults. This routine returns NULL if it fails.

int
get_myaddress(*struct sockaddr_in *addr*)

Stuff the machine's IP address into *addr*, without consulting the library routines that deal with */etc/hosts*. The port number is always set to **htons**(*PMAPPORT*). Returns zero on success, non-zero on failure.

*struct pmaplist **
pmap_getmaps(*struct sockaddr_in *addr*)

A user interface to the rpcbind(8) service, which returns a list of the current RPC program-to-port mappings on the host located at IP address *addr*. This routine can return NULL. The command "**rpcinfo -p**" uses this routine.

u_short

pmap_getport(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*, *u_long protocol*)

A user interface to the rpcbind(8) service, which returns the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely IPPROTO_UDP or IPPROTO_TCP. A return value of zero means that the mapping does not exist or that the RPC system failed to contact the remote rpcbind(8) service. In the latter case, the global variable *rpc_createerr* contains the RPC status.

enum clnt_stat
pmap_rmtcall(*struct sockaddr_in *addr*, *u_long prognum*, *u_long versnum*, *u_long procnum*, *xdrproc_t inproc*, *char *in*, *xdrproc_t outproc*, *char *out*, *struct timeval tout*, *u_long *portp*)

A user interface to the rpcbind(8) service, which instructs rpcbind(8) on the host at IP address *addr* to make an RPC call on your behalf to a procedure on that host. The *portp* argument will be modified to the program's port number if the procedure succeeds. The definitions of other arguments are discussed in **callrpc**() and **clnt_call**(). This procedure should be used for a "ping" and nothing else. See also **clnt_broadcast**().

bool_t **pmap_set**(*u_long prognum*, *u_long versnum*, *u_long protocol*, *u_short port*)

A user interface to the rpcbind(8) service, which establishes a mapping between the triple (*prognum*, *versnum*, *protocol*) and *port* on the machine's rpcbind(8) service. The value of *protocol* is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns one if it succeeds, zero otherwise. Automatically done by **svc_register**().

bool_t **pmap_unset**(*u_long prognum*, *u_long versnum*)

A user interface to the rpcbind(8) service, which destroys all mapping between the triple (*prognum*, *versnum*, *) and *ports* on the machine's rpcbind(8) service. This routine returns one if it succeeds, zero otherwise.

bool_t **registerrpc**(*u_long prognum*, *u_long versnum*, *u_long procnum*, *char *(*procname)(void)*, *xdrproc_t inproc*, *xdrproc_t outproc*)

Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its argument(s); *progname* should return a pointer to its static result(s); *inproc* is used to decode the arguments while *outproc* is used to encode the results. This routine returns zero if the registration succeeded, -1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create**() for restrictions.

struct rpc_createerr rpc_createerr;

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt_pcreateerror**() to print the reason why.

bool_t **svc_destroy**(*SVCXPRT * xprt*)

A macro that destroys the RPC service transport handle, *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

fd_set svc_fdset;

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a template argument to the select(2) system call. This is only of interest if a service implementor does not call **svc_run**(), but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to select(2)!), yet it may change after calls to **svc_getreqset**() or any creation routines. As well, note that if the process has descriptor limits which are extended beyond FD_SETSIZE, this variable will only be usable for the first FD_SETSIZE descriptors.

int svc_fds;

Similar to *svc_fdset*, but limited to 32 descriptors. This interface is obsoleted by *svc_fdset*.

bool_t **svc_freeargs**(*SVCXPRT *xprt*, *xdrproc_t inproc*, *char *in*)

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs**(). This routine returns 1 if the results were successfully freed, and zero otherwise.

bool_t **svc_getargs**(*SVCXPRT *xprt*, *xdrproc_t inproc*, *char *in*)

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, *xprt*. The *in* argument is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

*struct sockaddr_in **
svc_getcaller(*SVCXPRT *xprt*)

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, *xprt*.

void **svc_getreqset**(*fd_set *rdfds*)

This routine is only of interest if a service implementor does not call **svc_run**(), but instead implements custom asynchronous event processing. It is called when the select(2) system call has determined that an RPC request has arrived on some RPC socket(s); *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

void **svc_getreq**(*int rdfds*)

Similar to **svc_getreqset**(), but limited to 32 descriptors. This interface is obsoleted by **svc_getreqset**().

bool_t **svc_register**(*SVCXPRT *xprt*, *u_long prognum*, *u_long versnum*,
*void (*dispatch)(struct svc_req *, SVCXPRT *)*, *int protocol*)

Associates *prognum* and *versnum* with the service dispatch procedure, **dispatch**(). If *protocol* is zero, the service is not registered with the rpcbind(8) service. If *protocol* is non-zero, then a mapping of the triple (*prognum*, *versnum*, *protocol*) to *xprt->xp_port* is established with the local rpcbind(8) service (generally *protocol* is zero, IPPROTO_UDP or IPPROTO_TCP). The procedure **dispatch**() has the following form:

bool_t **dispatch**(*struct svc_req *request*, *SVCXPRT *xprt*)

The **svc_register**() routine returns one if it succeeds, and zero otherwise.

svc_run()

This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq**() when one arrives. This procedure is usually waiting for a select(2) system call to return.

bool_t **svc_sendreply**(*SVCXPRT *xprt*, *xdrproc_t outproc*, *char *out*)

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The *xprt* argument is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns one if it succeeds, zero otherwise.

void
svc_unregister(*u_long prognum*, *u_long versnum*)

Remove all mapping of the double (*prognum*, *versnum*) to dispatch routines, and of the triple (*prognum*, *versnum*, *) to port number.

void
svcerr_auth(*SVCXPRT *xprt*, *enum auth_stat why*)

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

void
svcerr_decode(*SVCXPRT *xprt*)

Called by a service dispatch routine that cannot successfully decode its arguments. See also **svc_getargs**().

void
svcerr_noproc(*SVCXPRT *xprt*)

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

void
svcerr_noprog(*SVCXPRT *xprt*)

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

void
svcerr_progvers(*SVCXPRT *xprt*, *u_long low_vers*, *u_long high_vers*)

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

void
svcerr_systemerr(*SVCXPRT *xprt*)

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

void
svcerr_weakauth(*SVCXPRT *xprt*)

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient authentication arguments. The routine calls **svcerr_auth**(*xprt*, *AUTH_TOOWEAK*).

*SVCXPRT **
svcraw_create(*void*)

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see **clntraw_create**(). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

*SVCXPRT **
svctcp_create(*int sock*, *u_int send_buf_size*, *u_int recv_buf_size*)

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_fd* is the transport's socket descriptor, and *xprt->xp_port* is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of zero choose suitable defaults.

*SVCXPRT **
svcunix_create(*int sock*, *u_int send_buf_size*, *u_int recv_buf_size*, *char *path*)

This routine creates a UNIX-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK, in which case a new socket is created. The **path* argument is a variable-length file system pathname of at most 104 characters. This file is *not* removed when the socket is closed. The unlink(2) system call must be used to remove the file. Upon completion, *xprt->xp_fd* is the transport's socket

descriptor. This routine returns NULL if it fails. Since UNIX-based RPC uses buffered I/O, users may specify the size of buffers; values of zero choose suitable defaults.

*SVCXPRT **
svcunixfd_create(*int fd*, *u_int sendsize*, *u_int recvsize*)

Create a service on top of any open descriptor. The *sendsize* and *recvsize* arguments indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

*SVCXPRT **
svcfd_create(*int fd*, *u_int sendsize*, *u_int recvsize*)

Create a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. The *sendsize* and *recvsize* arguments indicate sizes for the send and receive buffers. If they are zero, a reasonable default is chosen.

*SVCXPRT **
svcudp_bufcreate(*int sock*, *u_int sendsize*, *u_int recvsize*)

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket *sock*, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, *xprt->xp_fd* is the transport's socket descriptor, and *xprt->xp_port* is the transport's port number. This routine returns NULL if it fails.

This allows the user to specify the maximum packet size for sending and receiving UDP-based RPC messages.

bool_t **xdr_accepted_reply**(*XDR *xdrs*, *struct accepted_reply *ar*)

Used for encoding RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t **xdr_authunix_parms**(*XDR *xdrs*, *struct authunix_parms *aupp*)

Used for describing UNIX credentials. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

void
bool_t **xdr_callhdr**(*XDR *xdrs*, *struct rpc_msg *chdr*)

Used for describing RPC call header messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t **xdr_callmsg**(*XDR *xdrs*, *struct rpc_msg *cmsg*)

Used for describing RPC call messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t **xdr_opaque_auth**(*XDR *xdrs*, *struct opaque_auth *ap*)

Used for describing RPC authentication information messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

struct pmap;
bool_t **xdr_pmap**(*XDR *xdrs*, *struct pmap *regs*)

Used for describing arguments to various rpcbind(8) procedures, externally. This routine is useful for users who wish to generate these arguments without using the **pmap_***() interface.

bool_t **xdr_pmaplist**(*XDR *xdrs*, *struct pmaplist **rp*)

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these arguments without using the **pmap_***() interface.

bool_t **xdr_rejected_reply**(*XDR *xdrs*, *struct rejected_reply *rr*)

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

bool_t **xdr_replymsg**(*XDR *xdrs*, *struct rpc_msg *rmsg*)

Used for describing RPC reply messages. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

void
xprt_register(*SVCXPRT *xprt*)

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

void
xprt_unregister(*SVCXPRT *xprt*)

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable *svc_fds*. Service implementors usually do not need this routine.

SEE ALSO

rpc_secure(3), xdr(3)

Remote Procedure Calls: Protocol Specification.

Remote Procedure Call Programming Guide.

rpcgen Programming Guide.

RPC: Remote Procedure Call Protocol Specification, Sun Microsystems, Inc., USC-ISI, RFC1050.

NAME

bcmp - compare byte string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <strings.h>

int
bcmp(*const void *b1*, *const void *b2*, *size_t len*);

DESCRIPTION

The **bcmp**() function compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *len* bytes long. Zero-length strings are always identical.

The strings may overlap.

SEE ALSO

memcmp(3), strcasecmp(3), strcmp(3), strcoll(3), strxfrm(3), timingsafe_bcmp(3)

HISTORY

A **bcmp**() function first appeared in 4.2BSD. Its prototype existed previously in <*string.h*> before it was moved to <*strings.h*> for IEEE Std 1003.1-2001 ("POSIX.1") compliance.

NAME

bcopy - copy byte string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <strings.h>

void
bcopy(*const void *src*, *void *dst*, *size_t len*);

DESCRIPTION

The **bcopy**() function copies *len* bytes from string *src* to string *dst*. The two strings may overlap. If *len* is zero, no bytes are copied.

SEE ALSO

memccpy(3), memcpy(3), memmove(3), strcpy(3), strncpy(3)

HISTORY

A **bcopy**() function appeared in 4.2BSD. Its prototype existed previously in <*string.h*> before it was moved to <*strings.h*> for IEEE Std 1003.1-2001 ("POSIX.1") compliance.

IEEE Std 1003.1-2008 ("POSIX.1") removes the specification of **bcopy**() and it is marked as LEGACY in IEEE Std 1003.1-2004 ("POSIX.1"). New programs should use memmove(3). If the input and output buffer do not overlap, then memcpy(3) is more efficient. Note that **bcopy**() takes *src* and *dst* in the opposite order from **memmove**() and **memcpy**().

NAME

bind - assign a local protocol address to a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

int
bind(*int s*, *const struct sockaddr *addr*, *socklen_t addrlen*);

DESCRIPTION

The **bind**() system call assigns the local protocol address to a socket. When a socket is created with socket(2) it exists in an address family space but has no protocol address assigned. The **bind**() system call requests that *addr* be assigned to the socket.

NOTES

Binding an address in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using unlink(2)).

The rules used in address binding vary between communication domains. Consult the manual entries in section 4 for detailed information.

For maximum portability, you should always zero the socket address structure before populating it and passing it to **bind**().

RETURN VALUES

The **bind**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **bind**() system call will fail if:

[EAGAIN]	Kernel resources to complete the request are temporarily unavailable.
[EBADF]	The *s* argument is not a valid descriptor.
[EINVAL]	The socket is already bound to an address, and the protocol does not support binding to a new address; or the socket has been shut down.

[EINVAL] The *addrlen* argument is not a valid length for the address family.

[ENOTSOCK] The *s* argument is not a socket.

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

[EADDRINUSE] The specified address is already in use.

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

[EACCES] The requested address is protected, and the current user has inadequate permission to access it.

[EFAULT] The *addr* argument is not in a valid part of the user address space.

The following errors are specific to binding addresses in the UNIX domain.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] A prefix component of the path name does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EROFS] The name would reside on a read-only file system.

[EISDIR] An empty pathname was specified.

SEE ALSO

connect(2), getsockname(2), listen(2), socket(2)

HISTORY

The **bind**() system call appeared in 4.2BSD.

NAME

bindat - assign a local protocol address to a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

#include <fcntl.h>

int
bindat(*int fd*, *int s*, *const struct sockaddr *addr*, *socklen_t addrlen*);

DESCRIPTION

The **bindat**() system call assigns the local protocol address to a socket. When passed the special value AT_FDCWD in the *fd* parameter, the behavior is identical to a call to bind(2). Otherwise, **bindat**() works like the bind(2) system call with two exceptions:

1. It is limited to sockets in the PF_LOCAL domain.

2. If the file path stored in the *sun_path* field of the sockaddr_un structure is a relative path, it is located relative to the directory associated with the file descriptor *fd*.

RETURN VALUES

The **bindat**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **bindat**() system call may fail with the same errors as the bind(2) system call or with the following errors:

[EBADF] The *sun_path* field does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor.

[ENOTDIR] The *sun_path* field is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

bind(2), connectat(2), socket(2), unix(4)

AUTHORS

The **bindat** was developed by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

NAME

bindresvport, **bindresvport_sa** - bind a socket to a privileged IP port

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <rpc/rpc.h>

int
bindresvport(*int sd*, *struct sockaddr_in *sin*);

int
bindresvport_sa(*int sd*, *struct sockaddr *sa*);

DESCRIPTION

The **bindresvport**() and **bindresvport_sa**() functions are used to bind a socket descriptor to a privileged IP port, that is, a port number in the range 0-1023.

If *sin* is a pointer to a *struct sockaddr_in* then the appropriate fields in the structure should be defined. Note that *sin->sin_family* must be initialized to the address family of the socket, passed by *sd*. If *sin->sin_port* is '0' then an anonymous port (in the range 600-1023) will be chosen, and if bind(2) is successful, the *sin->sin_port* will be updated to contain the allocated port.

If *sin* is the NULL pointer, an anonymous port will be allocated (as above). However, there is no way for **bindresvport**() to return the allocated port in this case.

Only root can bind to a privileged port; this call will fail for any other users.

Function prototype of **bindresvport**() is biased to AF_INET socket. The **bindresvport_sa**() function acts exactly the same, with more neutral function prototype. Note that both functions behave exactly the same, and both support AF_INET6 sockets as well as AF_INET sockets.

RETURN VALUES

The **bindresvport**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EPFNOSUPPORT] If second argument was supplied, and address family did not match between

arguments.

The **bindresvport**() function may also fail and set *errno* for any of the errors specified for the calls bind(2), getsockopt(2), or setsockopt(2).

SEE ALSO

bind(2), getsockopt(2), setsockopt(2), ip(4)

NAME

brk, **sbrk** - change data segment size

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
brk(*const void *addr*);

*void **
sbrk(*intptr_t incr*);

DESCRIPTION

The brk() and sbrk() functions are legacy interfaces from before the advent of modern virtual memory management. They are deprecated and not present on the arm64 or riscv architectures. The mmap(2) interface should be used to allocate pages instead.

The **brk**() and **sbrk**() functions are used to change the amount of memory allocated in a process's data segment. They do this by moving the location of the "break". The break is the first address after the end of the process's uninitialized data segment (also known as the "BSS").

The **brk**() function sets the break to *addr*.

The **sbrk**() function raises the break by *incr* bytes, thus allocating at least *incr* bytes of new memory in the data segment. If *incr* is negative, the break is lowered by *incr* bytes.

NOTES

While the actual process data segment size maintained by the kernel will only grow or shrink in page sizes, these functions allow setting the break to unaligned values (i.e., it may point to any address inside the last page of the data segment).

The current value of the program break may be determined by calling **sbrk**(*0*). See also end(3).

The getrlimit(2) system call may be used to determine the maximum permissible size of the data segment. It will not be possible to set the break beyond "*etext + rlim.rlim_max*" where the *rlim.rlim_max* value is returned from a call to **getrlimit**(*RLIMIT_DATA*, *&rlim*). (See end(3) for the definition of *etext*).

RETURN VALUES

The **brk**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

The **sbrk**() function returns the prior break value if successful; otherwise the value (*void* *)-1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **brk**() and **sbrk**() functions will fail if:

[EINVAL]	The requested break value was beyond the beginning of the data segment.
[ENOMEM]	The data segment size limit, as set by setrlimit(2), was exceeded.
[ENOMEM]	Insufficient space existed in the swap area to support the expansion of the data segment.

SEE ALSO

execve(2), getrlimit(2), mmap(2), end(3), free(3), malloc(3)

HISTORY

The **brk**() function appeared in Version 7 AT&T UNIX. FreeBSD 11.0 introduced the arm64 and riscv architectures which do not support **brk**() or **sbrk**().

BUGS

Mixing **brk**() or **sbrk**() with malloc(3), free(3), or similar functions will result in non-portable program behavior.

Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting getrlimit(2).

sbrk() is sometimes used to monitor heap use by calling with an argument of 0. The result is unlikely to reflect actual utilization in combination with an mmap(2) based malloc.

brk() and **sbrk**() are not thread-safe.

NAME

bsearch - binary search of a sorted table

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*void **
bsearch(*const void *key*, *const void *base*, *size_t nmemb*, *size_t size*,
*int (*compar) (const void *, const void *)*);

DESCRIPTION

The **bsearch**() function searches an array of *nmemb* objects, the initial member of which is pointed to by *base*, for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*.

The contents of the array should be in ascending sorted order according to the comparison function referenced by *compar*. The *compar* routine is expected to have two arguments which point to the *key* object and to an array member, in that order, and should return an integer less than, equal to, or greater than zero if the *key* object is found, respectively, to be less than, to match, or be greater than the array member. See the *int_compare* sample function in qsort(3) for a comparison function that is also compatible with **bsearch**().

RETURN VALUES

The **bsearch**() function returns a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

SEE ALSO

db(3), lsearch(3), qsort(3)

STANDARDS

The **bsearch**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

NAME

btowc, **wctob** - convert between wide and single-byte characters

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

wint_t
btowc(*int c*);

int
wctob(*wint_t c*);

#include <wchar.h>
#include <xlocale.h>

wint_t
btowc_l(*int c*, *locale_t loc*);

int
wctob_l(*wint_t c*, *locale_t loc*);

DESCRIPTION

The **btowc**() function converts a single-byte character into a corresponding wide character. If the character is EOF or not valid in the initial shift state, **btowc**() returns WEOF.

The **wctob**() function converts a wide character into a corresponding single-byte character. If the wide character is WEOF or not able to be represented as a single byte in the initial shift state, **wctob**() returns EOF.

The _l-suffixed versions take an explicit locale argument, while the non-suffixed versions use the current global or per-thread locale.

SEE ALSO

mbrtowc(3), multibyte(3), wcrtomb(3)

STANDARDS

The **btowc**() and **wctob**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **btowc**() and **wctob**() functions first appeared in FreeBSD 5.0.

NAME

csio_build, **csio_build_visit**, **csio_decode**, **csio_decode_visit**, **buff_decode**, **buff_decode_visit**, **csio_encode**, **csio_encode_visit**, **buff_encode_visit** - CAM user library SCSI buffer parsing routines

LIBRARY

Common Access Method User Library (libcam, -lcam)

SYNOPSIS

#include <stdio.h>
#include <camlib.h>

int
csio_build(*struct ccb_scsiio *csio*, *uint8_t *data_ptr*, *uint32_t dxfer_len*, *uint32_t flags*, *int retry_count*, *int timeout*, *const char *cmd_spec*, ...);

int
csio_build_visit(*struct ccb_scsiio *csio*, *uint8_t *data_ptr*, *uint32_t dxfer_len*, *uint32_t flags*, *int retry_count*, *int timeout*, *const char *cmd_spec*, *int (*arg_get)(void *hook, char *field_name)*, *void *gethook*);

int
csio_decode(*struct ccb_scsiio *csio*, *const char *fmt*, ...);

int
csio_decode_visit(*struct ccb_scsiio *csio*, *const char *fmt*, *void (*arg_put)(void *hook*, *int letter*, *void *val*, *int count*, *char *name)*, *void *puthook*);

int
buff_decode(*uint8_t *buff*, *size_t len*, *const char *fmt*, ...);

int
buff_decode_visit(*uint8_t *buff*, *size_t len*, *const char *fmt*, *void (*arg_put)(void *, int, void *, int, char *)*, *void *puthook*);

int
csio_encode(*struct ccb_scsiio *csio*, *const char *fmt*, ...);

int
csio_encode_visit(*struct ccb_scsiio *csio*, *const char *fmt*, *int (*arg_get)(void *hook, char *field_name)*, *void *gethook*);

int
buff_encode_visit(*uint8_t *buff*, *size_t len*, *const char *fmt*,
*int (*arg_get)(void *hook, char *field_name)*, *void *gethook*);

DESCRIPTION

The CAM buffer/CDB encoding and decoding routines provide a relatively easy migration path for userland SCSI applications written with the similarly-named *scsireq_** functions from the old FreeBSD SCSI layer.

These functions may be used in new applications, but users may find it easier to use the various SCSI CCB building functions included with the cam(3) library, e.g., **cam_fill_csio**(), **scsi_start_stop**(), and **scsi_read_write**().

csio_build() builds up a *ccb_scsiio* structure based on the information provided in the variable argument list. It gracefully handles a NULL *data_ptr* argument passed to it.

dxfer_len is the length of the data phase; the data transfer direction is determined by the *flags* argument.

data_ptr is the data buffer used during the SCSI data phase. If no data is to be transferred for the SCSI command in question, this should be set to NULL. If there is data to transfer for the command, this buffer must be at least *dxfer_len* long.

flags are the flags defined in <*cam/cam_ccb.h*>:

```
/* Common CCB header */
/* CAM CCB flags */
typedef enum {
    CAM_CDB_POINTER       = 0x00000001,/* The CDB field is a pointer    */
    CAM_QUEUE_ENABLE      = 0x00000002,/* SIM queue actions are enabled */
    CAM_CDB_LINKED        = 0x00000004,/* CCB contains a linked CDB     */
    CAM_SCATTER_VALID     = 0x00000010,/* Scatter/gather list is valid  */
    CAM_DIS_AUTOSENSE     = 0x00000020,/* Disable autosense feature     */
    CAM_DIR_RESV          = 0x00000000,/* Data direction (00:reserved)  */
    CAM_DIR_IN            = 0x00000040,/* Data direction (01:DATA IN)   */
    CAM_DIR_OUT           = 0x00000080,/* Data direction (10:DATA OUT)  */
    CAM_DIR_NONE          = 0x000000C0,/* Data direction (11:no data)   */
    CAM_DIR_MASK          = 0x000000C0,/* Data direction Mask           */
    CAM_SOFT_RST_OP       = 0x00000100,/* Use Soft reset alternative    */
    CAM_ENG_SYNC          = 0x00000200,/* Flush resid bytes on complete */
    CAM_DEV_QFRZDIS       = 0x00000400,/* Disable DEV Q freezing        */
```

```
    CAM_DEV_QFREEZE      = 0x00000800,/* Freeze DEV Q on execution     */
    CAM_HIGH_POWER       = 0x00001000,/* Command takes a lot of power  */
    CAM_SENSE_PTR        = 0x00002000,/* Sense data is a pointer  */
    CAM_SENSE_PHYS       = 0x00004000,/* Sense pointer is physical addr*/
    CAM_TAG_ACTION_VALID  = 0x00008000,/* Use the tag action in this ccb*/
    CAM_PASS_ERR_RECOVER  = 0x00010000,/* Pass driver does err. recovery*/
    CAM_DIS_DISCONNECT    = 0x00020000,/* Disable disconnect                 */
    CAM_SG_LIST_PHYS      = 0x00040000,/* SG list has physical addrs.   */
    CAM_MSG_BUF_PHYS      = 0x00080000,/* Message buffer ptr is physical*/
    CAM_SNS_BUF_PHYS      = 0x00100000,/* Autosense data ptr is physical*/
    CAM_DATA_PHYS         = 0x00200000,/* SG/Buffer data ptrs are phys. */
    CAM_CDB_PHYS          = 0x00400000,/* CDB pointer is physical       */
    CAM_ENG_SGLIST        = 0x00800000,/* SG list is for the HBA engine */

/* Phase cognizant mode flags */
    CAM_DIS_AUTOSRP       = 0x01000000,/* Disable autosave/restore ptrs */
    CAM_DIS_AUTODISC      = 0x02000000,/* Disable auto disconnect       */
    CAM_TGT_CCB_AVAIL     = 0x04000000,/* Target CCB available          */
    CAM_TGT_PHASE_MODE    = 0x08000000,/* The SIM runs in phase mode    */
    CAM_MSGB_VALID        = 0x20000000,/* Message buffer valid          */
    CAM_STATUS_VALID      = 0x40000000,/* Status buffer valid           */
    CAM_DATAB_VALID       = 0x80000000,/* Data buffer valid             */

/* Host target Mode flags */
    CAM_TERM_IO           = 0x20000000,/* Terminate I/O Message sup.    */
    CAM_DISCONNECT        = 0x40000000,/* Disconnects are mandatory     */
    CAM_SEND_STATUS       = 0x80000000,/* Send status after data phase  */
} ccb_flags;
```

Multiple flags should be ORed together. Any of the CCB flags may be used, although it is worth noting several important ones here:

CAM_DIR_IN — This indicates that the operation in question is a read operation. i.e., data is being read from the SCSI device to the user-supplied buffer.

CAM_DIR_OUT — This indicates that the operation is a write operation. i.e., data is being written from the user-supplied buffer to the device.

CAM_DIR_NONE — This indicates that there is no data to be transferred for this command.

CAM_DEV_QFRZDIS	This flag disables device queue freezing as an error recovery mechanism.
CAM_PASS_ERR_RECOVER	This flag tells the pass(4) driver to enable error recovery. The default is to not perform error recovery, which means that the retry count will not be honored without this flag, among other things.
CAM_DATA_PHYS	This indicates that the address contained in *data_ptr* is a physical address, not a virtual address.

The *retry_count* tells the kernel how many times to retry the command in question. The retry count is ignored unless the pass(4) driver is told to enable error recovery via the CAM_PASS_ERR_RECOVER flag.

The *timeout* tells the kernel how long to wait for the given command to complete. If the timeout expires and the command has not completed, the CCB will be returned from the kernel with an appropriate error status.

cmd_spec is a CDB format specifier used to build up the SCSI CDB. This text string is made up of a list of field specifiers. Field specifiers specify the value for each CDB field (including indicating that the value be taken from the next argument in the variable argument list), the width of the field in bits or bytes, and an optional name. White space is ignored, and the pound sign ('#') introduces a comment that ends at the end of the current line.

The optional name is the first part of a field specifier and is in curly braces. The text in curly braces in this example are the names:

```
{PS} v:b1 {Reserved} 0:b1 {Page Code} v:b6 # Mode select page
```

This field specifier has two one bit fields and one six bit field. The second one bit field is the constant value 0 and the first one bit field and the six bit field are taken from the variable argument list. Multi byte fields are swapped into the SCSI byte order in the CDB and white space is ignored.

When the field is a hex value or the letter v, (e.g., *1A* or *v*) then a single byte value is copied to the next unused byte of the CDB. When the letter *v* is used the next integer argument is taken from the variable argument list and that value used.

A constant hex value followed by a field width specifier or the letter *v* followed by a field width specifier (e.g., *3:4*, *3:b4*, *3:i3*, *v:i3*) specifies a field of a given bit or byte width. Either the constant value or (for the V specifier) the next integer value from the variable argument list is copied to the next unused bits or bytes of the CDB.

A decimal number or the letter *b* followed by a decimal number field width indicates a bit field of that width. The bit fields are packed as tightly as possible beginning with the high bit (so that it reads the same as the SCSI spec), and a new byte of the CDB is started whenever a byte fills completely or when an *i* field is encountered.

A field width specifier consisting of the letter *i* followed by either 1, 2, 3 or 4 indicates a 1, 2, 3 or 4 byte integral value that must be swapped into SCSI byte order (MSB first).

For the *v* field specifier the next integer argument is taken from the variable argument list and that value is used swapped into SCSI byte order.

csio_build_visit() operates similarly to **csio_build**(), except that the values to substitute for variable arguments in *cmd_spec* are retrieved via the **arg_get**() function passed in to **csio_build_visit**() instead of via stdarg(3). The **arg_get**() function takes two arguments:

gethook is passed into the **arg_get**() function at each invocation. This enables the **arg_get**() function to keep some state in between calls without using global or static variables.

field_name is the field name supplied in *fmt*, if any.

csio_decode() is used to decode information from the data in phase of the SCSI transfer.

The decoding is similar to the command specifier processing of **csio_build**() except that the data is extracted from the data pointed to by *csio->data_ptr*. The stdarg list should be pointers to integers instead of integer values. A seek field type and a suppression modifier are added. The * suppression modifier (e.g., **i3* or **b4*) suppresses assignment from the field and can be used to skip over bytes or bits in the data, without having to copy them to a dummy variable in the arg list.

The seek field type *s* permits you to skip over data. This seeks to an absolute position (*s3*) or a relative position (*s+3*) in the data, based on whether or not the presence of the '+' sign. The seek value can be specified as *v* and the next integer value from the argument list will be used as the seek value.

csio_decode_visit() operates like **csio_decode**() except that instead of placing the decoded contents of the buffer in variadic arguments, the decoded buffer contents are returned to the user via the **arg_put**() function that is passed in. The **arg_put**() function takes several arguments:

hook The "hook" is a mechanism to allow the **arg_put**() function to save state in between calls.

letter is the letter describing the format of the argument being passed into the function.

val is a void pointer to the value being passed into the function.

count
is the size of the value being passed into the **arg_put**() function. The argument format determines the unit of measure.

name
This is a text description of the field, if one was provided in the *fmt*.

buff_decode() decodes an arbitrary data buffer using the method described above for **csio_decode**().

buff_decode_visit() decodes an arbitrary data buffer using the method described above for **csio_decode_visit**().

csio_encode() encodes the *data_ptr* portion (not the CDB!) of a *ccb_scsiio* structure, using the method described above for **csio_build**().

csio_encode_visit() encodes the *data_ptr* portion (not the CDB!) of a *ccb_scsiio* structure, using the method described above for **csio_build_visit**().

buff_encode_visit() encodes an arbitrary data pointer, using the method described above for **csio_build_visit**().

RETURN VALUES

csio_build(), **csio_build_visit**(), **csio_encode**(), **csio_encode_visit**(), and **buff_encode_visit**() return the number of fields processed.

csio_decode(), **csio_decode_visit**(), **buff_decode**(), and **buff_decode_visit**() return the number of assignments performed.

SEE ALSO

cam(3), pass(4), camcontrol(8)

HISTORY

The CAM versions of these functions are based upon similar functions implemented for the old FreeBSD SCSI layer. The encoding/decoding functions in the old SCSI code were written by Peter Dufault <*dufault@hda.com*>.

Many systems have comparable interfaces to permit a user to construct a SCSI command in user space.

The old *scsireq* data structure was almost identical to the SGI /dev/scsi data structure. If anyone knows the name of the authors it should go here; Peter Dufault first read about it in a 1989 Sun Expert magazine.

The new CCB data structures are derived from the CAM-2 and CAM-3 specifications.

Peter Dufault implemented a clone of SGI's interface in 386BSD that led to the original FreeBSD SCSI library and the related kernel ioctl. If anyone needs that for compatibility, contact *dufault@hda.com*.

AUTHORS

Kenneth Merry <*ken@FreeBSD.org*> implemented the CAM versions of these encoding and decoding functions. This current work is based upon earlier work by Peter Dufault <*dufault@hda.com*>.

BUGS

There should probably be a function that encodes both the CDB and the data buffer portions of a SCSI CCB. I discovered this while implementing the arbitrary command execution code in camcontrol(8), but I have not yet had time to implement such a function.

Some of the CCB flag descriptions really do not belong here. Rather they belong in a generic CCB man page. Since that man page has not yet been written, the shorter descriptions here will have to suffice.

NAME

bcmp, **bcopy**, **bzero**, **memccpy**, **memchr**, **memcmp**, **memcpy**, **memmove**, **memset** - byte string operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

int
bcmp(*const void *b1*, *const void *b2*, *size_t len*);

void
bcopy(*const void *src*, *void *dst*, *size_t len*);

void
bzero(*void *b*, *size_t len*);

*void **
memchr(*const void *b*, *int c*, *size_t len*);

int
memcmp(*const void *b1*, *const void *b2*, *size_t len*);

*void **
memccpy(*void *dst*, *const void *src*, *int c*, *size_t len*);

*void **
memcpy(*void *dst*, *const void *src*, *size_t len*);

*void **
memmove(*void *dst*, *const void *src*, *size_t len*);

*void **
memset(*void *b*, *int c*, *size_t len*);

DESCRIPTION

These functions operate on variable length strings of bytes. They do not check for terminating null bytes as the routines listed in string(3) do.

See the specific manual pages for more information.

SEE ALSO

bcmp(3), bcopy(3), bzero(3), memccpy(3), memchr(3), memcmp(3), memcpy(3), memmove(3), memset(3)

STANDARDS

The functions **memchr**(), **memcmp**(), **memcpy**(), **memmove**(), and **memset**() conform to ISO/IEC 9899:1990 ("ISO C90").

HISTORY

The functions **bzero**() and **memccpy**() appeared in 4.3BSD; the functions **bcmp**(), **bcopy**(), appeared in 4.2BSD.

NAME

wcrtomb, **c16rtomb**, **c32rtomb** - convert a wide-character code to a character (restartable)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

size_t
wcrtomb(*char * restrict s*, *wchar_t c*, *mbstate_t * restrict ps*);

#include <uchar.h>

size_t
c16rtomb(*char * restrict s*, *char16_t c*, *mbstate_t * restrict ps*);

size_t
c32rtomb(*char * restrict s*, *char32_t c*, *mbstate_t * restrict ps*);

DESCRIPTION

The **wcrtomb**(), **c16rtomb**() and **c32rtomb**() functions store a multibyte sequence representing the wide character *c*, including any necessary shift sequences, to the character array *s*, storing a maximum of MB_CUR_MAX bytes.

If *s* is NULL, these functions behave as if *s* pointed to an internal buffer and *c* was a null wide character (L'\0').

The *mbstate_t* argument, *ps*, is used to keep track of the shift state. If it is NULL, these functions use an internal, static *mbstate_t* object, which is initialized to the initial conversion state at program startup.

As certain multibyte characters may only be represented by a series of 16-bit characters, the **c16rtomb**() may need to invoked multiple times before a multibyte sequence is returned.

RETURN VALUES

These functions return the length (in bytes) of the multibyte sequence needed to represent *c*, or (*size_t*)-1 if *c* is not a valid wide character code.

ERRORS

The **wcrtomb**(), **c16rtomb**() and **c32rtomb**() functions will fail if:

[EILSEQ] An invalid wide character code was specified.

[EINVAL] The conversion state is invalid.

SEE ALSO

mbrtowc(3), multibyte(3), setlocale(3), wctomb(3)

STANDARDS

The **wcrtomb**(), **c16rtomb**() and **c32rtomb**() functions conform to ISO/IEC 9899:2011 ("ISO C11").

NAME

malloc, **free**, **realloc**, **calloc**, **alloca**, **mmap** - general memory allocation operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*void **
malloc(*size_t size*);

void
free(*void *ptr*);

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

*void **
calloc(*size_t nelem*, *size_t elsize*);

*void **
alloca(*size_t size*);

#include <sys/types.h>
#include <sys/mman.h>

*void **
mmap(*void * addr*, *size_t len*, *int prot*, *int flags*, *int fd*, *off_t offset*);

DESCRIPTION

These functions allocate and free memory for the calling process. They are described in the individual manual pages.

SEE ALSO

mmap(2), alloca(3), calloc(3), free(3), malloc(3), realloc(3)

STANDARDS

These functions, with the exception of **alloca**() and **mmap**() conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

cam_open_device, **cam_open_spec_device**, **cam_open_btl**, **cam_open_pass**, **cam_close_device**, **cam_close_spec_device**, **cam_getccb**, **cam_send_ccb**, **cam_freeccb**, **cam_path_string**, **cam_device_dup**, **cam_device_copy**, **cam_get_device** - CAM user library

LIBRARY

Common Access Method User Library (libcam, -lcam)

SYNOPSIS

#include <stdio.h>
#include <camlib.h>

*struct cam_device **
cam_open_device(*const char *path*, *int flags*);

*struct cam_device **
cam_open_spec_device(*const char *dev_name*, *int unit*, *int flags*, *struct cam_device *device*);

*struct cam_device **
cam_open_btl(*path_id_t path_id*, *target_id_t target_id*, *lun_id_t target_lun*, *int flags*, *struct cam_device *device*);

*struct cam_device **
cam_open_pass(*const char *path*, *int flags*, *struct cam_device *device*);

void
cam_close_device(*struct cam_device *dev*);

void
cam_close_spec_device(*struct cam_device *dev*);

*union ccb **
cam_getccb(*struct cam_device *dev*);

int
cam_send_ccb(*struct cam_device *device*, *union ccb *ccb*);

void
cam_freeccb(*union ccb *ccb*);

*char **
cam_path_string(*struct cam_device *dev*, *char *str*, *int len*);

*struct cam_device **
cam_device_dup(*struct cam_device *device*);

void
cam_device_copy(*struct cam_device *src*, *struct cam_device *dst*);

int
cam_get_device(*const char *path*, *char *dev_name*, *int devnamelen*, *int *unit*);

DESCRIPTION

The CAM library consists of a number of functions designed to aid in programming with the CAM subsystem. This man page covers the basic set of library functions. More functions are documented in the man pages listed below.

Many of the CAM library functions use the *cam_device* structure:

```
struct cam_device {
        char                    device_path[MAXPATHLEN+1];/*
                                                          * Pathname of the
                                                          * device given by the
                                                          * user. This may be
                                                          * null if the user
                                                          * states the device
                                                          * name and unit number
                                                          * separately.
                                                          */
        char                    given_dev_name[DEV_IDLEN+1];/*
                                                            * Device name given by
                                                            * the user.
                                                            */
        uint32_t  given_unit_number;            /*
                                                            * Unit number given by
                                                            * the user.
                                                            */
        char                    device_name[DEV_IDLEN+1];/*
                                                         * Name of the device,
                                                         * e.g., 'pass'
```

```
                                                        */
        uint32_t  dev_unit_num;       /* Unit number of the passthrough
                                       * device associated with this
                                       * particular device.
                                       */

        char                  sim_name[SIM_IDLEN+1];/*
                                       * Controller name, e.g., 'ahc'
                                       */
        uint32_t  sim_unit_number; /* Controller unit number */
        uint32_t  bus_id;             /* Controller bus number */
        lun_id_t  target_lun;         /* Logical Unit Number */
        target_id_t           target_id;  /* Target ID */
        path_id_t path_id;    /* System SCSI bus number */
        uint16_t  pd_type;    /* type of peripheral device */
        struct scsi_inquiry_data inq_data;  /* SCSI Inquiry data */
        uint8_t               serial_num[252]; /* device serial number */
        uint8_t               serial_num_len;  /* length of the serial number */
        uint8_t               sync_period;        /* Negotiated sync period */
        uint8_t               sync_offset;        /* Negotiated sync offset */
        uint8_t               bus_width;          /* Negotiated bus width */
        int                   fd;                 /* file descriptor for device */
};
```

cam_open_device() takes as arguments a string describing the device it is to open, and *flags* suitable for passing to open(2). The "path" passed in may actually be most any type of string that contains a device name and unit number to be opened. The string will be parsed by **cam_get_device**() into a device name and unit number. Once the device name and unit number are determined, a lookup is performed to determine the passthrough device that corresponds to the given device.

cam_open_spec_device() opens the pass(4) device that corresponds to the device name and unit number passed in. The *flags* should be flags suitable for passing to open(2). The *device* argument is optional. The user may supply pre-allocated space for the *cam_device* structure. If the *device* argument is NULL, **cam_open_spec_device**() will allocate space for the *cam_device* structure using malloc(3).

cam_open_btl() is similar to **cam_open_spec_device**(), except that it takes a SCSI bus, target and logical unit instead of a device name and unit number as arguments. The *path_id* argument is the CAM equivalent of a SCSI bus number. It represents the logical bus number in the system. The *flags* should be flags suitable for passing to open(2). As with **cam_open_spec_device**(), the *device* argument is optional.

cam_open_pass() takes as an argument the *path* of a pass(4) device to open. No translation or lookup is performed, so the path passed in must be that of a CAM pass(4) device. The *flags* should be flags suitable for passing to open(2). The *device* argument, as with **cam_open_spec_device**() and **cam_open_btl**(), should be NULL if the user wants the CAM library to allocate space for the *cam_device* structure. **cam_close_device**() frees the *cam_device* structure allocated by one of the above open() calls, and closes the file descriptor to the passthrough device. This routine should not be called if the user allocated space for the *cam_device* structure. Instead, the user should call **cam_close_spec_device**().

cam_close_spec_device() merely closes the file descriptor opened in one of the open() routines described above. This function should be called when the *cam_device* structure was allocated by the caller, rather than the CAM library.

cam_getccb() allocates a CCB using malloc(3) and sets fields in the CCB header using values from the *cam_device* structure.

cam_send_ccb() sends the given *ccb* to the *device* described in the *cam_device* structure.

cam_freeccb() frees CCBs allocated by **cam_getccb**(). If *ccb* is NULL, no action is taken.

cam_path_string() takes as arguments a *cam_device* structure, and a string with length *len*. It creates a colon-terminated printing prefix string similar to the ones used by the kernel. e.g.: "(cd0:ahc1:0:4:0): ". **cam_path_string**() will place at most *len*-1 characters into *str*. The *len*'th character will be the terminating '\0'.

cam_device_dup() operates in a fashion similar to strdup(3). It allocates space for a *cam_device* structure and copies the contents of the passed-in *device* structure to the newly allocated structure.

cam_device_copy() copies the *src* structure to *dst*.

cam_get_device() takes a *path* argument containing a string with a device name followed by a unit number. It then breaks the string down into a device name and unit number, and passes them back in *dev_name* and *unit*, respectively. **cam_get_device**() can handle strings of the following forms, at least:

/dev/foo1
foo0
nsa2

cam_get_device() is provided as a convenience function for applications that need to provide functionality similar to **cam_open_device**().

RETURN VALUES

cam_open_device(), **cam_open_spec_device**(), **cam_open_btl**(), and **cam_open_pass**() return a pointer to a *cam_device* structure, or NULL if there was an error.

cam_getccb() returns an allocated and partially initialized CCB, or NULL if allocation of the CCB failed.

cam_send_ccb() returns a value of -1 if an error occurred, and *errno* is set to indicate the error.

cam_path_string() returns a filled printing prefix string as a convenience. This is the same *str* that is passed into **cam_path_string**().

cam_device_dup() returns a copy of the *device* passed in, or NULL if an error occurred.

cam_get_device() returns 0 for success, and -1 to indicate failure.

If an error is returned from one of the base CAM library functions described here, the reason for the error is generally printed in the global string *cam_errbuf* which is CAM_ERRBUF_SIZE characters long.

SEE ALSO

cam_cdbparse(3), pass(4), camcontrol(8)

HISTORY

The CAM library first appeared in FreeBSD 3.0.

AUTHORS

Kenneth Merry <*ken@FreeBSD.org*>

BUGS

cam_open_device() does not check to see if the *path* passed in is a symlink to something. It also does not check to see if the *path* passed in is an actual pass(4) device. The former would be rather easy to implement, but the latter would require a definitive way to identify a device node as a pass(4) device.

Some of the functions are possibly misnamed or poorly named.

NAME

cap_init, **cap_wrap**, **cap_unwrap**, **cap_sock**, **cap_clone**, **cap_close**, **cap_limit_get**, **cap_limit_set**, **cap_send_nvlist**, **cap_recv_nvlist**, **cap_xfer_nvlist**, **cap_service_open** - library for handling application capabilities

LIBRARY

Casper Library (libcasper, -lcasper)

SYNOPSIS

#define WITH_CASPER

#include <sys/nv.h>
#include <libcasper.h>

*cap_channel_t **
cap_init(*void*);

*cap_channel_t **
cap_wrap(*int sock*, *int flags*);

int
cap_unwrap(*cap_channel_t *chan*, *int *flags*);

int
cap_sock(*const cap_channel_t *chan*);

*cap_channel_t **
cap_clone(*const cap_channel_t *chan*);

void
cap_close(*cap_channel_t *chan*);

int
cap_limit_get(*const cap_channel_t *chan*, *nvlist_t **limitsp*);

int
cap_limit_set(*const cap_channel_t *chan*, *nvlist_t *limits*);

int
cap_send_nvlist(*const cap_channel_t *chan*, *const nvlist_t *nvl*);

*nvlist_t **
cap_recv_nvlist(*const cap_channel_t *chan*);

*nvlist_t **
cap_xfer_nvlist(*const cap_channel_t *chan*, *nvlist_t *nvl*);

*cap_channel_t **
cap_service_open(*const cap_channel_t *chan*, *const char *name*);

DESCRIPTION

The **libcasper** library allows to manage application capabilities through the casper process.

The application capability (represented by the *cap_channel_t* type) is a communication channel between the caller and the casper process daemon or an instance of one of its services. A capability to the casper process obtained with the **cap_init**() function allows to create capabilities to casper's services via the **cap_service_open**() function.

The **cap_init**() function opens capability to the casper process.

The **cap_wrap**() function creates *cap_channel_t* based on the given socket. The function is used when capability is inherited through execve(2) or send over unix(4) domain socket as a regular file descriptor and has to be represented as *cap_channel_t* again. The *flags* argument defines the channel behavior. The supported flags are:

CASPER_NO_UNIQ
The communication between process and casper uses no unique version of nvlist.

The **cap_unwrap**() function is the opposite of the **cap_wrap**() function. It frees the *cap_channel_t* structure and returns unix(4) domain socket associated with it.

The **cap_clone**() function clones the given capability.

The **cap_close**() function closes the given capability.

The **cap_sock**() function returns unix(4) domain socket descriptor associated with the given capability for use with system calls like kevent(2), poll(2) and select(2).

The **cap_limit_get**() function stores current limits of the given capability in the *limitsp* argument. If the function return *0* and NULL is stored in *limitsp* it means there are no limits set.

The **cap_limit_set**() function sets limits for the given capability. The limits are provided as nvlist(9). The exact format depends on the service the capability represents.

The **cap_send_nvlist**() function sends the given nvlist(9) over the given capability. This is low level interface to communicate with casper services. Most services should provide higher level API.

The **cap_recv_nvlist**() function receives the given nvlist(9) over the given capability.

The **cap_xfer_nvlist**() function sends the given nvlist(9), destroys it and receives new nvlist(9) in response over the given capability. It does not matter if the function succeeds or fails, the nvlist(9) given for sending will always be destroyed once the function returns.

The **cap_service_open**() function opens casper service of the given name through casper capability obtained via the **cap_init**() function. The function returns capability that provides access to opened service. Casper supports the following services in the base system:

system.dns	provides DNS libc compatible API
system.grp	provides getgrent(3) compatible API
system.pwd	provides getpwent(3) compatible API
system.random	allows to obtain entropy from */dev/random*
system.sysctl	provides sysctlbyname(3) compatible API
system.syslog	provides syslog(3) compatible API

RETURN VALUES

The **cap_clone**(), **cap_init**(), **cap_recv_nvlist**(), **cap_service_open**(), **cap_wrap**() and **cap_xfer_nvlist**() functions return NULL and set the *errno* variable on failure.

The **cap_limit_get**(), **cap_limit_set**() and **cap_send_nvlist**() functions return -1 and set the *errno* variable on failure.

The **cap_close**(), **cap_sock**() and **cap_unwrap**() functions always succeed.

SEE ALSO

errno(2), execve(2), kevent(2), poll(2), select(2), cap_dns(3), cap_grp(3), cap_pwd(3), cap_random(3), cap_sysctl(3), cap_syslog(3), libcasper_service(3), capsicum(4), unix(4), nv(9)

AUTHORS

The **libcasper** library was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation. The **libcasper** new architecture was implemented by Mariusz Zaborski <*oshogbo@FreeBSD.org*>

NAME

cap_syslog cap_vsyslog cap_openlog cap_closelog cap_setlogmask - library for syslog in capability mode

LIBRARY

library "libcap_syslog"

SYNOPSIS

#include <libcasper.h>
#include <casper/cap_syslog.h>

void
cap_syslog(*cap_channel_t *chan*, *int pri*, *const char *fmt*, ...);

void
cap_vsyslog(*cap_channel_t *chan*, *int priority*, *const char *fmt*, *va_list ap*);

void
cap_openlog(*cap_channel_t *chan*, *const char *ident*, *int logopt*, *int facility*);

void
cap_closelog(*cap_channel_t *chan*);

int
cap_setlogmask(*cap_channel_t *chan*, *int maskpri*);

DESCRIPTION

The functions **cap_syslog**() **cap_vsyslog**() **cap_openlog**() **cap_closelog**() **cap_setlogmask**() are respectively equivalent to syslog(3), vsyslog(3), openlog(3), closelog(3), setlogmask(3) except that the connection to the **system.syslog** service needs to be provided.

EXAMPLES

The following example first opens a capability to casper and then uses this capability to create the **system.syslog** casper service to log messages.

```
cap_channel_t *capcas, *capsyslog;

/* Open capability to Casper. */
capcas = cap_init();
if (capcas == NULL)
```

```
        err(1, "Unable to contact Casper");

/* Enter capability mode sandbox. */
if (cap_enter() < 0 && errno != ENOSYS)
        err(1, "Unable to enter capability mode");

/* Use Casper capability to create capability to the system.syslog service. */
capsyslog = cap_service_open(capcas, "system.syslog");
if (capsyslog == NULL)
        err(1, "Unable to open system.syslog service");

/* Close Casper capability, we don't need it anymore. */
cap_close(capcas);

/* Let's log something. */
cap_syslog(capsyslog, LOG_NOTICE, "System logs from capability mode.");
```

SEE ALSO

cap_enter(2), closelog(3), err(3), openlog(3), setlogmask(3) syslog(3), vsyslog(3), capsicum(4), nv(9)

AUTHORS

Mariusz Zaborski <*oshogbo@FreeBSD.org*>

NAME

cap_gethostbyname, **cap_gethostbyname2**, **cap_gethostbyaddr**, **cap_getnameinfo**, **cap_dns_type_limit**, **cap_dns_family_limit** - library for getting network host entry in capability mode

LIBRARY

library "libcap_dns"

SYNOPSIS

#include <sys/nv.h>
#include <libcasper.h>
#include <casper/cap_dns.h>

*struct hostent **
cap_gethostbyname(*const cap_channel_t *chan*, *const char *name*);

*struct hostent **
cap_gethostbyname2(*const cap_channel_t *chan*, *const char *name*, *int af*);

*struct hostent **
cap_gethostbyaddr(*const cap_channel_t *chan*, *const void *addr*, *socklen_t len*, *int af*);

int
cap_getnameinfo(*const cap_channel_t *chan*, *const void *name*, *int namelen*);

int
cap_dns_type_limit(*cap_channel_t *chan*, *const char * const *types*, *size_t ntypes*);

int
cap_dns_family_limit(*const cap_channel_t *chan*, *const int *families*, *size_t nfamilies*);

DESCRIPTION

The functions **cap_gethostbyname**(), **cap_gethostbyname2**(), **cep_gethostbyaddr**() and **cap_getnameinfo**() are respectively equivalent to gethostbyname(2), gethostbyname2(2), gethostbyaddr(2) and getnameinfo(2) except that the connection to the **system.dns** service needs to be provided.

The **cap_dns_type_limit**() function limits the functions allowed in the service. The *types* variable can be set to ADDR or NAME. See the *LIMITS* section for more details. The *ntpyes* variable contains the number of *types* provided.

The **cap_dns_family_limit**() functions allows to limit address families. For details see *LIMITS*. The *nfamilies* variable contains the number of *families* provided.

LIMITS

The preferred way of setting limits is to use the **cap_dns_type_limit**() and **cap_dns_family_limit**() functions, but the limits of service can be set also using cap_limit_set(3). The nvlist(9) for that function can contain the following values and types:

type (NV_TYPE_STRING)
The *type* can have two values: ADDR or NAME. The ADDR means that functions **cap_gethostbyname**(), **cap_gethostbyname2**() and **cap_gethostbyaddr**() are allowed. In case when *type* is set to NAME the **cap_getnameinfo**() function is allowed.

family (NV_TYPE_NUMBER)
The *family* limits service to one of the address families (e.g. AF_INET, AF_INET6, etc.).

EXAMPLES

The following example first opens a capability to casper and then uses this capability to create the **system.dns** casper service and uses it to resolve an IP address.

```
cap_channel_t *capcas, *capdns;
const char *typelimit = "ADDR";
int familylimit;
const char *ipstr = "127.0.0.1";
struct in_addr ip;
struct hostent *hp;

/* Open capability to Casper. */
capcas = cap_init();
if (capcas == NULL)
        err(1, "Unable to contact Casper");

/* Enter capability mode sandbox. */
if (cap_enter() < 0 && errno != ENOSYS)
        err(1, "Unable to enter capability mode");

/* Use Casper capability to create capability to the system.dns service. */
capdns = cap_service_open(capcas, "system.dns");
if (capdns == NULL)
        err(1, "Unable to open system.dns service");
```

```
/* Close Casper capability, we don't need it anymore. */
cap_close(capcas);

/* Limit system.dns to reverse DNS lookups. */
if (cap_dns_type_limit(capdns, &typelimit, 1) < 0)
        err(1, "Unable to limit access to the system.dns service");

/* Limit system.dns to reserve IPv4 addresses */
familylimit = AF_INET;
if (cap_dns_family_limit(capdns, &familylimit, 1) < 0)
        err(1, "Unable to limit access to the system.dns service");

/* Convert IP address in C-string to in_addr. */
if (!inet_aton(ipstr, &ip))
        errx(1, "Unable to parse IP address %s.", ipstr);

/* Find hostname for the given IP address. */
hp = cap_gethostbyaddr(capdns, (const void *)&ip, sizeof(ip), AF_INET);
if (hp == NULL)
        errx(1, "No name associated with %s.", ipstr);

printf("Name associated with %s is %s.\n", ipstr, hp->h_name);
```

SEE ALSO

cap_enter(2), err(3), gethostbyaddr(3), gethostbyname(3), gethostbyname2(3), getnameinfo(3), capsicum(4), nv(9)

AUTHORS

The **cap_dns** service was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

This manual page was written by
Mariusz Zaborski <*oshogbo@FreeBSD.org*>.

NAME

cap_getgrent, **cap_getgrnam**, **cap_getgrgid**, **cap_getgrent_r**, **cap_getgrnam_r**, **cap_getgrgid_r**, **cap_setgroupent**, **cap_setgrent**, **cap_endgrent**, **cap_grp_limit_cmds**, **cap_grp_limit_fields**, **cap_grp_limit_groups** - library for group database operations in capability mode

LIBRARY

library "libcap_grp"

SYNOPSIS

#include <sys/nv.h>
#include <libcasper.h>
#include <casper/cap_grp.h>

*struct group **
cap_getgrent(*cap_channel_t *chan*);

*struct group **
cap_getgrnam(*cap_channel_t *chan*, *const char *name*);

*struct group **
cap_getgrgid(*cap_channel_t *chan*, *gid_t gid*);

int
cap_getgrent_r(*cap_channel_t *chan*, *struct group *grp*, *char *buffer*, *size_t bufsize*, *struct group **result*);

int
cap_getgrnam_r(*cap_channel_t *chan*, *const char *name*, *struct group *grp*, *char *buffer*, *size_t bufsize*, *struct group **result*);

int
cap_getgrgid_r(*cap_channel_t *chan*, *gid_t gid*, *struct group *grp*, *char *buffer*, *size_t bufsize*, *struct group **result*);

int
cap_setgroupent(*cap_channel_t *chan*, *int stayopen*);

int
cap_setgrent(*cap_channel_t *chan*);

void
cap_endgrent(*cap_channel_t *chan*);

int
cap_grp_limit_cmds(*cap_channel_t *chan*, *const char * const *cmds*, *size_t ncmds*);

int
cap_grp_limit_fields(*cap_channel_t *chan*, *const char * const *fields*, *size_t nfields*);

int
cap_grp_limit_groups(*cap_channel_t *chan*, *const char * const *names*, *size_t nnames*,
*const gid_t *gids*, *size_t ngids*);

DESCRIPTION

The functions **cap_getgrent**(), **cap_getgrnam**(), **cap_getgrgid**(), **cap_getgrent_r**(), **cap_getgrnam_r**(), **cap_getgrgid_r**(), **cap_setgroupent**(), **cap_setgrent**(), and **cap_endgrent**() are respectively equivalent to getgrent(3), getgrnam(3), getgrgid(3), getgrent_r(3), getgrnam_r(3), getgrgid_r(3), setgroupent(3), setgrent(3), and endgrent(3) except that the connection to the **system.grp** service needs to be provided.

The **cap_grp_limit_cmds**() function limits the functions allowed in the service. The *cmds* variable can be set to getgrent, getgrnam, getgrgid, getgrent_r, getgrnam_r, getgrgid_r, setgroupent, setgrent, or endgrent which will allow to use the function associated with the name. The *ncmds* variable contains the number of *cmds* provided.

The **cap_grp_limit_fields**() function allows limit fields returned in the structure *group*. The *fields* variable can be set to gr_name gr_passwd gr_gid or gr_mem. The field which was set as the limit will be returned, while the rest of the values not set this way will have default values. The *nfields* variable contains the number of *fields* provided.

The **cap_grp_limit_groups**() function allows to limit access to groups. The *names* variable allows to limit groups by name and the *gids* variable by the group number. The *nnames* and *ngids* variables provide numbers of limited names and gids.

EXAMPLES

The following example first opens a capability to casper and then uses this capability to create the **system.grp** casper service and uses it to get a group name.

```
cap_channel_t *capcas, *capgrp;
const char *cmds[] = { "getgrgid" };
const char *fields[] = { "gr_name" };
```

```
const gid_t gid[] = { 1 };
struct group *group;

/* Open capability to Casper. */
capcas = cap_init();
if (capcas == NULL)
        err(1, "Unable to contact Casper");

/* Enter capability mode sandbox. */
if (cap_enter() < 0 && errno != ENOSYS)
        err(1, "Unable to enter capability mode");

/* Use Casper capability to create capability to the system.grp service. */
capgrp = cap_service_open(capcas, "system.grp");
if (capgrp == NULL)
        err(1, "Unable to open system.grp service");

/* Close Casper capability, we don't need it anymore. */
cap_close(capcas);

/* Limit service to one single function. */
if (cap_grp_limit_cmds(capgrp, cmds, nitems(cmds)))
                err(1, "Unable to limit access to system.grp service");

/* Limit service to one field as we only need name of the group. */
if (cap_grp_limit_fields(capgrp, fields, nitems(fields)))
                err(1, "Unable to limit access to system.grp service");

/* Limit service to one gid. */
if (cap_grp_limit_groups(capgrp, NULL, 0, gid, nitems(gid)))
                err(1, "Unable to limit access to system.grp service");

group = cap_getgrgid(capgrp, gid[0]);
if (group == NULL)
                err(1, "Unable to get name of group");

printf("GID %d is associated with name %s.\n", gid[0], group->gr_name);

cap_close(capgrp);
```

SEE ALSO

cap_enter(2), endgrent(3), err(3), getgrent(3), getgrent_r(3), getgrgid(3), getgrgid_r(3), getgrnam(3), getgrnam_r(3), setgrent(3), setgroupent(3), capsicum(4), nv(9)

AUTHORS

The **cap_grp** service was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

This manual page was written by
Mariusz Zaborski <*oshogbo@FreeBSD.org*>.

NAME

cap_getpwent, **cap_getpwnam**, **cap_getpwuid**, **cap_getpwent_r**, **cap_getpwnam_r**, **cap_getpwuid_r**, **cap_setpassent**, **cap_setpwent**, **cap_endpwent**, **cap_pwd_limit_cmds**, **cap_pwd_limit_fields**, **cap_pwd_limit_users** - library for password database operations in capability mode

LIBRARY

library "libcap_pwd"

SYNOPSIS

#include <libcasper.h>
#include <casper/cap_pwd.h>

*struct passwd **
cap_getpwent(*cap_channel_t *chan*);

*struct passwd **
cap_getpwnam(*cap_channel_t *chan*, *const char *login*);

*struct passwd **
cap_getpwuid(*cap_channel_t *chan*, *uid_t uid*);

int
cap_getpwent_r(*cap_channel_t *chan*, *struct passwd *pwd*, *char *buffer*, *size_t bufsize*, *struct passwd **result*);

int
cap_getpwnam_r(*cap_channel_t *chan*, *const char *name*, *struct passwd *pwd*, *char *buffer*, *size_t bufsize*, *struct passwd **result*);

int
cap_getpwuid_r(*cap_channel_t *chan*, *uid_t uid*, *struct passwd *pwd*, *char *buffer*, *size_t bufsize*, *struct passwd **result*);

int
cap_setpassent(*cap_channel_t *chan*, *int stayopen*);

void
cap_setpwent(*cap_channel_t *chan*);

void

cap_endpwent(*cap_channel_t *chan*);

int
cap_pwd_limit_cmds(*cap_channel_t *chan*, *const char * const *cmds*, *size_t ncmds*);

int
cap_pwd_limit_fields(*cap_channel_t *chan*, *const char * const *fields*, *size_t nfields*);

int
cap_pwd_limit_users(*cap_channel_t *chan*, *const char * const *names*, *size_t nnames*, *uid_t *uids*, *size_t nuids*);

DESCRIPTION

The functions **cap_getpwent**(), **cap_getpwnam**(), **cap_getpwuid**(), **cap_getpwent_r**(), **cap_getpwnam_r**(), **cap_getpwuid_r**(), **cap_setpassent**(), **cap_setpwent**(), and **cap_endpwent**() are respectively equivalent to getpwent(3), getpwnam(3), getpwuid(3), getpwent_r(3), getpwnam_r(3), getpwuid_r(3), setpassent(3), setpwent(3), and cap_endpwent(3) except that the connection to the **system.pwd** service needs to be provided.

The **cap_pwd_limit_cmds**() function limits the functions allowed in the service. The *cmds* variable can be set to getpwent, getpwnam, getpwuid, getpwent_r, getpwnam_r, getpwuid_r, setpassent, setpwent, or endpwent which will allow to use the function associated with the name. The *ncmds* variable contains the number of *cmds* provided.

The **cap_pwd_limit_fields**() function allows limit fields returned in the structure *passwd*. The *fields* variable can be set to pw_name, pw_passwd, pw_uid, pw_gid, pw_change, pw_class, pw_gecos, pw_dir, pw_shell, pw_expire or pw_fields The field which was set as the limit will be returned, while the rest of the values not set this way will have default values. The *nfields* variable contains the number of *fields* provided.

The **cap_pwd_limit_users**() function allows to limit access to users. The *names* variable allows to limit users by name and the *uids* variable by the user number. The *nnames* and *nuids* variables provide numbers of limited names and uids.

EXAMPLES

The following example first opens a capability to casper and then uses this capability to create the **system.pwd** casper service and uses it to get a user name.

```
cap_channel_t *capcas, *cappwd;
const char *cmds[] = { "getpwuid" };
```

```
const char *fields[] = { "pw_name" };
uid_t uid[] = { 1 };
struct passwd *passwd;

/* Open capability to Casper. */
capcas = cap_init();
if (capcas == NULL)
        err(1, "Unable to contact Casper");

/* Enter capability mode sandbox. */
if (cap_enter() < 0 && errno != ENOSYS)
        err(1, "Unable to enter capability mode");

/* Use Casper capability to create capability to the system.pwd service. */
cappwd = cap_service_open(capcas, "system.pwd");
if (cappwd == NULL)
        err(1, "Unable to open system.pwd service");

/* Close Casper capability, we don't need it anymore. */
cap_close(capcas);

/* Limit service to one single function. */
if (cap_pwd_limit_cmds(cappwd, cmds, nitems(cmds)))
                err(1, "Unable to limit access to system.pwd service");

/* Limit service to one field as we only need name of the user. */
if (cap_pwd_limit_fields(cappwd, fields, nitems(fields)))
                err(1, "Unable to limit access to system.pwd service");

/* Limit service to one uid. */
if (cap_pwd_limit_users(cappwd, NULL, 0, uid, nitems(uid)))
                err(1, "Unable to limit access to system.pwd service");

passwd = cap_getpwuid(cappwd, uid[0]);
if (passwd == NULL)
                err(1, "Unable to get name of user");

printf("UID %d is associated with name %s.\n", uid[0], passwd->pw_name);

cap_close(cappwd);
```

SEE ALSO

cap_enter(2), endpwent(3), err(3), getpwent(3), getpwent_r(3), getpwnam(3), getpwnam_r(3), getpwuid(3), getpwuid_r(3), setpassent(3), setpwent(3), capsicum(4), nv(9)

AUTHORS

The **cap_pwd** service was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

This manual page was written by
Mariusz Zaborski <*oshogbo@FreeBSD.org*>.

NAME

cap_enter, **cap_getmode** - Capability mode system calls

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/capsicum.h>

int
cap_enter(*void*);

int
cap_getmode(*u_int *modep*);

DESCRIPTION

cap_enter() places the current process into capability mode, a mode of execution in which processes may only issue system calls operating on file descriptors or reading limited global system state. Access to global name spaces, such as file system or IPC name spaces, is prevented. If the process is already in a capability mode sandbox, the system call is a no-op. Future process descendants created with fork(2) or pdfork(2) will be placed in capability mode from inception.

When combined with cap_rights_limit(2), cap_ioctls_limit(2), cap_fcntls_limit(2), **cap_enter**() may be used to create kernel-enforced sandboxes in which appropriately-crafted applications or application components may be run.

cap_getmode() returns a flag indicating whether or not the process is in a capability mode sandbox.

RUN-TIME SETTINGS

If the kern.trap_enotcap sysctl MIB is set to a non-zero value, then for any process executing in a capability mode sandbox, any syscall which results in either an ENOTCAPABLE or ECAPMODE error also generates the synchronous SIGTRAP signal to the thread on the syscall return. On signal delivery, the *si_errno* member of the *siginfo* signal handler parameter is set to the syscall error value, and the *si_code* member is set to TRAP_CAP.

See also the PROC_TRAPCAP_CTL and PROC_TRAPCAP_STATUS operations of the procctl(2) function for similar per-process functionality.

CAVEAT

Creating effective process sandboxes is a tricky process that involves identifying the least possible rights

required by the process and then passing those rights into the process in a safe manner. Consumers of **cap_enter**() should also be aware of other inherited rights, such as access to VM resources, memory contents, and other process properties that should be considered. It is advisable to use fexecve(2) to create a runtime environment inside the sandbox that has as few implicitly acquired rights as possible.

RETURN VALUES

The **cap_enter**() and **cap_getmode**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

When the process is in capability mode, **cap_getmode**() sets the flag to a non-zero value. A zero value means the process is not in capability mode.

ERRORS

The **cap_enter**() and **cap_getmode**() system calls will fail if:

[ENOSYS] The kernel is compiled without:

options CAPABILITY_MODE

The **cap_getmode**() system call may also return the following error:

[EFAULT] Pointer *modep* points outside the process's allocated address space.

SEE ALSO

cap_fcntls_limit(2), cap_ioctls_limit(2), cap_rights_limit(2), fexecve(2), procctl(2), cap_sandboxed(3), capsicum(4), sysctl(9)

HISTORY

Support for capabilities and capabilities mode was developed as part of the TrustedBSD Project.

AUTHORS

These functions and the capability facility were created by Robert N. M. Watson at the University of Cambridge Computer Laboratory with support from a grant from Google, Inc.

NAME

cap_fcntls_limit, **cap_fcntls_get** - manage allowed fcntl commands

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/capsicum.h>

int
cap_fcntls_limit(*int fd*, *uint32_t fcntlrights*);

int
cap_fcntls_get(*int fd*, *uint32_t *fcntlrightsp*);

DESCRIPTION

If a file descriptor is granted the CAP_FCNTL capability right, the list of allowed fcntl(2) commands can be selectively reduced (but never expanded) with the **cap_fcntls_limit**() system call.

A bitmask of allowed fcntls commands for a given file descriptor can be obtained with the **cap_fcntls_get**() system call.

FLAGS

The following flags may be specified in the *fcntlrights* argument or returned in the *fcntlrightsp* argument:

CAP_FCNTL_GETFL Permit F_GETFL command.

CAP_FCNTL_SETFL Permit F_SETFL command.

CAP_FCNTL_GETOWN Permit F_GETOWN command.

CAP_FCNTL_SETOWN Permit F_SETOWN command.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

cap_fcntls_limit() succeeds unless:

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

[EINVAL] An invalid flag has been passed in *fcntlrights*.

[ENOTCAPABLE] *fcntlrights* would expand the list of allowed fcntl(2) commands.

cap_fcntls_get() succeeds unless:

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

[EFAULT] The *fcntlrightsp* argument points at an invalid address.

SEE ALSO

cap_ioctls_limit(2), cap_rights_limit(2), fcntl(2)

HISTORY

Support for capabilities and capabilities mode was developed as part of the TrustedBSD Project.

AUTHORS

This function was created by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship of the FreeBSD Foundation.

NAME

cap_ioctls_limit, **cap_ioctls_get** - manage allowed ioctl commands

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/capsicum.h>

int
cap_ioctls_limit(*int fd*, *const unsigned long *cmds*, *size_t ncmds*);

ssize_t
cap_ioctls_get(*int fd*, *unsigned long *cmds*, *size_t maxcmds*);

DESCRIPTION

If a file descriptor is granted the CAP_IOCTL capability right, the list of allowed ioctl(2) commands can be selectively reduced (but never expanded) with the **cap_ioctls_limit**() system call. The *cmds* argument is an array of ioctl(2) commands and the *ncmds* argument specifies the number of elements in the array. There can be up to *256* elements in the array. Including an element that has been previously revoked will generate an error. After a successful call only those listed in the array may be used.

The list of allowed ioctl commands for a given file descriptor can be obtained with the **cap_ioctls_get**() system call. The *cmds* argument points at memory that can hold up to *maxcmds* values. The function populates the provided buffer with up to *maxcmds* elements, but always returns the total number of ioctl commands allowed for the given file descriptor. The total number of ioctls commands for the given file descriptor can be obtained by passing NULL as the *cmds* argument and *0* as the *maxcmds* argument. If all ioctl commands are allowed (CAP_IOCTL capability right is assigned to the file descriptor and the **cap_ioctls_limit**() system call was never called for this file descriptor), the **cap_ioctls_get**() system call will return CAP_IOCTLS_ALL and will not modify the buffer pointed to by the *cmds* argument.

RETURN VALUES

The **cap_ioctls_limit**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

The **cap_ioctls_get**() function, if successful, returns the total number of allowed ioctl commands or the value CAP_IOCTLS_ALL if all ioctls commands are allowed. On failure the value *-1* is returned and the global variable errno is set to indicate the error.

ERRORS

cap_ioctls_limit() succeeds unless:

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

[EFAULT] The *cmds* argument points at an invalid address.

[EINVAL] The *ncmds* argument is greater than *256*.

[ENOTCAPABLE] *cmds* would expand the list of allowed ioctl(2) commands.

cap_ioctls_get() succeeds unless:

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

[EFAULT] The *cmds* argument points at invalid address.

SEE ALSO

cap_fcntls_limit(2), cap_rights_limit(2), ioctl(2)

HISTORY

Support for capabilities and capabilities mode was developed as part of the TrustedBSD Project.

AUTHORS

This function was created by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship of the FreeBSD Foundation.

NAME

cap_random_buf - library for getting entropy in capability mode

LIBRARY

library "libcap_random"

SYNOPSIS

#include <sys/nv.h>
#include <libcasper.h>
#include <casper/cap_random.h>

int
cap_random_buf(*cap_channel_t *chan*, *void *buf*, *size_t nbytes*);

DESCRIPTION

The function **cap_random_buf**() is equivalent to arc4random_buf(3) except that the connection to the **system.random** service needs to be provided.

EXAMPLES

The following example first opens a capability to casper and then uses this capability to create the **system.random** casper service to obtain entropy.

```
cap_channel_t *capcas, *caprandom;
unsigned char buf[16];
int i;

/* Open capability to Casper. */
capcas = cap_init();
if (capcas == NULL)
        err(1, "Unable to contact Casper");

/* Enter capability mode sandbox. */
if (cap_enter() < 0 && errno != ENOSYS)
        err(1, "Unable to enter capability mode");

/* Use Casper capability to create capability to the system.random service. */
caprandom = cap_service_open(capcas, "system.random");
if (caprandom == NULL)
        err(1, "Unable to open system.random service");
```

```
/* Close Casper capability, we don't need it anymore. */
cap_close(capcas);

/* Obtain entropy. */
if (cap_random_buf(caprandom, buf, sizeof(buf)) < 0)
        err(1, "Unable to obtain entropy");

for (i = 0; i < sizeof(buf); i++)
        printf("%.2x ", buf[i]);
printf("\n");
```

SEE ALSO

cap_enter(2), arc4random_buf(3), err(3), capsicum(4), nv(9)

AUTHORS

The **cap_random** service was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

This manual page was written by
Mariusz Zaborski <*oshogbo@FreeBSD.org*>.

NAME

cap_rights_init, **cap_rights_set**, **cap_rights_clear**, **cap_rights_is_set**, **cap_rights_is_valid**, **cap_rights_merge**, **cap_rights_remove**, **cap_rights_contains** - manage cap_rights_t structure

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/capsicum.h>

*cap_rights_t **
cap_rights_init(*cap_rights_t *rights*, ...);

*cap_rights_t **
cap_rights_set(*cap_rights_t *rights*, ...);

*cap_rights_t **
cap_rights_clear(*cap_rights_t *rights*, ...);

bool
cap_rights_is_set(*const cap_rights_t *rights*, ...);

bool
cap_rights_is_valid(*const cap_rights_t *rights*);

*cap_rights_t **
cap_rights_merge(*cap_rights_t *dst*, *const cap_rights_t *src*);

*cap_rights_t **
cap_rights_remove(*cap_rights_t *dst*, *const cap_rights_t *src*);

bool
cap_rights_contains(*const cap_rights_t *big*, *const cap_rights_t *little*);

DESCRIPTION

The functions documented here allow to manage the *cap_rights_t* structure.

Capability rights should be separated with comma when passed to the **cap_rights_init**(), **cap_rights_set**(), **cap_rights_clear**() and **cap_rights_is_set**() functions. For example:

```
cap_rights_set(&rights, CAP_READ, CAP_WRITE, CAP_FSTAT, CAP_SEEK);
```

The complete list of the capability rights can be found in the rights(4) manual page.

The **cap_rights_init**() function initialize provided *cap_rights_t* structure. Only properly initialized structure can be passed to the remaining functions. For convenience the structure can be filled with capability rights instead of calling the **cap_rights_set**() function later. For even more convenience pointer to the given structure is returned, so it can be directly passed to cap_rights_limit(2):

```
cap_rights_t rights;

if (cap_rights_limit(fd, cap_rights_init(&rights, CAP_READ, CAP_WRITE)) < 0)
        err(1, "Unable to limit capability rights");
```

The **cap_rights_set**() function adds the given capability rights to the given *cap_rights_t* structure.

The **cap_rights_clear**() function removes the given capability rights from the given *cap_rights_t* structure.

The **cap_rights_is_set**() function checks if all the given capability rights are set for the given *cap_rights_t* structure.

The **cap_rights_is_valid**() function verifies if the given *cap_rights_t* structure is valid.

The **cap_rights_merge**() function merges all capability rights present in the *src* structure into the *dst* structure.

The **cap_rights_remove**() function removes all capability rights present in the *src* structure from the *dst* structure.

The **cap_rights_contains**() function checks if the *big* structure contains all capability rights present in the *little* structure.

RETURN VALUES

The functions never fail. In case an invalid capability right or an invalid *cap_rights_t* structure is given as an argument, the program will be aborted.

The **cap_rights_init**(), **cap_rights_set**() and **cap_rights_clear**() functions return pointer to the *cap_rights_t* structure given in the *rights* argument.

The **cap_rights_merge**() and **cap_rights_remove**() functions return pointer to the *cap_rights_t* structure given in the *dst* argument.

The **cap_rights_is_set**() returns *true* if all the given capability rights are set in the *rights* argument.

The **cap_rights_is_valid**() function performs various checks to see if the given *cap_rights_t* structure is valid and returns *true* if it is.

The **cap_rights_contains**() function returns *true* if all capability rights set in the *little* structure are also present in the *big* structure.

EXAMPLES

The following example demonstrates how to prepare a *cap_rights_t* structure to be passed to the cap_rights_limit(2) system call.

```
cap_rights_t rights;
int fd;

fd = open("/tmp/foo", O_RDWR);
if (fd < 0)
        err(1, "open() failed");

cap_rights_init(&rights, CAP_FSTAT, CAP_READ);

if (allow_write_and_seek)
        cap_rights_set(&rights, CAP_WRITE, CAP_SEEK);

if (dont_allow_seek)
        cap_rights_clear(&rights, CAP_SEEK);

if (cap_rights_limit(fd, &rights) < 0 && errno != ENOSYS)
        err(1, "cap_rights_limit() failed");
```

SEE ALSO

cap_rights_limit(2), open(2), capsicum(4), rights(4)

HISTORY

Support for capabilities and capabilities mode was developed as part of the TrustedBSD Project.

AUTHORS

This family of functions was created by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

NAME

cap_rights_get - obtain capability rights

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/capsicum.h>

int
cap_rights_get(*int fd*, *cap_rights_t *rights*);

DESCRIPTION

The **cap_rights_get** function allows to obtain current capability rights for the given descriptor. The function will fill the *rights* argument with all capability rights if they were not limited or capability rights configured during the last successful call of cap_rights_limit(2) on the given descriptor.

The *rights* argument can be inspected using cap_rights_init(3) family of functions.

The complete list of the capability rights can be found in the rights(4) manual page.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

EXAMPLES

The following example demonstrates how to limit file descriptor capability rights and how to obtain them.

```
cap_rights_t setrights, getrights;
int fd;

memset(&setrights, 0, sizeof(setrights));
memset(&getrights, 0, sizeof(getrights));

fd = open("/tmp/foo", O_RDONLY);
if (fd < 0)
        err(1, "open() failed");

cap_rights_init(&setrights, CAP_FSTAT, CAP_READ);
```

```
if (cap_rights_limit(fd, &setrights) < 0 && errno != ENOSYS)
        err(1, "cap_rights_limit() failed");

if (cap_rights_get(fd, &getrights) < 0 && errno != ENOSYS)
        err(1, "cap_rights_get() failed");

assert(memcmp(&setrights, &getrights, sizeof(setrights)) == 0);
```

ERRORS

cap_rights_get() succeeds unless:

[EBADF] The *fd* argument is not a valid active descriptor.

[EFAULT] The *rights* argument points at an invalid address.

SEE ALSO

cap_rights_limit(2), errno(2), open(2), assert(3), cap_rights_init(3), err(3), memcmp(3), memset(3), capsicum(4), rights(4)

HISTORY

Support for capabilities and capabilities mode was developed as part of the TrustedBSD Project.

AUTHORS

This function was created by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship of the FreeBSD Foundation.

NAME

cap_rights_limit - limit capability rights

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/capsicum.h>

int
cap_rights_limit(*int fd*, *const cap_rights_t *rights*);

DESCRIPTION

When a file descriptor is created by a function such as accept(2), accept4(2), fhopen(2), kqueue(2), mq_open(2), open(2), openat(2), pdfork(2), pipe(2), shm_open(2), socket(2) or socketpair(2), it is assigned all capability rights. Those rights can be reduced (but never expanded) by using the **cap_rights_limit**() system call. Once capability rights are reduced, operations on the file descriptor will be limited to those permitted by *rights*.

The *rights* argument should be prepared using cap_rights_init(3) family of functions.

Capability rights assigned to a file descriptor can be obtained with the cap_rights_get(3) function.

The complete list of the capability rights can be found in the rights(4) manual page.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

EXAMPLES

The following example demonstrates how to limit file descriptor capability rights to allow reading only.

```
cap_rights_t setrights;
char buf[1];
int fd;

fd = open("/tmp/foo", O_RDWR);
if (fd < 0)
        err(1, "open() failed");
```

```
if (cap_enter() < 0)
        err(1, "cap_enter() failed");

cap_rights_init(&setrights, CAP_READ);
if (cap_rights_limit(fd, &setrights) < 0)
        err(1, "cap_rights_limit() failed");

buf[0] = 'X';

if (write(fd, buf, sizeof(buf)) > 0)
        errx(1, "write() succeeded!");

if (read(fd, buf, sizeof(buf)) < 0)
        err(1, "read() failed");
```

ERRORS

cap_rights_limit() succeeds unless:

[EBADF] The *fd* argument is not a valid active descriptor.

[EINVAL] An invalid right has been requested in *rights*.

[ENOTCAPABLE] The *rights* argument contains capability rights not present for the given file descriptor. Capability rights list can only be reduced, never expanded.

SEE ALSO

accept(2), accept4(2), cap_enter(2), fhopen(2), kqueue(2), mq_open(2), open(2), openat(2), pdfork(2), pipe(2), read(2), shm_open(2), socket(2), socketpair(2), write(2), cap_rights_get(3), cap_rights_init(3), err(3), capsicum(4), rights(4)

HISTORY

Support for capabilities and capabilities mode was developed as part of the TrustedBSD Project.

AUTHORS

This function was created by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship of the FreeBSD Foundation.

NAME

cap_sandboxed - Check if in a capability mode sandbox

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/capsicum.h>
#include <stdbool.h>

bool
cap_sandboxed(*void*);

DESCRIPTION

cap_sandboxed() returns *true* if the process is in a capability mode sandbox or *false* if it is not. This function is a more handy alternative to the cap_getmode(2) system call as it always succeeds, so there is no need for error checking. If the support for capability mode is not compiled into the kernel, **cap_sandboxed**() will always return *false*.

RETURN VALUES

Function **cap_sandboxed**() is always successful and will return either *true* or *false*.

SEE ALSO

cap_enter(2), capsicum(4)

AUTHORS

This function was implemented and manual page was written by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship of the FreeBSD Foundation.

NAME

cap_sysctlbyname - library for getting or setting system information in capability mode

LIBRARY

library "libcap_sysctl"

SYNOPSIS

#include <sys/nv.h>
#include <libcasper.h>
#include <casper/cap_sysctl.h>

int
cap_sysctlbyname(*cap_channel_t *chan*, *const char *name*, *void *oldp*, *size_t *oldlenp*, *const void *newp*, *size_t newlen*);

DESCRIPTION

The function **cap_sysctlbyname**() is equivalent to sysctlbyname(3) except that the connection to the **system.sysctl** service needs to be provided.

LIMITS

The service can be limited using cap_limit_set(3) function. The nvlist(9) for that function can contain the following values and types:

(NV_TYPE_NUMBER)
The name of the element with type number will be treated as the limited sysctl. The value of the element will describe the access rights for given sysctl. There are four different rights CAP_SYSCTL_READ, CAP_SYSCTL_WRITE, CAP_SYSCTL_RDWR, and CAP_SYSCTL_RECURSIVE. The CAP_SYSCTL_READ flag allows to fetch the value of a given sysctl. The CAP_SYSCTL_WIRTE flag allows to override the value of a given sysctl. The CAP_SYSCTL_RDWR is combination of the CAP_SYSCTL_WIRTE and CAP_SYSCTL_READ and allows to read and write the value of a given sysctl. The CAP_SYSCTL_RECURSIVE allows access to all children of a given sysctl. This right must be combined with at least one other right.

EXAMPLES

The following example first opens a capability to casper and then uses this capability to create the **system.sysctl** casper service and uses it to get the value of kern.trap_enotcap.

```
cap_channel_t *capcas, *capsysctl;
const char *name = "kern.trap_enotcap";
```

```
nvlist_t *limits;
int value;
size_t size;

/* Open capability to Casper. */
capcas = cap_init();
if (capcas == NULL)
        err(1, "Unable to contact Casper");

/* Enter capability mode sandbox. */
if (cap_enter() < 0 && errno != ENOSYS)
        err(1, "Unable to enter capability mode");

/* Use Casper capability to create capability to the system.sysctl service. */
capsysctl = cap_service_open(capcas, "system.sysctl");
if (capsysctl == NULL)
        err(1, "Unable to open system.sysctl service");

/* Close Casper capability, we don't need it anymore. */
cap_close(capcas);

/* Create limit for one MIB with read access only. */
limits = nvlist_create(0);
nvlist_add_number(limits, name, CAP_SYSCTL_READ);

/* Limit system.sysctl. */
if (cap_limit_set(capsysctl, limits) < 0)
        err(1, "Unable to set limits");

/* Fetch value. */
if (cap_sysctlbyname(capsysctl, name, &value, &size, NULL, 0) < 0)
        err(1, "Unable to get value of sysctl");

printf("The value of %s is %d.\n", name, value);

cap_close(capsysctl);
```

SEE ALSO

cap_enter(2), err(3), sysctlbyname(3), capsicum(4), nv(9)

AUTHORS

The **cap_sysctl** service was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

This manual page was written by
Mariusz Zaborski <*oshogbo@FreeBSD.org*>.

NAME

caph_limit_stream, **caph_limit_stdin**, **caph_limit_stderr**, **caph_limit_stdout**, **caph_limit_stdio**, **caph_cache_tzdata**, **caph_cache_catpages** - set of the capsicum helpers, part of the libcapsicum

LIBRARY

library "libcapsicum"

SYNOPSIS

#include <capsicum_helpers.h>

int
caph_enter(*void*);

int
caph_enter_casper(*void*);

int
caph_limit_stream(*int fd*, *int flags*);

int
caph_limit_stdin(*void*);

int
caph_limit_stderr(*void*);

int
caph_limit_stdout(*void*);

int
caph_limit_stdio(*void*);

void
caph_cache_tzdata(*void*);

void
caph_cache_catpages(*void*);

DESCRIPTION

The **caph_enter** is equivalent to the cap_enter(2) it returns success when the kernel is built without support of the capability mode.

The **caph_enter_casper** is equivalent to the **caph_enter** it returns success when the system is built without Casper support.

The **capsicum helpers** are a set of a inline functions which simplify modifying programs to use Capsicum. The goal is to reduce duplicated code patterns. The **capsicum helpers** are part of **libcapsicum** but there is no need to link to the library.

caph_limit_stream() restricts capabilities on *fd* to only those needed by POSIX stream objects (that is, FILEs).

These flags can be provided:

CAPH_IGNORE_EBADF	Do not return an error if file descriptor is invalid.
CAPH_READ	Set CAP_READ on limited descriptor.
CAPH_WRITE	Set CAP_WRITE on limited descriptor.

caph_limit_stdin(), **caph_limit_stderr**() and **caph_limit_stdout**() limit standard descriptors using the **caph_limit_stream** function.

caph_limit_stdio() limits stdin, stderr and stdout.

caph_cache_tzdata() precaches all timezone data needed to use libc local time functions.

caph_cache_catpages() caches Native Language Support (NLS) data. NLS data is used for localized error printing by strerror(3) and err(3), among others.

SEE ALSO

cap_enter(2), rights(4)

NAME

catclose - close message catalog

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <nl_types.h>

int
catclose(*nl_catd catd*);

DESCRIPTION

The **catclose**() function closes the message catalog specified by the argument *catd*.

RETURN VALUES

The **catclose**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EBADF] An invalid message catalog descriptor was passed by the *catd* argument.

SEE ALSO

gencat(1), catgets(3), catopen(3)

STANDARDS

The **catclose**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

catgets - retrieve string from message catalog

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <nl_types.h>

*char **
catgets(*nl_catd catd*, *int set_id*, *int msg_id*, *const char *s*);

DESCRIPTION

The **catgets**() function attempts to retrieve message *msg_id* of set *set_id* from the message catalog referenced by the descriptor *catd*. The argument *s* points to a default message which is returned if the function is unable to retrieve the specified message.

RETURN VALUES

If the specified message was retrieved successfully, **catgets**() returns a pointer to an internal buffer containing the message string; otherwise it returns *s*.

ERRORS

[EBADF] The *catd* argument is not a valid message catalog descriptor.

[EBADMSG] The message identified by *set_id* and *msg_id* is not in the message catalog.

SEE ALSO

gencat(1), catclose(3), catopen(3)

STANDARDS

The **catgets**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

catopen - open message catalog

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <nl_types.h>

nl_catd
catopen(*const char *name*, *int oflag*);

DESCRIPTION

The **catopen**() function opens the message catalog specified by *name* and returns a message catalog descriptor. If *name* contains a '/' then *name* specifies the full pathname for the message catalog, otherwise the value of the environment variable NLSPATH is used with the following substitutions:

%N The value of the *name* argument.

%L The value of the LANG environment variable or the LC_MESSAGES category (see below).

%l The language element from the LANG environment variable or from the LC_MESSAGES category.

%t The territory element from the LANG environment variable or from the LC_MESSAGES category.

%c The codeset element from the LANG environment variable or from the LC_MESSAGES category.

%% A single % character.

An empty string is substituted for undefined values.

Path names templates defined in NLSPATH are separated by colons (':'). A leading or two adjacent colons is equivalent to specifying %N.

If the *oflag* argument is set to the NL_CAT_LOCALE constant, LC_MESSAGES locale category used to open the message catalog; using NL_CAT_LOCALE conforms to the X/Open Portability Guide Issue 4 ("XPG4") standard. You can specify 0 for compatibility with X/Open Portability Guide Issue 3

("XPG3"); when *oflag* is set to 0, the LANG environment variable determines the message catalog locale.

A message catalog descriptor remains valid in a process until that process closes it, or until a successful call to one of the exec(3) function.

RETURN VALUES

Upon successful completion, **catopen**() returns a message catalog descriptor. Otherwise, (nl_catd) -1 is returned and *errno* is set to indicate the error.

ERRORS

[EINVAL] Argument *name* does not point to a valid message catalog, or catalog is corrupt.

[ENAMETOOLONG]
An entire path to the message catalog exceeded 1024 characters.

[ENOENT] The named message catalog does not exists, or the *name* argument points to an empty string.

[ENOMEM] Insufficient memory is available.

SEE ALSO

gencat(1), catclose(3), catgets(3), setlocale(3)

STANDARDS

The **catopen**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

cfgetispeed, **cfsetispeed**, **cfgetospeed**, **cfsetospeed**, **cfsetspeed**, **cfmakeraw**, **cfmakesane**, **tcgetattr**, **tcsetattr** - manipulating the termios structure

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <termios.h>

speed_t
cfgetispeed(*const struct termios *t*);

int
cfsetispeed(*struct termios *t*, *speed_t speed*);

speed_t
cfgetospeed(*const struct termios *t*);

int
cfsetospeed(*struct termios *t*, *speed_t speed*);

int
cfsetspeed(*struct termios *t*, *speed_t speed*);

void
cfmakeraw(*struct termios *t*);

void
cfmakesane(*struct termios *t*);

int
tcgetattr(*int fd*, *struct termios *t*);

int
tcsetattr(*int fd*, *int action*, *const struct termios *t*);

DESCRIPTION

The **cfmakeraw**(), **cfmakesane**(), **tcgetattr**() and **tcsetattr**() functions are provided for getting and setting the termios structure.

The **cfgetispeed**(), **cfsetispeed**(), **cfgetospeed**(), **cfsetospeed**() and **cfsetspeed**() functions are provided for getting and setting the baud rate values in the termios structure. The effects of the functions on the terminal as described below do not become effective, nor are all errors detected, until the **tcsetattr**() function is called. Certain values for baud rates set in the termios structure and passed to **tcsetattr**() have special meanings. These are discussed in the portion of the manual page that describes the **tcsetattr**() function.

GETTING AND SETTING THE BAUD RATE

The input and output baud rates are found in the termios structure. The unsigned integer speed_t is typedef'd in the include file <*termios.h*>. The value of the integer corresponds directly to the baud rate being represented, however, the following symbolic values are defined.

```
#define B0          0
#define B50         50
#define B75         75
#define B110        110
#define B134        134
#define B150        150
#define B200        200
#define B300        300
#define B600        600
#define B1200       1200
#define    B1800    1800
#define B2400       2400
#define B4800       4800
#define B9600       9600
#define B19200      19200
#define B38400      38400
#ifndef _POSIX_SOURCE
#define EXTA        19200
#define EXTB        38400
#endif  /*_POSIX_SOURCE */
```

The **cfgetispeed**() function returns the input baud rate in the termios structure referenced by *tp*.

The **cfsetispeed**() function sets the input baud rate in the termios structure referenced by *tp* to *speed*.

The **cfgetospeed**() function returns the output baud rate in the termios structure referenced by *tp*.

The **cfsetospeed**() function sets the output baud rate in the termios structure referenced by *tp* to *speed*.

The **cfsetspeed**() function sets both the input and output baud rate in the termios structure referenced by *tp* to *speed*.

Upon successful completion, the functions **cfsetispeed**(), **cfsetospeed**(), and **cfsetspeed**() return a value of 0. Otherwise, a value of -1 is returned and the global variable *errno* is set to indicate the error.

GETTING AND SETTING THE TERMIOS STATE

This section describes the functions that are used to control the general terminal interface. Unless otherwise noted for a specific command, these functions are restricted from use by background processes. Attempts to perform these operations shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation and the SIGTTOU signal is not sent.

In all the functions, although *fd* is an open file descriptor, the functions affect the underlying terminal file, not just the open file description associated with the particular file descriptor.

The **cfmakeraw**() function sets the flags stored in the termios structure to a state disabling all input and output processing, giving a "raw I/O path", while the **cfmakesane**() function sets them to a state similar to those of a newly created terminal device. It should be noted that there is no function to reverse this effect. This is because there are a variety of processing options that could be re-enabled and the correct method is for an application to snapshot the current terminal state using the function **tcgetattr**(), setting raw or sane mode with **cfmakeraw**() or **cfmakesane**() and the subsequent **tcsetattr**(), and then using another **tcsetattr**() with the saved state to revert to the previous terminal state.

The **tcgetattr**() function copies the parameters associated with the terminal referenced by *fd* in the termios structure referenced by *tp*. This function is allowed from a background process, however, the terminal attributes may be subsequently changed by a foreground process.

The **tcsetattr**() function sets the parameters associated with the terminal from the termios structure referenced by *tp*. The *action* argument is created by *or*'ing the following values, as specified in the include file <*termios.h*>.

TCSANOW The change occurs immediately.

TCSADRAIN The change occurs after all output written to *fd* has been transmitted to the terminal. This value of *action* should be used when changing parameters that affect output.

TCSAFLUSH

The change occurs after all output written to *fd* has been transmitted to the terminal. Additionally, any input that has been received but not read is discarded.

TCSASOFT If this value is *or*'ed into the *action* value, the values of the *c_cflag*, *c_ispeed*, and *c_ospeed* fields are ignored.

The 0 baud rate is used to terminate the connection. If 0 is specified as the output speed to the function **tcsetattr**(), modem control will no longer be asserted on the terminal, disconnecting the terminal.

If zero is specified as the input speed to the function **tcsetattr**(), the input baud rate will be set to the same value as that specified by the output baud rate.

If **tcsetattr**() is unable to make any of the requested changes, it returns -1 and sets errno. Otherwise, it makes all of the requested changes it can. If the specified input and output baud rates differ and are a combination that is not supported, neither baud rate is changed.

Upon successful completion, the functions **tcgetattr**() and **tcsetattr**() return a value of 0. Otherwise, they return -1 and the global variable *errno* is set to indicate the error, as follows:

[EBADF] The *fd* argument to **tcgetattr**() or **tcsetattr**() was not a valid file descriptor.

[EINTR] The **tcsetattr**() function was interrupted by a signal.

[EINVAL] The *action* argument to the **tcsetattr**() function was not valid, or an attempt was made to change an attribute represented in the termios structure to an unsupported value.

[ENOTTY] The file associated with the *fd* argument to **tcgetattr**() or **tcsetattr**() is not a terminal.

SEE ALSO

tcsendbreak(3), termios(4)

STANDARDS

The **cfgetispeed**(), **cfsetispeed**(), **cfgetospeed**(), **cfsetospeed**(), **tcgetattr**() and **tcsetattr**() functions are expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification. The **cfmakeraw**(), **cfmakesane**() and **cfsetspeed**() functions, as well as the TCSASOFT option to the **tcsetattr**() function are extensions to the IEEE Std 1003.1-1988 ("POSIX.1") specification.

NAME

cgetent, **cgetset**, **cgetmatch**, **cgetcap**, **cgetnum**, **cgetstr**, **cgetustr**, **cgetfirst**, **cgetnext**, **cgetclose** - capability database access routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
cgetent(*char **buf*, *char **db_array*, *const char *name*);

int
cgetset(*const char *ent*);

int
cgetmatch(*const char *buf*, *const char *name*);

*char **
cgetcap(*char *buf*, *const char *cap*, *int type*);

int
cgetnum(*char *buf*, *const char *cap*, *long *num*);

int
cgetstr(*char *buf*, *const char *cap*, *char **str*);

int
cgetustr(*char *buf*, *const char *cap*, *char **str*);

int
cgetfirst(*char **buf*, *char **db_array*);

int
cgetnext(*char **buf*, *char **db_array*);

int
cgetclose(*void*);

DESCRIPTION

The **cgetent**() function extracts the capability *name* from the database specified by the NULL terminated file array *db_array* and returns a pointer to a malloc(3)'d copy of it in *buf*. The **cgetent**() function will first look for files ending in *.db* (see cap_mkdb(1)) before accessing the ASCII file. The *buf* argument must be retained through all subsequent calls to **cgetmatch**(), **cgetcap**(), **cgetnum**(), **cgetstr**(), and **cgetustr**(), but may then be free(3)'d. On success 0 is returned, 1 if the returned record contains an unresolved **tc** expansion, -1 if the requested record could not be found, -2 if a system error was encountered (could not open/read a file, etc.) also setting *errno*, and -3 if a potential reference loop is detected (see **tc=** comments below).

The **cgetset**() function enables the addition of a character buffer containing a single capability record entry to the capability database. Conceptually, the entry is added as the first ''file'' in the database, and is therefore searched first on the call to **cgetent**(). The entry is passed in *ent*. If *ent* is NULL, the current entry is removed from the database. A call to **cgetset**() must precede the database traversal. It must be called before the **cgetent**() call. If a sequential access is being performed (see below), it must be called before the first sequential access call (**cgetfirst**() or **cgetnext**()), or be directly preceded by a **cgetclose**() call. On success 0 is returned and -1 on failure.

The **cgetmatch**() function will return 0 if *name* is one of the names of the capability record *buf*, -1 if not.

The **cgetcap**() function searches the capability record *buf* for the capability *cap* with type *type*. A *type* is specified using any single character. If a colon (':') is used, an untyped capability will be searched for (see below for explanation of types). A pointer to the value of *cap* in *buf* is returned on success, NULL if the requested capability could not be found. The end of the capability value is signaled by a ':' or ASCII NUL (see below for capability database syntax).

The **cgetnum**() function retrieves the value of the numeric capability *cap* from the capability record pointed to by *buf*. The numeric value is returned in the *long* pointed to by *num*. 0 is returned on success, -1 if the requested numeric capability could not be found.

The **cgetstr**() function retrieves the value of the string capability *cap* from the capability record pointed to by *buf*. A pointer to a decoded, NUL terminated, malloc(3)'d copy of the string is returned in the *char* * pointed to by *str*. The number of characters in the decoded string not including the trailing NUL is returned on success, -1 if the requested string capability could not be found, -2 if a system error was encountered (storage allocation failure).

The **cgetustr**() function is identical to **cgetstr**() except that it does not expand special characters, but rather returns each character of the capability string literally.

The **cgetfirst**() and **cgetnext**() functions comprise a function group that provides for sequential access of

the NULL pointer terminated array of file names, *db_array*. The **cgetfirst**() function returns the first record in the database and resets the access to the first record. The **cgetnext**() function returns the next record in the database with respect to the record returned by the previous **cgetfirst**() or **cgetnext**() call. If there is no such previous call, the first record in the database is returned. Each record is returned in a malloc(3)'d copy pointed to by *buf*. **Tc** expansion is done (see **tc=** comments below). Upon completion of the database 0 is returned, 1 is returned upon successful return of record with possibly more remaining (we have not reached the end of the database yet), 2 is returned if the record contains an unresolved **tc** expansion, -1 is returned if a system error occurred, and -2 is returned if a potential reference loop is detected (see **tc=** comments below). Upon completion of database (0 return) the database is closed.

The **cgetclose**() function closes the sequential access and frees any memory and file descriptors being used. Note that it does not erase the buffer pushed by a call to **cgetset**().

CAPABILITY DATABASE SYNTAX

Capability databases are normally ASCII and may be edited with standard text editors. Blank lines and lines beginning with a '#' are comments and are ignored. Lines ending with a '\' indicate that the next line is a continuation of the current line; the '\' and following newline are ignored. Long lines are usually continued onto several physical lines by ending each line except the last with a '\'.

Capability databases consist of a series of records, one per logical line. Each record contains a variable number of ':'-separated fields (capabilities). Empty fields consisting entirely of white space characters (spaces and tabs) are ignored.

The first capability of each record specifies its names, separated by '|' characters. These names are used to reference records in the database. By convention, the last name is usually a comment and is not intended as a lookup tag. For example, the *vt100* record from the termcap(5) database begins:

```
d0|vt100|vt100-am|vt100am|dec vt100:
```

giving four names that can be used to access the record.

The remaining non-empty capabilities describe a set of (name, value) bindings, consisting of a names optionally followed by a typed value:

name	typeless [boolean] capability *name* is present [true]
name*T*value	capability (*name*, *T*) has value *value*
name@	no capability *name* exists
name*T*@	capability (*name*, *T*) does not exist

Names consist of one or more characters. Names may contain any character except ':', but it is usually best to restrict them to the printable characters and avoid use of graphics like '#', '=', '%', '@', etc. Types are single characters used to separate capability names from their associated typed values. Types may be any character except a ':'. Typically, graphics like '#', '=', '%', etc. are used. Values may be any number of characters and may contain any character except ':'.

CAPABILITY DATABASE SEMANTICS

Capability records describe a set of (name, value) bindings. Names may have multiple values bound to them. Different values for a name are distinguished by their *types*. The **cgetcap**() function will return a pointer to a value of a name given the capability name and the type of the value.

The types '#' and '=' are conventionally used to denote numeric and string typed values, but no restriction on those types is enforced. The functions **cgetnum**() and **cgetstr**() can be used to implement the traditional syntax and semantics of '#' and '='. Typeless capabilities are typically used to denote boolean objects with presence or absence indicating truth and false values respectively. This interpretation is conveniently represented by:

```
(getcap(buf, name, ':') != NULL)
```

A special capability, **tc= name**, is used to indicate that the record specified by *name* should be substituted for the **tc** capability. **Tc** capabilities may interpolate records which also contain **tc** capabilities and more than one **tc** capability may be used in a record. A **tc** expansion scope (i.e., where the argument is searched for) contains the file in which the **tc** is declared and all subsequent files in the file array.

When a database is searched for a capability record, the first matching record in the search is returned. When a record is scanned for a capability, the first matching capability is returned; the capability **:nameT@:** will hide any following definition of a value of type *T* for *name*; and the capability **:name@:** will prevent any following values of *name* from being seen.

These features combined with **tc** capabilities can be used to generate variations of other databases and records by either adding new capabilities, overriding definitions with new definitions, or hiding following definitions via '@' capabilities.

EXAMPLES

```
example|an example of binding multiple values to names:\
        :foo%bar:foo^blah:foo@:\
        :abc%xyz:abc^frap:abc$@:\
        :tc=more:
```

The capability foo has two values bound to it (bar of type '%' and blah of type '^') and any other value bindings are hidden. The capability abc also has two values bound but only a value of type '$' is prevented from being defined in the capability record more.

```
file1:
        new|new_record|a modification of "old":\
                :fript=bar:who-cares@:tc=old:blah:tc=extensions:
file2:
        old|old_record|an old database record:\
                :fript=foo:who-cares:glork#200:
```

The records are extracted by calling **cgetent**() with file1 preceding file2. In the capability record new in file1, fript=bar overrides the definition of fript=foo interpolated from the capability record old in file2, who-cares@ prevents the definition of any who-cares definitions in old from being seen, glork#200 is inherited from old, and blah and anything defined by the record extensions is added to those definitions in old. Note that the position of the fript=bar and who-cares@ definitions before tc=old is important here. If they were after, the definitions in old would take precedence.

CGETNUM AND CGETSTR SYNTAX AND SEMANTICS

Two types are predefined by **cgetnum**() and **cgetstr**():

name#*number*	numeric capability *name* has value *number*
name=*string*	string capability *name* has value *string*
name#@	the numeric capability *name* does not exist
name=@	the string capability *name* does not exist

Numeric capability values may be given in one of three numeric bases. If the number starts with either '0x' or '0X' it is interpreted as a hexadecimal number (both upper and lower case a-f may be used to denote the extended hexadecimal digits). Otherwise, if the number starts with a '0' it is interpreted as an octal number. Otherwise the number is interpreted as a decimal number.

String capability values may contain any character. Non-printable ASCII codes, new lines, and colons may be conveniently represented by the use of escape sequences:

^X	('X' & 037)	control-X
\b, \B	(ASCII 010)	backspace
\t, \T	(ASCII 011)	tab
\n, \N	(ASCII 012)	line feed (newline)
\f, \F	(ASCII 014)	form feed
\r, \R	(ASCII 015)	carriage return

\e, \E	(ASCII 027)	escape
\c, \C	(:)	colon
\\	(\)	back slash
\^	(^)	caret
\nnn	(ASCII octal nnn)	

A '\' may be followed by up to three octal digits directly specifies the numeric code for a character. The use of ASCII NULs, while easily encoded, causes all sorts of problems and must be used with care since NULs are typically used to denote the end of strings; many applications use '\200' to represent a NUL.

DIAGNOSTICS

The **cgetent**(), **cgetset**(), **cgetmatch**(), **cgetnum**(), **cgetstr**(), **cgetustr**(), **cgetfirst**(), and **cgetnext**() functions return a value greater than or equal to 0 on success and a value less than 0 on failure. The **cgetcap**() function returns a character pointer on success and a NULL on failure.

The **cgetent**(), and **cgetset**() functions may fail and set *errno* for any of the errors specified for the library functions: fopen(3), fclose(3), open(2), and close(2).

The **cgetent**(), **cgetset**(), **cgetstr**(), and **cgetustr**() functions may fail and set *errno* as follows:

[ENOMEM] No memory to allocate.

SEE ALSO

cap_mkdb(1), malloc(3)

BUGS

Colons (':') cannot be used in names, types, or values.

There are no checks for **tc=name** loops in **cgetent**().

The buffer added to the database by a call to **cgetset**() is not unique to the database but is rather prepended to any database used.

NAME

chdir, **fchdir** - change current working directory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
chdir(*const char *path*);

int
fchdir(*int fd*);

DESCRIPTION

The *path* argument points to the pathname of a directory. The **chdir**() system call causes the named directory to become the current working directory, that is, the starting point for path searches of pathnames not beginning with a slash, '/'.

The **fchdir**() system call causes the directory referenced by *fd* to become the current working directory, the starting point for path searches of pathnames not beginning with a slash, '/'.

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **chdir**() system call will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named directory does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EACCES] Search permission is denied for any component of the path name.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

The **fchdir**() system call will fail and the current working directory will be unchanged if one or more of the following are true:

[EACCES] Search permission is denied for the directory referenced by the file descriptor.

[ENOTDIR] The file descriptor does not reference a directory.

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

SEE ALSO

chroot(2)

STANDARDS

The **chdir**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **chdir**() system call appeared in Version 1 AT&T UNIX. The **fchdir**() system call appeared in 4.2BSD.

NAME

check_utility_compat - determine whether a utility should be compatible

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
check_utility_compat(*const char *utility*);

DESCRIPTION

The **check_utility_compat**() function checks whether *utility* should behave in a traditional (FreeBSD 4.7-compatible) manner, or in accordance with IEEE Std 1003.1-2001 ("POSIX.1"). The configuration is given as a comma-separated list of utility names; if the list is present but empty, all supported utilities assume their most compatible mode. The **check_utility_compat**() function first checks for an environment variable named _COMPAT_FreeBSD_4. If that environment variable does not exist, then **check_utility_compat**() will attempt to read the contents of a symbolic link named */etc/compat-FreeBSD-4-util*. If no configuration is found, compatibility mode is disabled.

RETURN VALUES

The **check_utility_compat**() function returns zero if *utility* should implement strict IEEE Std 1003.1-2001 ("POSIX.1") behavior, and nonzero otherwise.

FILES

/etc/compat-FreeBSD-4-util	If present, a symbolic link whose expansion gives system-wide default settings for the **check_utility_compat**() function.

ERRORS

No errors are detected.

HISTORY

The **check_utility_compat**() function first appeared in FreeBSD 5.0.

AUTHORS

This manual page was written by Garrett Wollman <*wollman@FreeBSD.org*>.

NAME

chflags, **lchflags**, **fchflags**, **chflagsat** - set file flags

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>
#include <unistd.h>

int
chflags(*const char *path*, *unsigned long flags*);

int
lchflags(*const char *path*, *unsigned long flags*);

int
fchflags(*int fd*, *unsigned long flags*);

int
chflagsat(*int fd*, *const char *path*, *unsigned long flags*, *int atflag*);

DESCRIPTION

The file whose name is given by *path* or referenced by the descriptor *fd* has its flags changed to *flags*.

The **lchflags**() system call is like **chflags**() except in the case where the named file is a symbolic link, in which case **lchflags**() will change the flags of the link itself, rather than the file it points to.

The **chflagsat**() is equivalent to either **chflags**() or **lchflags**() depending on the *atflag* except in the case where *path* specifies a relative path. In this case the file to be changed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. The values for the *atflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, then the flags of the symbolic link are changed.

If **chflagsat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used. If also *atflag* is zero, the behavior is identical to a call to **chflags**().

The flags specified are formed by *or*'ing the following values

SF_APPEND	The file may only be appended to.
SF_ARCHIVED	The file has been archived. This flag means the opposite of the DOS, Windows and CIFS FILE_ATTRIBUTE_ARCHIVE attribute. This flag has been deprecated, and may be removed in a future release.
SF_IMMUTABLE	The file may not be changed.
SF_NOUNLINK	The file may not be renamed or deleted.
SF_SNAPSHOT	The file is a snapshot file.
UF_APPEND	The file may only be appended to.
UF_ARCHIVE	The file needs to be archived. This flag has the same meaning as the DOS, Windows and CIFS FILE_ATTRIBUTE_ARCHIVE attribute. Filesystems in FreeBSD may or may not have special handling for this flag. For instance, ZFS tracks changes to files and will set this bit when a file is updated. UFS only stores the flag, and relies on the application to change it when needed.
UF_HIDDEN	The file may be hidden from directory listings at the application's discretion. The file has the DOS, Windows and CIFS FILE_ATTRIBUTE_HIDDEN attribute.
UF_IMMUTABLE	The file may not be changed.
UF_NODUMP	Do not dump the file.
UF_NOUNLINK	The file may not be renamed or deleted.
UF_OFFLINE	The file is offline, or has the Windows and CIFS FILE_ATTRIBUTE_OFFLINE attribute. Filesystems in FreeBSD store and display this flag, but do not provide any special handling when it is set.
UF_OPAQUE	The directory is opaque when viewed through a union stack.
UF_READONLY	The file is read only, and may not be written or appended. Filesystems may use this flag to maintain compatibility with the DOS, Windows and CIFS FILE_ATTRIBUTE_READONLY attribute.
UF_REPARSE	The file contains a Windows reparse point and has the Windows and CIFS FILE_ATTRIBUTE_REPARSE_POINT attribute.
UF_SPARSE	The file has the Windows FILE_ATTRIBUTE_SPARSE_FILE attribute. This may also be used by a filesystem to indicate a sparse file.
UF_SYSTEM	The file has the DOS, Windows and CIFS FILE_ATTRIBUTE_SYSTEM attribute. Filesystems in FreeBSD may store and display this flag, but do not provide any special handling when it is set.

If one of SF_IMMUTABLE, SF_APPEND, or SF_NOUNLINK is set a non-super-user cannot change

any flags and even the super-user can change flags only if securelevel is 0. (See init(8) for details.)

The UF_IMMUTABLE, UF_APPEND, UF_NOUNLINK, UF_NODUMP, and UF_OPAQUE flags may be set or unset by either the owner of a file or the super-user.

The SF_IMMUTABLE, SF_APPEND, SF_NOUNLINK, and SF_ARCHIVED flags may only be set or unset by the super-user. Attempts to toggle these flags by non-super-users are rejected. These flags may be set at any time, but normally may only be unset when the system is in single-user mode. (See init(8) for details.)

The implementation of all flags is filesystem-dependent. See the description of the UF_ARCHIVE flag above for one example of the differences in behavior. Care should be exercised when writing applications to account for support or lack of support of these flags in various filesystems.

The SF_SNAPSHOT flag is maintained by the system and cannot be toggled.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **chflags**() system call will fail if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not the super-user.

[EPERM] One of SF_IMMUTABLE, SF_APPEND, or SF_NOUNLINK is set and the user is either not the super-user or securelevel is greater than 0.

[EPERM] A non-super-user attempted to toggle one of SF_ARCHIVED, SF_IMMUTABLE, SF_APPEND, or SF_NOUNLINK.

[EPERM] An attempt was made to toggle the SF_SNAPSHOT flag.

[EROFS] The named file resides on a read-only file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EOPNOTSUPP] The underlying file system does not support file flags, or does not support all of the flags set in *flags*.

The **fchflags**() system call will fail if:

[EBADF] The descriptor is not valid.

[EINVAL] The *fd* argument refers to a socket, not to a file.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not the super-user.

[EPERM] One of SF_IMMUTABLE, SF_APPEND, or SF_NOUNLINK is set and the user is either not the super-user or securelevel is greater than 0.

[EPERM] A non-super-user attempted to toggle one of SF_ARCHIVED, SF_IMMUTABLE, SF_APPEND, or SF_NOUNLINK.

[EPERM] An attempt was made to toggle the SF_SNAPSHOT flag.

[EROFS] The file resides on a read-only file system.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EOPNOTSUPP] The underlying file system does not support file flags, or does not support all of the flags set in *flags*.

SEE ALSO

chflags(1), fflagstostr(3), strtofflags(3), init(8), mount_unionfs(8)

HISTORY

The **chflags**() and **fchflags**() system calls first appeared in 4.4BSD. The **lchflags**() system call first appeared in FreeBSD 5.0. The **chflagsat**() system call first appeared in FreeBSD 10.0.

NAME

chmod, **fchmod**, **lchmod**, **fchmodat** - change mode of file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>

int
chmod(*const char *path*, *mode_t mode*);

int
fchmod(*int fd*, *mode_t mode*);

int
lchmod(*const char *path*, *mode_t mode*);

int
fchmodat(*int fd*, *const char *path*, *mode_t mode*, *int flag*);

DESCRIPTION

The file permission bits of the file named specified by *path* or referenced by the file descriptor *fd* are changed to *mode*. The **chmod**() system call verifies that the process owner (user) either owns the file specified by *path* (or *fd*), or is the super-user. The **chmod**() system call follows symbolic links to operate on the target of the link rather than the link itself.

The **lchmod**() system call is similar to **chmod**() but does not follow symbolic links.

The **fchmodat**() is equivalent to either **chmod**() or **lchmod**() depending on the *flag* except in the case where *path* specifies a relative path. In this case the file to be changed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. The values for the *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, then the mode of the symbolic link is changed.

If **fchmodat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used. If also *flag* is zero, the behavior is identical to a call to **chmod**().

A mode is created from *or'd* permission bit masks defined in <*sys/stat.h*>:

```
#define S_IRWXU 0000700    /* RWX mask for owner */
#define S_IRUSR 0000400    /* R for owner */
#define S_IWUSR 0000200    /* W for owner */
#define S_IXUSR 0000100    /* X for owner */

#define S_IRWXG 0000070    /* RWX mask for group */
#define S_IRGRP 0000040    /* R for group */
#define S_IWGRP 0000020    /* W for group */
#define S_IXGRP 0000010    /* X for group */

#define S_IRWXO 0000007    /* RWX mask for other */
#define S_IROTH 0000004    /* R for other */
#define S_IWOTH 0000002    /* W for other */
#define S_IXOTH 0000001    /* X for other */

#define S_ISUID 0004000    /* set user id on execution */
#define S_ISGID 0002000    /* set group id on execution */
#define S_ISVTX 0001000    /* sticky bit */
```

The non-standard S_ISTXT is a synonym for S_ISVTX.

The FreeBSD VM system totally ignores the sticky bit (S_ISVTX) for executables. On UFS-based file systems (FFS, LFS) the sticky bit may only be set upon directories.

If mode S_ISVTX (the 'sticky bit') is set on a directory, an unprivileged user may not delete or rename files of other users in that directory. The sticky bit may be set by any user on a directory which the user owns or has appropriate permissions. For more details of the properties of the sticky bit, see sticky(7).

If mode ISUID (set UID) is set on a directory, and the MNT_SUIDDIR option was used in the mount of the file system, then the owner of any new files and sub-directories created within this directory are set to be the same as the owner of that directory. If this function is enabled, new directories will inherit the bit from their parents. Execute bits are removed from the file, and it will not be given to root. This behavior does not change the requirements for the user to be allowed to write the file, but only the eventual owner after it has been created. Group inheritance is not affected.

This feature is designed for use on fileservers serving PC users via ftp, SAMBA, or netatalk. It provides security holes for shell users and as such should not be used on shell machines, especially on home directories. This option requires the SUIDDIR option in the kernel to work. Only UFS file systems

support this option. For more details of the suiddir mount option, see mount(8).

Writing or changing the owner of a file turns off the set-user-id and set-group-id bits unless the user is the super-user. This makes the system somewhat more secure by protecting set-user-id (set-group-id) files from remaining set-user-id (set-group-id) if they are modified, at the expense of a degree of compatibility.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **chmod**() system call will fail and the file mode will be unchanged if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not the super-user.

[EPERM] The effective user ID is not the super-user, the effective user ID do match the owner of the file, but the group ID of the file does not match the effective group ID nor one of the supplementary group IDs.

[EPERM] The named file has its immutable or append-only flag set, see the chflags(2) manual page for more information.

[EROFS] The named file resides on a read-only file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EFTYPE] The effective user ID is not the super-user, the mode includes the sticky bit (S_ISVTX), and path does not refer to a directory.

The **fchmod**() system call will fail if:

[EBADF] The descriptor is not valid.

[EINVAL] The *fd* argument refers to a socket, not to a file.

[EROFS] The file resides on a read-only file system.

[EIO] An I/O error occurred while reading from or writing to the file system.

In addition to the **chmod**() errors, **fchmodat**() fails if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither *AT_FDCWD* nor a valid file descriptor open for searching.

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chmod(1), chflags(2), chown(2), open(2), stat(2), sticky(7)

STANDARDS

The **chmod**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"), except for the return of EFTYPE. The S_ISVTX bit on directories is expected to conform to Version 3 of the Single UNIX Specification ("SUSv3"). The **fchmodat**() system call is expected to conform to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **chmod**() function appeared in Version 1 AT&T UNIX. The **fchmod**() system call appeared in 4.2BSD. The **lchmod**() system call appeared in FreeBSD 3.0. The **fchmodat**() system call appeared in FreeBSD 8.0.

NAME

chown, **fchown**, **lchown**, **fchownat** - change owner and group of a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
chown(*const char *path*, *uid_t owner*, *gid_t group*);

int
fchown(*int fd*, *uid_t owner*, *gid_t group*);

int
lchown(*const char *path*, *uid_t owner*, *gid_t group*);

int
fchownat(*int fd*, *const char *path*, *uid_t owner*, *gid_t group*, *int flag*);

DESCRIPTION

The owner ID and group ID of the file named by *path* or referenced by *fd* is changed as specified by the arguments *owner* and *group*. The owner of a file may change the *group* to a group of which he or she is a member, but the change *owner* capability is restricted to the super-user.

The **chown**() system call clears the set-user-id and set-group-id bits on the file to prevent accidental or mischievous creation of set-user-id and set-group-id programs if not executed by the super-user. The **chown**() system call follows symbolic links to operate on the target of the link rather than the link itself.

The **fchown**() system call is particularly useful when used in conjunction with the file locking primitives (see flock(2)).

The **lchown**() system call is similar to **chown**() but does not follow symbolic links.

The **fchownat**() system call is equivalent to the **chown**() and **lchown**() except in the case where *path* specifies a relative path. In this case the file to be changed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in

<*fcntl.h*>:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, ownership of the symbolic link is changed.

If **fchownat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **chown**() or **lchown**() respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in the *flag* argument.

One of the owner or group id's may be left unchanged by specifying it as -1.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **chown**() and **lchown**() will fail and the file will be unchanged if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The operation would change the ownership, but the effective user ID is not the super-user.

[EPERM] The named file has its immutable or append-only flag set, see the chflags(2) manual page for more information.

[EROFS] The named file resides on a read-only file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

The **fchown**() system call will fail if:

[EBADF] The *fd* argument does not refer to a valid descriptor.

[EINVAL] The *fd* argument refers to a socket, not a file.

[EPERM] The effective user ID is not the super-user.

[EROFS] The named file resides on a read-only file system.

[EIO] An I/O error occurred while reading from or writing to the file system.

In addition to the errors specified for **chown**() and **lchown**(), the **fchownat**() system call may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chgrp(1), chflags(2), chmod(2), flock(2), chown(8)

STANDARDS

The **chown**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **fchownat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **chown**() function appeared in Version 1 AT&T UNIX. The **fchown**() system call appeared in 4.2BSD.

The **chown**() system call was changed to follow symbolic links in 4.4BSD. The **lchown**() system call was added in FreeBSD 3.0 to compensate for the loss of functionality.

The **fchownat**() system call appeared in FreeBSD 8.0.

NAME

chroot - change root directory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
chroot(*const char *dirname*);

DESCRIPTION

The *dirname* argument is the address of the pathname of a directory, terminated by an ASCII NUL. The **chroot**() system call causes *dirname* to become the root directory, that is, the starting point for path searches of pathnames beginning with '/'.

In order for a directory to become the root directory a process must have execute (search) access for that directory.

It should be noted that **chroot**() has no effect on the process's current directory.

This call is restricted to the super-user.

Depending on the setting of the 'kern.chroot_allow_open_directories' sysctl variable, open filedescriptors which reference directories will make the **chroot**() fail as follows:

If 'kern.chroot_allow_open_directories' is set to zero, **chroot**() will always fail with EPERM if there are any directories open.

If 'kern.chroot_allow_open_directories' is set to one (the default), **chroot**() will fail with EPERM if there are any directories open and the process is already subject to the **chroot**() system call.

Any other value for 'kern.chroot_allow_open_directories' will bypass the check for open directories

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **chroot**() system call will fail and the root directory will be unchanged if:

[ENOTDIR] A component of the path name is not a directory.

[EPERM] The effective user ID is not the super-user, or one or more filedescriptors are open directories.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named directory does not exist.

[EACCES] Search permission is denied for any component of the path name.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EFAULT] The *dirname* argument points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

chdir(2), jail(2)

HISTORY

The **chroot**() system call appeared in 4.2BSD. It was marked as "legacy" in Version 2 of the Single UNIX Specification ("SUSv2"), and was removed in subsequent standards.

BUGS

If the process is able to change its working directory to the target directory, but another access control check fails (such as a check for open directories, or a MAC check), it is possible that this system call may return an error, with the working directory of the process left changed.

SECURITY CONSIDERATIONS

The system have many hardcoded paths to files where it may load after the process starts. It is generally recommended to drop privileges immediately after a successful **chroot** call, and restrict write access to a limited subtree of the **chroot** root, for instance, setup the sandbox so that the sandboxed user will have no write access to any well-known system directories.

NAME

clearerr, **clearerr_unlocked**, **feof**, **feof_unlocked**, **ferror**, **ferror_unlocked**, **fileno**, **fileno_unlocked** - check and reset stream status

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

void
clearerr(*FILE *stream*);

void
clearerr_unlocked(*FILE *stream*);

int
feof(*FILE *stream*);

int
feof_unlocked(*FILE *stream*);

int
ferror(*FILE *stream*);

int
ferror_unlocked(*FILE *stream*);

int
fileno(*FILE *stream*);

int
fileno_unlocked(*FILE *stream*);

DESCRIPTION

The function **clearerr**() clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function **feof**() tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator may be cleared by explicitly calling **clearerr**(), or as a side-effect of other operations, e.g. **fseek**().

The function **ferror**() tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

The **clearerr_unlocked**(), **feof_unlocked**(), **ferror_unlocked**(), and **fileno_unlocked**() functions are equivalent to **clearerr**(), **feof**(), **ferror**(), and **fileno**() respectively, except that the caller is responsible for locking the stream with flockfile(3) before calling them. These functions may be used to avoid the overhead of locking the stream and to prevent races when multiple threads are operating on the same stream.

ERRORS

These functions should not fail and do not set the external variable *errno*.

SEE ALSO

open(2), fdopen(3), flockfile(3), stdio(3)

STANDARDS

The functions **clearerr**(), **feof**(), and **ferror**() conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

rpc_clnt_create, **clnt_control**, **clnt_create**, **clnt_create_timed**, **clnt_create_vers**, **clnt_create_vers_timed**, **clnt_destroy**, **clnt_dg_create**, **clnt_pcreateerror**, **clnt_raw_create**, **clnt_spcreateerror**, **clnt_tli_create**, **clnt_tp_create**, **clnt_tp_create_timed**, **clnt_vc_create**, **rpc_createerr** - library routines for dealing with creation and manipulation of *CLIENT* handles

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

bool_t
clnt_control(*CLIENT *clnt*, *const u_int req*, *char *info*);

*CLIENT **
clnt_create(*const char * host*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *const char *nettype*);

*CLIENT **
clnt_create_timed(*const char * host*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *const char *nettype*, *const struct timeval *timeout*);

*CLIENT **
clnt_create_vers(*const char * host*, *const rpcprog_t prognum*, *rpcvers_t *vers_outp*, *const rpcvers_t vers_low*, *const rpcvers_t vers_high*, *const char *nettype*);

*CLIENT **
clnt_create_vers_timed(*const char * host*, *const rpcprog_t prognum*, *rpcvers_t *vers_outp*, *const rpcvers_t vers_low*, *const rpcvers_t vers_high*, *const char *nettype*, *const struct timeval *timeout*);

void
clnt_destroy(*CLIENT *clnt*);

*CLIENT **
clnt_dg_create(*const int fildes*, *const struct netbuf *svcaddr*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *const u_int sendsz*, *const u_int recvsz*);

void
clnt_pcreateerror(*const char *s*);

*char **
clnt_spcreateerror(*const char *s*);

*CLIENT **
clnt_raw_create(*const rpcprog_t prognum*, *const rpcvers_t versnum*);

*CLIENT **
clnt_tli_create(*const int fildes*, *const struct netconfig *netconf*, *struct netbuf *svcaddr*,
const rpcprog_t prognum, *const rpcvers_t versnum*, *const u_int sendsz*, *const u_int recvsz*);

*CLIENT **
clnt_tp_create(*const char * host*, *const rpcprog_t prognum*, *const rpcvers_t versnum*,
*const struct netconfig *netconf*);

*CLIENT **
clnt_tp_create_timed(*const char * host*, *const rpcprog_t prognum*, *const rpcvers_t versnum*,
*const struct netconfig *netconf*, *const struct timeval *timeout*);

*CLIENT **
clnt_vc_create(*const int fildes*, *const struct netbuf *svcaddr*, *const rpcprog_t prognum*,
const rpcvers_t versnum, *u_int sendsz*, *u_int recvsz*);

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First a *CLIENT* handle is created and then the client calls a procedure to send a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends a reply.

Routines

clnt_control()

A function macro to change or retrieve various information about a client object. The *req* argument indicates the type of operation, and *info* is a pointer to the information. For both connectionless and connection-oriented transports, the supported values of *req* and their argument types and what they do are:

CLSET_TIMEOUT	struct timeval *	set total timeout
CLGET_TIMEOUT	struct timeval *	get total timeout

Note: if you set the timeout using **clnt_control**(), the timeout argument passed by **clnt_call**() is ignored in all subsequent calls.

Note: If you set the timeout value to 0, **clnt_control**() immediately returns an error (RPC_TIMEDOUT). Set the timeout argument to 0 for batching calls.

CLGET_SVC_ADDR	struct netbuf *	get servers address
CLGET_FD	int *	get fd from handle
CLSET_FD_CLOSE	void	close fd on destroy
CLSET_FD_NCLOSE	void	do not close fd on destroy
CLGET_VERS	uint32_t *	get RPC program version
CLSET_VERS	uint32_t *	set RPC program version
CLGET_XID	uint32_t *	get XID of previous call
CLSET_XID	uint32_t *	set XID of next call

The following operations are valid for connectionless transports only:

CLSET_RETRY_TIMEOUT	struct timeval *	set the retry timeout
CLGET_RETRY_TIMEOUT	struct timeval *	get the retry timeout
CLSET_CONNECT	*int **	use connect(2)

The retry timeout is the time that RPC waits for the server to reply before retransmitting the request. The **clnt_control**() function returns TRUE on success and FALSE on failure.

clnt_create() Generic client creation routine for program *prognum* and version *versnum*. The *host* argument identifies the name of the remote host where the server is located. The *nettype* argument indicates the class of transport protocol to use. The transports are tried in left to right order in NETPATH environment variable or in top to bottom order in the netconfig database. The **clnt_create**() function tries all the transports of the *nettype* class available from the NETPATH environment variable and the netconfig database, and chooses the first successful one. A default timeout is set and can be modified using **clnt_control**(). This routine returns NULL if it fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure.

Note: **clnt_create**() returns a valid client handle even if the particular version number supplied to **clnt_create**() is not registered with the rpcbind(8) service. This mismatch will be discovered by a **clnt_call**() later (see rpc_clnt_calls(3)).

clnt_create_timed()

Generic client creation routine which is similar to **clnt_create**() but which also has the additional argument *timeout* that specifies the maximum amount of time allowed for each transport class tried. In all other respects, the **clnt_create_timed**() call behaves exactly like the **clnt_create**() call.

clnt_create_vers()

Generic client creation routine which is similar to **clnt_create**() but which also checks for the version availability. The *host* argument identifies the name of the remote host where the server is located. The *nettype* argument indicates the class transport protocols to be used. If the routine is successful it returns a client handle created for the highest version between *vers_low* and *vers_high* that is supported by the server. The *vers_outp* argument is set to this value. That is, after a successful return *vers_low* <= **vers_outp* <= *vers_high*. If no version between *vers_low* and *vers_high* is supported by the server then the routine fails and returns NULL. A default timeout is set and can be modified using **clnt_control**(). This routine returns NULL if it fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure. Note: **clnt_create**() returns a valid client handle even if the particular version number supplied to **clnt_create**() is not registered with the rpcbind(8) service. This mismatch will be discovered by a **clnt_call**() later (see rpc_clnt_calls(3)). However, **clnt_create_vers**() does this for you and returns a valid handle only if a version within the range supplied is supported by the server.

clnt_create_vers_timed()

Generic client creation routine which is similar to **clnt_create_vers**() but which also has the additional argument *timeout* that specifies the maximum amount of time allowed for each transport class tried. In all other respects, the **clnt_create_vers_timed**() call behaves exactly like the **clnt_create_vers**() call.

clnt_destroy()

A function macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy**(). If the RPC library opened the associated file descriptor, or CLSET_FD_CLOSE was set using **clnt_control**(), the file descriptor will be closed. The caller should call **auth_destroy**(*clnt->cl_auth*) (before calling **clnt_destroy**()) to destroy the associated *AUTH* structure (see rpc_clnt_auth(3)).

clnt_dg_create()

This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connectionless transport. The remote program is located at address *svcaddr*. The *fildes* argument is an open and bound file descriptor. This routine will resend the call message in intervals of 15 seconds until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**() (see **clnt_call**() in rpc_clnt_calls(3)). The retry time out and the total time out periods can be changed using **clnt_control**(). The user may set the size of the send and receive buffers with the *sendsz* and *recvsz* arguments; values of 0 choose suitable defaults. This routine returns NULL if it fails.

clnt_pcreateerror()

Print a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string *s* and a colon, and appended with a newline.

clnt_spcreateerror()

Like **clnt_pcreateerror**(), except that it returns a string instead of printing to the standard error. A newline is not appended to the message in this case. Warning: returns a pointer to a buffer that is overwritten on each call.

clnt_raw_create()

This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The transport used to pass messages to the service is a buffer within the process's address space, so the corresponding RPC server should live in the same address space; (see **svc_raw_create**() in rpc_svc_create(3)). This allows simulation of RPC and measurement of RPC overheads, such as round trip times, without any kernel or networking interference. This routine returns NULL if it fails. The **clnt_raw_create**() function should be called after **svc_raw_create**().

clnt_tli_create()

This routine creates an RPC client handle for the remote program *prognum* and version *versnum*. The remote program is located at address *svcaddr*. If *svcaddr* is NULL and it is connection-oriented, it is assumed that the file descriptor is connected. For connectionless transports, if *svcaddr* is NULL, RPC_UNKNOWNADDR error is set. The *fildes* argument is a file descriptor which may be open, bound and connected. If it is RPC_ANYFD, it opens a file descriptor on the transport specified by *netconf*. If *fildes* is RPC_ANYFD and *netconf* is NULL, a RPC_UNKNOWNPROTO error is set. If *fildes* is unbound, then it will attempt to bind the descriptor. The user may specify the size of the buffers with the *sendsz* and *recvsz* arguments; values of 0 choose suitable defaults. Depending upon the type of the transport (connection-oriented or connectionless), **clnt_tli_create**() calls appropriate client creation routines. This routine returns NULL if it fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure. The remote rpcbind service (see rpcbind(8)) is not consulted for the address of the remote service.

clnt_tp_create()

Like **clnt_create**() except **clnt_tp_create**() tries only one transport specified through *netconf*. The **clnt_tp_create**() function creates a client handle for the program *prognum*, the version *versnum*, and for the transport specified by *netconf*. Default options are set, which can be changed using **clnt_control**() calls. The remote rpcbind service on the host *host* is consulted for the address of the remote service. This routine returns NULL if it

fails. The **clnt_pcreateerror**() routine can be used to print the reason for failure.

clnt_tp_create_timed()

Like **clnt_tp_create**() except **clnt_tp_create_timed**() has the extra argument *timeout* which specifies the maximum time allowed for the creation attempt to succeed. In all other respects, the **clnt_tp_create_timed**() call behaves exactly like the **clnt_tp_create**() call.

clnt_vc_create()

This routine creates an RPC client for the remote program *prognum* and version *versnum*; the client uses a connection-oriented transport. The remote program is located at address *svcaddr*. The *fildes* argument is an open and bound file descriptor. The user may specify the size of the send and receive buffers with the *sendsz* and *recvsz* arguments; values of 0 choose suitable defaults. This routine returns NULL if it fails. The address *svcaddr* should not be NULL and should point to the actual address of the remote program. The **clnt_vc_create**() function does not consult the remote rpcbind service for this information.

struct rpc_createerr rpc_createerr;

A global variable whose value is set by any RPC client handle creation routine that fails. It is used by the routine **clnt_pcreateerror**() to print the reason for the failure.

SEE ALSO

rpc(3), rpc_clnt_auth(3), rpc_clnt_calls(3), rpcbind(8)

NAME

clock_getcpuclockid - access a process CPU-time clock

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

int
clock_getcpuclockid(*pid_t pid*, *clockid_t *clock_id*);

DESCRIPTION

The **clock_getcpuclockid**() returns the clock ID of the CPU-time clock of the process specified by *pid*. If the process described by *pid* exists and the calling process has permission, the clock ID of this clock will be returned in *clock_id*.

If *pid* is zero, the **clock_getcpuclockid**() function returns the clock ID of the CPU-time clock of the process making the call, in *clock_id*.

RETURN VALUES

Upon successful completion, **clock_getcpuclockid**() returns zero; otherwise, an error number is returned to indicate the error.

ERRORS

The **clock_getcpuclockid**() function will fail if:

[EPERM] The requesting process does not have permission to access the CPU-time clock for the process.

[ESRCH] No process can be found corresponding to the process specified by *pid*.

SEE ALSO

clock_gettime(2)

STANDARDS

The **clock_getcpuclockid**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **clock_getcpuclockid**() function first appeared in FreeBSD 10.0.

AUTHORS

David Xu <*davidxu@FreeBSD.org*>

NAME

clock_gettime, **clock_settime**, **clock_getres** - get/set/calibrate date and time

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

int
clock_gettime(*clockid_t clock_id*, *struct timespec *tp*);

int
clock_settime(*clockid_t clock_id*, *const struct timespec *tp*);

int
clock_getres(*clockid_t clock_id*, *struct timespec *tp*);

DESCRIPTION

The **clock_gettime**() and **clock_settime**() system calls allow the calling process to retrieve or set the value used by a clock which is specified by *clock_id*.

The *clock_id* argument can be a value obtained from clock_getcpuclockid(3) or pthread_getcpuclockid(3) as well as the following values:

CLOCK_REALTIME
CLOCK_REALTIME_PRECISE
CLOCK_REALTIME_FAST
: Increments as a wall clock should.

CLOCK_MONOTONIC
CLOCK_MONOTONIC_PRECISE
CLOCK_MONOTONIC_FAST
: Increments in SI seconds.

CLOCK_UPTIME
CLOCK_UPTIME_PRECISE
CLOCK_UPTIME_FAST
: Starts at zero when the kernel boots and increments monotonically in SI seconds while the machine is running.

CLOCK_VIRTUAL
: Increments only when the CPU is running in user mode on behalf of the calling process.

CLOCK_PROF
: Increments when the CPU is running in user or kernel mode.

CLOCK_SECOND
: Returns the current second without performing a full time counter query, using an in-kernel cached value of the current second.

CLOCK_PROCESS_CPUTIME_ID
: Returns the execution time of the calling process.

CLOCK_THREAD_CPUTIME_ID
: Returns the execution time of the calling thread.

The clock IDs *CLOCK_REALTIME_FAST*, *CLOCK_MONOTONIC_FAST*, *CLOCK_UPTIME_FAST* are analogs of corresponding IDs without _FAST suffix but do not perform a full time counter query, so their accuracy is one timer tick. Similarly, *CLOCK_REALTIME_PRECISE*, *CLOCK_MONOTONIC_PRECISE*, *CLOCK_UPTIME_PRECISE* are used to get the most exact value as possible, at the expense of execution time.

The structure pointed to by *tp* is defined in <*sys/timespec.h*> as:

```
struct timespec {
        time_t  tv_sec;         /* seconds */
        long    tv_nsec;  /* and nanoseconds */
};
```

Only the super-user may set the time of day, using only *CLOCK_REALTIME*. If the system securelevel is greater than 1 (see init(8)), the time may only be advanced. This limitation is imposed to prevent a malicious super-user from setting arbitrary time stamps on files. The system time can still be adjusted backwards using the adjtime(2) system call even when the system is secure.

The resolution (granularity) of a clock is returned by the **clock_getres**() system call. This value is placed in a (non-NULL) **tp*.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following error codes may be set in *errno*:

[EINVAL] The *clock_id* or *timespec* argument was not a valid value.

[EPERM] A user other than the super-user attempted to set the time.

SEE ALSO

date(1), adjtime(2), clock_getcpuclockid(3), ctime(3), pthread_getcpuclockid(3), timed(8)

STANDARDS

The **clock_gettime**(), **clock_settime**(), and **clock_getres**() system calls conform to IEEE Std 1003.1b-1993 ("POSIX.1b"). The clock IDs *CLOCK_REALTIME_FAST*, *CLOCK_REALTIME_PRECISE*, *CLOCK_MONOTONIC_FAST*, *CLOCK_MONOTONIC_PRECISE*, *CLOCK_UPTIME*, *CLOCK_UPTIME_FAST*, *CLOCK_UPTIME_PRECISE*, *CLOCK_SECOND* are FreeBSD extensions to the POSIX interface.

NAME

clock - determine processor time used

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

clock_t
clock(*void*);

DESCRIPTION

The **clock**() function determines the amount of processor time used since the invocation of the calling process, measured in CLOCKS_PER_SECs of a second.

RETURN VALUES

The **clock**() function returns the amount of time used unless an error occurs, in which case the return value is -1.

SEE ALSO

getrusage(2), clocks(7)

STANDARDS

The **clock**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). However, Version 2 of the Single UNIX Specification ("SUSv2") requires CLOCKS_PER_SEC to be defined as one million. FreeBSD does not conform to this requirement; changing the value would introduce binary incompatibility and one million is still inadequate on modern processors.

NAME

close - delete a descriptor

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
close(*int fd*);

DESCRIPTION

The **close**() system call deletes a descriptor from the per-process object reference table. If this is the last reference to the underlying object, the object will be deactivated. For example, on the last close of a file the current *seek* pointer associated with the file is lost; on the last close of a socket(2) associated naming information and queued data are discarded; on the last close of a file holding an advisory lock the lock is released (see further flock(2)). However, the semantics of System V and IEEE Std 1003.1-1988 ("POSIX.1") dictate that all fcntl(2) advisory record locks associated with a file for a given process are removed when *any* file descriptor for that file is closed by that process.

When a process exits, all associated file descriptors are freed, but since there is a limit on active descriptors per processes, the **close**() system call is useful when a large quantity of file descriptors are being handled.

When a process forks (see fork(2)), all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using execve(2), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with dup2(2) or deleted with **close**() before the execve(2) is attempted, but if some of these descriptors will still be needed if the execve fails, it is necessary to arrange for them to be closed if the execve succeeds. For this reason, the call "fcntl(d, F_SETFD, FD_CLOEXEC)" is provided, which arranges that a descriptor will be closed after a successful execve; the call "fcntl(d, F_SETFD, 0)" restores the default, which is to not close the descriptor.

RETURN VALUES

The **close**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **close**() system call will fail if:

[EBADF] The *fd* argument is not an active descriptor.

[EINTR] An interrupt was received.

[ENOSPC] The underlying object did not fit, cached data was lost.

[ECONNRESET] The underlying object was a stream socket that was shut down by the peer before all pending data was delivered.

In case of any error except EBADF, the supplied file descriptor is deallocated and therefore is no longer valid.

SEE ALSO

accept(2), closefrom(2), execve(2), fcntl(2), flock(2), open(2), pipe(2), socket(2), socketpair(2)

STANDARDS

The **close**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **close**() function appeared in Version 1 AT&T UNIX.

NAME

opendir, **fdopendir**, **readdir**, **readdir_r**, **telldir**, **seekdir**, **rewinddir**, **closedir**, **fdclosedir**, **dirfd** - directory operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <dirent.h>

*DIR **
opendir(*const char *filename*);

*DIR **
fdopendir(*int fd*);

*struct dirent **
readdir(*DIR *dirp*);

int
readdir_r(*DIR *dirp*, *struct dirent *entry*, *struct dirent **result*);

long
telldir(*DIR *dirp*);

void
seekdir(*DIR *dirp*, *long loc*);

void
rewinddir(*DIR *dirp*);

int
closedir(*DIR *dirp*);

int
fdclosedir(*DIR *dirp*);

int
dirfd(*DIR *dirp*);

DESCRIPTION

The readdir_r() interface is deprecated because it cannot be used correctly unless {*NAME_MAX*} is a fixed value.

The **opendir**() function opens the directory named by *filename*, associates a *directory stream* with it and returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed, or if it cannot malloc(3) enough memory to hold the whole thing.

The **fdopendir**() function is equivalent to the **opendir**() function except that the directory is specified by a file descriptor *fd* rather than by a name. The file offset associated with the file descriptor at the time of the call determines which entries are returned.

Upon successful return from **fdopendir**(), the file descriptor is under the control of the system, and if any attempt is made to close the file descriptor, or to modify the state of the associated description other than by means of **closedir**(), **readdir**(), **readdir_r**(), or **rewinddir**(), the behavior is undefined. Upon calling **closedir**() the file descriptor is closed. The FD_CLOEXEC flag is set on the file descriptor by a successful call to **fdopendir**().

The **readdir**() function returns a pointer to the next directory entry. The directory entry remains valid until the next call to **readdir**() or **closedir**() on the same *directory stream*. The function returns NULL upon reaching the end of the directory or on error. In the event of an error, *errno* may be set to any of the values documented for the getdirentries(2) system call.

The **readdir_r**() function provides the same functionality as **readdir**(), but the caller must provide a directory *entry* buffer to store the results in. The buffer must be large enough for a *struct dirent* with a *d_name* array with {*NAME_MAX*} + 1 elements. If the read succeeds, *result* is pointed at the *entry*; upon reaching the end of the directory *result* is set to NULL. The **readdir_r**() function returns 0 on success or an error number to indicate failure.

The **telldir**() function returns a token representing the current location associated with the named *directory stream*. Values returned by **telldir**() are good only for the lifetime of the DIR pointer, *dirp*, from which they are derived. If the directory is closed and then reopened, prior values returned by **telldir**() will no longer be valid. Values returned by **telldir**() are also invalidated by a call to **rewinddir**().

The **seekdir**() function sets the position of the next **readdir**() operation on the *directory stream*. The new position reverts to the one associated with the *directory stream* when the **telldir**() operation was performed.

The **rewinddir**() function resets the position of the named *directory stream* to the beginning of the

directory.

The **closedir**() function closes the named *directory stream* and frees the structure associated with the *dirp* pointer, returning 0 on success. On failure, -1 is returned and the global variable *errno* is set to indicate the error.

The **fdclosedir**() function is equivalent to the **closedir**() function except that this function returns directory file descriptor instead of closing it.

The **dirfd**() function returns the integer file descriptor associated with the named *directory stream*, see open(2).

Sample code which searches a directory for entry ‘‘name’’ is:

```
dirp = opendir(".");
if (dirp == NULL)
        return (ERROR);
len = strlen(name);
while ((dp = readdir(dirp)) != NULL) {
        if (dp->d_namlen == len && strcmp(dp->d_name, name) == 0) {
                (void)closedir(dirp);
                return (FOUND);
        }
}
(void)closedir(dirp);
return (NOT_FOUND);
```

SEE ALSO

close(2), lseek(2), open(2), read(2), dir(5)

HISTORY

The **opendir**(), **readdir**(), **telldir**(), **seekdir**(), **rewinddir**(), **closedir**(), and **dirfd**() functions appeared in 4.2BSD. The **fdopendir**() function appeared in FreeBSD 8.0. **fdclosedir**() function appeared in FreeBSD 10.0.

BUGS

The behaviour of **telldir**() and **seekdir**() is likely to be wrong if there are parallel unlinks happening and the directory is larger than one page. There is code to ensure that a **seekdir**() to the location given by a **telldir**() immediately before the last **readdir**() will always set the correct location to return the same value as that last **readdir**() performed. This is enough for some applications which want to "push back the last

entry read", e.g., Samba. Seeks back to any other location, other than the beginning of the directory, may result in unexpected behaviour if deletes are present. It is hoped that this situation will be resolved with changes to **getdirentries**() and the VFS.

NAME

closefrom - delete open file descriptors

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

void
closefrom(*int lowfd*);

DESCRIPTION

The **closefrom**() system call deletes all open file descriptors greater than or equal to *lowfd* from the per-process object reference table. Any errors encountered while closing file descriptors are ignored.

SEE ALSO

close(2)

HISTORY

The **closefrom**() function first appeared in FreeBSD 8.0.

NAME

syslog, **vsyslog**, **openlog**, **closelog**, **setlogmask** - control system log

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <syslog.h>
#include <stdarg.h>

void
syslog(*int priority*, *const char *message*, ...);

void
vsyslog(*int priority*, *const char *message*, *va_list args*);

void
openlog(*const char *ident*, *int logopt*, *int facility*);

void
closelog(*void*);

int
setlogmask(*int maskpri*);

DESCRIPTION

The **syslog**() function writes *message* to the system message logger. The message is then written to the system console, log files, logged-in users, or forwarded to other machines as appropriate. (See syslogd(8).)

The message is identical to a printf(3) format string, except that '%m' is replaced by the current error message. (As denoted by the global variable *errno*; see strerror(3).) A trailing newline is added if none is present.

The **vsyslog**() function is an alternate form in which the arguments have already been captured using the variable-length argument facilities of stdarg(3).

The message is tagged with *priority*. Priorities are encoded as a *facility* and a *level*. The facility describes the part of the system generating the message. The level is selected from the following *ordered* (high to low) list:

LOG_EMERG A panic condition. This is normally broadcast to all users.

LOG_ALERT A condition that should be corrected immediately, such as a corrupted system database.

LOG_CRIT Critical conditions, e.g., hard device errors.

LOG_ERR Errors.

LOG_WARNING Warning messages.

LOG_NOTICE Conditions that are not error conditions, but should possibly be handled specially.

LOG_INFO Informational messages.

LOG_DEBUG Messages that contain information normally of use only when debugging a program.

The **openlog**() function provides for more specialized processing of the messages sent by **syslog**() and **vsyslog**(). The *ident* argument is a string that will be prepended to every message. The *logopt* argument is a bit field specifying logging options, which is formed by OR'ing one or more of the following values:

LOG_CONS If **syslog**() cannot pass the message to syslogd(8) it will attempt to write the message to the console ("*/dev/console*").

LOG_NDELAY Open the connection to syslogd(8) immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.

LOG_PERROR Write the message to standard error output as well to the system log.

LOG_PID Log the process id with each message: useful for identifying instantiations of daemons. On FreeBSD, this option is enabled by default.

The *facility* argument encodes a default facility to be assigned to all messages that do not have an explicit facility encoded:

LOG_AUTH The authorization system: login(1), su(1), getty(8), etc.

LOG_AUTHPRIV The same as LOG_AUTH, but logged to a file readable only by selected individuals.

LOG_CONSOLE Messages written to */dev/console* by the kernel console output driver.

LOG_CRON The cron daemon: cron(8).

LOG_DAEMON System daemons, such as routed(8), that are not provided for explicitly by other facilities.

LOG_FTP The file transfer protocol daemons: ftpd(8), tftpd(8).

LOG_KERN Messages generated by the kernel. These cannot be generated by any user processes.

LOG_LPR The line printer spooling system: lpr(1), lpc(8), lpd(8), etc.

LOG_MAIL The mail system.

LOG_NEWS The network news system.

LOG_NTP The network time protocol system.

LOG_SECURITY Security subsystems, such as ipfw(4).

LOG_SYSLOG Messages generated internally by syslogd(8).

LOG_USER Messages generated by random user processes. This is the default facility identifier if none is specified.

LOG_UUCP The uucp system.

LOG_LOCAL0 Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

The **closelog**() function can be used to close the log file.

The **setlogmask**() function sets the log priority mask to *maskpri* and returns the previous mask. Calls to **syslog**() with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro **LOG_MASK**(*pri*); the mask for all priorities up to and including *toppri* is given by the macro **LOG_UPTO**(*toppri*);. The default allows all priorities to be logged.

RETURN VALUES

The routines **closelog**(), **openlog**(), **syslog**() and **vsyslog**() return no value.

The routine **setlogmask**() always returns the previous log mask level.

EXAMPLES

```
syslog(LOG_ALERT, "who: internal error 23");

openlog("ftpd", LOG_PID | LOG_NDELAY, LOG_FTP);

setlogmask(LOG_UPTO(LOG_ERR));

syslog(LOG_INFO, "Connection from host %d", CallingHost);

syslog(LOG_ERR|LOG_LOCAL2, "foobar error: %m");
```

SEE ALSO

logger(1), syslogd(8)

HISTORY

These functions appeared in 4.2BSD.

BUGS

Never pass a string with user-supplied data as a format without using '%s'. An attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if the string was built using a function like **snprintf**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by **syslog**().

Always use the proper secure idiom:

```
syslog(priority, "%s", string);
```

NAME

confstr - get string-valued configurable variables

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

size_t
confstr(*int name*, *char *buf*, *size_t len*);

DESCRIPTION

This interface is specified by IEEE Std 1003.1-2001 ("POSIX.1"). A more flexible (but non-portable) interface is provided by sysctl(3).

The **confstr**() function provides a method for applications to get configuration defined string values. Shell programmers needing access to these parameters should use the getconf(1) utility.

The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file <*unistd.h*>. The *len* argument specifies the size of the buffer referenced by the argument *buf*. If *len* is non-zero, *buf* is a non-null pointer, and *name* has a value, up to *len* - 1 bytes of the value are copied into the buffer *buf*. The copied value is always null terminated.

The available values are as follows:

_CS_PATH
Return a value for the PATH environment variable that finds all the standard utilities.

RETURN VALUES

If the call to **confstr**() is not successful, 0 is returned and *errno* is set appropriately. Otherwise, if the variable does not have a configuration defined value, 0 is returned and *errno* is not modified. Otherwise, the buffer size needed to hold the entire configuration-defined value is returned. If this size is greater than the argument *len*, the string in *buf* was truncated.

ERRORS

The **confstr**() function may fail and set *errno* for any of the errors specified for the library functions malloc(3) and sysctl(3).

In addition, the following errors may be reported:

[EINVAL] The value of the *name* argument is invalid.

SEE ALSO

getconf(1), pathconf(2), sysconf(3), sysctl(3)

HISTORY

The **confstr**() function first appeared in 4.4BSD.

NAME

connect - initiate a connection on a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

int
connect(*int s*, *const struct sockaddr *name*, *socklen_t namelen*);

DESCRIPTION

The *s* argument is a socket. If it is of type SOCK_DGRAM, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. *namelen* indicates the amount of space pointed to by *name*, in bytes; the *sa_len* member of *name* is ignored. Each communications space interprets the *name* argument in its own way. Generally, stream sockets may successfully **connect**() only once; datagram sockets may use **connect**() multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUES

The **connect**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **connect**() system call fails if:

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

[EINVAL] The *namelen* argument is not a valid length for the address family.

[ENOTSOCK] The *s* argument is a descriptor for a file, not a socket.

[EADDRNOTAVAIL] The specified address is not available on this machine.

[EAFNOSUPPORT] Addresses in the specified address family cannot be used with this socket.

[EISCONN] The socket is already connected.

[ETIMEDOUT] Connection establishment timed out without establishing a connection.

[ECONNREFUSED]
The attempt to connect was forcefully rejected.

[ECONNRESET] The connection was reset by the remote host.

[ENETUNREACH] The network is not reachable from this host.

[EHOSTUNREACH]
The remote host is not reachable from this host.

[EADDRINUSE] The address is already in use.

[EFAULT] The *name* argument specifies an area outside the process address space.

[EINPROGRESS] The socket is non-blocking and the connection cannot be completed immediately. It is possible to select(2) for completion by selecting the socket for writing.

[EINTR] The connection attempt was interrupted by the delivery of a signal. The connection will be established in the background, as in the case of EINPROGRESS.

[EALREADY] A previous connection attempt has not yet been completed.

[EACCES] An attempt is made to connect to a broadcast address (obtained through the INADDR_BROADCAST constant or the INADDR_NONE return value) through a socket that does not provide broadcast functionality.

[EAGAIN] An auto-assigned port number was requested but no auto-assigned ports are available. Increasing the port range specified by sysctl(3) MIB variables *net.inet.ip.portrange.first* and *net.inet.ip.portrange.last* may alleviate the problem.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named socket does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write access to the named socket is denied.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] Write access to the named socket is denied.

SEE ALSO

accept(2), getpeername(2), getsockname(2), select(2), socket(2), sysctl(3), sysctl(8)

HISTORY

The **connect**() system call appeared in 4.2BSD.

NAME

connectat - initiate a connection on a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

#include <fcntl.h>

int
connectat(*int fd*, *int s*, *const struct sockaddr *name*, *socklen_t namelen*);

DESCRIPTION

The **connectat**() system call initiates a connection on a socket. When passed the special value AT_FDCWD in the *fd* parameter, the behavior is identical to a call to connect(2). Otherwise, **connectat**() works like the connect(2) system call with two exceptions:

1. It is limited to sockets in the PF_LOCAL domain.

2. If the file path stored in the *sun_path* field of the sockaddr_un structure is a relative path, it is located relative to the directory associated with the file descriptor *fd*.

RETURN VALUES

The **connectat**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **connectat**() system call may fail with the same errors as the connect(2) system call or with the following errors:

[EBADF] The *sun_path* field does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor.

[ENOTDIR] The *sun_path* field is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

bindat(2), connect(2), socket(2), unix(4)

AUTHORS

The **connectat** was developed by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation.

NAME

cpuset_getaffinity, **cpuset_setaffinity** - manage CPU affinity

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/cpuset.h>

int
cpuset_getaffinity(*cpulevel_t level*, *cpuwhich_t which*, *id_t id*, *size_t setsize*, *cpuset_t *mask*);

int
cpuset_setaffinity(*cpulevel_t level*, *cpuwhich_t which*, *id_t id*, *size_t setsize*, *const cpuset_t *mask*);

DESCRIPTION

cpuset_getaffinity() and **cpuset_setaffinity**() allow the manipulation of sets of CPUs available to processes, threads, interrupts, jails and other resources. These functions may manipulate sets of CPUs that contain many processes or per-object anonymous masks that effect only a single object.

The valid values for the *level* and *which* arguments are documented in cpuset(2). These arguments specify which object and which set of the object we are referring to. Not all possible combinations are valid. For example, only processes may belong to a numbered set accessed by a *level* argument of CPU_LEVEL_CPUSET. All resources, however, have a mask which may be manipulated with CPU_LEVEL_WHICH.

Masks of type *cpuset_t* are composed using the CPU_SET macros. The kernel tolerates large sets as long as all CPUs specified in the set exist. Sets smaller than the kernel uses generate an error on calls to **cpuset_getaffinity**() even if the result set would fit within the user supplied set. Calls to **cpuset_setaffinity**() tolerate small sets with no restrictions.

The supplied mask should have a size of *setsize* bytes. This size is usually provided by calling sizeof(mask) which is ultimately determined by the value of CPU_SETSIZE as defined in *<sys/cpuset.h>*.

cpuset_getaffinity() retrieves the mask from the object specified by *level*, *which* and *id* and stores it in the space provided by *mask*.

cpuset_setaffinity() attempts to set the mask for the object specified by *level*, *which* and *id* to the value

in *mask*.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following error codes may be set in *errno*:

[EINVAL] The *level* or *which* argument was not a valid value.

[EINVAL] The *mask* argument specified when calling **cpuset_setaffinity**() was not a valid value.

[EDEADLK] The **cpuset_setaffinity**() call would leave a thread without a valid CPU to run on because the set does not overlap with the thread's anonymous mask.

[EFAULT] The mask pointer passed was invalid.

[ESRCH] The object specified by the *id* and *which* arguments could not be found.

[ERANGE] The *cpusetsize* was either preposterously large or smaller than the kernel set size.

[EPERM] The calling process did not have the credentials required to complete the operation.

[ECAPMODE] The calling process attempted to act on a process other than itself, while in capability mode. See capsicum(4).

SEE ALSO

capsicum(4), cpuset(1), cpuset(2), cpuset_getid(2), cpuset_setid(2), cpuset_getdomain(2), cpuset_setdomain(2), pthread_affinity_np(3), pthread_attr_affinity_np(3), cpuset(9)

HISTORY

The **cpuset_getaffinity** family of system calls first appeared in FreeBSD 7.1.

AUTHORS

Jeffrey Roberson <*jeff@FreeBSD.org*>

NAME

cpuset_getdomain, **cpuset_setdomain** - manage memory domain policy

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/domainset.h>

int
cpuset_getdomain(*cpulevel_t level*, *cpuwhich_t which*, *id_t id*, *size_t setsize*, *domainet_t *mask*, *int *policy*);

int
cpuset_setdomain(*cpulevel_t level*, *cpuwhich_t which*, *id_t id*, *size_t setsize*, *const domainset_t *mask*, *int policy*);

DESCRIPTION

cpuset_getdomain() and **cpuset_setdomain**() allow the manipulation of sets of memory domains and allocation policy available to processes, threads, jails and other resources. These functions may manipulate sets of memory domains that contain many processes or per-object anonymous masks that effect only a single object.

The valid values for the *level* and *which* arguments are documented in cpuset(2). These arguments specify which object and which set of the object we are referring to. Not all possible combinations are valid. For example, only processes may belong to a numbered set accessed by a *level* argument of CPU_LEVEL_CPUSET. All resources, however, have a mask which may be manipulated with CPU_LEVEL_WHICH.

Masks of type *domainset_t* are composed using the DOMAINSET macros. The kernel tolerates large sets as long as all domains specified in the set exist. Sets smaller than the kernel uses generate an error on calls to **cpuset_getdomain**() even if the result set would fit within the user supplied set. Calls to **cpuset_setdomain**() tolerate small sets with no restrictions.

The supplied mask should have a size of *setsize* bytes. This size is usually provided by calling sizeof(mask) which is ultimately determined by the value of DOMAINSET_SETSIZE as defined in <*sys/domainset.h*>.

cpuset_getdomain() retrieves the mask and policy from the object specified by *level*, *which* and *id* and

stores it in the space provided by *mask and policy*.

cpuset_setdomain() attempts to set the mask and policy for the object specified by *level*, *which* and *id* to the values in *mask and policy*.

ALLOCATION POLICIES

Valid policy values are as follows:

DOMAINSET_POLICY_ROUNDROBIN
: Memory is allocated on a round-robin basis by cycling through each domain in *mask*.

DOMAINSET_POLICY_FIRSTTOUCH
: Memory is allocated on the domain local to the CPU the requesting thread is running on. Failure to allocate from this domain will fallback to round-robin.

DOMAINSET_POLICY_PREFER
: Memory is allocated preferentially from the single domain specified in the mask. If memory is unavailable the domains listed in the parent cpuset will be visited in a round-robin order.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following error codes may be set in *errno*:

[EINVAL] The *level* or *which* argument was not a valid value.

[EINVAL] The *mask or policy* argument specified when calling **cpuset_setdomain**() was not a valid value.

[EDEADLK] The **cpuset_setdomain**() call would leave a thread without a valid CPU to run on because the set does not overlap with the thread's anonymous mask.

[EFAULT] The mask pointer passed was invalid.

[ESRCH] The object specified by the *id* and *which* arguments could not be found.

[ERANGE] The *domainsetsize* was either preposterously large or smaller than the kernel set size.

[EPERM] The calling process did not have the credentials required to complete the operation.

[ECAPMODE] The calling process attempted to act on a process other than itself, while in capability mode. See capsicum(4).

SEE ALSO

capsicum(4), cpuset(1), cpuset(2), cpuset_getid(2), cpuset_setid(2), cpuset_getaffinity(2), cpuset_setaffinity(2), cpuset(9)

HISTORY

The **cpuset_getdomain** family of system calls first appeared in FreeBSD 12.0.

AUTHORS

Jeffrey Roberson <*jeff@FreeBSD.org*>

NAME

cpuset, **cpuset_getid**, **cpuset_setid** - manage CPU affinity sets

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/cpuset.h>

int
cpuset(*cpusetid_t *setid*);

int
cpuset_setid(*cpuwhich_t which*, *id_t id*, *cpusetid_t setid*);

int
cpuset_getid(*cpulevel_t level*, *cpuwhich_t which*, *id_t id*, *cpusetid_t *setid*);

DESCRIPTION

The **cpuset** family of system calls allow applications to control sets of processors and memory domains and assign processes and threads to these sets. Processor sets contain lists of CPUs and domains that members may run on and exist only as long as some process is a member of the set. All processes in the system have an assigned set. The default set for all processes in the system is the set numbered 1. Threads belong to the same set as the process which contains them, however, they may further restrict their set with the anonymous per-thread mask to bind to a specific CPU or subset of CPUs and memory domains.

Sets are referenced by a number of type *cpuset_id_t*. Each thread has a root set, an assigned set, and an anonymous mask. Only the root and assigned sets are numbered. The root set is the set of all CPUs and memory domains available in the system or in the system partition the thread is running in. The assigned set is a subset of the root set and is administratively assignable on a per-process basis. Many processes and threads may be members of a numbered set.

The anonymous set is a further thread-specific refinement on the assigned set. It is intended that administrators will manipulate numbered sets using cpuset(1) while application developers will manipulate anonymous sets using cpuset_setaffinity(2) and cpuset_setdomain(2).

To select the correct set a value of type *cpulevel_t* is used. The following values for *level* are supported:

CPU_LEVEL_ROOT	Root set
CPU_LEVEL_CPUSET	Assigned set
CPU_LEVEL_WHICH	Set specified by which argument

The *which* argument determines how the value of *id* is interpreted and is of type *cpuwhich_t*. The *which* argument may have the following values:

CPU_WHICH_TID	id is lwpid_t (thread id)
CPU_WHICH_PID	id is pid_t (process id)
CPU_WHICH_JAIL	id is jid (jail id)
CPU_WHICH_CPUSET	id is a cpusetid_t (cpuset id)
CPU_WHICH_IRQ	id is an irq number
CPU_WHICH_INTRHANDLER	id is an irq number for an interrupt handler
CPU_WHICH_ITHREAD	id is an irq number for an ithread
CPU_WHICH_DOMAIN	id is a NUMA domain

An *id* of '-1' may be used with a *which* of CPU_WHICH_TID, CPU_WHICH_PID, or CPU_WHICH_CPUSET to mean the current thread, process, or current thread's cpuset. All cpuset syscalls allow this usage.

A *level* argument of CPU_LEVEL_WHICH combined with a *which* argument other than CPU_WHICH_CPUSET refers to the anonymous mask of the object. This mask does not have an id and may only be manipulated with cpuset_setaffinity(2).

cpuset() creates a new set containing the same CPUs as the root set of the current process and stores its id in the space provided by *setid*. On successful completion the calling process joins the set and is the only member. Children inherit this set after a call to fork(2).

cpuset_setid() attempts to set the id of the object specified by the *which* argument. Currently CPU_WHICH_PID is the only acceptable value for which as threads do not have an id distinct from their process and the API does not permit changing the id of an existing set. Upon successful completion all of the threads in the target process will be running on CPUs permitted by the set.

cpuset_getid() retrieves a set id from the object indicated by *which* and stores it in the space pointed to by *setid*. The retrieved id may be that of either the root or assigned set depending on the value of *level*. *level* should be CPU_LEVEL_CPUSET or CPU_LEVEL_ROOT to get the set id from the process or thread specified by the *id* argument. Specifying CPU_LEVEL_WHICH with a process or thread is unsupported since this references the unnumbered anonymous mask.

The actual contents of the sets may be retrieved or manipulated using cpuset_getaffinity(2), cpuset_setaffinity(2), cpuset_getdomain(2), and cpuset_setdomain(2). See those manual pages for more detail.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following error codes may be set in *errno*:

[EINVAL]	The *which* or *level* argument was not a valid value.
[EDEADLK]	The **cpuset_setid**() call would leave a thread without a valid CPU to run on because the set does not overlap with the thread's anonymous mask.
[EFAULT]	The setid pointer passed to **cpuset_getid**() or **cpuset**() was invalid.
[ESRCH]	The object specified by the *id* and *which* arguments could not be found.
[EPERM]	The calling process did not have the credentials required to complete the operation.
[ENFILE]	There was no free *cpusetid_t* for allocation.

SEE ALSO

cpuset(1), cpuset_getaffinity(2), cpuset_setaffinity(2), cpuset_getdomain(2), cpuset_setdomain(2), pthread_affinity_np(3), pthread_attr_affinity_np(3), cpuset(9)

HISTORY

The **cpuset** family of system calls first appeared in FreeBSD 7.1.

AUTHORS

Jeffrey Roberson <*jeff@FreeBSD.org*>

NAME

creat - create a new file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>

int
creat(*const char *path*, *mode_t mode*);

DESCRIPTION

This interface is made obsolete by: open(2).

The **creat**() function is the same as:

```
open(path, O_CREAT | O_TRUNC | O_WRONLY, mode);
```

SEE ALSO

open(2)

HISTORY

The **creat**() function appeared in Version 6 AT&T UNIX.

NAME

CREATE_SERVICE - casper service declaration macro

LIBRARY

Casper Library (libcasper, -lcasper)

SYNOPSIS

#include <sys/nv.h>
#include <libcasper.h>
#include <libcasper_service.h>

typedef int service_limit_func_t(const nvlist_t *, const nvlist_t *);

typedef int service_command_func_t(const char *, const nvlist_t *, nvlist_t *,
nvlist_t *);

CREATE_SERVICE(*name*, *limit_func*, *command_func*, *flags*);

DESCRIPTION

The **CREATE_SERVICE** macro to create a new Casper service. The *name* is a string containing the service name, which will be used in the cap_service_open(3,) function to identify it.

The *limit_func* is a function of type service_limit_func_t. The first argument of the function contains nvlist(9), old service limits and second one the new limits. If the services wasn't limited the old limits will be set to NULL. This function should not allow to extend service limits and only limit it further. The *command_func* is a function of type service_command_func_t. First argument is the name of the command that should be executed. The first nvlist(9) contains the current limits. Next one contains a nvlist(9) with current request. The last one contains an output nvlist(9) which contains the response from Casper.

The *flags* argument defines limits of the service. The supported flags are:

CASPER_SERVICE_STDIO
The Casper service has access to the stdio descriptors from the process it was spawned from.

CASPER_SERVICE_FD
The Casper service has access to all descriptors besides stdio descriptors from the process it was spawned from.

CASPER_SERVICE_NO_UNIQ_LIMITS

The whole Casper communication is using nvlist(9) with NVLIST_NO_UNIQ(9) flag.

SEE ALSO

cap_enter(2), libcasper(3), capsicum(4), nv(9)

AUTHORS

The **libcasper** library was implemented by Pawel Jakub Dawidek <*pawel@dawidek.net*> under sponsorship from the FreeBSD Foundation. The **libcasper** new architecture was implemented by Mariusz Zaborski <*oshogbo@FreeBSD.org*>

NAME

crypt - Trapdoor encryption

LIBRARY

Crypt Library (libcrypt, -lcrypt)

SYNOPSIS

#include <unistd.h>

*char **
crypt(*const char *key*, *const char *salt*);

*char **
crypt_r(*const char *key*, *const char *salt*, *struct crypt_data *data*);

*const char **
crypt_get_format(*void*);

int
crypt_set_format(*const char *string*);

DESCRIPTION

The **crypt**() function performs password hashing with additional code added to deter key search attempts. Different algorithms can be used to in the hash. Currently these include the NBS Data Encryption Standard (DES), MD5 hash, NT-Hash (compatible with Microsoft's NT scheme) and Blowfish. The algorithm used will depend upon the format of the Salt (following the Modular Crypt Format (MCF)), if DES and/or Blowfish is installed or not, and whether **crypt_set_format**() has been called to change the default.

The first argument to **crypt** is the data to hash (usually a password), in a NUL-terminated string. The second is the salt, in one of three forms:

Extended	If it begins with an underscore ("_") then the DES Extended Format is used in interpreting both the key and the salt, as outlined below.
Modular	If it begins with the string "$digit$" then the Modular Crypt Format is used, as outlined below.
Traditional	If neither of the above is true, it assumes the Traditional Format, using the entire string as the salt (or the first portion).

All routines are designed to be time-consuming.

DES Extended Format:

The *key* is divided into groups of 8 characters (the last group is NUL-padded) and the low-order 7 bits of each character (56 bits per group) are used to form the DES key as follows: the first group of 56 bits becomes the initial DES key. For each additional group, the XOR of the encryption of the current DES key with itself and the group bits becomes the next DES key.

The salt is a 9-character array consisting of an underscore followed by 4 bytes of iteration count and 4 bytes of salt. These are encoded as printable characters, 6 bits per character, least significant character first. The values 0 to 63 are encoded as "./0-9A-Za-z". This allows 24 bits for both *count* and *salt*.

The *salt* introduces disorder in the DES algorithm in one of 16777216 or 4096 possible ways (i.e., with 24 or 12 bits: if bit *i* of the *salt* is set, then bits *i* and *i+24* are swapped in the DES E-box output).

The DES key is used to encrypt a 64-bit constant using *count* iterations of DES. The value returned is a NUL-terminated string, 20 or 13 bytes (plus NUL) in length, consisting of the *salt* followed by the encoded 64-bit encryption.

Modular crypt:

If the salt begins with the string *$digit$* then the Modular Crypt Format is used. The *digit* represents which algorithm is used in encryption. Following the token is the actual salt to use in the encryption. The maximum length of the salt used depends upon the module. The salt must be terminated with the end of the string character (NUL) or a dollar sign. Any characters after the dollar sign are ignored.

Currently supported algorithms are:

1. MD5
2. Blowfish
3. NT-Hash
4. (unused)
5. SHA-256
6. SHA-512

Other crypt formats may be easily added. An example salt would be:

4thesalt$rest

Traditional crypt:

The algorithm used will depend upon whether **crypt_set_format**() has been called and whether a global default format has been specified. Unless a global default has been specified or **crypt_set_format**() has set the format to something else, the built-in default format is used. This is currently DES if it is

available, or MD5 if not.

How the salt is used will depend upon the algorithm for the hash. For best results, specify at least eight characters of salt.

The **crypt_get_format**() function returns a constant string that represents the name of the algorithm currently used. Valid values are ‘des’, ‘blf’, ‘md5’, ‘sha256’, ‘sha512’ and ‘nth’.

The **crypt_set_format**() function sets the default encoding format according to the supplied *string*.

The **crypt_r**() function behaves identically to **crypt**(), except that the resulting string is stored in *data*, making it thread-safe.

RETURN VALUES

The **crypt**() and **crypt_r**() functions return a pointer to the encrypted value on success, and NULL on failure. Note: this is not a standard behaviour, AT&T **crypt**() will always return a pointer to a string.

The **crypt_set_format**() function will return 1 if the supplied encoding format was valid. Otherwise, a value of 0 is returned.

SEE ALSO

login(1), passwd(1), getpass(3), passwd(5)

HISTORY

A rotor-based **crypt**() function appeared in Version 6 AT&T UNIX. The current style **crypt**() first appeared in Version 7 AT&T UNIX.

The DES section of the code (FreeSec 1.0) was developed outside the United States of America as an unencumbered replacement for the U.S.-only NetBSD libcrypt encryption library.

The **crypt_r**() function was added in FreeBSD 12.0.

AUTHORS

Originally written by David Burren <*davidb@werj.com.au*>, later additions and changes by Poul-Henning Kamp, Mark R V Murray, Michael Bretterklieber, Kris Kennaway, Brian Feldman, Paul Herman and Niels Provos.

BUGS

The **crypt**() function returns a pointer to static data, and subsequent calls to **crypt**() will modify the same data. Likewise, **crypt_set_format**() modifies static data.

The NT-hash scheme does not use a salt, and is not hard for a competent attacker to break. Its use is not recommended.

NAME

ctermid - generate terminal pathname

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*char **
ctermid(*char *buf*);

*char **
ctermid_r(*char *buf*);

DESCRIPTION

The **ctermid**() function generates a string, that, when used as a pathname, refers to the current controlling terminal of the calling process.

If *buf* is the NULL pointer, a pointer to a static area is returned. Otherwise, the pathname is copied into the memory referenced by *buf*. The argument *buf* is assumed to be at least L_ctermid (as defined in the include file <*stdio.h*>) bytes long.

The **ctermid_r**() function provides the same functionality as **ctermid**() except that if *buf* is a NULL pointer, NULL is returned.

If no suitable lookup of the controlling terminal name can be performed, this implementation returns '/dev/tty'.

RETURN VALUES

Upon successful completion, a non-NULL pointer is returned. Otherwise, a NULL pointer is returned and the global variable *errno* is set to indicate the error.

ERRORS

The current implementation detects no error conditions.

SEE ALSO

ttyname(3)

STANDARDS

The **ctermid**() function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

BUGS

By default the **ctermid**() function writes all information to an internal static object. Subsequent calls to **ctermid**() will modify the same object.

NAME

cuserid - get user name associated with effective UID

LIBRARY

Compatibility Library (libcompat, -lcompat)

SYNOPSIS

#include <stdio.h>

*char **
cuserid(*char *s*);

DESCRIPTION

The cuserid() function is made obsolete by getpwuid(3).

The function **cuserid**() gets the user name associated with the effective UID of the current process. If the argument *s* is non-NULL, the name is copied to the buffer it points to, and that address is being returned. This buffer must provide space for at least *L_cuserid* characters. The L_cuserid constant is defined in <*stdio.h*>.

If *s* is NULL, an internal array is used and its address will be returned.

RETURN VALUES

The **cuserid**() function returns the address of an array in which the name has been stored.

If the name associated with the effective UID of the current process could not be found, either a null pointer will be returned, or (if *s* is non-NULL) the buffer *s* will be filled with a null string.

SEE ALSO

geteuid(2), getpwuid(3)

NAME

daemon - run in the background

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
daemon(*int nochdir*, *int noclose*);

int
daemonfd(*int chdirfd*, *int nullfd*);

DESCRIPTION

The **daemon**() function is for programs wishing to detach themselves from the controlling terminal and run in the background as system daemons.

Unless the argument *nochdir* is non-zero, **daemon**() changes the current working directory to the root (/).

Unless the argument *noclose* is non-zero, **daemon**() will redirect standard input, standard output, and standard error to */dev/null*.

The **daemonfd**() function is equivalent to the **daemon**() function except that arguments are the descriptors for the current working directory and to the descriptor to */dev/null*.

If *chdirfd* is equal to (-1) the current working directory is not changed.

If *nullfd* is equals to (-1) the redirection of standard input, standard output, and standard error is not closed.

RETURN VALUES

The **daemon**() and **daemonfd**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **daemon**() and **daemonfd**() function may fail and set *errno* for any of the errors specified for the library functions fork(2) open(2,) and setsid(2).

SEE ALSO

fork(2), setsid(2), sigaction(2)

HISTORY

The **daemon**() function first appeared in 4.4BSD. The **daemonfd**() function first appeared in FreeBSD 12.0.

CAVEATS

Unless the *noclose* argument is non-zero, **daemon**() will close the first three file descriptors and redirect them to */dev/null*. Normally, these correspond to standard input, standard output, and standard error. However, if any of those file descriptors refer to something else, they will still be closed, resulting in incorrect behavior of the calling program. This can happen if any of standard input, standard output, or standard error have been closed before the program was run. Programs using **daemon**() should therefore either call **daemon**() before opening any files or sockets, or verify that any file descriptors obtained have values greater than 2.

The **daemon**() function temporarily ignores SIGHUP while calling setsid(2) to prevent a parent session group leader's calls to fork(2) and then _exit(2) from prematurely terminating the child process.

NAME

des_crypt, **ecb_crypt**, **cbc_crypt**, **des_setparity** - fast DES encryption

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/des_crypt.h>

int
ecb_crypt(*char *key*, *char *data*, *unsigned datalen*, *unsigned mode*);

int
cbc_crypt(*char *key*, *char *data*, *unsigned datalen*, *unsigned mode*, *char *ivec*);

void
des_setparity(*char *key*);

DESCRIPTION

The **ecb_crypt**() and **cbc_crypt**() functions implement the NBS DES (Data Encryption Standard). These routines are faster and more general purpose than crypt(3). They also are able to utilize DES hardware if it is available. The **ecb_crypt**() function encrypts in ECB (Electronic Code Book) mode, which encrypts blocks of data independently. The **cbc_crypt**() function encrypts in CBC (Cipher Block Chaining) mode, which chains together successive blocks. CBC mode protects against insertions, deletions and substitutions of blocks. Also, regularities in the clear text will not appear in the cipher text.

Here is how to use these routines. The first argument, *key*, is the 8-byte encryption key with parity. To set the key's parity, which for DES is in the low bit of each byte, use **des_setparity**(). The second argument, *data*, contains the data to be encrypted or decrypted. The third argument, *datalen*, is the length in bytes of *data*, which must be a multiple of 8. The fourth argument, *mode*, is formed by *OR*'ing together some things. For the encryption direction *OR* in either DES_ENCRYPT or DES_DECRYPT. For software versus hardware encryption, *OR* in either DES_HW or DES_SW. If DES_HW is specified, and there is no hardware, then the encryption is performed in software and the routine returns DESERR_NOHWDEVICE. For **cbc_crypt**(), the *ivec* argument is the 8-byte initialization vector for the chaining. It is updated to the next initialization vector upon return.

ERRORS

[DESERR_NONE]	No error.
[DESERR_NOHWDEVICE]	Encryption succeeded, but done in software instead of the requested hardware.

[DESERR_HWERROR] An error occurred in the hardware or driver.
[DESERR_BADPARAM] Bad argument to routine.

Given a result status *stat*, the macro **DES_FAILED**(*stat*) is false only for the first two statuses.

SEE ALSO

crypt(3)

RESTRICTIONS

These routines are not available in RPCSRC 4.0. This information is provided to describe the DES interface expected by Secure RPC.

NAME

devname - get device name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>
#include <stdlib.h>

*char **
devname(*dev_t dev*, *mode_t type*);

*char **
devname_r(*dev_t dev*, *mode_t type*, *char *buf*, *int len*);

*char **
fdevname(*int fd*);

*char **
fdevname_r(*int fd*, *char *buf*, *int len*);

DESCRIPTION

The **devname**() function returns a pointer to the name of the block or character device in */dev* with a device number of *dev*, and a file type matching the one encoded in *type* which must be one of S_IFBLK or S_IFCHR. To find the right name, **devname**() asks the kernel via the *kern.devname* sysctl. If it is unable to come up with a suitable name, it will format the information encapsulated in *dev* and *type* in a human-readable format.

The **fdevname**() and **fdevname_r**() function obtains the device name directly from a file descriptor pointing to a character device. If it is unable to come up with a suitable name, these functions will return a NULL pointer.

devname() and **fdevname**() return the name stored in a static buffer which will be overwritten on subsequent calls. **devname_r**() and **fdevname_r**() take a buffer and length as argument to avoid this problem.

EXAMPLES

```
int fd;
struct stat buf;
```

```
char *name;

        fd = open("/dev/tun");
        fstat(fd, &buf);
        printf("devname is /dev/%s\n", devname(buf.st_rdev, S_IFCHR));
        printf("fdevname is /dev/%s\n", fdevname(fd));
```

SEE ALSO

stat(2)

HISTORY

The **devname**() function appeared in 4.4BSD. The **fdevname**() function appeared in FreeBSD 8.0.

NAME

digittoint_l, **isalnum_l**, **isalpha_l**, **isascii_l**, **isblank_l**, **iscntrl_l**, **isdigit_l**, **isgraph_l**, **ishexnumber_l**, **isideogram_l**, **islower_l**, **isnumber_l**, **isphonogram_l**, **isprint_l**, **ispunct_l**, **isrune_l**, **isspace_l**, **isspecial_l**, **isupper_l**, **isxdigit_l**, **tolower_l**, **toupper_l** - character classification functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <ctype.h>

int
digittoint_l(*int c*, *locale_t loc*);

int
isalnum_l(*int c*, *locale_t loc*);

int
isalpha_l(*int c*, *locale_t loc*);

int
isascii_l(*int c*, *locale_t loc*);

int
iscntrl_l(*int c*, *locale_t loc*);

int
isdigit_l(*int c*, *locale_t loc*);

int
isgraph_l(*int c*, *locale_t loc*);

int
ishexnumber_l(*int c*, *locale_t loc*);

int
isideogram_l(*int c*, *locale_t loc*);

int
islower_l(*int c*, *locale_t loc*);

int
isnumber_l(*int c*, *locale_t loc*);

int
isphonogram_l(*int c*, *locale_t loc*);

int
isspecial_l(*int c*, *locale_t loc*);

int
isprint_l(*int c*, *locale_t loc*);

int
ispunct_l(*int c*, *locale_t loc*);

int
isrune_l(*int c*, *locale_t loc*);

int
isspace_l(*int c*, *locale_t loc*);

int
isupper_l(*int c*, *locale_t loc*);

int
isxdigit_l(*int c*, *locale_t loc*);

int
tolower_l(*int c*, *locale_t loc*);

int
toupper_l(*int c*, *locale_t loc*);

DESCRIPTION

The above functions perform character tests and conversions on the integer *c* in the locale *loc*. They behave in the same way as the versions without the _l suffix, but use the specified locale rather than the global or per-thread locale. <*ctype.h*>, or as true functions in the C library. See the specific manual pages for more information.

SEE ALSO

digittoint(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), isideogram(3), islower(3), isphonogram(3), isprint(3), ispunct(3), isrune(3), isspace(3), isspecial(3), isupper(3), isxdigit(3), tolower(3), toupper(3), wctype(3), xlocale(3)

STANDARDS

These functions conform to IEEE Std 1003.1-2008 ("POSIX.1"), except for **digittoint_l**(), **isascii_l**(), **ishexnumber_l**(), **isideogram_l**(), **isnumber_l**(), **isphonogram_l**(), **isrune_l**() and **isspecial_l**() which are FreeBSD extensions.

NAME

digittoint, **isalnum**, **isalpha**, **isascii**, **isblank**, **iscntrl**, **isdigit**, **isgraph**, **ishexnumber**, **isideogram**, **islower**, **isnumber**, **isphonogram**, **isprint**, **ispunct**, **isrune**, **isspace**, **isspecial**, **isupper**, **isxdigit**, **toascii**, **tolower**, **toupper** - character classification functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <ctype.h>

int
digittoint(*int c*);

int
isalnum(*int c*);

int
isalpha(*int c*);

int
isascii(*int c*);

int
iscntrl(*int c*);

int
isdigit(*int c*);

int
isgraph(*int c*);

int
ishexnumber(*int c*);

int
isideogram(*int c*);

int
islower(*int c*);

int
isnumber(*int c*);

int
isphonogram(*int c*);

int
isspecial(*int c*);

int
isprint(*int c*);

int
ispunct(*int c*);

int
isrune(*int c*);

int
isspace(*int c*);

int
isupper(*int c*);

int
isxdigit(*int c*);

int
toascii(*int c*);

int
tolower(*int c*);

int
toupper(*int c*);

DESCRIPTION

The above functions perform character tests and conversions on the integer *c*. They are available as macros, defined in the include file <*ctype.h*>, or as true functions in the C library. See the specific manual pages for more information.

SEE ALSO

digittoint(3), isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), isideogram(3), islower(3), isphonogram(3), isprint(3), ispunct(3), isrune(3), isspace(3), isspecial(3), isupper(3), isxdigit(3), toascii(3), tolower(3), toupper(3), wctype(3), ascii(7)

STANDARDS

These functions, except for **digittoint**(), **isascii**(), **ishexnumber**(), **isideogram**(), **isnumber**(), **isphonogram**(), **isrune**(), **isspecial**() and **toascii**(), conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

div - return quotient and remainder from division

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

div_t
div(*int num*, *int denom*);

DESCRIPTION

The **div**() function computes the value *num/denom* and returns the quotient and remainder in a structure named *div_t* that contains two *int* members named *quot* and *rem*.

SEE ALSO

imaxdiv(3), ldiv(3), lldiv(3)

STANDARDS

The **div**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

dladdr - find the shared object containing a given address

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <dlfcn.h>

int
dladdr(*const void *addr*, *Dl_info *info*);

DESCRIPTION

The **dladdr**() function queries the dynamic linker for information about the shared object containing the address *addr*. The information is returned in the structure specified by *info*. The structure contains at least the following members:

const char *dli_fname	The pathname of the shared object containing the address.
void *dli_fbase	The base address at which the shared object is mapped into the address space of the calling process.
const char *dli_sname	The name of the nearest run-time symbol with a value less than or equal to *addr*. When possible, the symbol name is returned as it would appear in C source code. If no symbol with a suitable value is found, both this field and *dli_saddr* are set to NULL.
void *dli_saddr	The value of the symbol returned in dli_sname.

The **dladdr**() function is available only in dynamically linked programs.

ERRORS

If a mapped shared object containing *addr* cannot be found, **dladdr**() returns 0. In that case, a message detailing the failure can be retrieved by calling **dlerror**().

On success, a non-zero value is returned.

SEE ALSO

rtld(1), dlopen(3)

HISTORY

The **dladdr**() function first appeared in the Solaris operating system.

BUGS

This implementation is bug-compatible with the Solaris implementation. In particular, the following bugs are present:

- If *addr* lies in the main executable rather than in a shared library, the pathname returned in *dli_fname* may not be correct. The pathname is taken directly from *argv[0]* of the calling process. When executing a program specified by its full pathname, most shells set *argv[0]* to the pathname. But this is not required of shells or guaranteed by the operating system.

- If *addr* is of the form *&func*, where *func* is a global function, its value may be an unpleasant surprise. In dynamically linked programs, the address of a global function is considered to point to its program linkage table entry, rather than to the entry point of the function itself. This causes most global functions to appear to be defined within the main executable, rather than in the shared libraries where the actual code resides.

- Returning 0 as an indication of failure goes against long-standing Unix tradition.

NAME

dlopen, **fdlopen**, **dlsym**, **dlfunc**, **dlerror**, **dlclose** - programmatic interface to the dynamic linker

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <dlfcn.h>

*void **
dlopen(*const char *path*, *int mode*);

*void **
fdlopen(*int fd*, *int mode*);

*void **
dlsym(*void * restrict handle*, *const char * restrict symbol*);

dlfunc_t
dlfunc(*void * restrict handle*, *const char * restrict symbol*);

*char **
dlerror(*void*);

int
dlclose(*void *handle*);

DESCRIPTION

These functions provide a simple programmatic interface to the services of the dynamic linker. Operations are provided to add new shared objects to a program's address space, to obtain the address bindings of symbols defined by such objects, and to remove such objects when their use is no longer required.

The **dlopen**() function provides access to the shared object in *path*, returning a descriptor that can be used for later references to the object in calls to **dlsym**() and **dlclose**(). If *path* was not in the address space prior to the call to **dlopen**(), it is placed in the address space. When an object is first loaded into the address space in this way, its function **_init**(), if any, is called by the dynamic linker. If *path* has already been placed in the address space in a previous call to **dlopen**(), it is not added a second time, although a reference count of **dlopen**() operations on *path* is maintained. A null pointer supplied for *path* is interpreted as a reference to the main executable of the process. The *mode* argument controls the way

in which external function references from the loaded object are bound to their referents. It must contain one of the following values, possibly ORed with additional flags which will be described subsequently:

RTLD_LAZY Each external function reference is resolved when the function is first called.

RTLD_NOW All external function references are bound immediately by **dlopen**().

RTLD_LAZY is normally preferred, for reasons of efficiency. However, RTLD_NOW is useful to ensure that any undefined symbols are discovered during the call to **dlopen**().

One of the following flags may be ORed into the *mode* argument:

RTLD_GLOBAL Symbols from this shared object and its directed acyclic graph (DAG) of needed objects will be available for resolving undefined references from all other shared objects.

RTLD_LOCAL Symbols in this shared object and its DAG of needed objects will be available for resolving undefined references only from other objects in the same DAG. This is the default, but it may be specified explicitly with this flag.

RTLD_TRACE When set, causes dynamic linker to exit after loading all objects needed by this shared object and printing a summary which includes the absolute pathnames of all objects, to standard output. With this flag **dlopen**() will return to the caller only in the case of error.

RTLD_NODELETE Prevents unload of the loaded object on **dlclose**(). The same behaviour may be requested by **-z nodelete** option of the static linker ld(1).

RTLD_NOLOAD Only return valid handle for the object if it is already loaded in the process address space, otherwise NULL is returned. Other mode flags may be specified, which will be applied for promotion for the found object.

If **dlopen**() fails, it returns a null pointer, and sets an error condition which may be interrogated with **dlerror**().

The **fdlopen**() function is similar to **dlopen**(), but it takes the file descriptor argument *fd*, which is used for the file operations needed to load an object into the address space. The file descriptor *fd* is not closed by the function regardless a result of execution, but a duplicate of the file descriptor is. This may be important if a lockf(3) lock is held on the passed descriptor. The *fd* argument -1 is interpreted as a reference to the main executable of the process, similar to *NULL* value for the *name* argument to

dlopen(). The **fdlopen**() function can be used by the code that needs to perform additional checks on the loaded objects, to prevent races with symlinking or renames.

The **dlsym**() function returns the address binding of the symbol described in the null-terminated character string *symbol*, as it occurs in the shared object identified by *handle*. The symbols exported by objects added to the address space by **dlopen**() can be accessed only through calls to **dlsym**(). Such symbols do not supersede any definition of those symbols already present in the address space when the object is loaded, nor are they available to satisfy normal dynamic linking references.

If **dlsym**() is called with the special *handle* NULL, it is interpreted as a reference to the executable or shared object from which the call is being made. Thus a shared object can reference its own symbols.

If **dlsym**() is called with the special *handle* RTLD_DEFAULT, the search for the symbol follows the algorithm used for resolving undefined symbols when objects are loaded. The objects searched are as follows, in the given order:

1. The referencing object itself (or the object from which the call to **dlsym**() is made), if that object was linked using the **-Bsymbolic** option to ld(1).

2. All objects loaded at program start-up.

3. All objects loaded via **dlopen**() with the RTLD_GLOBAL flag set in the *mode* argument.

4. All objects loaded via **dlopen**() which are in needed-object DAGs that also contain the referencing object.

If **dlsym**() is called with the special *handle* RTLD_NEXT, then the search for the symbol is limited to the shared objects which were loaded after the one issuing the call to **dlsym**(). Thus, if the function is called from the main program, all the shared libraries are searched. If it is called from a shared library, all subsequent shared libraries are searched. RTLD_NEXT is useful for implementing wrappers around library functions. For example, a wrapper function **getpid**() could access the "real" **getpid**() with dlsym(RTLD_NEXT, "getpid"). (Actually, the **dlfunc**() interface, below, should be used, since **getpid**() is a function and not a data object.)

If **dlsym**() is called with the special *handle* RTLD_SELF, then the search for the symbol is limited to the shared object issuing the call to **dlsym**() and those shared objects which were loaded after it.

The **dlsym**() function returns a null pointer if the symbol cannot be found, and sets an error condition which may be queried with **dlerror**().

The **dlfunc**() function implements all of the behavior of **dlsym**(), but has a return type which can be cast to a function pointer without triggering compiler diagnostics. (The **dlsym**() function returns a data pointer; in the C standard, conversions between data and function pointer types are undefined. Some compilers and lint(1) utilities warn about such casts.) The precise return type of **dlfunc**() is unspecified; applications must cast it to an appropriate function pointer type.

The **dlerror**() function returns a null-terminated character string describing the last error that occurred during a call to **dlopen**(), **dladdr**(), **dlinfo**(), **dlsym**(), **dlfunc**(), or **dlclose**(). If no such error has occurred, **dlerror**() returns a null pointer. At each call to **dlerror**(), the error indication is reset. Thus in the case of two calls to **dlerror**(), where the second call follows the first immediately, the second call will always return a null pointer.

The **dlclose**() function deletes a reference to the shared object referenced by *handle*. If the reference count drops to 0, the object is removed from the address space, and *handle* is rendered invalid. Just before removing a shared object in this way, the dynamic linker calls the object's **_fini**() function, if such a function is defined by the object. If **dlclose**() is successful, it returns a value of 0. Otherwise it returns -1, and sets an error condition that can be interrogated with **dlerror**().

The object-intrinsic functions **_init**() and **_fini**() are called with no arguments, and are not expected to return values.

NOTES

ELF executables need to be linked using the **-export-dynamic** option to ld(1) for symbols defined in the executable to become visible to **dlsym**().

Other ELF platforms require linking with Dynamic Linker Services Filter (libdl, -ldl) to provide **dlopen**() and other functions. FreeBSD does not require linking with the library, but supports it for compatibility.

In previous implementations, it was necessary to prepend an underscore to all external symbols in order to gain symbol compatibility with object code compiled from the C language. This is still the case when using the (obsolete) **-aout** option to the C language compiler.

ERRORS

The **dlopen**(), **fdlopen**(), **dlsym**(), and **dlfunc**() functions return a null pointer in the event of errors. The **dlclose**() function returns 0 on success, or -1 if an error occurred. Whenever an error has been detected, a message detailing it can be retrieved via a call to **dlerror**().

SEE ALSO

ld(1), rtld(1), dladdr(3), dlinfo(3), link(5)

NAME

dlinfo - information about dynamically loaded object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <link.h>
#include <dlfcn.h>

int
dlinfo(*void * restrict handle*, *int request*, *void * restrict p*);

DESCRIPTION

The **dlinfo**() function provides information about dynamically loaded object. The action taken by **dlinfo**() and exact meaning and type of *p* argument depend on value of the *request* argument provided by caller.

The *handle* argument is either the value returned from the dlopen(3) function call or special handle RTLD_SELF. If *handle* is the value returned from dlopen(3), the information returned by the **dlinfo**() function pertains to the specified object. If handle is the special handle RTLD_SELF, the information returned pertains to the caller itself.

Possible values for the *request* argument are:

RTLD_DI_LINKMAP

Retrieve the *Link_map* (*struct link_map*) structure pointer for the specified *handle*. On successful return, the *p* argument is filled with the pointer to the *Link_map* structure (*Link_map **p*) describing a shared object specified by the *handle* argument. The *Link_map* structures are maintained as a doubly linked list by ld.so(1), in the same order as dlopen(3) and dlclose(3) are called. See *EXAMPLES*, example 1.

The *Link_map* structure is defined in <*link.h*> and has the following members:

```
caddr_t        l_addr;    /* Base Address of library */
const char     *l_name;   /* Absolute Path to Library */
const void     *l_ld;     /* Pointer to .dynamic in memory */
struct link_map *l_next,   /* linked list of mapped libs */
                *l_prev;
```

l_addr The base address of the object loaded into memory.

l_name The full name of the loaded shared object.

l_ld The address of the dynamic linking information segment (PT_DYNAMIC) loaded into memory.

l_next The next *Link_map* structure on the link-map list.

l_prev The previous *Link_map* structure on the link-map list.

RTLD_DI_SERINFO

Retrieve the library search paths associated with the given *handle* argument. The *p* argument should point to *Dl_serinfo* structure buffer (*Dl_serinfo *p*). The *Dl_serinfo* structure must be initialized first with the RTLD_DI_SERINFOSIZE request.

The returned *Dl_serinfo* structure contains *dls_cnt Dl_serpath* entries. Each entry's *dlp_name* field points to the search path. The corresponding *dlp_info* field contains one of more flags indicating the origin of the path (see the LA_SER_* flags defined in the <*link.h*> header file). See *EXAMPLES*, example 2, for a usage example.

RTLD_DI_SERINFOSIZE

Initialize a *Dl_serinfo* structure for use in a RTLD_DI_SERINFO request. Both the *dls_cnt* and *dls_size* fields are returned to indicate the number of search paths applicable to the handle, and the total size of a *Dl_serinfo* buffer required to hold *dls_cnt Dl_serpath* entries and the associated search path strings. See *EXAMPLES*, example 2, for a usage example.

RTLD_DI_ORIGIN

Retrieve the origin of the dynamic object associated with the handle. On successful return, *p* argument is filled with the *char* pointer (*char *p*).

RETURN VALUES

The **dlinfo**() function returns 0 on success, or -1 if an error occurred. Whenever an error has been detected, a message detailing it can be retrieved via a call to dlerror(3).

EXAMPLES

Example 1: Using **dlinfo**() to retrieve *Link_map* structure.

The following example shows how dynamic library can detect the list of shared libraries loaded after caller's one. For simplicity, error checking has been omitted.

```
Link_map *map;

dlinfo(RTLD_SELF, RTLD_DI_LINKMAP, &map);

while (map != NULL) {
        printf("%p: %s\n", map->l_addr, map->l_name);
        map = map->l_next;
}
```

Example 2: Using **dlinfo**() to retrieve the library search paths.

The following example shows how a dynamic object can inspect the library search paths that would be used to locate a simple filename with dlopen(3). For simplicity, error checking has been omitted.

```
Dl_serinfo      _info, *info = &_info;
Dl_serpath      *path;
unsigned int    cnt;

/* determine search path count and required buffer size */
dlinfo(RTLD_SELF, RTLD_DI_SERINFOSIZE, (void *)info);

/* allocate new buffer and initialize */
info = malloc(_info.dls_size);
info->dls_size = _info.dls_size;
info->dls_cnt = _info.dls_cnt;

/* obtain sarch path information */
dlinfo(RTLD_SELF, RTLD_DI_SERINFO, (void *)info);

path = &info->dls_serpath[0];

for (cnt = 1; cnt <= info->dls_cnt; cnt++, path++) {
        (void) printf("%2d: %s\n", cnt, path->dls_name);
}
```

SEE ALSO

rtld(1), dladdr(3), dlopen(3), dlsym(3)

HISTORY

The **dlinfo**() function first appeared in the Solaris operating system. In FreeBSD, it first appeared in

FreeBSD 4.8.

AUTHORS

The FreeBSD implementation of the **dlinfo**() function was originally written by Alexey Zelkin <*phantom@FreeBSD.org*> and later extended and improved by Alexander Kabaev <*kan@FreeBSD.org*>.

The manual page for this function was written by Alexey Zelkin <*phantom@FreeBSD.org*>.

NAME

dllockinit - register thread locking methods with the dynamic linker

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <dlfcn.h>

void
dllockinit(*void *context*, *void *(*lock_create)(void *context)*, *void (*rlock_acquire)(void *lock)*,
*void (*wlock_acquire)(void *lock)*, *void (*lock_release)(void *lock)*,
*void (*lock_destroy)(void *lock)*, *void (*context_destroy)(void *context)*);

DESCRIPTION

Due to enhancements in the dynamic linker, this interface is no longer needed. It is deprecated and will be removed from future releases. In current releases it still exists, but only as a stub which does nothing.

Threads packages can call **dllockinit**() at initialization time to register locking functions for the dynamic linker to use. This enables the dynamic linker to prevent multiple threads from entering its critical sections simultaneously.

The *context* argument specifies an opaque context for creating locks. The dynamic linker will pass it to the *lock_create* function when creating the locks it needs. When the dynamic linker is permanently finished using the locking functions (e.g., if the program makes a subsequent call to **dllockinit**() to register new locking functions) it will call *context_destroy* to destroy the context.

The *lock_create* argument specifies a function for creating a read/write lock. It must return a pointer to the new lock.

The *rlock_acquire* and *wlock_acquire* arguments specify functions which lock a lock for reading or writing, respectively. The *lock_release* argument specifies a function which unlocks a lock. Each of these functions is passed a pointer to the lock.

The *lock_destroy* argument specifies a function to destroy a lock. It may be NULL if locks do not need to be destroyed. The *context_destroy* argument specifies a function to destroy the context. It may be NULL if the context does not need to be destroyed.

Until **dllockinit**() is called, the dynamic linker protects its critical sections using a default locking mechanism which works by blocking the SIGVTALRM, SIGPROF, and SIGALRM signals. This is

sufficient for many application level threads packages, which typically use one of these signals to implement preemption. An application which has registered its own locking methods with **dllockinit**() can restore the default locking by calling **dllockinit**() with all arguments NULL.

SEE ALSO

rtld(1), signal(3)

HISTORY

The **dllockinit**() function first appeared in FreeBSD 4.0.

NAME

res_query, **res_search**, **res_mkquery**, **res_send**, **res_init**, **dn_comp**, **dn_expand**, **dn_skipname**, **ns_get16**, **ns_get32**, **ns_put16**, **ns_put32** - resolver routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int
res_query(*const char *dname*, *int class*, *int type*, *u_char *answer*, *int anslen*);

int
res_search(*const char *dname*, *int class*, *int type*, *u_char *answer*, *int anslen*);

int
res_mkquery(*int op*, *const char *dname*, *int class*, *int type*, *const u_char *data*, *int datalen*, *const u_char *newrr_in*, *u_char *buf*, *int buflen*);

int
res_send(*const u_char *msg*, *int msglen*, *u_char *answer*, *int anslen*);

int
res_init(*void*);

int
dn_comp(*const char *exp_dn*, *u_char *comp_dn*, *int length*, *u_char **dnptrs*, *u_char **lastdnptr*);

int
dn_expand(*const u_char *msg*, *const u_char *eomorig*, *const u_char *comp_dn*, *char *exp_dn*, *int length*);

int
dn_skipname(*const u_char *comp_dn*, *const u_char *eom*);

u_int

ns_get16(*const u_char *src*);

u_long
ns_get32(*const u_char *src*);

void
ns_put16(*u_int src*, *u_char *dst*);

void
ns_put32(*u_long src*, *u_char *dst*);

DESCRIPTION

These routines are used for making, sending and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information that is used by the resolver routines is kept in the structure *_res*. Most of the values have reasonable defaults and can be ignored. Options stored in *_res.options* are defined in <*resolv.h*> and are as follows. Options are stored as a simple bit mask containing the bitwise ‘‘or’’ of the options enabled.

RES_INIT — True if the initial name server address and default domain name are initialized (i.e., **res_init**() has been called).

RES_DEBUG — Print debugging messages.

RES_AAONLY — Accept authoritative answers only. With this option, **res_send**() should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC — Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN — Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC — Unused currently (ignore truncation errors, i.e., do not retry with TCP).

RES_RECURSE — Set the recursion-desired bit in queries. This is the default. (**res_send**() does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
: If set, **res_search**() will append the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

RES_DNSRCH
: If this option is set, **res_search**() will search for host names in the current domain and in parent domains; see hostname(7). This is used by the standard host lookup routine gethostbyname(3). This option is enabled by default.

RES_NOALIASES
: This option turns off the user level aliasing feature controlled by the "HOSTALIASES" environment variable. Network daemons should set this option.

RES_USE_INET6
: Enables support for IPv6-only applications. This causes IPv4 addresses to be returned as an IPv4 mapped address. For example, 10.1.1.1 will be returned as ::ffff:10.1.1.1. The option is meaningful with certain kernel configuration only.

RES_USE_EDNS0
: Enables support for OPT pseudo-RR for EDNS0 extension. With the option, resolver code will attach OPT pseudo-RR into DNS queries, to inform of our receive buffer size. The option will allow DNS servers to take advantage of non-default receive buffer size, and to send larger replies. DNS query packets with EDNS0 extension is not compatible with non-EDNS0 DNS servers.

The **res_init**() routine reads the configuration file (if any; see resolver(5)) to get the default domain name, search list and the Internet address of the local name server(s). If no server is configured, the host running the resolver is tried. The current domain name is defined by the hostname if not specified in the configuration file; it can be overridden by the environment variable LOCALDOMAIN. This environment variable may contain several blank-separated tokens if you wish to override the *search list* on a per-process basis. This is similar to the **search** command in the configuration file. Another environment variable "RES_OPTIONS" can be set to override certain internal resolver options which are otherwise set by changing fields in the *_res* structure or are inherited from the configuration file's **options** command. The syntax of the "RES_OPTIONS" environment variable is explained in resolver(5). Initialization normally occurs on the first call to one of the following routines.

The **res_query**() function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

The **res_search**() routine makes a query and awaits a response like **res_query**(), but in addition, it

implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. It returns the first successful reply.

The remaining routines are lower-level routines used by **res_query**(). The **res_mkquery**() function constructs a standard query message and places it in *buf*. It returns the size of the query, or -1 if the query is larger than *buflen*. The query type *op* is usually QUERY, but can be any of the query types defined in <*arpa/nameser.h*>. The domain name for the query is given by *dname*. The *newrr_in* argument is currently unused but is intended for making update messages.

The **res_send**() routine sends a pre-formatted query and returns an answer. It will call **res_init**() if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or -1 if there were errors.

The **dn_comp**() function compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or -1 if there were errors. The size of the array pointed to by *comp_dn* is given by *length*. The compression uses an array of pointers *dnptrs* to previously-compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. The limit to the array is specified by *lastdnptr*. A side effect of **dn_comp**() is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

The **dn_expand**() entry expands the compressed domain name *comp_dn* to a full domain name The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of compressed name is returned or -1 if there was an error.

The **dn_skipname**() function skips over a compressed domain name, which starts at a location pointed to by *comp_dn*. The compressed name is contained in a query or reply message; *eom* is a pointer to the end of the message. The size of compressed name is returned or -1 if there was an error.

The **ns_get16**() function gets a 16-bit quantity from a buffer pointed to by *src*.

The **ns_get32**() function gets a 32-bit quantity from a buffer pointed to by *src*.

The **ns_put16**() function puts a 16-bit quantity *src* to a buffer pointed to by *dst*.

The **ns_put32**() function puts a 32-bit quantity *src* to a buffer pointed to by *dst*.

IMPLEMENTATION NOTES

This implementation of the resolver is thread-safe, but it will not function properly if the programmer

attempts to declare his or her own *_res* structure in an attempt to replace the per-thread version referred to by that macro.

The following compile-time option can be specified to change the default behavior of resolver routines when necessary.

RES_ENFORCE_RFC1034 If this symbol is defined during compile-time, **res_search**() will enforce RFC 1034 check, namely, disallow using of underscore character within host names. This is used by the standard host lookup routines like gethostbyname(3). For compatibility reasons this option is not enabled by default.

RETURN VALUES

The **res_init**() function will return 0 on success, or -1 in a threaded program if per-thread storage could not be allocated.

The **res_mkquery**(), **res_search**(), and **res_query**() functions return the size of the response on success, or -1 if an error occurs. The integer *h_errno* may be checked to determine the reason for error. See gethostbyname(3) for more information.

FILES

/etc/resolv.conf The configuration file, see resolver(5).

SEE ALSO

gethostbyname(3), resolver(5), hostname(7), named(8)

RFC1032, RFC1033, RFC1034, RFC1035, RFC974

Name Server Operations Guide for BIND.

HISTORY

The **res_query** function appeared in 4.3BSD.

NAME

drand48, **erand48**, **lrand48**, **nrand48**, **mrand48**, **jrand48**, **srand48**, **seed48**, **lcong48** - pseudo random number generators and initialization routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

double
drand48(*void*);

double
erand48(*unsigned short xseed[3]*);

long
lrand48(*void*);

long
nrand48(*unsigned short xseed[3]*);

long
mrand48(*void*);

long
jrand48(*unsigned short xseed[3]*);

void
srand48(*long seed*);

*unsigned short **
seed48(*unsigned short xseed[3]*);

void
lcong48(*unsigned short p[7]*);

DESCRIPTION

The functions described in this manual page are not cryptographically secure. Cryptographic applications should use arc4random(3) instead.

The **rand48**() family of functions generates pseudo-random numbers using a linear congruential algorithm working on integers 48 bits in size. The particular formula employed is r(n+1) = (a * r(n) + c) mod m where the default values are for the multiplicand a = 0x5deece66d = 25214903917 and the addend c = 0xb = 11. The modulo is always fixed at m = 2 ** 48. r(n) is called the seed of the random number generator.

For all the six generator routines described next, the first computational step is to perform a single iteration of the algorithm.

The **drand48**() and **erand48**() functions return values of type double. The full 48 bits of r(n+1) are loaded into the mantissa of the returned value, with the exponent set such that the values produced lie in the interval [0.0, 1.0).

The **lrand48**() and **nrand48**() functions return values of type long in the range [0, 2**31-1]. The high-order (31) bits of r(n+1) are loaded into the lower bits of the returned value, with the topmost (sign) bit set to zero.

The **mrand48**() and **jrand48**() functions return values of type long in the range [-2**31, 2**31-1]. The high-order (32) bits of r(n+1) are loaded into the returned value.

The **drand48**(), **lrand48**(), and **mrand48**() functions use an internal buffer to store r(n). For these functions the initial value of r(0) = 0x1234abcd330e = 20017429951246.

On the other hand, **erand48**(), **nrand48**(), and **jrand48**() use a user-supplied buffer to store the seed r(n), which consists of an array of 3 shorts, where the zeroth member holds the least significant bits.

All functions share the same multiplicand and addend.

The **srand48**() function is used to initialize the internal buffer r(n) of **drand48**(), **lrand48**(), and **mrand48**() such that the 32 bits of the seed value are copied into the upper 32 bits of r(n), with the lower 16 bits of r(n) arbitrarily being set to 0x330e. Additionally, the constant multiplicand and addend of the algorithm are reset to the default values given above.

The **seed48**() function also initializes the internal buffer r(n) of **drand48**(), **lrand48**(), and **mrand48**(), but here all 48 bits of the seed can be specified in an array of 3 shorts, where the zeroth member specifies the lowest bits. Again, the constant multiplicand and addend of the algorithm are reset to the default values given above. The **seed48**() function returns a pointer to an array of 3 shorts which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to **seed48**().

Finally, **lcong48**() allows full control over the multiplicand and addend used in **drand48**(), **erand48**(),

lrand48(), **nrand48**(), **mrand48**(), and **jrand48**(), and the seed used in **drand48**(), **lrand48**(), and **mrand48**(). An array of 7 shorts is passed as argument; the first three shorts are used to initialize the seed; the second three are used to initialize the multiplicand; and the last short is used to initialize the addend. It is thus not possible to use values greater than 0xffff as the addend.

Note that all three methods of seeding the random number generator always also set the multiplicand and addend for any of the six generator calls.

SEE ALSO

arc4random(3), rand(3), random(3)

AUTHORS

Martin Birgmeier

NAME

dup, **dup2** - duplicate an existing file descriptor

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
dup(*int oldd*);

int
dup2(*int oldd*, *int newd*);

DESCRIPTION

The **dup**() system call duplicates an existing object descriptor and returns its value to the calling process (*newd* = **dup**(*oldd*)). The argument *oldd* is a small non-negative integer index in the per-process descriptor table. The new descriptor returned by the call is the lowest numbered descriptor currently not in use by the process.

The object referenced by the descriptor does not distinguish between *oldd* and *newd* in any way. Thus if *newd* and *oldd* are duplicate references to an open file, read(2), write(2) and lseek(2) calls all move a single pointer into the file, and append mode, non-blocking I/O and asynchronous I/O options are shared between the references. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional open(2) system call. The close-on-exec flag on the new file descriptor is unset.

In **dup2**(), the value of the new descriptor *newd* is specified. If this descriptor is already in use and *oldd* != *newd*, the descriptor is first deallocated as if the close(2) system call had been used. If *oldd* is not a valid descriptor, then *newd* is not closed. If *oldd* == *newd* and *oldd* is a valid descriptor, then **dup2**() is successful, and does nothing.

RETURN VALUES

These calls return the new file descriptor if successful; otherwise the value -1 is returned and the external variable *errno* is set to indicate the cause of the error.

ERRORS

The **dup**() system call fails if:

[EBADF] The *oldd* argument is not a valid active descriptor

[EMFILE] Too many descriptors are active.

The **dup2**() system call fails if:

[EBADF] The *oldd* argument is not a valid active descriptor or the *newd* argument is negative or exceeds the maximum allowable descriptor number

SEE ALSO

accept(2), close(2), fcntl(2), getdtablesize(2), open(2), pipe(2), socket(2), socketpair(2), dup3(3)

STANDARDS

The **dup**() and **dup2**() system calls are expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **dup**() function appeared in Version 3 AT&T UNIX. The **dup2**() function appeared in Version 7 AT&T UNIX.

NAME

dup3 - duplicate an existing file descriptor

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>
#include <unistd.h>

int
dup3(*int oldd*, *int newd*, *int flags*);

DESCRIPTION

The **dup3**() function duplicates an existing object descriptor while allowing the value of the new descriptor to be specified.

The close-on-exec flag on the new file descriptor is determined by the O_CLOEXEC bit in *flags*.

If *oldd* != *newd* and *flags* == 0, the behavior is identical to dup2(oldd, newd).

If *oldd* == *newd*, then **dup3**() fails, unlike dup2(2).

RETURN VALUES

The value -1 is returned if an error occurs. The external variable *errno* indicates the cause of the error.

ERRORS

The **dup3**() function fails if:

[EBADF]	The *oldd* argument is not a valid active descriptor or the *newd* argument is negative or exceeds the maximum allowable descriptor number
[EINVAL]	The *oldd* argument is equal to the *newd* argument.
[EINVAL]	The *flags* argument has bits set other than O_CLOEXEC.

SEE ALSO

accept(2), close(2), dup2(2), fcntl(2), getdtablesize(2), open(2), pipe(2), socket(2), socketpair(2)

STANDARDS

The **dup3**() function does not conform to any standard.

HISTORY

The **dup3**() function appeared in FreeBSD 10.0.

NAME

duplocale - duplicate an locale

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <locale.h>

locale_t
duplocale(*locale_t locale*);

DESCRIPTION

Duplicates an existing *locale_t* returning a new *locale_t* that refers to the same locale values but has an independent internal state. Various functions, such as mblen(3) require a persistent state. These functions formerly used static variables and calls to them from multiple threads had undefined behavior. They now use fields in the *locale_t* associated with the current thread by uselocale(3). These calls are therefore only thread safe on threads with a unique per-thread locale. The locale returned by this call must be freed with freelocale(3).

SEE ALSO

freelocale(3), localeconv(3), newlocale(3), querylocale(3), uselocale(3), xlocale(3)

STANDARDS

This function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

BUGS

Ideally, uselocale(3) should make a copy of the *locale_t* implicitly to ensure thread safety, and a copy of the global locale should be installed lazily on each thread. The FreeBSD implementation does not do this, for compatibility with Darwin.

NAME

easterg, **easterog**, **easteroj**, **gdate**, **jdate**, **ndaysg**, **ndaysj**, **week**, **weekday** - Calendar arithmetic for the Christian era

LIBRARY

Calendar Arithmetic Library (libcalendar, -lcalendar)

SYNOPSIS

#include <calendar.h>

*struct date **
easterg(*int year*, *struct date *dt*);

*struct date **
easterog(*int year*, *struct date *dt*);

*struct date **
easteroj(*int year*, *struct date *dt*);

*struct date **
gdate(*int nd*, *struct date *dt*);

*struct date **
jdate(*int nd*, *struct date *dt*);

int
ndaysg(*struct date *dt*);

int
ndaysj(*struct date *dt*);

int
week(*int nd*, *int *year*);

int
weekday(*int nd*);

DESCRIPTION

These functions provide calendar arithmetic for a large range of years, starting at March 1st, year zero (i.e., 1 B.C.) and ending way beyond year 100000.

Programs should be linked with **-lcalendar**.

The functions **easterg**(), **easterog**() and **easteroj**() store the date of Easter Sunday into the structure pointed at by *dt* and return a pointer to this structure. The function **easterg**() assumes Gregorian Calendar (adopted by most western churches after 1582) and the functions **easterog**() and **easteroj**() compute the date of Easter Sunday according to the orthodox rules (Western churches before 1582, Greek and Russian Orthodox Church until today). The result returned by **easterog**() is the date in Gregorian Calendar, whereas **easteroj**() returns the date in Julian Calendar.

The functions **gdate**(), **jdate**(), **ndaysg**() and **ndaysj**() provide conversions between the common "year, month, day" notation of a date and the "number of days" representation, which is better suited for calculations. The days are numbered from March 1st year 1 B.C., starting with zero, so the number of a day gives the number of days since March 1st, year 1 B.C. The conversions work for nonnegative day numbers only.

The **gdate**() and **jdate**() functions store the date corresponding to the day number *nd* into the structure pointed at by *dt* and return a pointer to this structure.

The **ndaysg**() and **ndaysj**() functions return the day number of the date pointed at by *dt*.

The **gdate**() and **ndaysg**() functions assume Gregorian Calendar after October 4, 1582 and Julian Calendar before, whereas **jdate**() and **ndaysj**() assume Julian Calendar throughout.

The two calendars differ by the definition of the leap year. The Julian Calendar says every year that is a multiple of four is a leap year. The Gregorian Calendar excludes years that are multiples of 100 and not multiples of 400. This means the years 1700, 1800, 1900, 2100 are not leap years and the year 2000 is a leap year. The new rules were inaugurated on October 4, 1582 by deleting ten days following this date. Most catholic countries adopted the new calendar by the end of the 16th century, whereas others stayed with the Julian Calendar until the 20th century. The United Kingdom and their colonies switched on September 2, 1752. They already had to delete 11 days.

The function **week**() returns the number of the week which contains the day numbered *nd*. The argument **year* is set with the year that contains (the greater part of) the week. The weeks are numbered per year starting with week 1, which is the first week in a year that includes more than three days of the year. Weeks start on Monday. This function is defined for Gregorian Calendar only.

The function **weekday**() returns the weekday (Mo = 0 .. Su = 6) of the day numbered *nd*.

The structure *date* is defined in <*calendar.h*>. It contains these fields:

```
int y;      /* year (0000 - ????) */
int m;      /* month (1 - 12) */
int d;      /* day of month (1 - 31) */
```

The year zero is written as "1 B.C." by historians and "0" by astronomers and in this library.

SEE ALSO

ncal(1), strftime(3)

STANDARDS

The week number conforms to ISO 8601: 1988.

HISTORY

The **calendar** library first appeared in FreeBSD 3.0.

AUTHORS

This manual page and the library was written by Wolfgang Helbig <*helbig@FreeBSD.org*>.

BUGS

The library was coded with great care so there are no bugs left.

NAME

getfsent, **getfsspec**, **getfsfile**, **setfsent**, **endfsent** - get file system descriptor file entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fstab.h>

*struct fstab **
getfsent(*void*);

*struct fstab **
getfsspec(*const char *spec*);

*struct fstab **
getfsfile(*const char *file*);

int
setfsent(*void*);

void
endfsent(*void*);

void
setfstab(*const char *file*);

*const char **
getfstab(*void*);

DESCRIPTION

The **getfsent**(), **getfsspec**(), and **getfsfile**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file, <*fstab.h*>.

```
struct fstab {
        char    *fs_spec; /* block special device name */
        char    *fs_file;   /* file system path prefix */
        char    *fs_vfstype;        /* File system type, ufs, nfs */
        char    *fs_mntops;         /* Mount options ala -o */
        char    *fs_type; /* FSTAB_* from fs_mntops */
```

```
        int     fs_freq;   /* dump frequency, in days */
        int     fs_passno;         /* pass number on parallel fsck */
};
```

The fields have meanings described in fstab(5).

The **setfsent**() function opens the file (closing any previously opened file) or rewinds it if it is already open.

The **endfsent**() function closes the file.

The **setfstab**() function sets the file to be used by subsequent operations. The value set by **setfstab**() does not persist across calls to **endfsent**().

The **getfstab**() function returns the name of the file that will be used.

The **getfsspec**() and **getfsfile**() functions search the entire file (opening it if necessary) for a matching special file name or file system file name.

For programs wishing to read the entire database, **getfsent**() reads the next entry (opening the file if necessary).

All entries in the file with a type field equivalent to FSTAB_XX are ignored.

RETURN VALUES

The **getfsent**(), **getfsspec**(), and **getfsfile**() functions return a NULL pointer on EOF or error. The **setfsent**() function returns 0 on failure, 1 on success. The **endfsent**() function returns nothing.

ENVIRONMENT

PATH_FSTAB — If the environment variable PATH_FSTAB is set, all operations are performed against the specified file. PATH_FSTAB will not be honored if the process environment or memory address space is considered "tainted". (See issetugid(2) for more information.)

FILES

/etc/fstab

SEE ALSO

fstab(5)

HISTORY

The **getfsent**() function appeared in 4.0BSD; the **endfsent**(), **getfsfile**(), **getfsspec**(), and **setfsent**() functions appeared in 4.3BSD; the **setfstab**() and **getfstab**() functions appeared in FreeBSD 5.1.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

NAME

getgrent, **getgrent_r**, **getgrnam**, **getgrnam_r**, **getgrgid**, **getgrgid_r**, **setgroupent**, **setgrent**, **endgrent** - group database operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <grp.h>

*struct group **
getgrent(*void*);

int
getgrent_r(*struct group *grp*, *char *buffer*, *size_t bufsize*, *struct group **result*);

*struct group **
getgrnam(*const char *name*);

int
getgrnam_r(*const char *name*, *struct group *grp*, *char *buffer*, *size_t bufsize*, *struct group **result*);

*struct group **
getgrgid(*gid_t gid*);

int
getgrgid_r(*gid_t gid*, *struct group *grp*, *char *buffer*, *size_t bufsize*, *struct group **result*);

int
setgroupent(*int stayopen*);

void
setgrent(*void*);

void
endgrent(*void*);

DESCRIPTION

These functions operate on the group database file */etc/group* which is described in group(5). Each line of the database is defined by the structure *group* found in the include file <*grp.h*>:

```
struct group {
        char    *gr_name;       /* group name */
        char    *gr_passwd;     /* group password */
        gid_t   gr_gid;         /* group id */
        char    **gr_mem;       /* group members */
};
```

The functions **getgrnam**() and **getgrgid**() search the group database for the given group name pointed to by *name* or the group id pointed to by *gid*, respectively, returning the first one encountered. Identical group names or group gids may result in undefined behavior.

The **getgrent**() function sequentially reads the group database and is intended for programs that wish to step through the complete list of groups.

The functions **getgrent_r**(), **getgrnam_r**(), and **getgrgid_r**() are thread-safe versions of **getgrent**(), **getgrnam**(), and **getgrgid**(), respectively. The caller must provide storage for the results of the search in the *grp*, *buffer*, *bufsize*, and *result* arguments. When these functions are successful, the *grp* argument will be filled-in, and a pointer to that argument will be stored in *result*. If an entry is not found or an error occurs, *result* will be set to NULL.

These functions will open the group file for reading, if necessary.

The **setgroupent**() function opens the file, or rewinds it if it is already open. If *stayopen* is non-zero, file descriptors are left open, significantly speeding functions subsequent calls. This functionality is unnecessary for **getgrent**() as it does not close its file descriptors by default. It should also be noted that it is dangerous for long-running programs to use this functionality as the group file may be updated.

The **setgrent**() function is identical to **setgroupent**() with an argument of zero.

The **endgrent**() function closes any open files.

RETURN VALUES

The functions **getgrent**(), **getgrnam**(), and **getgrgid**(), return a pointer to a group structure on success or NULL if the entry is not found or if an error occurs. If an error does occur, *errno* will be set. Note that programs must explicitly set *errno* to zero before calling any of these functions if they need to distinguish between a non-existent entry and an error. The functions **getgrent_r**(), **getgrnam_r**(), and **getgrgid_r**() return 0 if no error occurred, or an error number to indicate failure. It is not an error if a matching entry is not found. (Thus, if *result* is set to NULL and the return value is 0, no matching entry exists.)

The function **setgroupent**() returns the value 1 if successful, otherwise the value 0 is returned. The functions **endgrent**(), **setgrent**() and **setgrfile**() have no return value.

FILES

/etc/group group database file

COMPATIBILITY

The historic function **setgrfile**(), which allowed the specification of alternate password databases, has been deprecated and is no longer available.

SEE ALSO

getpwent(3), group(5), nsswitch.conf(5), yp(8)

STANDARDS

The **getgrent**(), **getgrnam**(), **getgrnam_r**(), **getgrgid**(), **getgrgid_r**() and **endgrent**() functions conform to ISO/IEC 9945-1:1996 ("POSIX.1"). The **setgrent**() function differs from that standard in that its return type is *int* rather than *void*.

HISTORY

The functions **endgrent**(), **getgrent**(), **getgrnam**(), **getgrgid**(), and **setgrent**() appeared in Version 7 AT&T UNIX. The functions **setgrfile**() and **setgroupent**() appeared in 4.3BSD-Reno. The functions **getgrent_r**(), **getgrnam_r**(), and **getgrgid_r**() appeared in FreeBSD 5.1.

BUGS

The functions **getgrent**(), **getgrnam**(), **getgrgid**(), **setgroupent**() and **setgrent**() leave their results in an internal static object and return a pointer to that object. Subsequent calls to the same function will modify the same object.

The functions **getgrent**(), **getgrent_r**(), **endgrent**(), **setgroupent**(), and **setgrent**() are fairly useless in a networked environment and should be avoided, if possible. The **getgrent**() and **getgrent_r**() functions make no attempt to suppress duplicate information if multiple sources are specified in nsswitch.conf(5).

NAME

gethostbyname, **gethostbyname2**, **gethostbyaddr**, **gethostent**, **sethostent**, **endhostent**, **herror**, **hstrerror** - get network host entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <netdb.h>

int h_errno;

*struct hostent **
gethostbyname(*const char *name*);

*struct hostent **
gethostbyname2(*const char *name*, *int af*);

*struct hostent **
gethostbyaddr(*const void *addr*, *socklen_t len*, *int af*);

*struct hostent **
gethostent(*void*);

void
sethostent(*int stayopen*);

void
endhostent(*void*);

void
herror(*const char *string*);

*const char **
hstrerror(*int err*);

DESCRIPTION

The getaddrinfo(3) and getnameinfo(3) functions are preferred over the gethostbyname(), gethostbyname2(), and gethostbyaddr() functions.

The **gethostbyname**(), **gethostbyname2**() and **gethostbyaddr**() functions each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively.

The *name* argument passed to **gethostbyname**() or **gethostbyname2**() should point to a NUL-terminated hostname. The *addr* argument passed to **gethostbyaddr**() should point to an address which is *len* bytes long, in binary form (i.e., not an IP address in human readable ASCII form). The *af* argument specifies the address family (e.g. AF_INET, AF_INET6, etc.) of this address.

The structure returned contains either the information obtained from the name server, named(8), broken-out fields from a line in */etc/hosts*, or database entries supplied by the yp(8) system. The order of the lookups is controlled by the 'hosts' entry in nsswitch.conf(5).

```
struct    hostent {
          char    *h_name;/* official name of host */
          char    **h_aliases;        /* alias list */
          int     h_addrtype;         /* host address type */
          int     h_length; /* length of address */
          char    **h_addr_list;      /* list of addresses from name server */
};
#define   h_addr  h_addr_list[0]      /* address, for backward compatibility */
```

The members of this structure are:

h_name Official name of the host.

h_aliases A NULL-terminated array of alternate names for the host.

h_addrtype The type of address being returned; usually AF_INET.

h_length The length, in bytes, of the address.

h_addr_list A NULL-terminated array of network addresses for the host. Host addresses are returned in network byte order.

h_addr The first address in *h_addr_list*; this is for backward compatibility.

When using the nameserver, **gethostbyname**() and **gethostbyname2**() will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable "HOSTALIASES" contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See hostname(7) for the domain search procedure and

the alias file format.

The **gethostbyname2**() function is an evolution of **gethostbyname**() which is intended to allow lookups in address families other than AF_INET, for example AF_INET6.

The **sethostent**() function may be used to request the use of a connected TCP socket for queries. Queries will by default use UDP datagrams. If the *stayopen* flag is non-zero, a TCP connection to the name server will be used. It will remain open after calls to **gethostbyname**(), **gethostbyname2**() or **gethostbyaddr**() have completed.

The **endhostent**() function closes the TCP connection.

The **herror**() function writes a message to the diagnostic output consisting of the string argument *string*, the constant string ": ", and a message corresponding to the value of *h_errno*.

The **hstrerror**() function returns a string which is the message text corresponding to the value of the *err* argument.

FILES

/etc/hosts
/etc/nsswitch.conf
/etc/resolv.conf

EXAMPLES

Print out the hostname associated with a specific IP address:

```
const char *ipstr = "127.0.0.1";
struct in_addr ip;
struct hostent *hp;

if (!inet_aton(ipstr, &ip))
        errx(1, "can't parse IP address %s", ipstr);

if ((hp = gethostbyaddr((const void *)&ip,
   sizeof ip, AF_INET)) == NULL)
        errx(1, "no name associated with %s", ipstr);

printf("name associated with %s is %s\n", ipstr, hp->h_name);
```

DIAGNOSTICS

Error return status from **gethostbyname**(), **gethostbyname2**() and **gethostbyaddr**() is indicated by return of a NULL pointer. The integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine **herror**() can be used to print an error message describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

The variable *h_errno* can have the following values:

HOST_NOT_FOUND No such host is known.

TRY_AGAIN This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

NO_RECOVERY Some unexpected server failure was encountered. This is a non-recoverable error.

NO_DATA The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

SEE ALSO

getaddrinfo(3), getnameinfo(3), inet_aton(3), resolver(3), hosts(5), hostname(7), named(8)

CAVEAT

The **gethostent**() function is defined, and **sethostent**() and **endhostent**() are redefined, when Standard C Library (libc, -lc) is built to use only the routines to lookup in */etc/hosts* and not the name server.

The **gethostent**() function reads the next line of */etc/hosts*, opening the file if necessary.

The **sethostent**() function opens and/or rewinds the file */etc/hosts*. If the *stayopen* argument is non-zero, the file will not be closed after each call to **gethostbyname**(), **gethostbyname2**() or **gethostbyaddr**().

The **endhostent**() function closes the file.

HISTORY

The **herror**() function appeared in 4.3BSD. The **endhostent**(), **gethostbyaddr**(), **gethostbyname**(), **gethostent**(), and **sethostent**() functions appeared in 4.2BSD. The **gethostbyname2**() function first appeared in BIND version 4.9.4.

BUGS

These functions use a thread-specific data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

Though these functions are thread-safe, still it is recommended to use the getaddrinfo(3) family of functions, instead.

Only the Internet address format is currently understood.

NAME

getnetconfig, **setnetconfig**, **endnetconfig**, **getnetconfigent**, **freenetconfigent**, **nc_perror**, **nc_sperror** - get network configuration database entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <netconfig.h>

*struct netconfig **
getnetconfig(*void *handlep*);

*void **
setnetconfig(*void*);

int
endnetconfig(*void *handlep*);

*struct netconfig **
getnetconfigent(*const char *netid*);

void
freenetconfigent(*struct netconfig *netconfigp*);

void
nc_perror(*const char *msg*);

*char **
nc_sperror(*void*);

DESCRIPTION

The library routines described on this page provide the application access to the system network configuration database, */etc/netconfig*. The **getnetconfig**() function returns a pointer to the current entry in the netconfig database, formatted as a *struct netconfig*. Successive calls will return successive netconfig entries in the netconfig database. The **getnetconfig**() function can be used to search the entire netconfig file. The **getnetconfig**() function returns NULL at the end of the file. The *handlep* argument is the handle obtained through **setnetconfig**().

A call to **setnetconfig**() has the effect of "binding" to or "rewinding" the netconfig database. The

setnetconfig() function must be called before the first call to **getnetconfig**() and may be called at any other time. The **setnetconfig**() function need not be called before a call to **getnetconfigent**(). The **setnetconfig**() function returns a unique handle to be used by **getnetconfig**().

The **endnetconfig**() function should be called when processing is complete to release resources for reuse. The *handlep* argument is the handle obtained through **setnetconfig**(). Programmers should be aware, however, that the last call to **endnetconfig**() frees all memory allocated by **getnetconfig**() for the *struct netconfig* data structure. The **endnetconfig**() function may not be called before **setnetconfig**().

The **getnetconfigent**() function returns a pointer to the netconfig structure corresponding to *netid*. It returns NULL if *netid* is invalid (that is, does not name an entry in the netconfig database).

The **freenetconfigent**() function frees the netconfig structure pointed to by *netconfigp* (previously returned by **getnetconfigent**()).

The **nc_perror**() function prints a message to the standard error indicating why any of the above routines failed. The message is prepended with the string *msg* and a colon. A newline character is appended at the end of the message.

The **nc_sperror**() function is similar to **nc_perror**() but instead of sending the message to the standard error, will return a pointer to a string that contains the error message.

The **nc_perror**() and **nc_sperror**() functions can also be used with the NETPATH access routines defined in getnetpath(3).

RETURN VALUES

The **setnetconfig**() function returns a unique handle to be used by **getnetconfig**(). In the case of an error, **setnetconfig**() returns NULL and **nc_perror**() or **nc_sperror**() can be used to print the reason for failure.

The **getnetconfig**() function returns a pointer to the current entry in the netconfig database, formatted as a *struct netconfig*. The **getnetconfig**() function returns NULL at the end of the file, or upon failure.

The **endnetconfig**() function returns 0 on success and -1 on failure (for example, if **setnetconfig**() was not called previously).

On success, **getnetconfigent**() returns a pointer to the *struct netconfig* structure corresponding to *netid*; otherwise it returns NULL.

The **nc_sperror**() function returns a pointer to a buffer which contains the error message string. This buffer is overwritten on each call. In multithreaded applications, this buffer is implemented as thread-

specific data.

FILES

/etc/netconfig

SEE ALSO

getnetpath(3), netconfig(5)

NAME

getnetent, **getnetbyaddr**, **getnetbyname**, **setnetent**, **endnetent** - get network entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <netdb.h>

*struct netent **
getnetent(*void*);

*struct netent **
getnetbyname(*const char *name*);

*struct netent **
getnetbyaddr(*uint32_t net*, *int type*);

void
setnetent(*int stayopen*);

void
endnetent(*void*);

DESCRIPTION

The **getnetent**(), **getnetbyname**(), and **getnetbyaddr**() functions each return a pointer to an object with the following structure describing an internet network. This structure contains either the information obtained from the nameserver, named(8), broken-out fields of a line in the network data base */etc/networks*, or entries supplied by the yp(8) system. The order of the lookups is controlled by the 'networks' entry in nsswitch.conf(5).

```
struct netent {
        char    *n_name;/* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net number type */
        uint32_t  n_net;        /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

The **getnetent**() function reads the next line of the file, opening the file if necessary.

The **setnetent**() function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getnetbyname**() or **getnetbyaddr**().

The **endnetent**() function closes the file.

The **getnetbyname**() function and **getnetbyaddr**() sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. The *type* argument must be AF_INET. Network numbers are supplied in host order.

FILES

/etc/networks
/etc/nsswitch.conf
/etc/resolv.conf

DIAGNOSTICS

Null pointer returned on EOF or error.

SEE ALSO

networks(5)

RFC 1101

HISTORY

The **getnetent**(), **getnetbyaddr**(), **getnetbyname**(), **setnetent**(), and **endnetent**() functions appeared in 4.2BSD.

BUGS

The data space used by these functions is thread-specific; if future use requires the data, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

NAME

getnetgrent, **innetgr**, **setnetgrent**, **endnetgrent** - netgroup database operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <netdb.h>

int
getnetgrent(*char **host*, *char **user*, *char **domain*);

int
getnetgrent_r(*char **host*, *char **user*, *char **domain*, *char *buf*, *size_t bufsize*);

int
innetgr(*const char *netgroup*, *const char *host*, *const char *user*, *const char *domain*);

void
setnetgrent(*const char *netgroup*);

void
endnetgrent(*void*);

DESCRIPTION

These functions operate on the netgroup database file */etc/netgroup* which is described in netgroup(5). The database defines a set of netgroups, each made up of one or more triples:

(host, user, domain)

that defines a combination of host, user and domain. Any of the three fields may be specified as ‘‘wildcards’’ that match any string.

The function **getnetgrent**() sets the three pointer arguments to the strings of the next member of the current netgroup. If any of the string pointers are NULL that field is considered a wildcard.

The functions **setnetgrent**() and **endnetgrent**() set the current netgroup and terminate the current netgroup respectively. If **setnetgrent**() is called with a different netgroup than the previous call, an implicit **endnetgrent**() is implied. The **setnetgrent**() function also sets the offset to the first member of the netgroup.

The function **innetgr**() searches for a match of all fields within the specified group. If any of the **host**, **user**, or **domain** arguments are NULL those fields will match any string value in the netgroup member.

RETURN VALUES

The function **getnetgrent**() returns 0 for ‘‘no more netgroup members’’ and 1 otherwise. The function **innetgr**() returns 1 for a successful match and 0 otherwise. The functions **setnetgrent**() and **endnetgrent**() have no return value.

FILES

/etc/netgroup netgroup database file

COMPATIBILITY

The netgroup members have three string fields to maintain compatibility with other vendor implementations, however it is not obvious what use the **domain** string has within BSD.

SEE ALSO

netgroup(5)

BUGS

The function **getnetgrent**() returns pointers to dynamically allocated data areas that are freed when the function **endnetgrent**() is called.

NAME

getnetpath, **setnetpath**, **endnetpath** - get */etc/netconfig* entry corresponding to NETPATH component

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <netconfig.h>

*struct netconfig **
getnetpath(*void *handlep*);

*void **
setnetpath(*void*);

int
endnetpath(*void *handlep*);

DESCRIPTION

The routines described in this page provide the application access to the system network configuration database, */etc/netconfig*, as it is "filtered" by the NETPATH environment variable (see environ(7)). See getnetconfig(3) for other routines that also access the network configuration database directly. The NETPATH variable is a list of colon-separated network identifiers.

The **getnetpath**() function returns a pointer to the netconfig database entry corresponding to the first valid NETPATH component. The netconfig entry is formatted as a *struct netconfig*. On each subsequent call, **getnetpath**() returns a pointer to the netconfig entry that corresponds to the next valid NETPATH component. The **getnetpath**() function can thus be used to search the netconfig database for all networks included in the NETPATH variable. When NETPATH has been exhausted, **getnetpath**() returns NULL.

A call to **setnetpath**() "binds" to or "rewinds" NETPATH. The **setnetpath**() function must be called before the first call to **getnetpath**() and may be called at any other time. It returns a handle that is used by **getnetpath**().

The **getnetpath**() function silently ignores invalid NETPATH components. A NETPATH component is invalid if there is no corresponding entry in the netconfig database.

If the NETPATH variable is unset, **getnetpath**() behaves as if NETPATH were set to the sequence of "default" or "visible" networks in the netconfig database, in the order in which they are listed.

The **endnetpath**() function may be called to "unbind" from NETPATH when processing is complete, releasing resources for reuse. Programmers should be aware, however, that **endnetpath**() frees all memory allocated by **getnetpath**() for the struct netconfig data structure.

RETURN VALUES

The **setnetpath**() function returns a handle that is used by **getnetpath**(). In case of an error, **setnetpath**() returns NULL.

The **endnetpath**() function returns 0 on success and -1 on failure (for example, if **setnetpath**() was not called previously). The **nc_perror**() or **nc_sperror**() function can be used to print out the reason for failure. See getnetconfig(3).

When first called, **getnetpath**() returns a pointer to the netconfig database entry corresponding to the first valid NETPATH component. When NETPATH has been exhausted, **getnetpath**() returns NULL.

SEE ALSO

getnetconfig(3), netconfig(5), environ(7)

NAME

getprotoent, **getprotobynumber**, **getprotobyname**, **setprotoent**, **endprotoent** - get protocol entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <netdb.h>

*struct protoent **
getprotoent(*void*);

*struct protoent **
getprotobyname(*const char *name*);

*struct protoent **
getprotobynumber(*int proto*);

void
setprotoent(*int stayopen*);

void
endprotoent(*void*);

DESCRIPTION

The **getprotoent**(), **getprotobyname**(), and **getprotobynumber**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, */etc/protocols*.

```
struct protoent {
        char    *p_name;/* official name of protocol */
        char    **p_aliases;        /* alias list */
        int     p_proto;  /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

The **getprotoent**() function reads the next line of the file, opening the file if necessary.

The **setprotoent**() function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getprotobyname**() or **getprotobynumber**().

The **endprotoent**() function closes the file.

The **getprotobyname**() function and **getprotobynumber**() sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

RETURN VALUES

Null pointer returned on EOF or error.

FILES

/etc/protocols

SEE ALSO

protocols(5)

HISTORY

The **getprotoent**(), **getprotobynumber**(), **getprotobyname**(), **setprotoent**(), and **endprotoent**() functions appeared in 4.2BSD.

BUGS

These functions use a thread-specific data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

NAME

getpwent, **getpwent_r**, **getpwnam**, **getpwnam_r**, **getpwuid**, **getpwuid_r**, **setpassent**, **setpwent**, **endpwent** - password database operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *
getpwent(void);

int
getpwent_r(struct passwd *pwd, char *buffer, size_t bufsize, struct passwd **result);

struct passwd *
getpwnam(const char *login);

int
getpwnam_r(const char *name, struct passwd *pwd, char *buffer, size_t bufsize,
    struct passwd **result);

struct passwd *
getpwuid(uid_t uid);

int
getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer, size_t bufsize, struct passwd **result);

int
setpassent(int stayopen);

void
setpwent(void);

void
endpwent(void);
```

DESCRIPTION

These functions operate on the password database file which is described in passwd(5). Each entry in the database is defined by the structure *passwd* found in the include file <*pwd.h*>:

```
struct passwd {
        char    *pw_name;       /* user name */
        char    *pw_passwd;     /* encrypted password */
        uid_t   pw_uid;         /* user uid */
        gid_t   pw_gid;         /* user gid */
        time_t  pw_change;      /* password change time */
        char    *pw_class;      /* user access class */
        char    *pw_gecos;      /* Honeywell login info */
        char    *pw_dir;  /* home directory */
        char    *pw_shell;      /* default shell */
        time_t  pw_expire;      /* account expiration */
        int     pw_fields;      /* internal: fields filled in */
};
```

The functions **getpwnam**() and **getpwuid**() search the password database for the given login name or user uid, respectively, always returning the first one encountered.

The **getpwent**() function sequentially reads the password database and is intended for programs that wish to process the complete list of users.

The functions **getpwent_r**(), **getpwnam_r**(), and **getpwuid_r**() are thread-safe versions of **getpwent**(), **getpwnam**(), and **getpwuid**(), respectively. The caller must provide storage for the results of the search in the *pwd*, *buffer*, *bufsize*, and *result* arguments. When these functions are successful, the *pwd* argument will be filled-in, and a pointer to that argument will be stored in *result*. If an entry is not found or an error occurs, *result* will be set to NULL.

The **setpassent**() function accomplishes two purposes. First, it causes **getpwent**() to ‘‘rewind’’ to the beginning of the database. Additionally, if *stayopen* is non-zero, file descriptors are left open, significantly speeding up subsequent accesses for all of the routines. (This latter functionality is unnecessary for **getpwent**() as it does not close its file descriptors by default.)

It is dangerous for long-running programs to keep the file descriptors open as the database will become out of date if it is updated while the program is running.

The **setpwent**() function is identical to **setpassent**() with an argument of zero.

The **endpwent**() function closes any open files.

These routines have been written to ‘‘shadow’’ the password file, e.g. allow only certain programs to have access to the encrypted password. If the process which calls them has an effective uid of 0, the encrypted password will be returned, otherwise, the password field of the returned structure will point to the string ‘*’.

RETURN VALUES

The functions **getpwent**(), **getpwnam**(), and **getpwuid**() return a valid pointer to a passwd structure on success or NULL if the entry is not found or if an error occurs. If an error does occur, *errno* will be set. Note that programs must explicitly set *errno* to zero before calling any of these functions if they need to distinguish between a non-existent entry and an error. The functions **getpwent_r**(), **getpwnam_r**(), and **getpwuid_r**() return 0 if no error occurred, or an error number to indicate failure. It is not an error if a matching entry is not found. (Thus, if *result* is NULL and the return value is 0, no matching entry exists.)

The **setpassent**() function returns 0 on failure and 1 on success. The **endpwent**() and **setpwent**() functions have no return value.

FILES

/etc/pwd.db	The insecure password database file
/etc/spwd.db	The secure password database file
/etc/master.passwd	The current password file
/etc/passwd	A Version 7 format password file

COMPATIBILITY

The historic function setpwfile(3), which allowed the specification of alternate password databases, has been deprecated and is no longer available.

ERRORS

These routines may fail for any of the errors specified in open(2), dbopen(3), socket(2), and connect(2), in addition to the following:

[ERANGE] The buffer specified by the *buffer* and *bufsize* arguments was insufficiently sized to store the result. The caller should retry with a larger buffer.

SEE ALSO

getlogin(2), getgrent(3), nsswitch.conf(5), passwd(5), pwd_mkdb(8), vipw(8), yp(8)

STANDARDS

The **getpwent**(), **getpwnam**(), **getpwnam_r**(), **getpwuid**(), **getpwuid_r**(), **setpwent**(), and **endpwent**() functions conform to ISO/IEC 9945-1:1996 ("POSIX.1").

HISTORY

The **getpwent**(), **getpwnam**(), **getpwuid**(), **setpwent**(), and **endpwent**() functions appeared in Version 7 AT&T UNIX. The **setpassent**() function appeared in 4.3BSD-Reno. The **getpwent_r**(), **getpwnam_r**(), and **getpwuid_r**() functions appeared in FreeBSD 5.1.

BUGS

The functions **getpwent**(), **getpwnam**(), and **getpwuid**(), leave their results in an internal static object and return a pointer to that object. Subsequent calls to the same function will modify the same object.

The functions **getpwent**(), **getpwent_r**(), **endpwent**(), **setpassent**(), and **setpwent**() are fairly useless in a networked environment and should be avoided, if possible. The **getpwent**() and **getpwent_r**() functions make no attempt to suppress duplicate information if multiple sources are specified in nsswitch.conf(5).

NAME

getrpcent, **getrpcbyname**, **getrpcbynumber**, **endrpcent**, **setrpcent** - get RPC entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

*struct rpcent **
getrpcent(*void*);

*struct rpcent **
getrpcbyname(*const char *name*);

*struct rpcent **
getrpcbynumber(*int number*);

void
setrpcent(*int stayopen*);

void
endrpcent(*void*);

DESCRIPTION

The **getrpcent**(), **getrpcbyname**(), and **getrpcbynumber**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, */etc/rpc*:

```
struct rpcent {
        char    *r_name; /* name of server for this rpc program */
        char    **r_aliases;        /* alias list */
        long    r_number;           /* rpc program number */
};
```

The members of this structure are:

r_name The name of the server for this rpc program.

r_aliases A zero terminated list of alternate names for the rpc program.

r_number
The rpc program number for this service.

The **getrpcent**() function reads the next line of the file, opening the file if necessary.

The **setrpcent**() function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getrpcent**() (either directly, or indirectly through one of the other "getrpc" calls).

The **endrpcent**() function closes the file.

The **getrpcbyname**() and **getrpcbynumber**() functions sequentially search from the beginning of the file until a matching rpc program name or program number is found, or until end-of-file is encountered.

FILES
/etc/rpc

DIAGNOSTICS
A NULL pointer is returned on EOF or error.

SEE ALSO
rpc(5), rpcinfo(8), ypserv(8)

BUGS
All information is contained in a static area so it must be copied if it is to be saved.

NAME

getservent, **getservbyport**, **getservbyname**, **setservent**, **endservent** - get service entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <netdb.h>

*struct servent **
getservent();

*struct servent **
getservbyname(*const char *name*, *const char *proto*);

*struct servent **
getservbyport(*int port*, *const char *proto*);

void
setservent(*int stayopen*);

void
endservent(*void*);

DESCRIPTION

The **getservent**(), **getservbyname**(), and **getservbyport**() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, */etc/services*.

```
struct servent {
        char    *s_name; /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto; /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

The **getservent**() function reads the next line of the file, opening the file if necessary.

The **setservent**() function opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to **getservbyname**() or **getservbyport**().

The **endservent**() function closes the file.

The **getservbyname**() and **getservbyport**() functions sequentially search from the beginning of the file until a matching protocol name or port number (which must be specified in network byte order) is found, or until EOF is encountered. If a protocol name is also supplied (non- NULL), searches must also match the protocol.

FILES

/etc/services

DIAGNOSTICS

Null pointer returned on EOF or error.

SEE ALSO

getprotoent(3), services(5)

HISTORY

The **getservent**(), **getservbyport**(), **getservbyname**(), **setservent**(), and **endservent**() functions appeared in 4.2BSD.

BUGS

These functions use a thread-specific data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

getttyent, **getttynam**, **setttyent**, **endttyent**, **isdialuptty**, **isnettty** - ttys(5) file routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <ttyent.h>

*struct ttyent **
getttyent(*void*);

*struct ttyent **
getttynam(*const char *name*);

int
setttyent(*void*);

int
endttyent(*void*);

int
isdialuptty(*const char *name*);

int
isnettty(*const char *name*);

DESCRIPTION

The **getttyent**(), and **getttynam**() functions each return a pointer to an object, with the following structure, containing the broken-out fields of a line from the tty description file.

```
struct ttyent {
        char    *ty_name;       /* terminal device name */
        char    *ty_getty;      /* command to execute, usually getty */
        char    *ty_type; /* terminal type for termcap */
#define TTY_ON          0x01    /* enable logins (start ty_getty program) */
#define TTY_SECURE      0x02    /* allow uid of 0 to login */
#define TTY_DIALUP      0x04    /* is a dialup tty */
#define TTY_NETWORK     0x08    /* is a network tty */
#define TTY_IFEXISTS    0x10    /* configured as "onifexists" */
```

```
#define TTY_IFCONSOLE0x20       /* configured as "onifconsole" */
        int     ty_status;/* status flags */
        char    *ty_window;     /* command to start up window manager */
        char    *ty_comment;    /* comment field */
        char    *ty_group;      /* tty group name */
};
```

The fields are as follows:

ty_name The name of the character-special file.

ty_getty The name of the command invoked by init(8) to initialize tty line characteristics.

ty_type The name of the default terminal type connected to this tty line.

ty_status A mask of bit fields which indicate various actions allowed on this tty line. The possible flags are as follows:

TTY_ON Enables logins (i.e., init(8) will start the command referenced by *ty_getty* on this entry).

TTY_SECURE Allow users with a uid of 0 to login on this terminal.

TTY_DIALUP Identifies a tty as a dialin line. If this flag is set, then **isdialuptty**() will return a non-zero value.

TTY_NETWORK Identifies a tty used for network connections. If this flag is set, then **isnettty**() will return a non-zero value.

TTY_IFEXISTS Identifies a tty that does not necessarily exist.

TTY_IFCONSOLE
Identifies a tty that might be a system console.

ty_window The command to execute for a window system associated with the line.

ty_group A group name to which the tty belongs. If no group is specified in the ttys description file, then the tty is placed in an anonymous group called "none".

ty_comment Any trailing comment field, with any leading hash marks (‘‘#’’) or whitespace removed.

If any of the fields pointing to character strings are unspecified, they are returned as null pointers. The field *ty_status* will be zero if no flag values are specified.

See ttys(5) for a more complete discussion of the meaning and usage of the fields.

The **getttyent**() function reads the next line from the ttys file, opening the file if necessary. The **setttyent**() function rewinds the file if open, or opens the file if it is unopened. The **endttyent**() function closes any open files.

The **getttynam**() function searches from the beginning of the file until a matching *name* is found (or until EOF is encountered).

RETURN VALUES

The routines **getttyent**() and **getttynam**() return a null pointer on EOF or error. The **setttyent**() function and **endttyent**() return 0 on failure and 1 on success.

The routines **isdialuptty**() and **isnettty**() return non-zero if the dialup or network flag is set for the tty entry relating to the tty named by the argument, and zero otherwise.

FILES

/etc/ttys

SEE ALSO

login(1), gettytab(5), termcap(5), ttys(5), getty(8), init(8)

HISTORY

The **getttyent**(), **getttynam**(), **setttyent**(), and **endttyent**() functions appeared in 4.3BSD.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it.

NAME

getusershell, **setusershell**, **endusershell** - get valid user shells

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*char **
getusershell(*void*);

void
setusershell(*void*);

void
endusershell(*void*);

DESCRIPTION

The **getusershell**() function returns a pointer to a valid user shell as defined by the system manager in the shells database as described in shells(5). If the shells database is not available, **getusershell**() behaves as if */bin/sh* and */bin/csh* were listed.

The **getusershell**() function reads the next line (opening the file if necessary); **setusershell**() rewinds the file; **endusershell**() closes it.

FILES

/etc/shells

DIAGNOSTICS

The routine **getusershell**() returns a null pointer (0) on EOF.

SEE ALSO

nsswitch.conf(5), shells(5)

HISTORY

The **getusershell**() function appeared in 4.3BSD.

BUGS

The **getusershell**() function leaves its result in an internal static object and returns a pointer to that

object. Subsequent calls to **getusershell**() will modify the same object.

NAME

endutxent, **getutxent**, **getutxid**, **getutxline**, **getutxuser**, **pututxline**, **setutxdb**, **setutxent** - user accounting database functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <utmpx.h>

void
endutxent(*void*);

*struct utmpx **
getutxent(*void*);

*struct utmpx **
getutxid(*const struct utmpx *id*);

*struct utmpx **
getutxline(*const struct utmpx *line*);

*struct utmpx **
getutxuser(*const char *user*);

*struct utmpx **
pututxline(*const struct utmpx *utmpx*);

int
setutxdb(*int type*, *const char *file*);

void
setutxent(*void*);

DESCRIPTION

These functions operate on the user accounting database which stores records of various system activities, such as user login and logouts, but also system startups and shutdowns and modifications to the system's clock. The system stores these records in three databases, each having a different purpose:

/var/run/utx.active

Log of currently active user login sessions. This file is similar to the traditional *utmp* file. This file only contains process related entries, such as user login and logout records.

/var/log/utx.lastlogin

Log of last user login entries per user. This file is similar to the traditional *lastlog* file. This file only contains user login records for users who have at least logged in once.

/var/log/utx.log

Log of all entries, sorted by date of addition. This file is similar to the traditional *wtmp* file. This file may contain any type of record described below.

Each entry in these databases is defined by the structure *utmpx* found in the include file <*utmpx.h*>:

```
struct utmpx {
        short         ut_type;    /* Type of entry. */
        struct timeval  ut_tv;      /* Time entry was made. */
        char          ut_id[];    /* Record identifier. */
        pid_t         ut_pid;     /* Process ID. */
        char          ut_user[];  /* User login name. */
        char          ut_line[];  /* Device name. */
        char          ut_host[];  /* Remote hostname. */
};
```

The *ut_type* field indicates the type of the log entry, which can have one of the following values:

EMPTY No valid user accounting information.

BOOT_TIME Identifies time of system boot.

SHUTDOWN_TIME
Identifies time of system shutdown.

OLD_TIME Identifies time when system clock changed.

NEW_TIME Identifies time after system clock changed.

USER_PROCESS Identifies a process.

INIT_PROCESS Identifies a process spawned by the init process.

LOGIN_PROCESS Identifies the session leader of a logged-in user.

DEAD_PROCESS Identifies a session leader who has exited.

Entries of type INIT_PROCESS and LOGIN_PROCESS are not processed by this implementation.

Other fields inside the structure are:

ut_tv The time the event occurred. This field is used for all types of entries, except EMPTY.

ut_id An identifier that is used to refer to the entry. This identifier can be used to remove or replace a login entry by writing a new entry to the database containing the same value for *ut_id*. This field is only applicable to entries of type USER_PROCESS, INIT_PROCESS, LOGIN_PROCESS and DEAD_PROCESS.

ut_pid The process identifier of the session leader of the login session. This field is only applicable to entries of type USER_PROCESS, INIT_PROCESS, LOGIN_PROCESS and DEAD_PROCESS.

ut_user The user login name corresponding with the login session. This field is only applicable to entries of type USER_PROCESS and INIT_PROCESS. For INIT_PROCESS entries this entry typically contains the name of the login process.

ut_line The name of the TTY character device, without the leading */dev/* prefix, corresponding with the device used to facilitate the user login session. If no TTY character device is used, this field is left blank. This field is only applicable to entries of type USER_PROCESS and LOGIN_PROCESS.

ut_host
The network hostname of the remote system, connecting to perform a user login. If the user login session is not performed across a network, this field is left blank. This field is only applicable to entries of type USER_PROCESS.

This implementation guarantees all inapplicable fields are discarded. The *ut_user*, *ut_line* and *ut_host* fields of the structure returned by the library functions are also guaranteed to be null-terminated in this implementation.

The **getutxent**() function can be used to read the next entry from the user accounting database.

The **getutxid**() function searches for the next entry in the database of which the behaviour is based on the

ut_type field of *id*. If *ut_type* has a value of BOOT_TIME, SHUTDOWN_TIME, OLD_TIME or NEW_TIME, it will return the next entry whose *ut_type* has an equal value. If *ut_type* has a value of USER_PROCESS, INIT_PROCESS, LOGIN_PROCESS or DEAD_PROCESS, it will return the next entry whose *ut_type* has one of the previously mentioned values and whose *ut_id* is equal.

The **getutxline**() function searches for the next entry in the database whose *ut_type* has a value of USER_PROCESS or LOGIN_PROCESS and whose *ut_line* is equal to the same field in *line*.

The **getutxuser**() function searches for the next entry in the database whose *ut_type* has a value of USER_PROCESS and whose *ut_user* is equal to *user*.

The previously mentioned functions will automatically try to open the user accounting database if not already done so. The **setutxdb**() and **setutxent**() functions allow the database to be opened manually, causing the offset within the user accounting database to be rewound. The **endutxent**() function closes the database.

The **setutxent**() database always opens the active sessions database. The **setutxdb**() function opens the database identified by *type*, whose value is either UTXDB_ACTIVE, UTXDB_LASTLOGIN or UTXDB_LOG. It will open a custom file with filename *file* instead of the system-default if *file* is not null. Care must be taken that when using a custom filename, *type* still has to match with the actual format, since each database may use its own file format.

The **pututxline**() function writes record *utmpx* to the system-default user accounting databases. The value of *ut_type* determines which databases are modified.

Entries of type SHUTDOWN_TIME, OLD_TIME and NEW_TIME will only be written to */var/log/utx.log*.

Entries of type USER_PROCESS will also be written to */var/run/utx.active* and */var/log/utx.lastlogin*.

Entries of type DEAD_PROCESS will only be written to */var/log/utx.log* and */var/run/utx.active* if a corresponding USER_PROCESS, INIT_PROCESS or LOGIN_PROCESS entry whose *ut_id* is equal has been found in the latter.

In addition, entries of type BOOT_TIME and SHUTDOWN_TIME will cause all existing entries in */var/run/utx.active* to be discarded.

All entries whose type has not been mentioned previously, are discarded by this implementation of **pututxline**(). This implementation also ignores the value of *ut_tv*.

RETURN VALUES

The **getutxent**(), **getutxid**(), **getutxline**(), and **getutxuser**() functions return a pointer to an *utmpx* structure that matches the mentioned constraints on success or NULL when reaching the end-of-file or when an error occurs.

The **pututxline**() function returns a pointer to an *utmpx* structure containing a copy of the structure written to disk upon success. It returns NULL when the provided *utmpx* is invalid, or *ut_type* has a value of DEAD_PROCESS and an entry with an identifier with a value equal to the field *ut_id* was not found; the global variable *errno* is set to indicate the error.

The **setutxdb**() function returns 0 if the user accounting database was opened successfully. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

In addition to the error conditions described in open(2), fdopen(3), fopen(3), fseek(3), the **pututxline**() function can generate the following errors:

[ESRCH] The value of *ut_type* is DEAD_PROCESS, and the process entry could not be found.

[EINVAL] The value of *ut_type* is not supported by this implementation.

In addition to the error conditions described in fopen(3), the **setutxdb**() function can generate the following errors:

[EINVAL] The *type* argument contains a value not supported by this implementation.

[EFTYPE] The file format is invalid.

SEE ALSO

last(1), write(1), getpid(2), gettimeofday(2), tty(4), ac(8), newsyslog(8), utx(8)

STANDARDS

The **endutxent**(), **getutxent**(), **getutxid**(), **getutxline**() and **setutxent**() functions are expected to conform to IEEE Std 1003.1-2008 ("POSIX.1").

The **pututxline**() function deviates from the standard by writing its records to multiple database files, depending on its *ut_type*. This prevents the need for special utility functions to update the other databases, such as the **updlastlogx**() and **updwtmpx**() functions which are available in other implementations. It also tries to replace DEAD_PROCESS entries in the active sessions database when storing USER_PROCESS entries and no entry with the same value for *ut_id* has been found. The

standard always requires a new entry to be allocated, which could cause an unbounded growth of the database.

The **getutxuser**() and **setutxdb**() functions, the *ut_host* field of the *utmpx* structure and SHUTDOWN_TIME are extensions.

HISTORY

These functions appeared in FreeBSD 9.0. They replaced the <*utmp.h*> interface.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

err, **verr**, **errc**, **verrc**, **errx**, **verrx**, **warn**, **vwarn**, **warnc**, **vwarnc**, **warnx**, **vwarnx**, **err_set_exit**, **err_set_file** - formatted error messages

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <err.h>

void
err(*int eval*, *const char *fmt*, ...);

void
err_set_exit(*void (*exitf)(int)*);

void
err_set_file(*void *vfp*);

void
errc(*int eval*, *int code*, *const char *fmt*, ...);

void
errx(*int eval*, *const char *fmt*, ...);

void
warn(*const char *fmt*, ...);

void
warnc(*int code*, *const char *fmt*, ...);

void
warnx(*const char *fmt*, ...);

#include <stdarg.h>

void
verr(*int eval*, *const char *fmt*, *va_list args*);

void

verrc(*int eval*, *int code*, *const char *fmt*, *va_list args*);

void
verrx(*int eval*, *const char *fmt*, *va_list args*);

void
vwarn(*const char *fmt*, *va_list args*);

void
vwarnc(*int code*, *const char *fmt*, *va_list args*);

void
vwarnx(*const char *fmt*, *va_list args*);

DESCRIPTION

The **err**() and **warn**() family of functions display a formatted error message on the standard error output, or on another file specified using the **err_set_file**() function. In all cases, the last component of the program name, a colon character, and a space are output. If the *fmt* argument is not NULL, the printf(3)-like formatted error message is output. The output is terminated by a newline character.

The **err**(), **errc**(), **verr**(), **verrc**(), **warn**(), **warnc**(), **vwarn**(), and **vwarnc**() functions append an error message obtained from strerror(3) based on a supplied error code value or the global variable *errno*, preceded by another colon and space unless the *fmt* argument is NULL.

In the case of the **errc**(), **verrc**(), **warnc**(), and **vwarnc**() functions, the *code* argument is used to look up the error message.

The **err**(), **verr**(), **warn**(), and **vwarn**() functions use the global variable *errno* to look up the error message.

The **errx**() and **warnx**() functions do not append an error message.

The **err**(), **verr**(), **errc**(), **verrc**(), **errx**(), and **verrx**() functions do not return, but exit with the value of the argument *eval*. It is recommended that the standard values defined in sysexits(3) be used for the value of *eval*. The **err_set_exit**() function can be used to specify a function which is called before exit(3) to perform any necessary cleanup; passing a null function pointer for *exitf* resets the hook to do nothing. The **err_set_file**() function sets the output stream used by the other functions. Its *vfp* argument must be either a pointer to an open stream (possibly already converted to void *) or a null pointer (in which case the output stream is set to standard error).

EXAMPLES

Display the current errno information string and exit:

```
if ((p = malloc(size)) == NULL)
        err(EX_OSERR, NULL);
if ((fd = open(file_name, O_RDONLY, 0)) == -1)
        err(EX_NOINPUT, "%s", file_name);
```

Display an error message and exit:

```
if (tm.tm_hour < START_TIME)
        errx(EX_DATAERR, "too early, wait until %s",
          start_time_string);
```

Warn of an error:

```
if ((fd = open(raw_device, O_RDONLY, 0)) == -1)
        warnx("%s: %s: trying the block device",
          raw_device, strerror(errno));
if ((fd = open(block_device, O_RDONLY, 0)) == -1)
        err(EX_OSFILE, "%s", block_device);
```

Warn of an error without using the global variable *errno*:

```
error = my_function();          /* returns a value from <errno.h> */
if (error != 0)
        warnc(error, "my_function");
```

SEE ALSO

exit(3), fmtmsg(3), printf(3), strerror(3), sysexits(3)

STANDARDS

The **err**() and **warn**() families of functions are BSD extensions. As such they should not be used in truly portable code. Use **strerror**() or similar functions instead.

HISTORY

The **err**() and **warn**() functions first appeared in 4.4BSD. The **err_set_exit**() and **err_set_file**() functions first appeared in FreeBSD 2.1. The **errc**() and **warnc**() functions first appeared in FreeBSD 3.0.

NAME

ethers, **ether_line**, **ether_aton**, **ether_aton_r**, **ether_ntoa**, **ether_ntoa_r**, **ether_ntohost**, **ether_hostton** - Ethernet address conversion and lookup routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <net/ethernet.h>

int
ether_line(*const char *l*, *struct ether_addr *e*, *char *hostname*);

*struct ether_addr **
ether_aton(*const char *a*);

*struct ether_addr **
ether_aton_r(*const char *a*, *struct ether_addr *e*);

*char **
ether_ntoa(*const struct ether_addr *n*);

*char **
ether_ntoa_r(*const struct ether_addr *n*, *char *buf*);

int
ether_ntohost(*char *hostname*, *const struct ether_addr *e*);

int
ether_hostton(*const char *hostname*, *struct ether_addr *e*);

DESCRIPTION

These functions operate on ethernet addresses using an *ether_addr* structure, which is defined in the header file <*net/ethernet.h*>:

```
/*
 * The number of bytes in an ethernet (MAC) address.
 */
```

```
#define ETHER_ADDR_LEN          6

/*
 * Structure of a 48-bit Ethernet address.
 */
struct  ether_addr {
        u_char octet[ETHER_ADDR_LEN];
};
```

The function **ether_line**() scans *l*, an ASCII string in ethers(5) format and sets *e* to the ethernet address specified in the string and *h* to the hostname. This function is used to parse lines from */etc/ethers* into their component parts.

The **ether_aton**() and **ether_aton_r**() functions convert ASCII representation of ethernet addresses into *ether_addr* structures. Likewise, the **ether_ntoa**() and **ether_ntoa_r**() functions convert ethernet addresses specified as *ether_addr* structures into ASCII strings.

The **ether_ntohost**() and **ether_hostton**() functions map ethernet addresses to their corresponding hostnames as specified in the */etc/ethers* database. The **ether_ntohost**() function converts from ethernet address to hostname, and **ether_hostton**() converts from hostname to ethernet address.

RETURN VALUES

The **ether_line**() function returns zero on success and non-zero if it was unable to parse any part of the supplied line *l*. It returns the extracted ethernet address in the supplied *ether_addr* structure *e* and the hostname in the supplied string *h*.

On success, **ether_ntoa**() and **ether_ntoa_r**() functions return a pointer to a string containing an ASCII representation of an ethernet address. If it is unable to convert the supplied *ether_addr* structure, it returns a NULL pointer. **ether_ntoa**() stores the result in a static buffer; **ether_ntoa_r**() stores the result in a user-passed buffer.

Likewise, **ether_aton**() and **ether_aton_r**() return a pointer to an *ether_addr* structure on success and a NULL pointer on failure. **ether_aton**() stores the result in a static buffer; **ether_aton_r**() stores the result in a user-passed buffer.

The **ether_ntohost**() and **ether_hostton**() functions both return zero on success or non-zero if they were unable to find a match in the */etc/ethers* database.

NOTES

The user must ensure that the hostname strings passed to the **ether_line**(), **ether_ntohost**() and

ether_hostton() functions are large enough to contain the returned hostnames.

NIS INTERACTION

If the */etc/ethers* contains a line with a single + in it, the **ether_ntohost**() and **ether_hostton**() functions will attempt to consult the NIS *ethers.byname* and *ethers.byaddr* maps in addition to the data in the */etc/ethers* file.

SEE ALSO

ethers(5), yp(8)

HISTORY

This particular implementation of the **ethers** library functions were written for and first appeared in FreeBSD 2.1. Thread-safe function variants first appeared in FreeBSD 7.0.

BUGS

The **ether_aton**() and **ether_ntoa**() functions returns values that are stored in static memory areas which may be overwritten the next time they are called.

ether_ntoa_r() accepts a character buffer pointer, but not a buffer length. The caller must ensure adequate space is available in the buffer in order to avoid a buffer overflow.

NAME

eui64, **eui64_aton**, **eui64_ntoa**, **eui64_ntohost**, **eui64_hostton** - IEEE EUI-64 conversion and lookup routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/eui64.h>

int
eui64_aton(*const char *a*, *struct eui64 *e*);

int
eui64_ntoa(*const struct eui64 *id*, *char *a*, *size_t len*);

int
eui64_ntohost(*char *hostname*, *size_t len*, *const struct eui64 *id*);

int
eui64_hostton(*const char *hostname*, *struct eui64 *id*);

DESCRIPTION

These functions operate on IEEE EUI-64s using an *eui64* structure, which is defined in the header file <*sys/eui64.h*>:

```
/*
 * The number of bytes in an EUI-64.
 */
#define EUI64_LEN                       8

/*
 * Structure of an IEEE EUI-64.
 */
struct  eui64 {
        u_char octet[EUI64_LEN];
};
```

The **eui64_aton**() function converts an ASCII representation of an EUI-64 into an *eui64* structure.

Likewise, **eui64_ntoa**() converts an EUI-64 specified as an *eui64* structure into an ASCII string.

The **eui64_ntohost**() and **eui64_hostton**() functions map EUI-64s to their corresponding hostnames as specified in the */etc/eui64* database. The **eui64_ntohost**() function converts from EUI-64 to hostname, and **eui64_hostton**() converts from hostname to EUI-64.

RETURN VALUES

On success, **eui64_ntoa**() returns a pointer to a string containing an ASCII representation of an EUI-64. If it is unable to convert the supplied *eui64* structure, it returns a NULL pointer. Likewise, **eui64_aton**() returns a pointer to an *eui64* structure on success and a NULL pointer on failure.

The **eui64_ntohost**() and **eui64_hostton**() functions both return zero on success or non-zero if they were unable to find a match in the */etc/eui64* database.

NOTES

The user must ensure that the hostname strings passed to the **eui64_ntohost**() and **eui64_hostton**() functions are large enough to contain the returned hostnames.

NIS INTERACTION

If the */etc/eui64* contains a line with a single '+' in it, the **eui64_ntohost**() and **eui64_hostton**() functions will attempt to consult the NIS *eui64.byname* and *eui64.byid* maps in addition to the data in the */etc/eui64* file.

SEE ALSO

firewire(4), eui64(5), yp(8)

HISTORY

These functions first appears in FreeBSD 5.3. They are derived from the ethers(3) family of functions.

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp**, **execvP** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char **environ*;

int
execl(*const char *path*, *const char *arg*, ..., *NULL*);

int
execlp(*const char *file*, *const char *arg*, ..., *NULL*);

int
execle(*const char *path*, *const char *arg*, ..., *NULL*, *char *const envp[]*);

int
exect(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
execv(*const char *path*, *char *const argv[]*);

int
execvp(*const char *file*, *char *const argv[]*);

int
execvP(*const char *file*, *const char *search_path*, *char *const argv[]*);

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be

thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), **execvp**(), and **execvP**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**(), **execvp**(), and **execvP**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For **execlp**() and **execvp**(), search path is the path specified in the environment by "PATH" variable. If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin". For **execvP**(), the search path is specified as an argument to the function. In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except ENOEXEC as being ambiguous here, although only the critical error EACCES is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable *errno* restored to the value set by **execve**(). Otherwise, the search will be continued. If the search completes without performing a successful **execve**() or terminating due to an error, these functions will return with the global variable *errno* set to EACCES or ENOENT according to whether at least one file with suitable execute permissions was found.

If the header of a file is not recognized (the attempted **execve**() returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec**() functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was "*:/bin:/usr/bin*". This was changed to remove the current directory to enhance system security.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is not quite historic practice, and has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ETXTBSY, upon which they retried after sleeping for several seconds, and ENOMEM and E2BIG, upon which they returned. They now return for ETXTBSY, and determine existence and executability more carefully. In particular, EACCES for inaccessible directories in the path prefix is no longer confused with EACCES for files with unsuitable execute permissions. In 4.4BSD, they returned upon all errors except EACCES, ENOENT, ENOEXEC and ETXTBSY. This was inferior to the traditional error handling, since it breaks the ignoring of errors for path prefixes and only improves the handling of the unusual ambiguous error EFAULT and the unusual error EIO. The behaviour was changed to match the behaviour of sh(1).

ERRORS

The **execl**(), **execle**(), **execlp**(), **execvp**() and **execvP**() functions may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

The **exect**() and **execv**() functions may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ktrace(2), ptrace(2), environ(7)

STANDARDS

The **execl**(), **execv**(), **execle**(), **execlp**() and **execvp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **execvP**() function first appeared in FreeBSD 5.2.

BUGS

The type of the *argv* and *envp* parameters to **execle**(), **exect**(), **execv**(), **execvp**(), and **execvP**() is a

historical accident and no sane implementation should modify the provided strings. The bogus parameter types trigger false positives from const correctness analyzers. On FreeBSD, the **__DECONST**() macro may be used to work around this limitation.

Due to a fluke of the C standard, on platforms other than FreeBSD the definition of NULL may be the untyped number zero, rather than a *(void *)0* expression. To distinguish the concepts, they are referred to as a "null pointer constant" and a "null pointer", respectively. On exotic computer architectures that FreeBSD does not support, the null pointer constant and null pointer may have a different representation. In general, where this document and others reference a NULL value, they actually imply a null pointer. E.g., for portability to non-FreeBSD operating systems on exotic computer architectures, one may use (char *)NULL in place of NULL when invoking **execl**(), **execle**(), and **execlp**().

NAME

execve, **fexecve** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
execve(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
fexecve(*int fd*, *char *const argv[]*, *char *const envp[]*);

DESCRIPTION

The **execve**() system call transforms the calling process into a new process. The new process is constructed from an ordinary file, whose name is pointed to by *path*, called the *new process file*. The **fexecve**() system call is equivalent to **execve**() except that the file to be executed is determined by the file descriptor *fd* instead of a *path*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialized with zero data; see elf(5) and a.out(5).

An interpreter file begins with a line of the form:

#! *interpreter* [*arg*]

When an interpreter file is **execve**'d, the system actually **execve**'s the specified *interpreter*. If the optional *arg* is specified, it becomes the first argument to the *interpreter*, and the name of the originally **execve**'d file becomes the second argument; otherwise, the name of the originally **execve**'d file becomes the first argument. The original arguments are shifted over to become the subsequent arguments. The zeroth argument is set to the specified *interpreter*.

The argument *argv* is a pointer to a null-terminated array of character pointers to null-terminated character strings. These strings construct the argument list to be made available to the new process. At least one argument must be present in the array; by custom, the first element should be the name of the executed program (for example, the last component of *path*).

The argument *envp* is also a pointer to a null-terminated array of character pointers to null-terminated

strings. A pointer to this array is normally stored in the global variable *environ*. These strings pass information to the new process that is not directly an argument to the command (see environ(7)).

File descriptors open in the calling process image remain open in the new process image, except for those for which the close-on-exec flag is set (see close(2) and fcntl(2)). Descriptors that remain open are unaffected by **execve**(). If any of the standard descriptors (0, 1, and/or 2) are closed at the time **execve**() is called, and the process will gain privilege as a result of set-id semantics, those descriptors will be re-opened automatically. No programs, whether privileged or not, should assume that these descriptors will remain closed across a call to **execve**().

Signals set to be ignored in the calling process are set to be ignored in the new process. Signals which are set to be caught in the calling process image are set to default action in the new process image. Blocked signals remain blocked regardless of changes to the signal action. The signal stack is reset to be undefined (see sigaction(2) for more information).

If the set-user-ID mode bit of the new process image file is set (see chmod(2)), the effective user ID of the new process image is set to the owner ID of the new process image file. If the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. (The effective group ID is the first element of the group list.) The real user ID, real group ID and other group IDs of the new process image remain the same as the calling process image. After any set-user-ID and set-group-ID processing, the effective user ID is recorded as the saved set-user-ID, and the effective group ID is recorded as the saved set-group-ID. These values may be used in changing the effective IDs later (see setuid(2)).

The set-ID bits are not honored if the respective file system has the **nosuid** option enabled or if the new process file is an interpreter file. Syscall tracing is disabled if effective IDs are changed.

The new process also inherits the following attributes from the calling process:

process ID	see getpid(2)
parent process ID	see getppid(2)
process group ID	see getpgrp(2)
access groups	see getgroups(2)
working directory	see chdir(2)
root directory	see chroot(2)
control terminal	see termios(4)
resource usages	see getrusage(2)
interval timers	see getitimer(2)
resource limits	see getrlimit(2)
file mode mask	see umask(2)

signal mask see sigaction(2), sigprocmask(2)

When a program is executed as a result of an **execve**() system call, it is entered as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the number of elements in *argv* (the ‘‘arg count’’) and *argv* points to the array of character pointers to the arguments themselves.

The **fexecve**() ignores the file offset of *fd*. Since execute permission is checked by **fexecve**(), the file descriptor *fd* need not have been opened with the O_EXEC flag. However, if the file to be executed denies read permission for the process preparing to do the exec, the only way to provide the *fd* to **fexecve**() is to use the O_EXEC flag when opening *fd*. Note that the file to be executed can not be open for writing.

RETURN VALUES

As the **execve**() system call overlays the current process image with a new process image the successful call has no process to return to. If **execve**() does return to the calling process an error has occurred; the return value will be -1 and the global variable *errno* is set to indicate the error.

ERRORS

The **execve**() system call will fail and return to the calling process if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOEXEC] When invoking an interpreted script, the length of the first line, inclusive of the **#!** prefix and terminating newline, exceeds MAXSHELLCMDLEN characters.

[ENOENT] The new process file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The new process file is not an ordinary file.

[EACCES] The new process file mode denies execute permission.

[ENOEXEC] The new process file has the appropriate access permission, but has an invalid magic number in its header.

[ETXTBSY] The new process file is a pure procedure (shared text) file that is currently open for writing by some process.

[ENOMEM] The new process requires more virtual memory than is allowed by the imposed maximum (getrlimit(2)).

[E2BIG] The number of bytes in the new process' argument list is larger than the system-imposed limit. This limit is specified by the sysctl(3) MIB variable KERN_ARGMAX.

[EFAULT] The new process file is not as long as indicated by the size values in its header.

[EFAULT] The *path*, *argv*, or *envp* arguments point to an illegal address.

[EIO] An I/O error occurred while reading from the file system.

In addition, the **fexecve**() will fail and return to the calling process if:

[EBADF] The *fd* argument is not a valid file descriptor open for executing.

SEE ALSO

ktrace(1), _exit(2), fork(2), open(2), execl(3), exit(3), sysctl(3), a.out(5), elf(5), fdescfs(5), environ(7), mount(8)

STANDARDS

The **execve**() system call conforms to IEEE Std 1003.1-2001 ("POSIX.1"), with the exception of reopening descriptors 0, 1, and/or 2 in certain circumstances. A future update of the Standard is expected to require this behavior, and it may become the default for non-privileged processes as well. The support for executing interpreted programs is an extension. The **fexecve**() system call conforms to The Open Group Extended API Set 2 specification.

HISTORY

The **execve**() system call appeared in 4.2BSD. The **fexecve**() system call appeared in FreeBSD 8.0.

CAVEATS

If a program is *setuid* to a non-super-user, but is executed when the real *uid* is ‘‘root’’, then the program has some of the powers of a super-user as well.

When executing an interpreted program through **fexecve**(), kernel supplies */dev/fd/n* as a second argument to the interpreter, where *n* is the file descriptor passed in the *fd* argument to **fexecve**(). For this construction to work correctly, the fdescfs(5) filesystem shall be mounted on */dev/fd*.

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp**, **execvP** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char **environ*;

int
execl(*const char *path*, *const char *arg*, ..., *NULL*);

int
execlp(*const char *file*, *const char *arg*, ..., *NULL*);

int
execle(*const char *path*, *const char *arg*, ..., *NULL*, *char *const envp[]*);

int
exect(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
execv(*const char *path*, *char *const argv[]*);

int
execvp(*const char *file*, *char *const argv[]*);

int
execvP(*const char *file*, *const char *search_path*, *char *const argv[]*);

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be

thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), **execvp**(), and **execvP**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**(), **execvp**(), and **execvP**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For **execlp**() and **execvp**(), search path is the path specified in the environment by "PATH" variable. If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin". For **execvP**(), the search path is specified as an argument to the function. In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except ENOEXEC as being ambiguous here, although only the critical error EACCES is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable *errno* restored to the value set by **execve**(). Otherwise, the search will be continued. If the search completes without performing a successful **execve**() or terminating due to an error, these functions will return with the global variable *errno* set to EACCES or ENOENT according to whether at least one file with suitable execute permissions was found.

If the header of a file is not recognized (the attempted **execve**() returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec**() functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was "*:/bin:/usr/bin*". This was changed to remove the current directory to enhance system security.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is not quite historic practice, and has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ETXTBSY, upon which they retried after sleeping for several seconds, and ENOMEM and E2BIG, upon which they returned. They now return for ETXTBSY, and determine existence and executability more carefully. In particular, EACCES for inaccessible directories in the path prefix is no longer confused with EACCES for files with unsuitable execute permissions. In 4.4BSD, they returned upon all errors except EACCES, ENOENT, ENOEXEC and ETXTBSY. This was inferior to the traditional error handling, since it breaks the ignoring of errors for path prefixes and only improves the handling of the unusual ambiguous error EFAULT and the unusual error EIO. The behaviour was changed to match the behaviour of sh(1).

ERRORS

The **execl**(), **execle**(), **execlp**(), **execvp**() and **execvP**() functions may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

The **exect**() and **execv**() functions may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ktrace(2), ptrace(2), environ(7)

STANDARDS

The **execl**(), **execv**(), **execle**(), **execlp**() and **execvp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **execvP**() function first appeared in FreeBSD 5.2.

BUGS

The type of the *argv* and *envp* parameters to **execle**(), **exect**(), **execv**(), **execvp**(), and **execvP**() is a

historical accident and no sane implementation should modify the provided strings. The bogus parameter types trigger false positives from const correctness analyzers. On FreeBSD, the **__DECONST**() macro may be used to work around this limitation.

Due to a fluke of the C standard, on platforms other than FreeBSD the definition of NULL may be the untyped number zero, rather than a *(void *)0* expression. To distinguish the concepts, they are referred to as a "null pointer constant" and a "null pointer", respectively. On exotic computer architectures that FreeBSD does not support, the null pointer constant and null pointer may have a different representation. In general, where this document and others reference a NULL value, they actually imply a null pointer. E.g., for portability to non-FreeBSD operating systems on exotic computer architectures, one may use (char *)NULL in place of NULL when invoking **execl**(), **execle**(), and **execlp**().

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp**, **execvP** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char **environ*;

int
execl(*const char *path*, *const char *arg*, ..., *NULL*);

int
execlp(*const char *file*, *const char *arg*, ..., *NULL*);

int
execle(*const char *path*, *const char *arg*, ..., *NULL*, *char *const envp[]*);

int
exect(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
execv(*const char *path*, *char *const argv[]*);

int
execvp(*const char *file*, *char *const argv[]*);

int
execvP(*const char *file*, *const char *search_path*, *char *const argv[]*);

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be

thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), **execvp**(), and **execvP**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**(), **execvp**(), and **execvP**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For **execlp**() and **execvp**(), search path is the path specified in the environment by "PATH" variable. If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin". For **execvP**(), the search path is specified as an argument to the function. In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except ENOEXEC as being ambiguous here, although only the critical error EACCES is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable *errno* restored to the value set by **execve**(). Otherwise, the search will be continued. If the search completes without performing a successful **execve**() or terminating due to an error, these functions will return with the global variable *errno* set to EACCES or ENOENT according to whether at least one file with suitable execute permissions was found.

If the header of a file is not recognized (the attempted **execve**() returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec**() functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was "*:/bin:/usr/bin*". This was changed to remove the current directory to enhance system security.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is not quite historic practice, and has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ETXTBSY, upon which they retried after sleeping for several seconds, and ENOMEM and E2BIG, upon which they returned. They now return for ETXTBSY, and determine existence and executability more carefully. In particular, EACCES for inaccessible directories in the path prefix is no longer confused with EACCES for files with unsuitable execute permissions. In 4.4BSD, they returned upon all errors except EACCES, ENOENT, ENOEXEC and ETXTBSY. This was inferior to the traditional error handling, since it breaks the ignoring of errors for path prefixes and only improves the handling of the unusual ambiguous error EFAULT and the unusual error EIO. The behaviour was changed to match the behaviour of sh(1).

ERRORS

The **execl**(), **execle**(), **execlp**(), **execvp**() and **execvP**() functions may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

The **exect**() and **execv**() functions may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ktrace(2), ptrace(2), environ(7)

STANDARDS

The **execl**(), **execv**(), **execle**(), **execlp**() and **execvp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **execvP**() function first appeared in FreeBSD 5.2.

BUGS

The type of the *argv* and *envp* parameters to **execle**(), **exect**(), **execv**(), **execvp**(), and **execvP**() is a

historical accident and no sane implementation should modify the provided strings. The bogus parameter types trigger false positives from const correctness analyzers. On FreeBSD, the **__DECONST**() macro may be used to work around this limitation.

Due to a fluke of the C standard, on platforms other than FreeBSD the definition of NULL may be the untyped number zero, rather than a *(void *)0* expression. To distinguish the concepts, they are referred to as a "null pointer constant" and a "null pointer", respectively. On exotic computer architectures that FreeBSD does not support, the null pointer constant and null pointer may have a different representation. In general, where this document and others reference a NULL value, they actually imply a null pointer. E.g., for portability to non-FreeBSD operating systems on exotic computer architectures, one may use (char *)NULL in place of NULL when invoking **execl**(), **execle**(), and **execlp**().

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp**, **execvP** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char **environ*;

int
execl(*const char *path*, *const char *arg*, ..., *NULL*);

int
execlp(*const char *file*, *const char *arg*, ..., *NULL*);

int
execle(*const char *path*, *const char *arg*, ..., *NULL*, *char *const envp[]*);

int
exect(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
execv(*const char *path*, *char *const argv[]*);

int
execvp(*const char *file*, *char *const argv[]*);

int
execvP(*const char *file*, *const char *search_path*, *char *const argv[]*);

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be

thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), **execvp**(), and **execvP**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**(), **execvp**(), and **execvP**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For **execlp**() and **execvp**(), search path is the path specified in the environment by "PATH" variable. If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin". For **execvP**(), the search path is specified as an argument to the function. In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except ENOEXEC as being ambiguous here, although only the critical error EACCES is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable *errno* restored to the value set by **execve**(). Otherwise, the search will be continued. If the search completes without performing a successful **execve**() or terminating due to an error, these functions will return with the global variable *errno* set to EACCES or ENOENT according to whether at least one file with suitable execute permissions was found.

If the header of a file is not recognized (the attempted **execve**() returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec**() functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was "*:/bin:/usr/bin*". This was changed to remove the current directory to enhance system security.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is not quite historic practice, and has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ETXTBSY, upon which they retried after sleeping for several seconds, and ENOMEM and E2BIG, upon which they returned. They now return for ETXTBSY, and determine existence and executability more carefully. In particular, EACCES for inaccessible directories in the path prefix is no longer confused with EACCES for files with unsuitable execute permissions. In 4.4BSD, they returned upon all errors except EACCES, ENOENT, ENOEXEC and ETXTBSY. This was inferior to the traditional error handling, since it breaks the ignoring of errors for path prefixes and only improves the handling of the unusual ambiguous error EFAULT and the unusual error EIO. The behaviour was changed to match the behaviour of sh(1).

ERRORS

The **execl**(), **execle**(), **execlp**(), **execvp**() and **execvP**() functions may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

The **exect**() and **execv**() functions may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ktrace(2), ptrace(2), environ(7)

STANDARDS

The **execl**(), **execv**(), **execle**(), **execlp**() and **execvp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **execvP**() function first appeared in FreeBSD 5.2.

BUGS

The type of the *argv* and *envp* parameters to **execle**(), **exect**(), **execv**(), **execvp**(), and **execvP**() is a

historical accident and no sane implementation should modify the provided strings. The bogus parameter types trigger false positives from const correctness analyzers. On FreeBSD, the **__DECONST**() macro may be used to work around this limitation.

Due to a fluke of the C standard, on platforms other than FreeBSD the definition of NULL may be the untyped number zero, rather than a *(void *)0* expression. To distinguish the concepts, they are referred to as a "null pointer constant" and a "null pointer", respectively. On exotic computer architectures that FreeBSD does not support, the null pointer constant and null pointer may have a different representation. In general, where this document and others reference a NULL value, they actually imply a null pointer. E.g., for portability to non-FreeBSD operating systems on exotic computer architectures, one may use (char *)NULL in place of NULL when invoking **execl**(), **execle**(), and **execlp**().

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp**, **execvP** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char **environ*;

int
execl(*const char *path*, *const char *arg*, ..., *NULL*);

int
execlp(*const char *file*, *const char *arg*, ..., *NULL*);

int
execle(*const char *path*, *const char *arg*, ..., *NULL*, *char *const envp[]*);

int
exect(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
execv(*const char *path*, *char *const argv[]*);

int
execvp(*const char *file*, *char *const argv[]*);

int
execvP(*const char *file*, *const char *search_path*, *char *const argv[]*);

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be

thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), **execvp**(), and **execvP**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**(), **execvp**(), and **execvP**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For **execlp**() and **execvp**(), search path is the path specified in the environment by "PATH" variable. If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin". For **execvP**(), the search path is specified as an argument to the function. In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except ENOEXEC as being ambiguous here, although only the critical error EACCES is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable *errno* restored to the value set by **execve**(). Otherwise, the search will be continued. If the search completes without performing a successful **execve**() or terminating due to an error, these functions will return with the global variable *errno* set to EACCES or ENOENT according to whether at least one file with suitable execute permissions was found.

If the header of a file is not recognized (the attempted **execve**() returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec**() functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was "*:/bin:/usr/bin*". This was changed to remove the current directory to enhance system security.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is not quite historic practice, and has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ETXTBSY, upon which they retried after sleeping for several seconds, and ENOMEM and E2BIG, upon which they returned. They now return for ETXTBSY, and determine existence and executability more carefully. In particular, EACCES for inaccessible directories in the path prefix is no longer confused with EACCES for files with unsuitable execute permissions. In 4.4BSD, they returned upon all errors except EACCES, ENOENT, ENOEXEC and ETXTBSY. This was inferior to the traditional error handling, since it breaks the ignoring of errors for path prefixes and only improves the handling of the unusual ambiguous error EFAULT and the unusual error EIO. The behaviour was changed to match the behaviour of sh(1).

ERRORS

The **execl**(), **execle**(), **execlp**(), **execvp**() and **execvP**() functions may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

The **exect**() and **execv**() functions may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ktrace(2), ptrace(2), environ(7)

STANDARDS

The **execl**(), **execv**(), **execle**(), **execlp**() and **execvp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **execvP**() function first appeared in FreeBSD 5.2.

BUGS

The type of the *argv* and *envp* parameters to **execle**(), **exect**(), **execv**(), **execvp**(), and **execvP**() is a

historical accident and no sane implementation should modify the provided strings. The bogus parameter types trigger false positives from const correctness analyzers. On FreeBSD, the **__DECONST**() macro may be used to work around this limitation.

Due to a fluke of the C standard, on platforms other than FreeBSD the definition of NULL may be the untyped number zero, rather than a *(void *)0* expression. To distinguish the concepts, they are referred to as a "null pointer constant" and a "null pointer", respectively. On exotic computer architectures that FreeBSD does not support, the null pointer constant and null pointer may have a different representation. In general, where this document and others reference a NULL value, they actually imply a null pointer. E.g., for portability to non-FreeBSD operating systems on exotic computer architectures, one may use (char *)NULL in place of NULL when invoking **execl**(), **execle**(), and **execlp**().

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp**, **execvP** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char **environ*;

int
execl(*const char *path*, *const char *arg*, ..., *NULL*);

int
execlp(*const char *file*, *const char *arg*, ..., *NULL*);

int
execle(*const char *path*, *const char *arg*, ..., *NULL*, *char *const envp[]*);

int
exect(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
execv(*const char *path*, *char *const argv[]*);

int
execvp(*const char *file*, *char *const argv[]*);

int
execvP(*const char *file*, *const char *search_path*, *char *const argv[]*);

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be

thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), **execvp**(), and **execvP**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**(), **execvp**(), and **execvP**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For **execlp**() and **execvp**(), search path is the path specified in the environment by "PATH" variable. If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin". For **execvP**(), the search path is specified as an argument to the function. In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except ENOEXEC as being ambiguous here, although only the critical error EACCES is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable *errno* restored to the value set by **execve**(). Otherwise, the search will be continued. If the search completes without performing a successful **execve**() or terminating due to an error, these functions will return with the global variable *errno* set to EACCES or ENOENT according to whether at least one file with suitable execute permissions was found.

If the header of a file is not recognized (the attempted **execve**() returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec**() functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was "*:/bin:/usr/bin*". This was changed to remove the current directory to enhance system security.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is not quite historic practice, and has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ETXTBSY, upon which they retried after sleeping for several seconds, and ENOMEM and E2BIG, upon which they returned. They now return for ETXTBSY, and determine existence and executability more carefully. In particular, EACCES for inaccessible directories in the path prefix is no longer confused with EACCES for files with unsuitable execute permissions. In 4.4BSD, they returned upon all errors except EACCES, ENOENT, ENOEXEC and ETXTBSY. This was inferior to the traditional error handling, since it breaks the ignoring of errors for path prefixes and only improves the handling of the unusual ambiguous error EFAULT and the unusual error EIO. The behaviour was changed to match the behaviour of sh(1).

ERRORS

The **execl**(), **execle**(), **execlp**(), **execvp**() and **execvP**() functions may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

The **exect**() and **execv**() functions may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ktrace(2), ptrace(2), environ(7)

STANDARDS

The **execl**(), **execv**(), **execle**(), **execlp**() and **execvp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **execvP**() function first appeared in FreeBSD 5.2.

BUGS

The type of the *argv* and *envp* parameters to **execle**(), **exect**(), **execv**(), **execvp**(), and **execvP**() is a

historical accident and no sane implementation should modify the provided strings. The bogus parameter types trigger false positives from const correctness analyzers. On FreeBSD, the **__DECONST**() macro may be used to work around this limitation.

Due to a fluke of the C standard, on platforms other than FreeBSD the definition of NULL may be the untyped number zero, rather than a *(void *)0* expression. To distinguish the concepts, they are referred to as a "null pointer constant" and a "null pointer", respectively. On exotic computer architectures that FreeBSD does not support, the null pointer constant and null pointer may have a different representation. In general, where this document and others reference a NULL value, they actually imply a null pointer. E.g., for portability to non-FreeBSD operating systems on exotic computer architectures, one may use (char *)NULL in place of NULL when invoking **execl**(), **execle**(), and **execlp**().

NAME

execl, **execlp**, **execle**, **exect**, **execv**, **execvp**, **execvP** - execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char **environ*;

int
execl(*const char *path*, *const char *arg*, ..., *NULL*);

int
execlp(*const char *file*, *const char *arg*, ..., *NULL*);

int
execle(*const char *path*, *const char *arg*, ..., *NULL*, *char *const envp[]*);

int
exect(*const char *path*, *char *const argv[]*, *char *const envp[]*);

int
execv(*const char *path*, *char *const argv[]*);

int
execvp(*const char *file*, *char *const argv[]*);

int
execvP(*const char *file*, *const char *search_path*, *char *const argv[]*);

DESCRIPTION

The **exec** family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for the function execve(2). (See the manual page for execve(2) for detailed information about the replacement of the current process.)

The initial argument for these functions is the pathname of a file which is to be executed.

The *const char *arg* and subsequent ellipses in the **execl**(), **execlp**(), and **execle**() functions can be

thought of as *arg0*, *arg1*, ..., *argn*. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments *must* be terminated by a NULL pointer.

The **exect**(), **execv**(), **execvp**(), and **execvP**() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers **must** be terminated by a NULL pointer.

The **execle**() and **exect**() functions also specify the environment of the executed process by following the NULL pointer that terminates the list of arguments in the argument list or the pointer to the argv array with an additional argument. This additional argument is an array of pointers to null-terminated strings and *must* be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable *environ* in the current process.

Some of these functions have special semantics.

The functions **execlp**(), **execvp**(), and **execvP**() will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash "/" character. For **execlp**() and **execvp**(), search path is the path specified in the environment by "PATH" variable. If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin". For **execvP**(), the search path is specified as an argument to the function. In addition, certain errors are treated specially.

If an error is ambiguous (for simplicity, we shall consider all errors except ENOEXEC as being ambiguous here, although only the critical error EACCES is really ambiguous), then these functions will act as if they stat the file to determine whether the file exists and has suitable execute permissions. If it does, they will return immediately with the global variable *errno* restored to the value set by **execve**(). Otherwise, the search will be continued. If the search completes without performing a successful **execve**() or terminating due to an error, these functions will return with the global variable *errno* set to EACCES or ENOENT according to whether at least one file with suitable execute permissions was found.

If the header of a file is not recognized (the attempted **execve**() returned ENOEXEC), these functions will execute the shell with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

The function **exect**() executes a file with the program tracing facilities enabled (see ptrace(2)).

RETURN VALUES

If any of the **exec**() functions returns, an error will have occurred. The return value is -1, and the global variable *errno* will be set to indicate the error.

FILES

/bin/sh The shell.

COMPATIBILITY

Historically, the default path for the **execlp**() and **execvp**() functions was "*:/bin:/usr/bin*". This was changed to remove the current directory to enhance system security.

The behavior of **execlp**() and **execvp**() when errors occur while attempting to execute the file is not quite historic practice, and has not traditionally been documented and is not specified by the POSIX standard.

Traditionally, the functions **execlp**() and **execvp**() ignored all errors except for the ones described above and ETXTBSY, upon which they retried after sleeping for several seconds, and ENOMEM and E2BIG, upon which they returned. They now return for ETXTBSY, and determine existence and executability more carefully. In particular, EACCES for inaccessible directories in the path prefix is no longer confused with EACCES for files with unsuitable execute permissions. In 4.4BSD, they returned upon all errors except EACCES, ENOENT, ENOEXEC and ETXTBSY. This was inferior to the traditional error handling, since it breaks the ignoring of errors for path prefixes and only improves the handling of the unusual ambiguous error EFAULT and the unusual error EIO. The behaviour was changed to match the behaviour of sh(1).

ERRORS

The **execl**(), **execle**(), **execlp**(), **execvp**() and **execvP**() functions may fail and set *errno* for any of the errors specified for the library functions execve(2) and malloc(3).

The **exect**() and **execv**() functions may fail and set *errno* for any of the errors specified for the library function execve(2).

SEE ALSO

sh(1), execve(2), fork(2), ktrace(2), ptrace(2), environ(7)

STANDARDS

The **execl**(), **execv**(), **execle**(), **execlp**() and **execvp**() functions conform to IEEE Std 1003.1-1988 ("POSIX.1"). The **execvP**() function first appeared in FreeBSD 5.2.

BUGS

The type of the *argv* and *envp* parameters to **execle**(), **exect**(), **execv**(), **execvp**(), and **execvP**() is a

historical accident and no sane implementation should modify the provided strings. The bogus parameter types trigger false positives from const correctness analyzers. On FreeBSD, the **__DECONST**() macro may be used to work around this limitation.

Due to a fluke of the C standard, on platforms other than FreeBSD the definition of NULL may be the untyped number zero, rather than a *(void *)0* expression. To distinguish the concepts, they are referred to as a "null pointer constant" and a "null pointer", respectively. On exotic computer architectures that FreeBSD does not support, the null pointer constant and null pointer may have a different representation. In general, where this document and others reference a NULL value, they actually imply a null pointer. E.g., for portability to non-FreeBSD operating systems on exotic computer architectures, one may use (char *)NULL in place of NULL when invoking **execl**(), **execle**(), and **execlp**().

NAME

exit, **_Exit** - perform normal program termination

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

void
exit(*int status*);

void
_Exit(*int status*);

DESCRIPTION

The **exit**() and **_Exit**() functions terminate a process.

Before termination, **exit**() performs the following functions in the order listed:

1. Call the functions registered with the atexit(3) function, in the reverse order of their registration.

2. Flush all open output streams.

3. Close all open streams.

4. Unlink all files created with the tmpfile(3) function.

The **_Exit**() function terminates without calling the functions registered with the atexit(3) function, and may or may not perform the other actions listed. Both functions make the low-order eight bits of the *status* argument available to a parent process which has called a wait(2)-family function.

The C Standard (ISO/IEC 9899:1999 ("ISO C99")) defines the values 0, EXIT_SUCCESS, and EXIT_FAILURE as possible values of *status*. Cooperating processes may use other values; in a program which might be called by a mail transfer agent, the values described in sysexits(3) may be used to provide more information to the parent process.

Note that **exit**() does nothing to prevent bottomless recursion should a function registered using atexit(3) itself call **exit**(). Such functions must call **_Exit**() instead (although this has other effects as well which

may not be desired).

RETURN VALUES

The **exit**() and **_Exit**() functions never return.

SEE ALSO

_exit(2), abort2(2), wait(2), at_quick_exit(3), atexit(3), intro(3), quick_exit(3), sysexits(3), tmpfile(3)

STANDARDS

The **exit**() and **_Exit**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

bzero, **explicit_bzero** - write zeroes to a byte string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <strings.h>

void
bzero(*void *b*, *size_t len*);

void
explicit_bzero(*void *b*, *size_t len*);

DESCRIPTION

The **bzero**() function writes *len* zero bytes to the string *b*. If *len* is zero, **bzero**() does nothing.

The **explicit_bzero**() variant behaves the same, but will not be removed by a compiler's dead store optimization pass, making it useful for clearing sensitive memory such as a password.

SEE ALSO

memset(3), swab(3)

HISTORY

A **bzero**() function appeared in 4.3BSD. Its prototype existed previously in <*string.h*> before it was moved to <*strings.h*> for IEEE Std 1003.1-2001 ("POSIX.1") compliance.

The **explicit_bzero**() function first appeared in OpenBSD 5.5 and FreeBSD 11.0.

IEEE Std 1003.1-2008 ("POSIX.1") removes the specification of **bzero**() and it is marked as LEGACY in IEEE Std 1003.1-2004 ("POSIX.1"). For portability with other systems new programs should use memset(3).

NAME

extattr_get_fd, **extattr_set_fd**, **extattr_delete_fd**, **extattr_list_fd**, **extattr_get_file**, **extattr_set_file**, **extattr_delete_file**, **extattr_list_file**, **extattr_get_link**, **extattr_set_link**, **extattr_delete_link**, **extattr_list_link** - system calls to manipulate VFS extended attributes

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/extattr.h>

ssize_t
extattr_get_fd(*int fd*, *int attrnamespace*, *const char *attrname*, *void *data*, *size_t nbytes*);

ssize_t
extattr_set_fd(*int fd*, *int attrnamespace*, *const char *attrname*, *const void *data*, *size_t nbytes*);

int
extattr_delete_fd(*int fd*, *int attrnamespace*, *const char *attrname*);

ssize_t
extattr_list_fd(*int fd*, *int attrnamespace*, *void *data*, *size_t nbytes*);

ssize_t
extattr_get_file(*const char *path*, *int attrnamespace*, *const char *attrname*, *void *data*, *size_t nbytes*);

ssize_t
extattr_set_file(*const char *path*, *int attrnamespace*, *const char *attrname*, *const void *data*, *size_t nbytes*);

int
extattr_delete_file(*const char *path*, *int attrnamespace*, *const char *attrname*);

ssize_t
extattr_list_file(*const char *path*, *int attrnamespace*, *void *data*, *size_t nbytes*);

ssize_t
extattr_get_link(*const char *path*, *int attrnamespace*, *const char *attrname*, *void *data*, *size_t nbytes*);

ssize_t
extattr_set_link(*const char *path*, *int attrnamespace*, *const char *attrname*, *const void *data*, *size_t nbytes*);

int
extattr_delete_link(*const char *path*, *int attrnamespace*, *const char *attrname*);

ssize_t
extattr_list_link(*const char *path*, *int attrnamespace*, *void *data*, *size_t nbytes*);

DESCRIPTION

Named extended attributes are meta-data associated with vnodes representing files and directories. They exist as "name=value" pairs within a set of namespaces.

The **extattr_get_file**() system call retrieves the value of the specified extended attribute into a buffer pointed to by *data* of size *nbytes*. The **extattr_set_file**() system call sets the value of the specified extended attribute to the data described by *data*. The **extattr_delete_file**() system call deletes the extended attribute specified. The **extattr_list_file**() returns a list of attributes present in the requested namespace. Each list entry consists of a single byte containing the length of the attribute name, followed by the attribute name. The attribute name is not terminated by ASCII 0 (nul). The **extattr_get_file**(), and **extattr_list_file**() calls consume the *data* and *nbytes* arguments in the style of read(2); **extattr_set_file**() consumes these arguments in the style of write(2).

If *data* is NULL in a call to **extattr_get_file**() and **extattr_list_file**() then the size of defined extended attribute data will be returned, rather than the quantity read, permitting applications to test the size of the data without performing a read. The **extattr_delete_link**(), **extattr_get_link**(), and **extattr_set_link**() system calls behave in the same way as their _file counterparts, except that they do not follow symlinks.

The **extattr_get_fd**(), **extattr_set_fd**(), **extattr_delete_fd**(), and **extattr_list_fd**(), calls are identical to their "_file" counterparts except for the first argument. The "_fd" functions take a file descriptor, while the "_file" functions take a path. Both arguments describe a file associated with the extended attribute that should be manipulated.

The following arguments are common to all the system calls described here:

attrnamespace the namespace in which the extended attribute resides; see extattr(9)

attrname the name of the extended attribute

Named extended attribute semantics vary by file system implementing the call. Not all operations may

be supported for a particular attribute. Additionally, the format of the data in *data* is attribute-specific.

For more information on named extended attributes, please see extattr(9).

CAVEAT

This interface is under active development, and as such is subject to change as applications are adapted to use it. Developers are discouraged from relying on its stability.

RETURN VALUES

If successful, the **extattr_get_file**(), **extattr_set_file**(), and **extattr_list_file**() calls return the number of bytes that were read or written from the *data*, respectively, or if *data* was NULL, then **extattr_get_file**() and **extattr_list_file**() return the number of bytes available to read. If any of the calls are unsuccessful, the value -1 is returned and the global variable *errno* is set to indicate the error.

The **extattr_delete_file**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be returned by the system calls themselves. Additionally, the file system implementing the call may return any other errors it desires.

[EFAULT] The *attrnamespace* and *attrname* arguments, or the memory range defined by *data* and *nbytes* point outside the process's allocated address space.

[ENAMETOOLONG]
The attribute name was longer than EXTATTR_MAXNAMELEN.

The **extattr_get_fd**(), **extattr_set_fd**(), **extattr_delete_fd**(), and **extattr_list_fd**() system calls may also fail if:

[EBADF] The file descriptor referenced by *fd* was invalid.

Additionally, the **extattr_get_file**(), **extattr_set_file**(), and **extattr_delete_file**() calls may also fail due to the following errors:

[ENOATTR] The requested attribute was not defined for this file.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]

A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] A component of the path name that must exist does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

SEE ALSO

extattr(3), getextattr(8), setextattr(8), extattr(9), VOP_GETEXTATTR(9), VOP_SETEXTATTR(9)

HISTORY

Extended attribute support was developed as part of the TrustedBSD Project, and introduced in FreeBSD 5.0. It was developed to support security extensions requiring additional labels to be associated with each file or directory.

BUGS

In earlier versions of this API, passing an empty string for the attribute name to **extattr_get_fd**(), **extattr_get_file**(), or **extattr_get_link**() would return the list of attributes defined for the target object. This interface has been deprecated in preference to using the explicit list API, and should not be used.

NAME

fclose, **fdclose**, **fcloseall** - close a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
fclose(*FILE *stream*);

int
fdclose(*FILE *stream*, *int *fdp*);

void
fcloseall(*void*);

DESCRIPTION

The **fclose**() function dissociates the named *stream* from its underlying file or set of functions. If the stream was being used for output, any buffered data is written first, using fflush(3).

The **fdclose**() function is equivalent to **fclose**() except that it does not close the underlying file descriptor. If *fdp* is not NULL, the file descriptor will be written to it. If the *fdp* argument will be different then NULL the file descriptor will be returned in it, If the stream does not have an associated file descriptor, *fdp* will be set to -1. This type of stream is created with functions such as fmemopen(3), funopen(3), or open_memstream(3).

The **fcloseall**() function calls **fclose**() on all open streams.

RETURN VALUES

fcloseall() does not return a value.

Upon successful completion the **fclose**() and **fdclose**() functions return 0. Otherwise, EOF is returned and the global variable *errno* is set to indicate the error.

ERRORS

fdclose() fails if:

[EOPNOTSUPP] The stream does not have an associated file descriptor.

The **fclose**() and **fdclose**() functions may also fail and set *errno* for any of the errors specified for fflush(3).

The **fclose**() function may also fail and set errno for any of the errors specified for close(2).

NOTES

The **fclose**() and **fdclose**() functions do not handle NULL arguments in the *stream* variable; this will result in a segmentation violation. This is intentional. It makes it easier to make sure programs written under FreeBSD are bug free. This behaviour is an implementation detail, and programs should not rely upon it.

SEE ALSO

close(2), fflush(3), fopen(3), setbuf(3)

STANDARDS

The **fclose**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

HISTORY

The **fcloseall**() function first appeared in FreeBSD 7.0.

The **fdclose**() function first appeared in FreeBSD 11.0.

NAME

fcntl - file control

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>

int
fcntl(*int fd*, *int cmd*, ...);

DESCRIPTION

The **fcntl**() system call provides for control over descriptors. The argument *fd* is a descriptor to be operated on by *cmd* as described below. Depending on the value of *cmd*, **fcntl**() can take an additional third argument *int arg*.

F_DUPFD Return a new descriptor as follows:

- Lowest numbered available descriptor greater than or equal to *arg*.
- Same object references as the original descriptor.
- New descriptor shares the same file offset if the object was a file.
- Same access mode (read, write or read/write).
- Same file status flags (i.e., both file descriptors share the same file status flags).
- The close-on-exec flag FD_CLOEXEC associated with the new file descriptor is cleared, so the file descriptor is to remain open across execve(2) system calls.

F_DUPFD_CLOEXEC Like F_DUPFD, but the FD_CLOEXEC flag associated with the new file descriptor is set, so the file descriptor is closed when execve(2) system call executes.

F_DUP2FD It is functionally equivalent to

dup2(fd, arg)

F_DUP2FD_CLOEXEC Like F_DUP2FD, but the FD_CLOEXEC flag associated with the new file descriptor is set.

The F_DUP2FD and F_DUP2FD_CLOEXEC constants are not portable, so they should not be used if portability is needed. Use **dup2**() instead of F_DUP2FD.

F_GETFD — Get the close-on-exec flag associated with the file descriptor *fd* as FD_CLOEXEC. If the returned value ANDed with FD_CLOEXEC is 0, the file will remain open across **exec**(), otherwise the file will be closed upon execution of **exec**() (*arg* is ignored).

F_SETFD — Set the close-on-exec flag associated with *fd* to *arg*, where *arg* is either 0 or FD_CLOEXEC, as described above.

F_GETFL — Get descriptor status flags, as described below (*arg* is ignored).

F_SETFL — Set descriptor status flags to *arg*.

F_GETOWN — Get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values (*arg* is ignored).

F_SETOWN — Set the process or process group to receive SIGIO and SIGURG signals; process groups are specified by supplying *arg* as negative, otherwise *arg* is interpreted as a process ID.

F_READAHEAD — Set or clear the read ahead amount for sequential access to the third argument, *arg*, which is rounded up to the nearest block size. A zero value in *arg* turns off read ahead, a negative value restores the system default.

F_RDAHEAD — Equivalent to Darwin counterpart which sets read ahead amount of 128KB when the third argument, *arg* is non-zero. A zero value in *arg* turns off read ahead.

The flags for the F_GETFL and F_SETFL flags are as follows:

O_NONBLOCK — Non-blocking I/O; if no data is available to a read(2) system call, or if a write(2) operation would block, the read or write call returns -1 with the error EAGAIN.

O_APPEND — Force each write to append at the end of file; corresponds to the O_APPEND flag of open(2).

O_DIRECT — Minimize or eliminate the cache effects of reading and writing. The system will

attempt to avoid caching the data you read or write. If it cannot avoid caching the data, it will minimize the impact the data has on the cache. Use of this flag can drastically reduce performance if not used with care.

O_ASYNC Enable the SIGIO signal to be sent to the process group when I/O is possible, e.g., upon availability of data to be read.

Several commands are available for doing advisory file locking; they all operate on the following structure:

```
struct flock {
        off_t    l_start;  /* starting offset */
        off_t    l_len;            /* len = 0 means until end of file */
        pid_t    l_pid;            /* lock owner */
        short    l_type;           /* lock type: read/write, etc. */
        short    l_whence;         /* type of l_start */
        int      l_sysid;  /* remote system id or zero for local */
};
```

The commands available for advisory record locking are as follows:

F_GETLK Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock* (see above). The information retrieved overwrites the information passed to **fcntl**() in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure is left unchanged by this system call except for the lock type which is set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to a *struct flock* (see above). F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks, (F_WRLCK), as well as remove either type of lock (F_UNLCK). If a shared or exclusive lock cannot be set, **fcntl**() returns immediately with EAGAIN.

F_SETLKW This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process waits until the request can be satisfied. If a signal that is to be caught is received while **fcntl**() is waiting for a region, the **fcntl**() will be interrupted if the signal handler has not specified the SA_RESTART (see sigaction(2)).

When a shared lock has been set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened

with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file was not opened with write access.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. If *l_len* is negative, *l_start* means end edge of the region. The *l_pid* and *l_sysid* fields are only used with F_GETLK to return the process ID of the process holding a blocking lock and the system ID of the system that owns that process. Locks created by the local system will have a system ID of zero. After a successful F_GETLK request, the value of *l_whence* is SEEK_SET.

Locks may start and extend beyond the current end of a file, but may not start or extend before the beginning of the file. A lock is set to extend to the largest possible value of the file offset for that file if *l_len* is set to zero. If *l_whence* and *l_start* point to the beginning of the file, and *l_len* is zero, the entire file is locked. If an application wishes only to do entire file locking, the flock(2) system call is much more efficient.

There is at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region is replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request fails or blocks respectively when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

The queuing for F_SETLKW requests on local files is fair; that is, while the thread is blocked, subsequent requests conflicting with its requests will not be granted, even if these requests do not conflict with existing locks.

This interface follows the completely stupid semantics of System V and IEEE Std 1003.1-1988 ("POSIX.1") that require that all locks associated with a file for a given process are removed when *any* file descriptor for that file is closed by that process. This semantic means that applications must be aware of any files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls getpwnam(3) to retrieve a record, the lock will be lost because getpwnam(3) opens, reads, and closes the password database. The database close will release all locks that the process has associated with the database, even if the library routine never requested a lock on the database. Another minor semantic problem with

this interface is that locks are not inherited by a child process created using the fork(2) system call. The flock(2) interface has much more rational last close semantics and allows locks to be inherited by child processes. The flock(2) system call is recommended for applications that want to ensure the integrity of their locks when using library routines or wish to pass locks to their children.

The **fcntl**(), flock(2), and lockf(3) locks are compatible. Processes using different locking interfaces can cooperate over the same file safely. However, only one of such interfaces should be used within the same process. If a file is locked by a process through flock(2), any record within the file will be seen as locked from the viewpoint of another process using **fcntl**() or lockf(3), and vice versa. Note that **fcntl**(*F_GETLK*) returns -1 in *l_pid* if the process holding a blocking lock previously locked the file descriptor by flock(2).

All locks associated with a file for a given process are removed when the process terminates.

All locks obtained before a call to execve(2) remain in effect until the new program releases them. If the new program does not know about the locks, they will not be released until the program exits.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. This implementation detects that sleeping until a locked region is unlocked would cause a deadlock and fails with an EDEADLK error.

RETURN VALUES

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_DUP2FD	A file descriptor equal to *arg*.
F_GETFD	Value of flag (only the low-order bit is defined).
F_GETFL	Value of flags.
F_GETOWN	Value of file descriptor owner.
other	Value other than -1.

Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **fcntl**() system call will fail if:

[EAGAIN] The argument *cmd* is F_SETLK, the type of lock (*l_type*) is a shared lock (F_RDLCK) or exclusive lock (F_WRLCK), and the segment of a file to be locked is already exclusive-locked by another process; or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

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

The argument *cmd* is F_DUP2FD, and *arg* is not a valid file descriptor.

The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is a shared lock (F_RDLCK), and *fd* is not a valid file descriptor open for reading.

The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (*l_type*) is an exclusive lock (F_WRLCK), and *fd* is not a valid file descriptor open for writing.

[EDEADLK] The argument *cmd* is F_SETLKW, and a deadlock condition was detected.

[EINTR] The argument *cmd* is F_SETLKW, and the system call was interrupted by a signal.

[EINVAL] The *cmd* argument is F_DUPFD and *arg* is negative or greater than the maximum allowable number (see getdtablesize(2)).

The argument *cmd* is F_GETLK, F_SETLK or F_SETLKW and the data to which *arg* points is not valid.

[EMFILE] The argument *cmd* is F_DUPFD and the maximum number of file descriptors permitted for the process are already in use, or no file descriptors greater than or equal to *arg* are available.

[ENOTTY] The *fd* argument is not a valid file descriptor for the requested operation. This may be the case if *fd* is a device node, or a descriptor returned by kqueue(2).

[ENOLCK] The argument *cmd* is F_SETLK or F_SETLKW, and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

[EOPNOTSUPP] The argument *cmd* is F_GETLK, F_SETLK or F_SETLKW and *fd* refers to a file for which locking is not supported.

[EOVERFLOW] The argument *cmd* is F_GETLK, F_SETLK or F_SETLKW and an *off_t* calculation overflowed.

[EPERM] The *cmd* argument is F_SETOWN and the process ID or process group given as an argument is in a different session than the caller.

[ESRCH] The *cmd* argument is F_SETOWN and the process ID given as argument is not in use.

In addition, if *fd* refers to a descriptor open on a terminal device (as opposed to a descriptor open on a socket), a *cmd* of F_SETOWN can fail for the same reasons as in tcsetpgrp(3), and a *cmd* of F_GETOWN for the reasons as stated in tcgetpgrp(3).

SEE ALSO

close(2), dup2(2), execve(2), flock(2), getdtablesize(2), open(2), sigaction(2), lockf(3), tcgetpgrp(3), tcsetpgrp(3)

STANDARDS

The F_DUP2FD constant is non portable. It is provided for compatibility with AIX and Solaris.

Per Version 4 of the Single UNIX Specification ("SUSv4"), a call with F_SETLKW should fail with [EINTR] after any caught signal and should continue waiting during thread suspension such as a stop signal. However, in this implementation a call with F_SETLKW is restarted after catching a signal with a SA_RESTART handler or a thread suspension such as a stop signal.

HISTORY

The **fcntl**() system call appeared in 4.2BSD.

The F_DUP2FD constant first appeared in FreeBSD 7.1.

NAME

fdatasync, **fsync** - synchronise changes to a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
fdatasync(*int fd*);

int
fsync(*int fd*);

DESCRIPTION

The **fsync**() system call causes all modified data and attributes of the file referenced by the file descriptor *fd* to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk.

The **fdatasync**() system call causes all modified data of *fd* to be moved to a permanent storage device. Unlike **fsync**(), the system call does not guarantee that file attributes or metadata necessary to access the file are committed to the permanent storage.

The **fsync**() system call should be used by programs that require a file to be in a known state, for example, in building a simple transaction facility. If the file metadata has already been committed, using **fdatasync**() can be more efficient than **fsync**().

Both **fdatasync**() and **fsync**() calls are cancellation points.

RETURN VALUES

The **fsync**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **fsync**() and **fdatasync**() calls fail if:

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

[EINVAL] The *fd* argument refers to a socket, not to a file.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

fsync(1), sync(2), syncer(4), sync(8)

HISTORY

The **fsync**() system call appeared in 4.2BSD. The **fdatasync**() system call appeared in FreeBSD 11.1.

BUGS

The **fdatasync**() system call currently does not guarantee that enqueued aio(4) requests for the file referenced by *fd* are completed before the syscall returns.

NAME

fopen, **fdopen**, **freopen**, **fmemopen** - stream open functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*FILE **
fopen(*const char * restrict path*, *const char * restrict mode*);

*FILE **
fdopen(*int fildes*, *const char *mode*);

*FILE **
freopen(*const char *path*, *const char *mode*, *FILE *stream*);

*FILE **
fmemopen(*void *restrict *buf*, *size_t size*, *const char * restrict mode*);

DESCRIPTION

The **fopen**() function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following letters:

"r" Open for reading. The stream is positioned at the beginning of the file. Fail if the file does not exist.

"w" Open for writing. The stream is positioned at the beginning of the file. Truncate the file to zero length if it exists or create the file if it does not exist.

"a" Open for writing. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file, irrespective of any intervening fseek(3) or similar. Create the file if it does not exist.

An optional "+" following "r", "w", or "a" opens the file for both reading and writing. An optional "x" following "w" or "w+" causes the **fopen**() call to fail if the file already exists. An optional "e" following the above causes the **fopen**() call to set the FD_CLOEXEC flag on the underlying file descriptor.

The *mode* string can also include the letter "b" after either the "+" or the first letter. This is strictly for compatibility with ISO/IEC 9899:1990 ("ISO C90") and has effect only for **fmemopen**() ; otherwise "b" is ignored.

Any created files will have mode "S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH" (0666), as modified by the process' umask value (see umask(2)).

Reads and writes may be intermixed on read/write streams in any order, and do not require an intermediate seek as in previous versions of *stdio*. This is not portable to other systems, however; ANSI C requires that a file positioning function intervene between output and input, unless an input operation encounters end-of-file.

The **fdopen**() function associates a stream with the existing file descriptor, *fildes*. The mode of the stream must be compatible with the mode of the file descriptor. The "x" mode option is ignored. If the "e" mode option is present, the FD_CLOEXEC flag is set, otherwise it remains unchanged. When the stream is closed via fclose(3), *fildes* is closed also.

The **freopen**() function opens the file whose name is the string pointed to by *path* and associates the stream pointed to by *stream* with it. The original stream (if it exists) is closed. The *mode* argument is used just as in the **fopen**() function.

If the *path* argument is NULL, **freopen**() attempts to re-open the file associated with *stream* with a new mode. The new mode must be compatible with the mode that the stream was originally opened with: Streams open for reading can only be re-opened for reading, streams open for writing can only be re-opened for writing, and streams open for reading and writing can be re-opened in any mode. The "x" mode option is not meaningful in this context.

The primary use of the **freopen**() function is to change the file associated with a standard text stream (stderr, stdin, or stdout).

The **fmemopen**() function associates the buffer given by the *buf* and *size* arguments with a stream. The *buf* argument is either a null pointer or point to a buffer that is at least *size* bytes long. If a null pointer is specified as the *buf* argument, **fmemopen**() allocates *size* bytes of memory. This buffer is automatically freed when the stream is closed. Buffers can be opened in text-mode (default) or binary-mode (if "b" is present in the second or third position of the *mode* argument). Buffers opened in text-mode make sure that writes are terminated with a NULL byte, if the last write hasn't filled up the whole buffer. Buffers opened in binary-mode never append a NULL byte.

RETURN VALUES

Upon successful completion **fopen**(), **fdopen**() and **freopen**() return a FILE pointer. Otherwise, NULL is

returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The *mode* argument to **fopen**(), **fdopen**(), **freopen**(), or **fmemopen**() was invalid.

The **fopen**(), **fdopen**(), **freopen**() and **fmemopen**() functions may also fail and set *errno* for any of the errors specified for the routine malloc(3).

The **fopen**() function may also fail and set *errno* for any of the errors specified for the routine open(2).

The **fdopen**() function may also fail and set *errno* for any of the errors specified for the routine fcntl(2).

The **freopen**() function may also fail and set *errno* for any of the errors specified for the routines open(2), fclose(3) and fflush(3).

The **fmemopen**() function may also fail and set *errno* if the *size* argument is 0.

SEE ALSO

open(2), fclose(3), fileno(3), fseek(3), funopen(3)

STANDARDS

The **fopen**() and **freopen**() functions conform to ISO/IEC 9899:1990 ("ISO C90"), with the exception of the "x" mode option which conforms to ISO/IEC 9899:2011 ("ISO C11"). The **fdopen**() function conforms to IEEE Std 1003.1-1988 ("POSIX.1"). The "e" mode option does not conform to any standard but is also supported by glibc. The **fmemopen**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1"). The "b" mode does not conform to any standard but is also supported by glibc.

NAME

feature_present - query presence of a kernel feature

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
feature_present(*const char *feature*);

DESCRIPTION

The **feature_present**() function provides a method for an application to determine if a specific kernel feature is present in the currently running kernel. The *feature* argument specifies the name of the feature to check. The **feature_present**() function will return 1 if the specified feature is present, otherwise it will return 0. If the **feature_present**() function is not able to determine the presence of *feature* due to an internal error it will return 0.

RETURN VALUES

If *feature* is present then 1 is returned; otherwise 0 is returned.

SEE ALSO

sysconf(3), sysctl(3)

HISTORY

The **feature_present**() function first appeared in FreeBSD 8.0.

NAME

ffclock_getcounter, **ffclock_getestimate**, **ffclock_setestimate** - Retrieve feed-forward counter, get and set feed-forward clock estimates

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/timeffc.h>

int
ffclock_getcounter(*ffcounter *ffcount*);

int
ffclock_getestimate(*struct ffclock_estimate *cest*);

int
ffclock_setestimate(*struct ffclock_estimate *cest*);

DESCRIPTION

The ffclock is an alternative method to synchronise the system clock. The ffclock implements a feed-forward paradigm and decouples the timestamping and timekeeping kernel functions. This ensures that past clock errors do not affect current timekeeping, an approach radically different from the feedback alternative implemented by the ntpd daemon when adjusting the system clock. The feed-forward approach has demonstrated better performance and higher robustness than a feedback approach when synchronising over the network.

In the feed-forward context, a *timestamp* is a cumulative value of the ticks of the timecounter, which can be converted into seconds by using the feed-forward *clock estimates*.

The **ffclock_getcounter**() system call allows the calling process to retrieve the current value of the feed-forward counter maintained by the kernel.

The **ffclock_getestimate**() and **ffclock_setestimate**() system calls allow the caller to get and set the kernel's feed-forward clock parameter estimates respectively. The **ffclock_setestimate**() system call should be invoked by a single instance of a feed-forward synchronisation daemon. The **ffclock_getestimate**() system call can be called by any process to retrieve the feed-forward clock estimates.

The feed-forward approach does not require that the clock estimates be retrieved every time a timestamp

is to be converted into seconds. The number of system calls can therefore be greatly reduced if the calling process retrieves the clock estimates from the clock synchronisation daemon instead. The **ffclock_getestimate**() must be used when the feed-forward synchronisation daemon is not running (see *USAGE* below).

The clock parameter estimates structure pointed to by *cest* is defined in <*sys/timeffc.h*> as:

```
struct ffclock_estimate {
        struct bintime update_time;    /* Time of last estimates update. */
        ffcounter      update_ffcount; /* Counter value at last update. */
        ffcounter      leapsec_next;   /* Counter value of next leap second. */
        uint64_t       period;         /* Estimate of counter period. */
        uint32_t       errb_abs;       /* Bound on absolute clock error [ns]. */
        uint32_t       errb_rate;      /* Bound on counter rate error [ps/s]. */
        uint32_t       status;         /* Clock status. */
        int16_t        leapsec_total;  /* All leap seconds seen so far. */
        int8_t         leapsec;        /* Next leap second (in {-1,0,1}). */
};
```

Only the super-user may set the feed-forward clock estimates.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following error codes may be set in *errno*:

[EFAULT] The *ffcount* or *cest* pointer referenced invalid memory.

[EPERM] A user other than the super-user attempted to set the feed-forward clock parameter estimates.

USAGE

The feed-forward paradigm enables the definition of specialised clock functions.

In its simplest form, **ffclock_getcounter**() can be used to establish strict order between events or to measure small time intervals very accurately with a minimum performance cost.

Different methods exist to access absolute time (or "wall-clock time") tracked by the ffclock. The

simplest method uses the ffclock sysctl interface *kern.ffclock* to make the system clock return the ffclock time. The clock_gettime(2) system call can then be used to retrieve the current time seen by the feed-forward clock. Note that this setting affects the entire system and that a feed-forward synchronisation daemon should be running.

A less automated method consists of retrieving the feed-forward counter timestamp from the kernel and using the feed-forward clock parameter estimates to convert the timestamp into seconds. The feed-forward clock parameter estimates can be retrieved from the kernel or from the synchronisation daemon directly (preferred). This method allows converting timestamps using different clock models as needed by the application, while collecting meaningful upper bounds on current clock error.

SEE ALSO

date(1), adjtime(2), clock_gettime(2), ctime(3)

HISTORY

Feed-forward clock support first appeared in FreeBSD 10.0.

AUTHORS

The feed-forward clock support was written by Julien Ridoux <*jridoux@unimelb.edu.au*> in collaboration with Darryl Veitch <*dveitch@unimelb.edu.au*> at the University of Melbourne under sponsorship from the FreeBSD Foundation.

NAME

fflagstostr, **strtofflags** - convert between file flag bits and their string names

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*char **
fflagstostr(*u_long flags*);

int
strtofflags(*char **stringp*, *u_long *setp*, *u_long *clrp*);

DESCRIPTION

The **fflagstostr**() function returns a comma separated string of the file flags represented by *flags*. If no flags are set a zero length string is returned.

If memory cannot be allocated for the return value, **fflagstostr**() returns NULL.

The value returned from **fflagstostr**() is obtained from **malloc**() and should be returned to the system with **free**() when the program is done with it.

The **strtofflags**() function takes a string of file flags, as described in chflags(1), parses it, and returns the 'set' flags and 'clear' flags such as would be given as arguments to chflags(2). On success **strtofflags**() returns 0, otherwise it returns non-zero and *stringp* is left pointing to the offending token.

ERRORS

The **fflagstostr**() function may fail and set errno for any of the errors specified for the library routine malloc(3).

SEE ALSO

chflags(1), chflags(2), malloc(3)

HISTORY

The **fflagstostr**() and **strtofflags**() functions first appeared in FreeBSD 4.0.

NAME

fflush, **fpurge** - flush a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
fflush(*FILE *stream*);

int
fpurge(*FILE *stream*);

DESCRIPTION

The function **fflush**() forces a write of all buffered data for the given output or update *stream* via the stream's underlying write function. The open status of the stream is unaffected.

If the *stream* argument is NULL, **fflush**() flushes *all* open output streams.

The function **fpurge**() erases any input or output buffered in the given *stream*. For output streams this discards any unwritten output. For input streams this discards any input read from the underlying object but not yet obtained via getc(3); this includes any text pushed back via ungetc(3).

RETURN VALUES

Upon successful completion 0 is returned. Otherwise, EOF is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EBADF] The *stream* argument is not an open stream.

The function **fflush**() may also fail and set *errno* for any of the errors specified for the routine write(2), except that in case of *stream* being a read-only descriptor, **fflush**() returns 0.

SEE ALSO

write(2), fclose(3), fopen(3), setbuf(3)

STANDARDS

The **fflush**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

NAME

ffs, **ffsl**, **ffsll**, **fls**, **flsl**, **flsll** - find first or last bit set in a bit string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <strings.h>

int
ffs(*int value*);

int
ffsl(*long value*);

int
ffsll(*long long value*);

int
fls(*int value*);

int
flsl(*long value*);

int
flsll(*long long value*);

DESCRIPTION

The **ffs**(), **ffsl**() and **ffsll**() functions find the first (least significant) bit set in *value* and return the index of that bit.

The **fls**(), **flsl**() and **flsll**() functions find the last (most significant) bit set in *value* and return the index of that bit.

Bits are numbered starting at 1, the least significant bit. A return value of zero from any of these functions means that the argument was zero.

SEE ALSO

bitstring(3), bitset(9)

HISTORY

The **ffs**() function appeared in 4.3BSD. Its prototype existed previously in <*string.h*> before it was moved to <*strings.h*> for IEEE Std 1003.1-2001 ("POSIX.1") compliance.

The **ffsl**(), **fls**() and **flsl**() functions appeared in FreeBSD 5.3. The **ffsll**() and **flsll**() functions appeared in FreeBSD 7.1.

NAME

fgetc, **getc**, **getc_unlocked**, **getchar**, **getchar_unlocked**, **getw** - get next character or word from input stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
fgetc(*FILE *stream*);

int
getc(*FILE *stream*);

int
getc_unlocked(*FILE *stream*);

int
getchar(*void*);

int
getchar_unlocked(*void*);

int
getw(*FILE *stream*);

DESCRIPTION

The **fgetc**() function obtains the next input character (if present) from the stream pointed at by *stream*, or the next character pushed back on the stream via ungetc(3).

The **getc**() function acts essentially identically to **fgetc**(), but is a macro that expands in-line.

The **getchar**() function is equivalent to **getc**(*stdin*).

The **getw**() function obtains the next *int* (if present) from the stream pointed at by *stream*.

The **getc_unlocked**() and **getchar_unlocked**() functions are equivalent to **getc**() and **getchar**() respectively, except that the caller is responsible for locking the stream with flockfile(3) before calling

them. These functions may be used to avoid the overhead of locking the stream for each character, and to avoid input being dispersed among multiple threads reading from the same stream.

RETURN VALUES

If successful, these routines return the next requested object from the *stream*. Character values are returned as an *unsigned char* converted to an *int*. If the stream is at end-of-file or a read error occurs, the routines return EOF. The routines feof(3) and ferror(3) must be used to distinguish between end-of-file and error. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return EOF until the condition is cleared with clearerr(3).

SEE ALSO

ferror(3), flockfile(3), fopen(3), fread(3), getwc(3), putc(3), ungetc(3)

STANDARDS

The **fgetc**(), **getc**(), and **getchar**() functions conform to ISO/IEC 9899:1990 ("ISO C90"). The **getc_unlocked**() and **getchar_unlocked**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

BUGS

Since EOF is a valid integer value, feof(3) and ferror(3) must be used to check for failure after calling **getw**(). The size and byte order of an *int* varies from one machine to another, and **getw**() is not recommended for portable applications.

NAME

fgetln - get a line from a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*char **
fgetln(*FILE *stream*, *size_t *len*);

DESCRIPTION

The **fgetln**() function returns a pointer to the next line from the stream referenced by *stream*. This line is *not* a C string as it does not end with a terminating NUL character. The length of the line, including the final newline, is stored in the memory location to which *len* points. (Note, however, that if the line is the last in a file that does not end in a newline, the returned text will not contain a newline.)

RETURN VALUES

Upon successful completion a pointer is returned; this pointer becomes invalid after the next I/O operation on *stream* (whether successful or not) or as soon as the stream is closed. Otherwise, NULL is returned. The **fgetln**() function does not distinguish between end-of-file and error; the routines feof(3) and ferror(3) must be used to determine which occurred. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return NULL until the condition is cleared with clearerr(3).

The text to which the returned pointer points may be modified, provided that no changes are made beyond the returned size. These changes are lost as soon as the pointer becomes invalid.

ERRORS

[EBADF] The argument *stream* is not a stream open for reading.

[ENOMEM] The internal line buffer could not be expanded due to lack of available memory, or because it would need to expand beyond INT_MAX in size.

The **fgetln**() function may also fail and set *errno* for any of the errors specified for the routines fflush(3), malloc(3), read(2), stat(2), or realloc(3).

SEE ALSO

ferror(3), fgets(3), fgetwln(3), fopen(3), getline(3), putc(3)

HISTORY

The **fgetln**() function first appeared in 4.4BSD.

NAME

fgetpos, **fseek**, **fseeko**, **fsetpos**, **ftell**, **ftello**, **rewind** - reposition a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
fseek(*FILE *stream*, *long offset*, *int whence*);

long
ftell(*FILE *stream*);

void
rewind(*FILE *stream*);

int
fgetpos(*FILE * restrict stream*, *fpos_t * restrict pos*);

int
fsetpos(*FILE *stream*, *const fpos_t *pos*);

#include <sys/types.h>

int
fseeko(*FILE *stream*, *off_t offset*, *int whence*);

off_t
ftello(*FILE *stream*);

DESCRIPTION

The **fseek**() function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the **fseek**() function clears the end-of-file indicator for the stream and undoes any effects of the ungetc(3) and ungetwc(3) functions on the same stream.

The **ftell**() function obtains the current value of the file position indicator for the stream pointed to by *stream*.

The **rewind**() function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to:

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared (see clearerr(3)).

Since **rewind**() does not return a value, an application wishing to detect errors should clear *errno*, then call **rewind**(), and if *errno* is non-zero, assume an error has occurred.

The **fseeko**() function is identical to **fseek**(), except it takes an *off_t* argument instead of a *long*. Likewise, the **ftello**() function is identical to **ftell**(), except it returns an *off_t*.

The **fgetpos**() and **fsetpos**() functions are alternate interfaces for retrieving and setting the current position in the file, similar to **ftell**() and **fseek**(), except that the current position is stored in an opaque object of type *fpos_t* pointed to by *pos*. These functions provide a portable way to seek to offsets larger than those that can be represented by a *long int*. They may also store additional state information in the *fpos_t* object to facilitate seeking within files containing multibyte characters with state-dependent encodings. Although *fpos_t* has traditionally been an integral type, applications cannot assume that it is; in particular, they must not perform arithmetic on objects of this type.

If the stream is a wide character stream (see fwide(3)), the position specified by the combination of *offset* and *whence* must contain the first byte of a multibyte sequence.

RETURN VALUES

The **rewind**() function returns no value.

The **fgetpos**(), **fseek**(), **fseeko**(), and **fsetpos**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

Upon successful completion, **ftell**() and **ftello**() return the current offset. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EBADF] The *stream* argument is not a seekable stream.

[EINVAL] The *whence* argument is invalid or the resulting file-position indicator would be

set to a negative value.

[EOVERFLOW] The resulting file offset would be a value which cannot be represented correctly in an object of type *off_t* for **fseeko**() and **ftello**() or *long* for **fseek**() and **ftell**().

[ESPIPE] The file descriptor underlying stream is associated with a pipe or FIFO or file-position indicator value is unspecified (see ungetc(3)).

The functions **fgetpos**(), **fseek**(), **fseeko**(), **fsetpos**(), **ftell**(), **ftello**(), and **rewind**() may also fail and set *errno* for any of the errors specified for the routines fflush(3), fstat(2), lseek(2), and malloc(3).

SEE ALSO

lseek(2), clearerr(3), fwide(3), ungetc(3), ungetwc(3)

STANDARDS

The **fgetpos**(), **fsetpos**(), **fseek**(), **ftell**(), and **rewind**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

The **fseeko**() and **ftello**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

fgets, **gets**, **gets_s** - get a line from a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*char **
fgets(*char * restrict str*, *int size*, *FILE * restrict stream*);

*char **
gets_s(*char *str*, *rsize_t size*);

*char **
gets(*char *str*);

DESCRIPTION

The **fgets**() function reads at most one less than the number of characters specified by *size* from the given *stream* and stores them in the string *str*. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a '\0' character is appended to end the string.

The **gets_s**() function is equivalent to **fgets**() with a *stream* of stdin, except that the newline character (if any) is not stored in the string.

The **gets**() function is equivalent to **fgets**() with an infinite *size* and a *stream* of stdin, except that the newline character (if any) is not stored in the string. It is the caller's responsibility to ensure that the input line, if any, is sufficiently short to fit in the string.

RETURN VALUES

Upon successful completion, **fgets**(), **gets_s**(), and **gets**() return a pointer to the string. If end-of-file occurs before any characters are read, they return NULL and the buffer contents remain unchanged. If an error occurs, they return NULL and the buffer contents are indeterminate. The **fgets**(), **gets_s**(), and **gets**() functions do not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

ERRORS

[EBADF] The given *stream* is not a readable stream.

The function **fgets**() may also fail and set *errno* for any of the errors specified for the routines fflush(3), fstat(2), read(2), or malloc(3).

The function **gets**() and **gets_s**() may also fail and set *errno* for any of the errors specified for the routine getchar(3).

SEE ALSO

feof(3), ferror(3), fgetln(3), fgetws(3), getline(3)

STANDARDS

The functions **fgets**() and **gets**() conform to ISO/IEC 9899:1999 ("ISO C99"). **gets_s**() conforms to ISO/IEC 9899:2011 ("ISO C11") K.3.7.4.1. **gets**() has been removed from ISO/IEC 9899:2011 ("ISO C11").

SECURITY CONSIDERATIONS

The **gets**() function cannot be used securely. Because of its lack of bounds checking, and the inability for the calling program to reliably determine the length of the next incoming line, the use of this function enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. It is strongly suggested that the **fgets**() function be used in all cases.

NAME

fgetwc, **getwc**, **getwchar** - get next wide character from input stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

wint_t
fgetwc(*FILE *stream*);

wint_t
getwc(*FILE *stream*);

wint_t
getwchar(*void*);

DESCRIPTION

The **fgetwc**() function obtains the next input wide character (if present) from the stream pointed at by *stream*, or the next character pushed back on the stream via ungetwc(3).

The **getwc**() function acts essentially identically to **fgetwc**().

The **getwchar**() function is equivalent to **getwc**() with the argument stdin.

RETURN VALUES

If successful, these routines return the next wide character from the *stream*. If the stream is at end-of-file or a read error occurs, the routines return WEOF. The routines feof(3) and ferror(3) must be used to distinguish between end-of-file and error. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return WEOF until the condition is cleared with clearerr(3).

SEE ALSO

ferror(3), fopen(3), fread(3), getc(3), putwc(3), stdio(3), ungetwc(3)

STANDARDS

The **fgetwc**(), **getwc**() and **getwchar**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

fgetwln - get a line of wide characters from a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

*wchar_t **
fgetwln(*FILE * restrict stream*, *size_t * restrict len*);

DESCRIPTION

The **fgetwln**() function returns a pointer to the next line from the stream referenced by *stream*. This line is *not* a standard wide character string as it does not end with a terminating null wide character. The length of the line, including the final newline, is stored in the memory location to which *len* points. (Note, however, that if the line is the last in a file that does not end in a newline, the returned text will not contain a newline.)

RETURN VALUES

Upon successful completion a pointer is returned; this pointer becomes invalid after the next I/O operation on *stream* (whether successful or not) or as soon as the stream is closed. Otherwise, NULL is returned. The **fgetwln**() function does not distinguish between end-of-file and error; the routines feof(3) and ferror(3) must be used to determine which occurred. If an error occurs, the global variable *errno* is set to indicate the error. The end-of-file condition is remembered, even on a terminal, and all subsequent attempts to read will return NULL until the condition is cleared with clearerr(3).

The text to which the returned pointer points may be modified, provided that no changes are made beyond the returned size. These changes are lost as soon as the pointer becomes invalid.

ERRORS

[EBADF] The argument *stream* is not a stream open for reading.

The **fgetwln**() function may also fail and set *errno* for any of the errors specified for the routines mbrtowc(3), realloc(3), or read(2).

SEE ALSO

ferror(3), fgetln(3), fgetws(3), fopen(3)

NAME

fgetws - get a line of wide characters from a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

*wchar_t **
fgetws(*wchar_t * restrict ws*, *int n*, *FILE * restrict fp*);

DESCRIPTION

The **fgetws**() function reads at most one less than the number of characters specified by *n* from the given *fp* and stores them in the wide character string *ws*. Reading stops when a newline character is found, at end-of-file or error. The newline, if any, is retained. If any characters are read and there is no error, a '\0' character is appended to end the string.

RETURN VALUES

Upon successful completion, **fgetws**() returns *ws*. If end-of-file occurs before any characters are read, **fgetws**() returns NULL and the buffer contents remain unchanged. If an error occurs, **fgetws**() returns NULL and the buffer contents are indeterminate. The **fgetws**() function does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred.

ERRORS

The **fgetws**() function will fail if:

[EBADF] The given *fp* argument is not a readable stream.

[EILSEQ] The data obtained from the input stream does not form a valid multibyte character.

The function **fgetws**() may also fail and set *errno* for any of the errors specified for the routines fflush(3), fstat(2), read(2), or malloc(3).

SEE ALSO

feof(3), ferror(3), fgets(3)

STANDARDS

The **fgetws**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

fhopen, **fhstat**, **fhstatfs** - access file via file handle

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/mount.h>
#include <sys/stat.h>

int
fhopen(*const fhandle_t *fhp*, *int flags*);

int
fhstat(*const fhandle_t *fhp*, *struct stat *sb*);

int
fhstatfs(*const fhandle_t *fhp*, *struct statfs *buf*);

DESCRIPTION

These system calls provide a means to access a file given the file handle *fhp*. As this method bypasses directory access restrictions, these calls are restricted to the superuser.

The **fhopen**() system call opens the file referenced by *fhp* for reading and/or writing as specified by the argument *flags* and returns the file descriptor to the calling process. The *flags* argument is specified by *or*'ing together the flags used for the open(2) system call. All said flags are valid except for O_CREAT.

The **fhstat**() and **fhstatfs**() system calls provide the functionality of the fstat(2) and fstatfs(2) calls except that they return information for the file referred to by *fhp* rather than an open file.

RETURN VALUES

Upon successful completion, **fhopen**() returns the file descriptor for the opened file; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

The **fhstat**() and **fhstatfs**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

In addition to the errors returned by open(2), fstat(2), and fstatfs(2) respectively, **fhopen**(), **fhstat**(), and

fhstatfs() will return

[EINVAL] Calling **fhopen**() with O_CREAT set.

[ESTALE] The file handle *fhp* is no longer valid.

SEE ALSO

fstat(2), fstatfs(2), getfh(2), open(2)

HISTORY

The **fhopen**(), **fhstat**(), and **fhstatfs**() system calls first appeared in NetBSD 1.5 and were adapted to FreeBSD 4.0 by Alfred Perlstein.

AUTHORS

This manual page was written by William Studenmund for NetBSD.

NAME

flock - apply or remove an advisory lock on an open file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

#define   LOCK_SH      0x01    /* shared file lock */
#define   LOCK_EX      0x02    /* exclusive file lock */
#define   LOCK_NB      0x04    /* do not block when locking */
#define   LOCK_UN      0x08    /* unlock file */

int
flock(int fd, int operation);
```

DESCRIPTION

The **flock**() system call applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* argument that is one of LOCK_SH or LOCK_EX with the optional addition of LOCK_NB. To unlock an existing lock operation should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency (i.e., processes may still access files without using advisory locks possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. At any time multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that is already locked normally causes the caller to be blocked until the lock may be acquired. If LOCK_NB is included in *operation*, then this will not happen; instead the call will fail and the error EWOULDBLOCK will be returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through dup(2) or fork(2) do

not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock.

The **flock**(), fcntl(2), and lockf(3) locks are compatible. Processes using different locking interfaces can cooperate over the same file safely. However, only one of such interfaces should be used within the same process. If a file is locked by a process through **flock**(), any record within the file will be seen as locked from the viewpoint of another process using fcntl(2) or lockf(3), and vice versa.

Processes blocked awaiting a lock may be awakened by signals.

RETURN VALUES

The **flock**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **flock**() system call fails if:

[EWOULDBLOCK] The file is locked and the LOCK_NB option was specified.

[EBADF] The argument *fd* is an invalid descriptor.

[EINVAL] The argument *fd* refers to an object other than a file.

[EOPNOTSUPP] The argument *fd* refers to an object that does not support file locking.

[ENOLCK] A lock was requested, but no locks are available.

SEE ALSO

close(2), dup(2), execve(2), fcntl(2), fork(2), open(2), flopen(3), lockf(3)

HISTORY

The **flock**() system call appeared in 4.2BSD.

NAME

flockfile, **ftrylockfile**, **funlockfile** - stdio locking functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

void
flockfile(*FILE *stream*);

int
ftrylockfile(*FILE *stream*);

void
funlockfile(*FILE *stream*);

DESCRIPTION

These functions provide explicit application-level locking of stdio streams. They can be used to avoid output from multiple threads being interspersed, input being dispersed among multiple readers, and to avoid the overhead of locking the stream for each operation.

The **flockfile**() function acquires an exclusive lock on the specified stream. If another thread has already locked the stream, **flockfile**() will block until the lock is released.

The **ftrylockfile**() function is a non-blocking version of **flockfile**(); if the lock cannot be acquired immediately, **ftrylockfile**() returns non-zero instead of blocking.

The **funlockfile**() function releases the lock on a stream acquired by an earlier call to **flockfile**() or **ftrylockfile**().

These functions behave as if there is a lock count associated with each stream. Each time **flockfile**() is called on the stream, the count is incremented, and each time **funlockfile**() is called on the stream, the count is decremented. The lock is only actually released when the count reaches zero.

RETURN VALUES

The **flockfile**() and **funlockfile**() functions return no value.

The **ftrylockfile**() function returns zero if the stream was successfully locked, non-zero otherwise.

SEE ALSO

getc_unlocked(3), putc_unlocked(3)

STANDARDS

The **flockfile**(), **ftrylockfile**() and **funlockfile**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

fmtcheck - sanitizes user-supplied printf(3)-style format string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*const char **

fmtcheck(*const char *fmt_suspect*, *const char *fmt_default*);

DESCRIPTION

The **fmtcheck**() scans *fmt_suspect* and *fmt_default* to determine if *fmt_suspect* will consume the same argument types as *fmt_default* and to ensure that *fmt_suspect* is a valid format string.

The printf(3) family of functions cannot verify the types of arguments that they are passed at run-time. In some cases, like catgets(3), it is useful or necessary to use a user-supplied format string with no guarantee that the format string matches the specified arguments.

The **fmtcheck**() was designed to be used in these cases, as in:

```
printf(fmtcheck(user_format, standard_format), arg1, arg2);
```

In the check, field widths, fillers, precisions, etc. are ignored (unless the field width or precision is an asterisk ‘*’ instead of a digit string). Also, any text other than the format specifiers is completely ignored.

RETURN VALUES

If *fmt_suspect* is a valid format and consumes the same argument types as *fmt_default*, then the **fmtcheck**() will return *fmt_suspect*. Otherwise, it will return *fmt_default*.

SEE ALSO

printf(3)

BUGS

The **fmtcheck**() function does not recognize positional parameters.

SECURITY CONSIDERATIONS

Note that the formats may be quite different as long as they accept the same arguments. For example,

"%p %o %30s %#llx %-10.*e %n" is compatible with "This number %lu %d%% and string %s has %qd numbers and %.*g floats (%n)". However, "%o" is not equivalent to "%lx" because the first requires an integer and the second requires a long.

NAME

fmtmsg - display a detailed diagnostic message

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fmtmsg.h>

int
fmtmsg(*long classification*, *const char *label*, *int severity*, *const char *text*, *const char *action*, *const char *tag*);

DESCRIPTION

The **fmtmsg**() function displays a detailed diagnostic message, based on the supplied arguments, to stderr and/or the system console.

The *classification* argument is the bitwise inclusive OR of zero or one of the manifest constants from each of the classification groups below. The Output classification group is an exception since both MM_PRINT and MM_CONSOLE may be specified.

Output

MM_PRINT	Output should take place on stderr.
MM_CONSOLE	Output should take place on the system console.

Source of Condition (Major)

MM_HARD	The source of the condition is hardware related.
MM_SOFT	The source of the condition is software related.
MM_FIRM	The source of the condition is firmware related.

Source of Condition (Minor)

MM_APPL	The condition was detected at the application level.
MM_UTIL	The condition was detected at the utility level.

MM_OPSYS The condition was detected at the operating system level.

Status

MM_RECOVER The application can recover from the condition.

MM_NRECOV The application is unable to recover from the condition.

Alternatively, the MM_NULLMC manifest constant may be used to specify no classification.

The *label* argument indicates the source of the message. It is made up of two fields separated by a colon (':'). The first field can be up to 10 bytes, and the second field can be up to 14 bytes. The MM_NULLLBL manifest constant may be used to specify no label.

The *severity* argument identifies the importance of the condition. One of the following manifest constants should be used for this argument.

MM_HALT The application has confronted a serious fault and is halting.

MM_ERROR The application has detected a fault.

MM_WARNING The application has detected an unusual condition, that could be indicative of a problem.

MM_INFO The application is providing information about a non-error condition.

MM_NOSEV No severity level supplied.

The *text* argument details the error condition that caused the message. There is no limit on the size of this character string. The MM_NULLTXT manifest constant may be used to specify no text.

The *action* argument details how the error-recovery process should begin. Upon output, **fmtmsg**() will prefix "TO FIX:" to the beginning of the *action* argument. The MM_NULLACT manifest constant may be used to specify no action.

The *tag* argument should reference online documentation for the message. This usually includes the *label* and a unique identifying number. An example tag is "BSD:ls:168". The MM_NULLTAG manifest constant may be used to specify no tag.

RETURN VALUES

The **fmtmsg**() function returns MM_OK upon success, MM_NOMSG to indicate output to stderr failed, MM_NOCON to indicate output to the system console failed, or MM_NOTOK to indicate output to stderr and the system console failed.

ENVIRONMENT

The MSGVERB (message verbosity) environment variable specifies which arguments to **fmtmsg**() will be output to stderr, and in which order. MSGVERB should be a colon (':') separated list of identifiers. Valid identifiers include: label, severity, text, action, and tag. If invalid identifiers are specified or incorrectly separated, the default message verbosity and ordering will be used. The default ordering is equivalent to a MSGVERB with a value of "label:severity:text:action:tag".

EXAMPLES

The code:

```
fmtmsg(MM_UTIL | MM_PRINT, "BSD:ls", MM_ERROR,
    "illegal option -- z", "refer to manual", "BSD:ls:001");
```

will output:

```
BSD:ls: ERROR: illegal option -- z
TO FIX: refer to manual BSD:ls:001
```

to stderr.

The same code, with MSGVERB set to "text:severity:action:tag", produces:

```
illegal option -- z: ERROR
TO FIX: refer to manual BSD:ls:001
```

SEE ALSO

err(3), exit(3), strerror(3)

STANDARDS

The **fmtmsg**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **fmtmsg**() function first appeared in FreeBSD 5.0.

BUGS

Specifying MM_NULLMC for the *classification* argument makes little sense, since without an output

specified, **fmtmsg**() is unable to do anything useful.

In order for **fmtmsg**() to output to the system console, the effective user must have appropriate permission to write to */dev/console*. This means that on most systems **fmtmsg**() will return MM_NOCON unless the effective user is root.

NAME

fnmatch - test whether a filename or pathname matches a shell-style pattern

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fnmatch.h>

int
fnmatch(*const char *pattern*, *const char *string*, *int flags*);

DESCRIPTION

The **fnmatch**() function matches patterns according to the rules used by the shell. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. The value of *flags* is the bitwise inclusive OR of any of the following constants, which are defined in the include file <*fnmatch.h*>.

FNM_NOESCAPE Normally, every occurrence of a backslash (‘\’) followed by a character in *pattern* is replaced by that character. This is done to negate any special meaning for the character. If the FNM_NOESCAPE flag is set, a backslash character is treated as an ordinary character.

FNM_PATHNAME Slash characters in *string* must be explicitly matched by slashes in *pattern*. If this flag is not set, then slashes are treated as regular characters.

FNM_PERIOD Leading periods in *string* must be explicitly matched by periods in *pattern*. If this flag is not set, then leading periods are treated as regular characters. The definition of "leading" is related to the specification of FNM_PATHNAME. A period is always "leading" if it is the first character in *string*. Additionally, if FNM_PATHNAME is set, a period is leading if it immediately follows a slash.

FNM_LEADING_DIR
Ignore "/*" rest after successful *pattern* matching.

FNM_CASEFOLD Ignore case distinctions in both the *pattern* and the *string*.

RETURN VALUES

The **fnmatch**() function returns zero if *string* matches the pattern specified by *pattern*, otherwise, it

returns the value FNM_NOMATCH.

SEE ALSO

sh(1), glob(3), regex(3)

STANDARDS

The current implementation of the **fnmatch**() function *does not* conform to IEEE Std 1003.2 ("POSIX.2"). Collating symbol expressions, equivalence class expressions and character class expressions are not supported.

HISTORY

The **fnmatch**() function first appeared in 4.4BSD.

BUGS

The pattern '*' matches the empty string, even if FNM_PATHNAME is specified.

NAME

fopencookie - open a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

typedef ssize_t
(*cookie_read_function_t)(*void *cookie*, *char *buf*, *size_t size*);

typedef ssize_t
(*cookie_write_function_t)(*void *cookie*, *const char *buf*, *size_t size*);

typedef int
(*cookie_seek_function_t)(*void *cookie*, *off64_t *offset*, *int whence*);

typedef int
(*cookie_close_function_t)(*void *cookie*);

```
typedef struct {
        cookie_read_function_t      *read;
        cookie_write_function_t     *write;
        cookie_seek_function_t      *seek;
        cookie_close_function_t     *close;
} cookie_io_functions_t;
```

*FILE **
fopencookie(*void *cookie*, *const char *mode*, *cookie_io_functions_t io_funcs*);

DESCRIPTION

The **fopencookie** function associates a stream with up to four "I/O functions". These I/O functions will be used to read, write, seek and close the new stream.

In general, omitting a function means that any attempt to perform the associated operation on the resulting stream will fail. If the write function is omitted, data written to the stream is discarded. If the close function is omitted, closing the stream will flush any buffered output and then succeed.

The calling conventions of *read*, *write*, and *close* must match those, respectively, of read(2), write(2), and close(2) with the single exception that they are passed the *cookie* argument specified to **fopencookie**

in place of the traditional file descriptor argument. The *seek* function updates the current stream offset using **offset* and *whence*. If **offset* is non-NULL, it updates **offset* with the current stream offset.

fopencookie is implemented as a thin shim around the funopen(3) interface. Limitations, possibilities, and requirements of that interface apply to **fopencookie**.

RETURN VALUES

Upon successful completion, **fopencookie** returns a FILE pointer. Otherwise, NULL is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] A bogus *mode* was provided to **fopencookie**.

[ENOMEM] The **fopencookie** function may fail and set *errno* for any of the errors specified for the malloc(3) routine.

SEE ALSO

fcntl(2), open(2), fclose(3), fopen(3), fseek(3), funopen(3)

HISTORY

The **funopen**() functions first appeared in 4.4BSD. The **fopencookie** function first appeared in FreeBSD 11.

BUGS

The **fopencookie** function is a nonstandard glibc extension and may not be portable to systems other than FreeBSD and Linux.

NAME

fork - create a new process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
fork(*void*);

DESCRIPTION

The **fork**() system call causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors, except for descriptors returned by kqueue(2), which are not inherited from the parent process. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read(2) or write(2) by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child process' resource utilizations are set to 0; see setrlimit(2).
- All interval timers are cleared; see setitimer(2).
- The child process has only one thread, corresponding to the calling thread in the parent process. If the process has more than one thread, locks and other resources held by the other threads are not released and therefore only async-signal-safe functions (see sigaction(2)) are guaranteed to work in the child process until a call to execve(2) or a similar function.

RETURN VALUES

Upon successful completion, **fork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

ERRORS

The **fork**() system call will fail and no child process will be created if:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. The limit is given by the sysctl(3) MIB variable KERN_MAXPROC. (The limit is actually ten less than this except for the super user).

[EAGAIN] The user is not the super user, and the system-imposed limit on the total number of processes under execution by a single user would be exceeded. The limit is given by the sysctl(3) MIB variable KERN_MAXPROCPERUID.

[EAGAIN] The user is not the super user, and the soft resource limit corresponding to the *resource* argument RLIMIT_NPROC would be exceeded (see getrlimit(2)).

[ENOMEM] There is insufficient swap space for the new process.

SEE ALSO

execve(2), rfork(2), setitimer(2), setrlimit(2), sigaction(2), vfork(2), wait(2)

HISTORY

The **fork**() function appeared in Version 1 AT&T UNIX.

NAME

pathconf, **lpathconf**, **fpathconf** - get configurable pathname variables

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

long
pathconf(*const char *path*, *int name*);

long
lpathconf(*const char *path*, *int name*);

long
fpathconf(*int fd*, *int name*);

DESCRIPTION

The **pathconf**(), **lpathconf**() and **fpathconf**() system calls provide a method for applications to determine the current value of a configurable system limit or option variable associated with a pathname or file descriptor.

For **pathconf**() and **lpathconf**(), the *path* argument is the name of a file or directory. For **fpathconf**(), the *fd* argument is an open file descriptor. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file <unistd.h>.

The **lpathconf**() system call is like **pathconf**() except in the case where the named file is a symbolic link, in which case **lpathconf**() returns information about the link, while **pathconf**() returns information about the file the link references.

The available values are as follows:

_PC_LINK_MAX
The maximum file link count.

_PC_MAX_CANON
The maximum number of bytes in terminal canonical input line.

_PC_MAX_INPUT

The minimum maximum number of bytes for which space is available in a terminal input queue.

_PC_NAME_MAX
The maximum number of bytes in a file name.

_PC_PATH_MAX
The maximum number of bytes in a pathname.

_PC_PIPE_BUF
The maximum number of bytes which will be written atomically to a pipe.

_PC_CHOWN_RESTRICTED
Return 1 if appropriate privilege is required for the chown(2) system call, otherwise 0. IEEE Std 1003.1-2001 ("POSIX.1") requires appropriate privilege in all cases, but this behavior was optional in prior editions of the standard.

_PC_NO_TRUNC
Return greater than zero if attempts to use pathname components longer than {NAME_MAX} will result in an [ENAMETOOLONG] error; otherwise, such components will be truncated to {NAME_MAX}. IEEE Std 1003.1-2001 ("POSIX.1") requires the error in all cases, but this behavior was optional in prior editions of the standard, and some non-POSIX-compliant file systems do not support this behavior.

_PC_VDISABLE
Returns the terminal character disabling value.

_PC_ASYNC_IO
Return 1 if asynchronous I/O is supported, otherwise 0.

_PC_PRIO_IO
Returns 1 if prioritised I/O is supported for this file, otherwise 0.

_PC_SYNC_IO
Returns 1 if synchronised I/O is supported for this file, otherwise 0.

_PC_ALLOC_SIZE_MIN
Minimum number of bytes of storage allocated for any portion of a file.

_PC_FILESIZEBITS

Number of bits needed to represent the maximum file size.

_PC_REC_INCR_XFER_SIZE
Recommended increment for file transfer sizes between _PC_REC_MIN_XFER_SIZE and _PC_REC_MAX_XFER_SIZE.

_PC_REC_MAX_XFER_SIZE
Maximum recommended file transfer size.

_PC_REC_MIN_XFER_SIZE
Minimum recommended file transfer size.

_PC_REC_XFER_ALIGN
Recommended file transfer buffer alignment.

_PC_SYMLINK_MAX
Maximum number of bytes in a symbolic link.

_PC_ACL_EXTENDED
Returns 1 if an Access Control List (ACL) can be set on the specified file, otherwise 0.

_PC_ACL_NFS4
Returns 1 if an NFSv4 ACLs can be set on the specified file, otherwise 0.

_PC_ACL_PATH_MAX
Maximum number of ACL entries per file.

_PC_CAP_PRESENT
Returns 1 if a capability state can be set on the specified file, otherwise 0.

_PC_INF_PRESENT
Returns 1 if an information label can be set on the specified file, otherwise 0.

_PC_MAC_PRESENT
Returns 1 if a Mandatory Access Control (MAC) label can be set on the specified file, otherwise 0.

_PC_MIN_HOLE_SIZE
If a file system supports the reporting of holes (see lseek(2)), **pathconf**() and **fpathconf**() return a positive number that represents the minimum hole size returned in bytes. The offsets of holes

returned will be aligned to this same value. A special value of 1 is returned if the file system does not specify the minimum hole size but still reports holes.

RETURN VALUES

If the call to **pathconf**() or **fpathconf**() is not successful, -1 is returned and *errno* is set appropriately. Otherwise, if the variable is associated with functionality that does not have a limit in the system, -1 is returned and *errno* is not modified. Otherwise, the current variable value is returned.

ERRORS

If any of the following conditions occur, the **pathconf**() and **fpathconf**() system calls shall return -1 and set *errno* to the corresponding value.

[EINVAL] The value of the *name* argument is invalid.

[EINVAL] The implementation does not support an association of the variable name with the associated file.

The **pathconf**() system call will fail if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded {NAME_MAX} characters (but see _PC_NO_TRUNC above), or an entire path name exceeded {PATH_MAX} characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while reading from or writing to the file system.

The **fpathconf**() system call will fail if:

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

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

lseek(2), sysctl(3)

HISTORY

The **pathconf**() and **fpathconf**() system calls first appeared in 4.4BSD. The **lpathconf**() system call first appeared in FreeBSD 8.0.

NAME

fputc, **putc**, **putc_unlocked**, **putchar**, **putchar_unlocked**, **putw** - output a character or word to a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
fputc(*int c*, *FILE *stream*);

int
putc(*int c*, *FILE *stream*);

int
putc_unlocked(*int c*, *FILE *stream*);

int
putchar(*int c*);

int
putchar_unlocked(*int c*);

int
putw(*int w*, *FILE *stream*);

DESCRIPTION

The **fputc**() function writes the character *c* (converted to an ‘‘unsigned char’’) to the output stream pointed to by *stream*.

The **putc**() macro acts essentially identically to **fputc**(), but is a macro that expands in-line. It may evaluate *stream* more than once, so arguments given to **putc**() should not be expressions with potential side effects.

The **putchar**() function is identical to **putc**() with an output stream of stdout.

The **putw**() function writes the specified *int* to the named output *stream*.

The **putc_unlocked**() and **putchar_unlocked**() functions are equivalent to **putc**() and **putchar**()

respectively, except that the caller is responsible for locking the stream with flockfile(3) before calling them. These functions may be used to avoid the overhead of locking the stream for each character, and to avoid output being interspersed from multiple threads writing to the same stream.

RETURN VALUES

The functions, **fputc**(), **putc**(), **putchar**(), **putc_unlocked**() and **putchar_unlocked**() return the character written. If an error occurs, the value EOF is returned. The **putw**() function returns 0 on success; EOF is returned if a write error occurs, or if an attempt is made to write a read-only stream.

SEE ALSO

ferror(3), flockfile(3), fopen(3), getc(3), putwc(3), stdio(3)

STANDARDS

The functions **fputc**(), **putc**(), and **putchar**(), conform to ISO/IEC 9899:1990 ("ISO C90"). The **putc_unlocked**() and **putchar_unlocked**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1"). A function **putw**() function appeared in Version 6 AT&T UNIX.

BUGS

The size and byte order of an *int* varies from one machine to another, and **putw**() is not recommended for portable applications.

NAME

fputs, **puts** - output a line to a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
fputs(*const char *str*, *FILE *stream*);

int
puts(*const char *str*);

DESCRIPTION

The function **fputs**() writes the string pointed to by *str* to the stream pointed to by *stream*.

The function **puts**() writes the string *str*, and a terminating newline character, to the stream stdout.

RETURN VALUES

The functions **fputs**() and **puts**() return a nonnegative integer on success and EOF on error.

ERRORS

[EBADF] The *stream* argument is not a writable stream.

The functions **fputs**() and **puts**() may also fail and set *errno* for any of the errors specified for the routines write(2).

SEE ALSO

ferror(3), fputws(3), putc(3), stdio(3)

STANDARDS

The functions **fputs**() and **puts**() conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

fputwc, **putwc**, **putwchar** - output a wide character to a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

wint_t
fputwc(*wchar_t wc*, *FILE *stream*);

wint_t
putwc(*wchar_t wc*, *FILE *stream*);

wint_t
putwchar(*wchar_t wc*);

DESCRIPTION

The **fputwc**() function writes the wide character *wc* to the output stream pointed to by *stream*.

The **putwc**() function acts essentially identically to **fputwc**().

The **putwchar**() function is identical to **putwc**() with an output stream of stdout.

RETURN VALUES

The **fputwc**(), **putwc**(), and **putwchar**() functions return the wide character written. If an error occurs, the value WEOF is returned.

SEE ALSO

ferror(3), fopen(3), getwc(3), putc(3), stdio(3)

STANDARDS

The **fputwc**(), **putwc**(), and **putwchar**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

fputws - output a line of wide characters to a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

int
fputws(*const wchar_t * restrict ws*, *FILE * restrict fp*);

DESCRIPTION

The **fputws**() function writes the wide character string pointed to by *ws* to the stream pointed to by *fp*.

RETURN VALUES

The **fputws**() function returns 0 on success and -1 on error.

ERRORS

The **fputws**() function will fail if:

[EBADF] The *fp* argument supplied is not a writable stream.

The **fputws**() function may also fail and set *errno* for any of the errors specified for the routine write(2).

SEE ALSO

ferror(3), fputs(3), putwc(3), stdio(3)

STANDARDS

The **fputws**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

fread, **fwrite** - binary stream input/output

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

size_t
fread(*void * restrict ptr*, *size_t size*, *size_t nmemb*, *FILE * restrict stream*);

size_t
fwrite(*const void * restrict ptr*, *size_t size*, *size_t nmemb*, *FILE * restrict stream*);

DESCRIPTION

The function **fread**() reads *nmemb* objects, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function **fwrite**() writes *nmemb* objects, each *size* bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*.

RETURN VALUES

The functions **fread**() and **fwrite**() advance the file position indicator for the stream by the number of bytes read or written. They return the number of objects read or written. If an error occurs, or the end-of-file is reached, the return value is a short object count (or zero).

The function **fread**() does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to determine which occurred. The function **fwrite**() returns a value less than *nmemb* only if a write error has occurred.

SEE ALSO

read(2), write(2)

STANDARDS

The functions **fread**() and **fwrite**() conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

getipnodebyname, **getipnodebyaddr**, **freehostent** - nodename-to-address and address-to-nodename translation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

*struct hostent **
getipnodebyname(*const char *name*, *int af*, *int flags*, *int *error_num*);

*struct hostent **
getipnodebyaddr(*const void *src*, *size_t len*, *int af*, *int *error_num*);

void
freehostent(*struct hostent *ptr*);

DESCRIPTION

The **getipnodebyname**() and **getipnodebyaddr**() functions are very similar to gethostbyname(3), gethostbyname2(3) and gethostbyaddr(3). The functions cover all the functionalities provided by the older ones, and provide better interface to programmers. The functions require additional arguments, *af*, and *flags*, for specifying address family and operation mode. The additional arguments allow programmer to get address for a nodename, for specific address family (such as AF_INET or AF_INET6). The functions also require an additional pointer argument, *error_num* to return the appropriate error code, to support thread safe error code returns.

The type and usage of the return value, struct hostent is described in gethostbyname(3).

For **getipnodebyname**(), the *name* argument can be either a node name or a numeric address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex address). The *af* argument specifies the address family, either AF_INET or AF_INET6. The *flags* argument specifies the types of addresses that are searched for, and the types of addresses that are returned. We note that a special flags value of AI_DEFAULT (defined below) should handle most applications. That is, porting simple applications to use IPv6 replaces the call

```
hptr = gethostbyname(name);
```

with

```
hptr = getipnodebyname(name, AF_INET6, AI_DEFAULT, &error_num);
```

Applications desiring finer control over the types of addresses searched for and returned, can specify other combinations of the *flags* argument.

A *flags* of 0 implies a strict interpretation of the *af* argument:

- If *flags* is 0 and *af* is AF_INET, then the caller wants only IPv4 addresses. A query is made for A records. If successful, the IPv4 addresses are returned and the h_length member of the hostent structure will be 4, else the function returns a NULL pointer.

- If *flags* is 0 and if *af* is AF_INET6, then the caller wants only IPv6 addresses. A query is made for AAAA records. If successful, the IPv6 addresses are returned and the h_length member of the hostent structure will be 16, else the function returns a NULL pointer.

Other constants can be logically-ORed into the *flags* argument, to modify the behavior of the function.

- If the AI_V4MAPPED flag is specified along with an *af* of AF_INET6, then the caller will accept IPv4-mapped IPv6 addresses. That is, if no AAAA records are found then a query is made for A records and any found are returned as IPv4-mapped IPv6 addresses (h_length will be 16). The AI_V4MAPPED flag is ignored unless *af* equals AF_INET6.

- The AI_V4MAPPED_CFG flag is exact same as the AI_V4MAPPED flag only if the kernel supports IPv4-mapped IPv6 address.

- If the AI_ALL flag is used in conjunction with the AI_V4MAPPED flag, and only used with the IPv6 address family. When AI_ALL is logically or'd with AI_V4MAPPED flag then the caller wants all addresses: IPv6 and IPv4-mapped IPv6. A query is first made for AAAA records and if successful, the IPv6 addresses are returned. Another query is then made for A records and any found are returned as IPv4-mapped IPv6 addresses. h_length will be 16. Only if both queries fail does the function return a NULL pointer. This flag is ignored unless af equals AF_INET6. If both AI_ALL and AI_V4MAPPED are specified, AI_ALL takes precedence.

- The AI_ADDRCONFIG flag specifies that a query for AAAA records should occur only if the node has at least one IPv6 source address configured and a query for A records should occur only if the node has at least one IPv4 source address configured.

  For example, if the node has no IPv6 source addresses configured, and *af* equals AF_INET6, and the

node name being looked up has both AAAA and A records, then: (a) if only AI_ADDRCONFIG is specified, the function returns a NULL pointer; (b) if AI_ADDRCONFIG | AI_V4MAPPED is specified, the A records are returned as IPv4-mapped IPv6 addresses;

The special flags value of AI_DEFAULT is defined as

```
#define  AI_DEFAULT  (AI_V4MAPPED_CFG | AI_ADDRCONFIG)
```

We noted that the **getipnodebyname**() function must allow the *name* argument to be either a node name or a literal address string (i.e., a dotted-decimal IPv4 address or an IPv6 hex address). This saves applications from having to call inet_pton(3) to handle literal address strings. When the *name* argument is a literal address string, the *flags* argument is always ignored.

There are four scenarios based on the type of literal address string and the value of the *af* argument. The two simple cases are when *name* is a dotted-decimal IPv4 address and *af* equals AF_INET, or when *name* is an IPv6 hex address and *af* equals AF_INET6. The members of the returned hostent structure are: h_name points to a copy of the *name* argument, h_aliases is a NULL pointer, h_addrtype is a copy of the *af* argument, h_length is either 4 (for AF_INET) or 16 (for AF_INET6), h_addr_list[0] is a pointer to the 4-byte or 16-byte binary address, and h_addr_list[1] is a NULL pointer.

When *name* is a dotted-decimal IPv4 address and *af* equals AF_INET6, and AI_V4MAPPED is specified, an IPv4-mapped IPv6 address is returned: h_name points to an IPv6 hex address containing the IPv4-mapped IPv6 address, h_aliases is a NULL pointer, h_addrtype is AF_INET6, h_length is 16, h_addr_list[0] is a pointer to the 16-byte binary address, and h_addr_list[1] is a NULL pointer.

It is an error when *name* is an IPv6 hex address and *af* equals AF_INET. The function's return value is a NULL pointer and the value pointed to by *error_num* equals HOST_NOT_FOUND.

The **getipnodebyaddr**() function takes almost the same argument as gethostbyaddr(3), but adds a pointer to return an error number. Additionally it takes care of IPv4-mapped IPv6 addresses, and IPv4-compatible IPv6 addresses.

The **getipnodebyname**() and **getipnodebyaddr**() functions dynamically allocate the structure to be returned to the caller. The **freehostent**() function reclaims memory region allocated and returned by **getipnodebyname**() or **getipnodebyaddr**().

FILES

/etc/hosts
/etc/nsswitch.conf
/etc/resolv.conf

DIAGNOSTICS

The **getipnodebyname**() and **getipnodebyaddr**() functions returns NULL on errors. The integer values pointed to by *error_num* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The meanings of each error code are described in gethostbyname(3).

SEE ALSO

getaddrinfo(3), gethostbyaddr(3), gethostbyname(3), getnameinfo(3), hosts(5), nsswitch.conf(5), services(5), hostname(7), named(8)

R. Gilligan, S. Thomson, J. Bound, and W. Stevens, *Basic Socket Interface Extensions for IPv6*, RFC2553, March 1999.

STANDARDS

The **getipnodebyname**() and **getipnodebyaddr**() functions are documented in "Basic Socket Interface Extensions for IPv6" (RFC2553).

HISTORY

The implementation first appeared in KAME advanced networking kit.

BUGS

The **getipnodebyname**() and **getipnodebyaddr**() functions do not handle scoped IPv6 address properly. If you use these functions, your program will not be able to handle scoped IPv6 addresses. For IPv6 address manipulation, **getaddrinfo**(*3*) and **getnameinfo**(*3*) are recommended.

The text was shamelessly copied from RFC2553.

NAME

freelocale - Frees a locale created with duplocale(3) or newlocale(3)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <locale.h>

void
freelocale(*locale_t locale*);

DESCRIPTION

Frees a *locale_t*. This relinquishes any resources held exclusively by this locale. Note that locales share reference-counted components, so a call to this function is not guaranteed to free all of the components.

SEE ALSO

duplocale(3), localeconv(3), newlocale(3), querylocale(3), uselocale(3), xlocale(3)

STANDARDS

This function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

funopen, **fropen**, **fwopen** - open a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*FILE **
funopen(*const void *cookie*, *int (*readfn)(void *, char *, int)*, *int (*writefn)(void *, const char *, int)*, *fpos_t (*seekfn)(void *, fpos_t, int)*, *int (*closefn)(void *)*);

*FILE **
fropen(*void *cookie*, *int (*readfn)(void *, char *, int)*);

*FILE **
fwopen(*void *cookie*, *int (*writefn)(void *, const char *, int)*);

DESCRIPTION

The **funopen**() function associates a stream with up to four "I/O functions". Either *readfn* or *writefn* must be specified; the others can be given as an appropriately-typed NULL pointer. These I/O functions will be used to read, write, seek and close the new stream.

In general, omitting a function means that any attempt to perform the associated operation on the resulting stream will fail. If the close function is omitted, closing the stream will flush any buffered output and then succeed.

The calling conventions of *readfn*, *writefn*, *seekfn* and *closefn* must match those, respectively, of read(2), write(2), lseek(2), and close(2) with the single exception that they are passed the *cookie* argument specified to **funopen**() in place of the traditional file descriptor argument.

Read and write I/O functions are allowed to change the underlying buffer on fully buffered or line buffered streams by calling setvbuf(3). They are also not required to completely fill or empty the buffer. They are not, however, allowed to change streams from unbuffered to buffered or to change the state of the line buffering flag. They must also be prepared to have read or write calls occur on buffers other than the one most recently specified.

All user I/O functions can report an error by returning -1. Additionally, all of the functions should set the external variable *errno* appropriately if an error occurs.

An error on **closefn**() does not keep the stream open.

As a convenience, the include file <*stdio.h*> defines the macros **fropen**() and **fwopen**() as calls to **funopen**() with only a read or write function specified.

RETURN VALUES

Upon successful completion, **funopen**() returns a FILE pointer. Otherwise, NULL is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The **funopen**() function was called without either a read or write function. The **funopen**() function may also fail and set *errno* for any of the errors specified for the routine malloc(3).

SEE ALSO

fcntl(2), open(2), fclose(3), fopen(3), fopencookie(3), fseek(3), setbuf(3)

HISTORY

The **funopen**() functions first appeared in 4.4BSD.

BUGS

The **funopen**() function may not be portable to systems other than BSD.

The **funopen**() interface erroneously assumes that *fpos_t* is an integral type; see fseek(3) for a discussion of this issue.

NAME

scanf_l, **fscanf_l**, **sscanf_l**, **vfscanf_l**, **vscanf_l**, **vsscanf_l** - input format conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <xlocale.h>

int
scanf_l(*locale_t loc*, *const char * restrict format*, ...);

int
fscanf_l(*FILE * restrict stream*, *locale_t loc*, *const char * restrict format*, ...);

int
sscanf_l(*const char * restrict str*, *locale_t loc*, *const char * restrict format*, ...);

int
vfscanf_l(*FILE * restrict stream*, *locale_t loc*, *const char * restrict format*, *va_list ap*);

int
vscanf_l(*locale_t loc*, *const char * restrict format*, *va_list ap*);

int
vsscanf_l(*const char * restrict str*, *locale_t loc*, *const char * restrict format*, *va_list ap*);

DESCRIPTION

The above functions scan input according to a specified *format* in the locale *loc*. They behave in the same way as the versions without the _l suffix, but use the specific locale rather than the global or per-thread locale. See the specific manual pages for more information.

SEE ALSO

scanf(3), xlocale(3)

STANDARDS

These functions do not conform to any specific standard so they should be considered as non-portable local extensions.

HISTORY

These functions first appeared in Darwin and were first implemented in FreeBSD 9.1.

NAME

scanf, **fscanf**, **sscanf**, **vscanf**, **vsscanf**, **vfscanf** - input format conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
scanf(*const char * restrict format*, ...);

int
fscanf(*FILE * restrict stream*, *const char * restrict format*, ...);

int
sscanf(*const char * restrict str*, *const char * restrict format*, ...);

#include <stdarg.h>

int
vscanf(*const char * restrict format*, *va_list ap*);

int
vsscanf(*const char * restrict str*, *const char * restrict format*, *va_list ap*);

int
vfscanf(*FILE * restrict stream*, *const char * restrict format*, *va_list ap*);

DESCRIPTION

The **scanf**() family of functions scans input according to a *format* as described below. This format may contain *conversion specifiers*; the results from such conversions, if any, are stored through the *pointer* arguments. The **scanf**() function reads input from the standard input stream stdin, **fscanf**() reads input from the stream pointer *stream*, and **sscanf**() reads its input from the character string pointed to by *str*. The **vfscanf**() function is analogous to vfprintf(3) and reads input from the stream pointer *stream* using a variable argument list of pointers (see stdarg(3)). The **vscanf**() function scans a variable argument list from the standard input and the **vsscanf**() function scans it from a string; these are analogous to the **vprintf**() and **vsprintf**() functions respectively. Each successive *pointer* argument must correspond properly with each successive conversion specifier (but see the ***** conversion below). All conversions are introduced by the **%** (percent sign) character. The *format* string may also contain other characters.

White space (such as blanks, tabs, or newlines) in the *format* string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the **%** character introducing a conversion there may be a number of *flag* characters, as follows:

***** Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.

hh Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *char* (rather than *int*).

h Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *short int* (rather than *int*).

l (ell) Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long int* (rather than *int*), that the conversion will be one of **a**, **e**, **f**, or **g** and the next pointer is a pointer to *double* (rather than *float*), or that the conversion will be one of **c**, **s** or **[** and the next pointer is a pointer to an array of *wchar_t* (rather than *char*).

ll (ell ell) Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long long int* (rather than *int*).

L Indicates that the conversion will be one of **a**, **e**, **f**, or **g** and the next pointer is a pointer to *long double*.

j Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *intmax_t* (rather than *int*).

t Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *ptrdiff_t* (rather than *int*).

z Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *size_t* (rather than *int*).

q (deprecated.) Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long long int* (rather than *int*).

In addition to these flags, there may be an optional maximum field width, expressed as a decimal integer, between the **%** and the conversion. If no width is given, a default of "infinity" is used (with one exception, below); otherwise at most this many bytes are scanned in processing the conversion. In the case of the **lc**, **ls** and **l[** conversions, the field width specifies the maximum number of multibyte characters that will be scanned. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

The following conversions are available:

% Matches a literal '%'. That is, "%%" in the format string matches a single input '%' character. No conversion is done, and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

i Matches an optionally signed integer; the next pointer must be a pointer to *int*. The integer is read in base 16 if it begins with '0x' or '0X', in base 8 if it begins with '0', and in base 10 otherwise. Only characters that correspond to the base are used.

o Matches an octal integer; the next pointer must be a pointer to *unsigned int*.

u Matches an optionally signed decimal integer; the next pointer must be a pointer to *unsigned int*.

x, **X** Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to *unsigned int*.

a, **A**, **e**, **E**, **f**, **F**, **g**, **G**
Matches a floating-point number in the style of strtod(3). The next pointer must be a pointer to *float* (unless **l** or **L** is specified.)

s Matches a sequence of non-white-space characters; the next pointer must be a pointer to *char*, and the array must be large enough to accept all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed after conversion by mbrtowc(3).

S The same as **ls**.

c Matches a sequence of *width* count characters (default 1); the next pointer must be a pointer to

char, and there must be enough room for all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed after conversion by mbrtowc(3).

C The same as **lc**.

[Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to *char*, and there must be enough room for all the characters in the string, plus a terminating NUL character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket **[** character and a close bracket **]** character. The set *excludes* those characters if the first character after the open bracket is a circumflex **^**. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. The hyphen character **-** is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, '[^]0-9-]' means the set "everything except close bracket, zero through nine, and hyphen". The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed after conversion by mbrtowc(3).

p Matches a pointer value (as printed by '%p' in printf(3)); the next pointer must be a pointer to *void*.

n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to *int*. This is *not* a conversion, although it can be suppressed with the ***** flag.

The decimal point character is defined in the program's locale (category LC_NUMERIC).

For backwards compatibility, a "conversion" of '%\0' causes an immediate return of EOF.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no

conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a '%d' conversion. The value EOF is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

getc(3), mbrtowc(3), printf(3), strtod(3), strtol(3), strtoul(3), wscanf(3)

STANDARDS

The functions **fscanf**(), **scanf**(), **sscanf**(), **vfscanf**(), **vscanf**() and **vsscanf**() conform to ISO/IEC 9899:1999 ("ISO C99").

BUGS

Earlier implementations of **scanf** treated **%D**, **%E**, **%F**, **%O** and **%X** as their lowercase equivalents with an **l** modifier. In addition, **scanf** treated an unknown conversion character as **%d** or **%D**, depending on its case. This functionality has been removed.

Numerical strings are truncated to 512 characters; for example, **%f** and **%d** are implicitly **%512f** and **%512d**.

The **%n$** modifiers for positional arguments are not implemented.

The **scanf** family of functions do not correctly handle multibyte characters in the *format* argument.

NAME

stat, **lstat**, **fstat**, **fstatat** - get file status

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>

int
stat(*const char * restrict path*, *struct stat * restrict sb*);

int
lstat(*const char * restrict path*, *struct stat * restrict sb*);

int
fstat(*int fd*, *struct stat *sb*);

int
fstatat(*int fd*, *const char *path*, *struct stat *buf*, *int flag*);

DESCRIPTION

The **stat**() system call obtains information about the file pointed to by *path*. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The **lstat**() system call is like **stat**() except when the named file is a symbolic link, in which case **lstat**() returns information about the link, while **stat**() returns information about the file the link references.

The **fstat**() system call obtains the same information about an open file known by the file descriptor *fd*.

The **fstatat**() system call is equivalent to **stat**() and **lstat**() except when the *path* specifies a relative path. In this case the status is retrieved from a file relative to the directory associated with the file descriptor *fd* instead of the current working directory.

The values for the *flag* are constructed by a bitwise-inclusive OR of flags from this list, defined in <*fcntl.h*>:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, the status of the symbolic link is returned.

If **fstatat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **stat**() or **lstat**() respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in *flag*.

The *sb* argument is a pointer to a *stat* structure as defined by <*sys/stat.h*> and into which information is placed concerning the file.

The fields of *struct stat* related to the file system are:

st_dev Numeric ID of the device containing the file.

st_ino The file's inode number.

st_nlink Number of hard links to the file.

st_flags Flags enabled for the file. See chflags(2) for the list of flags and their description.

The *st_dev* and *st_ino* fields together identify the file uniquely within the system.

The time-related fields of *struct stat* are:

st_atim Time when file data was last accessed. Changed by the mknod(2), utimes(2), read(2) and readv(2) system calls.

st_mtim Time when file data was last modified. Changed by the mkdir(2), mkfifo(2), mknod(2), utimes(2), write(2) and writev(2) system calls.

st_ctim Time when file status was last changed (inode data modification). Changed by the chflags(2), chmod(2), chown(2), creat(2), link(2), mkdir(2), mkfifo(2), mknod(2), rename(2), rmdir(2), symlink(2), truncate(2), unlink(2), utimes(2), write(2) and writev(2) system calls.

st_birthtim Time when the inode was created.

These time-related macros are defined for compatibility:

```
#define    st_atime        st_atim.tv_sec
#define    st_mtime        st_mtim.tv_sec
#define    st_ctime        st_ctim.tv_sec
#ifndef _POSIX_SOURCE
```

```
#define    st_birthtime                st_birthtim.tv_sec
#endif

#ifndef _POSIX_SOURCE
#define    st_atimespec                st_atim
#define    st_mtimespec                st_mtim
#define    st_ctimespec                st_ctim
#define    st_birthtimespec    st_birthtim
#endif
```

Size-related fields of the *struct stat* are:

st_size File size in bytes.

st_blksize Optimal I/O block size for the file.

st_blocks Actual number of blocks allocated for the file in 512-byte units. As short symbolic links are stored in the inode, this number may be zero.

The access-related fields of *struct stat* are:

st_uid User ID of the file's owner.

st_gid Group ID of the file.

st_mode Status of the file (see below).

The status information word *st_mode* has these bits:

```
#define S_IFMT   0170000  /* type of file mask */
#define S_IFIFO  0010000  /* named pipe (fifo) */
#define S_IFCHR  0020000  /* character special */
#define S_IFDIR  0040000  /* directory */
#define S_IFBLK  0060000  /* block special */
#define S_IFREG  0100000  /* regular */
#define S_IFLNK  0120000  /* symbolic link */
#define S_IFSOCK 0140000  /* socket */
#define S_IFWHT  0160000  /* whiteout */
#define S_ISUID  0004000  /* set user id on execution */
#define S_ISGID  0002000  /* set group id on execution */
```

```
#define S_ISVTX  0001000  /* save swapped text even after use */
#define S_IRWXU  0000700  /* RWX mask for owner */
#define S_IRUSR  0000400  /* read permission, owner */
#define S_IWUSR  0000200  /* write permission, owner */
#define S_IXUSR  0000100  /* execute/search permission, owner */
#define S_IRWXG  0000070  /* RWX mask for group */
#define S_IRGRP  0000040  /* read permission, group */
#define S_IWGRP  0000020  /* write permission, group */
#define S_IXGRP  0000010  /* execute/search permission, group */
#define S_IRWXO  0000007  /* RWX mask for other */
#define S_IROTH  0000004  /* read permission, other */
#define S_IWOTH  0000002  /* write permission, other */
#define S_IXOTH  0000001  /* execute/search permission, other */
```

For a list of access modes, see <*sys/stat.h*>, access(2) and chmod(2). These macros are available to test whether a *st_mode* value passed in the *m* argument corresponds to a file of the specified type:

S_ISBLK(*m*) Test for a block special file.

S_ISCHR(*m*) Test for a character special file.

S_ISDIR(*m*) Test for a directory.

S_ISFIFO(*m*) Test for a pipe or FIFO special file.

S_ISLNK(*m*) Test for a symbolic link.

S_ISREG(*m*) Test for a regular file.

S_ISSOCK(*m*) Test for a socket.

S_ISWHT(*m*) Test for a whiteout.

The macros evaluate to a non-zero value if the test is true or to the value 0 if the test is false.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

COMPATIBILITY

Previous versions of the system used different types for the *st_dev*, *st_uid*, *st_gid*, *st_rdev*, *st_size*, *st_blksize* and *st_blocks* fields.

ERRORS

The **stat**() and **lstat**() system calls will fail if:

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] The *sb* or *path* argument points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EOVERFLOW] The file size in bytes cannot be represented correctly in the structure pointed to by *sb*.

The **fstat**() system call will fail if:

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

[EFAULT] The *sb* argument points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EOVERFLOW] The file size in bytes cannot be represented correctly in the structure pointed to by *sb*.

In addition to the errors returned by the **lstat**(), the **fstatat**() may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

access(2), chmod(2), chown(2), fhstat(2), statfs(2), utimes(2), sticky(7), symlink(7)

STANDARDS

The **stat**() and **fstat**() system calls are expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **fstatat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **stat**() and **fstat**() system calls appeared in Version 1 AT&T UNIX. The **lstat**() system call appeared in 4.2BSD. The **fstatat**() system call appeared in FreeBSD 8.0.

BUGS

Applying **fstat**() to a socket returns a zeroed buffer, except for the blocksize field, and a unique device and inode number.

NAME

statvfs, **fstatvfs** - retrieve file system information

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/statvfs.h>

int
statvfs(*const char * restrict path*, *struct statvfs * restrict buf*);

int
fstatvfs(*int fd*, *struct statvfs *buf*);

DESCRIPTION

The **statvfs**() and **fstatvfs**() functions fill the structure pointed to by *buf* with garbage. This garbage will occasionally bear resemblance to file system statistics, but portable applications must not depend on this. Applications must pass a pathname or file descriptor which refers to a file on the file system in which they are interested.

The *statvfs* structure contains the following members:

f_namemax	The maximum length in bytes of a file name on this file system. Applications should use pathconf(2) instead.
f_fsid	Not meaningful in this implementation.
f_frsize	The size in bytes of the minimum unit of allocation on this file system. (This corresponds to the *f_bsize* member of *struct statfs*.)
f_bsize	The preferred length of I/O requests for files on this file system. (Corresponds to the *f_iosize* member of *struct statfs*.)
f_flag	Flags describing mount options for this file system; see below.

In addition, there are three members of type *fsfilcnt_t*, which represent counts of file serial numbers (*i.e.*, inodes); these are named *f_files*, *f_favail*, and *f_ffree*, and represent the number of file serial numbers which exist in total, are available to unprivileged processes, and are available to privileged processes, respectively. Likewise, the members *f_blocks*, *f_bavail*, and *f_bfree* (all of type *fsblkcnt_t*) represent

the respective allocation-block counts.

There are two flags defined for the *f_flag* member:

ST_RDONLY The file system is mounted read-only.

ST_NOSUID The semantics of the S_ISUID and S_ISGID file mode bits are not supported by, or are disabled on, this file system.

IMPLEMENTATION NOTES

The **statvfs**() and **fstatvfs**() functions are implemented as wrappers around the **statfs**() and **fstatfs**() functions, respectively. Not all the information provided by those functions is made available through this interface.

RETURN VALUES

The **statvfs**() and **fstatvfs**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **statvfs**() and **fstatvfs**() functions may fail for any of the reasons documented for statfs(2) or fstatfs(2) and pathconf(2) or fpathconf(2), respectively. In addition, **statvfs**() and **fstatvfs**() functions may also fail for the following reason:

[EOVERFLOW] One or more of the file system statistics has a value which cannot be represented by the data types used in *struct statvfs*.

SEE ALSO

pathconf(2), statfs(2)

STANDARDS

The **statvfs**() and **fstatvfs**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1"). As standardized, portable applications cannot depend on these functions returning any valid information at all. This implementation attempts to provide as much useful information as is provided by the underlying file system, subject to the limitations of the specified data types.

HISTORY

The **statvfs**() and **fstatvfs**() functions first appeared in FreeBSD 5.0.

AUTHORS

The **statvfs**() and **fstatvfs**() functions and this manual page were written by Garrett Wollman

<*wollman@FreeBSD.org*>.

NAME

ftime - get date and time

LIBRARY

Compatibility Library (libcompat, -lcompat)

SYNOPSIS

#include <sys/types.h>
#include <sys/timeb.h>

int
ftime(*struct timeb *tp*);

DESCRIPTION

This interface is obsoleted by gettimeofday(2).

The **ftime**() routine fills in a structure pointed to by its argument, as defined by <*sys/timeb.h*>:

```
/*
 * Structure returned by ftime system call
 */
struct timeb
{
        time_t  time;
        unsigned short millitm;
        short   timezone;
        short   dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

SEE ALSO

gettimeofday(2), settimeofday(2), ctime(3), time(3)

HISTORY

The **ftime** function appeared in 4.2BSD.

NAME

ftok - create IPC identifier from path name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>

key_t
ftok(*const char *path*, *int id*);

DESCRIPTION

The **ftok**() function attempts to create a unique key suitable for use with the msgget(2), semget(2) and shmget(2) functions given the *path* of an existing file and a user-selectable *id*.

The specified *path* must specify an existing file that is accessible to the calling process or the call will fail. Also, note that links to files will return the same key, given the same *id*.

RETURN VALUES

The **ftok**() function will return -1 if *path* does not exist or if it cannot be accessed by the calling process.

SEE ALSO

msgget(2), semget(2), shmget(2)

HISTORY

The **ftok**() function originates with System V and is typically used by programs that use the System V IPC routines.

AUTHORS

Thorsten Lockert <*tholo@sigmasoft.com*>

BUGS

The returned key is computed based on the device minor number and inode of the specified *path* in combination with the lower 8 bits of the given *id*. Thus it is quite possible for the routine to return duplicate keys.

NAME

truncate, **ftruncate** - truncate or extend a file to a specified length

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
truncate(*const char *path*, *off_t length*);

int
ftruncate(*int fd*, *off_t length*);

DESCRIPTION

The **truncate**() system call causes the file named by *path* or referenced by *fd* to be truncated or extended to *length* bytes in size. If the file was larger than this size, the extra data is lost. If the file was smaller than this size, it will be extended as if by writing bytes with the value zero.

The **ftruncate**() system call causes the file or shared memory object backing the file descriptor *fd* to be truncated or extended to *length* bytes in size. The file descriptor must be a valid file descriptor open for writing. The file position pointer associated with the file descriptor *fd* will not be modified.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error. If the file to be modified is not a directory or a regular file, the **truncate**() call has no effect and returns the value 0.

ERRORS

The **truncate**() system call succeeds unless:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The named file is not writable by the user.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The named file has its immutable or append-only flag set, see the chflags(2) manual page for more information.

[EISDIR] The named file is a directory.

[EROFS] The named file resides on a read-only file system.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed.

[EFBIG] The *length* argument was greater than the maximum file size.

[EINVAL] The *length* argument was less than 0.

[EIO] An I/O error occurred updating the inode.

[EFAULT] The *path* argument points outside the process's allocated address space.

The **ftruncate**() system call succeeds unless:

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

[EINVAL] The *fd* argument references a file descriptor that is not a regular file or shared memory object.

[EINVAL] The *fd* descriptor is not open for writing.

SEE ALSO

chflags(2), open(2), shm_open(2)

HISTORY

The **truncate**() and **ftruncate**() system calls appeared in 4.2BSD.

BUGS

These calls should be generalized to allow ranges of bytes in a file to be discarded.

Use of **truncate**() to extend a file is not portable.

NAME

fts - traverse a file hierarchy

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fts.h>

*FTS **
fts_open(*char * const *path_argv*, *int options*,
*int (*compar)(const FTSENT * const *, const FTSENT * const *)*);

*FTSENT **
fts_read(*FTS *ftsp*);

*FTSENT **
fts_children(*FTS *ftsp*, *int options*);

int
fts_set(*FTS *ftsp*, *FTSENT *f*, *int options*);

void
fts_set_clientptr(*FTS *ftsp*, *void *clientdata*);

*void **
fts_get_clientptr(*FTS *ftsp*);

*FTS **
fts_get_stream(*FTSENT *f*);

int
fts_close(*FTS *ftsp*);

DESCRIPTION

The **fts** functions are provided for traversing UNIX file hierarchies. A simple overview is that the **fts_open**() function returns a "handle" on a file hierarchy, which is then supplied to the other **fts** functions. The function **fts_read**() returns a pointer to a structure describing one of the files in the file hierarchy. The function **fts_children**() returns a pointer to a linked list of structures, each of which describes one of the files contained in a directory in the hierarchy. In general, directories are visited two

distinguishable times; in pre-order (before any of their descendants are visited) and in post-order (after all of their descendants have been visited). Files are visited once. It is possible to walk the hierarchy "logically" (ignoring symbolic links) or physically (visiting symbolic links), order the walk of the hierarchy or prune and/or re-visit portions of the hierarchy.

Two structures are defined (and typedef'd) in the include file <*fts.h*>. The first is *FTS*, the structure that represents the file hierarchy itself. The second is *FTSENT*, the structure that represents a file in the file hierarchy. Normally, an *FTSENT* structure is returned for every file in the file hierarchy. In this manual page, "file" and "*FTSENT* structure" are generally interchangeable.

The *FTS* structure contains space for a single pointer, which may be used to store application data or per-hierarchy state. The **fts_set_clientptr**() and **fts_get_clientptr**() functions may be used to set and retrieve this pointer. This is likely to be useful only when accessed from the sort comparison function, which can determine the original *FTS* stream of its arguments using the **fts_get_stream**() function. The two get functions are also available as macros of the same name.

The *FTSENT* structure contains at least the following fields, which are described in greater detail below:

```
typedef struct _ftsent {
        int fts_info;                   /* status for FTSENT structure */
        char *fts_accpath;              /* access path */
        char *fts_path;                 /* root path */
        size_t fts_pathlen;             /* strlen(fts_path) */
        char *fts_name;                 /* file name */
        size_t fts_namelen;             /* strlen(fts_name) */
        long fts_level;                 /* depth (-1 to N) */
        int fts_errno;                  /* file errno */
        long long fts_number;           /* local numeric value */
        void *fts_pointer;              /* local address value */
        struct ftsent *fts_parent;      /* parent directory */
        struct ftsent *fts_link;        /* next file structure */
        struct ftsent *fts_cycle;       /* cycle structure */
        struct stat *fts_statp;         /* stat(2) information */
} FTSENT;
```

These fields are defined as follows:

fts_info One of the following values describing the returned *FTSENT* structure and the file it represents. With the exception of directories without errors (FTS_D), all of these entries are terminal, that is, they will not be revisited, nor will any of their descendants be visited.

FTS_D A directory being visited in pre-order.

FTS_DC A directory that causes a cycle in the tree. (The *fts_cycle* field of the *FTSENT* structure will be filled in as well.)

FTS_DEFAULT Any *FTSENT* structure that represents a file type not explicitly described by one of the other *fts_info* values.

FTS_DNR A directory which cannot be read. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_DOT A file named '.' or '..' which was not specified as a file name to **fts_open**() (see FTS_SEEDOT).

FTS_DP A directory being visited in post-order. The contents of the *FTSENT* structure will be unchanged from when the directory was visited in pre-order, except for the *fts_info* field.

FTS_ERR This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_F A regular file.

FTS_NS A file for which no stat(2) information was available. The contents of the *fts_statp* field are undefined. This is an error return, and the *fts_errno* field will be set to indicate what caused the error.

FTS_NSOK A file for which no stat(2) information was requested. The contents of the *fts_statp* field are undefined.

FTS_SL A symbolic link.

FTS_SLNONE A symbolic link with a non-existent target. The contents of the *fts_statp* field reference the file characteristic information for the symbolic link itself.

fts_accpath A path for accessing the file from the current directory.

fts_path The path for the file relative to the root of the traversal. This path contains the path specified to **fts_open**() as a prefix.

fts_pathlen The length of the string referenced by *fts_path*.

fts_name The name of the file.

fts_namelen The length of the string referenced by *fts_name*.

fts_level The depth of the traversal, numbered from -1 to N, where this file was found. The *FTSENT* structure representing the parent of the starting point (or root) of the traversal is numbered FTS_ROOTPARENTLEVEL (-1), and the *FTSENT* structure for the root itself is numbered FTS_ROOTLEVEL (0).

fts_errno Upon return of a *FTSENT* structure from the **fts_children**() or **fts_read**() functions, with its *fts_info* field set to FTS_DNR, FTS_ERR or FTS_NS, the *fts_errno* field contains the value of the external variable *errno* specifying the cause of the error. Otherwise, the contents of the *fts_errno* field are undefined.

fts_number This field is provided for the use of the application program and is not modified by the **fts** functions. It is initialized to 0.

fts_pointer This field is provided for the use of the application program and is not modified by the **fts** functions. It is initialized to NULL.

fts_parent A pointer to the *FTSENT* structure referencing the file in the hierarchy immediately above the current file, i.e., the directory of which this file is a member. A parent structure for the initial entry point is provided as well, however, only the *fts_level*, *fts_number* and *fts_pointer* fields are guaranteed to be initialized.

fts_link Upon return from the **fts_children**() function, the *fts_link* field points to the next structure in the NULL-terminated linked list of directory members. Otherwise, the contents of the *fts_link* field are undefined.

fts_cycle If a directory causes a cycle in the hierarchy (see FTS_DC), either because of a hard link between two directories, or a symbolic link pointing to a directory, the *fts_cycle* field of the structure will point to the *FTSENT* structure in the hierarchy that references the same file as the current *FTSENT* structure. Otherwise, the contents of the *fts_cycle* field are undefined.

fts_statp A pointer to stat(2) information for the file.

A single buffer is used for all of the paths of all of the files in the file hierarchy. Therefore, the *fts_path*

and *fts_accpath* fields are guaranteed to be NUL-terminated *only* for the file most recently returned by **fts_read**(). To use these fields to reference any files represented by other *FTSENT* structures will require that the path buffer be modified using the information contained in that *FTSENT* structure's *fts_pathlen* field. Any such modifications should be undone before further calls to **fts_read**() are attempted. The *fts_name* field is always NUL-terminated.

FTS_OPEN

The **fts_open**() function takes a pointer to an array of character pointers naming one or more paths which make up a logical file hierarchy to be traversed. The array must be terminated by a NULL pointer.

There are a number of options, at least one of which (either FTS_LOGICAL or FTS_PHYSICAL) must be specified. The options are selected by *or*'ing the following values:

FTS_COMFOLLOW
: This option causes any symbolic link specified as a root path to be followed immediately whether or not FTS_LOGICAL is also specified.

FTS_LOGICAL
: This option causes the **fts** routines to return *FTSENT* structures for the targets of symbolic links instead of the symbolic links themselves. If this option is set, the only symbolic links for which *FTSENT* structures are returned to the application are those referencing non-existent files. Either FTS_LOGICAL or FTS_PHYSICAL *must* be provided to the **fts_open**() function.

FTS_NOCHDIR
: To allow descending to arbitrary depths (independent of {PATH_MAX}) and improve performance, the **fts** functions change directories as they walk the file hierarchy. This has the side-effect that an application cannot rely on being in any particular directory during the traversal. The FTS_NOCHDIR option turns off this feature, and the **fts** functions will not change the current directory. Note that applications should not themselves change their current directory and try to access files unless FTS_NOCHDIR is specified and absolute pathnames were provided as arguments to **fts_open**().

FTS_NOSTAT
: By default, returned *FTSENT* structures reference file characteristic information (the *statp* field) for each file visited. This option relaxes that requirement as a performance optimization, allowing the **fts** functions to set the *fts_info* field to FTS_NSOK and leave the contents of the *statp* field undefined.

FTS_PHYSICAL
: This option causes the **fts** routines to return *FTSENT* structures for symbolic links themselves instead of the target files they point to. If this option is set, *FTSENT* structures for all symbolic links in the hierarchy are returned to the application.

Either FTS_LOGICAL or FTS_PHYSICAL *must* be provided to the **fts_open**() function.

FTS_SEEDOT By default, unless they are specified as path arguments to **fts_open**(), any files named '.' or '..' encountered in the file hierarchy are ignored. This option causes the **fts** routines to return *FTSENT* structures for them.

FTS_XDEV This option prevents **fts** from descending into directories that have a different device number than the file from which the descent began.

The argument **compar**() specifies a user-defined function which may be used to order the traversal of the hierarchy. It takes two pointers to pointers to *FTSENT* structures as arguments and should return a negative value, zero, or a positive value to indicate if the file referenced by its first argument comes before, in any order with respect to, or after, the file referenced by its second argument. The *fts_accpath*, *fts_path* and *fts_pathlen* fields of the *FTSENT* structures may *never* be used in this comparison. If the *fts_info* field is set to FTS_NS or FTS_NSOK, the *fts_statp* field may not either. If the **compar**() argument is NULL, the directory traversal order is in the order listed in *path_argv* for the root paths, and in the order listed in the directory for everything else.

FTS_READ

The **fts_read**() function returns a pointer to an *FTSENT* structure describing a file in the hierarchy. Directories (that are readable and do not cause cycles) are visited at least twice, once in pre-order and once in post-order. All other files are visited at least once. (Hard links between directories that do not cause cycles or symbolic links to symbolic links may cause files to be visited more than once, or directories more than twice.)

If all the members of the hierarchy have been returned, **fts_read**() returns NULL and sets the external variable *errno* to 0. If an error unrelated to a file in the hierarchy occurs, **fts_read**() returns NULL and sets *errno* appropriately. If an error related to a returned file occurs, a pointer to an *FTSENT* structure is returned, and *errno* may or may not have been set (see *fts_info*).

The *FTSENT* structures returned by **fts_read**() may be overwritten after a call to **fts_close**() on the same file hierarchy stream, or, after a call to **fts_read**() on the same file hierarchy stream unless they represent a file of type directory, in which case they will not be overwritten until after a call to **fts_read**() after the *FTSENT* structure has been returned by the function **fts_read**() in post-order.

FTS_CHILDREN

The **fts_children**() function returns a pointer to an *FTSENT* structure describing the first entry in a NULL-terminated linked list of the files in the directory represented by the *FTSENT* structure most recently returned by **fts_read**(). The list is linked through the *fts_link* field of the *FTSENT* structure, and

is ordered by the user-specified comparison function, if any. Repeated calls to **fts_children**() will recreate this linked list.

As a special case, if **fts_read**() has not yet been called for a hierarchy, **fts_children**() will return a pointer to the files in the logical directory specified to **fts_open**(), i.e., the arguments specified to **fts_open**(). Otherwise, if the *FTSENT* structure most recently returned by **fts_read**() is not a directory being visited in pre-order, or the directory does not contain any files, **fts_children**() returns NULL and sets *errno* to zero. If an error occurs, **fts_children**() returns NULL and sets *errno* appropriately.

The *FTSENT* structures returned by **fts_children**() may be overwritten after a call to **fts_children**(), **fts_close**() or **fts_read**() on the same file hierarchy stream.

Option may be set to the following value:

FTS_NAMEONLY Only the names of the files are needed. The contents of all the fields in the returned linked list of structures are undefined with the exception of the *fts_name* and *fts_namelen* fields.

FTS_SET

The function **fts_set**() allows the user application to determine further processing for the file *f* of the stream *ftsp*. The **fts_set**() function returns 0 on success, and -1 if an error occurs. *Option* must be set to one of the following values:

FTS_AGAIN Re-visit the file; any file type may be re-visited. The next call to **fts_read**() will return the referenced file. The *fts_stat* and *fts_info* fields of the structure will be reinitialized at that time, but no other fields will have been changed. This option is meaningful only for the most recently returned file from **fts_read**(). Normal use is for post-order directory visits, where it causes the directory to be re-visited (in both pre and post-order) as well as all of its descendants.

FTS_FOLLOW The referenced file must be a symbolic link. If the referenced file is the one most recently returned by **fts_read**(), the next call to **fts_read**() returns the file with the *fts_info* and *fts_statp* fields reinitialized to reflect the target of the symbolic link instead of the symbolic link itself. If the file is one of those most recently returned by **fts_children**(), the *fts_info* and *fts_statp* fields of the structure, when returned by **fts_read**(), will reflect the target of the symbolic link instead of the symbolic link itself. In either case, if the target of the symbolic link does not exist the fields of the returned structure will be unchanged and the *fts_info* field will be set to FTS_SLNONE.

If the target of the link is a directory, the pre-order return, followed by the return of all of its descendants, followed by a post-order return, is done.

FTS_SKIP No descendants of this file are visited. The file may be one of those most recently returned by either **fts_children**() or **fts_read**().

FTS_CLOSE

The **fts_close**() function closes a file hierarchy stream *ftsp* and restores the current directory to the directory from which **fts_open**() was called to open *ftsp*. The **fts_close**() function returns 0 on success, and -1 if an error occurs.

ERRORS

The function **fts_open**() may fail and set *errno* for any of the errors specified for the library functions open(2) and malloc(3).

The function **fts_close**() may fail and set *errno* for any of the errors specified for the library functions chdir(2) and close(2).

The functions **fts_read**() and **fts_children**() may fail and set *errno* for any of the errors specified for the library functions chdir(2), malloc(3), opendir(3), readdir(3) and stat(2).

In addition, **fts_children**(), **fts_open**() and **fts_set**() may fail and set *errno* as follows:

[EINVAL] The options were invalid, or the list were empty.

SEE ALSO

find(1), chdir(2), stat(2), ftw(3), qsort(3)

HISTORY

The **fts** interface was first introduced in 4.4BSD. The **fts_get_clientptr**(), **fts_get_stream**(), and **fts_set_clientptr**() functions were introduced in FreeBSD 5.0, principally to provide for alternative interfaces to the **fts** functionality using different data structures.

BUGS

The **fts_open**() function will automatically set the FTS_NOCHDIR option if the FTS_LOGICAL option is provided, or if it cannot open(2) the current directory.

NAME

futimens, **utimensat** - set file access and modification times

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>

int
futimens(*int fd*, *const struct timespec times[2]*);

int
utimensat(*int fd*, *const char *path*, *const struct timespec times[2]*, *int flag*);

DESCRIPTION

The access and modification times of the file named by *path* or referenced by *fd* are changed as specified by the argument *times*. The inode-change-time of the file is set to the current time.

If *path* specifies a relative path, it is relative to the current working directory if *fd* is AT_FDCWD and otherwise relative to the directory associated with the file descriptor *fd*.

The *tv_nsec* field of a *timespec* structure can be set to the special value UTIME_NOW to set the current time, or to UTIME_OMIT to leave the time unchanged. In either case, the *tv_sec* field is ignored.

If *times* is non-NULL, it is assumed to point to an array of two timespec structures. The access time is set to the value of the first element, and the modification time is set to the value of the second element. For file systems that support file birth (creation) times (such as UFS2), the birth time will be set to the value of the second element if the second element is older than the currently set birth time. To set both a birth time and a modification time, two calls are required; the first to set the birth time and the second to set the (presumably newer) modification time. Ideally a new system call will be added that allows the setting of all three times at once. If *times* is NULL, this is equivalent to passing a pointer to an array of two timespec structures with both *tv_nsec* fields set to UTIME_NOW.

If both *tv_nsec* fields are UTIME_OMIT, the timestamps remain unchanged and no permissions are needed for the file itself, although search permissions may be required for the path prefix. The call may or may not succeed if the named file does not exist.

If both *tv_nsec* fields are UTIME_NOW, the caller must be the owner of the file, have permission to write the file, or be the super-user.

For all other values of the timestamps, the caller must be the owner of the file or be the super-user.

The values for the *flag* argument of the **utimensat**() system call are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, the symbolic link's times are changed. By default, **utimensat**() changes the times of the file referenced by the symbolic link.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

COMPATIBILITY

If the running kernel does not support this system call, a wrapper emulates it using fstatat(2), futimesat(2) and lutimes(2). As a result, timestamps will be rounded down to the nearest microsecond, UTIME_OMIT is not atomic and AT_SYMLINK_NOFOLLOW is not available with a path relative to a file descriptor.

ERRORS

These system calls will fail if:

[EACCES] The *times* argument is NULL, or both *tv_nsec* values are UTIME_NOW, and the effective user ID of the process does not match the owner of the file, and is not the super-user, and write access is denied.

[EFAULT] The *times* argument points outside the process's allocated address space.

[EINVAL] The *tv_nsec* component of at least one of the values specified by the *times* argument has a value less than 0 or greater than 999999999 and is not equal to UTIME_NOW or UTIME_OMIT.

[EIO] An I/O error occurred while reading or writing the affected inode.

[EPERM] The *times* argument is not NULL nor are both *tv_nsec* values UTIME_NOW, nor are both *tv_nsec* values UTIME_OMIT and the calling process's effective user ID does not match the owner of the file and is not the super-user.

[EPERM] The named file has its immutable or append-only flag set, see the chflags(2) manual page for more information.

[EROFS] The file system containing the file is mounted read-only.

The **futimens**() system call will fail if:

[EBADF] The *fd* argument does not refer to a valid descriptor.

The **utimensat**() system call will fail if:

[EACCES] Search permission is denied for a component of the path prefix.

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor.

[EFAULT] The *path* argument points outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG]
A component of a pathname exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chflags(2), stat(2), symlink(2), utimes(2), utime(3), symlink(7)

STANDARDS

The **futimens**() and **utimensat**() system calls are expected to conform to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **futimens**() and **utimensat**() system calls appeared in FreeBSD 10.3.

NAME

utimes, **lutimes**, **futimes**, **futimesat** - set file access and modification times

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/time.h>

int
utimes(*const char *path*, *const struct timeval *times*);

int
lutimes(*const char *path*, *const struct timeval *times*);

int
futimes(*int fd*, *const struct timeval *times*);

int
futimesat(*int fd*, *const char *path*, *const struct timeval times[2]*);

DESCRIPTION

These interfaces are obsoleted by futimens(2) and utimensat(2) because they are not accurate to nanoseconds.

The access and modification times of the file named by *path* or referenced by *fd* are changed as specified by the argument *times*.

If *times* is NULL, the access and modification times are set to the current time. The caller must be the owner of the file, have permission to write the file, or be the super-user.

If *times* is non-NULL, it is assumed to point to an array of two timeval structures. The access time is set to the value of the first element, and the modification time is set to the value of the second element. For file systems that support file birth (creation) times (such as UFS2), the birth time will be set to the value of the second element if the second element is older than the currently set birth time. To set both a birth time and a modification time, two calls are required; the first to set the birth time and the second to set the (presumably newer) modification time. Ideally a new system call will be added that allows the setting of all three times at once. The caller must be the owner of the file or be the super-user.

In either case, the inode-change-time of the file is set to the current time.

The **lutimes**() system call is like **utimes**() except in the case where the named file is a symbolic link, in which case **lutimes**() changes the access and modification times of the link, while **utimes**() changes the times of the file the link references.

The **futimesat**() system call is equivalent to **utimes**() except in the case where *path* specifies a relative path. In this case the access and modification time is set to that of a file relative to the directory associated with the file descriptor *fd* instead of the current working directory. If **futimesat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **utimes**().

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

All of the system call will fail if:

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The *times* argument is NULL and the effective user ID of the process does not match the owner of the file, and is not the super-user, and write access is denied.

[EFAULT] The *path* or *times* argument points outside the process's allocated address space.

[EFAULT] The *times* argument points outside the process's allocated address space.

[EINVAL] The *tv_usec* component of at least one of the values specified by the *times* argument has a value less than 0 or greater than 999999.

[EIO] An I/O error occurred while reading or writing the affected inode.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENAMETOOLONG]
A component of a pathname exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The *times* argument is not NULL and the calling process's effective user ID does not match the owner of the file and is not the super-user.

[EPERM] The named file has its immutable or append-only flags set. See the chflags(2) manual page for more information.

[EROFS] The file system containing the file is mounted read-only.

The **futimes**() system call will fail if:

[EBADF] The *fd* argument does not refer to a valid descriptor.

In addition to the errors returned by the **utimes**(), the **futimesat**() may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chflags(2), stat(2), utimensat(2), utime(3)

STANDARDS

The **utimes**() function is expected to conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2"). The **futimesat**() system call follows The Open Group Extended API Set 2 specification but was replaced by **utimensat**() in IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **utimes**() system call appeared in 4.2BSD. The **futimes**() and **lutimes**() system calls first appeared in FreeBSD 3.0. The **futimesat**() system call appeared in FreeBSD 8.0.

NAME

fwide - get/set orientation of a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

int
fwide(*FILE *stream*, *int mode*);

DESCRIPTION

The **fwide**() function determines the orientation of the stream pointed at by *stream*.

If the orientation of *stream* has already been determined, **fwide**() leaves it unchanged. Otherwise, **fwide**() sets the orientation of *stream* according to *mode*.

If *mode* is less than zero, *stream* is set to byte-oriented. If it is greater than zero, *stream* is set to wide-oriented. Otherwise, *mode* is zero, and *stream* is unchanged.

RETURN VALUES

The **fwide**() function returns a value according to orientation after the call of **fwide**(); a value less than zero if byte-oriented, a value greater than zero if wide-oriented, and zero if the stream has no orientation.

SEE ALSO

ferror(3), fgetc(3), fgetwc(3), fopen(3), fputc(3), fputwc(3), freopen(3), stdio(3)

STANDARDS

The **fwide**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

wscanf, **fwscanf**, **swscanf**, **vwscanf**, **vswscanf**, **vfwscanf** - wide character input format conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

int
wscanf(*const wchar_t * restrict format*, ...);

int
fwscanf(*FILE * restrict stream*, *const wchar_t * restrict format*, ...);

int
swscanf(*const wchar_t * restrict str*, *const wchar_t * restrict format*, ...);

#include <stdarg.h>

int
vwscanf(*const wchar_t * restrict format*, *va_list ap*);

int
vswscanf(*const wchar_t * restrict str*, *const wchar_t * restrict format*, *va_list ap*);

int
vfwscanf(*FILE * restrict stream*, *const wchar_t * restrict format*, *va_list ap*);

DESCRIPTION

The **wscanf**() family of functions scans input according to a *format* as described below. This format may contain *conversion specifiers*; the results from such conversions, if any, are stored through the *pointer* arguments. The **wscanf**() function reads input from the standard input stream stdin, **fwscanf**() reads input from the stream pointer *stream*, and **swscanf**() reads its input from the wide character string pointed to by *str*. The **vfwscanf**() function is analogous to vfwprintf(3) and reads input from the stream pointer *stream* using a variable argument list of pointers (see stdarg(3)). The **vwscanf**() function scans a variable argument list from the standard input and the **vswscanf**() function scans it from a wide character string; these are analogous to the **vwprintf**() and **vswprintf**() functions respectively. Each successive *pointer* argument must correspond properly with each successive conversion specifier (but see the *****

conversion below). All conversions are introduced by the **%** (percent sign) character. The *format* string may also contain other characters. White space (such as blanks, tabs, or newlines) in the *format* string match any amount of white space, including none, in the input. Everything else matches only itself. Scanning stops when an input character does not match such a format character. Scanning also stops when an input conversion cannot be made (see below).

CONVERSIONS

Following the **%** character introducing a conversion there may be a number of *flag* characters, as follows:

***** Suppresses assignment. The conversion that follows occurs as usual, but no pointer is used; the result of the conversion is simply discarded.

hh Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *char* (rather than *int*).

h Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *short int* (rather than *int*).

l (ell) Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long int* (rather than *int*), that the conversion will be one of **a**, **e**, **f**, or **g** and the next pointer is a pointer to *double* (rather than *float*), or that the conversion will be one of **c** or **s** and the next pointer is a pointer to an array of *wchar_t* (rather than *char*).

ll (ell ell) Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *long long int* (rather than *int*).

L Indicates that the conversion will be one of **a**, **e**, **f**, or **g** and the next pointer is a pointer to *long double*.

j Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *intmax_t* (rather than *int*).

t Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *ptrdiff_t* (rather than *int*).

z Indicates that the conversion will be one of **dioux** or **n** and the next pointer is a pointer to a *size_t* (rather than *int*).

q (deprecated.) Indicates that the conversion will be one of **dioux** or **n** and the next pointer

is a pointer to a *long long int* (rather than *int*).

In addition to these flags, there may be an optional maximum field width, expressed as a decimal integer, between the **%** and the conversion. If no width is given, a default of "infinity" is used (with one exception, below); otherwise at most this many characters are scanned in processing the conversion. Before conversion begins, most conversions skip white space; this white space is not counted against the field width.

The following conversions are available:

% Matches a literal '%'. That is, "%%" in the format string matches a single input '%' character. No conversion is done, and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

i Matches an optionally signed integer; the next pointer must be a pointer to *int*. The integer is read in base 16 if it begins with '0x' or '0X', in base 8 if it begins with '0', and in base 10 otherwise. Only characters that correspond to the base are used.

o Matches an octal integer; the next pointer must be a pointer to *unsigned int*.

u Matches an optionally signed decimal integer; the next pointer must be a pointer to *unsigned int*.

x, **X** Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to *unsigned int*.

a, **A**, **e**, **E**, **f**, **F**, **g**, **G**
Matches a floating-point number in the style of wcstod(3). The next pointer must be a pointer to *float* (unless **l** or **L** is specified.)

s Matches a sequence of non-white-space wide characters; the next pointer must be a pointer to *char*, and the array must be large enough to accept the multibyte representation of all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed.

S The same as **ls**.

c Matches a sequence of *width* count wide characters (default 1); the next pointer must be a pointer to *char*, and there must be enough room for the multibyte representation of all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed.

C The same as **lc**.

[Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to *char*, and there must be enough room for the multibyte representation of all the characters in the string, plus a terminating NUL character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket **[** character and a close bracket **]** character. The set *excludes* those characters if the first character after the open bracket is a circumflex **^**. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. To include a hyphen in the set, make it the last character before the final close bracket; some implementations of **wscanf**() use "A-Z" to represent the range of characters between ‘A’ and ‘Z’. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out.

If an **l** qualifier is present, the next pointer must be a pointer to *wchar_t*, into which the input will be placed.

p Matches a pointer value (as printed by ‘%p’ in wprintf(3)); the next pointer must be a pointer to *void*.

n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to *int*. This is *not* a conversion, although it can be suppressed with the ***** flag.

The decimal point character is defined in the program’s locale (category LC_NUMERIC).

For backwards compatibility, a "conversion" of ‘%\0’ causes an immediate return of EOF.

RETURN VALUES

These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no

conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a ‘%d’ conversion. The value EOF is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

SEE ALSO

fgetwc(3), scanf(3), wcrtomb(3), wcstod(3), wcstol(3), wcstoul(3), wprintf(3)

STANDARDS

The **fwscanf**(), **wscanf**(), **swscanf**(), **vfwscanf**(), **vwscanf**() and **vswscanf**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

BUGS

In addition to the bugs documented in scanf(3), **wscanf**() does not support the "A-Z" notation for specifying character ranges with the character class conversion (‘**%[**’).

NAME

getbootfile - get kernel boot file name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <paths.h>

*const char **
getbootfile(*void*);

DESCRIPTION

The **getbootfile**() function retrieves the full pathname of the file from which the current kernel was loaded, and returns a static pointer to the name. A read/write interface to this information is available via the sysctl(3) MIB variable "kern.bootfile".

RETURN VALUES

If the call succeeds a string giving the pathname is returned. If it fails, a null pointer is returned and an error code is placed in the global location *errno*.

SEE ALSO

sysctl(3)

HISTORY

The **getbootfile**() function appeared in FreeBSD 2.0.

BUGS

If the boot blocks have not been modified to pass this information into the kernel at boot time, the static string "*/boot/kernel/kernel*" is returned instead of the real boot file.

NAME

getbsize - get preferred block size

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*char **
getbsize(*int *headerlenp*, *long *blocksizep*);

DESCRIPTION

The **getbsize**() function returns a preferred block size for reporting by system utilities df(1), du(1), ls(1) and systat(1), based on the value of the BLOCKSIZE environment variable. BLOCKSIZE may be specified directly in bytes, or in multiples of a kilobyte by specifying a number followed by ‘‘K’’ or ‘‘k’’, in multiples of a megabyte by specifying a number followed by ‘‘M’’ or ‘‘m’’ or in multiples of a gigabyte by specifying a number followed by ‘‘G’’ or ‘‘g’’. Multiples must be integers.

Valid values of BLOCKSIZE are 512 bytes to 1 gigabyte. Sizes less than 512 bytes are rounded up to 512 bytes, and sizes greater than 1 GB are rounded down to 1 GB. In each case **getbsize**() produces a warning message via warnx(3).

The **getbsize**() function returns a pointer to a null-terminated string describing the block size, something like "1K-blocks". The memory referenced by *headerlenp* is filled in with the length of the string (not including the terminating null). The memory referenced by *blocksizep* is filled in with block size, in bytes.

SEE ALSO

df(1), du(1), ls(1), systat(1), environ(7)

HISTORY

The **getbsize**() function first appeared in 4.4BSD.

NAME

getcontext, **getcontextx**, **setcontext** - get and set user thread context

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <ucontext.h>

int
getcontext(*ucontext_t *ucp*);

*ucontext_t **
getcontextx(*void*);

int
setcontext(*const ucontext_t *ucp*);

DESCRIPTION

The **getcontext**() function saves the current thread's execution context in the structure pointed to by *ucp*. This saved context may then later be restored by calling **setcontext**().

The **getcontextx**() function saves the current execution context in the newly allocated structure *ucontext_t*, which is returned on success. If architecture defines additional CPU states that can be stored in extended blocks referenced from the *ucontext_t*, the memory for them may be allocated and their context also stored. Memory returned by **getcontextx**() function shall be freed using **free**(*3*).

The **setcontext**() function makes a previously saved thread context the current thread context, i.e., the current context is lost and **setcontext**() does not return. Instead, execution continues in the context specified by *ucp*, which must have been previously initialized by a call to **getcontext**(), makecontext(3), or by being passed as an argument to a signal handler (see sigaction(2)).

If *ucp* was initialized by **getcontext**(), then execution continues as if the original **getcontext**() call had just returned (again).

If *ucp* was initialized by makecontext(3), execution continues with the invocation of the function specified to makecontext(3). When that function returns, *ucp->uc_link* determines what happens next: if *ucp->uc_link* is NULL, the process exits; otherwise, **setcontext**(*ucp->uc_link*) is implicitly invoked.

If *ucp* was initialized by the invocation of a signal handler, execution continues at the point the thread

was interrupted by the signal.

RETURN VALUES

If successful, **getcontext**() returns zero and **setcontext**() does not return; otherwise -1 is returned. The **getcontextx**() returns pointer to the allocated and initialized context on success, and *NULL* on failure.

ERRORS

No errors are defined for **getcontext**() or **setcontext**(). The **getcontextx**() may return the following errors in *errno*:

[ENOMEM] No memory was available to allocate for the context or some extended state.

SEE ALSO

sigaction(2), sigaltstack(2), makecontext(3), ucontext(3)

NAME

getcwd, **getwd** - get working directory pathname

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*char **
getcwd(*char *buf*, *size_t size*);

*char **
getwd(*char *buf*);

DESCRIPTION

The **getcwd**() function copies the absolute pathname of the current working directory into the memory referenced by *buf* and returns a pointer to *buf*. The *size* argument is the size, in bytes, of the array referenced by *buf*.

If *buf* is NULL, space is allocated as necessary to store the pathname. This space may later be free(3)'d.

The function **getwd**() is a compatibility routine which calls **getcwd**() with its *buf* argument and a size of MAXPATHLEN (as defined in the include file <*sys/param.h*>). Obviously, *buf* should be at least MAXPATHLEN bytes in length.

These routines have traditionally been used by programs to save the name of a working directory for the purpose of returning to it. A much faster and less error-prone method of accomplishing this is to open the current directory ('.') and use the fchdir(2) function to return.

RETURN VALUES

Upon successful completion, a pointer to the pathname is returned. Otherwise a NULL pointer is returned and the global variable *errno* is set to indicate the error. In addition, **getwd**() copies the error message associated with *errno* into the memory referenced by *buf*.

ERRORS

The **getcwd**() function will fail if:

[EINVAL] The *size* argument is zero.

[ENOENT] A component of the pathname no longer exists.

[ENOMEM] Insufficient memory is available.

[ERANGE] The *size* argument is greater than zero but smaller than the length of the pathname plus 1.

The **getcwd**() function may fail if:

[EACCES] Read or search permission was denied for a component of the pathname. This is only checked in limited cases, depending on implementation details.

SEE ALSO

chdir(2), fchdir(2), malloc(3), strerror(3)

STANDARDS

The **getcwd**() function conforms to IEEE Std 1003.1-1990 ("POSIX.1"). The ability to specify a NULL pointer and have **getcwd**() allocate memory as necessary is an extension.

HISTORY

The **getwd**() function appeared in 4.0BSD.

BUGS

The **getwd**() function does not do sufficient error checking and is not able to return very long, but valid, paths. It is provided for compatibility.

NAME

getdelim, **getline** - get a line from a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

ssize_t
getdelim(*char ** restrict linep*, *size_t * restrict linecapp*, *int delimiter*, *FILE * restrict stream*);

ssize_t
getline(*char ** restrict linep*, *size_t * restrict linecapp*, *FILE * restrict stream*);

DESCRIPTION

The **getdelim**() function reads a line from *stream*, delimited by the character *delimiter*. The **getline**() function is equivalent to **getdelim**() with the newline character as the delimiter. The delimiter character is included as part of the line, unless the end of the file is reached.

The caller may provide a pointer to a malloced buffer for the line in **linep*, and the capacity of that buffer in **linecapp*. These functions expand the buffer as needed, as if via **realloc**(). If *linep* points to a NULL pointer, a new buffer will be allocated. In either case, **linep* and **linecapp* will be updated accordingly.

RETURN VALUES

The **getdelim**() and **getline**() functions return the number of characters stored in the buffer, excluding the terminating NUL character. The value -1 is returned if an error occurs, or if end-of-file is reached.

EXAMPLES

The following code fragment reads lines from a file and writes them to standard output. The **fwrite**() function is used in case the line contains embedded NUL characters.

```
char *line = NULL;
size_t linecap = 0;
ssize_t linelen;
while ((linelen = getline(&line, &linecap, fp)) > 0)
        fwrite(line, linelen, 1, stdout);
free(line);
```

ERRORS

These functions may fail if:

[EINVAL] Either *linep* or *linecapp* is NULL.

[EOVERFLOW] No delimiter was found in the first SSIZE_MAX characters.

These functions may also fail due to any of the errors specified for **fgets**() and **malloc**().

SEE ALSO

fgetln(3), fgets(3), malloc(3)

STANDARDS

The **getdelim**() and **getline**() functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

These routines first appeared in FreeBSD 8.0.

BUGS

There are no wide character versions of **getdelim**() or **getline**().

NAME

getdirentries, **getdents** - get directory entries in a file system independent format

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>

ssize_t
getdirentries(*int fd*, *char *buf*, *size_t nbytes*, *off_t *basep*);

ssize_t
getdents(*int fd*, *char *buf*, *size_t nbytes*);

DESCRIPTION

The **getdirentries**() and **getdents**() system calls read directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a file system independent format. Up to *nbytes* of data will be transferred. The *nbytes* argument must be greater than or equal to the block size associated with the file, see stat(2). Some file systems may not support these system calls with buffers smaller than this size.

The data in the buffer is a series of *dirent* structures each containing the following entries:

```
ino_t     d_fileno;
off_t     d_off;
uint16_t  d_reclen;
uint8_t   d_type;
uint16_t  d_namlen;
char      d_name[MAXNAMLEN + 1];        /* see below */
```

The *d_fileno* entry is a number which is unique for each distinct file in the file system. Files that are linked by hard links (see link(2)) have the same *d_fileno*. The *d_reclen* entry is the length, in bytes, of the directory record. The *d_type* entry is the type of the file pointed to by the directory record. The file type values are defined in *<sys/dirent.h>*. The *d_name* entry contains a null terminated file name. The *d_namlen* entry specifies the length of the file name excluding the null byte. Thus the actual size of *d_name* may vary from 1 to MAXNAMLEN + 1.

Entries may be separated by extra space. The *d_reclen* entry may be used as an offset from the start of a

dirent structure to the next structure, if any.

The actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the next block of entries. The pointer may not advance by the number of bytes returned by **getdirentries**() or **getdents**(). A value of zero is returned when the end of the directory has been reached.

If the *basep* pointer value is non-NULL , the **getdirentries**() system call writes the position of the block read into the location pointed to by *basep*. Alternatively, the current position pointer may be set and retrieved by lseek(2). The current position pointer should only be set to a value returned by lseek(2), a value returned in the location pointed to by *basep* (**getdirentries**() only) or zero.

RETURN VALUES

If successful, the number of bytes actually transferred is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getdirentries**() system call will fail if:

[EBADF]	The *fd* argument is not a valid file descriptor open for reading.
[EFAULT]	Either *buf* or non-NULL *basep* point outside the allocated address space.
[EINVAL]	The file referenced by *fd* is not a directory, or *nbytes* is too small for returning a directory entry or block of entries, or the current position pointer is invalid.
[EIO]	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

lseek(2), open(2)

HISTORY

The **getdirentries**() system call first appeared in 4.4BSD. The **getdents**() system call first appeared in FreeBSD 3.0.

NAME

getdiskbyname - get generic disk description by its name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/disklabel.h>

*struct disklabel **
getdiskbyname(*const char *name*);

DESCRIPTION

The **getdiskbyname**() function takes a disk name (e.g. ‘rm03’) and returns a prototype disk label describing its geometry information and the standard disk partition tables. All information is obtained from the disktab(5) file.

SEE ALSO

disktab(5), disklabel(8)

HISTORY

The **getdiskbyname**() function appeared in 4.3BSD.

NAME

getdomainname, **setdomainname** - get/set the NIS domain name of current host

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getdomainname(*char *name*, *int namelen*);

int
setdomainname(*const char *name*, *int namelen*);

DESCRIPTION

The **getdomainname**() function returns the standard NIS domain name for the current host, as previously set by **setdomainname**(). The *namelen* argument specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

The **setdomainname**() function sets the NIS domain name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be returned by these calls:

[EFAULT] The *name* or *namelen* argument gave an invalid address.

[EPERM] The caller tried to set the hostname and was not the super-user.

SEE ALSO

gethostid(3), gethostname(3), sysctl(3)

HISTORY

The **getdomainname**() function appeared in 4.2BSD.

BUGS

Domain names are limited to MAXHOSTNAMELEN (from <*sys/param.h*>) characters, currently 256.

NAME

getdtablesize - get file descriptor limit

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getdtablesize(*void*);

DESCRIPTION

The **getdtablesize**() system call returns the maximum number of file descriptors that the current process may open. The maximum file descriptor number that the system may assign is the return value minus one. Existing file descriptor numbers may be higher if the limit was lowered after they were opened.

SEE ALSO

close(2), closefrom(2), dup(2), getrlimit(2), sysconf(3)

HISTORY

The **getdtablesize**() system call appeared in 4.2BSD.

NAME

getgid, **getegid** - get group process identification

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

gid_t
getgid(*void*);

gid_t
getegid(*void*);

DESCRIPTION

The **getgid**() system call returns the real group ID of the calling process, **getegid**() returns the effective group ID of the calling process.

The real group ID is specified at login time.

The real group ID is the group of the user who invoked the program. As the effective group ID gives the process additional permissions during the execution of "*set-group-ID*" mode processes, **getgid**() is used to determine the real-user-id of the calling process.

ERRORS

The **getgid**() and **getegid**() system calls are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getuid(2), issetugid(2), setgid(2), setregid(2)

STANDARDS

The **getgid**() and **getegid**() system calls are expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

NAME

getentropy - get entropy

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getentropy(*void *buf*, *size_t buflen*);

DESCRIPTION

getentropy() fills a buffer with high-quality random data.

The maximum *buflen* permitted is 256 bytes.

If it does not produce an error, **getentropy**() always provides the requested number of bytes of random data.

Similar to reading from */dev/urandom* just after boot, **getentropy**() may block until the system has collected enough entropy to seed the CSPRNG.

IMPLEMENTATION NOTES

The **getentropy**() function is implemented using getrandom(2).

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

getentropy() will succeed unless:

[EFAULT] The *buf* parameter points to an invalid address.

[EIO] Too many bytes requested, or some other fatal error occurred.

SEE ALSO

arc4random(3), getrandom(2), random(4)

STANDARDS

getentropy() is non-standard. It is present on OpenBSD and Linux.

HISTORY

The **getentropy**() function appeared in OpenBSD 5.6. The FreeBSD libc compatibility shim first appeared in FreeBSD 12.0.

NAME

getenv, **putenv**, **setenv**, **unsetenv** - environment variable functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*char **
getenv(*const char *name*);

int
setenv(*const char *name*, *const char *value*, *int overwrite*);

int
putenv(*char *string*);

int
unsetenv(*const char *name*);

DESCRIPTION

These functions set, unset and fetch environment variables from the host *environment list*.

The **getenv**() function obtains the current value of the environment variable, *name*. The application should not modify the string pointed to by the **getenv**() function.

The **setenv**() function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the given *value*. If the variable does exist, the argument *overwrite* is tested; if *overwrite* is zero, the variable is not reset, otherwise it is reset to the given *value*.

The **putenv**() function takes an argument of the form ‘‘name=value’’ and puts it directly into the current environment, so altering the argument shall change the environment. If the variable *name* does not exist in the list, it is inserted with the given *value*. If the variable *name* does exist, it is reset to the given *value*.

The **unsetenv**() function deletes all instances of the variable name pointed to by *name* from the list.

If corruption (e.g., a name without a value) is detected while making a copy of environ for internal

usage, then **setenv**(), **unsetenv**() and **putenv**() will output a warning to stderr about the issue, drop the corrupt entry and complete the task without error.

RETURN VALUES

The **getenv**() function returns the value of the environment variable as a NUL-terminated string. If the variable *name* is not in the current environment, NULL is returned.

The **setenv**(), **putenv**(), and **unsetenv**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The function **getenv**(), **setenv**() or **unsetenv**() failed because the *name* is a NULL pointer, points to an empty string, or points to a string containing an "=" character.

The function **putenv**() failed because *string* is a NULL pointer, *string* is without an "=" character or "=" is the first character in *string*. This does not follow the POSIX specification.

[ENOMEM] The function **setenv**(), **unsetenv**() or **putenv**() failed because they were unable to allocate memory for the environment.

SEE ALSO

csh(1), sh(1), execve(2), environ(7)

STANDARDS

The **getenv**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **setenv**(), **putenv**() and **unsetenv**() functions conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The functions **setenv**() and **unsetenv**() appeared in Version 7 AT&T UNIX. The **putenv**() function appeared in 4.3BSD-Reno.

Until FreeBSD 7.0, **putenv**() would make a copy of *string* and insert it into the environment using **setenv**(). This was changed to use *string* as the memory location of the ‘‘name=value’’ pair to follow the POSIX specification.

BUGS

Successive calls to **setenv**() that assign a larger-sized *value* than any previous value to the same *name* will result in a memory leak. The FreeBSD semantics for this function (namely, that the contents of *value* are copied and that old values remain accessible indefinitely) make this bug unavoidable. Future

versions may eliminate one or both of these semantic guarantees in order to fix the bug.

NAME

getuid, **geteuid** - get user identification

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

uid_t
getuid(*void*);

uid_t
geteuid(*void*);

DESCRIPTION

The **getuid**() system call returns the real user ID of the calling process. The **geteuid**() system call returns the effective user ID of the calling process.

The real user ID is that of the user who has invoked the program. As the effective user ID gives the process additional permissions during execution of "*set-user-ID*" mode processes, **getuid**() is used to determine the real-user-id of the calling process.

ERRORS

The **getuid**() and **geteuid**() system calls are always successful, and no return value is reserved to indicate an error.

SEE ALSO

getgid(2), issetugid(2), setgid(2), setreuid(2), setuid(2)

STANDARDS

The **geteuid**() and **getuid**() system calls are expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **getuid**() function appeared in Version 1 AT&T UNIX. The **geteuid**() function appeared in Version 4 AT&T UNIX.

NAME

getfh, **lgetfh** - get file handle

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/mount.h>

int
getfh(*const char *path*, *fhandle_t *fhp*);

int
lgetfh(*const char *path*, *fhandle_t *fhp*);

DESCRIPTION

The **getfh**() system call returns a file handle for the specified file or directory in the file handle pointed to by *fhp*. The **lgetfh**() system call is like **getfh**() except in the case where the named file is a symbolic link, in which case **lgetfh**() returns information about the link, while **getfh**() returns information about the file the link references. These system calls are restricted to the superuser.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getfh**() and **lgetfh**() system calls fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] The *fhp* argument points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

fhopen(2), open(2), stat(2)

HISTORY

The **getfh**() system call first appeared in 4.4BSD.

NAME

getfsstat - get list of all mounted file systems

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/ucred.h>
#include <sys/mount.h>

int
getfsstat(*struct statfs *buf*, *long bufsize*, *int mode*);

DESCRIPTION

The **getfsstat**() system call returns information about all mounted file systems. The *buf* argument is a pointer to *statfs* structures, as described in statfs(2).

Fields that are undefined for a particular file system are set to -1. The buffer is filled with an array of *statfs* structures, one for each mounted file system up to the byte count specified by *bufsize*. Note, the *bufsize* argument is the number of bytes that *buf* can hold, not the count of statfs structures it will hold.

If *buf* is given as NULL, **getfsstat**() returns just the number of mounted file systems.

Normally *mode* should be specified as MNT_WAIT. If *mode* is set to MNT_NOWAIT, **getfsstat**() will return the information it has available without requesting an update from each file system. Thus, some of the information will be out of date, but **getfsstat**() will not block waiting for information from a file system that is unable to respond. It will also skip any file system that is in the process of being unmounted, even if the unmount would eventually fail.

RETURN VALUES

Upon successful completion, the number of *statfs* structures is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getfsstat**() system call fails if one or more of the following are true:

[EFAULT] The *buf* argument points to an invalid address.

[EINVAL] *mode* is set to a value other than MNT_WAIT or MNT_NOWAIT.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

statfs(2), fstab(5), mount(8)

HISTORY

The **getfsstat**() system call first appeared in 4.4BSD.

NAME

getgrouplist - calculate group access list

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getgrouplist(*const char *name*, *gid_t basegid*, *gid_t *groups*, *int *ngroups*);

DESCRIPTION

The **getgrouplist**() function reads through the group file and calculates the group access list for the user specified in *name*. The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

The resulting group list is returned in the array pointed to by *groups*. The caller specifies the size of the *groups* array in the integer pointed to by *ngroups*; the actual number of groups found is returned in *ngroups*.

RETURN VALUES

The **getgrouplist**() function returns 0 on success and -1 if the size of the group list is too small to hold all the user's groups. Here, the group array will be filled with as many groups as will fit.

FILES

/etc/group group membership list

SEE ALSO

setgroups(2), initgroups(3)

HISTORY

The **getgrouplist**() function first appeared in 4.4BSD.

NAME

getgroups - get group access list

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getgroups(*int gidsetlen*, *gid_t *gidset*);

DESCRIPTION

The **getgroups**() system call gets the current group access list of the user process and stores it in the array *gidset*. The *gidsetlen* argument indicates the number of entries that may be placed in *gidset*. The **getgroups**() system call returns the actual number of groups returned in *gidset*. At least one and as many as {NGROUPS_MAX}+1 values may be returned. If *gidsetlen* is zero, **getgroups**() returns the number of supplementary group IDs associated with the calling process without modifying the array pointed to by *gidset*.

The value of {NGROUPS_MAX} should be obtained using sysconf(3) to avoid hard-coding it into the executable.

RETURN VALUES

A successful call returns the number of groups in the group set. A value of -1 indicates that an error occurred, and the error code is stored in the global variable *errno*.

ERRORS

The possible errors for **getgroups**() are:

[EINVAL] The argument *gidsetlen* is smaller than the number of groups in the group set.

[EFAULT] The argument *gidset* specifies an invalid address.

SEE ALSO

setgroups(2), initgroups(3), sysconf(3)

STANDARDS

The **getgroups**() system call conforms to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **getgroups**() system call appeared in 4.2BSD.

NAME

gethostid, **sethostid** - get/set unique identifier of current host

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

long
gethostid(*void*);

void
sethostid(*long hostid*);

DESCRIPTION

The **sethostid**() function establishes a 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

The **gethostid**() function returns the 32-bit identifier for the current processor.

This function has been deprecated. The hostid should be set or retrieved by use of sysctl(3).

SEE ALSO

gethostname(3), sysctl(3), sysctl(8)

HISTORY

The **gethostid**() and **sethostid**() syscalls appeared in 4.2BSD and were dropped in 4.4BSD.

BUGS

32 bits for the identifier is too small.

NAME

gethostname, **sethostname** - get/set name of current host

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
gethostname(*char *name*, *size_t namelen*);

int
sethostname(*const char *name*, *int namelen*);

DESCRIPTION

The **gethostname**() function returns the standard host name for the current processor, as previously set by **sethostname**(). The *namelen* argument specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

The **sethostname**() function sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

Host names are limited to {HOST_NAME_MAX} characters, not including the trailing null, currently 255.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be returned by these calls:

[EFAULT] The *name* or *namelen* argument gave an invalid address.

[ENAMETOOLONG]
The current host name is longer than *namelen*. (For **gethostname**() only.)

[EPERM] The caller tried to set the host name and was not the super-user.

SEE ALSO

sysconf(3), sysctl(3)

STANDARDS

The **gethostname**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1"). Callers should be aware that {HOST_NAME_MAX} may be variable or infinite, but is guaranteed to be no less than {_POSIX_HOST_NAME_MAX}. On older systems, this limit was defined in the non-standard header <*sys/param.h*> as MAXHOSTNAMELEN, and counted the terminating null. The **sethostname**() function and the error returns for **gethostname**() are not standardized.

HISTORY

The **gethostname**() function appeared in 4.2BSD. The *namelen* argument to **gethostname**() was changed to *size_t* in FreeBSD 5.2 for alignment with IEEE Std 1003.1-2001 ("POSIX.1").

NAME

getitimer, **setitimer** - get/set value of interval timer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

#define ITIMER_REAL     0
#define ITIMER_VIRTUAL  1
#define ITIMER_PROF     2

int
getitimer(int which, struct itimerval *value);

int
setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

DESCRIPTION

The system provides each process with three interval timers, defined in <*sys/time.h*>. The **getitimer**() system call returns the current value for the timer specified in *which* in the structure at *value*. The **setitimer**() system call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is not a null pointer).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval; /* timer interval */
        struct  timeval it_value;    /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution (typically 10 milliseconds).

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

The maximum number of seconds allowed for *it_interval* and *it_value* in **setitimer**() is 100000000.

NOTES

Three macros for manipulating time values are defined in <*sys/time.h*>. The **timerclear**() macro sets a time value to zero, **timerisset**() tests if a time value is non-zero, and **timercmp**() compares two time values.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getitimer**() and **setitimer**() system calls will fail if:

[EFAULT] The *value* argument specified a bad address.

[EINVAL] The *value* argument specified a time that was too large to be handled.

SEE ALSO

gettimeofday(2), select(2), sigaction(2), clocks(7)

HISTORY

The **getitimer**() system call appeared in 4.2BSD.

NAME

getloadavg - get system load averages

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
getloadavg(*double loadavg[]*, *int nelem*);

DESCRIPTION

The **getloadavg**() function returns the number of processes in the system run queue averaged over various periods of time. Up to *nelem* samples are retrieved and assigned to successive elements of *loadavg*[]. The system imposes a maximum of 3 samples, representing averages over the last 1, 5, and 15 minutes, respectively.

DIAGNOSTICS

If the load average was unobtainable, -1 is returned; otherwise, the number of samples actually retrieved is returned.

SEE ALSO

uptime(1), kvm_getloadavg(3), sysctl(3)

HISTORY

The **getloadavg**() function appeared in 4.3BSD-Reno.

NAME

getlogin, **getlogin_r**, **setlogin** - get/set login name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*char **
getlogin(*void*);

#include <sys/param.h>

int
getlogin_r(*char *name*, *int len*);

int
setlogin(*const char *name*);

DESCRIPTION

The **getlogin**() routine returns the login name of the user associated with the current session, as previously set by **setlogin**(). The name is normally associated with a login shell at the time a session is created, and is inherited by all processes descended from the login shell. (This is true even if some of those processes assume another user ID, for example when su(1) is used).

The **getlogin_r**() function provides the same service as **getlogin**() except the caller must provide the buffer *name* with length *len* bytes to hold the result. The buffer should be at least MAXLOGNAME bytes in length.

The **setlogin**() system call sets the login name of the user associated with the current session to *name*. This system call is restricted to the super-user, and is normally used only when a new session is being created on behalf of the named user (for example, at login time, or when a remote shell is invoked).

NOTE: There is only one login name per session.

It is *CRITICALLY* important to ensure that **setlogin**() is only ever called after the process has taken adequate steps to ensure that it is detached from its parent's session. Making a **setsid**() system call is the *ONLY* way to do this. The daemon(3) function calls **setsid**() which is an ideal way of detaching from a controlling terminal and forking into the background.

In particular, doing a **ioctl**(*ttyfd*, *TIOCNOTTY*, ...) or **setpgrp**(...) is *NOT* sufficient.

Once a parent process does a **setsid**() system call, it is acceptable for some child of that process to then do a **setlogin**() even though it is not the session leader, but beware that ALL processes in the session will change their login name at the same time, even the parent.

This is not the same as the traditional UNIX behavior of inheriting privilege.

Since the **setlogin**() system call is restricted to the super-user, it is assumed that (like all other privileged programs) the programmer has taken adequate precautions to prevent security violations.

RETURN VALUES

If a call to **getlogin**() succeeds, it returns a pointer to a null-terminated string in a static buffer, or NULL if the name has not been set. The **getlogin_r**() function returns zero if successful, or the error number upon failure.

The **setlogin**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* argument gave an invalid address.
[EINVAL]	The *name* argument pointed to a string that was too long. Login names are limited to MAXLOGNAME (from <*sys/param.h*>) characters, currently 33 including null.
[EPERM]	The caller tried to set the login name and was not the super-user.
[ERANGE]	The size of the buffer is smaller than the result to be returned.

SEE ALSO

setsid(2), daemon(3)

STANDARDS

The **getlogin**() system call and the **getlogin_r**() function conform to ISO/IEC 9945-1:1996 ("POSIX.1").

HISTORY

The **getlogin**() system call first appeared in 4.4BSD. The return value of **getlogin_r**() was changed from

earlier versions of FreeBSD to be conformant with ISO/IEC 9945-1:1996 ("POSIX.1").

BUGS

In earlier versions of the system, **getlogin**() failed unless the process was associated with a login terminal. The current implementation (using **setlogin**()) allows getlogin to succeed even when the process has no controlling terminal. In earlier versions of the system, the value returned by **getlogin**() could not be trusted without checking the user ID. Portable programs should probably still make this check.

NAME

getloginclass, **setloginclass** - get/set login class

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getloginclass(*char *name*, *size_t len*);

int
setloginclass(*const char *name*);

DESCRIPTION

The **getloginclass**() routine returns the login class name associated with the calling process, as previously set by **setloginclass**(). The caller must provide the buffer *name* with length *len* bytes to hold the result. The buffer should be at least MAXLOGNAME bytes in length.

The **setloginclass**() system call sets the login class of the calling process to *name*. This system call is restricted to the super-user, and is normally used only when a new session is being created on behalf of the named user (for example, at login time, or when a remote shell is invoked). Processes inherit login class from their parents.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be returned by these calls:

[EFAULT] The *name* argument gave an invalid address.

[EINVAL] The *name* argument pointed to a string that was too long. Login class names are limited to MAXLOGNAME (from <*sys/param.h*>) characters, currently 33 including null.

[EPERM] The caller tried to set the login class and was not the super-user.

[ENAMETOOLONG]
The size of the buffer is smaller than the result to be returned.

SEE ALSO

ps(1), setusercontext(3), login.conf(5), rctl(8)

HISTORY

The **getloginclass**() and **setloginclass**() system calls first appeared in FreeBSD 9.0.

NAME

getmntinfo - get information about mounted file systems

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/ucred.h>
#include <sys/mount.h>

int
getmntinfo(*struct statfs **mntbufp*, *int mode*);

DESCRIPTION

The **getmntinfo**() function returns an array of **statfs**() structures describing each currently mounted file system (see statfs(2)).

The **getmntinfo**() function passes its *mode* argument transparently to getfsstat(2).

RETURN VALUES

On successful completion, **getmntinfo**() returns a count of the number of elements in the array. The pointer to the array is stored into *mntbufp*.

If an error occurs, zero is returned and the external variable *errno* is set to indicate the error. Although the pointer *mntbufp* will be unmodified, any information previously returned by **getmntinfo**() will be lost.

ERRORS

The **getmntinfo**() function may fail and set errno for any of the errors specified for the library routines getfsstat(2) or malloc(3).

SEE ALSO

getfsstat(2), mount(2), statfs(2), mount(8)

HISTORY

The **getmntinfo**() function first appeared in 4.4BSD.

BUGS

The **getmntinfo**() function writes the array of structures to an internal static object and returns a pointer

to that object. Subsequent calls to **getmntinfo**() will modify the same object.

The memory allocated by **getmntinfo**() cannot be free(3)'d by the application.

NAME

getmode, **setmode** - modify mode bits

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

mode_t
getmode(*const void *set*, *mode_t mode*);

*void **
setmode(*const char *mode_str*);

DESCRIPTION

The **getmode**() function returns a copy of the file permission bits *mode* as altered by the values pointed to by *set*. While only the mode bits are altered, other parts of the file mode may be examined.

The **setmode**() function takes an absolute (octal) or symbolic value, as described in chmod(1), as an argument and returns a pointer to mode values to be supplied to **getmode**(). Because some of the symbolic values are relative to the file creation mask, **setmode**() may call umask(2). If this occurs, the file creation mask will be restored before **setmode**() returns. If the calling program changes the value of its file creation mask after calling **setmode**(), **setmode**() must be called again if **getmode**() is to modify future file modes correctly.

If the mode passed to **setmode**() is invalid or if memory cannot be allocated for the return value, **setmode**() returns NULL.

The value returned from **setmode**() is obtained from **malloc**() and should be returned to the system with **free**() when the program is done with it, generally after a call to **getmode**().

ERRORS

The **setmode**() function may fail and set errno for any of the errors specified for the library routine malloc(3) or strtol(3). In addition, **setmode**() will fail and set *errno* to:

[EINVAL] The *mode* argument does not represent a valid mode.

SEE ALSO

chmod(1), stat(2), umask(2), malloc(3)

HISTORY

The **getmode**() and **setmode**() functions first appeared in 4.4BSD.

NAME

getopt_long, **getopt_long_only** - get long options from command line argument list

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <getopt.h>

*extern char *optarg*;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;

int
getopt_long(*int argc*, *char * const *argv*, *const char *optstring*, *const struct option *longopts*, *int *longindex*);

int
getopt_long_only(*int argc*, *char * const *argv*, *const char *optstring*, *const struct option *longopts*, *int *longindex*);

DESCRIPTION

The **getopt_long**() function is similar to getopt(3) but it accepts options in two forms: words and characters. The **getopt_long**() function provides a superset of the functionality of getopt(3). The **getopt_long**() function can be used in two ways. In the first way, every long option understood by the program has a corresponding short option, and the option structure is only used to translate from long options to short options. When used in this fashion, **getopt_long**() behaves identically to getopt(3). This is a good way to add long option processing to an existing program with the minimum of rewriting.

In the second mechanism, a long option sets a flag in the *option* structure passed, or will store a pointer to the command line argument in the *option* structure passed to it for options that take arguments. Additionally, the long option's argument may be specified as a single argument with an equal sign, e.g.,

```
myprogram --myoption=somevalue
```

When a long option is processed, the call to **getopt_long**() will return 0. For this reason, long option processing without shortcuts is not backwards compatible with getopt(3).

It is possible to combine these methods, providing for long options processing with short option equivalents for some options. Less frequently used options would be processed as long options only.

The **getopt_long**() call requires a structure to be initialized describing the long options. The structure is:

```
struct option {
        char *name;
        int has_arg;
        int *flag;
        int val;
};
```

The *name* field should contain the option name without the leading double dash.

The *has_arg* field should be one of:

no_argument	no argument to the option is expected
required_argument	an argument to the option is required
optional_argument	an argument to the option may be presented

If *flag* is not NULL, then the integer pointed to by it will be set to the value in the *val* field. If the *flag* field is NULL, then the *val* field will be returned. Setting *flag* to NULL and setting *val* to the corresponding short option will make this function act just like getopt(3).

If the *longindex* field is not NULL, then the integer pointed to by it will be set to the index of the long option relative to *longopts*.

The last element of the *longopts* array has to be filled with zeroes.

The **getopt_long_only**() function behaves identically to **getopt_long**() with the exception that long options may start with '-' in addition to '--'. If an option starting with '-' does not match a long option but does match a single-character option, the single-character option is returned.

RETURN VALUES

If the *flag* field in *struct option* is NULL, **getopt_long**() and **getopt_long_only**() return the value specified in the *val* field, which is usually just the corresponding short option. If *flag* is not NULL, these functions return 0 and store *val* in the location pointed to by *flag*.

These functions return ':' if there was a missing option argument and error messages are suppressed, '?' if the user specified an unknown or ambiguous option, and -1 when the argument list has been

exhausted. The default behavior when a missing option argument is encountered is to write an error and return ‘?’. Specifying ‘:’ in *optstr* will cause the error message to be suppressed and ‘:’ to be returned instead.

In addition to ‘:’, a leading ‘+’ or ‘-’ in *optstr* also has special meaning. If either of these are specified, they must appear before ‘:’.

A leading ‘+’ indicates that processing should be halted at the first non-option argument, matching the default behavior of getopt(3). The default behavior without ‘+’ is to permute non-option argments to the end of *argv*.

A leading ‘-’ indicates that all non-option arguments should be treated as if they are arguments to a literal ‘1’ flag (i.e., the function call will return the value 1, rather than the char literal ’1’).

ENVIRONMENT

POSIXLY_CORRECT If set, option processing stops when the first non-option is found and a leading ‘-’ or ‘+’ in the *optstring* is ignored.

EXAMPLES

```
int bflag, ch, fd;
int daggerset;

/* options descriptor */
static struct option longopts[] = {
        { "buffy",no_argument,          NULL,       'b' },
        { "fluoride",   required_argument,NULL,     'f' },
        { "daggerset",  no_argument,        &daggerset,     1 },
        { NULL,         0,                  NULL,           0 }
};

bflag = 0;
while ((ch = getopt_long(argc, argv, "bf:", longopts, NULL)) != -1) {
        switch (ch) {
        case 'b':
                bflag = 1;
                break;
        case 'f':
                if ((fd = open(optarg, O_RDONLY, 0)) == -1)
                        err(1, "unable to open %s", optarg);
                break;
```

```
                case 0:
                        if (daggerset) {
                                fprintf(stderr,"Buffy will use her dagger to "
                                    "apply fluoride to dracula's teeth\n");
                        }
                        break;
                default:
                        usage();
                }
        }
        argc -= optind;
        argv += optind;
```

IMPLEMENTATION DIFFERENCES

This section describes differences to the GNU implementation found in glibc-2.1.3:

- Setting of *optopt* for long options with *flag* != NULL:

  GNU
  : sets *optopt* to *val*.

  BSD
  : sets *optopt* to 0 (since *val* would never be returned).

- Setting of *optarg* for long options without an argument that are invoked via '-W' ('W;' in option string):

  GNU
  : sets *optarg* to the option name (the argument of '-W').

  BSD
  : sets *optarg* to NULL (the argument of the long option).

- Handling of '-W' with an argument that is not (a prefix to) a known long option ('W;' in option string):

  GNU
  : returns '-W' with *optarg* set to the unknown option.

  BSD

treats this as an error (unknown option) and returns '?' with *optopt* set to 0 and *optarg* set to NULL (as GNU's man page documents).

- BSD does not permute the argument vector at the same points in the calling sequence as GNU does. The aspects normally used by the caller (ordering after -1 is returned, value of *optind* relative to current positions) are the same, though. (We do fewer variable swaps.)

SEE ALSO

getopt(3)

HISTORY

The **getopt_long**() and **getopt_long_only**() functions first appeared in the GNU libiberty library. The first BSD implementation of **getopt_long**() appeared in NetBSD 1.5, the first BSD implementation of **getopt_long_only**() in OpenBSD 3.3. FreeBSD first included **getopt_long**() in FreeBSD 5.0, **getopt_long_only**() in FreeBSD 5.2.

BUGS

The *argv* argument is not really *const* as its elements may be permuted (unless POSIXLY_CORRECT is set).

The implementation can completely replace getopt(3), but right now we are using separate code.

NAME

getopt - get option character from command line argument list

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*extern char *optarg*;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;

int
getopt(*int argc*, *char * const argv[]*, *const char *optstring*);

DESCRIPTION

The **getopt**() function incrementally parses a command line argument list *argv* and returns the next *known* option character. An option character is *known* if it has been specified in the string of accepted option characters, *optstring*.

The option string *optstring* may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow. If an individual character is followed by two colons, then the option argument is optional; *optarg* is set to the rest of the current *argv* word, or NULL if there were no more characters in the current word. This is a GNU extension. For example, an option string "x" recognizes an option "**-x**", and an option string "x:" recognizes an option and argument "**-x** *argument*". It does not matter to **getopt**() if a following argument has leading white space.

On return from **getopt**(), *optarg* points to an option argument, if it is anticipated, and the variable *optind* contains the index to the next *argv* argument for a subsequent call to **getopt**(). The variable *optopt* saves the last *known* option character returned by **getopt**().

The variables *opterr* and *optind* are both initialized to 1. The *optind* variable may be set to another value before a set of calls to **getopt**() in order to skip over more or less argv entries.

In order to use **getopt**() to evaluate multiple sets of arguments, or to evaluate a single set of arguments multiple times, the variable *optreset* must be set to 1 before the second and each additional set of calls to **getopt**(), and the variable *optind* must be reinitialized.

The **getopt**() function returns -1 when the argument list is exhausted. The interpretation of options in the argument list may be cancelled by the option '--' (double dash) which causes **getopt**() to signal the end of argument processing and return -1. When all options have been processed (i.e., up to the first non-option argument), **getopt**() returns -1.

RETURN VALUES

The **getopt**() function returns the next known option character in *optstring*. If **getopt**() encounters a character not found in *optstring* or if it detects a missing option argument, it returns '?' (question mark). If *optstring* has a leading ':' then a missing option argument causes ':' to be returned instead of '?'. In either case, the variable *optopt* is set to the character that caused the error. The **getopt**() function returns -1 when the argument list is exhausted.

EXAMPLES

```
#include <unistd.h>
int bflag, ch, fd;

bflag = 0;
while ((ch = getopt(argc, argv, "bf:")) != -1) {
        switch (ch) {
        case 'b':
                bflag = 1;
                break;
        case 'f':
                if ((fd = open(optarg, O_RDONLY, 0)) < 0) {
                        (void)fprintf(stderr,
                            "myname: %s: %s\n", optarg, strerror(errno));
                        exit(1);
                }
                break;
        case '?':
        default:
                usage();
        }
}
argc -= optind;
argv += optind;
```

DIAGNOSTICS

If the **getopt**() function encounters a character not found in the string *optstring* or detects a missing option argument it writes an error message to the stderr and returns '?'. Setting *opterr* to a zero will

disable these error messages. If *optstring* has a leading ‘:’ then a missing option argument causes a ‘:’ to be returned in addition to suppressing any error messages.

Option arguments are allowed to begin with "-"; this is reasonable but reduces the amount of error checking possible.

SEE ALSO

getopt(1), getopt_long(3), getsubopt(3)

STANDARDS

The *optreset* variable was added to make it possible to call the **getopt**() function multiple times. This is an extension to the IEEE Std 1003.2 ("POSIX.2") specification.

HISTORY

The **getopt**() function appeared in 4.3BSD.

BUGS

The **getopt**() function was once specified to return EOF instead of -1. This was changed by IEEE Std 1003.2-1992 ("POSIX.2") to decouple **getopt**() from <*stdio.h*>.

A single dash "-" may be specified as a character in *optstring*, however it should *never* have an argument associated with it. This allows **getopt**() to be used with programs that expect "-" as an option flag. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility *only*. Care should be taken not to use ‘-’ as the first character in *optstring* to avoid a semantic conflict with GNU **getopt**(), which assigns different meaning to an *optstring* that begins with a ‘-’. By default, a single dash causes **getopt**() to return -1.

It is also possible to handle digits as option letters. This allows **getopt**() to be used with programs that expect a number ("-3") as an option. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility *only*. The following code fragment works in most cases.

```
int ch;
long length;
char *p, *ep;

while ((ch = getopt(argc, argv, "0123456789")) != -1)
        switch (ch) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
```

```
                p = argv[optind - 1];
                if (p[0] == '-' && p[1] == ch && !p[2]) {
                        length = ch - '0';
                        ep = "";
                } else if (argv[optind] && argv[optind][1] == ch) {
                        length = strtol((p = argv[optind] + 1),
                            &ep, 10);
                        optind++;
                        optreset = 1;
                } else
                        usage();
                if (*ep != '\0')
                        errx(EX_USAGE, "illegal number -- %s", p);
                break;
        }
```

NAME

getosreldate - get the value of __FreeBSD_version

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getosreldate(*void*);

DESCRIPTION

The **getosreldate**() function returns an integer showing the version of the currently running FreeBSD kernel. Definitions of the values can be found in *The Porter's Handbook:*

https://www.FreeBSD.org/doc/en/books/porters-handbook/

RETURN VALUES

Upon successful completion, **getosreldate**() returns the value requested; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ENVIRONMENT

OSVERSION If the environment variable OSVERSION is set, it will override the **getosreldate**() return value.

EXAMPLES

An example can be found in */usr/share/examples/FreeBSD_version*.

ERRORS

The **getosreldate**() function may fail and set *errno* for any of the errors specified for the library function sysctl(3).

SEE ALSO

The Porter's Handbook.

HISTORY

The **getosreldate**() function appeared in FreeBSD 2.0.

NAME

getpagesize - get system page size

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
getpagesize(*void*);

DESCRIPTION

The **getpagesize**() function returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a system page size and may not be the same as the underlying hardware page size.

SEE ALSO

pagesize(1), sbrk(2)

HISTORY

The **getpagesize**() function appeared in 4.2BSD.

NAME

getpagesizes - get system page sizes

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
getpagesizes(*size_t pagesize[]*, *int nelem*);

DESCRIPTION

The **getpagesizes**() function retrieves page size information from the system. When it is called with *pagesize* specified as NULL and *nelem* specified as 0, it returns the number of distinct page sizes that are supported by the system. Otherwise, it assigns up to *nelem* of the system-supported page sizes to consecutive elements of the array referenced by *pagesize*. These page sizes are expressed in bytes. In this case, **getpagesizes**() returns the number of such page sizes that it assigned to the array.

RETURN VALUES

If successful, the **getpagesizes**() function returns either the number of page sizes that are supported by the system or the number of supported page sizes that it assigned to the array referenced by *pagesize*. Otherwise, it returns the value -1 and sets *errno* to indicate the error.

ERRORS

The **getpagesizes**() function will succeed unless:

[EINVAL] The *pagesize* argument is NULL and the *nelem* argument is non-zero.

[EINVAL] The *nelem* argument is less than zero.

SEE ALSO

getpagesize(3)

HISTORY

The **getpagesizes**() function first appeared in Solaris 9. This manual page was written in conjunction with a new but compatible implementation that was first released in FreeBSD 7.3.

AUTHORS

This manual page was written by Alan L. Cox <*alc@cs.rice.edu*>.

NAME

getpass - get a password

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <pwd.h>
#include <unistd.h>

*char **
getpass(*const char *prompt*);

DESCRIPTION

The **getpass**() function displays a prompt to, and reads in a password from, */dev/tty*. If this file is not accessible, **getpass**() displays the prompt on the standard error output and reads from the standard input.

The password may be up to _PASSWORD_LEN (currently 128) characters in length. Any additional characters and the terminating newline character are discarded.

The **getpass**() function turns off character echoing while reading the password.

RETURN VALUES

The **getpass**() function returns a pointer to the null terminated password.

FILES

/dev/tty

SEE ALSO

crypt(3), readpassphrase(3)

HISTORY

A **getpass**() function appeared in Version 7 AT&T UNIX.

BUGS

The **getpass**() function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **getpass**() will modify the same object.

The calling process should zero the password as soon as possible to avoid leaving the cleartext password visible in the process's address space.

Upon receipt of a SIGTSTP, the input buffer will be flushed, so any partially typed password must be retyped when the process continues.

NAME

getpeereid - get the effective credentials of a UNIX-domain peer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

int
getpeereid(*int s*, *uid_t *euid*, *gid_t *egid*);

DESCRIPTION

The **getpeereid**() function returns the effective user and group IDs of the peer connected to a UNIX-domain socket. The argument *s* must be a UNIX-domain socket (unix(4)) of type SOCK_STREAM on which either connect(2) or listen(2) has been called. The effective user ID is placed in *euid*, and the effective group ID in *egid*.

The credentials returned to the listen(2) caller are those of its peer at the time it called connect(2); the credentials returned to the connect(2) caller are those of its peer at the time it called listen(2). This mechanism is reliable; there is no way for either side to influence the credentials returned to its peer except by calling the appropriate system call (i.e., either connect(2) or listen(2)) under different effective credentials.

One common use of this routine is for a UNIX-domain server to verify the credentials of its client. Likewise, the client can verify the credentials of the server.

IMPLEMENTATION NOTES

On FreeBSD, **getpeereid**() is implemented in terms of the LOCAL_PEERCRED unix(4) socket option.

RETURN VALUES

The **getpeereid**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getpeereid**() function fails if:

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

[ENOTSOCK] The argument *s* is a file, not a socket.

[ENOTCONN] The argument *s* does not refer to a socket on which connect(2) or listen(2) have been called.

[EINVAL] The argument *s* does not refer to a socket of type SOCK_STREAM, or the kernel returned invalid data.

SEE ALSO

connect(2), getpeername(2), getsockname(2), getsockopt(2), listen(2), unix(4)

HISTORY

The **getpeereid**() function appeared in FreeBSD 4.6.

NAME

getpeername - get name of connected peer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

int
getpeername(*int s*, *struct sockaddr * restrict name*, *socklen_t * restrict namelen*);

DESCRIPTION

The **getpeername**() system call returns the name of the peer connected to socket *s*. The *namelen* argument should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

RETURN VALUES

The **getpeername**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ECONNRESET]	The connection has been reset by the peer.
[EINVAL]	The value of the *namelen* argument is not valid.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* argument points to memory not in a valid part of the process address space.

SEE ALSO

accept(2), bind(2), getsockname(2), socket(2)

HISTORY

The **getpeername**() system call appeared in 4.2BSD.

NAME

getpgrp - get process group

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
getpgrp(*void*);

pid_t
getpgid(*pid_t pid*);

DESCRIPTION

The process group of the current process is returned by **getpgrp**(). The process group of the process identified by *pid* is returned by **getpgid**(). If *pid* is zero, **getpgid**() returns the process group of the current process.

Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input: processes that have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read.

This system call is thus used by programs such as csh(1) to create process groups in implementing job control. The **tcgetpgrp**() and **tcsetpgrp**() calls are used to get/set the process group of the control terminal.

RETURN VALUES

The **getpgrp**() system call always succeeds. Upon successful completion, the **getpgid**() system call returns the process group of the specified process; otherwise, it returns a value of -1 and sets *errno* to indicate the error.

COMPATIBILITY

This version of **getpgrp**() differs from past Berkeley versions by not taking a *pid_t pid* argument. This incompatibility is required by IEEE Std 1003.1-1990 ("POSIX.1").

From the IEEE Std 1003.1-1990 ("POSIX.1") Rationale:

4.3BSD provides a **getpgrp**() system call that returns the process group ID for a specified process.

Although this function is used to support job control, all known job-control shells always specify the calling process with this function. Thus, the simpler AT&T System V UNIX **getpgrp**() suffices, and the added complexity of the 4.3BSD **getpgrp**() has been omitted from POSIX.1. The old functionality is available from the **getpgid**() system call.

ERRORS

The **getpgid**() system call will succeed unless:

[ESRCH] there is no process whose process ID equals *pid*

SEE ALSO

getsid(2), setpgid(2), termios(4)

STANDARDS

The **getpgrp**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **getpgrp**() system call appeared in 4.0BSD. The **getpgid**() system call is derived from its usage in AT&T System V Release 4 UNIX.

NAME

getpid, **getppid** - get parent or calling process identification

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
getpid(*void*);

pid_t
getppid(*void*);

DESCRIPTION

The **getpid**() system call returns the process ID of the calling process. Though the ID is guaranteed to be unique, it should *NOT* be used for constructing temporary file names, for security reasons; see mkstemp(3) instead.

The **getppid**() system call returns the process ID of the parent of the calling process.

ERRORS

The **getpid**() and **getppid**() system calls are always successful, and no return value is reserved to indicate an error.

SEE ALSO

fork(2), getpgrp(2), kill(2), setpgid(2), setsid(2), exec(3)

STANDARDS

The **getpid**() and **getppid**() system calls are expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **getpid**() function appeared in Version 7 AT&T UNIX.

NAME

getpriority, **setpriority** - get/set program scheduling priority

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/time.h>
#include <sys/resource.h>

int
getpriority(*int which*, *int who*);

int
setpriority(*int which*, *int who*, *int prio*);

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by *which* and *who* is obtained with the **getpriority**() system call and set with the **setpriority**() system call. The *which* argument is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and *who* is interpreted relative to *which* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER). A zero value of *who* denotes the current process, process group, or user. The *prio* argument is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favorable scheduling.

The **getpriority**() system call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The **setpriority**() system call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

RETURN VALUES

Since **getpriority**() can legitimately return the value -1, it is necessary to clear the external variable *errno* prior to the call, then check it afterward to determine if a -1 is an error or a legitimate value.

The **setpriority**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getpriority**() and **setpriority**() system calls will fail if:

[ESRCH] No process was located using the *which* and *who* values specified.

[EINVAL] The *which* argument was not one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER.

In addition to the errors indicated above, **setpriority**() will fail if:

[EPERM] A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.

[EACCES] A non super-user attempted to lower a process priority.

SEE ALSO

nice(1), fork(2), renice(8)

HISTORY

The **getpriority**() system call appeared in 4.2BSD.

NAME

getprogname, **setprogname** - get or set the program name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*const char **
getprogname(*void*);

void
setprogname(*const char *progname*);

DESCRIPTION

The **getprogname**() and **setprogname**() functions manipulate the name of the current program. They are used by error-reporting routines to produce consistent output.

The **getprogname**() function returns the name of the program. If the name has not been set yet, it will return NULL.

The **setprogname**() function sets the name of the program to be the last component of the *progname* argument. Since a pointer to the given string is kept as the program name, it should not be modified for the rest of the program's lifetime.

In FreeBSD, the name of the program is set by the start-up code that is run before **main**(); thus, running **setprogname**() is not necessary. Programs that desire maximum portability should still call it; on another operating system, these functions may be implemented in a portability library. Calling **setprogname**() allows the aforementioned library to learn the program name without modifications to the start-up code.

SEE ALSO

err(3), setproctitle(3)

HISTORY

These functions first appeared in NetBSD 1.6, and made their way into FreeBSD 4.4.

NAME

getrandom - get random data

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/random.h>

ssize_t
getrandom(*void *buf*, *size_t buflen*, *unsigned int flags*);

DESCRIPTION

getrandom() fills *buf* with up to *buflen* bytes of random data.

The *flags* argument may include zero or more of the following:

‘GRND_NONBLOCK’ Return EAGAIN instead of blocking, if the random(4) device has not yet been seeded. By default, **getrandom**() will block until the device is seeded.

‘GRND_RANDOM’ This flag does nothing on FreeBSD. */dev/random* and */dev/urandom* are identical.

If the random(4) device has been seeded, reads of up to 256 bytes will always return as many bytes as requested and will not be interrupted by signals.

RETURN VALUES

Upon successful completion, the number of bytes which were actually read is returned. For requests larger than 256 bytes, this can be fewer bytes than were requested. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getrandom**() operation returns the following errors:

[EAGAIN] The ‘GRND_NONBLOCK’ flag was set and the random(4) device was not yet seeded.

[EFAULT] The *buf* parameter points to an invalid address.

[EINTR] The sleep was interrupted by a signal.

[EINVAL] An invalid *flags* was specified.

[EINVAL] The requested *buflen* was larger than IOSIZE_MAX.

SEE ALSO

arc4random(3), getentropy(3), random(4)

STANDARDS

getentropy() is non-standard. It is present in Linux.

HISTORY

The **getrandom**() system call first appeared in FreeBSD 12.0.

NAME

getresgid, **getresuid**, **setresgid**, **setresuid** - get or set real, effective and saved user or group ID

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int
getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);

int
getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);

int
setresgid(gid_t rgid, gid_t egid, gid_t sgid);

int
setresuid(uid_t ruid, uid_t euid, uid_t suid);
```

DESCRIPTION

The **setresuid**() system call sets the real, effective and saved user IDs of the current process. The analogous **setresgid**() sets the real, effective and saved group IDs.

Privileged processes may set these IDs to arbitrary values. Unprivileged processes are restricted in that each of the new IDs must match one of the current IDs.

Passing -1 as an argument causes the corresponding value to remain unchanged.

The **getresgid**() and **getresuid**() calls retrieve the real, effective, and saved group and user IDs of the current process, respectively.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EPERM] The calling process was not privileged and tried to change one or more IDs to a

value which was not the current real ID, the current effective ID nor the current saved ID.

[EFAULT] An address passed to **getresgid**() or **getresuid**() was invalid.

SEE ALSO

getegid(2), geteuid(2), getgid(2), getuid(2), issetugid(2), setgid(2), setregid(2), setreuid(2), setuid(2)

HISTORY

These functions first appeared in HP-UX.

NAME

getrlimit, **setrlimit** - control maximum system resource consumption

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>

int
getrlimit(int resource, struct rlimit *rlp);

int
setrlimit(int resource, const struct rlimit *rlp);
```

DESCRIPTION

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the **getrlimit**() system call, and set with the **setrlimit**() system call.

The *resource* argument is one of the following:

RLIMIT_AS The maximum amount (in bytes) of virtual memory the process is allowed to map.

RLIMIT_CORE The largest size (in bytes) core(5) file that may be created.

RLIMIT_CPU The maximum amount of cpu time (in seconds) to be used by each process.

RLIMIT_DATA The maximum size (in bytes) of the data segment for a process; this defines how far a program may extend its break with the sbrk(2) function.

RLIMIT_FSIZE The largest size (in bytes) file that may be created.

RLIMIT_KQUEUES
The maximum number of kqueues this user id is allowed to create.

RLIMIT_MEMLOCK
The maximum size (in bytes) which a process may lock into memory using the mlock(2) system call.

RLIMIT_NOFILE The maximum number of open files for this process.

RLIMIT_NPROC The maximum number of simultaneous processes for this user id.

RLIMIT_NPTS The maximum number of pseudo-terminals this user id is allowed to create.

RLIMIT_RSS When there is memory pressure and swap is available, prioritize eviction of a process' resident pages beyond this amount (in bytes). When memory is not under pressure, this rlimit is effectively ignored. Even when there is memory pressure, the amount of available swap space and some sysctl settings like vm.swap_enabled and vm.swap_idle_enabled can affect what happens to processes that have exceeded this size.

Processes that exceed their set RLIMIT_RSS are not signalled or halted. The limit is merely a hint to the VM daemon to prefer to deactivate pages from processes that have exceeded their set RLIMIT_RSS.

RLIMIT_SBSIZE The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

RLIMIT_STACK The maximum size (in bytes) of the stack segment for a process; this defines how far a program's stack segment may be extended. Stack extension is performed automatically by the system.

RLIMIT_SWAP The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the *vm.overcommit* sysctl is set. Please see tuning(7) for a complete description of this sysctl.

RLIMIT_VMEM An alias for RLIMIT_AS.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded, a process might or might not receive a signal. For example, signals are generated when the cpu time or file size is exceeded, but not if the address space or RSS limit is exceeded. A program that exceeds the soft limit is allowed to continue execution until it reaches the hard limit, or modifies its own resource limit. Even reaching the hard limit does not necessarily halt a process. For example, if the RSS hard limit is exceeded, nothing happens.

The *rlimit* structure is used to specify the hard and soft limits on a resource.

```
struct rlimit {
        rlim_t  rlim_cur; /* current (soft) limit */
        rlim_t  rlim_max;       /* maximum value for rlim_cur */
};
```

Only the super-user may raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

An "infinite" value for a limit is defined as RLIM_INFINITY.

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; **limit** is thus a built-in command to csh(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a brk(2) function fails if the data space limit is reached. When the stack limit is reached, the process receives a segmentation fault (SIGSEGV); if this signal is not caught by a handler using the signal stack, this signal will kill the process.

A file I/O operation that would create a file larger that the process' soft limit will cause the write to fail and a signal SIGXFSZ to be generated; this normally terminates the process, but may be caught. When the soft cpu time limit is exceeded, a SIGXCPU signal is sent to the offending process.

When most operations would allocate more virtual memory than allowed by the soft limit of RLIMIT_AS, the operation fails with ENOMEM and no signal is raised. A notable exception is stack extension, described above. If stack extension would allocate more virtual memory than allowed by the soft limit of RLIMIT_AS, a SIGSEGV signal will be delivered. The caller is free to raise the soft address space limit up to the hard limit and retry the allocation.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getrlimit**() and **setrlimit**() system calls will fail if:

[EFAULT] The address specified for *rlp* is invalid.

[EPERM] The limit specified to **setrlimit**() would have raised the maximum limit value, and the caller is not the super-user.

SEE ALSO

csh(1), quota(1), quotactl(2), sigaction(2), sigaltstack(2), sysctl(3), ulimit(3)

HISTORY

The **getrlimit**() system call appeared in 4.2BSD.

NAME

getrpcport - get RPC port number

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

int
getrpcport(*char *host*, *int prognum*, *int versnum*, *int proto*);

DESCRIPTION

The **getrpcport**() function returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper, or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will still return a port number (for some version of the program) indicating that the program is indeed registered. The version mismatch will be detected upon the first call to the service.

NAME

getrusage - get information about resource utilization

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>

#define   RUSAGE_SELF     0
#define   RUSAGE_CHILDREN -1
#define   RUSAGE_THREAD   1

int
getrusage(int who, struct rusage *rusage);
```

DESCRIPTION

The **getrusage**() system call returns information describing the resources utilized by the current thread, the current process, or all its terminated child processes. The *who* argument is either RUSAGE_THREAD, RUSAGE_SELF, or RUSAGE_CHILDREN. The buffer to which *rusage* points will be filled in with the following structure:

```
struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long ru_maxrss;          /* max resident set size */
    long ru_ixrss;          /* integral shared text memory size */
    long ru_idrss;          /* integral unshared data size */
    long ru_isrss;          /* integral unshared stack size */
    long ru_minflt;          /* page reclaims */
    long ru_majflt;          /* page faults */
    long ru_nswap;           /* swaps */
    long ru_inblock;         /* block input operations */
    long ru_oublock;         /* block output operations */
    long ru_msgsnd;          /* messages sent */
    long ru_msgrcv;          /* messages received */
    long ru_nsignals;       /* signals received */
    long ru_nvcsw;           /* voluntary context switches */
```

```
        long ru_nivcsw;         /* involuntary context switches */
};
```

The fields are interpreted as follows:

ru_utime the total amount of time spent executing in user mode.

ru_stime the total amount of time spent in the system executing on behalf of the process(es).

ru_maxrss the maximum resident set size utilized (in kilobytes).

ru_ixrss an "integral" value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * ticks-of-execution. Ticks are statistics clock ticks. The statistics clock has a frequency of **sysconf**(*_SC_CLK_TCK*) ticks per second.

ru_idrss an integral value of the amount of unshared memory residing in the data segment of a process (expressed in units of kilobytes * ticks-of-execution).

ru_isrss an integral value of the amount of unshared memory residing in the stack segment of a process (expressed in units of kilobytes * ticks-of-execution).

ru_minflt the number of page faults serviced without any I/O activity; here I/O activity is avoided by "reclaiming" a page frame from the list of pages awaiting reallocation.

ru_majflt the number of page faults serviced that required I/O activity.

ru_nswap the number of times a process was "swapped" out of main memory.

ru_inblock the number of times the file system had to perform input.

ru_oublock the number of times the file system had to perform output.

ru_msgsnd the number of IPC messages sent.

ru_msgrcv the number of IPC messages received.

ru_nsignals the number of signals delivered.

ru_nvcsw the number of times a context switch resulted due to a process voluntarily giving up the

processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw the number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

NOTES

The numbers *ru_inblock* and *ru_oublock* account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data.

RETURN VALUES

The **getrusage**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **getrusage**() system call will fail if:

[EINVAL] The *who* argument is not a valid value.

[EFAULT] The address specified by the *rusage* argument is not in a valid part of the process address space.

SEE ALSO

gettimeofday(2), wait(2), clocks(7)

HISTORY

The **getrusage**() system call appeared in 4.2BSD. The RUSAGE_THREAD facility first appeared in FreeBSD 8.1.

BUGS

There is no way to obtain information about a child process that has not yet terminated.

NAME

getsid - get process session

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
getsid(*pid_t pid*);

DESCRIPTION

The session ID of the process identified by *pid* is returned by **getsid**(). If *pid* is zero, **getsid**() returns the session ID of the current process.

RETURN VALUES

Upon successful completion, the **getsid**() system call returns the session ID of the specified process; otherwise, it returns a value of -1 and sets errno to indicate an error.

ERRORS

The **getsid**() system call will succeed unless:

[ESRCH] if there is no process with a process ID equal to *pid*.

Note that an implementation may restrict this system call to processes within the same session ID as the calling process.

SEE ALSO

getpgid(2), getpgrp(2), setpgid(2), setsid(2), termios(4)

HISTORY

The **getsid**() system call appeared in FreeBSD 3.0. The **getsid**() system call is derived from its usage in AT&T System V UNIX.

NAME

getsockname - get socket name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

int
getsockname(*int s*, *struct sockaddr * restrict name*, *socklen_t * restrict namelen*);

DESCRIPTION

The **getsockname**() system call returns the current *name* for the specified socket. The *namelen* argument should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

RETURN VALUES

The **getsockname**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ECONNRESET]	The connection has been reset by the peer.
[EINVAL]	The value of the *namelen* argument is not valid.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* argument points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), getpeername(2), socket(2)

HISTORY

The **getsockname**() system call appeared in 4.2BSD.

BUGS

Names bound to sockets in the UNIX domain are inaccessible; **getsockname**() returns a zero length name.

NAME

getsockopt, **setsockopt** - get and set options on sockets

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

int
getsockopt(*int s*, *int level*, *int optname*, *void * restrict optval*, *socklen_t * restrict optlen*);

int
setsockopt(*int s*, *int level*, *int optname*, *const void *optval*, *socklen_t optlen*);

DESCRIPTION

The **getsockopt**() and **setsockopt**() system calls manipulate the *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see getprotoent(3).

The *optval* and *optlen* arguments are used to access option values for **setsockopt**(). For **getsockopt**() they identify a buffer in which the value for the requested option(s) are to be returned. For **getsockopt**(), *optlen* is a value-result argument, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be NULL.

The *optname* argument and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <*sys/socket.h*> contains definitions for socket level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section 4 of the manual.

Most socket-level options utilize an *int* argument for *optval*. For **setsockopt**(), the argument should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* argument, defined in <*sys/socket.h*>, which specifies the desired state of the option and the linger

interval (see below). SO_SNDTIMEO and SO_RCVTIMEO use a *struct timeval* argument, defined in <*sys/time.h*>.

The following options are recognized at the socket level. For protocol-specific options, see protocol manual pages, e.g. ip(4) or tcp(4). Except as noted, each may be examined with **getsockopt**() and set with **setsockopt**().

SO_DEBUG	enables recording of debugging information
SO_REUSEADDR	enables local address reuse
SO_REUSEPORT	enables duplicate address and port bindings
SO_REUSEPORT_LB	enables duplicate address and port bindings with load balancing
SO_KEEPALIVE	enables keep connections alive
SO_DONTROUTE	enables routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	enables permission to transmit broadcast messages
SO_OOBINLINE	enables reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_SNDLOWAT	set minimum count for output
SO_RCVLOWAT	set minimum count for input
SO_SNDTIMEO	set timeout value for output
SO_RCVTIMEO	set timeout value for input
SO_ACCEPTFILTER	set accept filter on listening socket
SO_NOSIGPIPE	controls generation of SIGPIPE for the socket
SO_TIMESTAMP	enables reception of a timestamp with datagrams
SO_BINTIME	enables reception of a timestamp with datagrams
SO_ACCEPTCONN	get listening status of the socket (get only)
SO_DOMAIN	get the domain of the socket (get only)
SO_TYPE	get the type of the socket (get only)
SO_PROTOCOL	get the protocol number for the socket (get only)
SO_PROTOTYPE	SunOS alias for the Linux SO_PROTOCOL (get only)
SO_ERROR	get and clear error on the socket (get only)
SO_SETFIB	set the associated FIB (routing table) for the socket (set only)

The following options are recognized in FreeBSD:

SO_LABEL	get MAC label of the socket (get only)
SO_PEERLABEL	get socket's peer's MAC label (get only)
SO_LISTENQLIMIT	get backlog limit of the socket (get only)
SO_LISTENQLEN	get complete queue length of the socket (get only)

SO_LISTENINCQLEN get incomplete queue length of the socket (get only)
SO_USER_COOKIE set the 'so_user_cookie' value for the socket (uint32_t, set only)
SO_TS_CLOCK set specific format of timestamp returned by SO_TIMESTAMP
SO_MAX_PACING_RATE
set the maximum transmit rate in bytes per second for the socket

SO_DEBUG enables debugging in the underlying protocol modules.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind(2) system call should allow reuse of local addresses.

SO_REUSEPORT allows completely duplicate bindings by multiple processes if they all set SO_REUSEPORT before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port.

SO_REUSEPORT_LB allows completely duplicate bindings by multiple processes if they all set SO_REUSEPORT_LB before binding the port. Incoming TCP and UDP connections are distributed among the sharing processes based on a hash function of local port number, foreign IP address and port number. A maximum of 256 processes can share one socket.

SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal when attempting to send data.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a close(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close(2) attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in seconds in the **setsockopt**() system call when SO_LINGER is requested). If SO_LINGER is disabled and a close(2) is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system.

With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data

be placed in the normal data input queue as received; it will then be accessible with recv(2) or read(2) calls without the MSG_OOB flag. Some protocols always behave as if this option is set.

SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute maximum on these values, which is accessible through the sysctl(3) MIB variable "kern.ipc.maxsockbuf".

SO_SNDLOWAT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A select(2) operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024.

SO_RCVLOWAT is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that which was returned.

SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a *struct timeval* argument with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error EWOULDBLOCK if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output.

SO_RCVTIMEO is an option to set a timeout value for input operations. It accepts a *struct timeval* argument with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error EWOULDBLOCK if no data were received.

SO_SETFIB can be used to over-ride the default FIB (routing table) for the given socket. The value must be from 0 to one less than the number returned from the sysctl *net.fibs*.

SO_USER_COOKIE can be used to set the uint32_t so_user_cookie field in the socket. The value is an uint32_t, and can be used in the kernel code that manipulates traffic related to the socket. The default value for the field is 0. As an example, the value can be used as the skipto target or pipe number in **ipfw/dummynet**.

SO_ACCEPTFILTER places an accept_filter(9) on the socket, which will filter incoming connections on a listening stream socket before being presented for accept(2). Once more, listen(2) must be called on the socket before trying to install the filter on it, or else the **setsockopt**() system call will fail.

```
struct  accept_filter_arg {
        char    af_name[16];
        char    af_arg[256-16];
};
```

The *optval* argument should point to a *struct accept_filter_arg* that will select and configure the accept_filter(9). The *af_name* argument should be filled with the name of the accept filter that the application wishes to place on the listening socket. The optional argument *af_arg* can be passed to the accept filter specified by *af_name* to provide additional configuration options at attach time. Passing in an *optval* of NULL will remove the filter.

The SO_NOSIGPIPE option controls generation of the SIGPIPE signal normally sent when writing to a connected socket where the other end has been closed returns with the error EPIPE.

If the SO_TIMESTAMP or SO_BINTIME option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call will return a timestamp corresponding to when the datagram was received. The *msg_control* field in the *msghdr* structure points to a buffer that contains a *cmsghdr* structure followed by a *struct timeval* for SO_TIMESTAMP and *struct bintime* for SO_BINTIME. The *cmsghdr* fields have the following values for TIMESTAMP by default:

```
cmsg_len = CMSG_LEN(sizeof(struct timeval));
cmsg_level = SOL_SOCKET;
cmsg_type = SCM_TIMESTAMP;
```

and for SO_BINTIME:

```
cmsg_len = CMSG_LEN(sizeof(struct bintime));
cmsg_level = SOL_SOCKET;
cmsg_type = SCM_BINTIME;
```

Additional timestamp types are available by following SO_TIMESTAMP with SO_TS_CLOCK, which

requests a specific timestamp format to be returned instead of SCM_TIMESTAMP when SO_TIMESTAMP is enabled. These SO_TS_CLOCK values are recognized in FreeBSD:

SO_TS_REALTIME_MICRO
realtime (SCM_TIMESTAMP, struct timeval), default
SO_TS_BINTIME
realtime (SCM_BINTIME, struct bintime)
SO_TS_REALTIME
realtime (SCM_REALTIME, struct timespec)
SO_TS_MONOTONIC
monotonic time (SCM_MONOTONIC, struct timespec)

SO_ACCEPTCONN, SO_TYPE, SO_PROTOCOL (and its alias SO_PROTOTYPE) and SO_ERROR are options used only with **getsockopt**(). SO_ACCEPTCONN returns whether the socket is currently accepting connections, that is, whether or not the listen(2) system call was invoked on the socket. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_PROTOCOL returns the protocol number for the socket, for AF_INET and AF_INET6 address families. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

Finally, SO_LABEL returns the MAC label of the socket. SO_PEERLABEL returns the MAC label of the socket's peer. Note that your kernel must be compiled with MAC support. See mac(3) for more information. SO_LISTENQLIMIT returns the maximal number of queued connections, as set by listen(2). SO_LISTENQLEN returns the number of unaccepted complete connections. SO_LISTENINCQLEN returns the number of unaccepted incomplete connections.

SO_MAX_PACING_RATE instruct the socket and underlying network adapter layers to limit the transfer rate to the given unsigned 32-bit value in bytes per second.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The call succeeds unless:

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

[ENOTSOCK] The argument *s* is a file, not a socket.

[ENOPROTOOPT] The option is unknown at the level indicated.

[EFAULT] The address pointed to by *optval* is not in a valid part of the process address space. For **getsockopt**(), this error may also be returned if *optlen* is not in a valid part of the process address space.

[EINVAL] Installing an accept_filter(9) on a non-listening socket was attempted.

[ENOMEM] A memory allocation failed that was required to service the request.

SEE ALSO

ioctl(2), listen(2), recvmsg(2), socket(2), getprotoent(3), mac(3), sysctl(3), ip(4), ip6(4), sctp(4), tcp(4), protocols(5), sysctl(8), accept_filter(9), bintime(9)

HISTORY

The **getsockopt**() and **setsockopt**() system calls appeared in 4.2BSD.

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

getsubopt - get sub options from an argument

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*extern char *suboptarg*;

int
getsubopt(*char **optionp*, *char * const *tokens*, *char **valuep*);

DESCRIPTION

The **getsubopt**() function parses a string containing tokens delimited by one or more tab, space or comma (‘,’) characters. It is intended for use in parsing groups of option arguments provided as part of a utility command line.

The argument *optionp* is a pointer to a pointer to the string. The argument *tokens* is a pointer to a NULL-terminated array of pointers to strings.

The **getsubopt**() function returns the zero-based offset of the pointer in the *tokens* array referencing a string which matches the first token in the string, or, -1 if the string contains no tokens or *tokens* does not contain a matching string.

If the token is of the form ‘‘name=value’’, the location referenced by *valuep* will be set to point to the start of the ‘‘value’’ portion of the token.

On return from **getsubopt**(), *optionp* will be set to point to the start of the next token in the string, or the null at the end of the string if no more tokens are present. The external variable *suboptarg* will be set to point to the start of the current token, or NULL if no tokens were present. The argument *valuep* will be set to point to the ‘‘value’’ portion of the token, or NULL if no ‘‘value’’ portion was present.

EXAMPLES

```
char *tokens[] = {
        #define   ONE     0
                  "one",
        #define   TWO     1
                  "two",
```

```
        NULL
};

...

extern char *optarg, *suboptarg;
char *options, *value;

while ((ch = getopt(argc, argv, "ab:")) != -1) {
        switch(ch) {
        case 'a':
                /* process ``a'' option */
                break;
        case 'b':
                options = optarg;
                while (*options) {
                        switch(getsubopt(&options, tokens, &value)) {
                        case ONE:
                                /* process ``one'' sub option */
                                break;
                        case TWO:
                                /* process ``two'' sub option */
                                if (!value)
                                        error("no value for two");
                                i = atoi(value);
                                break;
                        case -1:
                                if (suboptarg)
                                        error("illegal sub option %s",
                                          suboptarg);
                                else
                                        error("missing sub option");
                                break;
                        }
                }
                break;
        }
}
```

SEE ALSO

getopt(3), strsep(3)

HISTORY

The **getsubopt**() function first appeared in 4.4BSD.

NAME

gettimeofday, **settimeofday** - get/set date and time

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/time.h>

int
gettimeofday(*struct timeval *tp*, *struct timezone *tzp*);

int
settimeofday(*const struct timeval *tp*, *const struct timezone *tzp*);

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the **gettimeofday**() system call, and set with the **settimeofday**() system call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in "ticks". If *tp* or *tzp* is NULL, the associated time information will not be returned or set.

The structures pointed to by *tp* and *tzp* are defined in <*sys/time.h*> as:

```
struct timeval {
        time_t          tv_sec;         /* seconds */
        suseconds_t     tv_usec;  /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* minutes west of Greenwich */
        int     tz_dsttime;     /* type of dst correction */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Only the super-user may set the time of day or time zone. If the system is running at securelevel >= 2 (see init(8)), the time may only be advanced or retarded by a maximum of one second. This limitation is

imposed to prevent a malicious super-user from setting arbitrary time stamps on files. The system time can be adjusted backwards without restriction using the adjtime(2) system call even when the system is secure.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following error codes may be set in *errno*:

[EINVAL] The supplied *timeval* value is invalid.

[EPERM] A user other than the super-user attempted to set the time.

SEE ALSO

date(1), adjtime(2), clock_gettime(2), ctime(3), timeradd(3), clocks(7), timed(8)

HISTORY

The **gettimeofday**() system call appeared in 4.2BSD.

NAME

getvfsbyname - get information about a file system

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/mount.h>

int
getvfsbyname(*const char *name*, *struct xvfsconf *vfc*);

DESCRIPTION

The **getvfsbyname**() function provides access to information about a file system module that is configured in the kernel. If successful, the requested file system *xvfsconf* is returned in the location pointed to by *vfc*. The fields in a "struct xvfsconf" are defined as follows:

vfc_name	the name of the file system
vfc_typenum	the file system type number assigned by the kernel
vfc_refcount	the number of active mount points using the file system
vfc_flags	flag bits, as described below

The flags are defined as follows:

VFCF_STATIC	statically compiled into kernel
VFCF_NETWORK	may get data over the network
VFCF_READONLY	writes are not implemented
VFCF_SYNTHETIC	data does not represent real files
VFCF_LOOPBACK	aliases some other mounted FS
VFCF_UNICODE	stores file names as Unicode
VFCF_JAIL	can be mounted from within a jail if *allow.mount* and *allow.mount.<vfc_name>* jail parameters are set
VFCF_DELEGADMIN	supports delegated administration if *vfs.usermount* sysctl is set to 1

RETURN VALUES

The **getvfsbyname**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be reported:

[ENOENT] The *name* argument specifies a file system that is unknown or not configured in the kernel.

SEE ALSO

jail(2), mount(2), sysctl(3), jail(8), mount(8), sysctl(8)

HISTORY

A variant of the **getvfsbyname**() function first appeared in FreeBSD 2.0.

NAME

glob, **globfree** - generate pathnames matching a pattern

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <glob.h>

int
glob(*const char * restrict pattern*, *int flags*, *int (*errfunc)(const char *, int)*, *glob_t * restrict pglob*);

void
globfree(*glob_t *pglob*);

DESCRIPTION

The **glob**() function is a pathname generator that implements the rules for file name pattern matching used by the shell.

The include file <*glob.h*> defines the structure type *glob_t*, which contains at least the following fields:

```
typedef struct {
        size_t gl_pathc;    /* count of total paths so far */
        size_t gl_matchc;   /* count of paths matching pattern */
        size_t gl_offs;             /* reserved at beginning of gl_pathv */
        int gl_flags;               /* returned flags */
        char **gl_pathv;    /* list of paths matching pattern */
} glob_t;
```

The argument *pattern* is a pointer to a pathname pattern to be expanded. The **glob**() argument matches all accessible pathnames against the pattern and creates a list of the pathnames that match. In order to have access to a pathname, **glob**() requires search permission on every component of a path except the last and read permission on each directory of any filename component of *pattern* that contains any of the special characters '*', '?' or '['.

The **glob**() argument stores the number of matched pathnames into the *gl_pathc* field, and a pointer to a list of pointers to pathnames into the *gl_pathv* field. The first pointer after the last pathname is NULL. If the pattern does not match any pathnames, the returned number of matched paths is set to zero.

It is the caller's responsibility to create the structure pointed to by *pglob*. The **glob**() function allocates

other space as needed, including the memory pointed to by *gl_pathv*.

The argument *flags* is used to modify the behavior of **glob**(). The value of *flags* is the bitwise inclusive OR of any of the following values defined in <*glob.h*>:

GLOB_APPEND Append pathnames generated to the ones from a previous call (or calls) to **glob**(). The value of *gl_pathc* will be the total matches found by this call and the previous call(s). The pathnames are appended to, not merged with the pathnames returned by the previous call(s). Between calls, the caller must not change the setting of the GLOB_DOOFFS flag, nor change the value of *gl_offs* when GLOB_DOOFFS is set, nor (obviously) call **globfree**() for *pglob*.

GLOB_DOOFFS Make use of the *gl_offs* field. If this flag is set, *gl_offs* is used to specify how many NULL pointers to prepend to the beginning of the *gl_pathv* field. In other words, *gl_pathv* will point to *gl_offs* NULL pointers, followed by *gl_pathc* pathname pointers, followed by a NULL pointer.

GLOB_ERR Causes **glob**() to return when it encounters a directory that it cannot open or read. Ordinarily, **glob**() continues to find matches.

GLOB_MARK Each pathname that is a directory that matches *pattern* has a slash appended.

GLOB_NOCHECK If *pattern* does not match any pathname, then **glob**() returns a list consisting of only *pattern*, with the number of total pathnames set to 1, and the number of matched pathnames set to 0. The effect of backslash escaping is present in the pattern returned.

GLOB_NOESCAPE By default, a backslash (‘\’) character is used to escape the following character in the pattern, avoiding any special interpretation of the character. If GLOB_NOESCAPE is set, backslash escaping is disabled.

GLOB_NOSORT By default, the pathnames are sorted in ascending collation order; this flag prevents that sorting (speeding up **glob**()).

The following values may also be included in *flags*, however, they are non-standard extensions to IEEE Std 1003.2 ("POSIX.2").

GLOB_ALTDIRFUNC The following additional fields in the pglob structure have been initialized with alternate functions for glob to use to open, read, and close directories and to get stat information on names found in those directories.

```
void *(*gl_opendir)(const char * name);
struct dirent *(*gl_readdir)(void *);
void (*gl_closedir)(void *);
int (*gl_lstat)(const char *name, struct stat *st);
int (*gl_stat)(const char *name, struct stat *st);
```

This extension is provided to allow programs such as restore(8) to provide globbing from directories stored on tape.

GLOB_BRACE Pre-process the pattern string to expand ‘{pat,pat,...}’ strings like csh(1). The pattern ‘{}’ is left unexpanded for historical reasons (and csh(1) does the same thing to ease typing of find(1) patterns).

GLOB_MAGCHAR Set by the **glob**() function if the pattern included globbing characters. See the description of the usage of the *gl_matchc* structure member for more details.

GLOB_NOMAGIC Is the same as GLOB_NOCHECK but it only appends the *pattern* if it does not contain any of the special characters ‘‘*’’, ‘‘?’’ or ‘‘[’’. GLOB_NOMAGIC is provided to simplify implementing the historic csh(1) globbing behavior and should probably not be used anywhere else.

GLOB_TILDE Expand patterns that start with ‘~’ to user name home directories.

GLOB_LIMIT Limit the total number of returned pathnames to the value provided in *gl_matchc* (default ARG_MAX). This option should be set for programs that can be coerced into a denial of service attack via patterns that expand to a very large number of matches, such as a long string of ‘*/../*/..’.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is non-NULL, **glob**() calls *(*errfunc)*(*path*, *errno*), however, the GLOB_ERR flag will cause an immediate return when this happens.

If *errfunc* returns non-zero, **glob**() stops the scan and returns GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* to reflect any paths already matched. This also happens if an error is encountered and GLOB_ERR is set in *flags*, regardless of the return value of *errfunc*, if called. If GLOB_ERR is not set and either *errfunc* is NULL or *errfunc* returns zero, the error is ignored.

The **globfree**() function frees any space associated with *pglob* from a previous call(s) to **glob**().

RETURN VALUES

On successful completion, **glob**() returns zero. In addition the fields of *pglob* contain the values described below:

gl_pathc contains the total number of matched pathnames so far. This includes other matches from previous invocations of **glob**() if GLOB_APPEND was specified.

gl_matchc contains the number of matched pathnames in the current invocation of **glob**().

gl_flags contains a copy of the *flags* argument with the bit GLOB_MAGCHAR set if *pattern* contained any of the special characters ‘‘*’’, ‘‘?’’ or ‘‘[’’, cleared if not.

gl_pathv contains a pointer to a NULL-terminated list of matched pathnames. However, if *gl_pathc* is zero, the contents of *gl_pathv* are undefined.

If **glob**() terminates due to an error, it sets errno and returns one of the following non-zero constants, which are defined in the include file <*glob.h*>:

GLOB_NOSPACE An attempt to allocate memory failed, or if *errno* was E2BIG, GLOB_LIMIT was specified in the flags and *pglob->gl_matchc* or more patterns were matched.

GLOB_ABORTED The scan was stopped because an error was encountered and either GLOB_ERR was set or *(*errfunc)()* returned non-zero.

GLOB_NOMATCH
The pattern did not match a pathname and GLOB_NOCHECK was not set.

The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as specified above.

EXAMPLES

A rough equivalent of ‘ls -l *.c *.h’ can be obtained with the following code:

```
glob_t g;

g.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &g);
glob("*.h", GLOB_DOOFFS | GLOB_APPEND, NULL, &g);
g.gl_pathv[0] = "ls";
g.gl_pathv[1] = "-l";
execvp("ls", g.gl_pathv);
```

SEE ALSO

sh(1), fnmatch(3), regex(3)

STANDARDS

The current implementation of the **glob**() function *does not* conform to IEEE Std 1003.2 ("POSIX.2"). Collating symbol expressions, equivalence class expressions and character class expressions are not supported.

The flags GLOB_ALTDIRFUNC, GLOB_BRACE, GLOB_LIMIT, GLOB_MAGCHAR, GLOB_NOMAGIC, and GLOB_TILDE, and the fields *gl_matchc* and *gl_flags* are extensions to the POSIX standard and should not be used by applications striving for strict conformance.

HISTORY

The **glob**() and **globfree**() functions first appeared in 4.4BSD.

BUGS

Patterns longer than MAXPATHLEN may cause unchecked errors.

The **glob**() argument may fail and set errno for any of the errors specified for the library routines stat(2), closedir(3), opendir(3), readdir(3), malloc(3), and free(3).

NAME

grantpt, **ptsname**, **unlockpt** - pseudo-terminal access functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
grantpt(*int fildes*);

*char **
ptsname(*int fildes*);

int
unlockpt(*int fildes*);

DESCRIPTION

The **grantpt**(), **ptsname**(), and **unlockpt**() functions allow access to pseudo-terminal devices. These three functions accept a file descriptor that references the master half of a pseudo-terminal pair. This file descriptor is created with posix_openpt(2).

The **grantpt**() function is used to establish ownership and permissions of the slave device counterpart to the master device specified with *fildes*. The slave device's ownership is set to the real user ID of the calling process, and the permissions are set to user readable-writable and group writable. The group owner of the slave device is also set to the group "tty".

The **ptsname**() function returns the full pathname of the slave device counterpart to the master device specified with *fildes*. This value can be used to subsequently open the appropriate slave after posix_openpt(2) and **grantpt**() have been called.

The **unlockpt**() function clears the lock held on the pseudo-terminal pair for the master device specified with *fildes*.

RETURN VALUES

The **grantpt**() and **unlockpt**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

The **ptsname**() function returns a pointer to the name of the slave device on success; otherwise a NULL

pointer is returned.

ERRORS

The **grantpt**(), **ptsname**() and **unlockpt**() functions may fail and set *errno* to:

[EBADF] *fildes* is not a valid open file descriptor.

[EINVAL] *fildes* is not a master pseudo-terminal device.

In addition, the **grantpt**() function may set *errno* to:

[EACCES] The slave pseudo-terminal device could not be accessed.

SEE ALSO

posix_openpt(2), pts(4), tty(4)

STANDARDS

The **ptsname**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

This implementation of **grantpt**() and **unlockpt**() does not conform to IEEE Std 1003.1-2008 ("POSIX.1"), because it depends on posix_openpt(2) to create the pseudo-terminal device with proper permissions in place. It only validates whether *fildes* is a valid pseudo-terminal master device. Future revisions of the specification will likely allow this behaviour, as stated by the Austin Group.

HISTORY

The **grantpt**(), **ptsname**() and **unlockpt**() functions appeared in FreeBSD 5.0.

NAME

pwcache, **user_from_uid**, **group_from_gid** - cache password and group entries

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <pwd.h>

*const char **
user_from_uid(*uid_t uid*, *int nouser*);

int
uid_from_user(*const char *name*, *uid_t *uid*);

int
pwcache_userdb(*int (*setpassent)(int)*, *void (*endpwent)(void)*,
*struct passwd * (*getpwnam)(const char *)*, *struct passwd * (*getpwuid)(uid_t)*);

#include <grp.h>

*const char **
group_from_gid(*gid_t gid*, *int nogroup*);

int
gid_from_group(*const char *name*, *gid_t *gid*);

int
pwcache_groupdb(*int (*setgroupent)(int)*, *void (*endgrent)(void)*,
*struct group * (*getgrnam)(const char *)*, *struct group * (*getgrgid)(gid_t)*);

DESCRIPTION

The **user_from_uid**() function returns the user name associated with the argument *uid*. The user name is cached so that multiple calls with the same *uid* do not require additional calls to getpwuid(3). If there is no user associated with the *uid*, a pointer is returned to a string representation of the *uid*, unless the argument *nouser* is non-zero, in which case a NULL pointer is returned.

The **group_from_gid**() function returns the group name associated with the argument *gid*. The group name is cached so that multiple calls with the same *gid* do not require additional calls to getgrgid(3). If there is no group associated with the *gid*, a pointer is returned to a string representation of the *gid*, unless

the argument *nogroup* is non-zero, in which case a NULL pointer is returned.

The **uid_from_user**() function returns the uid associated with the argument *name*. The uid is cached so that multiple calls with the same *name* do not require additional calls to getpwnam(3). If there is no uid associated with the *name*, the **uid_from_user**() function returns -1; otherwise it stores the uid at the location pointed to by *uid* and returns 0.

The **gid_from_group**() function returns the gid associated with the argument *name*. The gid is cached so that multiple calls with the same *name* do not require additional calls to getgrnam(3). If there is no gid associated with the *name*, the **gid_from_group**() function returns -1; otherwise it stores the gid at the location pointed to by *gid* and returns 0.

The **pwcache_userdb**() function changes the user database access routines which **user_from_uid**() and **uid_from_user**() call to search for users. The caches are flushed and the existing **endpwent**() method is called before switching to the new routines. *getpwnam* and *getpwuid* must be provided, and *setpassent* and *endpwent* may be NULL pointers.

The **pwcache_groupdb**() function changes the group database access routines which **group_from_gid**() and **gid_from_group**() call to search for groups. The caches are flushed and the existing **endgrent**() method is called before switching to the new routines. *getgrnam* and *getgrgid* must be provided, and *setgroupent* and *endgrent* may be NULL pointers.

SEE ALSO

getgrgid(3), getgrnam(3), getpwnam(3), getpwuid(3)

HISTORY

The **user_from_uid**() and **group_from_gid**() functions first appeared in 4.4BSD.

The **uid_from_user**() and **gid_from_group**() functions first appeared in NetBSD 1.4.

The **pwcache_userdb**() and **pwcache_groupdb**() functions first appeared in NetBSD 1.6 and FreeBSD 10.0.

NAME

hcreate, **hcreate_r**, **hdestroy**, **hdestroy_r**, **hsearch**, **hsearch_r** - manage hash search table

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <search.h>

int
hcreate(*size_t nel*);

int
hcreate_r(*size_t nel*, *struct hsearch_data *table*);

void
hdestroy(*void*);

void
hdestroy_r(*struct hsearch_data *table*);

*ENTRY **
hsearch(*ENTRY item*, *ACTION action*);

int
hsearch_r(*ENTRY item*, *ACTION action*, *ENTRY ** itemp*, *struct hsearch_data *table*);

DESCRIPTION

The **hcreate**(), **hcreate_r**(), **hdestroy**(), **hdestroy_r**() **hsearch**(), and **hsearch_r**() functions manage hash search tables.

The **hcreate**() function allocates sufficient space for the table, and the application should ensure it is called before **hsearch**() is used. The *nel* argument is an estimate of the maximum number of entries that the table should contain. As this implementation resizes the hash table dynamically, this argument is ignored.

The **hdestroy**() function disposes of the search table, and may be followed by another call to **hcreate**(). After the call to **hdestroy**(), the data can no longer be considered accessible. The **hdestroy**() function calls free(3) for each comparison key in the search table but not the data item associated with the key.

The **hsearch**() function is a hash-table search routine. It returns a pointer into a hash table indicating the location at which an entry can be found. The *item* argument is a structure of type *ENTRY* (defined in the <*search.h*> header) that contains two pointers: *item.key* points to the comparison key (a *char* *), and *item.data* (a *void* *) points to any other data to be associated with that key. The comparison function used by **hsearch**() is strcmp(3). The *action* argument is a member of an enumeration type *ACTION* indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the *item* should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

The comparison key (passed to **hsearch**() as *item.key*) must be allocated using malloc(3) if *action* is ENTER and **hdestroy**() is called.

The **hcreate_r**(), **hdestroy_r**(), and **hsearch_r**() functions are re-entrant versions of the above functions that can operate on a table supplied by the user. The **hsearch_r**() function returns 0 if the action is ENTER and the element cannot be created, 1 otherwise. If the element exists or can be created, it will be placed in *itemp*, otherwise *itemp* will be set to NULL.

RETURN VALUES

The **hcreate**() and **hcreate_r**() functions return 0 if the table creation failed and the global variable *errno* is set to indicate the error; otherwise, a non-zero value is returned.

The **hdestroy**() and **hdestroy_r**() functions return no value.

The **hsearch**() and **hsearch_r**() functions return a NULL pointer if either the *action* is FIND and the *item* could not be found or the *action* is ENTER and the table is full.

EXAMPLES

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```
#include <stdio.h>
#include <search.h>
#include <string.h>
#include <stdlib.h>

struct info {                   /* This is the info stored in the table */
        int age, room;          /* other than the key. */
};
```

```
#define NUM_EMPL        5000    /* # of elements in search table. */

int
main(void)
{
        char str[BUFSIZ]; /* Space to read string */
        struct info info_space[NUM_EMPL]; /* Space to store employee info. */
        struct info *info_ptr = info_space; /* Next space in info_space. */
        ENTRY item;
        ENTRY *found_item; /* Name to look for in table. */
        char name_to_find[30];
        int i = 0;

        /* Create table; no error checking is performed. */
        (void) hcreate(NUM_EMPL);

        while (scanf("%s%d%d", str, &info_ptr->age,
           &info_ptr->room) != EOF && i++ < NUM_EMPL) {
                /* Put information in structure, and structure in item. */
                item.key = strdup(str);
                item.data = info_ptr;
                info_ptr++;
                /* Put item into table. */
                (void) hsearch(item, ENTER);
        }

        /* Access table. */
        item.key = name_to_find;
        while (scanf("%s", item.key) != EOF) {
                if ((found_item = hsearch(item, FIND)) != NULL) {
                        /* If item is in the table. */
                        (void)printf("found %s, age = %d, room = %d\n",
                            found_item->key,
                            ((struct info *)found_item->data)->age,
                            ((struct info *)found_item->data)->room);
                } else
                        (void)printf("no such employee %s\n", name_to_find);
        }
        hdestroy();
        return 0;
```

```
}
```

ERRORS

The **hcreate**(), **hcreate_r**(), **hsearch**(), and **hsearch_r**() functions will fail if:

[ENOMEM] Insufficient memory is available.

The **hsearch**() and **hsearch_r**() functions will also fail if the action is FIND and the element is not found:

[ESRCH] The *item* given is not found.

SEE ALSO

bsearch(3), lsearch(3), malloc(3), strcmp(3), tsearch(3)

STANDARDS

The **hcreate**(), **hdestroy**(), and **hsearch**() functions conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

The **hcreate**(), **hdestroy**(), and **hsearch**() functions first appeared in AT&T System V UNIX. The **hcreate_r**(), **hdestroy_r**() and **hsearch_r**() functions are GNU extensions.

BUGS

The original, non-GNU interface permits the use of only one hash table at a time.

NAME

qsort, **qsort_b**, **qsort_r**, **heapsort**, **heapsort_b**, **mergesort**, **mergesort_b** - sort functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

void
qsort(*void *base*, *size_t nmemb*, *size_t size*, *int (*compar)(const void *, const void *)*);

void
qsort_b(*void *base*, *size_t nmemb*, *size_t size*, *int (^compar)(const void *, const void *)*);

void
qsort_r(*void *base*, *size_t nmemb*, *size_t size*, *void *thunk*,
*int (*compar)(void *, const void *, const void *)*);

int
heapsort(*void *base*, *size_t nmemb*, *size_t size*, *int (*compar)(const void *, const void *)*);

int
heapsort_b(*void *base*, *size_t nmemb*, *size_t size*, *int (^compar)(const void *, const void *)*);

int
mergesort(*void *base*, *size_t nmemb*, *size_t size*, *int (*compar)(const void *, const void *)*);

int
mergesort_b(*void *base*, *size_t nmemb*, *size_t size*, *int (^compar)(const void *, const void *)*);

DESCRIPTION

The **qsort**() function is a modified partition-exchange sort, or quicksort. The **heapsort**() function is a modified selection sort. The **mergesort**() function is a modified merge sort with exponential search intended for sorting data with pre-existing order.

The **qsort**() and **heapsort**() functions sort an array of *nmemb* objects, the initial member of which is pointed to by *base*. The size of each object is specified by *size*. The **mergesort**() function behaves similarly, but *requires* that *size* be greater than "sizeof(void *) / 2".

The contents of the array *base* are sorted in ascending order according to a comparison function pointed to by *compar*, which requires two arguments pointing to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

The **qsort_r**() function behaves identically to **qsort**(), except that it takes an additional argument, *thunk*, which is passed unchanged as the first argument to function pointed to *compar*. This allows the comparison function to access additional data without using global variables, and thus **qsort_r**() is suitable for use in functions which must be reentrant. The **qsort_b**() function behaves identically to **qsort**(), except that it takes a block, rather than a function pointer.

The algorithms implemented by **qsort**(), **qsort_r**(), and **heapsort**() are *not* stable, that is, if two members compare as equal, their order in the sorted array is undefined. The **heapsort_b**() function behaves identically to **heapsort**(), except that it takes a block, rather than a function pointer. The **mergesort**() algorithm is stable. The **mergesort_b**() function behaves identically to **mergesort**(), except that it takes a block, rather than a function pointer.

The **qsort**() and **qsort_r**() functions are an implementation of C.A.R. Hoare's "quicksort" algorithm, a variant of partition-exchange sorting; in particular, see D.E. Knuth's *Algorithm Q*. **Quicksort** takes O N lg N average time. This implementation uses median selection to avoid its O N**2 worst-case behavior.

The **heapsort**() function is an implementation of J.W.J. William's "heapsort" algorithm, a variant of selection sorting; in particular, see D.E. Knuth's *Algorithm H*. **Heapsort** takes O N lg N worst-case time. Its *only* advantage over **qsort**() is that it uses almost no additional memory; while **qsort**() does not allocate memory, it is implemented using recursion.

The function **mergesort**() requires additional memory of size *nmemb * size* bytes; it should be used only when space is not at a premium. The **mergesort**() function is optimized for data with pre-existing order; its worst case time is O N lg N; its best case is O N.

Normally, **qsort**() is faster than **mergesort**() is faster than **heapsort**(). Memory availability and pre-existing order in the data can make this untrue.

RETURN VALUES

The **qsort**() and **qsort_r**() functions return no value.

The **heapsort**() and **mergesort**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

EXAMPLES

A sample program that sorts an array of *int* values in place using **qsort**(), and then prints the sorted array to standard output is:

```
#include <stdio.h>
#include <stdlib.h>

/*
 * Custom comparison function that compares 'int' values through pointers
 * passed by qsort(3).
 */
static int
int_compare(const void *p1, const void *p2)
{
        int left = *(const int *)p1;
        int right = *(const int *)p2;

        return ((left > right) - (left < right));
}

/*
 * Sort an array of 'int' values and print it to standard output.
 */
int
main(void)
{
        int int_array[] = { 4, 5, 9, 3, 0, 1, 7, 2, 8, 6 };
        size_t array_size = sizeof(int_array) / sizeof(int_array[0]);
        size_t k;

        qsort(&int_array, array_size, sizeof(int_array[0]), int_compare);
        for (k = 0; k < array_size; k++)
                printf(" %d", int_array[k]);
        puts("");
        return (EXIT_SUCCESS);
}
```

COMPATIBILITY

Previous versions of **qsort**() did not permit the comparison routine itself to call **qsort**(*3*). This is no longer true.

ERRORS

The **heapsort**() and **mergesort**() functions succeed unless:

[EINVAL] The *size* argument is zero, or, the *size* argument to **mergesort**() is less than "sizeof(void *) / 2".

[ENOMEM] The **heapsort**() or **mergesort**() functions were unable to allocate memory.

SEE ALSO

sort(1), radixsort(3)

Hoare, C.A.R., "Quicksort", *The Computer Journal*, 5:1, pp. 10-15, 1962.

Williams, J.W.J, "Heapsort", *Communications of the ACM*, 7:1, pp. 347-348, 1964.

Knuth, D.E., "Sorting and Searching", *The Art of Computer Programming*, Vol. 3, pp. 114-123, 145-149, 1968.

McIlroy, P.M., "Optimistic Sorting and Information Theoretic Complexity", *Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1992.

Bentley, J.L. and McIlroy, M.D., "Engineering a Sort Function", *Software--Practice and Experience*, Vol. 23(11), pp. 1249-1265, November 1993.

STANDARDS

The **qsort**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

HISTORY

The variants of these functions that take blocks as arguments first appeared in Mac OS X. This implementation was created by David Chisnall.

NAME

htonl, **htons**, **ntohl**, **ntohs** - convert values between host and network byte order

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <arpa/inet.h>

or

#include <netinet/in.h>

uint32_t
htonl(*uint32_t hostlong*);

uint16_t
htons(*uint16_t hostshort*);

uint32_t
ntohl(*uint32_t netlong*);

uint16_t
ntohs(*uint16_t netshort*);

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines which have a byte order which is the same as the network order, routines are defined as null macros.

These routines are most often used in conjunction with Internet addresses and ports as returned by gethostbyname(3) and getservent(3).

SEE ALSO

gethostbyname(3), getservent(3), byteorder(9)

STANDARDS

The **byteorder** functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **byteorder** functions appeared in 4.2BSD.

BUGS

On the VAX bytes are handled backwards from most everyone else in the world. This is not expected to be fixed in the near future.

NAME

iconv_canonicalize - resolving character encoding names to canonical form

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <iconv.h>

*const char **

iconv_canonicalize(*const char *name*);

DESCRIPTION

The **iconv_canonicalize**() function resolves the character encoding name specified by the *name* argument to its canonical form.

RETURN VALUES

Upon successful completion **iconv_canonicalize**(), returns the canonical name of the given encoding. If the specified name is already a canonical name, the same value is returned. If the specified name is not an existing character encoding name, NULL is returned.

SEE ALSO

iconv(3)

STANDARDS

The **iconv_canonicalize** function is a non-standard extension, which appeared in the GNU implementation and was adopted in FreeBSD 9.0 for compatibility's sake.

AUTHORS

This manual page was written by Gabor Kovesdan <*gabor@FreeBSD.org*>.

NAME

iconv_open, **iconv_open_into**, **iconv_close**, **iconv** - codeset conversion functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <iconv.h>

iconv_t
iconv_open(*const char *dstname*, *const char *srcname*);

int
iconv_open_into(*const char *dstname*, *const char *srcname*, *iconv_allocation_t *ptr*);

int
iconv_close(*iconv_t cd*);

size_t
iconv(*iconv_t cd*, *char ** restrict src*, *size_t * restrict srcleft*, *char ** restrict dst*, *size_t * restrict dstleft*);

size_t
__iconv(*iconv_t cd*, *char ** restrict src*, *size_t * restrict srcleft*, *char ** restrict dst*,
*size_t * restrict dstleft*, *uint32_t flags*, *size_t * invalids*);

DESCRIPTION

The **iconv_open**() function opens a converter from the codeset *srcname* to the codeset *dstname* and returns its descriptor. The arguments *srcname* and *dstname* accept "" and "char", which refer to the current locale encoding.

The **iconv_open_into**() creates a conversion descriptor on a preallocated space. The *iconv_allocation_t* is used as a spaceholder type when allocating such space. The *dstname* and *srcname* arguments are the same as in the case of **iconv_open**(). The *ptr* argument is a pointer of *iconv_allocation_t* to the preallocated space.

The **iconv_close**() function closes the specified converter *cd*.

The **iconv**() function converts the string in the buffer **src* of length **srcleft* bytes and stores the converted string in the buffer **dst* of size **dstleft* bytes. After calling **iconv**(), the values pointed to by *src*, *srcleft*, *dst*, and *dstleft* are updated as follows:

*src Pointer to the byte just after the last character fetched.

*srcleft Number of remaining bytes in the source buffer.

*dst Pointer to the byte just after the last character stored.

*dstleft Number of remainder bytes in the destination buffer.

If the string pointed to by **src* contains a byte sequence which is not a valid character in the source codeset, the conversion stops just after the last successful conversion. If the output buffer is too small to store the converted character, the conversion also stops in the same way. In these cases, the values pointed to by *src*, *srcleft*, *dst*, and *dstleft* are updated to the state just after the last successful conversion.

If the string pointed to by **src* contains a character which is valid under the source codeset but can not be converted to the destination codeset, the character is replaced by an "invalid character" which depends on the destination codeset, e.g., '?', and the conversion is continued. **iconv**() returns the number of such "invalid conversions".

There are two special cases of **iconv**():

src == NULL || *src == NULL
: If the source and/or destination codesets are stateful, **iconv**() places these into their initial state.

  If both *dst* and **dst* are non-NULL, **iconv**() stores the shift sequence for the destination switching to the initial state in the buffer pointed to by **dst*. The buffer size is specified by the value pointed to by *dstleft* as above. **iconv**() will fail if the buffer is too small to store the shift sequence.

  On the other hand, *dst* or **dst* may be NULL. In this case, the shift sequence for the destination switching to the initial state is discarded.

The **__iconv**() function works just like **iconv**() but if **iconv**() fails, the invalid character count is lost there. This is a not bug rather a limitation of IEEE Std 1003.1-2008 ("POSIX.1"), so **__iconv**() is provided as an alternative but non-standard interface. It also has a flags argument, where currently the following flags can be passed:

__ICONV_F_HIDE_INVALID
: Skip invalid characters, instead of returning with an error.

RETURN VALUES

Upon successful completion of **iconv_open**(), it returns a conversion descriptor. Otherwise,

iconv_open() returns (iconv_t)-1 and sets errno to indicate the error.

Upon successful completion of **iconv_open_into**(), it returns 0. Otherwise, **iconv_open_into**() returns -1, and sets errno to indicate the error.

Upon successful completion of **iconv_close**(), it returns 0. Otherwise, **iconv_close**() returns -1 and sets errno to indicate the error.

Upon successful completion of **iconv**(), it returns the number of "invalid" conversions. Otherwise, **iconv**() returns (size_t)-1 and sets errno to indicate the error.

ERRORS

The **iconv_open**() function may cause an error in the following cases:

[ENOMEM] Memory is exhausted.

[EINVAL] There is no converter specified by *srcname* and *dstname*.

The **iconv_open_into**() function may cause an error in the following cases:

[EINVAL] There is no converter specified by *srcname* and *dstname*.

The **iconv_close**() function may cause an error in the following case:

[EBADF] The conversion descriptor specified by *cd* is invalid.

The **iconv**() function may cause an error in the following cases:

[EBADF] The conversion descriptor specified by *cd* is invalid.

[EILSEQ] The string pointed to by **src* contains a byte sequence which does not describe a valid character of the source codeset.

[E2BIG] The output buffer pointed to by **dst* is too small to store the result string.

[EINVAL] The string pointed to by **src* terminates with an incomplete character or shift sequence.

SEE ALSO

iconv(1), mkcsmapper(1), mkesdb(1), __iconv_get_list(3), iconv_canonicalize(3), iconvctl(3), iconvlist(3)

STANDARDS

The **iconv_open**(), **iconv_close**(), and **iconv**() functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

The **iconv_open_into**() function is a GNU-specific extension and it is not part of any standard, thus its use may break portability. The **__iconv**() function is an own extension and it is not part of any standard, thus its use may break portability.

NAME

iconvctl - controlling and diagnostical facility for iconv(3)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <iconv.h>

int
iconvctl(*iconv_t cd*, *int request*, *void *argument*);

DESCRIPTION

The **iconvctl**() function can retrieve or set specific conversion setting from the *cd* conversion descriptor. The *request* parameter specifies the operation to accomplish and *argument* is an operation-specific argument.

The possible operations are the following:

ICONV_TRIVIALP
: In this case *argument* is an *int* * variable, which is set to 1 if the encoding is trivial one, i.e. the input and output encodings are the same. Otherwise, the variable will be 0.

ICONV_GET_TRANSLITERATE
: Determines if transliteration is enabled. The answer is stored in *argument*, which is of *int* *. It will be set to 1 if this feature is enabled or set to 0 otherwise.

ICONV_SET_TRANSLITERATE
: Enables transliteration if *argument*, which is of *int* * set to 1 or disables it if *argument* is set to 0.

ICONV_GET_DISCARD_ILSEQ
: Determines if illegal sequences are discarded or not. The answer is stored in *argument*, which is of *int* *. It will be set to 1 if this feature is enabled or set to 0 otherwise.

ICONV_SET_DISCARD_ILSEQ
: Sets whether illegal sequences are discarded or not. *argument*, which is of *int* * set to 1 or disables it if *argument* is set to 0.

ICONV_SET_HOOKS
: Sets callback functions, which will be called back after successful conversions. The callback

functions are stored in a *struct iconv_hooks* variable, which is passed to **iconvctl** via *argument* by its address.

ICONV_GET_ILSEQ_INVALID
: Determines if a character in the input buffer that is valid, but for which an identical character does not exist in the target codeset returns EILSEQ or not. The answer is stored in *argument*, which is of *int* *. It will be set to 1 if this feature is enabled or set to 0 otherwise.

ICONV_SET_ILSEQ_INVALID
: Sets whether a character in the input buffer that is valid, but for which an identical character does not exist in the target codeset returns EILSEQ or not. If *argument*, which is of *int* * is set to 1 it will be enabled, and if *argument* is set to 0 it will be disabled.

RETURN VALUES

Upon successful completion **iconvctl**(), returns 0. Otherwise, -1 is returned and errno is set to specify the kind of error.

ERRORS

The **iconvctl**() function may cause an error in the following cases:

[EINVAL] Unknown or unimplemented operation.

[EBADF] The conversion descriptor specified by *cd* is invalid.

SEE ALSO

iconv(1), iconv(3)

STANDARDS

The **iconvctl** facility is a non-standard extension, which appeared in the GNU implementation and was adopted in FreeBSD 9.0 for compatibility's sake.

AUTHORS

This manual page was written by Gabor Kovesdan <*gabor@FreeBSD.org*>.

BUGS

Transliteration is enabled in this implementation by default, so it is impossible by design to turn it off. Accordingly, trying to turn it off will always fail and -1 will be returned. Getting the transliteration state will always succeed and indicate that it is turned on, though.

NAME

iconvlist - retrieving a list of character encodings supported by iconv(3)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <iconv.h>

void
iconvlist(*int (*do_one)(unsigned int *count, const char * const *names, void *arg)*, *void *arg*);

DESCRIPTION

The **iconvlist**() function obtains a list of character encodings that are supported by the iconv(3) call. The **do_one**() callback function will be called, where the *count* argument will be set to the number of the encoding names found, the *names* argument will be the list of the supported encoding names and the *arg* argument will be the *arg* argument of the **iconvlist**() function. This argument can be used to interchange custom data between the caller of **iconvlist**() and the callback function.

If an error occurs, *names* will be NULL when calling **do_one**().

SEE ALSO

__iconv_free_list(3), __iconv_get_list(3), iconv(3)

STANDARDS

The **iconvlist** function is a non-standard extension, which appeared in the GNU implementation and was adopted in FreeBSD 9.0 for compatibility's sake.

AUTHORS

This manual page was written by Gabor Kovesdan <*gabor@FreeBSD.org*>.

NAME

imaxabs - returns absolute value

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <inttypes.h>

intmax_t
imaxabs(*intmax_t j*);

DESCRIPTION

The **imaxabs**() function returns the absolute value of *j*.

SEE ALSO

abs(3), fabs(3), hypot(3), labs(3), llabs(3), math(3)

STANDARDS

The **imaxabs**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The **imaxabs**() function first appeared in FreeBSD 5.0.

BUGS

The absolute value of the most negative integer remains negative.

NAME

imaxdiv - returns quotient and remainder

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <inttypes.h>

imaxdiv_t
imaxdiv(*intmax_t numer*, *intmax_t denom*);

DESCRIPTION

The **imaxdiv**() function computes the value of *numer* divided by *denom* and returns the stored result in the form of the *imaxdiv_t* type.

The *imaxdiv_t* type is defined as:

```
typedef struct {
        intmax_t quot; /* Quotient. */
        intmax_t rem;  /* Remainder. */
} imaxdiv_t;
```

SEE ALSO

div(3), ldiv(3), lldiv(3), math(3)

STANDARDS

The **imaxdiv**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The **imaxdiv**() function first appeared in FreeBSD 5.0.

NAME

index, **rindex** - locate character in string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <strings.h>

*char **
index(*const char *s*, *int c*);

*char **
rindex(*const char *s*, *int c*);

DESCRIPTION

The index() and rindex() functions have been deprecated in favor of strchr(3) and strrchr(3).

The **index**() function locates the first occurrence of *c* (converted to a *char*) in the string pointed to by *s*. The terminating null character is considered part of the string; therefore if *c* is '\0', the functions locate the terminating '\0'.

The **rindex**() function is identical to **index**(), except it locates the last occurrence of *c*.

RETURN VALUES

The functions **index**() and **rindex**() return a pointer to the located character, or NULL if the character does not appear in the string.

SEE ALSO

memchr(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

HISTORY

The **index**() and **rindex**() functions appeared in Version 6 AT&T UNIX. Their prototypes existed previously in <*string.h*> before they were moved to <*strings.h*> for IEEE Std 1003.1-2001 ("POSIX.1") compliance. The functions are not specified by IEEE Std 1003.1-2008 ("POSIX.1").

NAME

inet_aton, **inet_addr**, **inet_network**, **inet_ntoa**, **inet_ntoa_r**, **inet_ntop**, **inet_pton**, **inet_makeaddr**, **inet_lnaof**, **inet_netof** - Internet address manipulation routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int
inet_aton(*const char *cp*, *struct in_addr *pin*);

in_addr_t
inet_addr(*const char *cp*);

in_addr_t
inet_network(*const char *cp*);

*char **
inet_ntoa(*struct in_addr in*);

*char **
inet_ntoa_r(*struct in_addr in*, *char *buf*, *socklen_t size*);

*const char **
inet_ntop(*int af*, *const void * restrict src*, *char * restrict dst*, *socklen_t size*);

int
inet_pton(*int af*, *const char * restrict src*, *void * restrict dst*);

struct in_addr
inet_makeaddr(*in_addr_t net*, *in_addr_t lna*);

in_addr_t
inet_lnaof(*struct in_addr in*);

in_addr_t
inet_netof(*struct in_addr in*);

DESCRIPTION

The routines **inet_aton**(), **inet_addr**() and **inet_network**() interpret character strings representing numbers expressed in the Internet standard '.' notation.

The **inet_pton**() function converts a presentation format address (that is, printable form as held in a character string) to network format (usually a *struct in_addr* or some other internal binary representation, in network byte order). It returns 1 if the address was valid for the specified address family, or 0 if the address was not parseable in the specified address family, or -1 if some system error occurred (in which case *errno* will have been set). This function is presently valid for AF_INET and AF_INET6.

The **inet_aton**() routine interprets the specified character string as an Internet address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The **inet_addr**() and **inet_network**() functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively.

The function **inet_ntop**() converts an address **src* from network format (usually a *struct in_addr* or some other binary form, in network byte order) to presentation format (suitable for external display purposes). The *size* argument specifies the size, in bytes, of the buffer **dst*. INET_ADDRSTRLEN and INET6_ADDRSTRLEN define the maximum size required to convert an address of the respective type. It returns NULL if a system error occurs (in which case, *errno* will have been set), or it returns a pointer to the destination string. This function is presently valid for AF_INET and AF_INET6.

The routine **inet_ntoa**() takes an Internet address and returns an ASCII string representing the address in '.' notation. The routine **inet_ntoa_r**() is the reentrant version of **inet_ntoa**(). The routine **inet_makeaddr**() takes an Internet network number and a local network address and constructs an Internet address from it. The routines **inet_netof**() and **inet_lnaof**() break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine byte order integer values.

INTERNET ADDRESSES

Values specified using the '.' notation take one of the following forms:

a.b.c.d
a.b.c
a.b

a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on the VAX the bytes referred to above appear as "d.c.b.a". That is, VAX bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a ‘.’ notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by **inet_addr**() and **inet_network**() for malformed requests.

ERRORS

The **inet_ntop**() call fails if:

[ENOSPC] *size* was not large enough to store the presentation form of the address.

[EAFNOSUPPORT] **src* was not an AF_INET or AF_INET6 family address.

SEE ALSO

byteorder(3), getaddrinfo(3), gethostbyname(3), getnameinfo(3), getnetent(3), inet_net(3), hosts(5), networks(5)

IP Version 6 Addressing Architecture, RFC, 2373, July 1998.

STANDARDS

The **inet_ntop**() and **inet_pton**() functions conform to X/Open Networking Services Issue 5.2

("XNS5.2"). Note that **inet_pton**() does not accept 1-, 2-, or 3-part dotted addresses; all four parts must be specified and are interpreted only as decimal values. This is a narrower input set than that accepted by **inet_aton**().

HISTORY

These functions appeared in 4.2BSD.

BUGS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but **inet_addr**() cannot return that value without indicating failure. The newer **inet_aton**() function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by **inet_ntoa**() resides in a static memory area.

The **inet_addr**() function should return a *struct in_addr*.

NAME

inet_net_ntop, **inet_net_pton** - Internet network number manipulation routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

*char **
inet_net_ntop(*int af*, *const void *src*, *int bits*, *char *dst*, *size_t size*);

int
inet_net_pton(*int af*, *const char *src*, *void *dst*, *size_t size*);

DESCRIPTION

The **inet_net_ntop**() function converts an Internet network number from network format (usually a *struct in_addr* or some other binary form, in network byte order) to CIDR presentation format (suitable for external display purposes). The *bits* argument is the number of bits in *src* that are the network number. It returns NULL if a system error occurs (in which case, *errno* will have been set), or it returns a pointer to the destination string.

The **inet_net_pton**() function converts a presentation format Internet network number (that is, printable form as held in a character string) to network format (usually a *struct in_addr* or some other internal binary representation, in network byte order). It returns the number of bits (either computed based on the class, or specified with /CIDR), or -1 if a failure occurred (in which case *errno* will have been set. It will be set to ENOENT if the Internet network number was not valid).

The currently supported values for *af* are AF_INET and AF_INET6. The *size* argument is the size of the result buffer *dst*.

NETWORK NUMBERS (IP VERSION 4)

Internet network numbers may be specified in one of the following forms:

a.b.c.d/bits
a.b.c.d
a.b.c
a.b

a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet network number. Note that when an Internet network number is viewed as a 32-bit integer quantity on a system that uses little-endian byte order (such as the Intel 386, 486, and Pentium processors) the bytes referred to above appear as "d.c.b.a". That is, little-endian bytes are ordered from right to left.

When a three part number is specified, the last part is interpreted as a 16-bit quantity and placed in the rightmost two bytes of the Internet network number. This makes the three part number format convenient for specifying Class B network numbers as "128.net.host".

When a two part number is supplied, the last part is interpreted as a 24-bit quantity and placed in the rightmost three bytes of the Internet network number. This makes the two part number format convenient for specifying Class A network numbers as "net.host".

When only one part is given, the value is stored directly in the Internet network number without any byte rearrangement.

All numbers supplied as "parts" in a ‘.’ notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

byteorder(3), inet(3), networks(5)

HISTORY

The **inet_net_ntop**() and **inet_net_pton**() functions appeared in BIND 4.9.4.

NAME

initgroups - initialize group access list

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
initgroups(*const char *name*, *gid_t basegid*);

DESCRIPTION

The **initgroups**() function uses the getgrouplist(3) function to calculate the group access list for the user specified in *name*. This group list is then setup for the current process using setgroups(2). The *basegid* is automatically included in the groups list. Typically this value is given as the group number from the password file.

RETURN VALUES

The **initgroups**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **initgroups**() function may fail and set *errno* for any of the errors specified for the library function setgroups(2). It may also return:

[ENOMEM] The **initgroups**() function was unable to allocate temporary storage.

SEE ALSO

setgroups(2), getgrouplist(3)

HISTORY

The **initgroups**() function appeared in 4.2BSD.

NAME

random, **srandom**, **srandomdev**, **initstate**, **setstate** - better random number generator; routines for changing generators

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

long
random(*void*);

void
srandom(*unsigned int seed*);

void
srandomdev(*void*);

*char **
initstate(*unsigned int seed*, *char *state*, *size_t n*);

*char **
setstate(*char *state*);

DESCRIPTION

The functions described in this manual page are not secure. Applications which require unpredictable random numbers should use arc4random(3) instead.

The **random**() function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to (2**31)-1. The period of this random number generator is very large, approximately 16*((2**31)-1).

The **random**() and **srandom**() functions have (almost) the same calling sequence and initialization properties as the rand(3) and srand(3) functions. The difference is that rand(3) produces a much less random sequence -- in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by **random**() are usable. For example, 'random()&01' will produce a random binary value.

Like rand(3), **random**() will by default produce a sequence of numbers that can be duplicated by calling

srandom() with ‘1’ as the seed.

The **srandomdev**() routine initializes a state array using pseudo-random numbers obtained from the kernel. Note that this particular seeding procedure can generate states which are impossible to reproduce by calling **srandom**() with any value, since the succeeding terms in the state buffer are no longer derived from the LC algorithm applied to a fixed seed.

The **initstate**() routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by **initstate**() to decide how sophisticated a random number generator it should use -- the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error.) The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The **initstate**() function returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate**() routine provides for rapid switching between states. The **setstate**() function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to **initstate**() or **setstate**().

Once a state array has been initialized, it may be restarted at a different point either by calling **initstate**() (with the desired seed, the state array, and its size) or by calling both **setstate**() (with the state array) and **srandom**() (with the desired seed). The advantage of calling both **setstate**() and **srandom**() is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2**69 which should be sufficient for most purposes.

DIAGNOSTICS

If **initstate**() is called with less than 8 bytes of state information, or if **setstate**() detects that the state information has been garbled, NULL is returned.

SEE ALSO

arc4random(3), lrand48(3), rand(3), random(4)

HISTORY

These functions appeared in 4.2BSD.

AUTHORS

Earl T. Cohen

BUGS

About 2/3 the speed of rand(3).

The historical implementation used to have a very weak seeding; the random sequence did not vary much with the seed. The current implementation employs a better pseudo-random number generator for the initial state calculation.

NAME

insque, **remque** - doubly-linked list management

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <search.h>

void
insque(*void *element1*, *void *pred*);

void
remque(*void *element*);

DESCRIPTION

The **insque**() and **remque**() functions encapsulate the ever-repeating task of doing insertion and removal operations on doubly linked lists. The functions expect their arguments to point to a structure whose first and second members are pointers to the next and previous element, respectively. The **insque**() function also allows the *pred* argument to be a NULL pointer for the initialization of a new list's head element.

STANDARDS

The **insque**() and **remque**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **insque**() and **remque**() functions appeared in 4.2BSD. In FreeBSD 5.0, they reappeared conforming to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

intro - introduction to system calls and error numbers

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section provides an overview of the system calls, their error returns, and other common definitions and concepts.

RETURN VALUES

Nearly all of the system calls provide an error number referenced via the external identifier errno. This identifier is defined in <*sys/errno.h*> as

```
extern int * __error();
#define errno (* __error())
```

The *__error()* function returns a pointer to a field in the thread specific structure for threads other than the initial thread. For the initial thread and non-threaded processes, *__error()* returns a pointer to a global *errno* variable that is compatible with the previous definition.

When a system call detects an error, it returns an integer value indicating failure (usually -1) and sets the variable *errno* accordingly. (This allows interpretation of the failure on receiving a -1 and to take action accordingly.) Successful calls never set *errno*; once set, it remains until another error occurs. It should only be examined after an error. Note that a number of system calls overload the meanings of these error numbers, and that the meanings must be interpreted according to the type and circumstances of the call.

The following is a complete list of the errors and their names as given in <*sys/errno.h*>.

0 *Undefined error: 0*.Not used.

1 EPERM *Operation not permitted*.An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resources.

2 ENOENT *No such file or directory*.A component of a specified pathname did not exist, or the pathname was an empty string.

3 ESRCH *No such process*.No process could be found corresponding to that specified by the given process ID.

4 EINTR *Interrupted system call*.An asynchronous signal (such as SIGINT or SIGQUIT) was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted system call will seem to have returned the error condition.

5 EIO *Input/output error*.Some physical input or output error occurred. This error will not be reported until a subsequent operation on the same file descriptor and may be lost (over written) by any subsequent errors.

6 ENXIO *Device not configured*.Input or output on a special file referred to a device that did not exist, or made a request beyond the limits of the device. This error may also occur when, for example, a tape drive is not online or no disk pack is loaded on a drive.

7 E2BIG *Argument list too long*.The number of bytes used for the argument and environment list of the new process exceeded the current limit (NCARGS in <*sys/param.h*>).

8 ENOEXEC *Exec format error*.A request was made to execute a file that, although it has the appropriate permissions, was not in the format required for an executable file.

9 EBADF *Bad file descriptor*.A file descriptor argument was out of range, referred to no open file, or a read (write) request was made to a file that was only open for writing (reading).

10 ECHILD *No child processes*.A wait(2) or waitpid(2) function was executed by a process that had no existing or unwaited-for child processes.

11 EDEADLK *Resource deadlock avoided*.An attempt was made to lock a system resource that would have resulted in a deadlock situation.

12 ENOMEM *Cannot allocate memory*.The new process image required more memory than was allowed by the hardware or by system-imposed memory management constraints. A lack of swap space is normally temporary; however, a lack of core is not. Soft limits may be increased to their corresponding hard limits.

13 EACCES *Permission denied*.An attempt was made to access a file in a way forbidden by its file access permissions.

14 EFAULT *Bad address*.The system detected an invalid address in attempting to use an argument of a call.

15 ENOTBLK *Block device required*.A block device operation was attempted on a non-block device or file.

16 EBUSY *Device busy*.An attempt to use a system resource which was in use at the time in a manner which would have conflicted with the request.

17 EEXIST *File exists*.An existing file was mentioned in an inappropriate context, for instance, as the new link name in a link(2) system call.

18 EXDEV *Cross-device link*.A hard link to a file on another file system was attempted.

19 ENODEV *Operation not supported by device*.An attempt was made to apply an inappropriate function to a device, for example, trying to read a write-only device such as a printer.

20 ENOTDIR *Not a directory*.A component of the specified pathname existed, but it was not a directory, when a directory was expected.

21 EISDIR *Is a directory*.An attempt was made to open a directory with write mode specified.

22 EINVAL *Invalid argument*.Some invalid argument was supplied. (For example, specifying an undefined signal to a signal(3) function or a kill(2) system call).

23 ENFILE *Too many open files in system*.Maximum number of open files allowable on the system has been reached and requests for an open cannot be satisfied until at least one has been closed.

24 EMFILE *Too many open files*.Maximum number of file descriptors allowable in the process has been reached and requests for an open cannot be satisfied until at least one has been closed. The getdtablesize(2) system call will obtain the current limit.

25 ENOTTY *Inappropriate ioctl for device*.A control function (see ioctl(2)) was attempted for a file or special device for which the operation was inappropriate.

26 ETXTBSY *Text file busy*.The new process was a pure procedure (shared text) file which was open for writing by another process, or while the pure procedure file was being executed an open(2) call requested write access.

27 EFBIG *File too large*.The size of a file exceeded the maximum.

28 ENOSPC *No space left on device*.A write(2) to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks were

available on the file system, or the allocation of an inode for a newly created file failed because no more inodes were available on the file system.

29 ESPIPE *Illegal seek*.An lseek(2) system call was issued on a socket, pipe or FIFO.

30 EROFS *Read-only file system*.An attempt was made to modify a file or directory on a file system that was read-only at the time.

31 EMLINK *Too many links*.Maximum allowable hard links to a single file has been exceeded (limit of 32767 hard links per file).

32 EPIPE *Broken pipe*.A write on a pipe, socket or FIFO for which there is no process to read the data.

33 EDOM *Numerical argument out of domain*.A numerical input argument was outside the defined domain of the mathematical function.

34 ERANGE *Result too large*.A numerical result of the function was too large to fit in the available space (perhaps exceeded precision).

35 EAGAIN *Resource temporarily unavailable*.This is a temporary condition and later calls to the same routine may complete normally.

36 EINPROGRESS *Operation now in progress*.An operation that takes a long time to complete (such as a connect(2)) was attempted on a non-blocking object (see fcntl(2)).

37 EALREADY *Operation already in progress*.An operation was attempted on a non-blocking object that already had an operation in progress.

38 ENOTSOCK *Socket operation on non-socket*.Self-explanatory.

39 EDESTADDRREQ *Destination address required*.A required address was omitted from an operation on a socket.

40 EMSGSIZE *Message too long*.A message sent on a socket was larger than the internal message buffer or some other network limit.

41 EPROTOTYPE *Protocol wrong type for socket*.A protocol was specified that does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type SOCK_STREAM.

42 ENOPROTOOPT *Protocol not available*.A bad option or level was specified in a getsockopt(2) or setsockopt(2) call.

43 EPROTONOSUPPORT *Protocol not supported*.The protocol has not been configured into the system or no implementation for it exists.

44 ESOCKTNOSUPPORT *Socket type not supported*.The support for the socket type has not been configured into the system or no implementation for it exists.

45 EOPNOTSUPP *Operation not supported*.The attempted operation is not supported for the type of object referenced. Usually this occurs when a file descriptor refers to a file or socket that cannot support this operation, for example, trying to *accept* a connection on a datagram socket.

46 EPFNOSUPPORT *Protocol family not supported*.The protocol family has not been configured into the system or no implementation for it exists.

47 EAFNOSUPPORT *Address family not supported by protocol family*.An address incompatible with the requested protocol was used. For example, you should not necessarily expect to be able to use NS addresses with ARPA Internet protocols.

48 EADDRINUSE *Address already in use*.Only one usage of each address is normally permitted.

49 EADDRNOTAVAIL *Can't assign requested address*.Normally results from an attempt to create a socket with an address not on this machine.

50 ENETDOWN *Network is down*.A socket operation encountered a dead network.

51 ENETUNREACH *Network is unreachable*.A socket operation was attempted to an unreachable network.

52 ENETRESET *Network dropped connection on reset*.The host you were connected to crashed and rebooted.

53 ECONNABORTED *Software caused connection abort*.A connection abort was caused internal to your host machine.

54 ECONNRESET *Connection reset by peer*.A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.

55 ENOBUFS *No buffer space available*.An operation on a socket or pipe was not performed because

the system lacked sufficient buffer space or because a queue was full.

56 EISCONN *Socket is already connected*.A connect(2) request was made on an already connected socket; or, a sendto(2) or sendmsg(2) request on a connected socket specified a destination when already connected.

57 ENOTCONN *Socket is not connected*.An request to send or receive data was disallowed because the socket was not connected and (when sending on a datagram socket) no address was supplied.

58 ESHUTDOWN *Can't send after socket shutdown*.A request to send data was disallowed because the socket had already been shut down with a previous shutdown(2) call.

60 ETIMEDOUT *Operation timed out*.A connect(2) or send(2) request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

61 ECONNREFUSED *Connection refused*.No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

62 ELOOP *Too many levels of symbolic links*.A path name lookup involved more than 32 (MAXSYMLINKS) symbolic links.

63 ENAMETOOLONG *File name too long*.A component of a path name exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters. (See also the description of _PC_NO_TRUNC in pathconf(2).)

64 EHOSTDOWN *Host is down*.A socket operation failed because the destination host was down.

65 EHOSTUNREACH *No route to host*.A socket operation was attempted to an unreachable host.

66 ENOTEMPTY *Directory not empty*.A directory with entries other than '.' and '..' was supplied to a remove directory or rename call.

67 EPROCLIM *Too many processes*.

68 EUSERS *Too many users*.The quota system ran out of table entries.

69 EDQUOT *Disc quota exceeded*.A write(2) to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was

exhausted, or the allocation of an inode for a newly created file failed because the user's quota of inodes was exhausted.

70 ESTALE *Stale NFS file handle*.An attempt was made to access an open file (on an NFS file system) which is now unavailable as referenced by the file descriptor. This may indicate the file was deleted on the NFS server or some other catastrophic event occurred.

72 EBADRPC *RPC struct is bad*.Exchange of RPC information was unsuccessful.

73 ERPCMISMATCH *RPC version wrong*.The version of RPC on the remote peer is not compatible with the local version.

74 EPROGUNAVAIL *RPC prog. not avail*.The requested program is not registered on the remote host.

75 EPROGMISMATCH *Program version wrong*.The requested version of the program is not available on the remote host (RPC).

76 EPROCUNAVAIL *Bad procedure for program*.An RPC call was attempted for a procedure which does not exist in the remote program.

77 ENOLCK *No locks available*.A system-imposed limit on the number of simultaneous file locks was reached.

78 ENOSYS *Function not implemented*.Attempted a system call that is not available on this system.

79 EFTYPE *Inappropriate file type or format*.The file was the wrong type for the operation, or a data file had the wrong format.

80 EAUTH *Authentication error*.Attempted to use an invalid authentication ticket to mount a NFS file system.

81 ENEEDAUTH *Need authenticator*.An authentication ticket must be obtained before the given NFS file system may be mounted.

82 EIDRM *Identifier removed*.An IPC identifier was removed while the current process was waiting on it.

83 ENOMSG *No message of desired type*.An IPC message queue does not contain a message of the desired type, or a message catalog does not contain the requested message.

84 EOVERFLOW *Value too large to be stored in data type*.A numerical result of the function was too large to be stored in the caller provided space.

85 ECANCELED *Operation canceled*.The scheduled operation was canceled.

86 EILSEQ *Illegal byte sequence*.While decoding a multibyte character the function came along an invalid or an incomplete sequence of bytes or the given wide character is invalid.

87 ENOATTR *Attribute not found*.The specified extended attribute does not exist.

88 EDOOFUS *Programming error*.A function or API is being abused in a way which could only be detected at run-time.

89 EBADMSG *Bad message*.A corrupted message was detected.

90 EMULTIHOP *Multihop attempted*.This error code is unused, but present for compatibility with other systems.

91 ENOLINK *Link has been severed*.This error code is unused, but present for compatibility with other systems.

92 EPROTO *Protocol error*.A device or socket encountered an unrecoverable protocol error.

93 ENOTCAPABLE *Capabilities insufficient*.An operation on a capability file descriptor requires greater privilege than the capability allows.

94 ECAPMODE *Not permitted in capability mode*.The system call or operation is not permitted for capability mode processes.

95 ENOTRECOVERABLE *State not recoverable*.The state protected by a robust mutex is not recoverable.

96 EOWNERDEAD *Previous owner died*.The owner of a robust mutex terminated while holding the mutex lock.

DEFINITIONS

Process ID.

Each active process in the system is uniquely identified by a non-negative integer called a process ID. The range of this ID is from 0 to 99999.

Parent process ID

A new process is created by a currently active process (see fork(2)). The parent process ID of a process is initially the process ID of its creator. If the creating process exits, the parent process ID of each child is set to the ID of the calling process's reaper (see procctl(2)), normally init(8).

Process Group

Each active process is a member of a process group that is identified by a non-negative integer called the process group ID. This is the process ID of the group leader. This grouping permits the signaling of related processes (see termios(4)) and the job control mechanisms of csh(1).

Session

A session is a set of one or more process groups. A session is created by a successful call to setsid(2), which causes the caller to become the only member of the only process group in the new session.

Session leader

A process that has created a new session by a successful call to setsid(2), is known as a session leader. Only a session leader may acquire a terminal as its controlling terminal (see termios(4)).

Controlling process

A session leader with a controlling terminal is a controlling process.

Controlling terminal

A terminal that is associated with a session is known as the controlling terminal for that session and its members.

Terminal Process Group ID

A terminal may be acquired by a session leader as its controlling terminal. Once a terminal is associated with a session, any of the process groups within the session may be placed into the foreground by setting the terminal process group ID to the ID of the process group. This facility is used to arbitrate between multiple jobs contending for the same terminal; (see csh(1) and tty(4)).

Orphaned Process Group

A process group is considered to be *orphaned* if it is not under the control of a job control shell. More precisely, a process group is orphaned when none of its members has a parent process that is in the same session as the group, but is in a different process group. Note that when a process exits, the parent process for its children is normally changed to be init(8), which is in a separate session. Not all members of an orphaned process group are necessarily orphaned processes (those whose creating process has exited). The process group of a session leader is orphaned by

definition.

Real User ID and Real Group ID

Each user on the system is identified by a positive integer termed the real user ID.

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID.

All processes have a real user ID and real group ID. These are initialized from the equivalent attributes of the process that created it.

Effective User Id, Effective Group Id, and Group Access List

Access to system resources is governed by two values: the effective user ID, and the group access list. The first member of the group access list is also known as the effective group ID. (In POSIX.1, the group access list is known as the set of supplementary group IDs, and it is unspecified whether the effective group ID is a member of the list.)

The effective user ID and effective group ID are initially the process's real user ID and real group ID respectively. Either may be modified through execution of a set-user-ID or set-group-ID file (possibly by one its ancestors) (see execve(2)). By convention, the effective group ID (the first member of the group access list) is duplicated, so that the execution of a set-group-ID program does not result in the loss of the original (real) group ID.

The group access list is a set of group IDs used only in determining resource accessibility. Access checks are performed as described below in ''File Access Permissions''.

Saved Set User ID and Saved Set Group ID

When a process executes a new file, the effective user ID is set to the owner of the file if the file is set-user-ID, and the effective group ID (first element of the group access list) is set to the group of the file if the file is set-group-ID. The effective user ID of the process is then recorded as the saved set-user-ID, and the effective group ID of the process is recorded as the saved set-group-ID. These values may be used to regain those values as the effective user or group ID after reverting to the real ID (see setuid(2)). (In POSIX.1, the saved set-user-ID and saved set-group-ID are optional, and are used in setuid and setgid, but this does not work as desired for the super-user.)

Super-user

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Descriptor

An integer assigned by the system when a file is referenced by open(2) or dup(2), or when a socket is created by pipe(2), socket(2) or socketpair(2), which uniquely identifies an access path to that file or socket from a given process or any of its children.

File Name

Names consisting of up to {NAME_MAX} characters may be used to name an ordinary file, special file, or directory.

These characters may be arbitrary eight-bit values, excluding NUL (ASCII 0) and the '/' character (slash, ASCII 47).

Note that it is generally unwise to use '*', '?', '[' or ']' as part of file names because of the special meaning attached to these characters by the shell.

Path Name

A path name is a NUL-terminated character string starting with an optional slash '/', followed by zero or more directory names separated by slashes, optionally followed by a file name. The total length of a path name must be less than {PATH_MAX} characters. (On some systems, this limit may be infinite.)

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory. A slash by itself names the root directory. An empty pathname refers to the current directory.

Directory

A directory is a special type of file that contains entries that are references to other files. Directory entries are called links. By convention, a directory contains at least two links, '.' and '..', referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. A process's root directory need not be the root directory of the root file system.

File Access Permissions

Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They may be

changed at some later time through the chmod(2) call.

File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file's group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process's effective user ID is that of the super-user. (Note: even the super-user cannot execute a non-executable file.)

The process's effective user ID matches the user ID of the owner of the file and the owner permissions allow the access.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the group permissions allow the access.

Neither the effective user ID nor effective group ID and group access list of the process match the corresponding user ID and group ID of the file, but the permissions for ‘‘other users’’ allow access.

Otherwise, permission is denied.

Sockets and Address Families

A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types; consult socket(2) for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

SEE ALSO

intro(3), perror(3)

NAME

ioctl - control device

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/ioctl.h>

int
ioctl(*int fd*, *unsigned long request*, ...);

DESCRIPTION

The **ioctl**() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl**() requests. The argument *fd* must be an open file descriptor.

The third argument to **ioctl**() is traditionally named *char *argp*. Most uses of **ioctl**(), however, require the third argument to be a *caddr_t* or an *int*.

An **ioctl**() *request* has encoded in it whether the argument is an "in" argument or "out" argument, and the size of the argument *argp* in bytes. Macros and defines used in specifying an ioctl *request* are located in the file <*sys/ioctl.h*>.

GENERIC IOCTLS

Some generic ioctls are not implemented for all types of file descriptors. These include:

FIONREAD int
: Get the number of bytes that are immediately available for reading.

FIONWRITE int
: Get the number of bytes in the descriptor's send queue. These bytes are data which has been written to the descriptor but which are being held by the kernel for further processing. The nature of the required processing depends on the underlying device. For TCP sockets, these bytes have not yet been acknowledged by the other side of the connection.

FIONSPACE int
: Get the free space in the descriptor's send queue. This value is the size of the send queue minus the number of bytes being held in the queue. Note: while this value represents the number of bytes that may be added to the queue, other resource limitations may cause a write not larger

than the send queue's space to be blocked. One such limitation would be a lack of network buffers for a write to a network connection.

RETURN VALUES

If an error has occurred, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **ioctl**() system call will fail if:

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

[ENOTTY] The *fd* argument is not associated with a character special device.

[ENOTTY] The specified request does not apply to the kind of object that the descriptor *fd* references.

[EINVAL] The *request* or *argp* argument is not valid.

[EFAULT] The *argp* argument points outside the process's allocated address space.

SEE ALSO

execve(2), fcntl(2), intro(4), tty(4)

HISTORY

The **ioctl**() function appeared in Version 7 AT&T UNIX.

NAME

rcmd, **rresvport**, **iruserok**, **ruserok**, **rcmd_af**, **rresvport_af**, **iruserok_sa** - routines for returning a stream to a remote command

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
rcmd(*char **ahost*, *int inport*, *const char *locuser*, *const char *remuser*, *const char *cmd*, *int *fd2p*);

int
rresvport(*int *port*);

int
iruserok(*u_long raddr*, *int superuser*, *const char *ruser*, *const char *luser*);

int
ruserok(*const char *rhost*, *int superuser*, *const char *ruser*, *const char *luser*);

int
rcmd_af(*char **ahost*, *int inport*, *const char *locuser*, *const char *remuser*, *const char *cmd*, *int *fd2p*, *int af*);

int
rresvport_af(*int *port*, *int af*);

int
iruserok_sa(*const void *addr*, *int addrlen*, *int superuser*, *const char *ruser*, *const char *luser*);

DESCRIPTION

The **rcmd**() function is used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. The **rresvport**() function returns a descriptor to a socket with an address in the privileged port space. The **ruserok**() function is used by servers to authenticate clients requesting service with **rcmd**(). All three functions are present in the same file and are used by the rshd(8) server (among others).

The **rcmd**() function looks up the host **ahost* using gethostbyname(3), returning -1 if the host does not

exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as stdin and stdout. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the stderr (unit 2 of the remote command) will be made the same as the stdout and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in rshd(8).

The **rresvport**() function is used to obtain a socket to which an address with a Privileged Internet port is bound. This socket is suitable for use by **rcmd**() and several other functions. Privileged Internet ports are those in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

The **iruserok**() and **ruserok**() functions take a remote host's IP address or name, as returned by the gethostbyname(3) routines, two user names and a flag indicating whether the local user's name is that of the super-user. Then, if the user is *NOT* the super-user, it checks the */etc/hosts.equiv* file. If that lookup is not done, or is unsuccessful, the *.rhosts* in the local user's home directory is checked to see if the request for service is allowed.

If this file does not exist, is not a regular file, is owned by anyone other than the user or the super-user, or is writable by anyone other than the owner, the check automatically fails. Zero is returned if the machine name is listed in the "*hosts.equiv*" file, or the host and remote user name are found in the "*.rhosts*" file; otherwise **iruserok**() and **ruserok**() return -1. If the local domain (as obtained from gethostname(3)) is the same as the remote domain, only the machine name need be specified.

The **iruserok**() function is strongly preferred for security reasons. It requires trusting the local DNS at most, while the **ruserok**() function requires trusting the entire DNS, which can be spoofed.

The functions with an "_af" or "_sa" suffix, i.e., **rcmd_af**(), **rresvport_af**() and **iruserok_sa**(), work the same as the corresponding functions without a suffix, except that they are capable of handling both IPv6 and IPv4 ports.

The "_af" suffix means that the function has an additional *af* argument which is used to specify the address family, (see below). The *af* argument extension is implemented for functions that have no

binary address argument. Instead, the *af* argument specifies which address family is desired.

The "_sa" suffix means that the function has general socket address and length arguments. As the socket address is a protocol independent data structure, IPv4 and IPv6 socket address can be passed as desired. The *sa* argument extension is implemented for functions that pass a protocol dependent binary address argument. The argument needs to be replaced with a more general address structure to support multiple address families in a general way.

The functions with neither an "_af" suffix nor an "_sa" suffix work for IPv4 only, except for **ruserok**() which can handle both IPv6 and IPv4. To switch the address family, the *af* argument must be filled with AF_INET, or AF_INET6. For **rcmd_af**(), PF_UNSPEC is also allowed.

ENVIRONMENT

RSH When using the **rcmd**() function, this variable is used as the program to run instead of rsh(1).

DIAGNOSTICS

The **rcmd**() function returns a valid socket descriptor on success. It returns -1 on error and prints a diagnostic message on the standard error.

The **rresvport**() function returns a valid, bound socket descriptor on success. It returns -1 on error with the global value *errno* set according to the reason for failure. The error code EAGAIN is overloaded to mean ‘‘All network ports in use.’’

SEE ALSO

rlogin(1), rsh(1), intro(2), rlogind(8), rshd(8)

W. Stevens and M. Thomas, *Advanced Socket API for IPv6*, RFC2292.

W. Stevens, M. Thomas, and E. Nordmark, *Advanced Socket API for IPv6*, RFC3542.

HISTORY

Most of these functions appeared in 4.2BSD. The **rresvport_af**() function appeared in RFC2292, and was implemented by the WIDE project for the Hydrangea IPv6 protocol stack kit. The **rcmd_af**() function appeared in draft-ietf-ipngwg-rfc2292bis-01.txt, and was implemented in the WIDE/KAME IPv6 protocol stack kit. The **iruserok_sa**() function appeared in discussion on the IETF ipngwg mailing list, and was implemented in FreeBSD 4.0.

NAME

ttyname, **ttyname_r**, **isatty** - get name of associated terminal (tty) from file descriptor

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*char **
ttyname(*int fd*);

int
ttyname_r(*int fd*, *char *buf*, *size_t len*);

int
isatty(*int fd*);

DESCRIPTION

These functions operate on file descriptors for terminal type devices.

The **isatty**() function determines if the file descriptor *fd* refers to a valid terminal type device.

The **ttyname**() function gets the related device name of a file descriptor for which **isatty**() is true.

The **ttyname**() function returns the name stored in a static buffer which will be overwritten on subsequent calls. The **ttyname_r**() function takes a buffer and length as arguments to avoid this problem.

RETURN VALUES

The **isatty**() function returns 1 if *fd* refers to a terminal type device; otherwise, it returns 0 and may set *errno* to indicate the error. The **ttyname**() function returns the null terminated name if the device is found and **isatty**() is true; otherwise a NULL pointer is returned. The **ttyname_r**() function returns 0 if successful. Otherwise an error number is returned.

ERRORS

These functions may fail if:

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

[ENOTTY] The file associated with *fd* is not a terminal.

Additionally, **ttyname_r**() may fail if:

[ERANGE] The *bufsize* argument is smaller than the length of the string to be returned.

SEE ALSO

fdevname(3), ptsname(3), tcgetattr(3), tty(4)

HISTORY

The **isatty**() and **ttyname**() functions appeared in Version 7 AT&T UNIX. The **ttyname_r**() function appeared in FreeBSD 6.0.

NAME

isgreater, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isunordered** - compare two floating-point numbers

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <math.h>

int
isgreater(*real-floating x*, *real-floating y*);

int
isgreaterequal(*real-floating x*, *real-floating y*);

int
isless(*real-floating x*, *real-floating y*);

int
islessequal(*real-floating x*, *real-floating y*);

int
islessgreater(*real-floating x*, *real-floating y*);

int
isunordered(*real-floating x*, *real-floating y*);

DESCRIPTION

Each of the macros **isgreater**(), **isgreaterequal**(), **isless**(), **islessequal**(), and **islessgreater**() take arguments *x* and *y* and return a non-zero value if and only if its nominal relation on *x* and *y* is true. These macros always return zero if either argument is not a number (NaN), but unlike the corresponding C operators, they never raise a floating point exception.

The **isunordered**() macro takes arguments *x* and *y* and returns non-zero if and only if any of *x* or *y* are NaNs. For any pair of floating-point values, one of the relationships (less, greater, equal, unordered) holds.

SEE ALSO

fpclassify(3), math(3), signbit(3)

STANDARDS

The **isgreater**(), **isgreaterequal**(), **isless**(), **islessequal**(), **islessgreater**(), and **isunordered**() macros conform to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The relational macros described above first appeared in FreeBSD 5.1.

NAME

issetugid - is current process tainted by uid or gid changes

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
issetugid(*void*);

DESCRIPTION

The **issetugid**() system call returns 1 if the process environment or memory address space is considered "tainted", and returns 0 otherwise.

A process is tainted if it was created as a result of an execve(2) system call which had either of the setuid or setgid bits set (and extra privileges were given as a result) or if it has changed any of its real, effective or saved user or group ID's since it began execution.

This system call exists so that library routines (eg: libc, libtermcap) can reliably determine if it is safe to use information that was obtained from the user, in particular the results from getenv(3) should be viewed with suspicion if it is used to control operation.

A "tainted" status is inherited by child processes as a result of the fork(2) system call (or other library code that calls fork, such as popen(3)).

It is assumed that a program that clears all privileges as it prepares to execute another will also reset the environment, hence the "tainted" status will not be passed on. This is important for programs such as su(1) which begin setuid but need to be able to create an untainted process.

ERRORS

The **issetugid**() system call is always successful, and no return value is reserved to indicate an error.

SEE ALSO

execve(2), fork(2), setegid(2), seteuid(2), setgid(2), setregid(2), setreuid(2), setuid(2)

HISTORY

The **issetugid**() system call first appeared in OpenBSD 2.0 and was also implemented in FreeBSD 3.0.

NAME

iswalnum_l, **iswalpha_l**, **iswcntrl_l**, **iswctype_l**, **iswdigit_l**, **iswgraph_l**, **iswlower_l**, **iswprint_l**, **iswpunct_l**, **iswspace_l**, **iswupper_l**, **iswxdigit_l**, **towlower_l**, **towupper_l**, **wctype_l**, **iswblank_l**, **iswhexnumber_l**, **iswideogram_l**, **iswnumber_l**, **iswphonogram_l**, **iswrune_l**, **iswspecial_l**, **nextwctype_l**, **towctrans_l**, **wctrans_l** - wide character classification utilities

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wctype.h>

int
iswalnum_l(*wint_t wc*, *locale_t loc*);

int
iswalpha_l(*wint_t wc*, *locale_t loc*);

int
iswcntrl_l(*wint_t wc*, *locale_t loc*);

int
iswctype_l(*wint_t wc*, *locale_t loc*);

int
iswdigit_l(*wint_t wc*, *locale_t loc*);

int
iswgraph_l(*wint_t wc*, *locale_t loc*);

int
iswlower_l(*wint_t wc*, *locale_t loc*);

int
iswprint_l(*wint_t wc*, *locale_t loc*);

int
iswpunct_l(*wint_t wc*, *locale_t loc*);

int

iswspace_l(*wint_t wc*, *locale_t loc*);

int
iswupper_l(*wint_t wc*, *locale_t loc*);

int
iswxdigit_l(*wint_t wc*, *locale_t loc*);

wint_t
towlower_l(*wint_t wc*, *locale_t loc*);

wint_t
towupper_l(*wint_t wc*, *locale_t loc*);

wctype_t
wctype_l(*wint_t wc*, *locale_t loc*);

int
iswblank_l(*wint_t wc*, *locale_t loc*);

int
iswhexnumber_l(*wint_t wc*, *locale_t loc*);

int
iswideogram_l(*wint_t wc*, *locale_t loc*);

int
iswnumber_l(*wint_t wc*, *locale_t loc*);

int
iswphonogram_l(*wint_t wc*, *locale_t loc*);

int
iswrune_l(*wint_t wc*, *locale_t loc*);

int
iswspecial_l(*wint_t wc*, *locale_t loc*);

wint_t
nextwctype_l(*wint_t wc*, *locale_t loc*);

wint_t
towctrans_l(*wint_t wc*, *wctrans_t*, *locale_t loc*);

wctrans_t
wctrans_l(*const char **, *locale_t loc*);

DESCRIPTION

The above functions are character classification utility functions, for use with wide characters (*wchar_t* or *wint_t*) in the locale *loc*. They behave in the same way as the versions without the _l suffix, but use the specified locale rather than the global or per-thread locale. These functions may be implemented as inline functions in <*wctype.h*> and as functions in the C library. See the specific manual pages for more information.

RETURN VALUES

These functions return the same things as their non-locale versions. If the locale is invalid, their behaviors are undefined.

SEE ALSO

iswalnum(3), iswalpha(3), iswblank(3), iswcntrl(3), iswctype(3), iswdigit(3), iswgraph(3), iswhexnumber(3), iswideogram(3), iswlower(3), iswnumber(3), iswphonogram(3), iswprint(3), iswpunct(3), iswrune(3), iswspace(3), iswspecial(3), iswupper(3), iswxdigit(3), nextwctype(3), towctrans(3), towlower(3), towupper(3), wctrans(3), wctype(3)

STANDARDS

These functions conform to IEEE Std 1003.1-2008 ("POSIX.1"), except for **iswascii_l**(), **iswhexnumber_l**(), **iswideogram_l**(), **iswphonogram_l**(), **iswrune_l**(), **iswspecial_l**() and **nextwctype_l**() which are FreeBSD extensions.

NAME

iswalnum, **iswalpha**, **iswascii**, **iswblank**, **iswcntrl**, **iswdigit**, **iswgraph**, **iswhexnumber**, **iswideogram**, **iswlower**, **iswnumber**, **iswphonogram**, **iswprint**, **iswpunct**, **iswrune**, **iswspace**, **iswspecial**, **iswupper**, **iswxdigit** - wide character classification utilities

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wctype.h>

int
iswalnum(*wint_t wc*);

int
iswalpha(*wint_t wc*);

int
iswascii(*wint_t wc*);

int
iswblank(*wint_t wc*);

int
iswcntrl(*wint_t wc*);

int
iswdigit(*wint_t wc*);

int
iswgraph(*wint_t wc*);

int
iswhexnumber(*wint_t wc*);

int
iswideogram(*wint_t wc*);

int
iswlower(*wint_t wc*);

int
iswnumber(*wint_t wc*);

int
iswphonogram(*wint_t wc*);

int
iswprint(*wint_t wc*);

int
iswpunct(*wint_t wc*);

int
iswrune(*wint_t wc*);

int
iswspace(*wint_t wc*);

int
iswspecial(*wint_t wc*);

int
iswupper(*wint_t wc*);

int
iswxdigit(*wint_t wc*);

DESCRIPTION

The above functions are character classification utility functions, for use with wide characters (*wchar_t* or *wint_t*). See the description for the similarly-named single byte classification functions (like isalnum(3)), for details.

RETURN VALUES

The functions return zero if the character tests false and return non-zero if the character tests true.

SEE ALSO

isalnum(3), isalpha(3), isascii(3), isblank(3), iscntrl(3), isdigit(3), isgraph(3), ishexnumber(3), isideogram(3), islower(3), isnumber(3), isphonogram(3), isprint(3), ispunct(3), isrune(3), isspace(3), isspecial(3), isupper(3), isxdigit(3), wctype(3)

STANDARDS

These functions conform to IEEE Std 1003.1-2001 ("POSIX.1"), except **iswascii**(), **iswhexnumber**(), **iswideogram**(), **iswnumber**(), **iswphonogram**(), **iswrune**() and **iswspecial**(), which are FreeBSD extensions.

CAVEATS

The result of these functions is undefined unless the argument is WEOF or a valid *wchar_t* value for the current locale.

NAME

iswctype, **wctype** - wide character class functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wctype.h>

int
iswctype(*wint_t wc*, *wctype_t charclass*);

wctype_t
wctype(*const char *property*);

DESCRIPTION

The **wctype**() function returns a value of type *wctype_t* which represents the requested wide character class and may be used as the second argument for calls to **iswctype**().

The following character class names are recognised:

alnum	cntrl	ideogram	print	space	xdigit
alpha	digit	lower	punct	special	
blank	graph	phonogram	rune	upper	

The **iswctype**() function checks whether the wide character *wc* is in the character class *charclass*.

RETURN VALUES

The **iswctype**() function returns non-zero if and only if *wc* has the property described by *charclass*, or *charclass* is zero.

The **wctype**() function returns 0 if *property* is invalid, otherwise it returns a value of type *wctype_t* that can be used in subsequent calls to **iswctype**().

EXAMPLES

Reimplement iswalpha(3) in terms of **iswctype**() and **wctype**():

```
int
myiswalpha(wint_t wc)
{
```

```
        return (iswctype(wc, wctype("alpha")));
}
```

SEE ALSO

ctype(3), nextwctype(3)

STANDARDS

The **iswctype**() and **wctype**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1"). The "ideogram", "phonogram", "special", and "rune" character classes are extensions.

HISTORY

The **iswctype**() and **wctype**() functions first appeared in FreeBSD 5.0.

NAME

jail, **jail_get**, **jail_set**, **jail_remove**, **jail_attach** - create and manage system jails

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/jail.h>

int
jail(*struct jail *jail*);

int
jail_attach(*int jid*);

int
jail_remove(*int jid*);

#include <sys/uio.h>

int
jail_get(*struct iovec *iov*, *u_int niov*, *int flags*);

int
jail_set(*struct iovec *iov*, *u_int niov*, *int flags*);

DESCRIPTION

The **jail**() system call sets up a jail and locks the current process in it.

The argument is a pointer to a structure describing the prison:

```
struct jail {
        uint32_t   version;
        char            *path;
        char            *hostname;
        char            *jailname;
        unsigned int    ip4s;
        unsigned int    ip6s;
        struct in_addr  *ip4;
```

```
        struct in6_addr     *ip6;
};
```

"version" defines the version of the API in use. JAIL_API_VERSION is defined for the current version.

The "path" pointer should be set to the directory which is to be the root of the prison.

The "hostname" pointer can be set to the hostname of the prison. This can be changed from the inside of the prison.

The "jailname" pointer is an optional name that can be assigned to the jail for example for management purposes.

The "ip4s" and "ip6s" give the numbers of IPv4 and IPv6 addresses that will be passed via their respective pointers.

The "ip4" and "ip6" pointers can be set to an arrays of IPv4 and IPv6 addresses to be assigned to the prison, or NULL if none. IPv4 addresses must be in network byte order.

This is equivalent to, and deprecated in favor of, the **jail_set**() system call (see below), with the parameters *path*, *host.hostname*, *name*, *ip4.addr*, and *ip6.addr*, and with the JAIL_ATTACH flag.

The **jail_set**() system call creates a new jail, or modifies an existing one, and optionally locks the current process in it. Jail parameters are passed as an array of name-value pairs in the array *iov*, containing *niov* elements. Parameter names are a null-terminated string, and values may be strings, integers, or other arbitrary data. Some parameters are boolean, and do not have a value (their length is zero) but are set by the name alone with or without a "no" prefix, e.g. *persist* or *nopersist*. Any parameters not set will be given default values, generally based on the current environment.

Jails have a set of core parameters, and modules can add their own jail parameters. The current set of available parameters, and their formats, can be retrieved via the *security.jail.param* sysctl MIB entry. Notable parameters include those mentioned in the **jail**() description above, as well as *jid* and *name*, which identify the jail being created or modified. See jail(8) for more information on the core jail parameters.

The *flags* arguments consists of one or more of the following flags:

JAIL_CREATE
: Create a new jail. If a *jid* or *name* parameters exists, they must not refer to an existing jail.

JAIL_UPDATE
: Modify an existing jail. One of the *jid* or *name* parameters must exist, and must refer to an existing jail. If both JAIL_CREATE and JAIL_UPDATE are set, a jail will be created if it does not yet exist, and modified if it does exist.

JAIL_ATTACH
: In addition to creating or modifying the jail, attach the current process to it, as with the **jail_attach**() system call.

JAIL_DYING
: Allow setting a jail that is in the process of being removed.

The **jail_get**() system call retrieves jail parameters, using the same name-value list as **jail_set**() in the *iov* and *niov* arguments. The jail to read can be specified by either *jid* or *name* by including those parameters in the list. If they are included but are not intended to be the search key, they should be cleared (zero and the empty string respectively).

The special parameter *lastjid* can be used to retrieve a list of all jails. It will fetch the jail with the jid above and closest to the passed value. The first jail (usually but not always jid 1) can be found by passing a *lastjid* of zero.

The *flags* arguments consists of one or more following flags:

JAIL_DYING
: Allow getting a jail that is in the process of being removed.

The **jail_attach**() system call attaches the current process to an existing jail, identified by *jid*.

The **jail_remove**() system call removes the jail identified by *jid*. It will kill all processes belonging to the jail, and remove any children of that jail.

RETURN VALUES

If successful, **jail**(), **jail_set**(), and **jail_get**() return a non-negative integer, termed the jail identifier (JID). They return -1 on failure, and set *errno* to indicate the error.

The **jail_attach**() and **jail_remove**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **jail**() system call will fail if:

[EPERM] This process is not allowed to create a jail, either because it is not the super-user, or because it would exceed the jail's *children.max* limit.

[EFAULT] *jail* points to an address outside the allocated address space of the process.

[EINVAL] The version number of the argument is not correct.

[EAGAIN] No free JID could be found.

The **jail_set**() system call will fail if:

[EPERM] This process is not allowed to create a jail, either because it is not the super-user, or because it would exceed the jail's *children.max* limit.

[EPERM] A jail parameter was set to a less restrictive value then the current environment.

[EFAULT] *Iov*, or one of the addresses contained within it, points to an address outside the allocated address space of the process.

[ENOENT] The jail referred to by a *jid* or *name* parameter does not exist, and the JAIL_CREATE flag is not set.

[ENOENT] The jail referred to by a *jid* is not accessible by the process, because the process is in a different jail.

[EEXIST] The jail referred to by a *jid* or *name* parameter exists, and the JAIL_UPDATE flag is not set.

[EINVAL] A supplied parameter is the wrong size.

[EINVAL] A supplied parameter is out of range.

[EINVAL] A supplied string parameter is not null-terminated.

[EINVAL] A supplied parameter name does not match any known parameters.

[EINVAL] One of the JAIL_CREATE or JAIL_UPDATE flags is not set.

[ENAMETOOLONG]
A supplied string parameter is longer than allowed.

[EAGAIN] There are no jail IDs left.

The **jail_get**() system call will fail if:

[EFAULT] *Iov*, or one of the addresses contained within it, points to an address outside the allocated address space of the process.

[ENOENT] The jail referred to by a *jid* or *name* parameter does not exist.

[ENOENT] The jail referred to by a *jid* is not accessible by the process, because the process is in a different jail.

[ENOENT] The *lastjid* parameter is greater than the highest current jail ID.

[EINVAL] A supplied parameter is the wrong size.

[EINVAL] A supplied parameter name does not match any known parameters.

The **jail_attach**() and **jail_remove**() system calls will fail if:

[EPERM] A user other than the super-user attempted to attach to or remove a jail.

[EINVAL] The jail specified by *jid* does not exist.

Further **jail**(), **jail_set**(), and **jail_attach**() call chroot(2) internally, so they can fail for all the same reasons. Please consult the chroot(2) manual page for details.

SEE ALSO

chdir(2), chroot(2), jail(8)

HISTORY

The **jail**() system call appeared in FreeBSD 4.0. The **jail_attach**() system call appeared in FreeBSD 5.1. The **jail_set**(), **jail_get**(), and **jail_remove**() system calls appeared in FreeBSD 8.0.

AUTHORS

The jail feature was written by Poul-Henning Kamp for R&D Associates who contributed it to FreeBSD. James Gritton added the extensible jail parameters and hierarchical jails.

NAME

kenv - kernel environment

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <kenv.h>

int
kenv(*int action*, *const char *name*, *char *value*, *int len*);

DESCRIPTION

The **kenv**() system call manipulates kernel environment variables. It supports the well known userland actions of getting, setting and unsetting environment variables, as well as the ability to dump all of the entries in the kernel environment.

The *action* argument can be one of the following:

KENV_GET — Get the *value* of the variable with the given *name*. The size of the *value* buffer is given by *len*, which should be at least KENV_MVALLEN + 1 bytes to avoid truncation and to ensure NUL termination.

KENV_SET — Set or add a variable. The *name* and *value* are limited to KENV_MNAMELEN and KENV_MVALLEN characters, respectively (not including the NUL terminator.) The *len* argument indicates the length of the *value* and must include the NUL terminator. This option is only available to the superuser.

KENV_UNSET — Unset the variable with the given *name*. The *value* and *len* arguments are ignored. This option is only available to the superuser.

KENV_DUMP — Dump as much of the kernel environment as will fit in *value*, whose size is given in *len*. If *value* is NULL, **kenv**() will return the number of bytes required to copy out the entire environment. The *name* is ignored.

RETURN VALUES

The **kenv**() system call returns 0 if successful in the case of KENV_SET and KENV_UNSET, and the number of bytes copied into *value* in the case of KENV_DUMP and KENV_GET. If an error occurs, a value of -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **kenv**() system call will fail if:

[EINVAL] The *action* argument is not a valid option, or the length of the *value* is less than 1 for a KENV_SET.

[ENOENT] No value could be found for *name* for a KENV_GET or KENV_UNSET.

[EPERM] A user other than the superuser attempted to set or unset a kernel environment variable.

[EFAULT] A bad address was encountered while attempting to copy in user arguments or copy out value(s).

[ENAMETOOLONG] The *name* or the *value* is longer than KENV_MNAMELEN or KENV_MVALLEN characters, respectively, or *len* did not include the NUL terminator for a KENV_SET.

SEE ALSO

kenv(1)

AUTHORS

This manual page was written by Chad David <*davidc@FreeBSD.org*>.

The **kenv**() system call was written by Maxime Henrion <*mux@FreeBSD.org*>.

NAME

kqueue, **kevent** - kernel event notification mechanism

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/event.h>

int
kqueue(*void*);

int
kevent(*int kq*, *const struct kevent *changelist*, *int nchanges*, *struct kevent *eventlist*, *int nevents*, *const struct timespec *timeout*);

EV_SET(*kev*, *ident*, *filter*, *flags*, *fflags*, *data*, *udata*);

DESCRIPTION

The **kqueue**() system call provides a generic method of notifying the user when an event happens or a condition holds, based on the results of small pieces of kernel code termed filters. A kevent is identified by the (ident, filter) pair; there may only be one unique kevent per kqueue.

The filter is executed upon the initial registration of a kevent in order to detect whether a preexisting condition is present, and is also executed whenever an event is passed to the filter for evaluation. If the filter determines that the condition should be reported, then the kevent is placed on the kqueue for the user to retrieve.

The filter is also run when the user attempts to retrieve the kevent from the kqueue. If the filter indicates that the condition that triggered the event no longer holds, the kevent is removed from the kqueue and is not returned.

Multiple events which trigger the filter do not result in multiple kevents being placed on the kqueue; instead, the filter will aggregate the events into a single struct kevent. Calling **close**() on a file descriptor will remove any kevents that reference the descriptor.

The **kqueue**() system call creates a new kernel event queue and returns a descriptor. The queue is not inherited by a child created with fork(2). However, if rfork(2) is called without the RFFDG flag, then the descriptor table is shared, which will allow sharing of the kqueue between two processes.

The **kevent**() system call is used to register events with the queue, and return any pending events to the user. The *changelist* argument is a pointer to an array of *kevent* structures, as defined in <*sys/event.h*>. All changes contained in the *changelist* are applied before any pending events are read from the queue. The *nchanges* argument gives the size of *changelist*. The *eventlist* argument is a pointer to an array of kevent structures. The *nevents* argument determines the size of *eventlist*. When *nevents* is zero, **kevent**() will return immediately even if there is a *timeout* specified unlike select(2). If *timeout* is a non-NULL pointer, it specifies a maximum interval to wait for an event, which will be interpreted as a struct timespec. If *timeout* is a NULL pointer, **kevent**() waits indefinitely. To effect a poll, the *timeout* argument should be non-NULL, pointing to a zero-valued *timespec* structure. The same array may be used for the *changelist* and *eventlist*.

The **EV_SET**() macro is provided for ease of initializing a kevent structure.

The *kevent* structure is defined as:

```
struct kevent {
        uintptr_t  ident;      /* identifier for this event */
        short      filter;     /* filter for event */
        u_short    flags;      /* action flags for kqueue */
        u_int      fflags;     /* filter flag value */
        int64_t    data;                /* filter data value */
        void       *udata;     /* opaque user data identifier */
        uint64_t   ext[4];     /* extensions */
};
```

The fields of *struct kevent* are:

ident Value used to identify this event. The exact interpretation is determined by the attached filter, but often is a file descriptor.

filter Identifies the kernel filter used to process this event. The pre-defined system filters are described below.

flags Actions to perform on the event.

fflags Filter-specific flags.

data Filter-specific data value.

udata Opaque user-defined value passed through the kernel unchanged.

ext Extended data passed to and from kernel. The *ext[0]* and *ext[1]* members use is defined by the filter. If the filter does not use them, the members are copied unchanged. The *ext[2]* and *ext[3]* members are always passed through the kernel as-is, making additional context available to application.

The *flags* field can contain the following values:

EV_ADD Adds the event to the kqueue. Re-adding an existing event will modify the parameters of the original event, and not result in a duplicate entry. Adding an event automatically enables it, unless overridden by the EV_DISABLE flag.

EV_ENABLE Permit **kevent**() to return the event if it is triggered.

EV_DISABLE Disable the event so **kevent**() will not return it. The filter itself is not disabled.

EV_DISPATCH Disable the event source immediately after delivery of an event. See EV_DISABLE above.

EV_DELETE Removes the event from the kqueue. Events which are attached to file descriptors are automatically deleted on the last close of the descriptor.

EV_RECEIPT This flag is useful for making bulk changes to a kqueue without draining any pending events. When passed as input, it forces EV_ERROR to always be returned. When a filter is successfully added the *data* field will be zero.

EV_ONESHOT Causes the event to return only the first occurrence of the filter being triggered. After the user retrieves the event from the kqueue, it is deleted.

EV_CLEAR After the event is retrieved by the user, its state is reset. This is useful for filters which report state transitions instead of the current state. Note that some filters may automatically set this flag internally.

EV_EOF Filters may set this flag to indicate filter-specific EOF condition.

EV_ERROR See *RETURN VALUES* below.

The predefined system filters are listed below. Arguments may be passed to and from the filter via the *fflags* and *data* fields in the kevent structure.

EVFILT_READ Takes a descriptor as the identifier, and returns whenever there is data

available to read. The behavior of the filter is slightly different depending on the descriptor type.

Sockets
: Sockets which have previously been passed to **listen**() return when there is an incoming connection pending. *data* contains the size of the listen backlog.

  Other socket descriptors return when there is data to be read, subject to the SO_RCVLOWAT value of the socket buffer. This may be overridden with a per-filter low water mark at the time the filter is added by setting the NOTE_LOWAT flag in *fflags*, and specifying the new low water mark in *data*. On return, *data* contains the number of bytes of protocol data available to read.

  If the read direction of the socket has shutdown, then the filter also sets EV_EOF in *flags*, and returns the socket error (if any) in *fflags*. It is possible for EOF to be returned (indicating the connection is gone) while there is still data pending in the socket buffer.

Vnodes
: Returns when the file pointer is not at the end of file. *data* contains the offset from current position to end of file, and may be negative.

  This behavior is different from poll(2), where read events are triggered for regular files unconditionally. This event can be triggered unconditionally by setting the NOTE_FILE_POLL flag in *fflags*.

Fifos, Pipes
: Returns when the there is data to read; *data* contains the number of bytes available.

  When the last writer disconnects, the filter will set EV_EOF in *flags*. This may be cleared by passing in EV_CLEAR, at which point the filter will resume waiting for data to become available before returning.

BPF devices
: Returns when the BPF buffer is full, the BPF timeout has expired, or when the BPF has "immediate mode" enabled and there is any data to read; *data* contains the number of bytes available.

EVFILT_WRITE Takes a descriptor as the identifier, and returns whenever it is possible to write to the descriptor. For sockets, pipes and fifos, *data* will contain the amount of space remaining in the write buffer. The filter will set EV_EOF when the reader disconnects, and for the fifo case, this may be cleared by use of EV_CLEAR. Note that this filter is not supported for vnodes or BPF devices.

For sockets, the low water mark and socket error handling is identical to the EVFILT_READ case.

EVFILT_EMPTY Takes a descriptor as the identifier, and returns whenever there is no remaining data in the write buffer.

EVFILT_AIO Events for this filter are not registered with **kevent**() directly but are registered via the *aio_sigevent* member of an asynchronous I/O request when it is scheduled via an asynchronous I/O system call such as **aio_read**(). The filter returns under the same conditions as **aio_error**(). For more details on this filter see sigevent(3) and aio(4).

EVFILT_VNODE Takes a file descriptor as the identifier and the events to watch for in *fflags*, and returns when one or more of the requested events occurs on the descriptor. The events to monitor are:

NOTE_ATTRIB The file referenced by the descriptor had its attributes changed.

NOTE_CLOSE A file descriptor referencing the monitored file, was closed. The closed file descriptor did not have write access.

NOTE_CLOSE_WRITE A file descriptor referencing the monitored file, was closed. The closed file descriptor had write access.

This note, as well as NOTE_CLOSE, are not activated when files are closed forcibly by unmount(2) or revoke(2). Instead, NOTE_REVOKE is sent for such events.

NOTE_DELETE The **unlink**() system call was called on the file

referenced by the descriptor.

NOTE_EXTEND — For regular file, the file referenced by the descriptor was extended.

For directory, reports that a directory entry was added or removed, as the result of rename operation. The NOTE_EXTEND event is not reported when a name is changed inside the directory.

NOTE_LINK — The link count on the file changed. In particular, the NOTE_LINK event is reported if a subdirectory was created or deleted inside the directory referenced by the descriptor.

NOTE_OPEN — The file referenced by the descriptor was opened.

NOTE_READ — A read occurred on the file referenced by the descriptor.

NOTE_RENAME — The file referenced by the descriptor was renamed.

NOTE_REVOKE — Access to the file was revoked via revoke(2) or the underlying file system was unmounted.

NOTE_WRITE — A write occurred on the file referenced by the descriptor.

On return, *fflags* contains the events which triggered the filter.

EVFILT_PROC — Takes the process ID to monitor as the identifier and the events to watch for in *fflags*, and returns when the process performs one or more of the requested events. If a process can normally see another process, it can attach an event to it. The events to monitor are:

NOTE_EXIT — The process has exited. The exit status will be stored in *data*.

NOTE_FORK The process has called **fork**().

NOTE_EXEC The process has executed a new process via execve(2) or a similar call.

NOTE_TRACK Follow a process across **fork**() calls. The parent process registers a new kevent to monitor the child process using the same *fflags* as the original event. The child process will signal an event with NOTE_CHILD set in *fflags* and the parent PID in *data*.

If the parent process fails to register a new kevent (usually due to resource limitations), it will signal an event with NOTE_TRACKERR set in *fflags*, and the child process will not signal a NOTE_CHILD event.

On return, *fflags* contains the events which triggered the filter.

EVFILT_PROCDESC Takes the process descriptor created by pdfork(2) to monitor as the identifier and the events to watch for in *fflags*, and returns when the associated process performs one or more of the requested events. The events to monitor are:

NOTE_EXIT The process has exited. The exit status will be stored in *data*.

On return, *fflags* contains the events which triggered the filter.

EVFILT_SIGNAL Takes the signal number to monitor as the identifier and returns when the given signal is delivered to the process. This coexists with the **signal**() and **sigaction**() facilities, and has a lower precedence. The filter will record all attempts to deliver a signal to a process, even if the signal has been marked as SIG_IGN, except for the SIGCHLD signal, which, if ignored, will not be recorded by the filter. Event notification happens after normal signal delivery processing. *data* returns the number of times the signal has occurred since the last call to **kevent**(). This filter automatically sets the EV_CLEAR flag internally.

EVFILT_TIMER Establishes an arbitrary timer identified by *ident*. When adding a timer, *data*

specifies the moment to fire the timer (for NOTE_ABSTIME) or the timeout period. The timer will be periodic unless EV_ONESHOT or NOTE_ABSTIME is specified. On return, *data* contains the number of times the timeout has expired since the last call to **kevent**(). For non-monotonic timers, this filter automatically sets the EV_CLEAR flag internally.

The filter accepts the following flags in the *fflags* argument:

NOTE_SECONDS	*data* is in seconds.
NOTE_MSECONDS	*data* is in milliseconds.
NOTE_USECONDS	*data* is in microseconds.
NOTE_NSECONDS	*data* is in nanoseconds.
NOTE_ABSTIME	The specified expiration time is absolute.

If *fflags* is not set, the default is milliseconds. On return, *fflags* contains the events which triggered the filter.

If an existing timer is re-added, the existing timer will be effectively canceled (throwing away any undelivered record of previous timer expiration) and re-started using the new parameters contained in *data* and *fflags*.

There is a system wide limit on the number of timers which is controlled by the *kern.kq_calloutmax* sysctl.

EVFILT_USER Establishes a user event identified by *ident* which is not associated with any kernel mechanism but is triggered by user level code. The lower 24 bits of the *fflags* may be used for user defined flags and manipulated using the following:

NOTE_FFNOP	Ignore the input *fflags*.
NOTE_FFAND	Bitwise AND *fflags*.
NOTE_FFOR	Bitwise OR *fflags*.

NOTE_FFCOPY Copy *fflags*.

NOTE_FFCTRLMASK Control mask for *fflags*.

NOTE_FFLAGSMASK User defined flag mask for *fflags*.

A user event is triggered for output with the following:

NOTE_TRIGGER Cause the event to be triggered.

On return, *fflags* contains the users defined flags in the lower 24 bits.

CANCELLATION BEHAVIOUR

If *nevents* is non-zero, i.e., the function is potentially blocking, the call is a cancellation point. Otherwise, i.e., if *nevents* is zero, the call is not cancellable. Cancellation can only occur before any changes are made to the kqueue, or when the call was blocked and no changes to the queue were requested.

RETURN VALUES

The **kqueue**() system call creates a new kernel event queue and returns a file descriptor. If there was an error creating the kernel event queue, a value of -1 is returned and errno set.

The **kevent**() system call returns the number of events placed in the *eventlist*, up to the value given by *nevents*. If an error occurs while processing an element of the *changelist* and there is enough room in the *eventlist*, then the event will be placed in the *eventlist* with EV_ERROR set in *flags* and the system error in *data*. Otherwise, -1 will be returned, and errno will be set to indicate the error condition. If the time limit expires, then **kevent**() returns 0.

EXAMPLES

```
#include <sys/event.h>
#include <err.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char **argv)
{
   struct kevent event;    /* Event we want to monitor */
```

```
    struct kevent tevent;   /* Event triggered */
    int kq, fd, ret;

    if (argc != 2)
            err(EXIT_FAILURE, "Usage: %s path\n", argv[0]);
    fd = open(argv[1], O_RDONLY);
    if (fd == -1)
            err(EXIT_FAILURE, "Failed to open '%s'", argv[1]);

    /* Create kqueue. */
    kq = kqueue();
    if (kq == -1)
            err(EXIT_FAILURE, "kqueue() failed");

    /* Initialize kevent structure. */
    EV_SET(&event, fd, EVFILT_VNODE, EV_ADD | EV_CLEAR, NOTE_WRITE,
            0, NULL);
    /* Attach event to the kqueue. */
    ret = kevent(kq, &event, 1, NULL, 0, NULL);
    if (ret == -1)
            err(EXIT_FAILURE, "kevent register");
    if (event.flags & EV_ERROR)
            errx(EXIT_FAILURE, "Event error: %s", strerror(event.data));

    for (;;) {
            /* Sleep until something happens. */
            ret = kevent(kq, NULL, 0, &tevent, 1, NULL);
            if (ret == -1) {
               err(EXIT_FAILURE, "kevent wait");
            } else if (ret > 0) {
               printf("Something was written in '%s'\n", argv[1]);
            }
    }
}
```

ERRORS

The **kqueue**() system call fails if:

[ENOMEM] The kernel failed to allocate enough memory for the kernel queue.

[ENOMEM] The RLIMIT_KQUEUES rlimit (see getrlimit(2)) for the current user would be exceeded.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

The **kevent**() system call fails if:

[EACCES] The process does not have permission to register a filter.

[EFAULT] There was an error reading or writing the *kevent* structure.

[EBADF] The specified descriptor is invalid.

[EINTR] A signal was delivered before the timeout expired and before any events were placed on the kqueue for return.

[EINTR] A cancellation request was delivered to the thread, but not yet handled.

[EINVAL] The specified time limit or filter is invalid.

[ENOENT] The event could not be found to be modified or deleted.

[ENOMEM] No memory was available to register the event or, in the special case of a timer, the maximum number of timers has been exceeded. This maximum is configurable via the *kern.kq_calloutmax* sysctl.

[ESRCH] The specified process to attach to does not exist.

When **kevent**() call fails with EINTR error, all changes in the *changelist* have been applied.

SEE ALSO

aio_error(2), aio_read(2), aio_return(2), poll(2), read(2), select(2), sigaction(2), write(2), pthread_setcancelstate(3), signal(3)

HISTORY

The **kqueue**() and **kevent**() system calls first appeared in FreeBSD 4.1.

AUTHORS

The **kqueue**() system and this manual page were written by Jonathan Lemon <*jlemon@FreeBSD.org*>.

BUGS

The *timeout* value is limited to 24 hours; longer timeouts will be silently reinterpreted as 24 hours.

In versions older than FreeBSD 12.0, <*sys/event.h*> failed to parse without including <*sys/types.h*> manually.

NAME

kill - send signal to a process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <signal.h>

int
kill(*pid_t pid*, *int sig*);

DESCRIPTION

The **kill**() system call sends the signal given by *sig* to *pid*, a process or a group of processes. The *sig* argument may be one of the signals specified in sigaction(2) or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

For a process to have permission to send a signal to a process designated by *pid*, the user must be the super-user, or the real or saved user ID of the receiving process must match the real or effective user ID of the sending process. A single exception is the signal SIGCONT, which may always be sent to any process with the same session ID as the sender. In addition, if the *security.bsd.conservative_signals* sysctl(9) is set to 1, the user is not a super-user, and the receiver is set-uid, then only job control and terminal control signals may be sent (in particular, only SIGKILL, SIGINT, SIGTERM, SIGALRM, SIGSTOP, SIGTTIN, SIGTTOU, SIGTSTP, SIGHUP, SIGUSR1, SIGUSR2).

If *pid* is greater than zero:

The *sig* signal is sent to the process whose ID is equal to *pid*.

If *pid* is zero:

The *sig* signal is sent to all processes whose group ID is equal to the process group ID of the sender, and for which the process has permission; this is a variant of killpg(2).

If *pid* is -1:

If the user has super-user privileges, the signal is sent to all processes excluding system processes (with P_SYSTEM flag set), process with ID 1 (usually init(8)), and the process sending the signal. If the user is not the super user, the signal is sent to all processes with the same uid as the user excluding the process sending the signal. No error is returned if any process could be signaled.

For compatibility with System V, if the process number is negative but not -1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number. This is a variant of killpg(2).

RETURN VALUES

The **kill**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **kill**() system call will fail and no signal will be sent if:

[EINVAL] The *sig* argument is not a valid signal number.

[ESRCH] No process or process group can be found corresponding to that specified by *pid*.

[EPERM] The sending process does not have permission to send *sig* to the receiving process.

SEE ALSO

getpgrp(2), getpid(2), killpg(2), sigaction(2), sigqueue(2), raise(3), init(8)

STANDARDS

The **kill**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

A version of the **kill**() function appeared in Version 3 AT&T UNIX. The signal number was added to the **kill**() function in Version 4 AT&T UNIX.

NAME

killpg - send signal to a process group

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <signal.h>

int
killpg(*pid_t pgrp*, *int sig*);

DESCRIPTION

The **killpg**() function sends the signal *sig* to the process group *pgrp*. See sigaction(2) for a list of signals. If *pgrp* is 0, **killpg**() sends the signal to the sending process's process group.

The sending process must be able to **kill**() at least one process in the receiving process group.

RETURN VALUES

The **killpg**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **killpg**() function will fail and no signal will be sent if:

[EINVAL] The *sig* argument is not a valid signal number.

[ESRCH] No process can be found in the process group specified by *pgrp*.

[EPERM] **kill**() returns EPERM for all processes in the process group.

SEE ALSO

getpgrp(2), kill(2), sigaction(2)

HISTORY

The **killpg**() function appeared in 4.0BSD.

NAME

kldfind - returns the fileid of a kld file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/linker.h>

int
kldfind(*const char *file*);

DESCRIPTION

The **kldfind**() system call returns the fileid of the kld file referenced by *file*.

RETURN VALUES

The **kldfind**() system call returns the fileid of the kld file referenced by *file*. Upon error, **kldfind**() returns -1 and sets *errno* to indicate the error.

ERRORS

errno is set to the following if **kldfind**() fails:

[EFAULT] The data required for this operation could not be read from the kernel space.

[ENOENT] The file specified is not loaded in the kernel.

SEE ALSO

kldfirstmod(2), kldload(2), kldnext(2), kldstat(2), kldsym(2), kldunload(2), modfind(2), modfnext(2), modnext(2), modstat(2), kld(4), kldstat(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

kldfirstmod - return first module id from the kld file specified

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/linker.h>

int
kldfirstmod(*int fileid*);

DESCRIPTION

The **kldfirstmod**() system call returns the module id pertaining to the first module referenced by *fileid*.

RETURN VALUES

The **kldfirstmod**() will return the id of the first module referenced by *fileid* or 0 if there are no references.

ERRORS

[ENOENT] The kld file referenced by *fileid* was not found.

SEE ALSO

kldfind(2), kldload(2), kldnext(2), kldstat(2), kldsym(2), kldunload(2), modfind(2), modfnext(2), modnext(2), modstat(2), kld(4), kldstat(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

kldload - load KLD files into the kernel

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/linker.h>

int
kldload(*const char *file*);

DESCRIPTION

The **kldload**() system call loads a kld file into the kernel using the kernel linker.

RETURN VALUES

The **kldload**() system call returns the fileid of the kld file which was loaded into the kernel. If an error occurs, **kldload**() will return -1 and set *errno* to indicate the error.

ERRORS

The named file is loaded unless:

[EPERM]	You do not have access to read the file or link it with the kernel. You should be the root user to be able to use the **kld** system calls.
[EFAULT]	Bad address encountered when adding kld info into the kernel space.
[ENOMEM]	There is no memory to load the file into the kernel.
[ENOENT]	The file was not found.
[ENOEXEC]	The file format of *file* was unrecognized.
[EEXIST]	The supplied *file* has already been loaded.

SEE ALSO

kldfind(2), kldfirstmod(2), kldnext(2), kldstat(2), kldsym(2), kldunload(2), modfind(2), modfnext(2), modnext(2), modstat(2), kld(4), kldload(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

kldnext - return the fileid of the next kld file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/linker.h>

int
kldnext(*int fileid*);

DESCRIPTION

The **kldnext**() system call returns the fileid of the next kld file (that is, the one after *fileid*) or 0 if *fileid* is the last file loaded. To get the fileid of the first kld file, pass *fileid* of 0 to **kldnext**().

RETURN VALUES

The **kldnext**() system call returns the fileid of the next kld file or 0 if successful. Otherwise **kldnext**() returns the value -1 and sets the global variable *errno* to indicate the error.

ERRORS

The only error set by **kldnext**() is ENOENT, which is set when *fileid* refers to a kld file that does not exist (is not loaded).

SEE ALSO

kldfind(2), kldfirstmod(2), kldload(2), kldstat(2), kldsym(2), kldunload(2), modfind(2), modfnext(2), modnext(2), modstat(2), kld(4), kldstat(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

kldstat - get status of kld file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/linker.h>

int
kldstat(*int fileid*, *struct kld_file_stat *stat*);

DESCRIPTION

The **kldstat**() system call writes the info for the file referred to by *fileid* into *stat*.

```
struct kld_file_stat {
        int         version;    /* set to sizeof(struct kld_file_stat) */
        char        name[MAXPATHLEN];
        int         refs;
        int         id;
        caddr_t     address;
        size_t      size;
        char        pathname[MAXPATHLEN];
};
```

version — This field is set to the size of the structure mentioned above by the code calling **kldstat**(), and not **kldstat**() itself.

name — The name of the file referred to by *fileid*.

refs — The number of modules referenced by *fileid*.

id — The id of the file specified in *fileid*.

address — The load address of the kld file.

size — The amount of memory in bytes allocated by the file.

pathname — The full name of the file referred to by *fileid*, including the path.

RETURN VALUES

The **kldstat**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The information for the file referred to by *fileid* is filled into the structure pointed to by *stat* unless:

[ENOENT] The file was not found (probably not loaded).

[EINVAL] The version specified in the *version* field of stat is not the proper version. You would need to rebuild world, the kernel, or your application, if this error occurs, given that you did properly fill in the *version* field.

[EFAULT] There was a problem copying one, some, or all of the fields into *stat* in the copyout(9) function.

SEE ALSO

kldfind(2), kldfirstmod(2), kldload(2), kldnext(2), kldsym(2), kldunload(2), modfind(2), modfnext(2), modnext(2), modstat(2), kld(4), kldstat(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

BUGS

The pathname may not be accurate if the file system mounts have changed since the module was loaded, or if this function is called within a chrooted environment.

NAME

kldsym - look up address by symbol name in a KLD

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/linker.h>

int
kldsym(*int fileid*, *int cmd*, *void *data*);

DESCRIPTION

The **kldsym**() system call returns the address of the symbol specified in *data* in the module specified by *fileid*. If *fileid* is 0, all loaded modules are searched. Currently, the only *cmd* implemented is KLDSYM_LOOKUP.

The *data* argument is of the following structure:

```
struct kld_sym_lookup {
    int         version;        /* sizeof(struct kld_sym_lookup) */
    char        *symname;       /* Symbol name we are looking up */
    u_long      symvalue;
    size_t      symsize;
};
```

The *version* member is to be set by the code calling **kldsym**() to **sizeof**(*struct kld_sym_lookup*). The next two members, *version* and *symname*, are specified by the user. The last two, *symvalue* and *symsize*, are filled in by **kldsym**() and contain the address associated with *symname* and the size of the data it points to, respectively.

RETURN VALUES

The **kldsym**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **kldsym**() system call will fail if:

[EINVAL] Invalid value in *data->version* or *cmd*.

[ENOENT] The *fileid* argument is invalid, or the specified symbol could not be found.

SEE ALSO

kldfind(2), kldfirstmod(2), kldload(2), kldnext(2), kldunload(2), modfind(2), modnext(2), modstat(2), kld(4)

HISTORY

The **kldsym**() system call first appeared in FreeBSD 3.0.

NAME

kldunload, **kldunloadf** - unload kld files

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/linker.h>

int
kldunload(*int fileid*);

int
kldunloadf(*int fileid*, *int flags*);

DESCRIPTION

The **kldunload**() system call unloads a kld file from the kernel that was previously linked via kldload(2).

The **kldunloadf**() system call accepts an additional flags argument, which may be one of LINKER_UNLOAD_NORMAL, giving the same behavior as **kldunload**(), or LINKER_UNLOAD_FORCE, which causes the unload to ignore a failure to quiesce the module.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The file referred to by *fileid* is unloaded unless:

[EPERM] You do not have access to unlink the file from the kernel.

[ENOENT] The file was not found.

[EBUSY] You attempted to unload a file linked by the kernel.

[EINVAL] The **kldunloadf**() system call was passed invalid flags.

SEE ALSO

kldfind(2), kldfirstmod(2), kldload(2), kldnext(2), kldstat(2), kldsym(2), modfind(2), modfnext(2),

modnext(2), modstat(2), kld(4), kldunload(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

ktrace - process tracing

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/time.h>
#include <sys/uio.h>
#include <sys/ktrace.h>

int
ktrace(*const char *tracefile*, *int ops*, *int trpoints*, *int pid*);

DESCRIPTION

The **ktrace**() system call enables or disables tracing of one or more processes. Users may only trace their own processes. Only the super-user can trace setuid or setgid programs.

The *tracefile* argument gives the pathname of the file to be used for tracing. The file must exist and be a regular file writable by the calling process. All trace records are always appended to the file, so the file must be truncated to zero length to discard previous trace data. If tracing points are being disabled (see KTROP_CLEAR below), *tracefile* may be NULL.

The *ops* argument specifies the requested ktrace operation. The defined operations are:

KTROP_SET	Enable trace points specified in *trpoints*.
KTROP_CLEAR	Disable trace points specified in *trpoints*.
KTROP_CLEARFILE	Stop all tracing.
KTRFLAG_DESCEND	The tracing change should apply to the specified process and all its current children.

The *trpoints* argument specifies the trace points of interest. The defined trace points are:

KTRFAC_SYSCALL	Trace system calls.
KTRFAC_SYSRET	Trace return values from system calls.
KTRFAC_NAMEI	Trace name lookup operations.
KTRFAC_GENIO	Trace all I/O (note that this option can generate much output).
KTRFAC_PSIG	Trace posted signals.
KTRFAC_CSW	Trace context switch points.

KTRFAC_USER	Trace application-specific events.
KTRFAC_STRUCT	Trace certain data structures.
KTRFAC_SYSCTL	Trace sysctls.
KTRFAC_PROCCTOR	Trace process construction.
KTRFAC_PROCDTOR	Trace process destruction.
KTRFAC_CAPFAIL	Trace capability failures.
KTRFAC_FAULT	Trace page faults.
KTRFAC_FAULTEND	Trace the end of page faults.
KTRFAC_INHERIT	Inherit tracing to future children.

Each tracing event outputs a record composed of a generic header followed by a trace point specific structure. The generic header is:

```
struct ktr_header {
        int             ktr_len;        /* length of buf */
        short           ktr_type;       /* trace record type */
        pid_t           ktr_pid;        /* process id */
        char            ktr_comm[MAXCOMLEN+1];      /* command name */
        struct timeval  ktr_time;       /* timestamp */
        intptr_t    ktr_tid;            /* was ktr_buffer */
};
```

The *ktr_len* field specifies the length of the *ktr_type* data that follows this header. The *ktr_pid* and *ktr_comm* fields specify the process and command generating the record. The *ktr_time* field gives the time (with microsecond resolution) that the record was generated. The *ktr_tid* field holds a thread id.

The generic header is followed by *ktr_len* bytes of a *ktr_type* record. The type specific records are defined in the <*sys/ktrace.h*> include file.

SYSCTL TUNABLES

The following sysctl(8) tunables influence the behaviour of **ktrace**():

kern.ktrace.geniosize
: bounds the amount of data a traced I/O request will log to the trace file.

kern.ktrace.request_pool
: bounds the number of trace events being logged at a time.

Sysctl tunables that control process debuggability (as determined by p_candebug(9)) also affect the operation of **ktrace**().

RETURN VALUES

The **ktrace**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **ktrace**() system call will fail if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named tracefile does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENOSYS] The kernel was not compiled with **ktrace** support.

A thread may be unable to log one or more tracing events due to a temporary shortage of resources. This condition is remembered by the kernel, and the next tracing request that succeeds will have the flag KTR_DROP set in its *ktr_type* field.

SEE ALSO

kdump(1), ktrace(1), utrace(2), sysctl(8), p_candebug(9)

HISTORY

The **ktrace**() system call first appeared in 4.4BSD.

NAME

labs - return the absolute value of a long integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

long
labs(*long j*);

DESCRIPTION

The **labs**() function returns the absolute value of the long integer *j*.

SEE ALSO

abs(3), cabs(3), floor(3), imaxabs(3), llabs(3), math(3)

STANDARDS

The **labs**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

BUGS

The absolute value of the most negative integer remains negative.

NAME

ldiv - return quotient and remainder from division

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

ldiv_t
ldiv(*long num*, *long denom*);

DESCRIPTION

The **ldiv**() function computes the value *num*/*denom* and returns the quotient and remainder in a structure named *ldiv_t* that contains two *long* members named *quot* and *rem*.

SEE ALSO

div(3), imaxdiv(3), lldiv(3), math(3)

STANDARDS

The **ldiv**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

lsearch, **lfind** - linear search and append

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <search.h>

*void **
lsearch(*const void *key*, *void *base*, *size_t *nelp*, *size_t width*,
*int (*compar)(const void *, const void *)*);

*void **
lfind(*const void *key*, *const void *base*, *size_t *nelp*, *size_t width*,
*int (*compar)(const void *, const void *)*);

DESCRIPTION

The **lsearch**() and **lfind**() functions walk linearly through an array and compare each element with the one to be sought using a supplied comparison function.

The *key* argument points to an element that matches the one that is searched. The array's address in memory is denoted by the *base* argument. The width of one element (i.e., the size as returned by **sizeof**()) is passed as the *width* argument. The number of valid elements contained in the array (not the number of elements the array has space reserved for) is given in the integer pointed to by *nelp*. The *compar* argument points to a function which compares its two arguments and returns zero if they are matching, and non-zero otherwise.

If no matching element was found in the array, **lsearch**() copies *key* into the position after the last element and increments the integer pointed to by *nelp*.

RETURN VALUES

The **lsearch**() and **lfind**() functions return a pointer to the first element found. If no element was found, **lsearch**() returns a pointer to the newly added element, whereas **lfind**() returns NULL. Both functions return NULL if an error occurs.

EXAMPLES

```
#include <search.h>
#include <stdio.h>
#include <stdlib.h>
```

```
static int
element_compare(const void *p1, const void *p2)
{
        int left = *(const int *)p1;
        int right = *(const int *)p2;

        return (left - right);
}

int
main(int argc, char **argv)
{
        const int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        size_t element_size = sizeof(array[0]);
        size_t array_size = sizeof(array) / element_size;
        int key;
        void *element;

        printf("Enter a number: ");
        if (scanf("%d", &key) != 1) {
                printf("Bad input\n");
                return (EXIT_FAILURE);
        }

        element = lfind(&key, array, &array_size, element_size,
           element_compare);

        if (element != NULL)
                printf("Element found: %d\n", *(int *)element);
        else
                printf("Element not found\n");

        return (EXIT_SUCCESS);
}
```

SEE ALSO

bsearch(3), hsearch(3), tsearch(3)

STANDARDS

The **lsearch**() and **lfind**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **lsearch**() and **lfind**() functions appeared in 4.2BSD. In FreeBSD 5.0, they reappeared conforming to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

libcuse - Userland character device library

LIBRARY

Userland Character Device Library (libcuse, -lcuse)

SYNOPSIS

To load the required kernel module at boot time, place the following line in loader.conf(5):

```
cuse_load="YES"
```

#include <cuse.h>

DESCRIPTION

The **libcuse** library contains functions to create a character device in userspace. The **libcuse** library is thread safe.

LIBRARY INITIALISATION / DEINITIALISATION

int **cuse_init**(*void*) This function initialises **libcuse**. Must be called at the beginning of the program. This function returns 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes. If the cuse kernel module is not loaded, CUSE_ERR_NOT_LOADED is returned.

int **cuse_uninit**(*void*) Deinitialise **libcuse**. Can be called at the end of the application. This function returns 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes.

UNIT MANAGEMENT

int **cuse_alloc_unit_number**(*int* *) This function stores a uniq system unit number at the pointed integer loation. This function returns 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes.

int **cuse_alloc_unit_number_by_id**(*int* *, *int id*) This function stores a unique system unit number at the pointed integer loation. The returned unit number is uniq within the given ID. Valid ID values are defined by the cuse include file. See the **CUSE_ID_XXX**() macros for more information. This function returns 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes.

int **cuse_free_unit_number**(*int*) This function frees the given allocated system unit number. This function returns 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes.

int **cuse_free_unit_number_by_id**(*int unit*, *int id*) This function frees the given allocated system unit

number belonging to the given ID. If both the unit and id argument is -1, all allocated units will be freed. This function returns 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes.

LIBRARY USAGE

*void ** **cuse_vmalloc**(*int size*) This function allocates *size* bytes of memory. Only memory allocated by this function can be memory mapped by mmap(2). This function returns a valid data pointer on success or NULL on failure.

int **cuse_is_vmalloc_addr**(*void **) This function returns non-zero if the passed pointer points to a valid and non-freed allocation, as returned by **cuse_vmalloc**(). Else this function returns zero.

void **cuse_vmfree**(*void **) This function frees memory allocated by **cuse_vmalloc**(). Note that the cuse library will internally not free the memory until the **cuse_uninit**() function is called and that the number of unique allocations is limited.

unsigned long **cuse_vmoffset**(*void **) This function returns the mmap offset that the client must use to access the allocated memory.

*struct cuse_dev ** **cuse_dev_create**(*const struct cuse_methods *mtod*, *void *priv0*, *void *priv1*, *uid_t*, *gid_t*, *int permission*, *const char *fmt*, ...) This function creates a new character device according to the given parameters. This function returns a valid cuse_dev structure pointer on success or NULL on failure. The device name can only contain a-z, A-Z, 0-9, dot, / and underscore characters.

void **cuse_dev_destroy**(*struct cuse_dev **) This functions destroys a previously created character device.

*void ** **cuse_dev_get_priv0**(*struct cuse_dev **), *void ** **cuse_dev_get_priv1**(*struct cuse_dev **), *void* **cuse_dev_set_priv0**(*struct cuse_dev **, *void **), *void* **cuse_dev_set_priv1**(*struct cuse_dev **, *void **) These functions are used to set and get the private data of the given cuse device.

int **cuse_wait_and_process**(*void*) This function will block and do event processing. If parallel I/O is required multiple threads must be created looping on this function. This function returns 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes.

*void ** **cuse_dev_get_per_file_handle**(*struct cuse_dev **), *void* **cuse_dev_set_per_file_handle**(*struct cuse_dev **, *void **) These functions are used to set and get the per-file-open specific handle and should only be used inside the cuse file operation callbacks.

void **cuse_set_local**(*int*) This function instructs **cuse_copy_out**() and **cuse_copy_in**() that the user pointer is local, if the argument passed to it is non-zero. Else the user pointer is assumed to be at the

peer application. This function should only be used inside the cuse file operation callbacks. The value is reset to zero when the given file operation returns, and does not affect any other file operation callbacks.

int **cuse_get_local**(*void*) Returns the current local state. See **cuse_set_local**().

int **cuse_copy_out**(*const void *src*, *void *peer_dst*, *int len*), *int* **cuse_copy_in**(*const void *peer_src*, *void *dst*, *int len*) These functions are used to transfer data between the local application and the peer application. These functions must be used when operating on the data pointers passed to the **cm_read**(), **cm_write**(), and **cm_ioctl**() callback functions. These functions return 0 on success or a negative value on failure. See CUSE_ERR_XXX for known error codes.

int **cuse_got_peer_signal**(*void*) This function is used to check if a signal has been delivered to the peer application and should only be used inside the cuse file operation callbacks. This function returns 0 if a signal has been delivered to the caller. Else it returns a negative value. See CUSE_ERR_XXX for known error codes.

struct cuse_dev * **cuse_dev_get_current**(*int *pcmd*) This function is used to get the current cuse device pointer and the currently executing command, by CUSE_CMD_XXX value. The *pcmd* argument is allowed to be NULL. This function should only be used inside the cuse file operation callbacks. On success a valid cuse device pointer is returned. On failure NULL is returned.

void **cuse_poll_wakeup**(*void*) This function will wake up any file pollers.

LIBRARY LIMITATIONS

Transfer lengths for **read**(), **write**(), **cuse_copy_in**(), and **cuse_copy_out**() should not exceed what can fit into a 32-bit signed integer and is defined by the **CUSE_LENGTH_MAX**() macro. Transfer lengths for ioctls should not exceed what is defined by the **CUSE_BUFFER_MAX**() macro.

LIBRARY CALLBACK METHODS

In general fflags are defined by CUSE_FFLAG_XXX and errors are defined by CUSE_ERR_XXX.

```
enum {
  CUSE_ERR_NONE
  CUSE_ERR_BUSY
  CUSE_ERR_WOULDBLOCK
  CUSE_ERR_INVALID
  CUSE_ERR_NO_MEMORY
  CUSE_ERR_FAULT
  CUSE_ERR_SIGNAL
```

```
CUSE_ERR_OTHER
CUSE_ERR_NOT_LOADED
CUSE_ERR_NO_DEVICE

CUSE_POLL_NONE
CUSE_POLL_READ
CUSE_POLL_WRITE
CUSE_POLL_ERROR

CUSE_FFLAG_NONE
CUSE_FFLAG_READ
CUSE_FFLAG_WRITE
CUSE_FFLAG_NONBLOCK

CUSE_CMD_NONE
CUSE_CMD_OPEN
CUSE_CMD_CLOSE
CUSE_CMD_READ
CUSE_CMD_WRITE
CUSE_CMD_IOCTL
CUSE_CMD_POLL
CUSE_CMD_SIGNAL
CUSE_CMD_SYNC
CUSE_CMD_MAX
};
```

int **cuse_open_t**(*struct cuse_dev **, *int fflags*) This function returns a CUSE_ERR_XXX value.

int **cuse_close_t**(*struct cuse_dev **, *int fflags*) This function returns a CUSE_ERR_XXX value.

int **cuse_read_t**(*struct cuse_dev **, *int fflags*, *void *peer_ptr*, *int len*) This function returns a CUSE_ERR_XXX value in case of failure or the actually transferred length in case of success. **cuse_copy_in**() and **cuse_copy_out**() must be used to transfer data to and from the *peer_ptr*.

int **cuse_write_t**(*struct cuse_dev **, *int fflags*, *const void *peer_ptr*, *int len*) This function returns a CUSE_ERR_XXX value in case of failure or the actually transferred length in case of success. **cuse_copy_in**() and **cuse_copy_out**() must be used to transfer data to and from the *peer_ptr*.

int **cuse_ioctl_t**(*struct cuse_dev **, *int fflags*, *unsigned long cmd*, *void *peer_data*) This function returns a CUSE_ERR_XXX value in case of failure or zero in case of success. **cuse_copy_in**() and

cuse_copy_out() must be used to transfer data to and from the *peer_data*.

int **cuse_poll_t**(*struct cuse_dev **, *int fflags*, *int events*) This function returns a mask of CUSE_POLL_XXX values in case of failure and success. The events argument is also a mask of CUSE_POLL_XXX values.

```
struct cuse_methods {
  cuse_open_t *cm_open;
  cuse_close_t *cm_close;
  cuse_read_t *cm_read;
  cuse_write_t *cm_write;
  cuse_ioctl_t *cm_ioctl;
  cuse_poll_t *cm_poll;
};
```

HISTORY

libcuse was written by Hans Petter Selasky.

NAME

libthr - 1:1 POSIX threads library

LIBRARY

1:1 Threading Library (libthr, -lthr)

SYNOPSIS

#include <pthread.h>

DESCRIPTION

The **libthr** library provides a 1:1 implementation of the pthread(3) library interfaces for application threading. It has been optimized for use by applications expecting system scope thread semantics.

The library is tightly integrated with the run-time link editor ld-elf.so.1(1) and Standard C Library (libc, -lc); all three components must be built from the same source tree. Mixing libc and **libthr** libraries from different versions of FreeBSD is not supported. The run-time linker ld-elf.so.1(1) has some code to ensure backward-compatibility with older versions of **libthr**.

The man page documents the quirks and tunables of the **libthr**. When linking with -lpthread, the run-time dependency libthr.so.3 is recorded in the produced object.

MUTEX ACQUISITION

A locked mutex (see pthread_mutex_lock(3)) is represented by a volatile variable of type lwpid_t, which records the global system identifier of the thread owning the lock. **libthr** performs a contested mutex acquisition in three stages, each of which is more resource-consuming than the previous. The first two stages are only applied for a mutex of PTHREAD_MUTEX_ADAPTIVE_NP type and PTHREAD_PRIO_NONE protocol (see pthread_mutexattr(3)).

First, on SMP systems, a spin loop is performed, where the library attempts to acquire the lock by atomic(9) operations. The loop count is controlled by the LIBPTHREAD_SPINLOOPS environment variable, with a default value of 2000.

If the spin loop was unable to acquire the mutex, a yield loop is executed, performing the same atomic(9) acquisition attempts as the spin loop, but each attempt is followed by a yield of the CPU time of the thread using the sched_yield(2) syscall. By default, the yield loop is not executed. This is controlled by the LIBPTHREAD_YIELDLOOPS environment variable.

If both the spin and yield loops failed to acquire the lock, the thread is taken off the CPU and put to sleep in the kernel with the _umtx_op(2) syscall. The kernel wakes up a thread and hands the ownership of the lock to the woken thread when the lock becomes available.

THREAD STACKS

Each thread is provided with a private user-mode stack area used by the C runtime. The size of the main (initial) thread stack is set by the kernel, and is controlled by the RLIMIT_STACK process resource limit (see getrlimit(2)).

By default, the main thread's stack size is equal to the value of RLIMIT_STACK for the process. If the LIBPTHREAD_SPLITSTACK_MAIN environment variable is present in the process environment (its value does not matter), the main thread's stack is reduced to 4MB on 64bit architectures, and to 2MB on 32bit architectures, when the threading library is initialized. The rest of the address space area which has been reserved by the kernel for the initial process stack is used for non-initial thread stacks in this case. The presence of the LIBPTHREAD_BIGSTACK_MAIN environment variable overrides LIBPTHREAD_SPLITSTACK_MAIN; it is kept for backward-compatibility.

The size of stacks for threads created by the process at run-time with the pthread_create(3) call is controlled by thread attributes: see pthread_attr(3), in particular, the pthread_attr_setstacksize(3), pthread_attr_setguardsize(3) and pthread_attr_setstackaddr(3) functions. If no attributes for the thread stack size are specified, the default non-initial thread stack size is 2MB for 64bit architectures, and 1MB for 32bit architectures.

RUN-TIME SETTINGS

The following environment variables are recognized by **libthr** and adjust the operation of the library at run-time:

LIBPTHREAD_BIGSTACK_MAIN	Disables the reduction of the initial thread stack enabled by LIBPTHREAD_SPLITSTACK_MAIN.
LIBPTHREAD_SPLITSTACK_MAIN	Causes a reduction of the initial thread stack, as described in the section *THREAD STACKS*. This was the default behaviour of **libthr** before FreeBSD 11.0.
LIBPTHREAD_SPINLOOPS	The integer value of the variable overrides the default count of iterations in the spin loop of the mutex acquisition. The default count is 2000, set by the MUTEX_ADAPTIVE_SPINS constant in the **libthr** sources.
LIBPTHREAD_YIELDLOOPS	A non-zero integer value enables the yield loop in the process of the mutex acquisition. The value is the count of loop operations.
LIBPTHREAD_QUEUE_FIFO	The integer value of the variable specifies how often

blocked threads are inserted at the head of the sleep queue, instead of its tail. Bigger values reduce the frequency of the FIFO discipline. The value must be between 0 and 255.

The following sysctl MIBs affect the operation of the library:

kern.ipc.umtx_vnode_persistent — By default, a shared lock backed by a mapped file in memory is automatically destroyed on the last unmap of the corresponding file's page, which is allowed by POSIX. Setting the sysctl to 1 makes such a shared lock object persist until the vnode is recycled by the Virtual File System. Note that in case file is not opened and not mapped, the kernel might recycle it at any moment, making this sysctl less useful than it sounds.

kern.ipc.umtx_max_robust — The maximal number of robust mutexes allowed for one thread. The kernel will not unlock more mutexes than specified, see _umtx_op for more details. The default value is large enough for most useful applications.

debug.umtx.robust_faults_verbose — A non zero value makes kernel emit some diagnostic when the robust mutexes unlock was prematurely aborted after detecting some inconsistency, as a measure to prevent memory corruption.

The RLIMIT_UMTXP limit (see getrlimit(2)) defines how many shared locks a given user may create simultaneously.

INTERACTION WITH RUN-TIME LINKER

On load, **libthr** installs interposing handlers into the hooks exported by libc. The interposers provide real locking implementation instead of the stubs for single-threaded processes in libc, cancellation support and some modifications to the signal operations.

libthr cannot be unloaded; the dlclose(3) function does not perform any action when called with a handle for **libthr**. One of the reasons is that the internal interposing of libc functions cannot be undone.

SIGNALS

The implementation interposes the user-installed signal(3) handlers. This interposing is done to postpone signal delivery to threads which entered (libthr-internal) critical sections, where the calling of the user-provided signal handler is unsafe. An example of such a situation is owning the internal library lock. When a signal is delivered while the signal handler cannot be safely called, the call is postponed and performed until after the exit from the critical section. This should be taken into account when

interpreting ktrace(1) logs.

SEE ALSO

ktrace(1), ld-elf.so.1(1), getrlimit(2), errno(2), thr_exit(2), thr_kill(2), thr_kill2(2), thr_new(2), thr_self(2), thr_set_name(2), _umtx_op(2), dlclose(3), dlopen(3), getenv(3), pthread_attr(3), pthread_attr_setstacksize(3), pthread_create(3), signal(3), atomic(9)

AUTHORS

The **libthr** library was originally created by Jeff Roberson <*jeff@FreeBSD.org*>, and enhanced by Jonathan Mini <*mini@FreeBSD.org*> and Mike Makonnen <*mtm@FreeBSD.org*>. It has been substantially rewritten and optimized by David Xu <*davidxu@FreeBSD.org*>.

NAME

link_addr, **link_ntoa** - elementary address specification routines for link level access

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <net/if_dl.h>

void
link_addr(*const char *addr*, *struct sockaddr_dl *sdl*);

*char **
link_ntoa(*const struct sockaddr_dl *sdl*);

DESCRIPTION

The routine **link_addr**() interprets character strings representing link-level addresses, returning binary information suitable for use in system calls. The routine **link_ntoa**() takes a link-level address and returns an ASCII string representing some of the information present, including the link level address itself, and the interface name or number, if present. This facility is experimental and is still subject to change.

For **link_addr**(), the string *addr* may contain an optional network interface identifier of the form "name unit-number", suitable for the first argument to ifconfig(8), followed in all cases by a colon and an interface address in the form of groups of hexadecimal digits separated by periods. Each group represents a byte of address; address bytes are filled left to right from low order bytes through high order bytes.

Thus le0:8.0.9.13.d.30 represents an ethernet address to be transmitted on the first Lance ethernet interface.

RETURN VALUES

The **link_ntoa**() function always returns a null terminated string. The **link_addr**() function has no return value. (See *BUGS*.)

SEE ALSO

getnameinfo(3)

HISTORY

The **link_addr**() and **link_ntoa**() functions appeared in 4.3BSD-Reno.

BUGS

The returned values for link_ntoa reside in a static memory area.

The function **link_addr**() should diagnose improperly formed input, and there should be an unambiguous way to recognize this.

If the *sdl_len* field of the link socket address *sdl* is 0, **link_ntoa**() will not insert a colon before the interface address bytes. If this translated address is given to **link_addr**() without inserting an initial colon, the latter will not interpret it correctly.

NAME

link, **linkat** - make a hard file link

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
link(*const char *name1*, *const char *name2*);

int
linkat(*int fd1*, *const char *name1*, *int fd2*, *const char *name2*, *int flag*);

DESCRIPTION

The **link**() system call atomically creates the specified directory entry (hard link) *name2* with the attributes of the underlying object pointed at by *name1*. If the link is successful: the link count of the underlying object is incremented; *name1* and *name2* share equal access and rights to the underlying object.

If *name1* is removed, the file *name2* is not deleted and the link count of the underlying object is decremented.

The object pointed at by the *name1* argument must exist for the hard link to succeed and both *name1* and *name2* must be in the same file system. The *name1* argument may not be a directory.

The **linkat**() system call is equivalent to *link* except in the case where either *name1* or *name2* or both are relative paths. In this case a relative path *name1* is interpreted relative to the directory associated with the file descriptor *fd1* instead of the current working directory and similarly for *name2* and the file descriptor *fd2*.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

AT_SYMLINK_FOLLOW
If *name1* names a symbolic link, a new link for the target of the symbolic link is created.

If **linkat**() is passed the special value AT_FDCWD in the *fd1* or *fd2* parameter, the current working directory is used for the respective *name* argument. If both *fd1* and *fd2* have value AT_FDCWD, the

behavior is identical to a call to **link**(). Unless *flag* contains the AT_SYMLINK_FOLLOW flag, if *name1* names a symbolic link, a new link is created for the symbolic link *name1* and not its target.

RETURN VALUES

The **link**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **link**() system call will fail and no link will be created if:

[ENOTDIR] A component of either path prefix is not a directory.

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or entire length of either path name exceeded 1023 characters.

[ENOENT] A component of either path prefix does not exist.

[EOPNOTSUPP] The file system containing the file named by *name1* does not support links.

[EMLINK] The link count of the file named by *name1* would exceed 32767.

[EACCES] A component of either path prefix denies search permission.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission.

[ELOOP] Too many symbolic links were encountered in translating one of the pathnames.

[ENOENT] The file named by *name1* does not exist.

[EEXIST] The link named by *name2* does exist.

[EPERM] The file named by *name1* is a directory.

[EPERM] The file named by *name1* has its immutable or append-only flag set, see the chflags(2) manual page for more information.

[EPERM] The parent directory of the file named by *name2* has its immutable flag set.

[EXDEV] The link named by *name2* and the file named by *name1* are on different file systems.

[ENOSPC] The directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.

[EDQUOT] The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EIO] An I/O error occurred while reading from or writing to the file system to make the directory entry.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] One of the pathnames specified is outside the process's allocated address space.

In addition to the errors returned by the **link**(), the **linkat**() system call may fail if:

[EBADF] The *name1* or *name2* argument does not specify an absolute path and the *fd1* or *fd2* argument, respectively, is neither AT_FDCWD nor a valid file descriptor open for searching.

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *name1* or *name2* argument is not an absolute path and *fd1* or *fd2*, respectively, is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chflags(2), readlink(2), symlink(2), unlink(2)

STANDARDS

The **link**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **linkat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **link**() function appeared in Version 1 AT&T UNIX. The **linkat**() system call appeared in FreeBSD 8.0.

The **link**() system call traditionally allows the super-user to link directories which corrupts the file system coherency. This implementation no longer permits it.

NAME

lio_listio - list directed I/O (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <aio.h>

int
lio_listio(*int mode*, *struct aiocb * const list[]*, *int nent*, *struct sigevent *sig*);

DESCRIPTION

The **lio_listio**() function initiates a list of I/O requests with a single function call. The *list* argument is an array of pointers to *aiocb* structures describing each operation to perform, with *nent* elements. NULL elements are ignored.

The *aio_lio_opcode* field of each *aiocb* specifies the operation to be performed. The following operations are supported:

LIO_READ Read data as if by a call to aio_read(2).

LIO_NOP No operation.

LIO_WRITE Write data as if by a call to aio_write(2).

If the *mode* argument is LIO_WAIT, **lio_listio**() does not return until all the requested operations have been completed. If *mode* is LIO_NOWAIT, *sig* can be used to request asynchronous notification when all operations have completed. If *sig* is NULL, no notification is sent.

For SIGEV_KEVENT notifications, the posted kevent will contain:

Member	**Value**
ident	*list*
filter	EVFILT_LIO
udata	value stored in *sig->sigev_value*

For SIGEV_SIGNO and SIGEV_THREAD_ID notifications, the information for the queued signal will include SI_ASYNCIO in the *si_code* field and the value stored in *sig->sigev_value* in the *si_value* field.

For SIGEV_THREAD notifications, the value stored in *sig->sigev_value* is passed to the *sig->sigev_notify_function* as described in sigevent(3).

The order in which the requests are carried out is not specified; in particular, there is no guarantee that they will be executed in the order 0, 1, ..., *nent*-1.

RETURN VALUES

If *mode* is LIO_WAIT, the **lio_listio**() function returns 0 if the operations completed successfully, otherwise -1.

If *mode* is LIO_NOWAIT, the **lio_listio**() function returns 0 if the operations are successfully queued, otherwise -1.

ERRORS

The **lio_listio**() function will fail if:

[EAGAIN]	There are not enough resources to enqueue the requests.
[EAGAIN]	The request would cause the system-wide limit {AIO_MAX} to be exceeded.
[EINVAL]	The *mode* argument is neither LIO_WAIT nor LIO_NOWAIT, or *nent* is greater than {AIO_LISTIO_MAX}.
[EINVAL]	The asynchronous notification method in *sig->sigev_notify* is invalid or not supported.
[EINTR]	A signal interrupted the system call before it could be completed.
[EIO]	One or more requests failed.

In addition, the **lio_listio**() function may fail for any of the reasons listed for aio_read(2) and aio_write(2).

If **lio_listio**() succeeds, or fails with an error code of EAGAIN, EINTR, or EIO, some of the requests may have been initiated. The caller should check the error status of each *aiocb* structure individually by calling aio_error(2).

SEE ALSO

aio_error(2), aio_read(2), aio_write(2), read(2), write(2), sigevent(3), siginfo(3), aio(4)

STANDARDS

The **lio_listio**() function is expected to conform to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

listen - listen for connections on a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

int
listen(*int s*, *int backlog*);

DESCRIPTION

To accept connections, a socket is first created with socket(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**(), and then the connections are accepted with accept(2). The **listen**() system call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* argument defines the maximum length the queue of pending connections may grow to. The real maximum queue length will be 1.5 times more than the value specified in the *backlog* argument. A subsequent **listen**() system call on the listening socket allows the caller to change the maximum queue length using a new *backlog* argument. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, in the case of TCP, the connection will be silently dropped.

Current queue lengths of listening sockets can be queried using netstat(1) command.

Note that before FreeBSD 4.5 and the introduction of the syncache, the *backlog* argument also determined the length of the incomplete connection queue, which held TCP sockets in the process of completing TCP's 3-way handshake. These incomplete connections are now held entirely in the syncache, which is unaffected by queue lengths. Inflated *backlog* values to help handle denial of service attacks are no longer necessary.

The sysctl(3) MIB variable *kern.ipc.soacceptqueue* specifies a hard limit on *backlog*; if a value greater than *kern.ipc.soacceptqueue* or less than zero is specified, *backlog* is silently forced to *kern.ipc.soacceptqueue*.

INTERACTION WITH ACCEPT FILTERS

When accept filtering is used on a socket, a second queue will be used to hold sockets that have connected, but have not yet met their accept filtering criteria. Once the criteria has been met, these

sockets will be moved over into the completed connection queue to be accept(2)ed. If this secondary queue is full and a new connection comes in, the oldest socket which has not yet met its accept filter criteria will be terminated.

This secondary queue, like the primary listen queue, is sized according to the *backlog* argument.

RETURN VALUES

The **listen**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **listen**() system call will fail if:

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

[EDESTADDRREQ] The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

[EINVAL] The socket is already connected, or in the process of being connected.

[ENOTSOCK] The argument *s* is not a socket.

[EOPNOTSUPP] The socket is not of a type that supports the operation **listen**().

SEE ALSO

netstat(1), accept(2), connect(2), socket(2), sysctl(3), sysctl(8), accept_filter(9)

HISTORY

The **listen**() system call appeared in 4.2BSD. The ability to configure the maximum *backlog* at run-time, and to use a negative *backlog* to request the maximum allowable value, was introduced in FreeBSD 2.2. The *kern.ipc.somaxconn* sysctl(3) has been replaced with *kern.ipc.soacceptqueue* in FreeBSD 10.0 to prevent confusion about its actual functionality. The original sysctl(3) *kern.ipc.somaxconn* is still available but hidden from a sysctl(3) -a output so that existing applications and scripts continue to work.

NAME

llabs - returns absolute value

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

long long
llabs(*long long j*);

DESCRIPTION

The **llabs**() function returns the absolute value of *j*.

SEE ALSO

abs(3), fabs(3), hypot(3), imaxabs(3), labs(3), math(3)

STANDARDS

The **llabs**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The **llabs**() function first appeared in FreeBSD 5.0.

BUGS

The absolute value of the most negative integer remains negative.

NAME

lldiv - returns quotient and remainder

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

lldiv_t
lldiv(*long long numer*, *long long denom*);

DESCRIPTION

The **lldiv**() function computes the value of *numer* divided by *denom* and returns the stored result in the form of the *lldiv_t* type.

The *lldiv_t* type is defined as:

```
typedef struct {
        long long quot; /* Quotient. */
        long long rem;  /* Remainder. */
} lldiv_t;
```

SEE ALSO

div(3), imaxdiv(3), ldiv(3), math(3)

STANDARDS

The **lldiv**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The **lldiv**() function first appeared in FreeBSD 5.0.

NAME

localeconv - natural language formatting for C

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <locale.h>

*struct lconv **
localeconv(*void*);

#include <xlocale.h>

*struct lconv **
localeconv_l(*locale_t locale*);

DESCRIPTION

The **localeconv**() function returns a pointer to a structure which provides parameters for formatting numbers, especially currency values:

```
struct lconv {
        char    *decimal_point;
        char    *thousands_sep;
        char    *grouping;
        char    *int_curr_symbol;
        char    *currency_symbol;
        char    *mon_decimal_point;
        char    *mon_thousands_sep;
        char    *mon_grouping;
        char    *positive_sign;
        char    *negative_sign;
        char    int_frac_digits;
        char    frac_digits;
        char    p_cs_precedes;
        char    p_sep_by_space;
        char    n_cs_precedes;
        char    n_sep_by_space;
        char    p_sign_posn;
        char    n_sign_posn;
```

```
        char    int_p_cs_precedes;
        char    int_n_cs_precedes;
        char    int_p_sep_by_space;
        char    int_n_sep_by_space;
        char    int_p_sign_posn;
        char    int_n_sign_posn;
};
```

The individual fields have the following meanings:

decimal_point The decimal point character, except for currency values, cannot be an empty string.

thousands_sep The separator between groups of digits before the decimal point, except for currency values.

grouping The sizes of the groups of digits, except for currency values. This is a pointer to a vector of integers, each of size *char*, representing group size from low order digit groups to high order (right to left). The list may be terminated with 0 or CHAR_MAX. If the list is terminated with 0, the last group size before the 0 is repeated to account for all the digits. If the list is terminated with CHAR_MAX, no more grouping is performed.

int_curr_symbol The standardized international currency symbol.

currency_symbol The local currency symbol.

mon_decimal_point The decimal point character for currency values.

mon_thousands_sep
The separator for digit groups in currency values.

mon_grouping Like *grouping* but for currency values.

positive_sign The character used to denote nonnegative currency values, usually the empty string.

negative_sign The character used to denote negative currency values, usually a minus sign.

int_frac_digits The number of digits after the decimal point in an international-style currency value.

frac_digits The number of digits after the decimal point in the local style for currency values.

p_cs_precedes 1 if the currency symbol precedes the currency value for nonnegative values, 0 if it follows.

p_sep_by_space 1 if a space is inserted between the currency symbol and the currency value for nonnegative values, 0 otherwise.

n_cs_precedes Like *p_cs_precedes* but for negative values.

n_sep_by_space Like *p_sep_by_space* but for negative values.

p_sign_posn The location of the *positive_sign* with respect to a nonnegative quantity and the *currency_symbol*, coded as follows:

0 Parentheses around the entire string.
1 Before the string.
2 After the string.
3 Just before *currency_symbol*.
4 Just after *currency_symbol*.

n_sign_posn Like *p_sign_posn* but for negative currency values.

int_p_cs_precedes Same as *p_cs_precedes*, but for internationally formatted monetary quantities.

int_n_cs_precedes Same as *n_cs_precedes*, but for internationally formatted monetary quantities.

int_p_sep_by_space
Same as *p_sep_by_space*, but for internationally formatted monetary quantities.

int_n_sep_by_space
Same as *n_sep_by_space*, but for internationally formatted monetary quantities.

int_p_sign_posn Same as *p_sign_posn*, but for internationally formatted monetary quantities.

int_n_sign_posn Same as *n_sign_posn*, but for internationally formatted monetary quantities.

Unless mentioned above, an empty string as a value for a field indicates a zero length result or a value that is not in the current locale. A CHAR_MAX result similarly denotes an unavailable value.

The **localeconv_l**() function takes an explicit locale parameter. For more information, see xlocale(3).

RETURN VALUES

The **localeconv**() function returns a pointer to a static object which may be altered by later calls to setlocale(3) or **localeconv**(). The return value for **localeconv_l**() is stored with the locale. It will remain valid until a subsequent call to freelocale(3). If a thread-local locale is in effect then the return value from **localeconv**() will remain valid until the locale is destroyed.

ERRORS

No errors are defined.

SEE ALSO

setlocale(3), strfmon(3)

STANDARDS

The **localeconv**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The **localeconv**() function first appeared in 4.4BSD.

NAME

lockf - record locking on files

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
lockf(*int fd*, *int function*, *off_t size*);

DESCRIPTION

The **lockf**() function allows sections of a file to be locked with advisory-mode locks. Calls to **lockf**() from other processes which attempt to lock the locked file section will either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates.

The argument *fd* is an open file descriptor. The file descriptor must have been opened either for write-only (O_WRONLY) or read/write (O_RDWR) operation.

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are as follows:

Function	**Description**
F_ULOCK	unlock locked sections
F_LOCK	lock a section for exclusive use
F_TLOCK	test and lock a section for exclusive use
F_TEST	test a section for locks by other processes

F_ULOCK removes locks from a section of the file; F_LOCK and F_TLOCK both lock a section of a file if the section is available; F_TEST detects if a lock by another process is present on the specified section.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive size or backward for a negative size (the preceding bytes up to but not including the current offset). However, it is not permitted to lock a section that starts or extends before the beginning of the file. If *size* is 0, the section from the current offset through the largest possible file offset is locked (that is, from the current offset through the present or any future end-of-file).

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections are combined into a single locked section. If the request would cause the number of locks to exceed a system-imposed limit, the request will fail.

F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available. F_LOCK blocks the calling process until the section is available. F_TLOCK makes the function fail if the section is already locked by another process.

File locks are released on first close by the locking process of any file descriptor for the file.

F_ULOCK requests release (wholly or in part) one or more locked sections controlled by the process. Locked sections will be unlocked starting at the current file offset through *size* bytes or to the end of file if size is 0. When all of a locked section is not released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section are still locked by the process. Releasing the center portion of a locked section will cause the remaining locked beginning and end portions to become two separate locked sections. If the request would cause the number of locks in the system to exceed a system-imposed limit, the request will fail.

An F_ULOCK request in which size is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type off_t, when the process has an existing lock in which size is 0 and which includes the last byte of the requested section, will be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise an F_ULOCK request will attempt to unlock only the requested section.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. This implementation detects that sleeping until a locked region is unlocked would cause a deadlock and fails with an EDEADLK error.

The **lockf**(), fcntl(2), and flock(2) locks are compatible. Processes using different locking interfaces can cooperate over the same file safely. However, only one of such interfaces should be used within the same process. If a file is locked by a process through flock(2), any record within the file will be seen as locked from the viewpoint of another process using fcntl(2) or **lockf**(), and vice versa.

Blocking on a section is interrupted by any signal.

RETURN VALUES

The **lockf**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error. In the case of a failure, existing locks are not changed.

ERRORS

The **lockf**() function will fail if:

[EAGAIN] The argument *function* is F_TLOCK or F_TEST and the section is already locked by another process.

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

The argument *function* is F_LOCK or F_TLOCK, and *fd* is not a valid file descriptor open for writing.

[EDEADLK] The argument *function* is F_LOCK and a deadlock is detected.

[EINTR] The argument *function* is F_LOCK and **lockf**() was interrupted by the delivery of a signal.

[EINVAL] The argument *function* is not one of F_ULOCK, F_LOCK, F_TLOCK or F_TEST.

The argument *fd* refers to a file that does not support locking.

[ENOLCK] The argument *function* is F_ULOCK, F_LOCK or F_TLOCK, and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

SEE ALSO

fcntl(2), flock(2)

STANDARDS

The **lockf**() function conforms to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

NAME

lseek - reposition read/write file offset

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

off_t
lseek(*int fildes*, *off_t offset*, *int whence*);

DESCRIPTION

The **lseek**() system call repositions the offset of the file descriptor *fildes* to the argument *offset* according to the directive *whence*. The argument *fildes* must be an open file descriptor. The **lseek**() system call repositions the file position pointer associated with the file descriptor *fildes* as follows:

If *whence* is SEEK_SET, the offset is set to *offset* bytes.

If *whence* is SEEK_CUR, the offset is set to its current location plus *offset* bytes.

If *whence* is SEEK_END, the offset is set to the size of the file plus *offset* bytes.

If *whence* is SEEK_HOLE, the offset is set to the start of the next hole greater than or equal to the supplied *offset*. The definition of a hole is provided below.

If *whence* is SEEK_DATA, the offset is set to the start of the next non-hole file region greater than or equal to the supplied *offset*.

The **lseek**() system call allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

Some devices are incapable of seeking. The value of the pointer associated with such a device is undefined.

A "hole" is defined as a contiguous range of bytes in a file, all having the value of zero, but not all zeros in a file are guaranteed to be represented as holes returned with SEEK_HOLE. File systems are allowed to expose ranges of zeros with SEEK_HOLE, but not required to. Applications can use SEEK_HOLE to optimise their behavior for ranges of zeros, but must not depend on it to find all such ranges in a file.

Each file is presented as having a zero-size virtual hole at the very end of the file. The existence of a hole at the end of every data region allows for easy programming and also provides compatibility to the original implementation in Solaris. It also causes the current file size (i.e., end-of-file offset) to be returned to indicate that there are no more holes past the supplied *offset*. Applications should use **fpathconf**(*_PC_MIN_HOLE_SIZE*) or **pathconf**(*_PC_MIN_HOLE_SIZE*) to determine if a file system supports SEEK_HOLE. See pathconf(2).

For file systems that do not supply information about holes, the file will be represented as one entire data region.

RETURN VALUES

Upon successful completion, **lseek**() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **lseek**() system call will fail and the file position pointer will remain unchanged if:

[EBADF] The *fildes* argument is not an open file descriptor.

[EINVAL] The *whence* argument is not a proper value or the resulting file offset would be negative for a non-character special file.

[ENXIO] For SEEK_DATA, there are no more data regions past the supplied offset. Due to existence of the hole at the end of the file, for SEEK_HOLE this error is only returned when the *offset* already points to the end-of-file position.

[EOVERFLOW] The resulting file offset would be a value which cannot be represented correctly in an object of type *off_t*.

[ESPIPE] The *fildes* argument is associated with a pipe, socket, or FIFO.

SEE ALSO

dup(2), open(2), pathconf(2)

STANDARDS

The **lseek**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **lseek**() function appeared in Version 7 AT&T UNIX.

BUGS

This document's use of *whence* is incorrect English, but is maintained for historical reasons.

NAME

mac_free - free MAC label

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mac.h>

int
mac_free(*mac_t label*);

DESCRIPTION

The **mac_free**() function frees the storage allocated to contain a *mac_t*.

RETURN VALUES

The **mac_free**() function always returns 0. WARNING: see the notes in the *BUGS* section regarding the use of this function.

SEE ALSO

mac(3), mac_get(3), mac_prepare(3), mac_set(3), mac_text(3), posix1e(3), mac(4), mac(9)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

Support for Mandatory Access Control was introduced in FreeBSD 5.0 as part of the TrustedBSD Project.

BUGS

POSIX.1e specifies that **mac_free**() will be used to free text strings created using mac_to_text(3). Because *mac_t* is a complex structure in the TrustedBSD implementation, **mac_free**() is specific to *mac_3*, and must not be used to free the character strings returned from **mac_to_text**(). Doing so may result in undefined behavior.

NAME

mac_from_text, **mac_to_text** - convert MAC label to/from text representation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mac.h>

int
mac_from_text(*mac_t *mac*, *const char *text*);

int
mac_to_text(*mac_t label*, *char **text*);

DESCRIPTION

The **mac_from_text**() function converts the text representation of a label into the internal policy label format (*mac_t*) and places it in **mac*, which must later be freed with free(3).

The **mac_to_text**() function allocates storage for **text*, which will be set to the text representation of *label*.

Refer to maclabel(7) for the MAC label format.

RETURN VALUES

The **mac_from_text**() and **mac_to_text**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

COMPATIBILITY

POSIX.1e does not define a format for text representations of MAC labels.

POSIX.1e requires that text strings allocated using **mac_to_text**() be freed using mac_free(3); in the FreeBSD implementation, they must be freed using free(3), as mac_free(3) is used only to free memory used for type *mac_t*.

ERRORS

[ENOMEM] Insufficient memory was available to allocate internal storage.

SEE ALSO

free(3), mac(3), mac_get(3), mac_is_present(3), mac_prepare(3), mac_set(3), posix1e(3), mac(4),

maclabel(7)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

Support for Mandatory Access Control was introduced in FreeBSD 5.0 as part of the TrustedBSD Project.

NAME

mac_get_file, **mac_get_link**, **mac_get_fd**, **mac_get_peer**, **mac_get_pid**, **mac_get_proc** - get the label of a file, socket, socket peer or process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mac.h>

int
mac_get_file(*const char *path*, *mac_t label*);

int
mac_get_link(*const char *path*, *mac_t label*);

int
mac_get_fd(*int fd*, *mac_t label*);

int
mac_get_peer(*int fd*, *mac_t label*);

int
mac_get_pid(*pid_t pid*, *mac_t label*);

int
mac_get_proc(*mac_t label*);

DESCRIPTION

The **mac_get_file**() system call returns the label associated with a file specified by pathname. The **mac_get_link**() function is the same as **mac_get_file**(), except that it does not follow symlinks.

The **mac_get_fd**() system call returns the label associated with an object referenced by the specified file descriptor. Note that in the case of a file system socket, the label returned will be the socket label, which may be different from the label of the on-disk node acting as a rendezvous for the socket. The **mac_get_peer**() system call returns the label associated with the remote endpoint of a socket; the exact semantics of this call will depend on the protocol domain, communications type, and endpoint; typically this label will be cached when a connection-oriented protocol instance is first set up, and is undefined for datagram protocols.

The **mac_get_pid**() and **mac_get_proc**() system calls return the process label associated with an arbitrary process ID, or the current process.

Label storage for use with these calls must first be allocated and prepared using the mac_prepare(3) functions. When an application is done using a label, the memory may be returned using mac_free(3).

ERRORS

[EACCES] A component of *path* is not searchable, or MAC read access to the file is denied.

[EINVAL] The requested label operation is not valid for the object referenced by *fd*.

[ENAMETOOLONG]
The pathname pointed to by *path* exceeds PATH_MAX, or a component of the pathname exceeds NAME_MAX.

[ENOENT] A component of *path* does not exist.

[ENOMEM] Insufficient memory is available to allocate a new MAC label structure.

[ENOTDIR] A component of *path* is not a directory.

SEE ALSO

mac(3), mac_free(3), mac_prepare(3), mac_set(3), mac_text(3), posix1e(3), mac(4), mac(9)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17. Discussion of the draft continues on the cross-platform POSIX.1e implementation mailing list. To join this list, see the FreeBSD POSIX.1e implementation page for more information.

HISTORY

Support for Mandatory Access Control was introduced in FreeBSD 5.0 as part of the TrustedBSD Project.

NAME

mac_is_present - report whether the running system has MAC support

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mac.h>

int
mac_is_present(*const char *policyname*);

DESCRIPTION

The **mac_is_present**() function determines whether the currently-running kernel supports MAC for a given policy or not. If *policyname* is non-NULL, the presence of the named policy (e.g. "biba", "mls", "te") is checked, otherwise the presence of any MAC policies at all is checked.

RETURN VALUES

If the system supports the given MAC policy, the value 1 is returned. If the specified MAC policy is not supported, the value 0 is returned. If an error occurs, the value -1 is returned.

ERRORS

[EINVAL] The value of *policyname* is not valid.

[ENOMEM] Insufficient memory was available to allocate internal storage.

SEE ALSO

mac(3), mac_free(3), mac_get(3), mac_prepare(3), mac_set(3), mac_text(3), mac(4), mac(9)

HISTORY

Support for Mandatory Access Control was introduced in FreeBSD 5.0 as part of the TrustedBSD Project.

NAME

mac_set_file, **mac_set_fd**, **mac_set_proc** - set the MAC label for a file or process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mac.h>

int
mac_set_file(*const char *path*, *mac_t label*);

int
mac_set_link(*const char *path*, *mac_t label*);

int
mac_set_fd(*int fd*, *mac_t label*);

int
mac_set_proc(*mac_t label*);

DESCRIPTION

The **mac_set_file**() and **mac_set_fd**() functions associate a MAC label specified by *label* to the file referenced to by *path_p*, or to the file descriptor *fd*, respectively. Note that when a file descriptor references a socket, label operations on the file descriptor act on the socket, not on the file that may have been used as a rendezvous when binding the socket. The **mac_set_link**() function is the same as **mac_set_file**(), except that it does not follow symlinks.

The **mac_set_proc**() function associates the MAC label specified by *label* to the calling process.

A process is allowed to set a label for a file only if it has MAC write access to the file, and its effective user ID is equal to the owner of the file, or has appropriate privileges.

RETURN VALUES

The **mac_set_fd**(), **mac_set_file**(), **mac_set_link**(), and **mac_set_proc**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EACCES] MAC write access to the file is denied.

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

[EINVAL] The *label* argument is not a valid MAC label, or the object referenced by *fd* is not appropriate for label operations.

[EOPNOTSUPP] Setting MAC labels is not supported by the file referenced by *fd*.

[EPERM] The calling process had insufficient privilege to change the MAC label.

[EROFS] File system for the object being modified is read only.

[ENAMETOOLONG]
The length of the pathname in *path_p* exceeds PATH_MAX, or a component of the pathname is longer than NAME_MAX.

[ENOENT] The file referenced by *path_p* does not exist.

[ENOTDIR] A component of the pathname referenced by *path_p* is not a directory.

SEE ALSO

mac(3), mac_free(3), mac_get(3), mac_is_present(3), mac_prepare(3), mac_text(3), posix1e(3), mac(4), mac(9)

HISTORY

Support for Mandatory Access Control was introduced in FreeBSD 5.0 as part of the TrustedBSD Project.

NAME

mac - introduction to the MAC security API

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mac.h>

In the kernel configuration file:
options MAC

DESCRIPTION

Mandatory Access Control labels describe confidentiality, integrity, and other security attributes of operating system objects, overriding discretionary access control. Not all system objects support MAC labeling, and MAC policies must be explicitly enabled by the administrator. This API, based on POSIX.1e, includes routines to retrieve, manipulate, set, and convert to and from text the MAC labels on files and processes.

MAC labels consist of a set of (name, value) tuples, representing security attributes from MAC policies. For example, this label contains security labels defined by two policies, mac_biba(4) and mac_mls(4):

```
biba/low,mls/low
```

Further syntax and semantics of MAC labels may be found in maclabel(7).

Applications operate on labels stored in *mac_t*, but can convert between this internal format and a text format for the purposes of presentation to uses or external storage. When querying a label on an object, a *mac_t* must first be prepared using the interfaces described in mac_prepare(3), allowing the application to declare which policies it wishes to interrogate. The application writer can also rely on default label names declared in mac.conf(5).

When finished with a *mac_t*, the application must call mac_free(3) to release its storage.

The following functions are defined:

mac_is_present()
: This function, described in mac_is_present(3), allows applications to test whether MAC is configured, as well as whether specific policies are configured.

mac_get_fd(), **mac_get_file**(), **mac_get_link**(), **mac_get_peer**()
These functions, described in mac_get(3), retrieve the MAC labels associated with file descriptors, files, and socket peers.

mac_get_pid(), **mac_get_proc**()
These functions, described in mac_get(3), retrieve the MAC labels associated with processes.

mac_set_fd(), **mac_set_file**(), **mac_set_link**()
These functions, described in mac_set(3), set the MAC labels associated with file descriptors and files.

mac_set_proc()
This function, described in mac_set(3), sets the MAC label associated with the current process.

mac_free()
This function, described in mac_free(3), frees working MAC label storage.

mac_from_text()
This function, described in mac_text(3), converts a text-form MAC label into working MAC label storage, *mac_t*.

mac_prepare(), **mac_prepare_file_label**(), **mac_prepare_ifnet_label**(), **mac_prepare_process_label**(), **mac_prepare_type**()
These functions, described in mac_prepare(3), allocate working storage for MAC label operations. mac_prepare(3) prepares a label based on caller-specified label names; the other calls rely on the default configuration specified in mac.conf(5).

mac_to_text()
This function is described in mac_text(3), and may be used to convert a *mac_t* into a text-form MAC label.

FILES

/etc/mac.conf MAC library configuration file, documented in mac.conf(5). Provides default behavior for applications aware of MAC labels on system objects, but without policy-specific knowledge.

SEE ALSO

mac_free(3), mac_get(3), mac_is_present(3), mac_prepare(3), mac_set(3), mac_text(3), posix1e(3), mac(4), mac.conf(5), mac(9)

STANDARDS

These APIs are loosely based on the APIs described in POSIX.1e, as described in IEEE POSIX.1e draft 17. However, the resemblance of these APIs to the POSIX APIs is loose, as the POSIX APIs were unable to express some notions required for flexible and extensible access control.

HISTORY

Support for Mandatory Access Control was introduced in FreeBSD 5.0 as part of the TrustedBSD Project.

NAME

madvise, **posix_madvise** - give advice about use of memory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
madvise(*void *addr*, *size_t len*, *int behav*);

int
posix_madvise(*void *addr*, *size_t len*, *int behav*);

DESCRIPTION

The **madvise**() system call allows a process that has knowledge of its memory behavior to describe it to the system. The **posix_madvise**() interface is identical, except it returns an error number on error and does not modify *errno*, and is provided for standards conformance.

The known behaviors are:

MADV_NORMAL Tells the system to revert to the default paging behavior.

MADV_RANDOM Is a hint that pages will be accessed randomly, and prefetching is likely not advantageous.

MADV_SEQUENTIAL Causes the VM system to depress the priority of pages immediately preceding a given page when it is faulted in.

MADV_WILLNEED Causes pages that are in a given virtual address range to temporarily have higher priority, and if they are in memory, decrease the likelihood of them being freed. Additionally, the pages that are already in memory will be immediately mapped into the process, thereby eliminating unnecessary overhead of going through the entire process of faulting the pages in. This WILL NOT fault pages in from backing store, but quickly map the pages already in memory into the calling process.

MADV_DONTNEED Allows the VM system to decrease the in-memory priority of pages in the specified address range. Consequently, future references to this address range

are more likely to incur a page fault.

MADV_FREE Gives the VM system the freedom to free pages, and tells the system that information in the specified page range is no longer important. This is an efficient way of allowing malloc(3) to free pages anywhere in the address space, while keeping the address space valid. The next time that the page is referenced, the page might be demand zeroed, or might contain the data that was there before the MADV_FREE call. References made to that address space range will not make the VM system page the information back in from backing store until the page is modified again.

MADV_NOSYNC Request that the system not flush the data associated with this map to physical backing store unless it needs to. Typically this prevents the file system update daemon from gratuitously writing pages dirtied by the VM system to physical disk. Note that VM/file system coherency is always maintained, this feature simply ensures that the mapped data is only flush when it needs to be, usually by the system pager.

This feature is typically used when you want to use a file-backed shared memory area to communicate between processes (IPC) and do not particularly need the data being stored in that area to be physically written to disk. With this feature you get the equivalent performance with mmap that you would expect to get with SysV shared memory calls, but in a more controllable and less restrictive manner. However, note that this feature is not portable across UNIX platforms (though some may do the right thing by default). For more information see the MAP_NOSYNC section of mmap(2)

MADV_AUTOSYNC Undoes the effects of MADV_NOSYNC for any future pages dirtied within the address range. The effect on pages already dirtied is indeterminate - they may or may not be reverted. You can guarantee reversion by using the msync(2) or fsync(2) system calls.

MADV_NOCORE Region is not included in a core file.

MADV_CORE Include region in a core file.

MADV_PROTECT Informs the VM system this process should not be killed when the swap space is exhausted. The process must have superuser privileges. This should be used judiciously in processes that must remain running for the system to properly function.

Portable programs that call the **posix_madvise**() interface should use the aliases POSIX_MADV_NORMAL, POSIX_MADV_SEQUENTIAL, POSIX_MADV_RANDOM, POSIX_MADV_WILLNEED, and POSIX_MADV_DONTNEED rather than the flags described above.

RETURN VALUES

The **madvise**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **madvise**() system call will fail if:

[EINVAL]	The *behav* argument is not valid.
[ENOMEM]	The virtual address range specified by the *addr* and *len* arguments is not valid.
[EPERM]	MADV_PROTECT was specified and the process does not have superuser privileges.

SEE ALSO

mincore(2), mprotect(2), msync(2), munmap(2), posix_fadvise(2)

STANDARDS

The **posix_madvise**() interface conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **madvise**() system call first appeared in 4.4BSD.

NAME

makecontext, **swapcontext** - modify and exchange user thread contexts

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <ucontext.h>

void
makecontext(*ucontext_t *ucp*, *void (*func)(void)*, *int argc*, ...);

int
swapcontext(*ucontext_t *oucp*, *const ucontext_t *ucp*);

DESCRIPTION

The **makecontext**() function modifies the user thread context pointed to by *ucp*, which must have previously been initialized by a call to getcontext(3) and had a stack allocated for it. The context is modified so that it will continue execution by invoking **func**() with the arguments provided. The *argc* argument must be equal to the number of additional arguments of type *int* provided to **makecontext**() and also equal to the number of arguments of type *int* to **func**(); otherwise , the behavior is undefined.

The *ucp->uc_link* argument must be initialized before calling **makecontext**() and determines the action to take when **func**() returns: if equal to NULL, the process exits; otherwise, **setcontext**(*ucp->uc_link*) is implicitly invoked.

The **swapcontext**() function saves the current thread context in **oucp* and makes **ucp* the currently active context.

RETURN VALUES

If successful, **swapcontext**() returns zero; otherwise -1 is returned and the global variable *errno* is set appropriately.

ERRORS

The **swapcontext**() function will fail if:

[ENOMEM] There is not enough stack space in *ucp* to complete the operation.

SEE ALSO

setcontext(3), ucontext(3)

NAME

mblen - get number of bytes in a character

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
mblen(*const char *mbchar*, *size_t nbytes*);

DESCRIPTION

The **mblen**() function computes the length in bytes of a multibyte character *mbchar* according to the current conversion state. Up to *nbytes* bytes are examined.

A call with a null *mbchar* pointer returns nonzero if the current locale requires shift states, zero otherwise; if shift states are required, the shift state is reset to the initial state.

RETURN VALUES

If *mbchar* is NULL, the **mblen**() function returns nonzero if shift states are supported, zero otherwise.

Otherwise, if *mbchar* is not a null pointer, **mblen**() either returns 0 if *mbchar* represents the null wide character, or returns the number of bytes processed in *mbchar*, or returns -1 if no multibyte character could be recognized or converted. In this case, **mblen**()'s internal conversion state is undefined.

ERRORS

The **mblen**() function will fail if:

[EILSEQ] An invalid multibyte sequence was detected.

[EINVAL] The internal conversion state is not valid.

SEE ALSO

mbrlen(3), mbtowc(3), multibyte(3)

STANDARDS

The **mblen**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

mbrlen - get number of bytes in a character (restartable)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

size_t
mbrlen(*const char * restrict s*, *size_t n*, *mbstate_t * restrict ps*);

DESCRIPTION

The **mbrlen**() function inspects at most *n* bytes pointed to by *s* to determine the number of bytes needed to complete the next multibyte character.

The *mbstate_t* argument, *ps*, is used to keep track of the shift state. If it is NULL, **mbrlen**() uses an internal, static *mbstate_t* object, which is initialized to the initial conversion state at program startup.

It is equivalent to:

```
mbrtowc(NULL, s, n, ps);
```

Except that when *ps* is a NULL pointer, **mbrlen**() uses its own static, internal *mbstate_t* object to keep track of the shift state.

RETURN VALUES

The **mbrlen**() functions returns:

0 The next *n* or fewer bytes represent the null wide character (L'\0').

>0 The next *n* or fewer bytes represent a valid character, **mbrlen**() returns the number of bytes used to complete the multibyte character.

(*size_t*)-2
The next *n* contribute to, but do not complete, a valid multibyte character sequence, and all *n* bytes have been processed.

(*size_t*)-1
An encoding error has occurred. The next *n* or fewer bytes do not contribute to a valid multibyte

character.

EXAMPLES

A function that calculates the number of characters in a multibyte character string:

```
size_t
nchars(const char *s)
{
        size_t charlen, chars;
        mbstate_t mbs;

        chars = 0;
        memset(&mbs, 0, sizeof(mbs));
        while ((charlen = mbrlen(s, MB_CUR_MAX, &mbs)) != 0 &&
           charlen != (size_t)-1 && charlen != (size_t)-2) {
                s += charlen;
                chars++;
        }

        return (chars);
}
```

ERRORS

The **mbrlen**() function will fail if:

[EILSEQ] An invalid multibyte sequence was detected.

[EINVAL] The conversion state is invalid.

SEE ALSO

mblen(3), mbrtowc(3), multibyte(3)

STANDARDS

The **mbrlen**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

mbrtowc, **mbrtoc16**, **mbrtoc32** - convert a character to a wide-character code (restartable)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

size_t
mbrtowc(*wchar_t * restrict pc*, *const char * restrict s*, *size_t n*, *mbstate_t * restrict ps*);

#include <uchar.h>

size_t
mbrtoc16(*char16_t * restrict pc*, *const char * restrict s*, *size_t n*, *mbstate_t * restrict ps*);

size_t
mbrtoc32(*char32_t * restrict pc*, *const char * restrict s*, *size_t n*, *mbstate_t * restrict ps*);

DESCRIPTION

The **mbrtowc**(), **mbrtoc16**() and **mbrtoc32**() functions inspect at most *n* bytes pointed to by *s* to determine the number of bytes needed to complete the next multibyte character. If a character can be completed, and *pc* is not NULL, the wide character which is represented by *s* is stored in the *wchar_t*, *char16_t* or *char32_t* it points to.

If *s* is NULL, these functions behave as if *pc* was NULL, *s* was an empty string ("") and *n* was 1.

The *mbstate_t* argument, *ps*, is used to keep track of the shift state. If it is NULL, these functions use an internal, static *mbstate_t* object, which is initialized to the initial conversion state at program startup.

As a single *char16_t* is not large enough to represent certain multibyte characters, the **mbrtoc16**() function may need to be invoked multiple times to convert a single multibyte character sequence.

RETURN VALUES

The **mbrtowc**(), **mbrtoc16**() and **mbrtoc32**() functions return:

0 The next *n* or fewer bytes represent the null wide character (L'\0').

>0 The next *n* or fewer bytes represent a valid character, these functions return the number of bytes

used to complete the multibyte character.

(*size_t*)-1
: An encoding error has occurred. The next *n* or fewer bytes do not contribute to a valid multibyte character.

(*size_t*)-2
: The next *n* contribute to, but do not complete, a valid multibyte character sequence, and all *n* bytes have been processed.

The **mbrtoc16**() function also returns:

(*size_t*)-3
: The next character resulting from a previous call has been stored. No bytes from the input have been consumed.

ERRORS

The **mbrtowc**(), **mbrtoc16**() and **mbrtoc32**() functions will fail if:

[EILSEQ] An invalid multibyte sequence was detected.

[EINVAL] The conversion state is invalid.

SEE ALSO

mbtowc(3), multibyte(3), setlocale(3), wcrtomb(3)

STANDARDS

The **mbrtowc**(), **mbrtoc16**() and **mbrtoc32**() functions conform to ISO/IEC 9899:2011 ("ISO C11").

NAME

mbsinit - determine conversion object status

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

int
mbsinit(*const mbstate_t *ps*);

DESCRIPTION

The **mbsinit**() function determines whether the *mbstate_t* object pointed to by *ps* describes an initial conversion state.

RETURN VALUES

The **mbsinit**() function returns non-zero if *ps* is NULL or describes an initial conversion state, otherwise it returns zero.

SEE ALSO

mbrlen(3), mbrtowc(3), mbsrtowcs(3), multibyte(3), wcrtomb(3), wcsrtombs(3)

STANDARDS

The **mbsinit**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

mbsrtowcs, **mbsnrtowcs** - convert a character string to a wide-character string (restartable)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

size_t
mbsrtowcs(*wchar_t * restrict dst*, *const char ** restrict src*, *size_t len*, *mbstate_t * restrict ps*);

size_t
mbsnrtowcs(*wchar_t * restrict dst*, *const char ** restrict src*, *size_t nms*, *size_t len*,
*mbstate_t * restrict ps*);

DESCRIPTION

The **mbsrtowcs**() function converts a sequence of multibyte characters pointed to indirectly by *src* into a sequence of corresponding wide characters and stores at most *len* of them in the *wchar_t* array pointed to by *dst*, until it encounters a terminating null character ('\0').

If *dst* is NULL, no characters are stored.

If *dst* is not NULL, the pointer pointed to by *src* is updated to point to the character after the one that conversion stopped at. If conversion stops because a null character is encountered, **src* is set to NULL.

The *mbstate_t* argument, *ps*, is used to keep track of the shift state. If it is NULL, **mbsrtowcs**() uses an internal, static *mbstate_t* object, which is initialized to the initial conversion state at program startup.

The **mbsnrtowcs**() function behaves identically to **mbsrtowcs**(), except that conversion stops after reading at most *nms* bytes from the buffer pointed to by *src*.

RETURN VALUES

The **mbsrtowcs**() and **mbsnrtowcs**() functions return the number of wide characters stored in the array pointed to by *dst* if successful, otherwise it returns (*size_t*)-1.

ERRORS

The **mbsrtowcs**() and **mbsnrtowcs**() functions will fail if:

[EILSEQ] An invalid multibyte character sequence was encountered.

[EINVAL] The conversion state is invalid.

SEE ALSO

mbrtowc(3), mbstowcs(3), multibyte(3), wcsrtombs(3)

STANDARDS

The **mbsrtowcs**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

The **mbsnrtowcs**() function is an extension to the standard.

NAME

mbstowcs - convert a character string to a wide-character string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

size_t
mbstowcs(*wchar_t * restrict wcstring*, *const char * restrict mbstring*, *size_t nwchars*);

DESCRIPTION

The **mbstowcs**() function converts a multibyte character string *mbstring* beginning in the initial conversion state into a wide character string *wcstring*. No more than *nwchars* wide characters are stored. A terminating null wide character is appended if there is room.

RETURN VALUES

The **mbstowcs**() function returns the number of wide characters converted, not counting any terminating null wide character, or -1 if an invalid multibyte character was encountered.

ERRORS

The **mbstowcs**() function will fail if:

[EILSEQ] An invalid multibyte sequence was detected.

[EINVAL] The conversion state is invalid.

SEE ALSO

mbsrtowcs(3), mbtowc(3), multibyte(3)

STANDARDS

The **mbstowcs**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

mbtowc - convert a character to a wide-character code

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
mbtowc(*wchar_t * restrict wcharp*, *const char * restrict mbchar*, *size_t nbytes*);

DESCRIPTION

The **mbtowc**() function converts a multibyte character *mbchar* into a wide character according to the current conversion state, and stores the result in the object pointed to by *wcharp*. Up to *nbytes* bytes are examined.

A call with a null *mbchar* pointer returns nonzero if the current encoding requires shift states, zero otherwise; if shift states are required, the shift state is reset to the initial state.

RETURN VALUES

If *mbchar* is NULL, the **mbtowc**() function returns nonzero if shift states are supported, zero otherwise.

Otherwise, if *mbchar* is not a null pointer, **mbtowc**() either returns 0 if *mbchar* represents the null wide character, or returns the number of bytes processed in *mbchar*, or returns -1 if no multibyte character could be recognized or converted. In this case, **mbtowc**()'s internal conversion state is undefined.

ERRORS

The **mbtowc**() function will fail if:

[EILSEQ] An invalid multibyte sequence was detected.

[EINVAL] The internal conversion state is invalid.

SEE ALSO

btowc(3), mblen(3), mbrtowc(3), mbstowcs(3), multibyte(3), wctomb(3)

STANDARDS

The **mbtowc**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

memmem - locate a byte substring in a byte string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

*void **

memmem(*const void *big*, *size_t big_len*, *const void *little*, *size_t little_len*);

DESCRIPTION

The **memmem**() function locates the first occurrence of the byte string *little* in the byte string *big*.

RETURN VALUES

If *little_len* is zero *big* is returned (that is, an empty little is deemed to match at the beginning of big); if *little* occurs nowhere in *big*, NULL is returned; otherwise a pointer to the first character of the first occurrence of *little* is returned.

SEE ALSO

memchr(3), strchr(3), strstr(3)

CONFORMING TO

memmem() is a GNU extension.

HISTORY

The **memmem**() function first appeared in FreeBSD 6.0. It was replaced with an optimized O(n) implementation from the musl libc project in FreeBSD 12.0. Pascal Gloor <*pascal.gloor@spale.com*> provided this man page along with the previous implementation.

BUGS

This function was broken in Linux libc up to and including version 5.0.9 and in GNU libc prior to version 2.1. Prior to FreeBSD 11.0 **memmem** returned NULL when *little_len* equals 0.

NAME

mincore - determine residency of memory pages

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
mincore(*const void *addr*, *size_t len*, *char *vec*);

DESCRIPTION

The **mincore**() system call determines whether each of the pages in the region beginning at *addr* and continuing for *len* bytes is resident. The status is returned in the *vec* array, one character per page. Each character is either 0 if the page is not resident, or a combination of the following flags (defined in <*sys/mman.h*>):

MINCORE_INCORE	Page is in core (resident).
MINCORE_REFERENCED	Page has been referenced by us.
MINCORE_MODIFIED	Page has been modified by us.
MINCORE_REFERENCED_OTHER	Page has been referenced.
MINCORE_MODIFIED_OTHER	Page has been modified.
MINCORE_SUPER	Page is part of a large ("super") page.

The information returned by **mincore**() may be out of date by the time the system call returns. The only way to ensure that a page is resident is to lock it into memory with the mlock(2) system call.

RETURN VALUES

The **mincore**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **mincore**() system call will fail if:

[ENOMEM] The virtual address range specified by the *addr* and *len* arguments is not fully mapped.

[EFAULT] The *vec* argument points to an illegal address.

SEE ALSO

madvise(2), mlock(2), mprotect(2), msync(2), munmap(2), getpagesize(3)

HISTORY

The **mincore**() system call first appeared in 4.4BSD.

NAME

minherit - control the inheritance of pages

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
minherit(*void *addr*, *size_t len*, *int inherit*);

DESCRIPTION

The **minherit**() system call changes the specified pages to have the inheritance characteristic *inherit*. Not all implementations will guarantee that the inheritance characteristic can be set on a page basis; the granularity of changes may be as large as an entire region. FreeBSD is capable of adjusting inheritance characteristics on a page basis. Inheritance only effects children created by **fork**(). It has no effect on **exec**(). exec'd processes replace their address space entirely. This system call also has no effect on the parent's address space (other than to potentially share the address space with its children).

Inheritance is a rather esoteric feature largely superseded by the MAP_SHARED feature of **mmap**(). However, it is possible to use **minherit**() to share a block of memory between parent and child that has been mapped MAP_PRIVATE. That is, modifications made by parent or child are shared but the original underlying file is left untouched.

INHERIT_SHARE — This option causes the address space in question to be shared between parent and child. It has no effect on how the original underlying backing store was mapped.

INHERIT_NONE — This option prevents the address space in question from being inherited at all. The address space will be unmapped in the child.

INHERIT_COPY — This option causes the child to inherit the address space as copy-on-write. This option also has an unfortunate side effect of causing the parent address space to become copy-on-write when the parent forks. If the original mapping was MAP_SHARED, it will no longer be shared in the parent after the parent forks and there is no way to get the previous shared-backing-store mapping without unmapping and remapping the address space in the parent.

INHERIT_ZERO — This option causes the address space in question to be mapped as new

anonymous pages, which would be initialized to all zero bytes, in the child process.

RETURN VALUES

The **minherit**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **minherit**() system call will fail if:

[EINVAL] The virtual address range specified by the *addr* and *len* arguments is not valid.

[EACCES] The flags specified by the *inherit* argument were not valid for the pages specified by the *addr* and *len* arguments.

SEE ALSO

fork(2), madvise(2), mincore(2), mprotect(2), msync(2), munmap(2), rfork(2)

HISTORY

The **minherit**() system call first appeared in OpenBSD and then in FreeBSD 2.2.

The INHERIT_ZERO support first appeared in OpenBSD 5.6 and then in FreeBSD 12.0.

BUGS

Once you set inheritance to MAP_PRIVATE or MAP_SHARED, there is no way to recover the original copy-on-write semantics short of unmapping and remapping the area.

NAME

mkdir, **mkdirat** - make a directory file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>

int
mkdir(*const char *path*, *mode_t mode*);

int
mkdirat(*int fd*, *const char *path*, *mode_t mode*);

DESCRIPTION

The directory *path* is created with the access permissions specified by *mode* and restricted by the umask(2) of the calling process.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to that of the parent directory in which it is created.

The **mkdirat**() system call is equivalent to **mkdir**() except in the case where *path* specifies a relative path. In this case the newly created directory is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. If **mkdirat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **mkdir**().

RETURN VALUES

The **mkdir**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **mkdir**() system call will fail and no directory will be created if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] A component of the path prefix does not exist.

[EACCES] Search permission is denied for a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The parent directory of the directory to be created has its immutable flag set, see the chflags(2) manual page for more information.

[EROFS] The named directory would reside on a read-only file system.

[EMLINK] The new directory cannot be created because the parent directory contains too many subdirectories.

[EEXIST] The named file exists.

[ENOSPC] The new directory cannot be created because there is no space left on the file system that will contain the directory.

[ENOSPC] There are no free inodes on the file system on which the directory is being created.

[EDQUOT] The new directory cannot be created because the user's quota of disk blocks on the file system that will contain the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the directory is being created has been exhausted.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

In addition to the errors returned by the **mkdir**(), the **mkdirat**() may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chflags(2), chmod(2), stat(2), umask(2)

STANDARDS

The **mkdir**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **mkdirat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **mkdirat**() system call appeared in FreeBSD 8.0. The **mkdir**() system call appeared in Version 1 AT&T UNIX.

NAME

mkfifo, **mkfifoat** - make a fifo file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>

int
mkfifo(*const char *path*, *mode_t mode*);

int
mkfifoat(*int fd*, *const char *path*, *mode_t mode*);

DESCRIPTION

The **mkfifo**() system call creates a new fifo file with name *path*. The access permissions are specified by *mode* and restricted by the umask(2) of the calling process.

The fifo's owner ID is set to the process's effective user ID. The fifo's group ID is set to that of the parent directory in which it is created.

The **mkfifoat**() system call is equivalent to **mkfifo**() except in the case where *path* specifies a relative path. In this case the newly created FIFO is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. If **mkfifoat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **mkfifo**().

RETURN VALUES

The **mkfifo**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **mkfifo**() system call will fail and no fifo will be created if:

[ENOTSUP] The kernel has not been configured to support fifo's.

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] A component of the path prefix does not exist.

[EACCES] A component of the path prefix denies search permission, or write permission is denied on the parent directory of the fifo to be created.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EROFS] The named file would reside on a read-only file system.

[EEXIST] The named file exists.

[EPERM] The parent directory of the named file has its immutable flag set, see the chflags(2) manual page for more information.

[ENOSPC] The directory in which the entry for the new fifo is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] There are no free inodes on the file system on which the fifo is being created.

[EDQUOT] The directory in which the entry for the new fifo is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the fifo is being created has been exhausted.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

In addition to the errors returned by the **mkfifo**(), the **mkfifoat**() may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chflags(2), chmod(2), mknod(2), stat(2), umask(2)

STANDARDS

The **mkfifo**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **mkfifoat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **mkfifoat**() system call appeared in FreeBSD 8.0.

NAME

mknod, **mknodat** - make a special file node

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>

int
mknod(*const char *path*, *mode_t mode*, *dev_t dev*);

int
mknodat(*int fd*, *const char *path*, *mode_t mode*, *dev_t dev*);

DESCRIPTION

The file system node *path* is created with the file type and access permissions specified in *mode*. The access permissions are modified by the process's umask value.

If *mode* indicates a block or character special file, *dev* is a configuration dependent specification denoting a particular device on the system. Otherwise, *dev* is ignored.

The **mknod**() system call requires super-user privileges.

The **mknodat**() system call is equivalent to **mknod**() except in the case where *path* specifies a relative path. In this case the newly created device node is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. If **mknodat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **mknod**().

RETURN VALUES

The **mknod**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **mknod**() system call will fail and the file will be not created if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]

A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] A component of the path prefix does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The process's effective user ID is not super-user.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[ENOSPC] The directory in which the entry for the new node is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] There are no free inodes on the file system on which the node is being created.

[EDQUOT] The directory in which the entry for the new node is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the node is being created has been exhausted.

[EROFS] The named file resides on a read-only file system.

[EEXIST] The named file exists.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EINVAL] Creating anything else than a block or character special file (or a *whiteout*) is not supported.

In addition to the errors returned by the **mknod**(), the **mknodat**() may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file

descriptor associated with a directory.

SEE ALSO

chmod(2), mkfifo(2), stat(2), umask(2)

STANDARDS

The **mknodat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **mknod**() function appeared in Version 4 AT&T UNIX. The **mknodat**() system call appeared in FreeBSD 8.0.

NAME

mktemp - make temporary file name (unique)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*char **
mktemp(*char *template*);

int
mkstemp(*char *template*);

int
mkostemp(*char *template*, *int oflags*);

int
mkostemps(*char *template*, *int suffixlen*, *int oflags*);

*char **
mkdtemp(*char *template*);

#include <unistd.h>

int
mkstemps(*char *template*, *int suffixlen*);

DESCRIPTION

The **mktemp**() function takes the given file name template and overwrites a portion of it to create a file name. This file name is guaranteed not to exist at the time of function invocation and is suitable for use by the application. The template may be any file name with some number of 'Xs' appended to it, for example */tmp/temp.XXXXXX*. The trailing 'Xs' are replaced with a unique alphanumeric combination. The number of unique file names **mktemp**() can return depends on the number of 'Xs' provided; six 'Xs' will result in **mktemp**() selecting one of 56800235584 (62 ** 6) possible temporary file names.

The **mkstemp**() function makes the same replacement to the template and creates the template file, mode 0600, returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

The **mkostemp**() function is like **mkstemp**() but allows specifying additional open(2) flags (defined in <*fcntl.h*>). The permitted flags are O_APPEND, O_DIRECT, O_SHLOCK, O_EXLOCK, O_SYNC and O_CLOEXEC.

The **mkstemps**() and **mkostemps**() functions act the same as **mkstemp**() and **mkostemp**() respectively, except they permit a suffix to exist in the template. The template should be of the form */tmp/tmpXXXXXXsuffix*. The **mkstemps**() and **mkostemps**() function are told the length of the suffix string.

The **mkdtemp**() function makes the same replacement to the template as in **mktemp**() and creates the template directory, mode 0700.

RETURN VALUES

The **mktemp**() and **mkdtemp**() functions return a pointer to the template on success and NULL on failure. The **mkstemp**(), **mkostemp**() **mkstemps**() and **mkostemps**() functions return -1 if no suitable file could be created. If either call fails an error code is placed in the global variable *errno*.

ERRORS

The **mkstemp**(), **mkostemp**(), **mkstemps**(), **mkostemps**() and **mkdtemp**() functions may set *errno* to one of the following values:

[ENOTDIR] The pathname portion of the template is not an existing directory.

The **mkostemp**() and **mkostemps**() functions may also set *errno* to the following value:

[EINVAL] The *oflags* argument is invalid.

The **mkstemp**(), **mkostemp**(), **mkstemps**(), **mkostemps**() and **mkdtemp**() functions may also set *errno* to any value specified by the stat(2) function.

The **mkstemp**(), **mkostemp**(), **mkstemps**() and **mkostemps**() functions may also set *errno* to any value specified by the open(2) function.

The **mkdtemp**() function may also set *errno* to any value specified by the mkdir(2) function.

NOTES

A common problem that results in a core dump is that the programmer passes in a read-only string to **mktemp**(), **mkstemp**(), **mkstemps**() or **mkdtemp**(). This is common with programs that were developed before ISO/IEC 9899:1990 ("ISO C90") compilers were common. For example, calling **mkstemp**() with an argument of "/tmp/tempfile.XXXXXX" will result in a core dump due to **mkstemp**() attempting to

modify the string constant that was given.

The **mkdtemp**(), **mkstemp**() and **mktemp**() function prototypes are also available from <*unistd.h*>.

SEE ALSO

chmod(2), getpid(2), mkdir(2), open(2), stat(2)

STANDARDS

The **mkstemp**() and **mkdtemp**() functions are expected to conform to IEEE Std 1003.1-2008 ("POSIX.1"). The **mktemp**() function is expected to conform to IEEE Std 1003.1-2001 ("POSIX.1") and is not specified by IEEE Std 1003.1-2008 ("POSIX.1"). The **mkostemp**(), **mkstemps**() and **mkostemps**() functions do not conform to any standard.

HISTORY

A **mktemp**() function appeared in Version 7 AT&T UNIX. The **mkstemp**() function appeared in 4.4BSD. The **mkdtemp**() function first appeared in OpenBSD 2.2, and later in FreeBSD 3.2. The **mkstemps**() function first appeared in OpenBSD 2.4, and later in FreeBSD 3.4. The **mkostemp**() and **mkostemps**() functions appeared in FreeBSD 10.0.

BUGS

This family of functions produces filenames which can be guessed, though the risk is minimized when large numbers of ‘Xs’ are used to increase the number of possible temporary filenames. This makes the race in **mktemp**(), between testing for a file’s existence (in the **mktemp**() function call) and opening it for use (later in the user application) particularly dangerous from a security perspective. Whenever it is possible, **mkstemp**() or **mkostemp**() should be used instead, since it does not have the race condition. If **mkstemp**() cannot be used, the filename created by **mktemp**() should be created using the O_EXCL flag to open(2) and the return status of the call should be tested for failure. This will ensure that the program does not continue blindly in the event that an attacker has already created the file with the intention of manipulating or reading its contents.

NAME

mlock, **munlock** - lock (unlock) physical pages in memory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
mlock(*const void *addr*, *size_t len*);

int
munlock(*const void *addr*, *size_t len*);

DESCRIPTION

The **mlock**() system call locks into memory the physical pages associated with the virtual address range starting at *addr* for *len* bytes. The **munlock**() system call unlocks pages previously locked by one or more **mlock**() calls. For both, the *addr* argument should be aligned to a multiple of the page size. If the *len* argument is not a multiple of the page size, it will be rounded up to be so. The entire range must be allocated.

After an **mlock**() system call, the indicated pages will cause neither a non-resident page nor address-translation fault until they are unlocked. They may still cause protection-violation faults or TLB-miss faults on architectures with software-managed TLBs. The physical pages remain in memory until all locked mappings for the pages are removed. Multiple processes may have the same physical pages locked via their own virtual address mappings. A single process may likewise have pages multiply-locked via different virtual mappings of the same physical pages. Unlocking is performed explicitly by **munlock**() or implicitly by a call to **munmap**() which deallocates the unmapped address range. Locked mappings are not inherited by the child process after a fork(2).

Since physical memory is a potentially scarce resource, processes are limited in how much they can lock down. The amount of memory that a single process can **mlock**() is limited by both the per-process RLIMIT_MEMLOCK resource limit and the system-wide "wired pages" limit *vm.max_wired*. *vm.max_wired* applies to the system as a whole, so the amount available to a single process at any given time is the difference between *vm.max_wired* and *vm.stats.vm.v_wire_count*.

If *security.bsd.unprivileged_mlock* is set to 0 these calls are only available to the super-user.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

If the call succeeds, all pages in the range become locked (unlocked); otherwise the locked status of all pages in the range remains unchanged.

ERRORS

The **mlock**() system call will fail if:

[EPERM] *security.bsd.unprivileged_mlock* is set to 0 and the caller is not the super-user.

[EINVAL] The address range given wraps around zero.

[EAGAIN] Locking the indicated range would exceed the system limit for locked memory.

[ENOMEM] Some portion of the indicated address range is not allocated. There was an error faulting/mapping a page. Locking the indicated range would exceed the per-process limit for locked memory.

The **munlock**() system call will fail if:

[EPERM] *security.bsd.unprivileged_mlock* is set to 0 and the caller is not the super-user.

[EINVAL] The address range given wraps around zero.

[ENOMEM] Some or all of the address range specified by the addr and len arguments does not correspond to valid mapped pages in the address space of the process.

[ENOMEM] Locking the pages mapped by the specified range would exceed a limit on the amount of memory that the process may lock.

SEE ALSO

fork(2), mincore(2), minherit(2), mlockall(2), mmap(2), munlockall(2), munmap(2), setrlimit(2), getpagesize(3)

HISTORY

The **mlock**() and **munlock**() system calls first appeared in 4.4BSD.

BUGS

Allocating too much wired memory can lead to a memory-allocation deadlock which requires a reboot to recover from.

The per-process resource limit is a limit on the amount of virtual memory locked, while the system-wide limit is for the number of locked physical pages. Hence a process with two distinct locked mappings of the same physical page counts as 2 pages against the per-process limit and as only a single page in the system limit.

The per-process resource limit is not currently supported.

NAME

mlockall, **munlockall** - lock (unlock) the address space of a process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
mlockall(*int flags*);

int
munlockall(*void*);

DESCRIPTION

The **mlockall**() system call locks into memory the physical pages associated with the address space of a process until the address space is unlocked, the process exits, or execs another program image.

The following flags affect the behavior of **mlockall**():

MCL_CURRENT Lock all pages currently mapped into the process's address space.

MCL_FUTURE Lock all pages mapped into the process's address space in the future, at the time the mapping is established. Note that this may cause future mappings to fail if those mappings cause resource limits to be exceeded.

Since physical memory is a potentially scarce resource, processes are limited in how much they can lock down. A single process can lock the minimum of a system-wide "wired pages" limit *vm.max_wired* and the per-process RLIMIT_MEMLOCK resource limit.

If *security.bsd.unprivileged_mlock* is set to 0 these calls are only available to the super-user. If *vm.old_mlock* is set to 1 the per-process RLIMIT_MEMLOCK resource limit will not be applied for **mlockall**() calls.

The **munlockall**() call unlocks any locked memory regions in the process address space. Any regions mapped after an **munlockall**() call will not be locked.

RETURN VALUES

A return value of 0 indicates that the call succeeded and all pages in the range have either been locked or

unlocked. A return value of -1 indicates an error occurred and the locked status of all pages in the range remains unchanged. In this case, the global location *errno* is set to indicate the error.

ERRORS

mlockall() will fail if:

[EINVAL]	The *flags* argument is zero, or includes unimplemented flags.
[ENOMEM]	Locking the indicated range would exceed either the system or per-process limit for locked memory.
[EAGAIN]	Some or all of the memory mapped into the process's address space could not be locked when the call was made.
[EPERM]	The calling process does not have the appropriate privilege to perform the requested operation.

SEE ALSO

mincore(2), mlock(2), mmap(2), munmap(2), setrlimit(2)

STANDARDS

The **mlockall**() and **munlockall**() functions are believed to conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **mlockall**() and **munlockall**() functions first appeared in FreeBSD 5.1.

BUGS

The per-process resource limit is a limit on the amount of virtual memory locked, while the system-wide limit is for the number of locked physical pages. Hence a process with two distinct locked mappings of the same physical page counts as 2 pages against the per-process limit and as only a single page in the system limit.

NAME

mmap - allocate memory, or map files or devices into memory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

*void **
mmap(*void *addr*, *size_t len*, *int prot*, *int flags*, *int fd*, *off_t offset*);

DESCRIPTION

The **mmap**() system call causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the object described by *fd*, starting at byte offset *offset*. If *len* is not a multiple of the page size, the mapped region may extend past the specified range. Any such extension beyond the end of the mapped object will be zero-filled.

If *fd* references a regular file or a shared memory object, the range of bytes starting at *offset* and continuing for *len* bytes must be legitimate for the possible (not necessarily current) offsets in the object. In particular, the *offset* value cannot be negative. If the object is truncated and the process later accesses a page that is wholly within the truncated region, the access is aborted and a SIGBUS signal is delivered to the process.

If *fd* references a device file, the interpretation of the *offset* value is device specific and defined by the device driver. The virtual memory subsystem does not impose any restrictitions on the *offset* value in this case, passing it unchanged to the driver.

If *addr* is non-zero, it is used as a hint to the system. (As a convenience to the system, the actual address of the region may differ from the address supplied.) If *addr* is zero, an address will be selected by the system. The actual starting address of the region is returned. A successful *mmap* deletes any previous mapping in the allocated address range.

The protections (region accessibility) are specified in the *prot* argument by *or*'ing the following values:

PROT_NONE Pages may not be accessed.
PROT_READ Pages may be read.
PROT_WRITE Pages may be written.
PROT_EXEC Pages may be executed.

The *flags* argument specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. Sharing, mapping type and options are specified in the *flags* argument by *or*'ing the following values:

MAP_32BIT — Request a region in the first 2GB of the current process's address space. If a suitable region cannot be found, **mmap**() will fail. This flag is only available on 64-bit platforms.

MAP_ALIGNED(*n*) — Align the region on a requested boundary. If a suitable region cannot be found, **mmap**() will fail. The *n* argument specifies the binary logarithm of the desired alignment.

MAP_ALIGNED_SUPER — Align the region to maximize the potential use of large ("super") pages. If a suitable region cannot be found, **mmap**() will fail. The system will choose a suitable page size based on the size of mapping. The page size used as well as the alignment of the region may both be affected by properties of the file being mapped. In particular, the physical address of existing pages of a file may require a specific alignment. The region is not guaranteed to be aligned on any specific boundary.

MAP_ANON — Map anonymous memory not associated with any specific file. The file descriptor used for creating MAP_ANON must be -1. The *offset* argument must be 0.

MAP_ANONYMOUS — This flag is identical to MAP_ANON and is provided for compatibility.

MAP_EXCL — This flag can only be used in combination with MAP_FIXED. Please see the definition of MAP_FIXED for the description of its effect.

MAP_FIXED — Do not permit the system to select a different address than the one specified. If the specified address cannot be used, **mmap**() will fail. If MAP_FIXED is specified, *addr* must be a multiple of the page size. If MAP_EXCL is not specified, a successful MAP_FIXED request replaces any previous mappings for the process' pages in the range from *addr* to *addr* + *len*. In contrast, if MAP_EXCL is specified, the request will fail if a mapping already exists within the range.

MAP_GUARD — Instead of a mapping, create a guard of the specified size. Guards allow a process to create reservations in its address space, which can later be replaced by actual mappings.

mmap will not create mappings in the address range of a guard unless the request specifies MAP_FIXED. Guards can be destroyed with munmap(2). Any memory access by a thread to the guarded range results in the delivery of a SIGSEGV signal to that thread.

MAP_NOCORE Region is not included in a core file.

MAP_NOSYNC Causes data dirtied via this VM map to be flushed to physical media only when necessary (usually by the pager) rather than gratuitously. Typically this prevents the update daemons from flushing pages dirtied through such maps and thus allows efficient sharing of memory across unassociated processes using a file-backed shared memory map. Without this option any VM pages you dirty may be flushed to disk every so often (every 30-60 seconds usually) which can create performance problems if you do not need that to occur (such as when you are using shared file-backed mmap regions for IPC purposes). Dirty data will be flushed automatically when all mappings of an object are removed and all descriptors referencing the object are closed. Note that VM/file system coherency is maintained whether you use MAP_NOSYNC or not. This option is not portable across UNIX platforms (yet), though some may implement the same behavior by default.

WARNING! Extending a file with ftruncate(2), thus creating a big hole, and then filling the hole by modifying a shared **mmap**() can lead to severe file fragmentation. In order to avoid such fragmentation you should always pre-allocate the file's backing store by **write**()ing zero's into the newly extended area prior to modifying the area via your **mmap**(). The fragmentation problem is especially sensitive to MAP_NOSYNC pages, because pages may be flushed to disk in a totally random order.

The same applies when using MAP_NOSYNC to implement a file-based shared memory store. It is recommended that you create the backing store by **write**()ing zero's to the backing file rather than **ftruncate**()ing it. You can test file fragmentation by observing the KB/t (kilobytes per transfer) results from an "iostat 1" while reading a large file sequentially, e.g., using "dd if=filename of=/dev/null bs=32k".

The fsync(2) system call will flush all dirty data and metadata associated with a file, including dirty NOSYNC VM data, to physical media. The sync(8) command and sync(2) system call generally do not flush dirty

NOSYNC VM data. The msync(2) system call is usually not needed since BSD implements a coherent file system buffer cache. However, it may be used to associate dirty VM pages with file system buffers and thus cause them to be flushed to physical media sooner rather than later.

MAP_PREFAULT_READ Immediately update the calling process's lowest-level virtual address translation structures, such as its page table, so that every memory resident page within the region is mapped for read access. Ordinarily these structures are updated lazily. The effect of this option is to eliminate any soft faults that would otherwise occur on the initial read accesses to the region. Although this option does not preclude *prot* from including PROT_WRITE, it does not eliminate soft faults on the initial write accesses to the region.

MAP_PRIVATE Modifications are private.

MAP_SHARED Modifications are shared.

MAP_STACK MAP_STACK implies MAP_ANON, and *offset* of 0. The *fd* argument must be -1 and *prot* must include at least PROT_READ and PROT_WRITE.

This option creates a memory region that grows to at most *len* bytes in size, starting from the stack top and growing down. The stack top is the starting address returned by the call, plus *len* bytes. The bottom of the stack at maximum growth is the starting address returned by the call.

Stacks created with MAP_STACK automatically grow. Guards prevent inadvertent use of the regions into which those stacks can grow without requiring mapping the whole stack in advance.

The close(2) system call does not unmap pages, see munmap(2) for further information.

NOTES

Although this implementation does not impose any alignment restrictions on the *offset* argument, a portable program must only use page-aligned values.

Large page mappings require that the pages backing an object be aligned in matching blocks in both the virtual address space and RAM. The system will automatically attempt to use large page mappings when mapping an object that is already backed by large pages in RAM by aligning the mapping request

in the virtual address space to match the alignment of the large physical pages. The system may also use large page mappings when mapping portions of an object that are not yet backed by pages in RAM. The MAP_ALIGNED_SUPER flag is an optimization that will align the mapping request to the size of a large page similar to MAP_ALIGNED, except that the system will override this alignment if an object already uses large pages so that the mapping will be consistent with the existing large pages. This flag is mostly useful for maximizing the use of large pages on the first mapping of objects that do not yet have pages present in RAM.

RETURN VALUES

Upon successful completion, **mmap**() returns a pointer to the mapped region. Otherwise, a value of MAP_FAILED is returned and *errno* is set to indicate the error.

ERRORS

The **mmap**() system call will fail if:

[EACCES] The flag PROT_READ was specified as part of the *prot* argument and *fd* was not open for reading. The flags MAP_SHARED and PROT_WRITE were specified as part of the *flags* and *prot* argument and *fd* was not open for writing.

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

[EINVAL] An invalid (negative) value was passed in the *offset* argument, when *fd* referenced a regular file or shared memory.

[EINVAL] An invalid value was passed in the *prot* argument.

[EINVAL] An undefined option was set in the *flags* argument.

[EINVAL] Both MAP_PRIVATE and MAP_SHARED were specified.

[EINVAL] None of MAP_ANON, MAP_GUARD, MAP_PRIVATE, MAP_SHARED, or MAP_STACK was specified. At least one of these flags must be included.

[EINVAL] MAP_FIXED was specified and the *addr* argument was not page aligned, or part of the desired address space resides out of the valid address space for a user process.

[EINVAL] Both MAP_FIXED and MAP_32BIT were specified and part of the desired address space resides outside of the first 2GB of user address space.

[EINVAL] The *len* argument was equal to zero.

[EINVAL] MAP_ALIGNED was specified and the desired alignment was either larger than the virtual address size of the machine or smaller than a page.

[EINVAL] MAP_ANON was specified and the *fd* argument was not -1.

[EINVAL] MAP_ANON was specified and the *offset* argument was not 0.

[EINVAL] Both MAP_FIXED and MAP_EXCL were specified, but the requested region is already used by a mapping.

[EINVAL] MAP_EXCL was specified, but MAP_FIXED was not.

[EINVAL] MAP_GUARD was specified, but the *offset* argument was not zero, the *fd* argument was not -1, or the *prot* argument was not PROT_NONE.

[EINVAL] MAP_GUARD was specified together with one of the flags MAP_ANON, MAP_PREFAULT, MAP_PREFAULT_READ, MAP_PRIVATE, MAP_SHARED, MAP_STACK.

[ENODEV] MAP_ANON has not been specified and *fd* did not reference a regular or character special file.

[ENOMEM] MAP_FIXED was specified and the *addr* argument was not available. MAP_ANON was specified and insufficient memory was available.

SEE ALSO

madvise(2), mincore(2), minherit(2), mlock(2), mprotect(2), msync(2), munlock(2), munmap(2), getpagesize(3), getpagesizes(3)

NAME

modfind - returns the modid of a kernel module

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/module.h>

int
modfind(*const char *modname*);

DESCRIPTION

The **modfind**() system call returns the modid of the kernel module referenced by *modname*.

RETURN VALUES

The **modfind**() system call returns the modid of the kernel module referenced by *modname*. Upon error, **modfind**() returns -1 and sets *errno* to indicate the error.

ERRORS

errno is set to the following if **modfind**() fails:

[EFAULT] The data required for this operation could not be read from the kernel space.

[ENOENT] The file specified is not loaded in the kernel.

SEE ALSO

kldfind(2), kldfirstmod(2), kldload(2), kldnext(2), kldstat(2), kldsym(2), kldunload(2), modfnext(2), modnext(2), modstat(2), kld(4), kldstat(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

modnext - return the modid of the next kernel module

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/module.h>

int
modnext(*int modid*);

int
modfnext(*int modid*);

DESCRIPTION

The **modnext**() system call returns the modid of the next kernel module (that is, the one after *modid*) or 0 if *modid* is the last module in the list.

If the *modid* value is 0, then **modnext**() will return the modid of the first module. The **modfnext**() system call must always be given a valid modid.

RETURN VALUES

The **modnext**() system call returns the modid of the next module (see *DESCRIPTION*) or 0. If an error occurs, *errno* is set to indicate the error.

ERRORS

The only error set by **modnext**() is ENOENT, which is set when *modid* refers to a kernel module that does not exist (is not loaded).

SEE ALSO

kldfind(2), kldfirstmod(2), kldload(2), kldnext(2), kldstat(2), kldsym(2), kldunload(2), modfind(2), modstat(2), kld(4), kldstat(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

modstat - get status of kernel module

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/module.h>

int
modstat(*int modid*, *struct module_stat *stat*);

DESCRIPTION

The **modstat**() system call writes the info for the kernel module referred to by *modid* into *stat*.

```
struct module_stat {
        int         version;        /* set to sizeof(module_stat) */
        char        name[MAXMODNAME];
        int         refs;
        int         id;
        modspecific_t data;
};
typedef union modspecific {
        int         intval;
        u_int       uintval;
        long        longval;
        u_long      ulongval;
} modspecific_t;
```

version — This field is set to the size of the structure mentioned above by the code calling **modstat**(), and not **modstat**() itself.

name — The name of the module referred to by *modid*.

refs — The number of modules referenced by *modid*.

id — The id of the module specified in *modid*.

data — Module specific data.

RETURN VALUES

The **modstat**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The information for the module referred to by *modid* is filled into the structure pointed to by *stat* unless:

[ENOENT] The module was not found (probably not loaded).

[EINVAL] The version specified in the *version* field of stat is not the proper version. You would need to rebuild world, the kernel, or your application, if this error occurs, given that you did properly fill in the *version* field.

[EFAULT] There was a problem copying one, some, or all of the fields into *stat* in the copyout(9) function.

SEE ALSO

kldfind(2), kldfirstmod(2), kldload(2), kldnext(2), kldstat(2), kldsym(2), kldunload(2), modfind(2), modfnext(2), modnext(2), kld(4), kldstat(8)

HISTORY

The **kld** interface first appeared in FreeBSD 3.0.

NAME

moncontrol, **monstartup** - control execution profile

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/gmon.h>

void
moncontrol(*int mode*);

void
monstartup(*u_long lowpc*, *u_long highpc*);

DESCRIPTION

An executable program compiled using the **-pg** option to cc(1) automatically includes calls to collect statistics for the gprof(1) call-graph execution profiler. In typical operation, profiling begins at program startup and ends when the program calls exit. When the program exits, the profiling data are written to the file *progname.gmon*, where progname is the name of the program, then gprof(1) can be used to examine the results.

The **moncontrol**() function selectively controls profiling within a program. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts use **moncontrol**(*0*); to resume the collection of histogram ticks and call counts use **moncontrol**(*1*). This feature allows the cost of particular operations to be measured. Note that an output file will be produced on program exit regardless of the state of **moncontrol**().

Programs that are not loaded with **-pg** may selectively collect profiling statistics by calling **monstartup**() with the range of addresses to be profiled. The *lowpc* and *highpc* arguments specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. Only functions in that range that have been compiled with the **-pg** option to cc(1) will appear in the call graph part of the output; however, all functions in that address range will have their execution time measured. Profiling begins on return from **monstartup**().

ENVIRONMENT

The following environment variables affect the execution of **moncontrol**:

PROFIL_USE_PID If set, the pid of the process is inserted into the filename.

FILES

progname.gmon execution data file

SEE ALSO

cc(1), gprof(1), profil(2), clocks(7)

NAME

mount, **nmount**, **unmount** - mount or dismount a file system

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/mount.h>

int
mount(*const char *type*, *const char *dir*, *int flags*, *void *data*);

int
unmount(*const char *dir*, *int flags*);

#include <sys/uio.h>

int
nmount(*struct iovec *iov*, *u_int niov*, *int flags*);

DESCRIPTION

The **mount**() system call grafts a file system object onto the system file tree at the point *dir*. The argument *data* describes the file system object to be mounted. The argument *type* tells the kernel how to interpret *data* (See *type* below). The contents of the file system become available through the new mount point *dir*. Any files in *dir* at the time of a successful mount are swept under the carpet so to speak, and are unavailable until the file system is unmounted.

The **nmount**() system call behaves similarly to **mount**(), except that the mount options (file system type name, device to mount, mount-point name, etc.) are passed as an array of name-value pairs in the array *iov*, containing *niov* elements. The following options are required by all file systems:

fstype	file system type name (e.g., "procfs")
fspath	mount point pathname (e.g., "/proc")

Depending on the file system type, other options may be recognized or required; for example, most disk-based file systems require a "from" option containing the pathname of a special device in addition to the options listed above.

By default only the super-user may call the **mount**() system call. This restriction can be removed by

setting the *vfs.usermount* sysctl(8) variable to a non-zero value; see the BUGS section for more information.

The following *flags* may be specified to suppress default semantics which affect file system access.

MNT_RDONLY The file system should be treated as read-only; even the super-user may not write on it. Specifying MNT_UPDATE without this option will upgrade a read-only file system to read/write.

MNT_NOEXEC Do not allow files to be executed from the file system.

MNT_NOSUID Do not honor setuid or setgid bits on files when executing them. This flag is set automatically when the caller is not the super-user.

MNT_NOATIME Disable update of file access times.

MNT_SNAPSHOT Create a snapshot of the file system. This is currently only supported on UFS2 file systems, see mksnap_ffs(8) for more information.

MNT_SUIDDIR Directories with the SUID bit set chown new files to their own owner. This flag requires the SUIDDIR option to have been compiled into the kernel to have any effect. See the mount(8) and chmod(2) pages for more information.

MNT_SYNCHRONOUS All I/O to the file system should be done synchronously.

MNT_ASYNC All I/O to the file system should be done asynchronously.

MNT_FORCE Force a read-write mount even if the file system appears to be unclean. Dangerous. Together with MNT_UPDATE and MNT_RDONLY, specify that the file system is to be forcibly downgraded to a read-only mount even if some files are open for writing.

MNT_NOCLUSTERR Disable read clustering.

MNT_NOCLUSTERW Disable write clustering.

The flag MNT_UPDATE indicates that the mount command is being applied to an already mounted file system. This allows the mount flags to be changed without requiring that the file system be unmounted and remounted. Some file systems may not allow all flags to be changed. For example, many file systems will not allow a change from read-write to read-only.

The flag MNT_RELOAD causes the vfs subsystem to update its data structures pertaining to the specified already mounted file system.

The *type* argument names the file system. The types of file systems known to the system can be obtained with lsvfs(1).

The *data* argument is a pointer to a structure that contains the type specific arguments to mount. The format for these argument structures is described in the manual page for each file system. By convention file system manual pages are named by prefixing ‘‘mount_’’ to the name of the file system as returned by lsvfs(1). Thus the NFS file system is described by the mount_nfs(8) manual page. It should be noted that a manual page for default file systems, known as UFS and UFS2, does not exist.

The **unmount**() system call disassociates the file system from the specified mount point *dir*.

The *flags* argument may include MNT_FORCE to specify that the file system should be forcibly unmounted even if files are still active. Active special devices continue to work, but any further accesses to any other active files result in errors even if the file system is later remounted.

If the MNT_BYFSID flag is specified, *dir* should instead be a file system ID encoded as "FSID:*val0*:*val1*", where *val0* and *val1* are the contents of the *fsid_t val[]* array in decimal. The file system that has the specified file system ID will be unmounted.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **mount**() and **nmount**() system calls will fail when one of the following occurs:

[EPERM] The caller is neither the super-user nor the owner of *dir*.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or the entire length of a path name exceeded 1023 characters.

[ELOOP] Too many symbolic links were encountered in translating a pathname.

[ENOENT] A component of *dir* does not exist.

[ENOTDIR] A component of *name* is not a directory, or a path prefix of *special* is not a

directory.

[EBUSY] Another process currently holds a reference to *dir*.

[EFAULT] The *dir* argument points outside the process's allocated address space.

The following errors can occur for a *ufs* file system mount:

[ENODEV] A component of ufs_args *fspec* does not exist.

[ENOTBLK] The *fspec* argument is not a block device.

[ENXIO] The major device number of *fspec* is out of range (this indicates no device driver exists for the associated hardware).

[EBUSY] *fspec* is already mounted.

[EMFILE] No space remains in the mount table.

[EINVAL] The super block for the file system had a bad magic number or an out of range block size.

[ENOMEM] Not enough memory was available to read the cylinder group information for the file system.

[EIO] An I/O error occurred while reading the super block or cylinder group information.

[EFAULT] The *fspec* argument points outside the process's allocated address space.

The following errors can occur for a *nfs* file system mount:

[ETIMEDOUT] *Nfs* timed out trying to contact the server.

[EFAULT] Some part of the information described by nfs_args points outside the process's allocated address space.

The **unmount**() system call may fail with one of the following errors:

[EPERM] The caller is neither the super-user nor the user who issued the corresponding

mount() call.

[ENAMETOOLONG]
The length of the path name exceeded 1023 characters.

[EINVAL] The requested directory is not in the mount table.

[ENOENT] The file system ID specified using MNT_BYFSID was not found in the mount table.

[EINVAL] The file system ID specified using MNT_BYFSID could not be decoded.

[EINVAL] The specified file system is the root file system.

[EBUSY] A process is holding a reference to a file located on the file system.

[EIO] An I/O error occurred while writing cached file system information.

[EFAULT] The *dir* argument points outside the process's allocated address space.

SEE ALSO

lsvfs(1), mksnap_ffs(8), mount(8), umount(8)

HISTORY

The **mount**() and **unmount**() functions appeared in Version 1 AT&T UNIX. The **nmount**() system call first appeared in FreeBSD 5.0.

BUGS

Some of the error codes need translation to more obvious messages.

Allowing untrusted users to mount arbitrary media, e.g. by enabling *vfs.usermount*, should not be considered safe. Most file systems in FreeBSD were not built to safeguard against malicious devices.

NAME

mprotect - control the protection of pages

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
mprotect(*void *addr*, *size_t len*, *int prot*);

DESCRIPTION

The **mprotect**() system call changes the specified pages to have protection *prot*. Not all implementations will guarantee protection on a page basis; the granularity of protection changes may be as large as an entire region. A region is the virtual address space defined by the start and end addresses of a *struct vm_map_entry*.

Currently these protection bits are known, which can be combined, OR'd together:

PROT_NONE	No permissions at all.
PROT_READ	The pages can be read.
PROT_WRITE	The pages can be written.
PROT_EXEC	The pages can be executed.

RETURN VALUES

The **mprotect**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **mprotect**() system call will fail if:

[EINVAL]	The virtual address range specified by the *addr* and *len* arguments is not valid.
[EACCES]	The calling process was not allowed to change the protection to the value specified by the *prot* argument.

SEE ALSO

madvise(2), mincore(2), msync(2), munmap(2)

HISTORY

The **mprotect**() system call first appeared in 4.4BSD.

NAME

msgctl - message control operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int
msgctl(*int msqid*, *int cmd*, *struct msqid_ds *buf*);

DESCRIPTION

The **msgctl**() system call performs some control operations on the message queue specified by *msqid*.

Each message queue has a data structure associated with it, parts of which may be altered by **msgctl**() and parts of which determine the actions of **msgctl**(). The data structure is defined in <*sys/msg.h*> and contains (amongst others) the following members:

```
struct msqid_ds {
        struct  ipc_perm msg_perm;      /* msg queue permission bits */
        struct  msg *__msg_first;  /* kernel data, don't use */
        struct  msg *__msg_last;   /* kernel data, don't use */
        msglen_t msg_cbytes;        /* number of bytes in use on the queue */
        msgqnum_t msg_qnum;         /* number of msgs in the queue */
        msglen_t msg_qbytes;        /* max # of bytes on the queue */
        pid_t   msg_lspid;          /* pid of last msgsnd() */
        pid_t   msg_lrpid;          /* pid of last msgrcv() */
        time_t  msg_stime;          /* time of last msgsnd() */
        time_t  msg_rtime;          /* time of last msgrcv() */
        time_t  msg_ctime;          /* time of last msgctl() */
};
```

The *ipc_perm* structure used inside the *msqid_ds* structure is defined in <*sys/ipc.h*> and looks like this:

```
struct ipc_perm {
        uid_t           cuid;   /* creator user id */
        gid_t           cgid;   /* creator group id */
```

```
        uid_t           uid;    /* user id */
        gid_t           gid;    /* group id */
        mode_t          mode;   /* r/w permission */
        unsigned short  seq;    /* sequence # (to generate unique ipcid) */
        key_t           key;    /* user specified msg/sem/shm key */
};
```

The operation to be performed by **msgctl**() is specified in *cmd* and is one of:

IPC_STAT Gather information about the message queue and place it in the structure pointed to by *buf*.

IPC_SET Set the value of the *msg_perm.uid*, *msg_perm.gid*, *msg_perm.mode* and *msg_qbytes* fields in the structure associated with *msqid*. The values are taken from the corresponding fields in the structure pointed to by *buf*. This operation can only be executed by the super-user, or a process that has an effective user id equal to either *msg_perm.cuid* or *msg_perm.uid* in the data structure associated with the message queue. The value of *msg_qbytes* can only be increased by the super-user. Values for *msg_qbytes* that exceed the system limit (MSGMNB from <*sys/msg.h*>) are silently truncated to that limit.

IPC_RMID Remove the message queue specified by *msqid* and destroy the data associated with it. Only the super-user or a process with an effective uid equal to the *msg_perm.cuid* or *msg_perm.uid* values in the data structure associated with the queue can do this.

The permission to read from or write to a message queue (see msgsnd(2) and msgrcv(2)) is determined by the *msg_perm.mode* field in the same way as is done with files (see chmod(2)), but the effective uid can match either the *msg_perm.cuid* field or the *msg_perm.uid* field, and the effective gid can match either *msg_perm.cgid* or *msg_perm.gid*.

RETURN VALUES

The **msgctl**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **msgctl**() function will fail if:

[EPERM] The *cmd* argument is equal to IPC_SET or IPC_RMID and the caller is not the super-user, nor does the effective uid match either the *msg_perm.uid* or *msg_perm.cuid* fields of the data structure associated with the message queue.

An attempt is made to increase the value of *msg_qbytes* through IPC_SET but the caller is not the super-user.

[EACCES] The command is IPC_STAT and the caller has no read permission for this message queue.

[EINVAL] The *msqid* argument is not a valid message queue identifier.

cmd is not a valid command.

[EFAULT] The *buf* argument specifies an invalid address.

SEE ALSO

msgget(2), msgrcv(2), msgsnd(2)

HISTORY

Message queues appeared in the first release of AT&T System V UNIX.

NAME

msgget - get message queue

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/msg.h>

int
msgget(*key_t key*, *int msgflg*);

DESCRIPTION

The **msgget**() function returns the message queue identifier associated with *key*. A message queue identifier is a unique integer greater than zero.

A message queue is created if either *key* is equal to IPC_PRIVATE, or *key* does not have a message queue identifier associated with it, and the IPC_CREAT bit is set in *msgflg*.

If a new message queue is created, the data structure associated with it (the *msqid_ds* structure, see msgctl(2)) is initialized as follows:

- *msg_perm.cuid* and *msg_perm.uid* are set to the effective uid of the calling process.
- *msg_perm.gid* and *msg_perm.cgid* are set to the effective gid of the calling process.
- *msg_perm.mode* is set to the lower 9 bits of *msgflg* which are set by ORing these constants:

  0400 Read access for user.

  0200 Write access for user.

  0040 Read access for group.

  0020 Write access for group.

  0004 Read access for other.

  0002 Write access for other.

- *msg_cbytes*, *msg_qnum*, *msg_lspid*, *msg_lrpid*, *msg_rtime*, and *msg_stime* are set to 0.
- *msg_qbytes* is set to the system wide maximum value for the number of bytes in a queue (MSGMNB).
- *msg_ctime* is set to the current time.

RETURN VALUES

Upon successful completion a positive message queue identifier is returned. Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EACCES] A message queue is already associated with *key* and the caller has no permission to access it.

[EEXIST] Both IPC_CREAT and IPC_EXCL are set in *msgflg*, and a message queue is already associated with *key*.

[ENOSPC] A new message queue could not be created because the system limit for the number of message queues has been reached.

[ENOENT] IPC_CREAT was not set in *msgflg* and no message queue associated with *key* was found.

SEE ALSO

msgctl(2), msgrcv(2), msgsnd(2)

HISTORY

Message queues appeared in the first release of AT&T System V UNIX.

NAME

msgrcv - receive a message from a message queue

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

ssize_t
msgrcv(*int msqid*, *void *msgp*, *size_t msgsz*, *long msgtyp*, *int msgflg*);

DESCRIPTION

The **msgrcv**() function receives a message from the message queue specified in *msqid*, and places it into the structure pointed to by *msgp*. This structure should consist of the following members:

```
long mtype;    /* message type */
char mtext[1]; /* body of message */
```

mtype is an integer greater than 0 that can be used for selecting messages, *mtext* is an array of bytes, with a size up to that of the system limit (MSGMAX).

The value of *msgtyp* has one of the following meanings:

- The *msgtyp* argument is greater than 0. The first message of type *msgtyp* will be received.
- The *msgtyp* argument is equal to 0. The first message on the queue will be received.
- The *msgtyp* argument is less than 0. The first message of the lowest message type that is less than or equal to the absolute value of *msgtyp* will be received.

The *msgsz* argument specifies the maximum length of the requested message. If the received message has a length greater than *msgsz* it will be silently truncated if the MSG_NOERROR flag is set in *msgflg*, otherwise an error will be returned.

If no matching message is present on the message queue specified by *msqid*, the behavior of **msgrcv**() depends on whether the IPC_NOWAIT flag is set in *msgflg* or not. If IPC_NOWAIT is set, **msgrcv**() will immediately return a value of -1, and set *errno* to ENOMSG. If IPC_NOWAIT is not set, the

calling process will be blocked until:

- A message of the requested type becomes available on the message queue.
- The message queue is removed, in which case -1 will be returned, and *errno* set to EINVAL.
- A signal is received and caught. -1 is returned, and *errno* set to EINTR.

If a message is successfully received, the data structure associated with *msqid* is updated as follows:

- *msg_cbytes* is decremented by the size of the message.
- *msg_lrpid* is set to the pid of the caller.
- *msg_lrtime* is set to the current time.
- *msg_qnum* is decremented by 1.

RETURN VALUES

Upon successful completion, **msgrcv**() returns the number of bytes received into the *mtext* field of the structure pointed to by *msgp*. Otherwise, -1 is returned, and *errno* set to indicate the error.

ERRORS

The **msgrcv**() function will fail if:

[EINVAL]	The *msqid* argument is not a valid message queue identifier.
	The message queue was removed while **msgrcv**() was waiting for a message of the requested type to become available on it.
	The *msgsz* argument is less than 0.
[E2BIG]	A matching message was received, but its size was greater than *msgsz* and the MSG_NOERROR flag was not set in *msgflg*.
[EACCES]	The calling process does not have read access to the message queue.
[EFAULT]	The *msgp* argument points to an invalid address.
[EINTR]	The system call was interrupted by the delivery of a signal.

[ENOMSG] There is no message of the requested type available on the message queue, and IPC_NOWAIT is set in *msgflg*.

SEE ALSO

msgctl(2), msgget(2), msgsnd(2)

HISTORY

Message queues appeared in the first release of AT&T System V UNIX.

NAME

msgsnd - send a message to a message queue

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int
msgsnd(*int msqid*, *const void *msgp*, *size_t msgsz*, *int msgflg*);

DESCRIPTION

The **msgsnd**() function sends a message to the message queue specified in *msqid*. The *msgp* argument points to a structure containing the message. This structure should consist of the following members:

```
long mtype;    /* message type */
char mtext[1]; /* body of message */
```

mtype is an integer greater than 0 that can be used for selecting messages (see msgrcv(2)), *mtext* is an array of *msgsz* bytes. The argument *msgsz* can range from 0 to a system-imposed maximum, MSGMAX.

If the number of bytes already on the message queue plus *msgsz* is bigger than the maximum number of bytes on the message queue (*msg_qbytes*, see msgctl(2)), or the number of messages on all queues system-wide is already equal to the system limit, *msgflg* determines the action of **msgsnd**(). If *msgflg* has IPC_NOWAIT mask set in it, the call will return immediately. If *msgflg* does not have IPC_NOWAIT set in it, the call will block until:

- The condition which caused the call to block does no longer exist. The message will be sent.
- The message queue is removed, in which case -1 will be returned, and *errno* is set to EINVAL.
- The caller catches a signal. The call returns with *errno* set to EINTR.

After a successful call, the data structure associated with the message queue is updated in the following way:

- *msg_cbytes* is incremented by the size of the message.
- *msg_qnum* is incremented by 1.
- *msg_lspid* is set to the pid of the calling process.
- *msg_stime* is set to the current time.

RETURN VALUES

The **msgsnd**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **msgsnd**() function will fail if:

[EINVAL] The *msqid* argument is not a valid message queue identifier.

The message queue was removed while **msgsnd**() was waiting for a resource to become available in order to deliver the message.

The *msgsz* argument is greater than *msg_qbytes*.

The *mtype* argument is not greater than 0.

[EACCES] The calling process does not have write access to the message queue.

[EAGAIN] There was no space for this message either on the queue, or in the whole system, and IPC_NOWAIT was set in *msgflg*.

[EFAULT] The *msgp* argument points to an invalid address.

[EINTR] The system call was interrupted by the delivery of a signal.

HISTORY

Message queues appeared in the first release of AT&T Unix System V.

BUGS

NetBSD and FreeBSD do not define the EIDRM error value, which should be used in the case of a removed message queue.

NAME

msync - synchronize a mapped region

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
msync(*void *addr*, *size_t len*, *int flags*);

DESCRIPTION

The **msync**() system call writes any modified pages back to the file system and updates the file modification time. If *len* is 0, all modified pages within the region containing *addr* will be flushed; if *len* is non-zero, only those pages containing *addr* and *len-1* succeeding locations will be examined. The *flags* argument may be specified as follows:

MS_ASYNC	Return immediately
MS_SYNC	Perform synchronous writes
MS_INVALIDATE	Invalidate all cached data

RETURN VALUES

The **msync**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **msync**() system call will fail if:

[EBUSY]	Some or all of the pages in the specified region are locked and MS_INVALIDATE is specified.
[EINVAL]	The *addr* argument is not a multiple of the hardware page size.
[ENOMEM]	The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.
[EINVAL]	The *flags* argument was both MS_ASYNC and MS_INVALIDATE. Only one of these flags is allowed.

[EIO] An error occurred while writing at least one of the pages in the specified region.

SEE ALSO

madvise(2), mincore(2), mlock(2), mprotect(2), munmap(2)

HISTORY

The **msync**() system call first appeared in 4.4BSD.

BUGS

The **msync**() system call is usually not needed since BSD implements a coherent file system buffer cache. However, it may be used to associate dirty VM pages with file system buffers and thus cause them to be flushed to physical media sooner rather than later.

NAME

multibyte - multibyte and wide character manipulation functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <limits.h>
#include <stdlib.h>
#include <wchar.h>

DESCRIPTION

The basic elements of some written natural languages, such as Chinese, cannot be represented uniquely with single C *char*s. The C standard supports two different ways of dealing with extended natural language encodings: wide characters and multibyte characters. Wide characters are an internal representation which allows each basic element to map to a single object of type *wchar_t*. Multibyte characters are used for input and output and code each basic element as a sequence of C *char*s. Individual basic elements may map into one or more (up to MB_LEN_MAX) bytes in a multibyte character.

The current locale (setlocale(3)) governs the interpretation of wide and multibyte characters. The locale category LC_CTYPE specifically controls this interpretation. The *wchar_t* type is wide enough to hold the largest value in the wide character representations for all locales.

Multibyte strings may contain ‘shift’ indicators to switch to and from particular modes within the given representation. If explicit bytes are used to signal shifting, these are not recognized as separate characters but are lumped with a neighboring character. There is always a distinguished ‘initial’ shift state. Some functions (e.g., mblen(3), mbtowc(3) and wctomb(3)) maintain static shift state internally, whereas others store it in an *mbstate_t* object passed by the caller. Shift states are undefined after a call to setlocale(3) with the LC_CTYPE or LC_ALL categories.

For convenience in processing, the wide character with value 0 (the null wide character) is recognized as the wide character string terminator, and the character with value 0 (the null byte) is recognized as the multibyte character string terminator. Null bytes are not permitted within multibyte characters.

The C library provides the following functions for dealing with multibyte characters:

Function	Description
mblen(3)	get number of bytes in a character
mbrlen(3)	get number of bytes in a character (restartable)

mbrtowc(3) convert a character to a wide-character code (restartable)
mbsrtowcs(3)
convert a character string to a wide-character string (restartable)
mbstowcs(3) convert a character string to a wide-character string
mbtowc(3) convert a character to a wide-character code
wcrtomb(3) convert a wide-character code to a character (restartable)
wcstombs(3) convert a wide-character string to a character string
wcsrtombs(3)
convert a wide-character string to a character string (restartable)
wctomb(3) convert a wide-character code to a character

SEE ALSO

mklocale(1), setlocale(3), stdio(3), big5(5), euc(5), gb18030(5), gb2312(5), gbk(5), mskanji(5), utf8(5)

STANDARDS

These functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

munmap - remove a mapping

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/mman.h>

int
munmap(*void *addr*, *size_t len*);

DESCRIPTION

The **munmap**() system call deletes the mappings and guards for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

RETURN VALUES

The **munmap**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **munmap**() system call will fail if:

[EINVAL] The *addr* argument was not page aligned, the *len* argument was zero or negative, or some part of the region being unmapped is outside the valid address range for a process.

SEE ALSO

madvise(2), mincore(2), mmap(2), mprotect(2), msync(2), getpagesize(3)

HISTORY

The **munmap**() system call first appeared in 4.4BSD.

NAME

nanosleep - high resolution sleep

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

int
clock_nanosleep(*clockid_t clock_id*, *int flags*, *const struct timespec *rqtp*, *struct timespec *rmtp*);

int
nanosleep(*const struct timespec *rqtp*, *struct timespec *rmtp*);

DESCRIPTION

If the TIMER_ABSTIME flag is not set in the *flags* argument, then **clock_nanosleep**() suspends execution of the calling thread until either the time interval specified by the *rqtp* argument has elapsed, or a signal is delivered to the calling process and its action is to invoke a signal-catching function or to terminate the process. The clock used to measure the time is specified by the *clock_id* argument.

If the TIMER_ABSTIME flag is set in the *flags* argument, then **clock_nanosleep**() suspends execution of the calling thread until either the value of the clock specified by the *clock_id* argument reaches the absolute time specified by the *rqtp* argument, or a signal is delivered to the calling process and its action is to invoke a signal-catching function or to terminate the process. If, at the time of the call, the time value specified by *rqtp* is less than or equal to the time value of the specified clock, then **clock_nanosleep**() returns immediately and the calling thread is not suspended.

The suspension time may be longer than requested due to the scheduling of other activity by the system. An unmasked signal will terminate the sleep early, regardless of the SA_RESTART value on the interrupting signal. The *rqtp* and *rmtp* arguments can point to the same object.

The following *clock_id* values are supported:

CLOCK_MONOTONIC
CLOCK_MONOTONIC_FAST
CLOCK_MONOTONIC_PRECISE
CLOCK_REALTIME
CLOCK_REALTIME_FAST
CLOCK_REALTIME_PRECISE

CLOCK_SECOND
CLOCK_UPTIME
CLOCK_UPTIME_FAST
CLOCK_UPTIME_PRECISE

The **nanosleep**() function behaves like **clock_nanosleep**() with a *clock_id* argument of CLOCK_REALTIME and without the TIMER_ABSTIME flag in the *flags* argument.

RETURN VALUES

These functions return zero when the requested time has elapsed.

If these functions return for any other reason, then **clock_nanosleep**() will directly return the error number, and **nanosleep**() will return -1 with the global variable *errno* set to indicate the error. If a relative sleep is interrupted by a signal and *rmtp* is non-NULL, the timespec structure it references is updated to contain the unslept amount (the request time minus the time actually slept).

ERRORS

These functions can fail with the following errors.

[EFAULT] Either *rqtp* or *rmtp* points to memory that is not a valid part of the process address space.

[EINTR] The function was interrupted by the delivery of a signal.

[EINVAL] The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

[EINVAL] The *flags* argument contained an invalid flag.

[EINVAL] The *clock_id* argument was CLOCK_THREAD_CPUTIME_ID or an unrecognized value.

[ENOTSUP] The *clock_id* argument was valid but not supported by this implementation of **clock_nanosleep**().

SEE ALSO

clock_gettime(2), sigaction(2), sleep(3)

STANDARDS

These functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

newlocale - Creates a new locale

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <locale.h>

locale_t
newlocale(*int mask*, *const char * locale*, *locale_t base*);

DESCRIPTION

Creates a new locale, inheriting some properties from an existing locale. The *mask* defines the components that the new locale will have set to the locale with the name specified in the *locale* parameter. Any other components will be inherited from *base*. The *mask* is either *LC_ALL_MASK*, indicating all possible locale components, or the logical OR of some combination of the following:

LC_COLLATE_MASK The locale for string collation routines. This controls alphabetic ordering in strcoll(3) and strxfrm(3).

LC_CTYPE_MASK The locale for the ctype(3) and multibyte(3) functions. This controls recognition of upper and lower case, alphabetic or non-alphabetic characters, and so on.

LC_MESSAGES_MASK Set a locale for message catalogs, see catopen(3) function.

LC_MONETARY_MASK
Set a locale for formatting monetary values; this affects the localeconv(3) function.

LC_NUMERIC_MASK Set a locale for formatting numbers. This controls the formatting of decimal points in input and output of floating point numbers in functions such as printf(3) and scanf(3), as well as values returned by localeconv(3).

LC_TIME_MASK Set a locale for formatting dates and times using the strftime(3) function.

This function uses the same rules for loading locale components as setlocale(3).

RETURN VALUES

Returns a new, valid, *locale_t* or NULL if an error occurs. You must free the returned locale with freelocale(3).

SEE ALSO

duplocale(3), freelocale(3), localeconv(3), querylocale(3), uselocale(3), xlocale(3)

STANDARDS

This function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

nextwctype - iterate through character classes

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wctype.h>

wint_t
nextwctype(*wint_t ch*, *wctype_t wct*);

DESCRIPTION

The **nextwctype**() function determines the next character after *ch* that is a member of character class *wct*. If *ch* is -1, the search begins at the first member of *wct*.

RETURN VALUES

The **nextwctype**() function returns the next character, or -1 if there are no more.

COMPATIBILITY

This function is a non-standard FreeBSD extension and should not be used where the standard **iswctype**() function would suffice.

SEE ALSO

wctype(3)

HISTORY

The **nextwctype**() function appeared in FreeBSD 5.4.

NAME

nfssvc - NFS services

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/param.h>
#include <sys/mount.h>
#include <sys/time.h>
#include <nfs/rpcv2.h>
#include <nfsserver/nfs.h>
#include <unistd.h>

int
nfssvc(int flags, void *argstructp);
```

DESCRIPTION

The **nfssvc**() system call is used by the NFS daemons to pass information into and out of the kernel and also to enter the kernel as a server daemon. The *flags* argument consists of several bits that show what action is to be taken once in the kernel and the *argstructp* points to one of three structures depending on which bits are set in flags.

On the client side, nfsiod(8) calls **nfssvc**() with the *flags* argument set to NFSSVC_BIOD and *argstructp* set to NULL to enter the kernel as a block I/O server daemon. For NQNFS, mount_nfs(8) calls **nfssvc**() with the NFSSVC_MNTD flag, optionally or'd with the flags NFSSVC_GOTAUTH and NFSSVC_AUTHINFAIL along with a pointer to a

```
struct nfsd_cargs {
        char            *ncd_dirp;      /* Mount dir path */
        uid_t           ncd_authuid;    /* Effective uid */
        int             ncd_authtype;   /* Type of authenticator */
        int             ncd_authlen;    /* Length of authenticator string */
        u_char          *ncd_authstr;   /* Authenticator string */
        int             ncd_verflen;    /* and the verifier */
        u_char          *ncd_verfstr;
        NFSKERBKEY_T ncd_key; /* Session key */
};
```

structure. The initial call has only the NFSSVC_MNTD flag set to specify service for the mount point.

If the mount point is using Kerberos, then the mount_nfs(8) utility will return from **nfssvc**() with *errno* == ENEEDAUTH whenever the client side requires an ‘‘rcmd’’ authentication ticket for the user. The mount_nfs(8) utility will attempt to get the Kerberos ticket, and if successful will call **nfssvc**() with the flags NFSSVC_MNTD and NFSSVC_GOTAUTH after filling the ticket into the ncd_authstr field and setting the ncd_authlen and ncd_authtype fields of the nfsd_cargs structure. If mount_nfs(8) failed to get the ticket, **nfssvc**() will be called with the flags NFSSVC_MNTD, NFSSVC_GOTAUTH and NFSSVC_AUTHINFAIL to denote a failed authentication attempt.

On the server side, **nfssvc**() is called with the flag NFSSVC_NFSD and a pointer to a

```
struct nfsd_srvargs {
        struct nfsd         *nsd_nfsd;      /* Pointer to in kernel nfsd struct */
        uid_t               nsd_uid;  /* Effective uid mapped to cred */
        uint32_t   nsd_haddr;         /* Ip address of client */
        struct ucred        nsd_cr;         /* Cred. uid maps to */
        int                 nsd_authlen;    /* Length of auth string (ret) */
        u_char              *nsd_authstr;   /* Auth string (ret) */
        int                 nsd_verflen;    /* and the verifier */
        u_char              *nsd_verfstr;
        struct timeval      nsd_timestamp;  /* timestamp from verifier */
        uint32_t   nsd_ttl;   /* credential ttl (sec) */
        NFSKERBKEY_T nsd_key; /* Session key */
};
```

to enter the kernel as an nfsd(8) daemon. Whenever an nfsd(8) daemon receives a Kerberos authentication ticket, it will return from **nfssvc**() with *errno* == ENEEDAUTH. The nfsd(8) utility will attempt to authenticate the ticket and generate a set of credentials on the server for the ‘‘user id’’ specified in the field nsd_uid. This is done by first authenticating the Kerberos ticket and then mapping the Kerberos principal to a local name and getting a set of credentials for that user via getpwnam(3) and getgrouplist(3). If successful, the nfsd(8) utility will call **nfssvc**() with the NFSSVC_NFSD and NFSSVC_AUTHIN flags set to pass the credential mapping in nsd_cr into the kernel to be cached on the server socket for that client. If the authentication failed, nfsd(8) calls **nfssvc**() with the flags NFSSVC_NFSD and NFSSVC_AUTHINFAIL to denote an authentication failure.

The master nfsd(8) server daemon calls **nfssvc**() with the flag NFSSVC_ADDSOCK and a pointer to a

```
struct nfsd_args {
        int         sock;     /* Socket to serve */
        caddr_t     name;     /* Client address for connection based sockets */
        int         namelen;/* Length of name */
```

};

to pass a server side NFS socket into the kernel for servicing by the nfsd(8) daemons.

RETURN VALUES

Normally **nfssvc**() does not return unless the server is terminated by a signal when a value of 0 is returned. Otherwise, -1 is returned and the global variable *errno* is set to specify the error.

ERRORS

[ENEEDAUTH] This special error value is really used for authentication support, particularly Kerberos, as explained above.

[EPERM] The caller is not the super-user.

SEE ALSO

mount_nfs(8), nfsd(8), nfsiod(8)

HISTORY

The **nfssvc**() system call first appeared in 4.4BSD.

BUGS

The **nfssvc**() system call is designed specifically for the NFS support daemons and as such is specific to their requirements. It should really return values to indicate the need for authentication support, since ENEEDAUTH is not really an error. Several fields of the argument structures are assumed to be valid and sometimes to be unchanged from a previous call, such that **nfssvc**() must be used with extreme care.

NAME

nice - set program scheduling priority

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
nice(*int incr*);

DESCRIPTION

This interface is obsoleted by setpriority(2).

The **nice**() function adds *incr* to the scheduling priority of the process. The priority is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favorable scheduling. Only the super-user may lower priorities.

Children inherit the priority of their parent processes via fork(2).

RETURN VALUES

Upon successful completion, **nice**() returns 0, and *errno* is unchanged. Otherwise, -1 is returned, the process' nice value is not changed, and *errno* is set to indicate the error.

ERRORS

The **nice**() function will fail if:

[EPERM] The *incr* argument is negative and the caller does not have appropriate privileges.

SEE ALSO

nice(1), fork(2), setpriority(2), renice(8)

STANDARDS

The **nice**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1") except for the return value. This implementation returns 0 upon successful completion but the standard requires returning the new nice value, which could be -1.

HISTORY

A **nice**() syscall appeared in Version 6 AT&T UNIX.

NAME

nl_langinfo - language information

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <langinfo.h>

*char **
nl_langinfo(*nl_item item*);

*char **
nl_langinfo_l(*nl_item item*, *locale_t loc*);

DESCRIPTION

The **nl_langinfo**() function returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program or thread's locale, or in the case of **nl_langinfo_l**(), the locale passed as the second argument. The manifest constant names and values of *item* are defined in <*langinfo.h*>.

Calls to **setlocale**() with a category corresponding to the category of *item*, or to the category LC_ALL, may overwrite the buffer pointed to by the return value.

RETURN VALUES

In a locale where langinfo data is not defined, **nl_langinfo**() returns a pointer to the corresponding string in the POSIX locale. **nl_langinfo_l**() returns the same values as **nl_langinfo**(). In all locales, **nl_langinfo**() returns a pointer to an empty string if *item* contains an invalid setting.

EXAMPLES

For example:

```
nl_langinfo(ABDAY_1)
```

would return a pointer to the string "Dom" if the identified language was Portuguese, and "Sun" if the identified language was English.

SEE ALSO

setlocale(3)

STANDARDS

The **nl_langinfo**() function conforms to Version 2 of the Single UNIX Specification ("SUSv2"). The **nl_langinfo_l**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **nl_langinfo**() function first appeared in FreeBSD 4.6.

NAME

nlist - retrieve symbol table name list from an executable file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <nlist.h>

int
nlist(*const char *filename*, *struct nlist *nl*);

DESCRIPTION

The **nlist**() function retrieves name list entries from the symbol table of an executable file (see a.out(5)). The argument *nl* is set to reference the beginning of the list. The list is preened of binary and invalid data; if an entry in the name list is valid, the *n_type* and *n_value* for the entry are copied into the list referenced by *nl*. No other data is copied. The last entry in the list is always NULL.

RETURN VALUES

The number of invalid entries is returned if successful; otherwise, if the file *filename* does not exist or is not executable, the returned value is -1.

SEE ALSO

a.out(5)

HISTORY

A **nlist**() function appeared in Version 6 AT&T UNIX.

NAME

nsdispatch - name-service switch dispatcher routine

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <stdarg.h>
#include <nsswitch.h>

int
nsdispatch(void *retval, const ns_dtab dtab[], const char *database, const char *method_name,
    const ns_src defaults[], ...);
```

DESCRIPTION

The **nsdispatch**() function invokes the methods specified in *dtab* in the order given by nsswitch.conf(5) for the database *database* until a successful entry is found.

retval is passed to each method to modify as necessary, to pass back results to the caller of **nsdispatch**().

Each method has the function signature described by the typedef:

typedef int **(*nss_method)**(*void *retval*, *void *mdata*, *va_list *ap*);

dtab is an array of *ns_dtab* structures, which have the following format:

```
typedef struct _ns_dtab {
        const char      *src;
        nss_method      method;
        void            *mdata;
} ns_dtab;
```

The *dtab* array should consist of one entry for each source type that is implemented, with *src* as the name of the source, *method* as a function which handles that source, and *mdata* as a handle on arbitrary data to be passed to the method. The last entry in *dtab* should contain NULL values for *src*, *method*, and *mdata*.

Additionally, methods may be implemented in NSS modules, in which case they are selected using the *database* and *method_name* arguments along with the configured source. Modules must use source names different from the built-in ones.

defaults contains a list of default sources to try if nsswitch.conf(5) is missing or corrupted, or if there is no relevant entry for *database*. It is an array of *ns_src* structures, which have the following format:

```
typedef struct _ns_src {
        const char      *src;
        uint32_t    flags;
} ns_src;
```

The *defaults* array should consist of one entry for each source to be configured by default indicated by *src*, and *flags* set to the criterion desired (usually NS_SUCCESS; refer to *Method return values* for more information). The last entry in *defaults* should have *src* set to NULL and *flags* set to 0.

For convenience, a global variable defined as:

```
extern const ns_src __nsdefaultsrc[];
```

exists which contains a single default entry for the source ‘files’ that may be used by callers which do not require complicated default rules.

‘...’ are optional extra arguments, which are passed to the appropriate method as a variable argument list of the type *va_list*.

Valid source types

While there is support for arbitrary sources, the following #defines for commonly implemented sources are available:

#define	value
NSSRC_FILES	"files"
NSSRC_DB	"db"
NSSRC_DNS	"dns"
NSSRC_NIS	"nis"
NSSRC_COMPAT	"compat"

Refer to nsswitch.conf(5) for a complete description of what each source type is.

Method return values

The *nss_method* functions must return one of the following values depending upon status of the lookup:

Return value	Status code
NS_SUCCESS	

success
NS_NOTFOUND
notfound
NS_UNAVAIL
unavail
NS_TRYAGAIN
tryagain
NS_RETURN -none-

Refer to nsswitch.conf(5) for a complete description of each status code.

The **nsdispatch**() function returns the value of the method that caused the dispatcher to terminate, or NS_NOTFOUND otherwise.

NOTES

FreeBSD's Standard C Library (libc, -lc) provides stubs for compatibility with NSS modules written for the GNU C Library **nsswitch** interface. However, these stubs only support the use of the "passwd" and "group" databases.

SEE ALSO

hesiod(3), stdarg(3), nsswitch.conf(5), yp(8)

HISTORY

The **nsdispatch**() function first appeared in FreeBSD 5.0. It was imported from the NetBSD Project, where it appeared first in NetBSD 1.4. Support for NSS modules first appeared in FreeBSD 5.1.

AUTHORS

Luke Mewburn <*lukem@netbsd.org*> wrote this freely-distributable name-service switch implementation, using ideas from the ULTRIX svc.conf(5) and Solaris nsswitch.conf(4) manual pages. The FreeBSD Project added the support for threads and NSS modules, and normalized the uses of **nsdispatch**() within the standard C library.

NAME

ntp_adjtime, **ntp_gettime** - Network Time Protocol (NTP) daemon interface system calls

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/timex.h>

int
ntp_adjtime(*struct timex* *);

int
ntp_gettime(*struct ntptimeval* *);

DESCRIPTION

The two system calls **ntp_adjtime**() and **ntp_gettime**() are the kernel interface to the Network Time Protocol (NTP) daemon ntpd(8).

The **ntp_adjtime**() function is used by the NTP daemon to adjust the system clock to an externally derived time. The time offset and related variables which are set by **ntp_adjtime**() are used by **hardclock**() to adjust the phase and frequency of the phase- or frequency-lock loop (PLL resp. FLL) which controls the system clock.

The **ntp_gettime**() function provides the time, maximum error (sync distance) and estimated error (dispersion) to client user application programs.

In the following, all variables that refer PPS are only relevant if the *PPS_SYNC* option is enabled in the kernel.

ntp_adjtime() has as argument a *struct timex* * of the following form:

```
struct timex {
        unsigned int modes;     /* clock mode bits (wo) */
        long offset;            /* time offset (us) (rw) */
        long freq;      /* frequency offset (scaled ppm) (rw) */
        long maxerror;          /* maximum error (us) (rw) */
        long esterror;          /* estimated error (us) (rw) */
        int status;     /* clock status bits (rw) */
        long constant;          /* pll time constant (rw) */
```

```
        long precision;         /* clock precision (us) (ro) */
        long tolerance;         /* clock frequency tolerance (scaled
                                 * ppm) (ro) */
        /*
         * The following read-only structure members are implemented
         * only if the PPS signal discipline is configured in the
         * kernel.
         */
        long ppsfreq;           /* pps frequency (scaled ppm) (ro) */
        long jitter;            /* pps jitter (us) (ro) */
        int shift;          /* interval duration (s) (shift) (ro) */
        long stabil;            /* pps stability (scaled ppm) (ro) */
        long jitcnt;            /* jitter limit exceeded (ro) */
        long calcnt;            /* calibration intervals (ro) */
        long errcnt;            /* calibration errors (ro) */
        long stbcnt;            /* stability limit exceeded (ro) */
};
```

The members of this struct have the following meanings when used as argument for **ntp_adjtime**():

modes Defines what settings should be changed with the current **ntp_adjtime**() call (write-only). Bitwise OR of the following:

MOD_OFFSET	set time offset
MOD_FREQUENCY	set frequency offset
MOD_MAXERROR	set maximum time error
MOD_ESTERROR	set estimated time error
MOD_STATUS	set clock status bits
MOD_TIMECONST	set PLL time constant
MOD_CLKA	set clock A
MOD_CLKB	set clock B

offset Time offset (in microseconds), used by the PLL/FLL to adjust the system time in small increments (read-write).

freq Frequency offset (scaled ppm) (read-write).

maxerror Maximum error (in microseconds). Initialized by an **ntp_adjtime**() call, and increased by the kernel once each second to reflect the maximum error bound growth (read-write).

esterror Estimated error (in microseconds). Set and read by **ntp_adjtime**(), but unused by the kernel (read-write).

status System clock status bits (read-write). Bitwise OR of the following:

STA_PLL	Enable PLL updates (read-write).
STA_PPSFREQ	Enable PPS freq discipline (read-write).

STA_PPSTIME Enable PPS time discipline (read-write).
STA_FLL Select frequency-lock mode (read-write).
STA_INS Insert leap (read-write).
STA_DEL Delete leap (read-write).
STA_UNSYNC Clock unsynchronized (read-write).
STA_FREQHOLD
Hold frequency (read-write).
STA_PPSSIGNAL
PPS signal present (read-only).
STA_PPSJITTER PPS signal jitter exceeded (read-only).
STA_PPSWANDER
PPS signal wander exceeded (read-only).
STA_PPSERROR
PPS signal calibration error (read-only).
STA_CLOCKERR
Clock hardware fault (read-only).

constant PLL time constant, determines the bandwidth, or "stiffness", of the PLL (read-write).

precision
Clock precision (in microseconds). In most cases the same as the kernel tick variable (see hz(9)). If a precision clock counter or external time-keeping signal is available, it could be much lower (and depend on the state of the signal) (read-only).

tolerance Maximum frequency error, or tolerance of the CPU clock oscillator (scaled ppm). Ordinarily a property of the architecture, but could change under the influence of external time-keeping signals (read-only).

ppsfreq PPS frequency offset produced by the frequency median filter (scaled ppm) (read-only).

jitter PPS jitter measured by the time median filter in microseconds (read-only).

shift Logarithm to base 2 of the interval duration in seconds (PPS, read-only).

stabil PPS stability (scaled ppm); dispersion (wander) measured by the frequency median filter (read-only).

jitcnt Number of seconds that have been discarded because the jitter measured by the time median filter exceeded the limit *MAXTIME* (PPS, read-only).

calcnt Count of calibration intervals (PPS, read-only).

errcnt Number of calibration intervals that have been discarded because the wander exceeded the limit *MAXFREQ* or where the calibration interval jitter exceeded two ticks (PPS, read-only).

stbcnt Number of calibration intervals that have been discarded because the frequency wander exceeded the limit *MAXFREQ*/4 (PPS, read-only).

After the **ntp_adjtime**() call, the *struct timex* * structure contains the current values of the corresponding variables.

ntp_gettime() has as argument a *struct ntptimeval* * with the following members:

```
struct ntptimeval {
        struct timeval time; /* current time (ro) */
        long maxerror;          /* maximum error (us) (ro) */
        long esterror;          /* estimated error (us) (ro) */
};
```

These have the following meaning:

time Current time (read-only).
maxerror Maximum error in microseconds (read-only).
esterror Estimated error in microseconds (read-only).

RETURN VALUES

ntp_adjtime() and **ntp_gettime**() return the current state of the clock on success, or any of the errors of copyin(9) and copyout(9). **ntp_adjtime**() may additionally return EPERM if the user calling **ntp_adjtime**() does not have sufficient permissions.

Possible states of the clock are:

TIME_OK	Everything okay, no leap second warning.
TIME_INS	"insert leap second" warning. At the end of the day, a leap second will be inserted after 23:59:59.
TIME_DEL	"delete leap second" warning. At the end of the day, second 23:59:59 will be skipped.
TIME_OOP	Leap second in progress.
TIME_WAIT	Leap second has occurred within the last few seconds.
TIME_ERROR	Clock not synchronized.

ERRORS

The **ntp_adjtime**() system call may return EPERM if the caller does not have sufficient permissions.

SEE ALSO

options(4), ntpd(8), hardclock(9), hz(9)

http://www.bipm.fr/enus/5_Scientific/c_time/time_1.html

http://www.boulder.nist.gov/timefreq/general/faq.htm

ftp://time.nist.gov/pub/leap-seconds.list

BUGS

Take note that this API is extremely complex and stateful. Users should not attempt modification

without first reviewing the ntpd(8) sources in depth.

NAME

vis, **nvis**, **strvis**, **stravis**, **strnvis**, **strvisx**, **strnvisx**, **strenvisx**, **svis**, **snvis**, **strsvis**, **strsnvis**, **strsvisx**, **strsnvisx**, **strsenvisx** - visually encode characters

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <vis.h>

*char **
vis(*char *dst*, *int c*, *int flag*, *int nextc*);

*char **
nvis(*char *dst*, *size_t dlen*, *int c*, *int flag*, *int nextc*);

int
strvis(*char *dst*, *const char *src*, *int flag*);

int
stravis(*char **dst*, *const char *src*, *int flag*);

int
strnvis(*char *dst*, *size_t dlen*, *const char *src*, *int flag*);

int
strvisx(*char *dst*, *const char *src*, *size_t len*, *int flag*);

int
strnvisx(*char *dst*, *size_t dlen*, *const char *src*, *size_t len*, *int flag*);

int
strenvisx(*char *dst*, *size_t dlen*, *const char *src*, *size_t len*, *int flag*, *int *cerr_ptr*);

*char **
svis(*char *dst*, *int c*, *int flag*, *int nextc*, *const char *extra*);

*char **
snvis(*char *dst*, *size_t dlen*, *int c*, *int flag*, *int nextc*, *const char *extra*);

int
strsvis(*char *dst*, *const char *src*, *int flag*, *const char *extra*);

int
strsnvis(*char *dst*, *size_t dlen*, *const char *src*, *int flag*, *const char *extra*);

int
strsvisx(*char *dst*, *const char *src*, *size_t len*, *int flag*, *const char *extra*);

int
strsnvisx(*char *dst*, *size_t dlen*, *const char *src*, *size_t len*, *int flag*, *const char *extra*);

int
strsenvisx(*char *dst*, *size_t dlen*, *const char *src*, *size_t len*, *int flag*, *const char *extra*, *int *cerr_ptr*);

DESCRIPTION

The **vis**() function copies into *dst* a string which represents the character *c*. If *c* needs no encoding, it is copied in unaltered. The string is null terminated, and a pointer to the end of the string is returned. The maximum length of any encoding is four bytes (not including the trailing NUL); thus, when encoding a set of characters into a buffer, the size of the buffer should be four times the number of bytes encoded, plus one for the trailing NUL. The flag parameter is used for altering the default range of characters considered for encoding and for altering the visual representation. The additional character, *nextc*, is only used when selecting the VIS_CSTYLE encoding format (explained below).

The **strvis**(), **stravis**(), **strnvis**(), **strvisx**(), and **strnvisx**() functions copy into *dst* a visual representation of the string *src*. The **strvis**() and **strnvis**() functions encode characters from *src* up to the first NUL. The **strvisx**() and **strnvisx**() functions encode exactly *len* characters from *src* (this is useful for encoding a block of data that may contain NUL's). Both forms NUL terminate *dst*. The size of *dst* must be four times the number of bytes encoded from *src* (plus one for the NUL). Both forms return the number of characters in *dst* (not including the trailing NUL). The **stravis**() function allocates space dynamically to hold the string. The "**n**" versions of the functions also take an additional argument *dlen* that indicates the length of the *dst* buffer. If *dlen* is not large enough to fit the converted string then the **strnvis**() and **strnvisx**() functions return -1 and set *errno* to ENOSPC. The **strenvisx**() function takes an additional argument, *cerr_ptr*, that is used to pass in and out a multibyte conversion error flag. This is useful when processing single characters at a time when it is possible that the locale may be set to something other than the locale of the characters in the input data.

The functions **svis**(), **snvis**(), **strsvis**(), **strsnvis**(), **strsvisx**(), **strsnvisx**(), and **strsenvisx**() correspond to **vis**(), **nvis**(), **strvis**(), **strnvis**(), **strvisx**(), **strnvisx**(), and **strenvisx**() but have an additional argument *extra*, pointing to a NUL terminated list of characters. These characters will be copied encoded or

backslash-escaped into *dst*. These functions are useful e.g. to remove the special meaning of certain characters to shells.

The encoding is a unique, invertible representation composed entirely of graphic characters; it can be decoded back into the original form using the unvis(3), strunvis(3) or strnunvis(3) functions.

There are two parameters that can be controlled: the range of characters that are encoded (applies only to **vis**(), **nvis**(), **strvis**(), **strnvis**(), **strvisx**(), and **strnvisx**()), and the type of representation used. By default, all non-graphic characters, except space, tab, and newline are encoded (see isgraph(3)). The following flags alter this:

VIS_DQ Also encode double quotes

VIS_GLOB Also encode the magic characters (‘*’, ‘?’, ‘[’, and ‘#’) recognized by glob(3).

VIS_SHELL Also encode the meta characters used by shells (in addition to the glob characters): (‘’’, ‘‘’, ‘"’, ‘;’, ‘&’, ‘<’, ‘>’, ‘(’, ‘)’, ‘|’, ‘]’, ‘\’, ‘$’, ‘!’, ‘^’, and ‘~’).

VIS_SP Also encode space.

VIS_TAB Also encode tab.

VIS_NL Also encode newline.

VIS_WHITE Synonym for VIS_SP | VIS_TAB | VIS_NL.

VIS_META Synonym for VIS_WHITE | VIS_GLOB | VIS_SHELL.

VIS_SAFE Only encode "unsafe" characters. Unsafe means control characters which may cause common terminals to perform unexpected functions. Currently this form allows space, tab, newline, backspace, bell, and return -- in addition to all graphic characters -- unencoded.

(The above flags have no effect for **svis**(), **snvis**(), **strsvis**(), **strsnvis**(), **strsvisx**(), and **strsnvisx**(). When using these functions, place all graphic characters to be encoded in an array pointed to by *extra*. In general, the backslash character should be included in this array, see the warning on the use of the VIS_NOSLASH flag below).

There are six forms of encoding. All forms use the backslash character ‘\’ to introduce a special sequence; two backslashes are used to represent a real backslash, except VIS_HTTPSTYLE that uses

‘%’, or VIS_MIMESTYLE that uses ‘=’. These are the visual formats:

(default) Use an ‘M’ to represent meta characters (characters with the 8th bit set), and use caret ‘^’ to represent control characters (see iscntrl(3)). The following formats are used:

\^C Represents the control character ‘C’. Spans characters ‘\000’ through ‘\037’, and ‘\177’ (as ‘\^?’).

\M-C Represents character ‘C’ with the 8th bit set. Spans characters ‘\241’ through ‘\376’.

\M^C Represents control character ‘C’ with the 8th bit set. Spans characters ‘\200’ through ‘\237’, and ‘\377’ (as ‘\M^?’).

\040 Represents ASCII space.

\240 Represents Meta-space.

VIS_CSTYLE Use C-style backslash sequences to represent standard non-printable characters. The following sequences are used to represent the indicated characters:

\a -- BEL (007)
\b -- BS (010)
\f -- NP (014)
\n -- NL (012)
\r -- CR (015)
\s -- SP (040)
\t -- HT (011)
\v -- VT (013)
\0 -- NUL (000)

When using this format, the *nextc* parameter is looked at to determine if a NUL character can be encoded as ‘\0’ instead of ‘\000’. If *nextc* is an octal digit, the latter representation is used to avoid ambiguity.

Non-printable characters without C-style backslash sequences use the default representation.

VIS_OCTAL Use a three digit octal sequence. The form is ‘\ddd’ where *d* represents an octal digit.

VIS_CSTYLE | VIS_OCTAL
: Same as VIS_CSTYLE except that non-printable characters without C-style backslash sequences use a three digit octal sequence.

VIS_HTTPSTYLE
: Use URI encoding as described in RFC 1738. The form is '%xx' where *x* represents a lower case hexadecimal digit.

VIS_MIMESTYLE
: Use MIME Quoted-Printable encoding as described in RFC 2045, only don't break lines and don't handle CRLF. The form is '=XX' where *X* represents an upper case hexadecimal digit.

There is one additional flag, VIS_NOSLASH, which inhibits the doubling of backslashes and the backslash before the default format (that is, control characters are represented by '^C' and meta characters as 'M-C'). With this flag set, the encoding is ambiguous and non-invertible.

MULTIBYTE CHARACTER SUPPORT

These functions support multibyte character input. The encoding conversion is influenced by the setting of the LC_CTYPE environment variable which defines the set of characters that can be copied without encoding.

If VIS_NOLOCALE is set, processing is done assuming the C locale and overriding any other environment settings.

When 8-bit data is present in the input, LC_CTYPE must be set to the correct locale or to the C locale. If the locales of the data and the conversion are mismatched, multibyte character recognition may fail and encoding will be performed byte-by-byte instead.

As noted above, *dst* must be four times the number of bytes processed from *src*. But note that each multibyte character can be up to MB_LEN_MAX bytes so in terms of multibyte characters, *dst* must be four times MB_LEN_MAX times the number of characters processed from *src*.

ENVIRONMENT

LC_CTYPE Specify the locale of the input data. Set to C if the input data locale is unknown.

ERRORS

The functions **nvis**() and **snvis**() will return NULL and the functions **strnvis**(), **strnvisx**(), **strsnvis**(), and **strsnvisx**(), will return -1 when the *dlen* destination buffer size is not enough to perform the conversion while setting *errno* to:

[ENOSPC] The destination buffer size is not large enough to perform the conversion.

SEE ALSO

unvis(1), vis(1), glob(3), unvis(3)

T. Berners-Lee, *Uniform Resource Locators (URL)*, RFC 1738.

Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, RFC 2045.

HISTORY

The **vis**(), **strvis**(), and **strvisx**() functions first appeared in 4.4BSD. The **svis**(), **strsvis**(), and **strsvisx**() functions appeared in NetBSD 1.5 and FreeBSD 9.2. The buffer size limited versions of the functions (**nvis**(), **strnvis**(), **strnvisx**(), **snvis**(), **strsnvis**(), and **strsnvisx**()) appeared in NetBSD 6.0 and FreeBSD 9.2. Multibyte character support was added in NetBSD 7.0 and FreeBSD 9.2.

NAME

open_memstream, **open_wmemstream** - dynamic memory buffer stream open functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*FILE **
open_memstream(*char **bufp*, *size_t *sizep*);

#include <wchar.h>

*FILE **
open_wmemstream(*wchar_t **bufp*, *size_t *sizep*);

DESCRIPTION

The **open_memstream**() and **open_wmemstream**() functions create a write-only, seekable stream backed by a dynamically allocated memory buffer. The **open_memstream**() function creates a byte-oriented stream, while the **open_wmemstream**() function creates a wide-oriented stream.

Each stream maintains a current position and size. Initially, the position and size are set to zero. Each write begins at the current position and advances it the number of successfully written bytes for **open_memstream**() or wide characters for **open_wmemstream**(). If a write moves the current position beyond the length of the buffer, the length of the buffer is extended and a null character is appended to the buffer.

A stream's buffer always contains a null character at the end of the buffer that is not included in the current length.

If a stream's current position is moved beyond the current length via a seek operation and a write is performed, the characters between the current length and the current position are filled with null characters before the write is performed.

After a successful call to fclose(3) or fflush(3), the pointer referenced by *bufp* will contain the start of the memory buffer and the variable referenced by *sizep* will contain the smaller of the current position and the current buffer length.

After a successful call to fflush(3), the pointer referenced by *bufp* and the variable referenced by *sizep*

are only valid until the next write operation or a call to fclose(3).

Once a stream is closed, the allocated buffer referenced by *bufp* should be released via a call to free(3) when it is no longer needed.

IMPLEMENTATION NOTES

Internally all I/O streams are effectively byte-oriented, so using wide-oriented operations to write to a stream opened via **open_wmemstream**() results in wide characters being expanded to a stream of multibyte characters in stdio's internal buffers. These multibyte characters are then converted back to wide characters when written into the stream. As a result, the wide-oriented streams maintain an internal multibyte character conversion state that is cleared on any seek opertion that changes the current position. This should have no effect as long as wide-oriented output operations are used on a wide-oriented stream.

RETURN VALUES

Upon successful completion, **open_memstream**() and **open_wmemstream**() return a FILE pointer. Otherwise, NULL is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The *bufp* or *sizep* argument was NULL.

[ENOMEM] Memory for the stream or buffer could not be allocated.

SEE ALSO

fclose(3), fflush(3), fopen(3), free(3), fseek(3), stdio(3), sbuf(9)

STANDARDS

The **open_memstream**() and **open_wmemstream**() functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

open, **openat** - open or create a file for reading, writing or executing

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>

int
open(*const char *path*, *int flags*, ...);

int
openat(*int fd*, *const char *path*, *int flags*, ...);

DESCRIPTION

The file name specified by *path* is opened for either execution or reading and/or writing as specified by the argument *flags* and the file descriptor returned to the calling process. The *flags* argument may indicate the file is to be created if it does not exist (by specifying the O_CREAT flag). In this case **open**() and **openat**() require an additional argument *mode_t mode*, and the file is created with mode *mode* as described in chmod(2) and modified by the process' umask value (see umask(2)).

The **openat**() function is equivalent to the **open**() function except in the case where the *path* specifies a relative path. In this case the file to be opened is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. The *flag* parameter and the optional fourth parameter correspond exactly to the parameters of **open**(). If **openat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **open**().

In capsicum(4) capability mode, **open**() is not permitted. The *path* argument to **openat**() must be strictly relative to a file descriptor *fd*, as defined in *sys/kern/vfs_lookup.c*. *path* must not be an absolute path and must not contain ".." components. Additionally, no symbolic link in *path* may contain ".." components either. *fd* must not be AT_FDCWD.

The flags specified are formed by *or*'ing the following values

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXEC	open for execute only

O_NONBLOCK	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if create and file exists
O_SHLOCK	atomically obtain a shared lock
O_EXLOCK	atomically obtain an exclusive lock
O_DIRECT	eliminate or reduce cache effects
O_FSYNC	synchronous writes
O_SYNC	synchronous writes
O_NOFOLLOW	do not follow symlinks
O_NOCTTY	ignored
O_TTY_INIT	ignored
O_DIRECTORY	error if file is not a directory
O_CLOEXEC	set FD_CLOEXEC upon open
O_VERIFY	verify the contents of the file

Opening a file with O_APPEND set causes each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length. If O_EXCL is set with O_CREAT and the file already exists, **open**() returns an error. This may be used to implement a simple exclusive access locking mechanism. If O_EXCL is set and the last component of the pathname is a symbolic link, **open**() will fail even if the symbolic link points to a non-existent name. If the O_NONBLOCK flag is specified and the **open**() system call would result in the process being blocked for some reason (e.g., waiting for carrier on a dialup line), **open**() returns immediately. The descriptor remains in non-blocking mode for subsequent operations.

If O_FSYNC is used in the mask, all writes will immediately be written to disk, the kernel will not cache written data and all writes on the descriptor will not return until the data to be written completes.

O_SYNC is a synonym for O_FSYNC required by POSIX.

If O_NOFOLLOW is used in the mask and the target file passed to **open**() is a symbolic link then the **open**() will fail.

When opening a file, a lock with flock(2) semantics can be obtained by setting O_SHLOCK for a shared lock, or O_EXLOCK for an exclusive lock. If creating a file with O_CREAT, the request for the lock will never fail (provided that the underlying file system supports locking).

O_DIRECT may be used to minimize or eliminate the cache effects of reading and writing. The system will attempt to avoid caching the data you read or write. If it cannot avoid caching the data, it will

minimize the impact the data has on the cache. Use of this flag can drastically reduce performance if not used with care.

O_NOCTTY may be used to ensure the OS does not assign this file as the controlling terminal when it opens a tty device. This is the default on FreeBSD, but is present for POSIX compatibility. The **open**() system call will not assign controlling terminals on FreeBSD.

O_TTY_INIT may be used to ensure the OS restores the terminal attributes when initially opening a TTY. This is the default on FreeBSD, but is present for POSIX compatibility. The initial call to **open**() on a TTY will always restore default terminal attributes on FreeBSD.

O_DIRECTORY may be used to ensure the resulting file descriptor refers to a directory. This flag can be used to prevent applications with elevated privileges from opening files which are even unsafe to open with O_RDONLY, such as device nodes.

O_CLOEXEC may be used to set FD_CLOEXEC flag for the newly returned file descriptor.

O_VERIFY may be used to indicate to the kernel that the contents of the file should be verified before allowing the open to proceed. The details of what "verified" means is implementation specific. The run-time linker (rtld) uses this flag to ensure shared objects have been verified before operating on them.

If successful, **open**() returns a non-negative integer, termed a file descriptor. It returns -1 on failure. The file pointer used to mark the current position within the file is set to the beginning of the file.

If a sleeping open of a device node from devfs(5) is interrupted by a signal, the call always fails with EINTR, even if the SA_RESTART flag is set for the signal. A sleeping open of a fifo (see mkfifo(2)) is restarted as normal.

When a new file is created it is given the group of the directory which contains it.

Unless O_CLOEXEC flag was specified, the new descriptor is set to remain open across execve(2) system calls; see close(2), fcntl(2) and O_CLOEXEC description.

The system imposes a limit on the number of file descriptors open simultaneously by one process. The getdtablesize(2) system call returns the current system limit.

RETURN VALUES

If successful, **open**() and **openat**() return a non-negative integer, termed a file descriptor. They return -1 on failure, and set *errno* to indicate the error.

ERRORS

The named file is opened unless:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] O_CREAT is not set and the named file does not exist.

[ENOENT] A component of the path name that must exist does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The required permissions (for reading and/or writing) are denied for the given flags.

[EACCES] O_TRUNC is specified and write permission is denied.

[EACCES] O_CREAT is specified, the file does not exist, and the directory in which it is to be created does not permit writing.

[EPERM] O_CREAT is specified, the file does not exist, and the directory in which it is to be created has its immutable flag set, see the chflags(2) manual page for more information.

[EPERM] The named file has its immutable flag set and the file is to be modified.

[EPERM] The named file has its append-only flag set, the file is to be modified, and O_TRUNC is specified or O_APPEND is not specified.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EISDIR] The named file is a directory, and the arguments specify it is to be modified.

[EROFS] The named file resides on a read-only file system, and the file is to be modified.

[EROFS] O_CREAT is specified and the named file would reside on a read-only file system.

[EMFILE] The process has already reached its limit for open file descriptors.

[ENFILE] The system file table is full.

[EMLINK] O_NOFOLLOW was specified and the target is a symbolic link.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[ENXIO] O_NONBLOCK is set, the named file is a fifo, O_WRONLY is set, and no process has the file open for reading.

[EINTR] The **open**() operation was interrupted by a signal.

[EOPNOTSUPP] O_SHLOCK or O_EXLOCK is specified but the underlying file system does not support locking.

[EOPNOTSUPP] The named file is a special file mounted through a file system that does not support access to it (e.g. NFS).

[EWOULDBLOCK] O_NONBLOCK and one of O_SHLOCK or O_EXLOCK is specified and the file is locked.

[ENOSPC] O_CREAT is specified, the file does not exist, and the directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] O_CREAT is specified, the file does not exist, and there are no free inodes on the file system on which the file is being created.

[EDQUOT] O_CREAT is specified, the file does not exist, and the directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] O_CREAT is specified, the file does not exist, and the user's quota of inodes on the file system on which the file is being created has been exhausted.

[EIO] An I/O error occurred while making the directory entry or allocating the inode for O_CREAT.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and the **open**() system call requests write access.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EEXIST] O_CREAT and O_EXCL were specified and the file exists.

[EOPNOTSUPP] An attempt was made to open a socket (not currently implemented).

[EINVAL] An attempt was made to open a descriptor with an illegal combination of O_RDONLY, O_WRONLY, O_RDWR and O_EXEC.

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

[ENOTDIR] O_DIRECTORY is specified and the file is not a directory.

[ECAPMODE] AT_FDCWD is specified and the process is in capability mode.

[ECAPMODE] **open**() was called and the process is in capability mode.

[ENOTCAPABLE] *path* is an absolute path or contained a ".." component leading to a directory outside of the directory hierarchy specified by *fd*.

SEE ALSO

chmod(2), close(2), dup(2), fexecve(2), fhopen(2), getdtablesize(2), getfh(2), lgetfh(2), lseek(2), read(2), umask(2), write(2), fopen(3), capsicum(4)

STANDARDS

These functions are specified by IEEE Std 1003.1-2008 ("POSIX.1"). FreeBSD sets *errno* to EMLINK instead of ELOOP as specified by POSIX when O_NOFOLLOW is set in flags and the final component of pathname is a symbolic link to distinguish it from the case of too many symbolic link traversals in one of its non-final components.

HISTORY

The **open**() function appeared in Version 1 AT&T UNIX. The **openat**() function was introduced in FreeBSD 8.0.

BUGS

The Open Group Extended API Set 2 specification requires that the test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches. The present implementation of the *openat* checks the current permissions of directory instead.

NAME

pause - stop until signal

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
pause(*void*);

DESCRIPTION

Pause is made obsolete by sigsuspend(2).

The **pause**() function forces a process to pause until a signal is received from either the kill(2) function or an interval timer. (See setitimer(2).) Upon termination of a signal handler started during a **pause**(), the **pause**() call will return.

RETURN VALUES

Always returns -1.

ERRORS

The **pause**() function always returns:

[EINTR] The call was interrupted.

SEE ALSO

kill(2), select(2), sigsuspend(2)

HISTORY

A **pause**() syscall appeared in Version 6 AT&T UNIX.

NAME

popen, **pclose** - process I/O

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*FILE **
popen(*const char *command*, *const char *type*);

int
pclose(*FILE *stream*);

DESCRIPTION

The **popen**() function "opens" a process by creating a bidirectional pipe forking, and invoking the shell. Any streams opened by previous **popen**() calls in the parent process are closed in the new child process. Historically, **popen**() was implemented with a unidirectional pipe; hence many implementations of **popen**() only allow the *type* argument to specify reading or writing, not both. Since **popen**() is now implemented using a bidirectional pipe, the *type* argument may request a bidirectional data flow. The *type* argument is a pointer to a null-terminated string which must be ‘r’ for reading, ‘w’ for writing, or ‘r+’ for reading and writing.

A letter ‘e’ may be appended to that to request that the underlying file descriptor be set close-on-exec.

The *command* argument is a pointer to a null-terminated string containing a shell command line. This command is passed to */bin/sh* using the **-c** flag; interpretation, if any, is performed by the shell.

The return value from **popen**() is a normal standard I/O stream in all respects save that it must be closed with **pclose**() rather than **fclose**(). Writing to such a stream writes to the standard input of the command; the command’s standard output is the same as that of the process that called **popen**(), unless this is altered by the command itself. Conversely, reading from a "popened" stream reads the command’s standard output, and the command’s standard input is the same as that of the process that called **popen**().

Note that output **popen**() streams are fully buffered by default.

The **pclose**() function waits for the associated process to terminate and returns the exit status of the command as returned by wait4(2).

RETURN VALUES

The **popen**() function returns NULL if the fork(2) or pipe(2) calls fail, or if it cannot allocate memory.

The **pclose**() function returns -1 if *stream* is not associated with a "popened" command, if *stream* already "pclosed", or if wait4(2) returns an error.

ERRORS

The **popen**() function does not reliably set *errno*.

SEE ALSO

sh(1), fork(2), pipe(2), wait4(2), fclose(3), fflush(3), fopen(3), stdio(3), system(3)

HISTORY

A **popen**() and a **pclose**() function appeared in Version 7 AT&T UNIX.

Bidirectional functionality was added in FreeBSD 2.2.6.

BUGS

Since the standard input of a command opened for reading shares its seek offset with the process that called **popen**(), if the original process has done a buffered read, the command's input position may not be as expected. Similarly, the output from a command opened for writing may become intermingled with that of the original process. The latter can be avoided by calling fflush(3) before **popen**().

Failure to execute the shell is indistinguishable from the shell's failure to execute command, or an immediate exit of the command. The only hint is an exit status of 127.

The **popen**() function always calls sh(1), never calls csh(1).

NAME

pdfork, **pdgetpid**, **pdkill** - System calls to manage process descriptors

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/procdesc.h>

pid_t
pdfork(*int *fdp*, *int flags*);

int
pdgetpid(*int fd*, *pid_t *pidp*);

int
pdkill(*int fd*, *int signum*);

DESCRIPTION

Process descriptors are special file descriptors that represent processes, and are created using **pdfork**(), a variant of fork(2), which, if successful, returns a process descriptor in the integer pointed to by *fdp*. Processes created via **pdfork**() will not cause SIGCHLD on termination. **pdfork**() can accept the flags:

PD_DAEMON Instead of the default terminate-on-close behaviour, allow the process to live until it is explicitly killed with kill(2).

This option is not permitted in capsicum(4) capability mode (see cap_enter(2)).

PD_CLOEXEC Set close-on-exec on process descriptor.

pdgetpid() queries the process ID (PID) in the process descriptor *fd*.

pdkill() is functionally identical to kill(2), except that it accepts a process descriptor, *fd*, rather than a PID.

The following system calls also have effects specific to process descriptors:

fstat(2) queries status of a process descriptor; currently only the *st_mode*, *st_birthtime*, *st_atime*, *st_ctime* and *st_mtime* fields are defined. If the owner read, write, and execute bits are set then the process represented by the process descriptor is still alive.

poll(2) and select(2) allow waiting for process state transitions; currently only POLLHUP is defined, and will be raised when the process dies. Process state transitions can also be monitored using kqueue(2) filter EVFILT_PROCDESC; currently only NOTE_EXIT is implemented.

close(2) will close the process descriptor unless PD_DAEMON is set; if the process is still alive and this is the last reference to the process descriptor, the process will be terminated with the signal SIGKILL.

RETURN VALUES

pdfork() returns a PID, 0 or -1, as fork(2) does.

pdgetpid() and **pdkill**() return 0 on success and -1 on failure.

ERRORS

These functions may return the same error numbers as their PID-based equivalents (e.g. **pdfork**() may return the same error numbers as fork(2)), with the following additions:

[EINVAL] The signal number given to **pdkill**() is invalid.

[ENOTCAPABLE] The process descriptor being operated on has insufficient rights (e.g. CAP_PDKILL for **pdkill**()).

SEE ALSO

close(2), fork(2), fstat(2), kill(2), poll(2), kqueue(2), wait4(2), capsicum(4), procdesc(4)

HISTORY

The **pdfork**(), **pdgetpid**(), and **pdkill**() system calls first appeared in FreeBSD 9.0.

Support for process descriptors mode was developed as part of the TrustedBSD Project.

AUTHORS

These functions and the capability facility were created by Robert N. M. Watson <*rwatson@FreeBSD.org*> and Jonathan Anderson <*jonathan@FreeBSD.org*> at the University of Cambridge Computer Laboratory with support from a grant from Google, Inc.

NAME

pdfork, **pdgetpid**, **pdkill**, **pdwait4** - System calls to manage process descriptors

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/procdesc.h>

pid_t
pdfork(*int *fdp*, *int flags*);

int
pdgetpid(*int fd*, *pid_t *pidp*);

int
pdkill(*int fd*, *int signum*);

int
pdwait4(*int fd*, *int *status*, *int options*, *struct rusage *rusage*);

DESCRIPTION

Process descriptors are special file descriptors that represent processes, and are created using **pdfork**(), a variant of fork(2), which, if successful, returns a process descriptor in the integer pointed to by *fdp*. Processes created via **pdfork**() will not cause SIGCHLD on termination. **pdfork**() can accept the flags:

PD_DAEMON Instead of the default terminate-on-close behaviour, allow the process to live until it is explicitly killed with kill(2).

This option is not permitted in capsicum(4) capability mode (see cap_enter(2)).

PD_CLOEXEC Set close-on-exec on process descriptor.

pdgetpid() queries the process ID (PID) in the process descriptor *fd*.

pdkill() is functionally identical to kill(2), except that it accepts a process descriptor, *fd*, rather than a PID.

pdwait4() behaves identically to wait4(2), but operates with respect to a process descriptor argument rather than a PID.

The following system calls also have effects specific to process descriptors:

fstat(2) queries status of a process descriptor; currently only the *st_mode*, *st_birthtime*, *st_atime*, *st_ctime* and *st_mtime* fields are defined. If the owner read, write, and execute bits are set then the process represented by the process descriptor is still alive.

poll(2) and select(2) allow waiting for process state transitions; currently only POLLHUP is defined, and will be raised when the process dies. Process state transitions can also be monitored using kqueue(2) filter EVFILT_PROCDESC; currently only NOTE_EXIT is implemented.

close(2) will close the process descriptor unless PD_DAEMON is set; if the process is still alive and this is the last reference to the process descriptor, the process will be terminated with the signal SIGKILL.

RETURN VALUES

pdfork() returns a PID, 0 or -1, as fork(2) does.

pdgetpid() and **pdkill**() return 0 on success and -1 on failure.

pdwait4() returns a PID on success and -1 on failure.

ERRORS

These functions may return the same error numbers as their PID-based equivalents (e.g. **pdfork**() may return the same error numbers as fork(2)), with the following additions:

[EINVAL] The signal number given to **pdkill**() is invalid.

[ENOTCAPABLE] The process descriptor being operated on has insufficient rights (e.g. CAP_PDKILL for **pdkill**()).

SEE ALSO

close(2), fork(2), fstat(2), kill(2), poll(2), wait4(2), capsicum(4), procdesc(4)

HISTORY

The **pdfork**(), **pdgetpid**(), **pdkill**() and **pdwait4**() system calls first appeared in FreeBSD 9.0.

Support for process descriptors mode was developed as part of the TrustedBSD Project.

AUTHORS

These functions and the capability facility were created by Robert N. M. Watson <*rwatson@FreeBSD.org*> and Jonathan Anderson <*jonathan@FreeBSD.org*> at the University of

Cambridge Computer Laboratory with support from a grant from Google, Inc.

BUGS

pdwait4() has not yet been implemented.

NAME

perror, **strerror**, **strerror_r**, **sys_errlist**, **sys_nerr** - system error messages

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

void
perror(*const char *string*);

*extern const char * const sys_errlist[]*;
extern const int sys_nerr;

#include <string.h>

*char **
strerror(*int errnum*);

int
strerror_r(*int errnum*, *char *strerrbuf*, *size_t buflen*);

DESCRIPTION

The **strerror**(), **strerror_r**() and **perror**() functions look up the error message string corresponding to an error number.

The **strerror**() function accepts an error number argument *errnum* and returns a pointer to the corresponding message string.

The **strerror_r**() function renders the same result into *strerrbuf* for a maximum of *buflen* characters and returns 0 upon success.

The **perror**() function finds the error message corresponding to the current value of the global variable *errno* (intro(2)) and writes it, followed by a newline, to the standard error file descriptor. If the argument *string* is non-NULL and does not point to the null character, this string is prepended to the message string and separated from it by a colon and space (": "); otherwise, only the error message string is printed.

If the error number is not recognized, these functions return an error message string containing

"Unknown error: " followed by the error number in decimal. The **strerror**() and **strerror_r**() functions return EINVAL as a warning. Error numbers recognized by this implementation fall in the range 0 < *errnum* < *sys_nerr*. The number 0 is also recognized, although applications that take advantage of this are likely to use unspecified values of *errno*.

If insufficient storage is provided in *strerrbuf* (as specified in *buflen*) to contain the error string, **strerror_r**() returns ERANGE and *strerrbuf* will contain an error message that has been truncated and NUL terminated to fit the length specified by *buflen*.

The message strings can be accessed directly using the external array *sys_errlist*. The external value *sys_nerr* contains a count of the messages in *sys_errlist*. The use of these variables is deprecated; **strerror**() or **strerror_r**() should be used instead.

SEE ALSO

intro(2), err(3), psignal(3)

STANDARDS

The **perror**() and **strerror**() functions conform to ISO/IEC 9899:1999 ("ISO C99"). The **strerror_r**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **strerror**() and **perror**() functions first appeared in 4.4BSD. The **strerror_r**() function was implemented in FreeBSD 4.4 by Wes Peters <*wes@FreeBSD.org*>.

BUGS

The **strerror**() function returns its result in a static buffer which will be overwritten by subsequent calls.

The return type for **strerror**() is missing a type-qualifier; it should actually be *const char* *.

Programs that use the deprecated *sys_errlist* variable often fail to compile because they declare it inconsistently.

NAME

pipe, **pipe2** - create descriptor pair for interprocess communication

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
pipe(*int fildes[2]*);

int
pipe2(*int fildes[2]*, *int flags*);

DESCRIPTION

The **pipe**() function creates a *pipe*, which is an object allowing bidirectional data flow, and allocates a pair of file descriptors.

The **pipe2**() system call allows control over the attributes of the file descriptors via the *flags* argument. Values for *flags* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

O_CLOEXEC Set the close-on-exec flag for the new file descriptors.

O_NONBLOCK Set the non-blocking flag for the ends of the pipe.

If the *flags* argument is 0, the behavior is identical to a call to **pipe**().

By convention, the first descriptor is normally used as the *read end* of the pipe, and the second is normally the *write end*, so that data written to *fildes[1]* appears on (i.e., can be read from) *fildes[0]*. This allows the output of one program to be sent to another program: the source's standard output is set up to be the write end of the pipe, and the sink's standard input is set up to be the read end of the pipe. The pipe itself persists until all its associated descriptors are closed.

A pipe that has had an end closed is considered *widowed*. Writing on such a pipe causes the writing process to receive a SIGPIPE signal. Widowing a pipe is the only way to deliver end-of-file to a reader: after the reader consumes any buffered data, reading a widowed pipe returns a zero count.

The bidirectional nature of this implementation of pipes is not portable to older systems, so it is

recommended to use the convention for using the endpoints in the traditional manner when using a pipe in one direction.

IMPLEMENTATION NOTES

The **pipe**() function calls the **pipe2**() system call. As a result, system call traces such as those captured by dtrace(1) or ktrace(1) will show calls to **pipe2**().

RETURN VALUES

The **pipe**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **pipe**() and **pipe2**() system calls will fail if:

[EFAULT] *fildes* argument points to an invalid memory location.

[EMFILE] Too many descriptors are active.

[ENFILE] The system file table is full.

[ENOMEM] Not enough kernel memory to establish a pipe.

The **pipe2**() system call will also fail if:

[EINVAL] The *flags* argument is invalid.

SEE ALSO

sh(1), fork(2), read(2), socketpair(2), write(2)

HISTORY

The **pipe**() function appeared in Version 3 AT&T UNIX.

Bidirectional pipes were first used on AT&T System V Release 4 UNIX.

The **pipe2**() function appeared in FreeBSD 10.0.

The **pipe**() function became a wrapper around **pipe2**() in FreeBSD 11.0.

NAME

poll - synchronous I/O multiplexing

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <poll.h>

int
poll(*struct pollfd fds[]*, *nfds_t nfds*, *int timeout*);

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

DESCRIPTION

The **poll**() system call examines a set of file descriptors to see if some of them are ready for I/O. The *fds* argument is a pointer to an array of pollfd structures as defined in <*poll.h*> (shown below). The *nfds* argument determines the size of the *fds* array.

```
struct pollfd {
   int    fd;      /* file descriptor */
   short  events;  /* events to look for */
   short  revents; /* events returned */
};
```

The fields of *struct pollfd* are as follows:

fd — File descriptor to poll. If fd is equal to -1 then *revents* is cleared (set to zero), and that pollfd is not checked.

events — Events to poll for. (See below.)

revents — Events which may occur. (See below.)

The event bitmasks in *events* and *revents* have the following bits:

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

POLLRDNORM	Normal data may be read without blocking.
POLLRDBAND	Data with a non-zero priority may be read without blocking.
POLLPRI	High priority data may be read without blocking.
POLLOUT	
POLLWRNORM	Normal data may be written without blocking.
POLLWRBAND	Data with a non-zero priority may be written without blocking.
POLLERR	An exceptional condition has occurred on the device or socket. This flag is always checked, even if not present in the *events* bitmask.
POLLHUP	The device or socket has been disconnected. This flag is always checked, even if not present in the *events* bitmask. Note that POLLHUP and POLLOUT should never be present in the *revents* bitmask at the same time.
POLLNVAL	The file descriptor is not open. This flag is always checked, even if not present in the *events* bitmask.

If *timeout* is neither zero nor INFTIM (-1), it specifies a maximum interval to wait for any file descriptor to become ready, in milliseconds. If *timeout* is INFTIM (-1), the poll blocks indefinitely. If *timeout* is zero, then **poll**() will return without blocking.

The **ppoll**() system call, unlike **poll**(), is used to safely wait until either a set of file descriptors becomes ready or until a signal is caught. The *fds* and *nfds* arguments are identical to the analogous arguments of **poll**(). The *timeout* argument in **ppoll**() points to a *const struct timespec* which is defined in <*sys/timespec.h*> (shown below) rather than the *int timeout* used by **poll**(). A null pointer may be passed to indicate that **ppoll**() should wait indefinitely. Finally, *newsigmask* specifies a signal mask which is set while waiting for input. When **ppoll**() returns, the original signal mask is restored.

```
struct timespec {
        time_t  tv_sec;       /* seconds */
        long    tv_nsec;      /* and nanoseconds */
};
```

RETURN VALUES

The **poll**() system call returns the number of descriptors that are ready for I/O, or -1 if an error occurred.

If the time limit expires, **poll**() returns 0. If **poll**() returns with an error, including one due to an interrupted system call, the *fds* array will be unmodified.

COMPATIBILITY

This implementation differs from the historical one in that a given file descriptor may not cause **poll**() to return with an error. In cases where this would have happened in the historical implementation (e.g. trying to poll a revoke(2)ed descriptor), this implementation instead copies the *events* bitmask to the *revents* bitmask. Attempting to perform I/O on this descriptor will then return an error. This behaviour is believed to be more useful.

ERRORS

An error return from **poll**() indicates:

[EFAULT] The *fds* argument points outside the process's allocated address space.

[EINTR] A signal was delivered before the time limit expired and before any of the selected events occurred.

[EINVAL] The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), kqueue(2), pselect(2), read(2), recv(2), select(2), send(2), write(2)

STANDARDS

The **poll**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1"). The **ppoll**() is not specified by POSIX.

HISTORY

The **poll**() function appeared in AT&T System V UNIX. This manual page and the core of the implementation was taken from NetBSD. The **ppoll**() function first appeared in FreeBSD 11.0

BUGS

The distinction between some of the fields in the *events* and *revents* bitmasks is really not useful without STREAMS. The fields are defined for compatibility with existing software.

NAME

posix_fadvise - give advice about use of file data

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>

int
posix_fadvise(*int fd*, *off_t offset*, *off_t len*, *int advice*);

DESCRIPTION

The **posix_fadvise**() system call allows a process to describe to the system its data access behavior for an open file descriptor *fd*. The advice covers the data starting at offset *offset* and continuing for *len* bytes. If *len* is zero, all data from *offset* to the end of the file is covered.

The behavior is specified by the *advice* parameter and may be one of:

POSIX_FADV_NORMAL Tells the system to revert to the default data access behavior.

POSIX_FADV_RANDOM Is a hint that file data will be accessed randomly, and prefetching is likely not advantageous.

POSIX_FADV_SEQUENTIAL Tells the system that file data will be accessed sequentially. This currently does nothing as the default behavior uses heuristics to detect sequential behavior.

POSIX_FADV_WILLNEED Tells the system that the specified data will be accessed in the near future. The system may initiate an asynchronous read of the data if it is not already present in memory.

POSIX_FADV_DONTNEED Tells the system that the specified data will not be accessed in the near future. The system may decrease the in-memory priority of clean data within the specified range and future access to this data may require a read operation.

POSIX_FADV_NOREUSE Tells the system that the specified data will only be accessed once and then not reused. The system may decrease the in-memory priority of data once it has been read or written. Future access to this data may

require a read operation.

RETURN VALUES

If successful, **posix_fadvise**() returns zero. It returns an error on failure, without setting *errno*.

ERRORS

Possible failure conditions:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EINVAL]	The *advice* argument is not valid.
[EINVAL]	The *offset* or *len* arguments are negative, or *offset* + *len* is greater than the maximum file size.
[ENODEV]	The *fd* argument does not refer to a regular file.
[ESPIPE]	The *fd* argument is associated with a pipe or FIFO.

SEE ALSO

madvise(2)

STANDARDS

The **posix_fadvise**() interface conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_fadvise**() system call first appeared in FreeBSD 9.1.

NAME

posix_fallocate - pre-allocate storage for a range in a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>

int
posix_fallocate(*int fd*, *off_t offset*, *off_t len*);

DESCRIPTION

Required storage for the range *offset* to *offset + len* in the file referenced by *fd* is guaranteed to be allocated upon successful return. That is, if **posix_fallocate**() returns successfully, subsequent writes to the specified file data will not fail due to lack of free space on the file system storage media. Any existing file data in the specified range is unmodified. If *offset + len* is beyond the current file size, then **posix_fallocate**() will adjust the file size to *offset + len*. Otherwise, the file size will not be changed.

Space allocated by **posix_fallocate**() will be freed by a successful call to creat(2) or open(2) that truncates the size of the file. Space allocated via **posix_fallocate**() may be freed by a successful call to ftruncate(2) that reduces the file size to a size smaller than *offset + len*.

RETURN VALUES

If successful, **posix_fallocate**() returns zero. It returns an error on failure, without setting *errno*.

ERRORS

Possible failure conditions:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EBADF]	The *fd* argument references a file that was opened without write permission.
[EFBIG]	The value of *offset + len* is greater than the maximum file size.
[EINTR]	A signal was caught during execution.
[EINVAL]	The *len* argument was less than or equal to zero, the *offset* argument was less than zero, or the operation is not supported by the file system.

[EIO] An I/O error occurred while reading from or writing to a file system.

[ENODEV] The *fd* argument does not refer to a regular file.

[ENOSPC] There is insufficient free space remaining on the file system storage media.

[ENOTCAPABLE] The file descriptor *fd* has insufficient rights.

[ESPIPE] The *fd* argument is associated with a pipe or FIFO.

SEE ALSO

creat(2), ftruncate(2), open(2), unlink(2)

STANDARDS

The **posix_fallocate**() system call conforms to IEEE Std 1003.1-2004 ("POSIX.1").

HISTORY

The **posix_fallocate**() function appeared in FreeBSD 9.0.

AUTHORS

posix_fallocate() and this manual page were initially written by Matthew Fleming <*mdf@FreeBSD.org*>.

NAME

posix_openpt - open a pseudo-terminal device

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <fcntl.h>
#include <stdlib.h>

int
posix_openpt(*int oflag*);

DESCRIPTION

The **posix_openpt**() function allocates a new pseudo-terminal and establishes a connection with its master device. A slave device shall be created in */dev/pts*. After the pseudo-terminal has been allocated, the slave device should have the proper permissions before it can be used (see grantpt(3)). The name of the slave device can be determined by calling ptsname(3).

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*. Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

O_RDWR Open for reading and writing.

O_NOCTTY If set **posix_openpt**() shall not cause the terminal device to become the controlling terminal for the process.

O_CLOEXEC Set the close-on-exec flag for the new file descriptor.

The **posix_openpt**() function shall fail when *oflag* contains other values.

RETURN VALUES

Upon successful completion, the **posix_openpt**() function shall allocate a new pseudo-terminal device and return a non-negative integer representing a file descriptor, which is connected to its master device. Otherwise, -1 shall be returned and errno set to indicate the error.

ERRORS

The **posix_openpt**() function shall fail if:

[ENFILE] The system file table is full.

[EINVAL] The value of *oflag* is not valid.

[EAGAIN] Out of pseudo-terminal resources.

SEE ALSO

ptsname(3), pts(4), tty(4)

STANDARDS

The **posix_openpt**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1"). The ability to use O_CLOEXEC is an extension to the standard.

HISTORY

The **posix_openpt**() function appeared in FreeBSD 5.0. In FreeBSD 8.0, this function was changed to a system call.

NOTES

The flag O_NOCTTY is included for compatibility; in FreeBSD, opening a terminal does not cause it to become a process's controlling terminal.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawn_file_actions_addopen, **posix_spawn_file_actions_adddup2**, **posix_spawn_file_actions_addclose** - add open, dup2 or close action to spawn file actions object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawn_file_actions_addopen(*posix_spawn_file_actions_t * file_actions*, *int fildes*, *const char *restrict path*, *int oflag*, *mode_t mode*);

int
posix_spawn_file_actions_adddup2(*posix_spawn_file_actions_t * file_actions*, *int fildes*, *int newfildes*);

int
posix_spawn_file_actions_addclose(*posix_spawn_file_actions_t * file_actions*, *int fildes*);

DESCRIPTION

These functions add an open, dup2 or close action to a spawn file actions object.

A spawn file actions object is of type *posix_spawn_file_actions_t* (defined in <*spawn.h*>) and is used to specify a series of actions to be performed by a **posix_spawn**() or **posix_spawnp**() operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent.

A spawn file actions object, when passed to **posix_spawn**() or **posix_spawnp**(), specify how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation is as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to execution of the new process image), in the order in which the actions were added to the object; additionally, when the new process image is executed, any file descriptor (from this new set) which has its FD_CLOEXEC flag set is closed (see **posix_spawn**()).

The **posix_spawn_file_actions_addopen**() function adds an open action to the object referenced by *file_actions* that causes the file named by *path* to be opened (as if

open(path, oflag, mode)

had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a new process is spawned using this file actions object. If *fildes* was already an open file descriptor, it is closed before the new file is opened.

The string described by *path* is copied by the **posix_spawn_file_actions_addopen**() function.

The **posix_spawn_file_actions_adddup2**() function adds a dup2 action to the object referenced by *file_actions* that causes the file descriptor *fildes* to be duplicated as *newfildes* (as if

dup2(fildes, newfildes)

had been called) when a new process is spawned using this file actions object, except that the FD_CLOEXEC flag for *newfildes* is cleared even if *fildes* is equal to *newfildes*. The difference from **dup2**() is useful for passing a particular file descriptor to a particular child process.

The **posix_spawn_file_actions_addclose**() function adds a close action to the object referenced by *file_actions* that causes the file descriptor *fildes* to be closed (as if

close(fildes)

had been called) when a new process is spawned using this file actions object.

RETURN VALUES

Upon successful completion, these functions return zero; otherwise, an error number is returned to indicate the error.

ERRORS

These functions fail if:

[EBADF] The value specified by *fildes* or *newfildes* is negative.

[ENOMEM] Insufficient memory exists to add to the spawn file actions object.

SEE ALSO

close(2), dup2(2), open(2), posix_spawn(3), posix_spawn_file_actions_destroy(3), posix_spawn_file_actions_init(3), posix_spawnp(3)

STANDARDS

The **posix_spawn_file_actions_addopen**(), **posix_spawn_file_actions_adddup2**() and **posix_spawn_file_actions_addclose**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1"), with

the exception of the behavior of **posix_spawn_file_actions_adddup2**() if *fildes* is equal to *newfildes* (clearing FD_CLOEXEC). A future update of the Standard is expected to require this behavior.

HISTORY

The **posix_spawn_file_actions_addopen**(), **posix_spawn_file_actions_adddup2**() and **posix_spawn_file_actions_addclose**() functions first appeared in FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawn_file_actions_init, **posix_spawn_file_actions_destroy** - initialize and destroy spawn file actions object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawn_file_actions_init(*posix_spawn_file_actions_t * file_actions*);

int
posix_spawn_file_actions_destroy(*posix_spawn_file_actions_t * file_actions*);

DESCRIPTION

The **posix_spawn_file_actions_init**() function initialize the object referenced by **file_actions**() to contain no file actions for **posix_spawn**() or **posix_spawnp**(). Initializing an already initialized spawn file actions object may cause memory to be leaked.

The **posix_spawn_file_actions_destroy**() function destroy the object referenced by *file_actions*; the object becomes, in effect, uninitialized. A destroyed spawn file actions object can be reinitialized using **posix_spawn_file_actions_init**(). The object should not be used after it has been destroyed.

RETURN VALUES

Upon successful completion, these functions return zero; otherwise, an error number is returned to indicate the error.

ERRORS

The **posix_spawn_file_actions_init**() function will fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

SEE ALSO

posix_spawn(3), posix_spawn_file_actions_addclose(3), posix_spawn_file_actions_adddup2(3), posix_spawn_file_actions_addopen(3), posix_spawnp(3)

STANDARDS

The **posix_spawn_file_actions_init**() and **posix_spawn_file_actions_destroy**() functions conform to

IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawn_file_actions_init**() and **posix_spawn_file_actions_destroy**() functions first appeared in FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawn, **posix_spawnp** - spawn a process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawn(*pid_t *restrict pid*, *const char *restrict path*,
*const posix_spawn_file_actions_t *file_actions*, *const posix_spawnattr_t *restrict attrp*,
*char *const argv[restrict]*, *char *const envp[restrict]*);

int
posix_spawnp(*pid_t *restrict pid*, *const char *restrict file*,
*const posix_spawn_file_actions_t *file_actions*, *const posix_spawnattr_t *restrict attrp*,
*char *const argv[restrict]*, *char *const envp[restrict]*);

DESCRIPTION

The **posix_spawn**() and **posix_spawnp**() functions create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it is entered as a C-language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the variable:

```
extern char **environ;
```

points to an array of character pointers to the environment strings.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array is a null pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process image being started by the **posix_spawn**() or **posix_spawnp**() function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The *path* argument to **posix_spawn**() is a pathname that identifies the new process image file to execute.

The *file* parameter to **posix_spawnp**() is used to construct a pathname that identifies the new process image file. If the file parameter contains a slash character, the file parameter is used as the pathname for the new process image file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable "PATH". If this variable is not specified, the default path is set according to the _PATH_DEFPATH definition in <*paths.h*>, which is set to "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin".

If *file_actions* is a null pointer, then file descriptors open in the calling process remain open in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see **fcntl**()). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see **fcntl**()), remain unchanged.

If *file_actions* is not NULL, then the file descriptors open in the child process are those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions are:

1. The set of open file descriptors for the child process initially are the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see **fcntl**()), remain unchanged.

2. The signal mask, signal default actions, and the effective user and group IDs for the child process are changed as specified in the attributes object referenced by *attrp*.

3. The file actions specified by the spawn file actions object are performed in the order in which they were added to the spawn file actions object.

4. Any file descriptor that has its FD_CLOEXEC flag set (see **fcntl**()) is closed.

The *posix_spawnattr_t* spawn attributes object type is defined in <*spawn.h*>. It contains the attributes defined below.

If the POSIX_SPAWN_SETPGROUP flag is set in the spawn-flags attribute of the object referenced by *attrp*, and the spawn-pgroup attribute of the same object is non-zero, then the child's process group is as specified in the spawn-pgroup attribute of the object referenced by *attrp*.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the spawn-flags attribute of the object referenced by *attrp*, and the spawn-pgroup attribute of the same object is set to zero, then the child is in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the spawn-flags attribute of the object referenced by *attrp*, the new child process inherits the parent's process group.

If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the spawn-flags attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image initially has the scheduling policy of the calling process with the scheduling parameters specified in the spawn-schedparam attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSCHEDULER flag is set in the spawn-flags attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image initially has the scheduling policy specified in the spawn-schedpolicy attribute of the object referenced by *attrp* and the scheduling parameters specified in the spawn-schedparam attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the spawn-flags attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process inherits the parent process' effective user ID. If this flag is set, the child process' effective user ID is reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process becomes that file's owner ID before the new process image begins execution.

The POSIX_SPAWN_RESETIDS flag in the spawn-flags attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process inherits the parent process' effective group ID. If this flag is set, the child process' effective group ID is reset to the parent's real group ID. In either case, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the child process becomes that file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the spawn-flags attribute of the object referenced by *attrp*, the child process initially has the signal mask specified in the spawn-sigmask attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSIGDEF flag is set in the spawn-flags attribute of the object referenced by *attrp*, the signals specified in the spawn-sigdefault attribute of the same object are set to their default actions in the child process. Signals set to the default action in the parent process are set to the default action in the child process.

Signals set to be caught by the calling process are set to the default action in the child process.

Signals set to be ignored by the calling process image are set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the spawn-flags attribute of the object referenced by *attrp* and the signals being indicated in the spawn-sigdefault attribute of the object referenced by *attrp*.

If the value of the *attrp* pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, appear in the new process image as though **vfork**() had been called to create a child process and then **execve**() had been called by the child process to execute the new process image.

The implementation uses vfork(), thus the fork handlers are not run when **posix_spawn**() or **posix_spawnp**() is called.

RETURN VALUES

Upon successful completion, **posix_spawn**() and **posix_spawnp**() return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and return zero as the function return value. Otherwise, no child process is created, no value is stored into the variable pointed to by *pid*, and an error number is returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

ERRORS

1. If **posix_spawn**() and **posix_spawnp**() fail for any of the reasons that would cause **vfork**() or one of the **exec** to fail, an error value is returned as described by **vfork**() and **exec**, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

2. If **POSIX_SPAWN_SETPGROUP** is set in the spawn-flags attribute of the object referenced by attrp, and **posix_spawn**() or **posix_spawnp**() fails while changing the child's process group, an error value is returned as described by **setpgid**() (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

3. If **POSIX_SPAWN_SETSCHEDPARAM** is set and **POSIX_SPAWN_SETSCHEDULER** is not set in the spawn-flags attribute of the object referenced by attrp, then if **posix_spawn**() or **posix_spawnp**() fails for any of the reasons that would cause **sched_setparam**() to fail, an error value is returned as described by **sched_setparam**() (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

4. If **POSIX_SPAWN_SETSCHEDULER** is set in the spawn-flags attribute of the object referenced by attrp, and if **posix_spawn**() or **posix_spawnp**() fails for any of the reasons that would cause **sched_setscheduler**() to fail, an error value is returned as described by **sched_setscheduler**() (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

5. If the *file_actions* argument is not NULL, and specifies any dup2 or open actions to be performed, and if **posix_spawn**() or **posix_spawnp**() fails for any of the reasons that would cause **dup2**() or **open**() to fail, an error value is returned as described by **dup2**() and **open**(), respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127). An open file action may, by itself, result in any of the errors described by **dup2**(), in addition to those described by **open**(). This implementation ignores any errors from **close**(), including trying to close a descriptor that is not open.

SEE ALSO

close(2), dup2(2), execve(2), fcntl(2), open(2), sched_setparam(2), sched_setscheduler(2), setpgid(2), vfork(2), posix_spawn_file_actions_addclose(3), posix_spawn_file_actions_adddup2(3), posix_spawn_file_actions_addopen(3), posix_spawn_file_actions_destroy(3), posix_spawn_file_actions_init(3), posix_spawnattr_destroy(3), posix_spawnattr_getflags(3), posix_spawnattr_getpgroup(3), posix_spawnattr_getschedparam(3), posix_spawnattr_getschedpolicy(3), posix_spawnattr_getsigdefault(3), posix_spawnattr_getsigmask(3), posix_spawnattr_init(3), posix_spawnattr_setflags(3), posix_spawnattr_setpgroup(3), posix_spawnattr_setschedparam(3), posix_spawnattr_setschedpolicy(3), posix_spawnattr_setsigdefault(3), posix_spawnattr_setsigmask(3)

STANDARDS

The **posix_spawn**() and **posix_spawnp**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1"), except that they ignore all errors from **close**(). A future update of the Standard is expected to require that these functions not fail because a file descriptor to be closed (via **posix_spawn_file_actions_addclose**()) is not open.

HISTORY

The **posix_spawn**() and **posix_spawnp**() functions first appeared in FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawnattr_init, **posix_spawnattr_destroy** - initialize and destroy spawn attributes object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawnattr_init(*posix_spawnattr_t * attr*);

int
posix_spawnattr_destroy(*posix_spawnattr_t * attr*);

DESCRIPTION

The **posix_spawnattr_init**() function initializes a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation. Initializing an already initialized spawn attributes object may cause memory to be leaked.

The **posix_spawnattr_destroy**() function destroys a spawn attributes object. A destroyed *attr* attributes object can be reinitialized using **posix_spawnattr_init**(). The object should not be used after it has been destroyed.

A spawn attributes object is of type *posix_spawnattr_t* (defined in <*spawn.h*>) and is used to specify the inheritance of process attributes across a spawn operation.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of **posix_spawn**() or **posix_spawnp**(). After a spawn attributes object has been used to spawn a process by a call to a **posix_spawn**() or **posix_spawnp**(), any function affecting the attributes object (including destruction) will not affect any process that has been spawned in this way.

RETURN VALUES

Upon successful completion, **posix_spawnattr_init**() and **posix_spawnattr_destroy**() return zero; otherwise, an error number is returned to indicate the error.

ERRORS

The **posix_spawnattr_init**() function will fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

SEE ALSO

posix_spawn(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_init**() and **posix_spawnattr_destroy**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawnattr_init**() and **posix_spawnattr_destroy**() functions first appeared in FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawnattr_getflags, **posix_spawnattr_setflags** - get and set the spawn-flags attribute of a spawn attributes object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawnattr_getflags(*const posix_spawnattr_t *restrict attr*, *short *restrict flags*);

int
posix_spawnattr_setflags(*posix_spawnattr_t *attr*, *short flags*);

DESCRIPTION

The **posix_spawnattr_getflags**() function obtains the value of the spawn-flags attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setflags**() function sets the spawn-flags attribute in an initialized attributes object referenced by *attr*.

The spawn-flags attribute is used to indicate which process attributes are to be changed in the new process image when invoking **posix_spawn**() or **posix_spawnp**(). It is the bitwise-inclusive OR of zero or more of the following flags (see **posix_spawn**()):

POSIX_SPAWN_RESETIDS

POSIX_SPAWN_SETPGROUP

POSIX_SPAWN_SETSIGDEF

POSIX_SPAWN_SETSIGMASK

POSIX_SPAWN_SETSCHEDPARAM

POSIX_SPAWN_SETSCHEDULER

These flags are defined in <*spawn.h*>. The default value of this attribute is as if no flags were set.

RETURN VALUES

The **posix_spawnattr_getflags**() and **posix_spawnattr_setflags**() functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_init(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getflags**() and **posix_spawnattr_setflags**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawnattr_getflags**() and **posix_spawnattr_setflags**() functions first appeared in FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawnattr_getpgroup, **posix_spawnattr_setpgroup** - get and set the spawn-pgroup attribute of a spawn attributes object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawnattr_getpgroup(*const posix_spawnattr_t *restrict attr*, *pid_t *restrict pgroup*);

int
posix_spawnattr_setpgroup(*posix_spawnattr_t *attr*, *pid_t pgroup*);

DESCRIPTION

The **posix_spawnattr_getpgroup**() function obtains the value of the spawn-pgroup attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setpgroup**() function sets the spawn-pgroup attribute in an initialized attributes object referenced by *attr*.

The spawn-pgroup attribute represents the process group to be joined by the new process image in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the spawn-flags attribute). The default value of this attribute is zero.

RETURN VALUES

The **posix_spawnattr_getpgroup**() and **posix_spawnattr_setpgroup**() functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_init(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getpgroup**() and **posix_spawnattr_setpgroup**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawnattr_getpgroup**() and **posix_spawnattr_setpgroup**() functions first appeared in FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawnattr_getschedparam, **posix_spawnattr_setschedparam** - get and set the spawn-schedparam attribute of a spawn attributes object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawnattr_getschedparam(*const posix_spawnattr_t *restrict attr*,
*struct sched_param *restrict schedparam*);

int
posix_spawnattr_setschedparam(*posix_spawnattr_t *attr*,
*const struct sched_param *restrict schedparam*);

DESCRIPTION

The **posix_spawnattr_getschedparam**() function obtains the value of the spawn-schedparam attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setschedparam**() function sets the spawn-schedparam attribute in an initialized attributes object referenced by *attr*.

The spawn-schedparam attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER or POSIX_SPAWN_SETSCHEDPARAM is set in the spawn-flags attribute). The default value of this attribute is unspecified.

RETURN VALUES

The **posix_spawnattr_getschedparam**() and **posix_spawnattr_setschedparam**() functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_getschedpolicy(3), posix_spawnattr_init(3), posix_spawnattr_setschedpolicy(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getschedparam**() and **posix_spawnattr_setschedparam**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawnattr_getschedparam**() and **posix_spawnattr_setschedparam**() functions first appeared in FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawnattr_getschedpolicy, **posix_spawnattr_setschedpolicy** - get and set the spawn-schedpolicy attribute of a spawn attributes object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawnattr_getschedpolicy(*const posix_spawnattr_t *restrict attr*, *int *restrict schedpolicy*);

int
posix_spawnattr_setschedpolicy(*posix_spawnattr_t *attr*, *int schedpolicy*);

DESCRIPTION

The **posix_spawnattr_getschedpolicy**() function obtains the value of the spawn-schedpolicy attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setschedpolicy**() function sets the spawn-schedpolicy attribute in an initialized attributes object referenced by *attr*.

The spawn-schedpolicy attribute represents the scheduling policy to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER is set in the spawn-flags attribute). The default value of this attribute is unspecified.

RETURN VALUES

The **posix_spawnattr_getschedpolicy**() and **posix_spawnattr_setschedpolicy**() functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_getschedparam(3), posix_spawnattr_init(3), posix_spawnattr_setschedparam(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getschedpolicy**() and **posix_spawnattr_setschedpolicy**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawnattr_getschedpolicy**() and **posix_spawnattr_setschedpolicy**() functions first appeared in

FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawnattr_getsigdefault, **posix_spawnattr_setsigdefault** - get and set the spawn-sigdefault attribute of a spawn attributes object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawnattr_getsigdefault(*const posix_spawnattr_t *restrict attr*, *sigset_t *restrict sigdefault*);

int
posix_spawnattr_setsigdefault(*posix_spawnattr_t *attr*, *const sigset_t *restrict sigdefault*);

DESCRIPTION

The **posix_spawnattr_getsigdefault**() function obtains the value of the spawn-sigdefault attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setsigdefault**() function sets the spawn-sigdefault attribute in an initialized attributes object referenced by *attr*.

The spawn-sigdefault attribute represents the set of signals to be forced to default signal handling in the new process image (if POSIX_SPAWN_SETSIGDEF is set in the spawn-flags attribute) by a spawn operation. The default value of this attribute is an empty signal set.

RETURN VALUES

The **posix_spawnattr_getsigdefault**() and **posix_spawnattr_setsigdefault**() functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_getsigmask(3), posix_spawnattr_init(3), posix_spawnattr_setsigmask(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getsigdefault**() and **posix_spawnattr_setsigdefault**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawnattr_getsigdefault**() and **posix_spawnattr_setsigdefault**() functions first appeared in

FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix_spawnattr_getsigmask, **posix_spawnattr_setsigmask** - get and set the spawn-sigmask attribute of a spawn attributes object

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <spawn.h>

int
posix_spawnattr_getsigmask(*const posix_spawnattr_t *restrict attr*, *sigset_t *restrict sigmask*);

int
posix_spawnattr_setsigmask(*posix_spawnattr_t *attr*, *const sigset_t *restrict sigmask*);

DESCRIPTION

The **posix_spawnattr_getsigmask**() function obtains the value of the spawn-sigmask attribute from the attributes object referenced by *attr*.

The **posix_spawnattr_setsigmask**() function sets the spawn-sigmask attribute in an initialized attributes object referenced by *attr*.

The spawn-sigmask attribute represents the signal mask in effect in the new process image of a spawn operation (if POSIX_SPAWN_SETSIGMASK is set in the spawn-flags attribute). The default value of this attribute is unspecified.

RETURN VALUES

The **posix_spawnattr_getsigmask**() and **posix_spawnattr_setsigmask**() functions return zero.

SEE ALSO

posix_spawn(3), posix_spawnattr_destroy(3), posix_spawnattr_getsigmask(3), posix_spawnattr_init(3), posix_spawnattr_setsigmask(3), posix_spawnp(3)

STANDARDS

The **posix_spawnattr_getsigmask**() and **posix_spawnattr_setsigmask**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **posix_spawnattr_getsigmask**() and **posix_spawnattr_setsigmask**() functions first appeared in

FreeBSD 8.0.

AUTHORS

Ed Schouten <*ed@FreeBSD.org*>

NAME

posix1e - introduction to the POSIX.1e security API

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/acl.h>
#include <sys/mac.h>

DESCRIPTION

POSIX.1e describes five security extensions to the POSIX.1 API: Access Control Lists (ACLs), Auditing, Capabilities, Mandatory Access Control, and Information Flow Labels. While IEEE POSIX.1e D17 specification has not been standardized, several of its interfaces are widely used.

FreeBSD implements POSIX.1e interface for access control lists, described in acl(3), and supports ACLs on the ffs(7) file system; ACLs must be administratively enabled using tunefs(8).

FreeBSD implements a POSIX.1e-like mandatory access control interface, described in mac(3), although with a number of extensions and important semantic differences.

FreeBSD does not implement the POSIX.1e audit, privilege (capability), or information flow label APIs. However, FreeBSD does implement the libbsm(3) audit API. It also provides capsicum(4), a lightweight OS capability and sandbox framework implementing a hybrid capability system model.

ENVIRONMENT

POSIX.1e assigns security attributes to all objects, extending the security functionality described in POSIX.1. These additional attributes store fine-grained discretionary access control information and mandatory access control labels; for files, they are stored in extended attributes, described in extattr(3).

POSIX.2c describes a set of userland utilities for manipulating these attributes, including getfacl(1) and setfacl(1) for access control lists, and getfmac(8) and setfmac(8) for mandatory access control labels.

SEE ALSO

getfacl(1), setfacl(1), extattr(2), acl(3), extattr(3), libbsm(3), libcasper(3), mac(3), capsicum(4), ffs(7), getfmac(8), setfmac(8), tunefs(8), acl(9), extattr(9), mac(9)

STANDARDS

POSIX.1e is described in IEEE POSIX.1e draft 17.

HISTORY

POSIX.1e support was introduced in FreeBSD 4.0; most features were available as of FreeBSD 5.0.

AUTHORS

Robert N M Watson
Chris D. Faulhaber
Thomas Moestl
Ilmar S Habibulin

NAME

time2posix, **posix2time** - convert seconds since the Epoch

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

time_t
time2posix(*time_t t*);

time_t
posix2time(*time_t t*);

DESCRIPTION

IEEE Std 1003.1-1988 ("POSIX.1") legislates that a time_t value of 536457599 shall correspond to "Wed Dec 31 23:59:59 GMT 1986." This effectively implies that POSIX time_t's cannot include leap seconds and, therefore, that the system time must be adjusted as each leap occurs.

If the time package is configured with leap-second support enabled, however, no such adjustment is needed and time_t values continue to increase over leap events (as a true 'seconds since...' value). This means that these values will differ from those required by POSIX by the net number of leap seconds inserted since the Epoch.

Typically this is not a problem as the type time_t is intended to be (mostly) opaque--time_t values should only be obtained-from and passed-to functions such as time(3), localtime(3), mktime(3) and difftime(3). However, IEEE Std 1003.1-1988 ("POSIX.1") gives an arithmetic expression for directly computing a time_t value from a given date/time, and the same relationship is assumed by some (usually older) applications. Any programs creating/dissecting time_t's using such a relationship will typically not handle intervals over leap seconds correctly.

The **time2posix**() and **posix2time**() functions are provided to address this time_t mismatch by converting between local time_t values and their POSIX equivalents. This is done by accounting for the number of time-base changes that would have taken place on a POSIX system as leap seconds were inserted or deleted. These converted values can then be used in lieu of correcting the older applications, or when communicating with POSIX-compliant systems.

The **time2posix**() function is single-valued. That is, every local time_t corresponds to a single POSIX time_t. The **posix2time**() function is less well-behaved: for a positive leap second hit the result is not

unique, and for a negative leap second hit the corresponding POSIX time_t does not exist so an adjacent value is returned. Both of these are good indicators of the inferiority of the POSIX representation.

The following table summarizes the relationship between time_t and its conversion to, and back from, the POSIX representation over the leap second inserted at the end of June, 1993.

DATE	TIME	T	X=time2posix(T)	posix2time(X)
93/06/30	23:59:59	A+0	B+0	A+0
93/06/30	23:59:60	A+1	B+1	A+1 or A+2
93/07/01	00:00:00	A+2	B+1	A+1 or A+2
93/07/01	00:00:01	A+3	B+2	A+3

A leap second deletion would look like...

DATE	TIME	T	X=time2posix(T)	posix2time(X)
??/06/30	23:59:58	A+0	B+0	A+0
??/07/01	00:00:00	A+1	B+2	A+1
??/07/01	00:00:01	A+2	B+3	A+2

[Note: posix2time(B+1) => A+0 or A+1]

If leap-second support is not enabled, local time_t's and POSIX time_t's are equivalent, and both **time2posix**() and **posix2time**() degenerate to the identity function.

SEE ALSO

difftime(3), localtime(3), mktime(3), time(3)

NAME

read, **readv**, **pread**, **preadv** - read input

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

ssize_t
read(*int fd*, *void *buf*, *size_t nbytes*);

ssize_t
pread(*int fd*, *void *buf*, *size_t nbytes*, *off_t offset*);

#include <sys/uio.h>

ssize_t
readv(*int fd*, *const struct iovec *iov*, *int iovcnt*);

ssize_t
preadv(*int fd*, *const struct iovec *iov*, *int iovcnt*, *off_t offset*);

DESCRIPTION

The **read**() system call attempts to read *nbytes* of data from the object referenced by the descriptor *fd* into the buffer pointed to by *buf*. The **readv**() system call performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt-1]. The **pread**() and **preadv**() system calls perform the same functions, but read from the specified position in the file without modifying the file pointer.

For **readv**() and **preadv**(), the *iovec* structure is defined as:

```
struct iovec {
        void   *iov_base;  /* Base address. */
        size_t iov_len;    /* Length. */
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The **readv**() system call will always fill an area completely before proceeding to the next.

On objects capable of seeking, the **read**() starts at a position given by the pointer associated with *fd* (see lseek(2)). Upon return from **read**(), the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Upon successful completion, **read**(), **readv**(), **pread**() and **preadv**() return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

RETURN VALUES

If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **read**(), **readv**(), **pread**() and **preadv**() system calls will succeed unless:

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

[ECONNRESET] The *fd* argument refers to a socket, and the remote socket end is forcibly closed.

[EFAULT] The *buf* argument points outside the allocated address space.

[EIO] An I/O error occurred while reading from the file system.

[EBUSY] Failed to read from a file, e.g. /proc/<pid>/regs while <pid> is not stopped

[EINTR] A read from a slow device (i.e. one that might block for an arbitrary amount of time) was interrupted by the delivery of a signal before any data arrived.

[EINVAL] The pointer associated with *fd* was negative.

[EAGAIN] The file was marked for non-blocking I/O, and no data were ready to be read.

[EISDIR] The file descriptor is associated with a directory residing on a file system that does not allow regular read operations on directories (e.g. NFS).

[EOPNOTSUPP] The file descriptor is associated with a file system and file type that do not allow regular read operations on it.

[EOVERFLOW] The file descriptor is associated with a regular file, *nbytes* is greater than 0, *offset* is before the end-of-file, and *offset* is greater than or equal to the offset maximum established for this file system.

[EINVAL] The value *nbytes* is greater than INT_MAX.

In addition, **readv**() and **preadv**() may return one of the following errors:

[EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than IOV_MAX.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

[EFAULT] Part of the *iov* array points outside the process's allocated address space.

The **pread**() and **preadv**() system calls may also return the following errors:

[EINVAL] The *offset* value was negative.

[ESPIPE] The file descriptor is associated with a pipe, socket, or FIFO.

SEE ALSO

dup(2), fcntl(2), getdirentries(2), open(2), pipe(2), select(2), socket(2), socketpair(2), fread(3), readdir(3)

STANDARDS

The **read**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **readv**() and **pread**() system calls are expected to conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

The **preadv**() system call appeared in FreeBSD 6.0. The **pread**() function appeared in AT&T System V Release 4 UNIX. The **readv**() system call appeared in 4.2BSD. The **read**() function appeared in Version 1 AT&T UNIX.

NAME

procctl - control processes

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/procctl.h>

int
procctl(*idtype_t idtype*, *id_t id*, *int cmd*, *void *arg*);

DESCRIPTION

The **procctl**() system call provides for control over processes. The *idtype* and *id* arguments specify the set of processes to control. If multiple processes match the identifier, **procctl** will make a "best effort" to control as many of the selected processes as possible. An error is only returned if no selected processes successfully complete the request. The following identifier types are supported:

P_PID Control the process with the process ID *id*.

P_PGID Control processes belonging to the process group with the ID *id*.

The control request to perform is specified by the *cmd* argument. The following commands are supported:

PROC_SPROTECT Set process protection state. This is used to mark a process as protected from being killed if the system exhausts the available memory and swap. The *arg* parameter must point to an integer containing an operation and zero or more optional flags. The following operations are supported:

PPROT_SET Mark the selected processes as protected.

PPROT_CLEAR Clear the protected state of selected processes.

The following optional flags are supported:

PPROT_DESCEND Apply the requested operation to all child processes of each selected process in addition to each selected process.

PPROT_INHERIT — When used with PPROT_SET, mark all future child processes of each selected process as protected. Future child processes will also mark all of their future child processes.

PROC_REAP_ACQUIRE — Acquires the reaper status for the current process. Reaper status means that children orphaned by the reaper's descendants that were forked after the acquisition of reaper status are reparented to the reaper process. After system initialization, init(8) is the default reaper.

PROC_REAP_RELEASE — Release the reaper state for the current process. The reaper of the current process becomes the new reaper of the current process's descendants.

PROC_REAP_STATUS — Provides information about the reaper of the specified process, or the process itself when it is a reaper. The *data* argument must point to a *procctl_reaper_status* structure which is filled in by the syscall on successful return.

```
struct procctl_reaper_status {
        u_int   rs_flags;
        u_int   rs_children;
        u_int   rs_descendants;
        pid_t   rs_reaper;
        pid_t   rs_pid;
};
```

The *rs_flags* may have the following flags returned:

REAPER_STATUS_OWNED — The specified process has acquired reaper status and has not released it. When the flag is returned, the specified process *id*, pid, identifies the reaper, otherwise the *rs_reaper* field of the structure is set to the pid of the reaper for the specified process id.

REAPER_STATUS_REALINIT The specified process is the root of the reaper tree, i.e., init(8).

The *rs_children* field returns the number of children of the reaper among the descendants. It is possible to have a child whose reaper is not the specified process, since the reaper for any existing children is not reset on the PROC_REAP_ACQUIRE operation. The *rs_descendants* field returns the total number of descendants of the reaper(s), not counting descendants of the reaper in the subtree. The *rs_reaper* field returns the reaper pid. The *rs_pid* returns the pid of one reaper child if there are any descendants.

PROC_REAP_GETPIDS Queries the list of descendants of the reaper of the specified process. The request takes a pointer to a *procctl_reaper_pids* structure in the *data* parameter.

```
struct procctl_reaper_pids {
        u_int       rp_count;
        struct procctl_reaper_pidinfo *rp_pids;
};
```

When called, the *rp_pids* field must point to an array of *procctl_reaper_pidinfo* structures, to be filled in on return, and the *rp_count* field must specify the size of the array, into which no more than *rp_count* elements will be filled in by the kernel.

The *struct procctl_reaper_pidinfo* structure provides some information about one of the reaper's descendants. Note that for a descendant that is not a child, it may be incorrectly identified because of a race in which the original child process exited and the exited process's pid was reused for an unrelated process.

```
struct procctl_reaper_pidinfo {
        pid_t       pi_pid;
        pid_t       pi_subtree;
        u_int       pi_flags;
};
```

The *pi_pid* field is the process id of the descendant. The *pi_subtree* field provides the pid of the child of the reaper, which is the (grand-)parent of the process. The *pi_flags* field returns the following flags, further describing the descendant:

REAPER_PIDINFO_VALID — Set to indicate that the *procctl_reaper_pidinfo* structure was filled in by the kernel. Zero-filling the *rp_pids* array and testing the REAPER_PIDINFO_VALID flag allows the caller to detect the end of the returned array.

REAPER_PIDINFO_CHILD — The *pi_pid* field identifies the direct child of the reaper.

REAPER_PIDINFO_REAPER — The reported process is itself a reaper. The descendants of the subordinate reaper are not reported.

PROC_REAP_KILL — Request to deliver a signal to some subset of the descendants of the reaper. The *data* parameter must point to a *procctl_reaper_kill* structure, which is used both for parameters and status return.

```
struct procctl_reaper_kill {
        int     rk_sig;
        u_int   rk_flags;
        pid_t   rk_subtree;
        u_int   rk_killed;
        pid_t   rk_fpid;
};
```

The *rk_sig* field specifies the signal to be delivered. Zero is not a valid signal number, unlike for kill(2). The *rk_flags* field further directs the operation. It is or-ed from the following flags:

REAPER_KILL_CHILDREN — Deliver the specified signal only to direct children of the reaper.

REAPER_KILL_SUBTREE — Deliver the specified signal only to descendants that were forked by the direct child with pid specified in the *rk_subtree* field.

If neither the REAPER_KILL_CHILDREN nor the

REAPER_KILL_SUBTREE flags are specified, all current descendants of the reaper are signalled.

If a signal was delivered to any process, the return value from the request is zero. In this case, the *rk_killed* field identifies the number of processes signalled. The *rk_fpid* field is set to the pid of the first process for which signal delivery failed, e.g., due to permission problems. If no such process exists, the *rk_fpid* field is set to -1.

PROC_TRACE_CTL

Enable or disable tracing of the specified process(es), according to the value of the integer argument. Tracing includes attachment to the process using the ptrace(2) and ktrace(2), debugging sysctls, hwpmc(4), dtrace(1), and core dumping. Possible values for the *data* argument are:

PROC_TRACE_CTL_ENABLE

Enable tracing, after it was disabled by PROC_TRACE_CTL_DISABLE. Only allowed for self.

PROC_TRACE_CTL_DISABLE

Disable tracing for the specified process. Tracing is re-enabled when the process changes the executing program with the execve(2) syscall. A child inherits the trace settings from the parent on fork(2).

PROC_TRACE_CTL_DISABLE_EXEC

Same as PROC_TRACE_CTL_DISABLE, but the setting persists for the process even after execve(2).

PROC_TRACE_STATUS

Returns the current tracing status for the specified process in the integer variable pointed to by *data*. If tracing is disabled, *data* is set to -1. If tracing is enabled, but no debugger is attached by the ptrace(2)

syscall, *data* is set to 0. If a debugger is attached, *data* is set to the pid of the debugger process.

PROC_TRAPCAP_CTL

Controls the capability mode sandbox actions for the specified sandboxed processes, on a return from any syscall which gives either a ENOTCAPABLE or ECAPMODE error. If the control is enabled, such errors from the syscalls cause delivery of the synchronous SIGTRAP signal to the thread immediately before returning from the syscalls.

Possible values for the *data* argument are:

PROC_TRAPCAP_CTL_ENABLE

Enable the SIGTRAP signal delivery on capability mode access violations. The enabled mode is inherited by the children of the process, and is kept after fexecve(2) calls.

PROC_TRAPCAP_CTL_DISABLE

Disable the signal delivery on capability mode access violations. Note that the global sysctl kern.trap_enotcap might still cause the signal to be delivered. See capsicum(4).

On signal delivery, the *si_errno* member of the *siginfo* signal handler parameter is set to the syscall error value, and the *si_code* member is set to TRAP_CAP.

See capsicum(4) for more information about the capability mode.

PROC_TRAPCAP_STATUS

Return the current status of signalling capability mode access violations for the specified process. The integer value pointed to by the *data* argument is set to the PROC_TRAPCAP_CTL_ENABLE value if the process control enables signal delivery, and to PROC_TRAPCAP_CTL_DISABLE otherwise.

See the note about sysctl kern.trap_enotcap above, which gives independent global control of signal delivery.

PROC_PDEATHSIG_CTL — Request the delivery of a signal when the parent of the calling process exits. *idtype* must be P_PID and *id* must be the either caller's pid or zero, with no difference in effect. The value is cleared for child processes and when executing set-user-ID or set-group-ID binaries. *arg* must point to a value of type *int* indicating the signal that should be delivered to the caller. Use zero to cancel a previously requested signal delivery.

PROC_PDEATHSIG_STATUS — Query the current signal number that will be delivered when the parent of the calling process exits. *idtype* must be P_PID and *id* must be the either caller's pid or zero, with no difference in effect. *arg* must point to a memory location that can hold a value of type *int*. If signal delivery has not been requested, it will contain zero on return.

NOTES

Disabling tracing on a process should not be considered a security feature, as it is bypassable both by the kernel and privileged processes, and via other system mechanisms. As such, it should not be utilized to reliably protect cryptographic keying material or other confidential data.

RETURN VALUES

If an error occurs, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **procctl**() system call will fail if:

[EFAULT] The *arg* parameter points outside the process's allocated address space.

[EINVAL] The *cmd* argument specifies an unsupported command.

The *idtype* argument specifies an unsupported identifier type.

[EPERM] The calling process does not have permission to perform the requested operation on any of the selected processes.

[ESRCH] No processes matched the requested *idtype* and *id*.

[EINVAL] An invalid operation or flag was passed in *arg* for a PROC_SPROTECT

command.

[EPERM] The *idtype* argument is not equal to P_PID, or *id* is not equal to the pid of the calling process, for PROC_REAP_ACQUIRE or PROC_REAP_RELEASE requests.

[EINVAL] Invalid or undefined flags were passed to a PROC_REAP_KILL request.

[EINVAL] An invalid or zero signal number was requested for a PROC_REAP_KILL request.

[EINVAL] The PROC_REAP_RELEASE request was issued by the init(8) process.

[EBUSY] The PROC_REAP_ACQUIRE request was issued by a process that had already acquired reaper status and has not yet released it.

[EBUSY] The PROC_TRACE_CTL request was issued for a process already being traced.

[EPERM] The PROC_TRACE_CTL request to re-enable tracing of the process (PROC_TRACE_CTL_ENABLE), or to disable persistence of PROC_TRACE_CTL_DISABLE on execve(2) was issued for a non-current process.

[EINVAL] The value of the integer *data* parameter for the PROC_TRACE_CTL or PROC_TRAPCAP_CTL request is invalid.

[EINVAL] The PROC_PDEATHSIG_CTL or PROC_PDEATHSIG_STATUS request referenced an unsupported *id*, *idtype* or invalid signal number.

SEE ALSO

dtrace(1), cap_enter(2,) kill(2), ktrace(2), ptrace(2), wait(2), capsicum(4), hwpmc(4), init(8)

HISTORY

The **procctl**() function appeared in FreeBSD 10.0. The reaper facility is based on a similar feature of Linux and DragonflyBSD, and first appeared in FreeBSD 10.2. The PROC_PDEATHSIG_CTL facility is based on the prctl(PR_SET_PDEATHSIG, ...) feature of Linux, and first appeared in FreeBSD 11.2.

NAME

profil - control process profiling

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
profil(*char *samples*, *size_t size*, *vm_offset_t offset*, *int scale*);

DESCRIPTION

The **profil**() system call enables or disables program counter profiling of the current process. If profiling is enabled, then at every profiling clock tick, the kernel updates an appropriate count in the *samples* buffer. The frequency of the profiling clock is recorded in the header in the profiling output file.

The buffer *samples* contains *size* bytes and is divided into a series of 16-bit bins. Each bin counts the number of times the program counter was in a particular address range in the process when a profiling clock tick occurred while profiling was enabled. For a given program counter address, the number of the corresponding bin is given by the relation:

[(pc - offset) / 2] * scale / 65536

The *offset* argument is the lowest address at which the kernel takes program counter samples. The *scale* argument ranges from 1 to 65536 and can be used to change the span of the bins. A scale of 65536 maps each bin to 2 bytes of address range; a scale of 32768 gives 4 bytes, 16384 gives 8 bytes and so on. Intermediate values provide approximate intermediate ranges. A *scale* value of 0 disables profiling.

RETURN VALUES

The **profil**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

FILES

/usr/lib/gcrt0.o profiling C run-time startup file
gmon.out conventional name for profiling output file

ERRORS

The following error may be reported:

[EFAULT] The buffer *samples* contains an invalid address.

SEE ALSO

gprof(1)

HISTORY

The **profil**() function appeared in Version 6 AT&T UNIX.

BUGS

This routine should be named **profile**().

The *samples* argument should really be a vector of type *unsigned short*.

The format of the gmon.out file is undocumented.

NAME

pselect - synchronous I/O multiplexing a la POSIX.1g

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/select.h>

int
pselect(*int nfds*, *fd_set * restrict readfds*, *fd_set * restrict writefds*, *fd_set * restrict exceptfds*,
*const struct timespec * restrict timeout*, *const sigset_t * restrict newsigmask*);

DESCRIPTION

The **pselect**() function was introduced by IEEE Std 1003.1g-2000 ("POSIX.1g") as a slightly stronger version of select(2). The *nfds*, *readfds*, *writefds*, and *exceptfds* arguments are all identical to the analogous arguments of **select**(). The *timeout* argument in **pselect**() points to a *const struct timespec* rather than the (modifiable) *struct timeval* used by **select**(); as in **select**(), a null pointer may be passed to indicate that **pselect**() should wait indefinitely. Finally, *newsigmask* specifies a signal mask which is set while waiting for input. When **pselect**() returns, the original signal mask is restored.

See select(2) for a more detailed discussion of the semantics of this interface, and for macros used to manipulate the *fd_set* data type.

RETURN VALUES

The **pselect**() function returns the same values and under the same conditions as **select**().

ERRORS

The **pselect**() function may fail for any of the reasons documented for select(2) and (if a signal mask is provided) sigprocmask(2).

SEE ALSO

kqueue(2), poll(2), select(2), sigprocmask(2), sigsuspend(2)

STANDARDS

The **pselect**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **pselect**() function first appeared in FreeBSD 5.0.

AUTHORS

The first implementation of **pselect**() function and this manual page were written by Garrett Wollman <*wollman@FreeBSD.org*>.

NAME

psignal, **strsignal**, **sys_siglist**, **sys_signame** - system signal messages

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

void
psignal(*int sig*, *const char *s*);

*extern const char * const sys_siglist[]*;
*extern const char * const sys_signame[]*;

#include <string.h>

*char **
strsignal(*int sig*);

DESCRIPTION

The **psignal**() and **strsignal**() functions locate the descriptive message string for a signal number.

The **strsignal**() function accepts a signal number argument *sig* and returns a pointer to the corresponding message string.

The **psignal**() function accepts a signal number argument *sig* and writes it to the standard error. If the argument *s* is non-NULL and does not point to the null character, *s* is written to the standard error file descriptor prior to the message string, immediately followed by a colon and a space. If the signal number is not recognized (sigaction(2)), the string "Unknown signal" is produced.

The message strings can be accessed directly through the external array *sys_siglist*, indexed by recognized signal numbers. The external array *sys_signame* is used similarly and contains short, upper-case abbreviations for signals which are useful for recognizing signal names in user input. The defined variable NSIG contains a count of the strings in *sys_siglist* and *sys_signame*.

SEE ALSO

sigaction(2), perror(3), strerror(3)

HISTORY

The **psignal**() function appeared in 4.2BSD.

NAME

pthread - POSIX thread functions

LIBRARY

POSIX Threads Library (libpthread, -lpthread)

SYNOPSIS

#include <pthread.h>

DESCRIPTION

POSIX threads are a set of functions that support applications with requirements for multiple flows of control, called *threads*, within a process. Multithreading is used to improve the performance of a program.

The POSIX thread functions are summarized in this section in the following groups:

- Thread Routines
- Attribute Object Routines
- Mutex Routines
- Condition Variable Routines
- Read/Write Lock Routines
- Per-Thread Context Routines
- Cleanup Routines

Thread Routines

int **pthread_create**(*pthread_t *thread*, *const pthread_attr_t *attr*, *void *(*start_routine)(void *)*, *void *arg*)
Creates a new thread of execution.

int **pthread_cancel**(*pthread_t thread*)
Cancels execution of a thread.

int **pthread_detach**(*pthread_t thread*)
Marks a thread for deletion.

int **pthread_equal**(*pthread_t t1*, *pthread_t t2*)
Compares two thread IDs.

void **pthread_exit**(*void *value_ptr*)
Terminates the calling thread.

int **pthread_join**(*pthread_t thread*, *void **value_ptr*)
Causes the calling thread to wait for the termination of the specified thread.

int **pthread_kill**(*pthread_t thread*, *int sig*)
Delivers a signal to a specified thread.

int **pthread_once**(*pthread_once_t *once_control*, *void (*init_routine)(void)*)
Calls an initialization routine once.

pthread_t **pthread_self**(*void*)
Returns the thread ID of the calling thread.

int **pthread_setcancelstate**(*int state*, *int *oldstate*)
Sets the current thread's cancelability state.

int **pthread_setcanceltype**(*int type*, *int *oldtype*)
Sets the current thread's cancelability type.

void **pthread_testcancel**(*void*)
Creates a cancellation point in the calling thread.

void **pthread_yield**(*void*)
Allows the scheduler to run another thread instead of the current one.

Attribute Object Routines

int **pthread_attr_destroy**(*pthread_attr_t *attr*)
Destroy a thread attributes object.

int **pthread_attr_getinheritsched**(*const pthread_attr_t *attr*, *int *inheritsched*)
Get the inherit scheduling attribute from a thread attributes object.

int **pthread_attr_getschedparam**(*const pthread_attr_t *attr*, *struct sched_param *param*)
Get the scheduling parameter attribute from a thread attributes object.

int **pthread_attr_getschedpolicy**(*const pthread_attr_t *attr*, *int *policy*)
Get the scheduling policy attribute from a thread attributes object.

int **pthread_attr_getscope**(*const pthread_attr_t *attr*, *int *contentionscope*)
Get the contention scope attribute from a thread attributes object.

int **pthread_attr_getstacksize**(*const pthread_attr_t *attr*, *size_t *stacksize*)
Get the stack size attribute from a thread attributes object.

int **pthread_attr_getstackaddr**(*const pthread_attr_t *attr*, *void **stackaddr*)
Get the stack address attribute from a thread attributes object.

int **pthread_attr_getdetachstate**(*const pthread_attr_t *attr*, *int *detachstate*)
Get the detach state attribute from a thread attributes object.

int **pthread_attr_init**(*pthread_attr_t *attr*)
Initialize a thread attributes object with default values.

int **pthread_attr_setinheritsched**(*pthread_attr_t *attr*, *int inheritsched*)
Set the inherit scheduling attribute in a thread attributes object.

int **pthread_attr_setschedparam**(*pthread_attr_t *attr*, *const struct sched_param *param*)
Set the scheduling parameter attribute in a thread attributes object.

int **pthread_attr_setschedpolicy**(*pthread_attr_t *attr*, *int policy*)
Set the scheduling policy attribute in a thread attributes object.

int **pthread_attr_setscope**(*pthread_attr_t *attr*, *int contentionscope*)
Set the contention scope attribute in a thread attributes object.

int **pthread_attr_setstacksize**(*pthread_attr_t *attr*, *size_t stacksize*)
Set the stack size attribute in a thread attributes object.

int **pthread_attr_setstackaddr**(*pthread_attr_t *attr*, *void *stackaddr*)
Set the stack address attribute in a thread attributes object.

int **pthread_attr_setdetachstate**(*pthread_attr_t *attr*, *int detachstate*)
Set the detach state in a thread attributes object.

Mutex Routines

int **pthread_mutexattr_destroy**(*pthread_mutexattr_t *attr*)
Destroy a mutex attributes object.

int **pthread_mutexattr_getprioceiling**(*const pthread_mutexattr_t *restrict attr*, *int *restrict ceiling*)
Obtain priority ceiling attribute of mutex attribute object.

int **pthread_mutexattr_getprotocol**(*const pthread_mutexattr_t *restrict attr*, *int *restrict protocol*)
Obtain protocol attribute of mutex attribute object.

int **pthread_mutexattr_gettype**(*const pthread_mutexattr_t *restrict attr*, *int *restrict type*)
Obtain the mutex type attribute in the specified mutex attributes object.

int **pthread_mutexattr_init**(*pthread_mutexattr_t *attr*)
Initialize a mutex attributes object with default values.

int **pthread_mutexattr_setprioceiling**(*pthread_mutexattr_t *attr*, *int ceiling*)
Set priority ceiling attribute of mutex attribute object.

int **pthread_mutexattr_setprotocol**(*pthread_mutexattr_t *attr*, *int protocol*)
Set protocol attribute of mutex attribute object.

int **pthread_mutexattr_settype**(*pthread_mutexattr_t *attr*, *int type*)
Set the mutex type attribute that is used when a mutex is created.

int **pthread_mutex_destroy**(*pthread_mutex_t *mutex*)
Destroy a mutex.

int **pthread_mutex_init**(*pthread_mutex_t *mutex*, *const pthread_mutexattr_t *attr*)
Initialize a mutex with specified attributes.

int **pthread_mutex_lock**(*pthread_mutex_t *mutex*)
Lock a mutex and block until it becomes available.

int **pthread_mutex_timedlock**(*pthread_mutex_t *mutex*, *const struct timespec *abstime*)
Lock a mutex and block until it becomes available or until the timeout expires.

int **pthread_mutex_trylock**(*pthread_mutex_t *mutex*)
Try to lock a mutex, but do not block if the mutex is locked by another thread, including the current thread.

int **pthread_mutex_unlock**(*pthread_mutex_t *mutex*)
Unlock a mutex.

Condition Variable Routines

int **pthread_condattr_destroy**(*pthread_condattr_t *attr*)
Destroy a condition variable attributes object.

int **pthread_condattr_init**(*pthread_condattr_t *attr*)
Initialize a condition variable attributes object with default values.

int **pthread_cond_broadcast**(*pthread_cond_t *cond*)
Unblock all threads currently blocked on the specified condition variable.

int **pthread_cond_destroy**(*pthread_cond_t *cond*)
Destroy a condition variable.

int **pthread_cond_init**(*pthread_cond_t *cond*, *const pthread_condattr_t *attr*)
Initialize a condition variable with specified attributes.

int **pthread_cond_signal**(*pthread_cond_t *cond*)
Unblock at least one of the threads blocked on the specified condition variable.

int **pthread_cond_timedwait**(*pthread_cond_t *cond*, *pthread_mutex_t *mutex*,
*const struct timespec *abstime*)
Unlock the specified mutex, wait no longer than the specified time for a condition, and then relock the mutex.

int **pthread_cond_wait**(*pthread_cond_t **, *pthread_mutex_t *mutex*)
Unlock the specified mutex, wait for a condition, and relock the mutex.

Read/Write Lock Routines

int **pthread_rwlock_destroy**(*pthread_rwlock_t *lock*)
Destroy a read/write lock object.

int **pthread_rwlock_init**(*pthread_rwlock_t *lock*, *const pthread_rwlockattr_t *attr*)
Initialize a read/write lock object.

int **pthread_rwlock_rdlock**(*pthread_rwlock_t *lock*)
Lock a read/write lock for reading, blocking until the lock can be acquired.

int **pthread_rwlock_tryrdlock**(*pthread_rwlock_t *lock*)
Attempt to lock a read/write lock for reading, without blocking if the lock is unavailable.

int **pthread_rwlock_trywrlock**(*pthread_rwlock_t *lock*)
Attempt to lock a read/write lock for writing, without blocking if the lock is unavailable.

int **pthread_rwlock_unlock**(*pthread_rwlock_t *lock*)

Unlock a read/write lock.

int **pthread_rwlock_wrlock**(*pthread_rwlock_t *lock*)
Lock a read/write lock for writing, blocking until the lock can be acquired.

int **pthread_rwlockattr_destroy**(*pthread_rwlockattr_t *attr*)
Destroy a read/write lock attribute object.

int **pthread_rwlockattr_getpshared**(*const pthread_rwlockattr_t *attr*, *int *pshared*)
Retrieve the process shared setting for the read/write lock attribute object.

int **pthread_rwlockattr_init**(*pthread_rwlockattr_t *attr*)
Initialize a read/write lock attribute object.

int **pthread_rwlockattr_setpshared**(*pthread_rwlockattr_t *attr*, *int pshared*)
Set the process shared setting for the read/write lock attribute object.

Per-Thread Context Routines

int **pthread_key_create**(*pthread_key_t *key*, *void (*routine)(void *)*)
Create a thread-specific data key.

int **pthread_key_delete**(*pthread_key_t key*)
Delete a thread-specific data key.

*void ** **pthread_getspecific**(*pthread_key_t key*)
Get the thread-specific value for the specified key.

int **pthread_setspecific**(*pthread_key_t key*, *const void *value_ptr*)
Set the thread-specific value for the specified key.

Cleanup Routines

int **pthread_atfork**(*void (*prepare)(void)*, *void (*parent)(void)*, *void (*child)(void)*)
Register fork handlers.

void **pthread_cleanup_pop**(*int execute*)
Remove the routine at the top of the calling thread's cancellation cleanup stack and optionally invoke it.

void **pthread_cleanup_push**(*void (*routine)(void *)*, *void *routine_arg*)
Push the specified cancellation cleanup handler onto the calling thread's cancellation stack.

IMPLEMENTATION NOTES

The current FreeBSD POSIX thread implementation is built into the 1:1 Threading Library (libthr, -lthr) library. It contains thread-safe versions of Standard C Library (libc, -lc) functions and the thread functions. Threaded applications are linked with this library.

SEE ALSO

libthr(3), pthread_affinity_np(3), pthread_atfork(3), pthread_attr(3), pthread_cancel(3), pthread_cleanup_pop(3), pthread_cleanup_push(3), pthread_cond_broadcast(3), pthread_cond_destroy(3), pthread_cond_init(3), pthread_cond_signal(3), pthread_cond_timedwait(3), pthread_cond_wait(3), pthread_condattr_destroy(3), pthread_condattr_init(3), pthread_create(3), pthread_detach(3), pthread_equal(3), pthread_exit(3), pthread_getspecific(3), pthread_join(3), pthread_key_delete(3), pthread_kill(3), pthread_mutex_destroy(3), pthread_mutex_init(3), pthread_mutex_lock(3), pthread_mutex_trylock(3), pthread_mutex_unlock(3), pthread_mutexattr_destroy(3), pthread_mutexattr_getprioceiling(3), pthread_mutexattr_getprotocol(3), pthread_mutexattr_gettype(3), pthread_mutexattr_init(3), pthread_mutexattr_setprioceiling(3), pthread_mutexattr_setprotocol(3), pthread_mutexattr_settype(3), pthread_once(3), pthread_rwlock_destroy(3), pthread_rwlock_init(3), pthread_rwlock_rdlock(3), pthread_rwlock_unlock(3), pthread_rwlock_wrlock(3), pthread_rwlockattr_destroy(3), pthread_rwlockattr_getpshared(3), pthread_rwlockattr_init(3), pthread_rwlockattr_setpshared(3), pthread_self(3), pthread_setcancelstate(3), pthread_setcanceltype(3), pthread_setspecific(3), pthread_testcancel(3)

STANDARDS

The functions with the **pthread_** prefix and not **_np** suffix or **pthread_rwlock** prefix conform to ISO/IEC 9945-1:1996 ("POSIX.1").

The functions with the **pthread_** prefix and **_np** suffix are non-portable extensions to POSIX threads.

The functions with the **pthread_rwlock** prefix are extensions created by The Open Group as part of the Version 2 of the Single UNIX Specification ("SUSv2").

NAME

ptrace - process tracing and debugging

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ptrace.h>

int
ptrace(*int request*, *pid_t pid*, *caddr_t addr*, *int data*);

DESCRIPTION

The **ptrace**() system call provides tracing and debugging facilities. It allows one process (the *tracing* process) to control another (the *traced* process). The tracing process must first attach to the traced process, and then issue a series of **ptrace**() system calls to control the execution of the process, as well as access process memory and register state. For the duration of the tracing session, the traced process will be "re-parented", with its parent process ID (and resulting behavior) changed to the tracing process. It is permissible for a tracing process to attach to more than one other process at a time. When the tracing process has completed its work, it must detach the traced process; if a tracing process exits without first detaching all processes it has attached, those processes will be killed.

Most of the time, the traced process runs normally, but when it receives a signal (see sigaction(2)), it stops. The tracing process is expected to notice this via wait(2) or the delivery of a SIGCHLD signal, examine the state of the stopped process, and cause it to terminate or continue as appropriate. The signal may be a normal process signal, generated as a result of traced process behavior, or use of the kill(2) system call; alternatively, it may be generated by the tracing facility as a result of attaching, stepping by the tracing process, or an event in the traced process. The tracing process may choose to intercept the signal, using it to observe process behavior (such as SIGTRAP), or forward the signal to the process if appropriate. The **ptrace**() system call is the mechanism by which all this happens.

A traced process may report additional signal stops corresponding to events in the traced process. These additional signal stops are reported as SIGTRAP or SIGSTOP signals. The tracing process can use the PT_LWPINFO request to determine which events are associated with a SIGTRAP or SIGSTOP signal. Note that multiple events may be associated with a single signal. For example, events indicated by the PL_FLAG_BORN, PL_FLAG_FORKED, and PL_FLAG_EXEC flags are also reported as a system call exit event (PL_FLAG_SCX). The signal stop for a new child process enabled via PTRACE_FORK will report a SIGSTOP signal. All other additional signal stops use SIGTRAP.

Each traced process has a tracing event mask. An event in the traced process only reports a signal stop if the corresponding flag is set in the tracing event mask. The current set of tracing event flags include:

PTRACE_EXEC
: Report a stop for a successful invocation of execve(2). This event is indicated by the PL_FLAG_EXEC flag in the *pl_flags* member of *struct ptrace_lwpinfo*.

PTRACE_SCE
: Report a stop on each system call entry. This event is indicated by the PL_FLAG_SCE flag in the *pl_flags* member of *struct ptrace_lwpinfo*.

PTRACE_SCX
: Report a stop on each system call exit. This event is indicated by the PL_FLAG_SCX flag in the *pl_flags* member of *struct ptrace_lwpinfo*.

PTRACE_SYSCALL
: Report stops for both system call entry and exit.

PTRACE_FORK
: This event flag controls tracing for new child processes of a traced process.

  When this event flag is enabled, new child processes will enable tracing and stop before executing their first instruction. The new child process will include the PL_FLAG_CHILD flag in the *pl_flags* member of *struct ptrace_lwpinfo*. The traced process will report a stop that includes the PL_FLAG_FORKED flag. The process ID of the new child process will also be present in the *pl_child_pid* member of *struct ptrace_lwpinfo*. If the new child process was created via vfork(2), the traced process's stop will also include the PL_FLAG_VFORKED flag. Note that new child processes will be attached with the default tracing event mask; they do not inherit the event mask of the traced process.

  When this event flag is not enabled, new child processes will execute without tracing enabled.

PTRACE_LWP
: This event flag controls tracing of LWP (kernel thread) creation and destruction. When this event is enabled, new LWPs will stop and report an event with PL_FLAG_BORN set before executing their first instruction, and exiting LWPs will stop and report an event with PL_FLAG_EXITED set before completing their termination.

  Note that new processes do not report an event for the creation of their initial thread, and exiting processes do not report an event for the termination of the last thread.

PTRACE_VFORK Report a stop event when a parent process resumes after a vfork(2).

When a thread in the traced process creates a new child process via vfork(2), the stop that reports PL_FLAG_FORKED and PL_FLAG_SCX occurs just after the child process is created, but before the thread waits for the child process to stop sharing process memory. If a debugger is not tracing the new child process, it must ensure that no breakpoints are enabled in the shared process memory before detaching from the new child process. This means that no breakpoints are enabled in the parent process either.

The PTRACE_VFORK flag enables a new stop that indicates when the new child process stops sharing the process memory of the parent process. A debugger can reinsert breakpoints in the parent process and resume it in response to this event. This event is indicated by setting the PL_FLAG_VFORK_DONE flag.

The default tracing event mask when attaching to a process via PT_ATTACH, PT_TRACE_ME, or PTRACE_FORK includes only PTRACE_EXEC events. All other event flags are disabled.

The *request* argument specifies what operation is being performed; the meaning of the rest of the arguments depends on the operation, but except for one special case noted below, all **ptrace**() calls are made by the tracing process, and the *pid* argument specifies the process ID of the traced process or a corresponding thread ID. The *request* argument can be:

PT_TRACE_ME This request is the only one used by the traced process; it declares that the process expects to be traced by its parent. All the other arguments are ignored. (If the parent process does not expect to trace the child, it will probably be rather confused by the results; once the traced process stops, it cannot be made to continue except via **ptrace**().) When a process has used this request and calls execve(2) or any of the routines built on it (such as execv(3)), it will stop before executing the first instruction of the new image. Also, any setuid or setgid bits on the executable being executed will be ignored. If the child was created by vfork(2) system call or rfork(2) call with the RFMEM flag specified, the debugging events are reported to the parent only after the execve(2) is executed.

PT_READ_I, PT_READ_D These requests read a single *int* of data from the traced process's address space. Traditionally, **ptrace**() has allowed for machines with distinct address spaces for instruction and data, which is why there are two requests: conceptually, PT_READ_I reads from the instruction space

and PT_READ_D reads from the data space. In the current FreeBSD implementation, these two requests are completely identical. The *addr* argument specifies the address (in the traced process's virtual address space) at which the read is to be done. This address does not have to meet any alignment constraints. The value read is returned as the return value from **ptrace**().

PT_WRITE_I, PT_WRITE_D These requests parallel PT_READ_I and PT_READ_D, except that they write rather than read. The *data* argument supplies the value to be written.

PT_IO This request allows reading and writing arbitrary amounts of data in the traced process's address space. The *addr* argument specifies a pointer to a *struct ptrace_io_desc*, which is defined as follows:

```
struct ptrace_io_desc {
        int     piod_op;  /* I/O operation */
        void    *piod_offs;     /* child offset */
        void    *piod_addr;     /* parent offset */
        size_t  piod_len; /* request length */
};

/*
 * Operations in piod_op.
 */
#define PIOD_READ_D     1       /* Read from D space */
#define PIOD_WRITE_D    2       /* Write to D space */
#define PIOD_READ_I     3       /* Read from I space */
#define PIOD_WRITE_I    4       /* Write to I space */
```

The *data* argument is ignored. The actual number of bytes read or written is stored in *piod_len* upon return.

PT_CONTINUE The traced process continues execution. The *addr* argument is an address specifying the place where execution is to be resumed (a new value for the program counter), or (*caddr_t*)1 to indicate that execution is to pick up where it left off. The *data* argument provides a signal number to be delivered to the traced process as it resumes execution, or 0 if no signal is to be sent.

PT_STEP — The traced process is single stepped one instruction. The *addr* argument should be passed (*caddr_t*)1. The *data* argument provides a signal number to be delivered to the traced process as it resumes execution, or 0 if no signal is to be sent.

PT_KILL — The traced process terminates, as if PT_CONTINUE had been used with SIGKILL given as the signal to be delivered.

PT_ATTACH — This request allows a process to gain control of an otherwise unrelated process and begin tracing it. It does not need any cooperation from the to-be-traced process. In this case, *pid* specifies the process ID of the to-be-traced process, and the other two arguments are ignored. This request requires that the target process must have the same real UID as the tracing process, and that it must not be executing a setuid or setgid executable. (If the tracing process is running as root, these restrictions do not apply.) The tracing process will see the newly-traced process stop and may then control it as if it had been traced all along.

PT_DETACH — This request is like PT_CONTINUE, except that it does not allow specifying an alternate place to continue execution, and after it succeeds, the traced process is no longer traced and continues execution normally.

PT_GETREGS — This request reads the traced process's machine registers into the "*struct reg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_SETREGS — This request is the converse of PT_GETREGS; it loads the traced process's machine registers from the "*struct reg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_GETFPREGS — This request reads the traced process's floating-point registers into the "*struct fpreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_SETFPREGS — This request is the converse of PT_GETFPREGS; it loads the traced process's floating-point registers from the "*struct fpreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_GETDBREGS — This request reads the traced process's debug registers into the "*struct dbreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_SETDBREGS — This request is the converse of PT_GETDBREGS; it loads the traced

process's debug registers from the "*struct dbreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_LWPINFO This request can be used to obtain information about the kernel thread, also known as light-weight process, that caused the traced process to stop. The *addr* argument specifies a pointer to a *struct ptrace_lwpinfo*, which is defined as follows:

```
struct ptrace_lwpinfo {
        lwpid_t pl_lwpid;
        int     pl_event;
        int     pl_flags;
        sigset_t pl_sigmask;
        sigset_t pl_siglist;
        siginfo_t pl_siginfo;
        char    pl_tdname[MAXCOMLEN + 1];
        pid_t   pl_child_pid;
        u_int   pl_syscall_code;
        u_int   pl_syscall_narg;
};
```

The *data* argument is to be set to the size of the structure known to the caller. This allows the structure to grow without affecting older programs.

The fields in the *struct ptrace_lwpinfo* have the following meaning:

pl_lwpid
: LWP id of the thread

pl_event
: Event that caused the stop. Currently defined events are:

PL_EVENT_NONE	No reason given
PL_EVENT_SIGNAL	Thread stopped due to the pending signal

pl_flags
: Flags that specify additional details about observed stop. Currently defined flags are:

PL_FLAG_SCE
: The thread stopped due to system call entry, right after the kernel is entered. The debugger may examine syscall arguments that are stored in memory and registers

according to the ABI of the current process, and modify them, if needed.

PL_FLAG_SCX
: The thread is stopped immediately before syscall is returning to the usermode. The debugger may examine system call return values in the ABI-defined registers and/or memory.

PL_FLAG_EXEC
: When PL_FLAG_SCX is set, this flag may be additionally specified to inform that the program being executed by debuggee process has been changed by successful execution of a system call from the **execve**(*2*) family.

PL_FLAG_SI
: Indicates that *pl_siginfo* member of *struct ptrace_lwpinfo* contains valid information.

PL_FLAG_FORKED
: Indicates that the process is returning from a call to **fork**(*2*) that created a new child process. The process identifier of the new process is available in the *pl_child_pid* member of *struct ptrace_lwpinfo*.

PL_FLAG_CHILD
: The flag is set for first event reported from a new child which is automatically attached when PTRACE_FORK is enabled.

PL_FLAG_BORN
: This flag is set for the first event reported from a new LWP when PTRACE_LWP is enabled. It is reported along with PL_FLAG_SCX.

PL_FLAG_EXITED
: This flag is set for the last event reported by an exiting LWP when PTRACE_LWP is enabled. Note that this event is not reported when the last LWP in a process exits. The termination of the last thread is reported via a normal process exit event.

PL_FLAG_VFORKED
: Indicates that the thread is returning from a call to vfork(2) that created a new child process. This flag is set in addition to PL_FLAG_FORKED.

PL_FLAG_VFORK_DONE

Indicates that the thread has resumed after a child process created via vfork(2) has stopped sharing its address space with the traced process.

pl_sigmask
The current signal mask of the LWP

pl_siglist
The current pending set of signals for the LWP. Note that signals that are delivered to the process would not appear on an LWP siglist until the thread is selected for delivery.

pl_siginfo
The siginfo that accompanies the signal pending. Only valid for PL_EVENT_SIGNAL stop when PL_FLAG_SI is set in *pl_flags*.

pl_tdname
The name of the thread.

pl_child_pid
The process identifier of the new child process. Only valid for a PL_EVENT_SIGNAL stop when PL_FLAG_FORKED is set in *pl_flags*.

pl_syscall_code
The ABI-specific identifier of the current system call. Note that for indirect system calls this field reports the indirected system call. Only valid when PL_FLAG_SCE or PL_FLAG_SCX is set in *pl_flags*.

pl_syscall_narg
The number of arguments passed to the current system call not counting the system call identifier. Note that for indirect system calls this field reports the arguments passed to the indirected system call. Only valid when PL_FLAG_SCE or PL_FLAG_SCX is set in *pl_flags*.

PT_GETNUMLWPS This request returns the number of kernel threads associated with the traced process.

PT_GETLWPLIST This request can be used to get the current thread list. A pointer to an array of type *lwpid_t* should be passed in *addr*, with the array size specified by *data*. The return value from **ptrace**() is the count of array entries filled in.

PT_SETSTEP This request will turn on single stepping of the specified process.

Stepping is automatically disabled when a single step trap is caught.

PT_CLEARSTEP — This request will turn off single stepping of the specified process.

PT_SUSPEND — This request will suspend the specified thread.

PT_RESUME — This request will resume the specified thread.

PT_TO_SCE — This request will set the PTRACE_SCE event flag to trace all future system call entries and continue the process. The *addr* and *data* arguments are used the same as for PT_CONTINUE.

PT_TO_SCX — This request will set the PTRACE_SCX event flag to trace all future system call exits and continue the process. The *addr* and *data* arguments are used the same as for PT_CONTINUE.

PT_SYSCALL — This request will set the PTRACE_SYSCALL event flag to trace all future system call entries and exits and continue the process. The *addr* and *data* arguments are used the same as for PT_CONTINUE.

PT_GET_SC_ARGS — For the thread which is stopped in either PL_FLAG_SCE or PL_FLAG_SCX state, that is, on entry or exit to a syscall, this request fetches the syscall arguments.

The arguments are copied out into the buffer pointed to by the *addr* pointer, sequentially. Each syscall argument is stored as the machine word. Kernel copies out as many arguments as the syscall accepts, see the *pl_syscall_narg* member of the *struct ptrace_lwpinfo*, but not more than the *data* bytes in total are copied.

PT_FOLLOW_FORK — This request controls tracing for new child processes of a traced process. If *data* is non-zero, PTRACE_FORK is set in the traced process's event tracing mask. If *data* is zero, PTRACE_FORK is cleared from the traced process's event tracing mask.

PT_LWP_EVENTS — This request controls tracing of LWP creation and destruction. If *data* is non-zero, PTRACE_LWP is set in the traced process's event tracing mask. If *data* is zero, PTRACE_LWP is cleared from the traced process's event tracing mask.

PT_GET_EVENT_MASK This request reads the traced process's event tracing mask into the integer pointed to by *addr*. The size of the integer must be passed in *data*.

PT_SET_EVENT_MASK This request sets the traced process's event tracing mask from the integer pointed to by *addr*. The size of the integer must be passed in *data*.

PT_VM_TIMESTAMP This request returns the generation number or timestamp of the memory map of the traced process as the return value from **ptrace**(). This provides a low-cost way for the tracing process to determine if the VM map changed since the last time this request was made.

PT_VM_ENTRY This request is used to iterate over the entries of the VM map of the traced process. The *addr* argument specifies a pointer to a *struct ptrace_vm_entry*, which is defined as follows:

```
struct ptrace_vm_entry {
        int             pve_entry;
        int             pve_timestamp;
        u_long          pve_start;
        u_long          pve_end;
        u_long          pve_offset;
        u_int           pve_prot;
        u_int           pve_pathlen;
        long            pve_fileid;
        uint32_t        pve_fsid;
        char            *pve_path;
};
```

The first entry is returned by setting *pve_entry* to zero. Subsequent entries are returned by leaving *pve_entry* unmodified from the value returned by previous requests. The *pve_timestamp* field can be used to detect changes to the VM map while iterating over the entries. The tracing process can then take appropriate action, such as restarting. By setting *pve_pathlen* to a non-zero value on entry, the pathname of the backing object is returned in the buffer pointed to by *pve_path*, provided the entry is backed by a vnode. The *pve_pathlen* field is updated with the actual length of the pathname (including the terminating null character). The *pve_offset* field is the offset within the backing object at

which the range starts. The range is located in the VM space at *pve_start* and extends up to *pve_end* (inclusive).

The *data* argument is ignored.

ARM MACHINE-SPECIFIC REQUESTS

PT_GETVFPREGS Return the thread's VFP machine state in the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_SETVFPREGS Set the thread's VFP machine state from the buffer pointed to by *addr*.

The *data* argument is ignored.

x86 MACHINE-SPECIFIC REQUESTS

PT_GETXMMREGS Copy the XMM FPU state into the buffer pointed to by the argument *addr*. The buffer has the same layout as the 32-bit save buffer for the machine instruction FXSAVE.

This request is only valid for i386 programs, both on native 32-bit systems and on amd64 kernels. For 64-bit amd64 programs, the XMM state is reported as part of the FPU state returned by the PT_GETFPREGS request.

The *data* argument is ignored.

PT_SETXMMREGS Load the XMM FPU state for the thread from the buffer pointed to by the argument *addr*. The buffer has the same layout as the 32-bit load buffer for the machine instruction FXRSTOR.

As with PT_GETXMMREGS, this request is only valid for i386 programs.

The *data* argument is ignored.

PT_GETXSTATE_INFO Report which XSAVE FPU extensions are supported by the CPU and allowed in userspace programs. The *addr* argument must point to a variable of type *struct ptrace_xstate_info*, which contains the information on the request return. *struct ptrace_xstate_info* is defined as follows:

```
struct ptrace_xstate_info {
        uint64_t   xsave_mask;
        uint32_t   xsave_len;
};
```

The xsave_mask field is a bitmask of the currently enabled extensions. The meaning of the bits is defined in the Intel and AMD processor documentation. The xsave_len field reports the length of the XSAVE area for storing the hardware state for currently enabled extensions in the format defined by the x86 XSAVE machine instruction.

The *data* argument value must be equal to the size of the *struct ptrace_xstate_info*.

PT_GETXSTATE — Return the content of the XSAVE area for the thread. The *addr* argument points to the buffer where the content is copied, and the *data* argument specifies the size of the buffer. The kernel copies out as much content as allowed by the buffer size. The buffer layout is specified by the layout of the save area for the XSAVE machine instruction.

PT_SETXSTATE — Load the XSAVE state for the thread from the buffer specified by the *addr* pointer. The buffer size is passed in the *data* argument. The buffer must be at least as large as the *struct savefpu* (defined in *x86/fpu.h*) to allow the complete x87 FPU and XMM state load. It must not be larger than the XSAVE state length, as reported by the xsave_len field from the *struct ptrace_xstate_info* of the PT_GETXSTATE_INFO request. Layout of the buffer is identical to the layout of the load area for the XRSTOR machine instruction.

PT_GETFSBASE — Return the value of the base used when doing segmented memory addressing using the %fs segment register. The *addr* argument points to an *unsigned long* variable where the base value is stored.

The *data* argument is ignored.

PT_GETGSBASE — Like the PT_GETFSBASE request, but returns the base for the %gs segment register.

PT_SETFSBASE — Set the base for the %fs segment register to the value pointed to by the *addr* argument. *addr* must point to the *unsigned long* variable containing the new base.

The *data* argument is ignored.

PT_SETGSBASE Like the PT_SETFSBASE request, but sets the base for the %gs segment register.

PowerPC MACHINE-SPECIFIC REQUESTS

PT_GETVRREGS Return the thread's ALTIVEC machine state in the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_SETVRREGS Set the thread's ALTIVEC machine state from the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_GETVSRREGS Return doubleword 1 of the thread's VSX registers VSR0-VSR31 in the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_SETVSRREGS Set doubleword 1 of the thread's VSX registers VSR0-VSR31 from the buffer pointed to by *addr*.

The *data* argument is ignored.

Additionally, other machine-specific requests can exist.

RETURN VALUES

Most requests return 0 on success and -1 on error. Some requests can cause **ptrace**() to return -1 as a non-error value, among them are PT_READ_I and PT_READ_D, which return the value read from the process memory on success. To disambiguate, *errno* can be set to 0 before the call and checked afterwards.

The current **ptrace**() implementation always sets *errno* to 0 before calling into the kernel, both for historic reasons and for consistency with other operating systems. It is recommended to assign zero to *errno* explicitly for forward compatibility.

ERRORS

The **ptrace**() system call may fail if:

[ESRCH]

- No process having the specified process ID exists.

[EINVAL]

- A process attempted to use PT_ATTACH on itself.
- The *request* argument was not one of the legal requests.
- The signal number (in *data*) to PT_CONTINUE was neither 0 nor a legal signal number.
- PT_GETREGS, PT_SETREGS, PT_GETFPREGS, PT_SETFPREGS, PT_GETDBREGS, or PT_SETDBREGS was attempted on a process with no valid register set. (This is normally true only of system processes.)
- PT_VM_ENTRY was given an invalid value for *pve_entry*. This can also be caused by changes to the VM map of the process.
- The size (in *data*) provided to PT_LWPINFO was less than or equal to zero, or larger than the *ptrace_lwpinfo* structure known to the kernel.
- The size (in *data*) provided to the x86-specific PT_GETXSTATE_INFO request was not equal to the size of the *struct ptrace_xstate_info*.
- The size (in *data*) provided to the x86-specific PT_SETXSTATE request was less than the size of the x87 plus the XMM save area.
- The size (in *data*) provided to the x86-specific PT_SETXSTATE request was larger than returned in the xsave_len member of the *struct ptrace_xstate_info* from the PT_GETXSTATE_INFO request.
- The base value, provided to the amd64-specific requests PT_SETFSBASE or PT_SETGSBASE, pointed outside of the valid user address space. This error will not occur in 32-bit programs.

[EBUSY]

- PT_ATTACH was attempted on a process that was already being traced.
- A request attempted to manipulate a process that was being traced by some process other than the one making the request.
- A request (other than PT_ATTACH) specified a process that was not stopped.

[EPERM]

- A request (other than PT_ATTACH) attempted to manipulate a process that was not being traced at all.
- An attempt was made to use PT_ATTACH on a process in violation of the requirements listed under PT_ATTACH above.

[ENOENT]

- PT_VM_ENTRY previously returned the last entry of the memory map. No more entries exist.

[ENAMETOOLONG]

- PT_VM_ENTRY cannot return the pathname of the backing object because the buffer is not big enough. *pve_pathlen* holds the minimum buffer size required on return.

SEE ALSO

execve(2), sigaction(2), wait(2), execv(3), i386_clr_watch(3), i386_set_watch(3)

HISTORY

The **ptrace**() function appeared in Version 6 AT&T UNIX.

NAME

write, **writev**, **pwrite**, **pwritev** - write output

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

ssize_t
write(*int fd*, *const void *buf*, *size_t nbytes*);

ssize_t
pwrite(*int fd*, *const void *buf*, *size_t nbytes*, *off_t offset*);

#include <sys/uio.h>

ssize_t
writev(*int fd*, *const struct iovec *iov*, *int iovcnt*);

ssize_t
pwritev(*int fd*, *const struct iovec *iov*, *int iovcnt*, *off_t offset*);

DESCRIPTION

The **write**() system call attempts to write *nbytes* of data to the object referenced by the descriptor *fd* from the buffer pointed to by *buf*. The **writev**() system call performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt-1]. The **pwrite**() and **pwritev**() system calls perform the same functions, but write to the specified position in the file without modifying the file pointer.

For **writev**() and **pwritev**(), the *iovec* structure is defined as:

```
struct iovec {
        void   *iov_base;  /* Base address. */
        size_t iov_len;    /* Length. */
};
```

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. The **writev**() system call will always write a complete area before proceeding to the next.

On objects capable of seeking, the **write**() starts at a position given by the pointer associated with *fd*, see lseek(2). Upon return from **write**(), the pointer is incremented by the number of bytes which were written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

If the real user is not the super-user, then **write**() clears the set-user-id bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-id file owned by the super-user.

When using non-blocking I/O on objects such as sockets that are subject to flow control, **write**() and **writev**() may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

RETURN VALUES

Upon successful completion the number of bytes which were written is returned. Otherwise a -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **write**(), **writev**(), **pwrite**() and **pwritev**() system calls will fail and the file pointer will remain unchanged if:

[EBADF]	The *fd* argument is not a valid descriptor open for writing.
[EPIPE]	An attempt is made to write to a pipe that is not open for reading by any process.
[EPIPE]	An attempt is made to write to a socket of type SOCK_STREAM that is not connected to a peer socket.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
[EFAULT]	Part of *iov* or data to be written to the file points outside the process's allocated address space.
[EINVAL]	The pointer associated with *fd* was negative.
[ENOSPC]	There is no free space remaining on the file system containing the file.

[EDQUOT] The user's quota of disk blocks on the file system containing the file has been exhausted.

[EIO] An I/O error occurred while reading from or writing to the file system.

[EINTR] A signal interrupted the write before it could be completed.

[EAGAIN] The file was marked for non-blocking I/O, and no data could be written immediately.

[EROFS] An attempt was made to write over a disk label area at the beginning of a slice. Use disklabel(8) **-W** to enable writing on the disk label area.

[EINVAL] The value *nbytes* is greater than SSIZE_MAX (or greater than INT_MAX, if the sysctl *debug.iosize_max_clamp* is non-zero).

In addition, **writev**() and **pwritev**() may return one of the following errors:

[EDESTADDRREQ] The destination is no longer available when writing to a UNIX domain datagram socket on which connect(2) had been used to set a destination address.

[EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than IOV_MAX.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

[ENOBUFS] The mbuf pool has been completely exhausted when writing to a socket.

The **pwrite**() and **pwritev**() system calls may also return the following errors:

[EINVAL] The *offset* value was negative.

[ESPIPE] The file descriptor is associated with a pipe, socket, or FIFO.

SEE ALSO

fcntl(2), lseek(2), open(2), pipe(2), select(2)

STANDARDS

The **write**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The **writev**() and **pwrite**() system calls are expected to conform to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

HISTORY

The **pwritev**() system call appeared in FreeBSD 6.0. The **pwrite**() function appeared in AT&T System V Release 4 UNIX. The **writev**() system call appeared in 4.2BSD. The **write**() function appeared in Version 1 AT&T UNIX.

NAME

querylocale - Look up the locale name for a specified category

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <locale.h>

*const char **
querylocale(*int mask*, *locale_t locale*);

DESCRIPTION

Returns the name of the locale for the category specified by *mask*. This possible values for the mask are the same as those in newlocale(3). If more than one bit in the mask is set, the returned value is undefined.

SEE ALSO

duplocale(3), freelocale(3), localeconv(3), newlocale(3), uselocale(3), xlocale(3)

NAME

quick_exit - exits a program quickly, running minimal cleanup

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

_Noreturn void
quick_exit(*int status*);

DESCRIPTION

The **quick_exit**() function exits the program quickly calling any cleanup functions registered with at_quick_exit(3) but not any C++ destructors or cleanup code registered with atexit(3). The stdio(3) file buffers are not flushed.

The function **quick_exit**() is *async-signal safe* when the functions registered with at_quick_exit(3) are.

RETURN VALUES

The **quick_exit**() function does not return.

SEE ALSO

at_quick_exit(3), exit(3)

STANDARDS

The **quick_exit**() function conforms to ISO/IEC 9899:2011 ("ISO C11").

NAME

quotactl - manipulate file system quotas

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <ufs/ufs/quota.h>

int
quotactl(*const char *path*, *int cmd*, *int id*, *void *addr*);

DESCRIPTION

The **quotactl**() system call enables, disables and manipulates file system quotas. A quota control command given by *cmd* operates on the given filename *path* for the given user or group *id*. (NOTE: One should use the QCMD macro defined in <*ufs/ufs/quota.h*> to formulate the value for *cmd*.) The address of an optional command specific data structure, *addr*, may be given; its interpretation is discussed below with each command.

For commands that use the *id* identifier, it must be either -1 or any positive value. The value of -1 indicates that the current UID or GID should be used. Any other negative value will return an error.

Currently quotas are supported only for the "ufs" file system. For "ufs", a command is composed of a primary command (see below) and a command type used to interpret the *id*. Types are supported for interpretation of user identifiers (USRQUOTA) and group identifiers (GRPQUOTA). The "ufs" specific commands are:

Q_QUOTAON — Enable disk quotas for the file system specified by *path*. The command type specifies the type of the quotas being enabled. The *addr* argument specifies a file from which to take the quotas. The quota file must exist; it is normally created with the quotacheck(8) program. The *id* argument is unused. Only the super-user may turn quotas on.

Q_QUOTAOFF — Disable disk quotas for the file system specified by *path*. The command type specifies the type of the quotas being disabled. The *addr* and *id* arguments are unused. Only the super-user may turn quotas off.

Q_GETQUOTASIZE — Get the wordsize used to represent the quotas for the user or group (as determined by the command type). Possible values are 32 for the old-style

quota file and 64 for the new-style quota file. The *addr* argument is a pointer to an integer into which the size is stored. The identifier *id* is not used.

Q_GETQUOTA Get disk quota limits and current usage for the user or group (as determined by the command type) with identifier *id*. The *addr* argument is a pointer to a *struct dqblk* structure (defined in <*ufs/ufs/quota.h*>).

Q_SETQUOTA Set disk quota limits for the user or group (as determined by the command type) with identifier *id*. The *addr* argument is a pointer to a *struct dqblk* structure (defined in <*ufs/ufs/quota.h*>). The usage fields of the *dqblk* structure are ignored. This system call is restricted to the super-user.

Q_SETUSE Set disk usage limits for the user or group (as determined by the command type) with identifier *id*. The *addr* argument is a pointer to a *struct dqblk* structure (defined in <*ufs/ufs/quota.h*>). Only the usage fields are used. This system call is restricted to the super-user.

Q_SYNC Update the on-disk copy of quota usages. The command type specifies which type of quotas are to be updated. The *id* and *addr* arguments are ignored.

RETURN VALUES

The **quotactl**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **quotactl**() system call will fail if:

[EOPNOTSUPP] The kernel has not been compiled with the QUOTA option.

[EUSERS] The quota table cannot be expanded.

[EINVAL] The *cmd* argument or the command type is invalid. In Q_GETQUOTASIZE, Q_GETQUOTA, Q_SETQUOTA, and Q_SETUSE, quotas are not currently enabled for this file system.

The *id* argument to Q_GETQUOTA, Q_SETQUOTA or Q_SETUSE is a negative value.

[EACCES] In Q_QUOTAON, the quota file is not a plain file.

[EACCES] Search permission is denied for a component of a path prefix.

[ENOTDIR] A component of a path prefix was not a directory.

[ENAMETOOLONG]
A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

[ENOENT] A filename does not exist.

[ELOOP] Too many symbolic links were encountered in translating a pathname.

[EROFS] In Q_QUOTAON, either the file system on which quotas are to be enabled is mounted read-only or the quota file resides on a read-only file system.

[EIO] An I/O error occurred while reading from or writing to a file containing quotas.

[EFAULT] An invalid *addr* was supplied; the associated structure could not be copied in or out of the kernel.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EPERM] The call was privileged and the caller was not the super-user.

SEE ALSO

quota(1), fstab(5), edquota(8), quotacheck(8), quotaon(8), repquota(8)

HISTORY

The **quotactl**() system call appeared in 4.3BSD-Reno.

BUGS

There should be some way to integrate this call with the resource limit interface provided by setrlimit(2) and getrlimit(2).

NAME

radixsort, **sradixsort** - radix sort

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <limits.h>
#include <stdlib.h>

int
radixsort(*const unsigned char **base*, *int nmemb*, *const unsigned char *table*, *unsigned endbyte*);

int
sradixsort(*const unsigned char **base*, *int nmemb*, *const unsigned char *table*, *unsigned endbyte*);

DESCRIPTION

The **radixsort**() and **sradixsort**() functions are implementations of radix sort.

These functions sort an array of pointers to byte strings, the initial member of which is referenced by *base*. The byte strings may contain any values; the end of each string is denoted by the user-specified value *endbyte*.

Applications may specify a sort order by providing the *table* argument. If non-NULL, *table* must reference an array of UCHAR_MAX + 1 bytes which contains the sort weight of each possible byte value. The end-of-string byte must have a sort weight of 0 or 255 (for sorting in reverse order). More than one byte may have the same sort weight. The *table* argument is useful for applications which wish to sort different characters equally, for example, providing a table with the same weights for A-Z as for a-z will result in a case-insensitive sort. If *table* is NULL, the contents of the array are sorted in ascending order according to the ASCII order of the byte strings they reference and *endbyte* has a sorting weight of 0.

The **sradixsort**() function is stable, that is, if two elements compare as equal, their order in the sorted array is unchanged. The **sradixsort**() function uses additional memory sufficient to hold *nmemb* pointers.

The **radixsort**() function is not stable, but uses no additional memory.

These functions are variants of most-significant-byte radix sorting; in particular, see D.E. Knuth's *Algorithm R* and section 5.2.5, exercise 10. They take linear time relative to the number of bytes in the

strings.

RETURN VALUES

The **radixsort**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL] The value of the *endbyte* element of *table* is not 0 or 255.

Additionally, the **sradixsort**() function may fail and set *errno* for any of the errors specified for the library routine malloc(3).

SEE ALSO

sort(1), qsort(3)

Knuth, D.E., "Sorting and Searching", *The Art of Computer Programming*, Vol. 3, pp. 170-178, 1968.

Paige, R., "Three Partition Refinement Algorithms", *SIAM J. Comput.*, No. 6, Vol. 16, 1987.

McIlroy, P., "Computing Systems", *Engineering Radix Sort*, Vol. 6:1, pp. 5-27, 1993.

HISTORY

The **radixsort**() function first appeared in 4.4BSD.

NAME

raise - send a signal to the current thread

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
raise(*int sig*);

DESCRIPTION

The **raise**() function sends the signal *sig* to the current thread.

RETURN VALUES

The **raise**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **raise**() function may fail and set *errno* for any of the errors specified for the library functions getpid(2) and kill(2).

SEE ALSO

kill(2)

STANDARDS

The **raise**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

NAME

rand, **srand**, **sranddev**, **rand_r** - bad random number generator

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

void
srand(*unsigned seed*);

void
sranddev(*void*);

int
rand(*void*);

int
rand_r(*unsigned *ctx*);

DESCRIPTION

The functions described in this manual page are not cryptographically secure. Cryptographic applications should use arc4random(3) instead.

These interfaces are obsoleted by random(3).

The **rand**() function computes a sequence of pseudo-random integers in the range of 0 to RAND_MAX (as defined by the header file <*stdlib.h*>).

The **srand**() function sets its argument *seed* as the seed for a new sequence of pseudo-random numbers to be returned by **rand**(). These sequences are repeatable by calling **srand**() with the same seed value.

If no *seed* value is provided, the functions are automatically seeded with a value of 1.

The **sranddev**() function initializes a seed using pseudo-random numbers obtained from the kernel.

The **rand_r**() function provides the same functionality as **rand**(). A pointer to the context value *ctx* must be supplied by the caller.

For better generator quality, use random(3) or lrand48(3).

SEE ALSO

arc4random(3), lrand48(3), random(3), random(4)

STANDARDS

The **rand**() and **srand**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

The **rand_r**() function is as proposed in the POSIX.4a Draft #6 document.

NAME

rctl_add_rule, **rctl_get_limits**, **rctl_get_racct**, **rctl_get_rules**, **rctl_remove_rule** - manipulate and query the resource limits database

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/rctl.h>

int
rctl_add_rule(*const char *inbufp*, *size_t inbuflen*, *char *outbufp*, *size_t outbuflen*);

int
rctl_get_limits(*const char *inbufp*, *size_t inbuflen*, *char *outbufp*, *size_t outbuflen*);

int
rctl_get_racct(*const char *inbufp*, *size_t inbuflen*, *char *outbufp*, *size_t outbuflen*);

int
rctl_get_rules(*const char *inbufp*, *size_t inbuflen*, *char *outbufp*, *size_t outbuflen*);

int
rctl_remove_rule(*const char *inbufp*, *size_t inbuflen*, *char *outbufp*, *size_t outbuflen*);

DESCRIPTION

These system calls are used to manipulate and query the resource limits database. For all functions, *inbuflen* refers to the length of the buffer pointed to by *inbufp* and *outbuflen* refers to the length of the buffer pointed to by *outbufp*.

The **rctl_add_rule**() function adds the rule pointed to by *inbufp* to the resource limits database. The *outbufp* and *outbuflen* arguments are unused. Rule format is as described in rctl(8), with exceptions noted in the *RULES AND FILTERS* section.

The **rctl_get_limits**() function returns in *outbufp* a comma-separated list of rules that apply to the process that matches the filter specified in *inbufp*. This includes rules with a subject of the process itself as well as rules with a different subject (such as user or loginclass) that apply to the process.

The **rctl_get_racct**() function returns resource usage information for a given subject. The subject is specified by passing a filter in *inbufp*. Filter syntax is as described in rctl(8), with exceptions noted in

the *RULES AND FILTERS* section. A comma-separated list of resources and the amount used of each by the specified subject is returned in *outbufp*. The resource and amount is formatted as "resource=amount".

The **rctl_get_rules**() function returns in *outbufp* a comma-separated list of rules from the resource limits database that match the filter passed in *inbufp*. Filter syntax is as described in rctl(8), with exceptions noted in the *RULES AND FILTERS* section. A filter of *::* may be passed to return all rules.

The **rctl_remove_rule**() function removes all rules matching the filter passed in *inbufp* from the resource limits database. Filter syntax is as described in rctl(8), with exceptions noted in the *RULES AND FILTERS* section. *outbufp* and *outbuflen* are unused.

RULES AND FILTERS

This section explains how the rule and filter format described in rctl(8) differs from the format passed to the system calls themselves. The rctl tool provides several conveniences that the system calls do not. When using the system call:

- The subject must be fully specified. For example, abbreviating ‘user’ to ‘u’ is not acceptable.
- User and group IDs must be numeric. For example, ‘root’ must be expressed as ‘0’.
- Units are not permitted on resource amounts. For example, a quantity of 1024 bytes must be expressed as ‘1024’ and not ‘1k’.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The rctl system calls may fail if:

[ENOSYS]	RACCT/RCTL support is not present in the kernel or the *kern.racct.enable* sysctl is 0.
[EINVAL]	The rule or filter passed in *inbufp* is invalid.
[EPERM]	User has insufficient privileges to carry out the requested operation.
[E2BIG]	*inbufp* or *outbufp* are too large.

[ESRCH] No process matched the provided rule or filter.

[ENAMETOOLONG]
The loginclass or jail name specified is too long.

[ERANGE] The rule amount is outside of the allowable range or *outbufp* is too small.

[EOPNOTSUPP] The requested operation is not supported for the given rule or filter.

[EFAULT] *inbufp* or *outbufp* refer to invalid addresses.

SEE ALSO

rctl(8)

HISTORY

The rctl family of system calls appeared in FreeBSD 9.0.

AUTHORS

The rctl system calls were developed by Edward Tomasz Napierala <*trasz@FreeBSD.org*> under sponsorship from the FreeBSD Foundation. This manual page was written by Eric Badger <*badger@FreeBSD.org*>.

NAME

re_comp, **re_exec** - regular expression handler

LIBRARY

Compatibility Library (libcompat, -lcompat)

SYNOPSIS

#include <unistd.h>

*char **
re_comp(*const char *s*);

int
re_exec(*const char *s*);

DESCRIPTION

This interface is made obsolete by regex(3).

The **re_comp**() function compiles a string into an internal form suitable for pattern matching. The **re_exec**() function checks the argument string against the last string passed to **re_comp**().

The **re_comp**() function returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If **re_comp**() is passed 0 or a null string, it returns without changing the currently compiled regular expression.

The **re_exec**() function returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and -1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both **re_comp**() and **re_exec**() may have trailing or embedded newline characters; they are terminated by NULs. The regular expressions recognized are described in the manual entry for ed(1), given the above difference.

DIAGNOSTICS

The **re_exec**() function returns -1 for an internal error.

The **re_comp**() function returns "no previous regular expression" or one of the strings generated by regerror(3).

SEE ALSO

ed(1), egrep(1), ex(1), fgrep(1), grep(1), regex(3)

HISTORY

The **re_comp**() and **re_exec**() functions appeared in 4.0BSD.

NAME

readlink, **readlinkat** - read value of a symbolic link

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

ssize_t
readlink(*const char *restrict path*, *char *restrict buf*, *size_t bufsiz*);

ssize_t
readlinkat(*int fd*, *const char *restrict path*, *char *restrict buf*, *size_t bufsize*);

DESCRIPTION

The **readlink**() system call places the contents of the symbolic link *path* in the buffer *buf*, which has size *bufsiz*. The **readlink**() system call does not append a NUL character to *buf*.

The **readlinkat**() system call is equivalent to **readlink**() except in the case where *path* specifies a relative path. In this case the symbolic link whose content is read relative to the directory associated with the file descriptor *fd* instead of the current working directory. If **readlinkat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **readlink**().

RETURN VALUES

The call returns the count of characters placed in the buffer if it succeeds, or a -1 if an error occurs, placing the error code in the global variable *errno*.

ERRORS

The **readlink**() system call will fail if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EINVAL] The named file is not a symbolic link.

[EIO] An I/O error occurred while reading from the file system.

[EFAULT] The *buf* argument extends outside the process's allocated address space.

In addition to the errors returned by the **readlink**(), the **readlinkat**() may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

lstat(2), stat(2), symlink(2), symlink(7)

STANDARDS

The **readlinkat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **readlink**() system call appeared in 4.2BSD. The **readlinkat**() system call appeared in FreeBSD 8.0.

NAME

reallocarray - memory reallocation function

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*void **

reallocarray(*void *ptr*, *size_t nmemb*, *size_t size*);

DESCRIPTION

The **reallocarray**() function is similar to the **realloc**() function except it operates on *nmemb* members of size *size* and checks for integer overflow in the calculation *nmemb * size*.

RETURN VALUES

The **reallocarray**() function returns a pointer to the allocated space; otherwise, a NULL pointer is returned and *errno* is set to ENOMEM.

EXAMPLES

Consider **reallocarray**() when there is multiplication in the *size* argument of **malloc**() or **realloc**(). For example, avoid this common idiom as it may lead to integer overflow:

```
if ((p = malloc(num * size)) == NULL)
        err(1, "malloc");
```

A drop-in replacement is the OpenBSD extension **reallocarray**():

```
if ((p = reallocarray(NULL, num, size)) == NULL)
        err(1, "reallocarray");
```

When using **realloc**(), be careful to avoid the following idiom:

```
size += 50;
if ((p = realloc(p, size)) == NULL)
        return (NULL);
```

Do not adjust the variable describing how much memory has been allocated until the allocation has been successful. This can cause aberrant program behavior if the incorrect size value is used. In most cases,

the above sample will also result in a leak of memory. As stated earlier, a return value of NULL indicates that the old object still remains allocated. Better code looks like this:

```
newsize = size + 50;
if ((newp = realloc(p, newsize)) == NULL) {
        free(p);
        p = NULL;
        size = 0;
        return (NULL);
}
p = newp;
size = newsize;
```

As with **malloc**(), it is important to ensure the new size value will not overflow; i.e. avoid allocations like the following:

```
if ((newp = realloc(p, num * size)) == NULL) {
        ...
```

Instead, use **reallocarray**():

```
if ((newp = reallocarray(p, num, size)) == NULL) {
        ...
```

SEE ALSO

realloc(3)

HISTORY

The **reallocarray**() function first appeared in OpenBSD 5.6 and FreeBSD 11.0.

NAME

reallocf - memory reallocation function

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*void **
reallocf(*void *ptr*, *size_t size*);

DESCRIPTION

The **reallocf**() function is identical to the **realloc**() function, except that it will free the passed pointer when the requested memory cannot be allocated. This is a FreeBSD specific API designed to ease the problems with traditional coding styles for **realloc**() causing memory leaks in libraries.

RETURN VALUES

The **reallocf**() function returns a pointer, possibly identical to *ptr*, to the allocated memory if successful; otherwise a NULL pointer is returned, and *errno* is set to ENOMEM if the error was the result of an allocation failure. The **reallocf**() function deletes the original buffer when an error occurs.

SEE ALSO

realloc(3)

HISTORY

The **reallocf**() function first appeared in FreeBSD 3.0.

NAME

realpath - returns the canonicalized absolute pathname

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

*char **

realpath(*const char * restrict pathname*, *char * restrict resolved_path*);

DESCRIPTION

The **realpath**() function resolves all symbolic links, extra "/" characters and references to */./* and */../* in *pathname*, and copies the resulting absolute pathname into the memory pointed to by *resolved_path*. The *resolved_path* argument *must* point to a buffer capable of storing at least PATH_MAX characters, or be NULL.

The **realpath**() function will resolve both absolute and relative paths and return the absolute pathname corresponding to *pathname*. All components of *pathname* must exist when **realpath**() is called, and all but the last component must name either directories or symlinks pointing to the directories.

RETURN VALUES

The **realpath**() function returns *resolved_path* on success. If the function was supplied NULL as *resolved_path*, and operation did not cause errors, the returned value is a null-terminated string in a buffer allocated by a call to **malloc**(*3*). If an error occurs, **realpath**() returns NULL, and if *resolved_path* is not NULL, the array that it points to contains the pathname which caused the problem.

ERRORS

The function **realpath**() may fail and set the external variable *errno* for any of the errors specified for the library functions lstat(2), readlink(2) and getcwd(3).

SEE ALSO

getcwd(3)

HISTORY

The **realpath**() function first appeared in 4.4BSD.

CAVEATS

This implementation of **realpath**() differs slightly from the Solaris implementation. The 4.4BSD version

always returns absolute pathnames, whereas the Solaris implementation will, under certain circumstances, return a relative *resolved_path* when given a relative *pathname*.

NAME

reboot - reboot system or halt processor

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>
#include <sys/reboot.h>

int
reboot(*int howto*);

DESCRIPTION

The **reboot**() system call reboots the system. Only the super-user may reboot a machine on demand. However, a reboot is invoked automatically in the event of unrecoverable system failures.

The *howto* argument is a mask of options; the system call interface allows the following options, defined in the include file <*sys/reboot.h*>, to be passed to the new kernel or the new bootstrap and init programs.

RB_AUTOBOOT The default, causing the system to reboot in its usual fashion.

RB_ASKNAME Normally the system only prompts the user if the loader specified root file system has an error. This flag forces it to always prompt the user for the root partition.

RB_DFLTROOT Use the compiled in root device. Normally, the system uses the device from which it was booted as the root device if possible. (The default behavior is dependent on the ability of the bootstrap program to determine the drive from which it was loaded, which is not possible on all systems.)

RB_DUMP Dump kernel memory before rebooting; see savecore(8) for more information.

RB_HALT The processor is simply halted; no reboot takes place. This option should be used with caution.

RB_POWERCYCLE After halting, the shutdown code will do what it can to turn off the power and then turn the power back on. This requires hardware support, usually an auxiliary microprocessor that can sequence the power supply. At present only the ipmi(4) driver implements this feature.

RB_POWEROFF After halting, the shutdown code will do what it can to turn off the power. This requires hardware support.

RB_KDB Load the symbol table and enable a built-in debugger in the system. This option will have no useful function if the kernel is not configured for debugging. Several other options have different meaning if combined with this option, although their use may not be possible via the **reboot**() system call. See ddb(4) for more information.

RB_NOSYNC Normally, the disks are sync'd (see sync(8)) before the processor is halted or rebooted. This option may be useful if file system changes have been made manually or if the processor is on fire.

RB_REROOT Instead of rebooting, unmount all filesystems except the one containing currently-running executable, and mount root filesystem using the same mechanism which is used during normal boot, based on vfs.root.mountfrom kenv(1) variable.

RB_RDONLY Initially mount the root file system read-only. This is currently the default, and this option has been deprecated.

RB_SINGLE Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. RB_SINGLE prevents this, booting the system with a single-user shell on the console. RB_SINGLE is actually interpreted by the init(8) program in the newly booted system.

When no options are given (i.e., RB_AUTOBOOT is used), the system is rebooted from file "kernel" in the root file system of unit 0 of a disk chosen in a processor specific way. An automatic consistency check of the disks is normally performed (see fsck(8)).

RETURN VALUES

If successful, this call never returns. Otherwise, a -1 is returned and an error is returned in the global variable *errno*.

ERRORS

[EPERM] The caller is not the super-user.

SEE ALSO

crash(8), halt(8), init(8), reboot(8), savecore(8)

HISTORY

The **reboot**() system call appeared in 4.0BSD.

NAME

recv, **recvfrom**, **recvmsg**, **recvmmsg** - receive message(s) from a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

ssize_t
recv(*int s*, *void *buf*, *size_t len*, *int flags*);

ssize_t
recvfrom(*int s*, *void *buf*, *size_t len*, *int flags*, *struct sockaddr * restrict from*,
*socklen_t * restrict fromlen*);

ssize_t
recvmsg(*int s*, *struct msghdr *msg*, *int flags*);

ssize_t
recvmmsg(*int s*, *struct mmsghdr * restrict msgvec*, *size_t vlen*, *int flags*,
*const struct timespec * restrict timeout*);

DESCRIPTION

The **recvfrom**(), **recvmsg**(), and **recvmmsg**() system calls are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

If *from* is not a null pointer and the socket is not connection-oriented, the source address of the message is filled in. The *fromlen* argument is a value-result argument, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there.

The **recv**() function is normally used only on a *connected* socket (see connect(2)) and is identical to **recvfrom**() with a null pointer passed as its *from* argument.

The **recvmmsg**() function is used to receive multiple messages at a call. Their number is supplied by *vlen*. The messages are placed in the buffers described by *msgvec* vector, after reception. The size of each received message is placed in the *msg_len* field of each element of the vector. If *timeout* is NULL the call blocks until the data is available for each supplied message buffer. Otherwise it waits for data for the specified amount of time. If the timeout expired and there is no data received, a value 0 is returned. The ppoll(2) system call is used to implement the timeout mechanism, before first receive is

performed.

The **recv**(), **recvfrom**() and **recvmsg**() return the length of the message on successful completion, whereas **recvmmsg**() returns the number of received messages. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see fcntl(2)) in which case the value -1 is returned and the global variable *errno* is set to EAGAIN. The receive calls except **recvmmsg**() normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in getsockopt(2). The **recvmmsg**() function implements this behaviour for each message in the vector.

The select(2) system call may be used to determine when more data arrives.

The *flags* argument to a **recv**() function is formed by *or*'ing one or more of the values:

MSG_OOB	process out-of-band data
MSG_PEEK	peek at incoming message
MSG_WAITALL	wait for full request or error
MSG_DONTWAIT	do not block
MSG_CMSG_CLOEXEC	set received fds close-on-exec
MSG_WAITFORONE	do not block after receiving the first message (only for **recvmmsg**())

The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned. The MSG_DONTWAIT flag requests the call to return when it would block otherwise. If no data is available, *errno* is set to EAGAIN. This flag is not available in ANSI X3.159-1989 ("ANSI C89") or ISO/IEC 9899:1999 ("ISO C99") compilation mode. The MSG_WAITFORONE flag sets MSG_DONTWAIT after the first message has been received. This flag is only relevant for **recvmmsg**().

The **recvmsg**() system call uses a *msghdr* structure to minimize the number of directly supplied

arguments. This structure has the following form, as defined in <*sys/socket.h*>:

```
struct msghdr {
        void            *msg_name;      /* optional address */
        socklen_t       msg_namelen;    /* size of address */
        struct iovec    *msg_iov;       /* scatter/gather array */
        int             msg_iovlen;     /* # elements in msg_iov */
        void            *msg_control;   /* ancillary data, see below */
        socklen_t       msg_controllen;/* ancillary data buffer len */
        int             msg_flags;      /* flags on received message */
};
```

Here *msg_name* and *msg_namelen* specify the source address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* arguments describe scatter gather locations, as discussed in read(2). The *msg_control* argument, which has length *msg_controllen*, points to a buffer for other protocol control related messages or other miscellaneous ancillary data. The messages are of the form:

```
struct cmsghdr {
        socklen_t       cmsg_len;       /* data byte count, including hdr */
        int             cmsg_level;     /* originating protocol */
        int             cmsg_type;      /* protocol-specific type */
/* followed by
        u_char          cmsg_data[]; */
};
```

As an example, one could use this to learn of changes in the data-stream in XNS/SPP, or in ISO, to obtain user-connection-request data by requesting a **recvmsg**() with no data buffer provided immediately after an **accept**() system call.

With AF_UNIX domain sockets, ancillary data can be used to pass file descriptors and process credentials. See unix(4) for details.

The *msg_flags* field is set on return according to the message received. MSG_EOR indicates end-of-record; the data returned completed a record (generally used with sockets of type SOCK_SEQPACKET). MSG_TRUNC indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. MSG_CTRUNC indicates that some control data were discarded due to lack of space in the buffer for ancillary data. MSG_OOB is returned to indicate that expedited or out-of-band data were received.

The **recvmmsg**() system call uses the *mmsghdr* structure, defined as follows in the <*sys/socket.h*> header:

```
struct mmsghdr {
        struct msghdr   msg_hdr;        /* message header */
        ssize_t         msg_len;/* message length */
};
```

On data reception the *msg_len* field is updated to the length of the received message.

RETURN VALUES

These calls except **recvmmsg**() return the number of bytes received. **recvmmsg**() returns the number of messages received. A value of -1 is returned if an error occurred.

ERRORS

The calls fail if:

[EBADF] The argument *s* is an invalid descriptor.

[ECONNRESET] The remote socket end is forcibly closed.

[ENOTCONN] The socket is associated with a connection-oriented protocol and has not been connected (see connect(2) and accept(2)).

[ENOTSOCK] The argument *s* does not refer to a socket.

[EMSGSIZE] The **recvmsg**() system call was used to receive rights (file descriptors) that were in flight on the connection. However, the receiving program did not have enough free file descriptor slots to accept them. In this case the descriptors are closed, any pending data can be returned by another call to **recvmsg**().

[EAGAIN] The socket is marked non-blocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data were received.

[EINTR] The receive was interrupted by delivery of a signal before any data were available.

[EFAULT] The receive buffer pointer(s) point outside the process's address space.

SEE ALSO

fcntl(2), getsockopt(2), read(2), select(2), socket(2), CMSG_DATA(3), unix(4)

HISTORY

The **recv**() function appeared in 4.2BSD. The **recvmmsg**() function appeared in FreeBSD 11.0.

NAME

regcomp, **regexec**, **regerror**, **regfree** - regular-expression library

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <regex.h>

int
regcomp(*regex_t * restrict preg*, *const char * restrict pattern*, *int cflags*);

int
regexec(*const regex_t * restrict preg*, *const char * restrict string*, *size_t nmatch*,
regmatch_t pmatch[restrict], *int eflags*);

size_t
regerror(*int errcode*, *const regex_t * restrict preg*, *char * restrict errbuf*, *size_t errbuf_size*);

void
regfree(*regex_t *preg*);

DESCRIPTION

These routines implement IEEE Std 1003.2 ("POSIX.2") regular expressions ("RE"s); see re_format(7). The **regcomp**() function compiles an RE written as a string into an internal form, **regexec**() matches that internal form against a string and reports results, **regerror**() transforms error codes from either into human-readable messages, and **regfree**() frees any dynamically-allocated storage used by the internal form of an RE.

The header <*regex.h*> declares two structure types, *regex_t* and *regmatch_t*, the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t*, and a number of constants with names starting with "REG_".

The **regcomp**() function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags*, and places the results in the *regex_t* structure pointed to by *preg*. The *cflags* argument is the bitwise OR of zero or more of the following flags:

REG_EXTENDED Compile modern ("extended") REs, rather than the obsolete ("basic") REs that are the default.

REG_BASIC This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.

REG_NOSPEC Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the "RE" is a literal string. This is an extension, compatible with but not specified by IEEE Std 1003.2 ("POSIX.2"), and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to **regcomp**().

REG_ICASE Compile for matching that ignores upper/lower case distinctions. See re_format(7).

REG_NOSUB Compile for matching that need only report success or failure, not what was matched.

REG_NEWLINE Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.

REG_PEND The regular expression ends, not at the first NUL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg*. The *re_endp* member is of type *const char **. This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with but not specified by IEEE Std 1003.2 ("POSIX.2"), and should be used with caution in software intended to be portable to other systems.

When successful, **regcomp**() returns 0 and fills in the structure pointed to by *preg*. One member of that structure (other than *re_endp*) is publicized: *re_nsub*, of type *size_t*, contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If **regcomp**() fails, it returns a non-zero error code; see *DIAGNOSTICS*.

The **regexec**() function matches the compiled RE pointed to by *preg* against the *string*, subject to the flags in *eflags*, and reports results using *nmatch*, *pmatch*, and the returned value. The RE must have been compiled by a previous invocation of **regcomp**(). The compiled form is not altered during execution of **regexec**(), so a single compiled RE can be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line,

minus any terminating newline. The *eflags* argument is the bitwise OR of zero or more of the following flags:

REG_NOTBOL The first character of the string is treated as the continuation of a line. This means that the anchors '^', '[[:<:]]', and '\<' do not match before it; but see REG_STARTEND below. This does not affect the behavior of newlines under REG_NEWLINE.

REG_NOTEOL The NUL terminating the string does not end a line, so the '$' anchor does not match before it. This does not affect the behavior of newlines under REG_NEWLINE.

REG_STARTEND The string is considered to start at *string* + *pmatch*[0].*rm_so* and to end before the byte located at *string* + *pmatch*[0].*rm_eo*, regardless of the value of *nmatch*. See below for the definition of *pmatch* and *nmatch*. This is an extension, compatible with but not specified by IEEE Std 1003.2 ("POSIX.2"), and should be used with caution in software intended to be portable to other systems.

Without REG_NOTBOL, the position *rm_so* is considered the beginning of a line, such that '^' matches before it, and the beginning of a word if there is a word character at this position, such that '[[:<:]]' and '\<' match before it.

With REG_NOTBOL, the character at position *rm_so* is treated as the continuation of a line, and if *rm_so* is greater than 0, the preceding character is taken into consideration. If the preceding character is a newline and the regular expression was compiled with REG_NEWLINE, '^' matches before the string; if the preceding character is not a word character but the string starts with a word character, '[[:<:]]' and '\<' match before the string.

See re_format(7) for a discussion of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string*.

Normally, **regexec**() returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see *DIAGNOSTICS*.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, **regexec**() ignores the *pmatch* argument (but see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of type *regmatch_t*. Such a structure has at least the members *rm_so* and *rm_eo*, both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first

character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to **regexec**(). An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i*, with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array (corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg->re_nsub*)) have both *rm_so* and *rm_eo* set to -1. If a subexpression participated in the match several times, the reported substring is the last one it matched. (Note, as an example in particular, that when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', so the reported substring is one of the empties.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch*; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch*[0] will not be changed by a successful **regexec**().

The **regerror**() function maps a non-zero *errcode* from either **regcomp**() or **regexec**() to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg*, and if the error code came from **regcomp**(), it should have been the result from the most recent **regcomp**() using that *regex_t*. The (**regerror**() may be able to supply a more detailed message using information from the *regex_t*.) The **regerror**() function places the NUL-terminated message into the buffer pointed to by *errbuf*, limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message will not fit, as much of it as will fit before the terminating NUL is supplied. In any case, the returned value is the size of buffer needed to hold the whole message (including terminating NUL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to **regerror**() is first ORed with REG_ITOA, the "message" that results is the printable name of the error code, e.g. "REG_NOMATCH", rather than an explanation thereof. If *errcode* is REG_ATOI, then *preg* shall be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). REG_ITOA and REG_ATOI are intended primarily as debugging facilities; they are extensions, compatible with but not specified by IEEE Std 1003.2 ("POSIX.2"), and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

The **regfree**() function frees any dynamically-allocated storage associated with the compiled RE pointed

to by *preg*. The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to **regexec**() or **regerror**() is undefined.

None of these functions references global variables except for tables of constants; all are safe for use from multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that IEEE Std 1003.2 ("POSIX.2") leaves up to the implementor, either by explicitly saying "undefined" or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See re_format(7) for a discussion of the definition of case-independent matching.

There is no particular limit on the length of REs, except insofar as memory is limited. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See *BUGS* for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by IEEE Std 1003.2 ("POSIX.2") (such magic meanings occur only in obsolete ["basic"] REs) is taken as an ordinary character.

Any unmatched ‘[’ is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

RE_DUP_MAX, the limit on repetition counts in bounded repetitions, is 255.

A repetition operator (‘?’, ‘*’, ‘+’, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

‘|’ cannot appear first or last in a (sub)expression or after another ‘|’, i.e., an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A ‘{’ followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ *not* followed by a digit is considered an ordinary character.

‘^’ and ‘$’ beginning and ending subexpressions in obsolete ("basic") REs are anchors, not ordinary characters.

DIAGNOSTICS

Non-zero error codes from **regcomp**() and **regexec**() include the following:

REG_NOMATCH	The **regexec**() function failed to match
REG_BADPAT	invalid regular expression
REG_ECOLLATE	invalid collating element
REG_ECTYPE	invalid character class
REG_EESCAPE	‘\’ applied to unescapable character
REG_ESUBREG	invalid backreference number
REG_EBRACK	brackets ‘[ ]’ not balanced
REG_EPAREN	parentheses ‘( )’ not balanced
REG_EBRACE	braces ‘{ }’ not balanced
REG_BADBR	invalid repetition count(s) in ‘{ }’
REG_ERANGE	invalid character range in ‘[ ]’
REG_ESPACE	ran out of memory
REG_BADRPT	‘?’, ‘*’, or ‘+’ operand invalid
REG_EMPTY	empty (sub)expression
REG_ASSERT	cannot happen - you found a bug
REG_INVARG	invalid argument, e.g. negative-length string
REG_ILLSEQ	illegal byte sequence (bad multibyte character)

SEE ALSO

grep(1), re_format(7)

IEEE Std 1003.2 ("POSIX.2"), sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

The back-reference code is subtle and doubts linger about its correctness in complex cases.

The **regexec**() function performance is poor. This will improve with later releases. The *nmatch* argument exceeding 0 is expensive; *nmatch* exceeding 1 is worse. The **regexec**() function is largely insensitive to RE complexity *except* that back references are massively expensive. RE length does matter; in particular, there is a strong speed bonus for keeping RE length under about 30 characters, with most special characters counting roughly double.

The **regcomp**() function implements bounded repetitions by macro expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like, say, ‘((((a{1,100}){1,100}){1,100}){1,100}){1,100}’ will (eventually) run almost any existing machine out of swap space.

There are suspected problems with response to obscure error conditions. Notably, certain kinds of internal overflow, produced only by truly enormous REs or by multiply nested bounded repetitions, are probably not handled well.

Due to a mistake in IEEE Std 1003.2 ("POSIX.2"), things like ‘a)b’ are legal REs because ‘)’ is a special character only in the presence of a previous unmatched ‘(’. This cannot be fixed until the spec is fixed.

The standard’s definition of back references is vague. For example, does ‘a\(\(b\)*\2\)*d’ match ‘abbbd’? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is a bit of a kludge, and bugs may lurk in combinations of word-boundary matching and anchoring.

Word-boundary matching does not work properly in multibyte locales.

NAME

remove - remove directory entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
remove(*const char *path*);

DESCRIPTION

The **remove**() function removes the file or directory specified by *path*.

If *path* specifies a directory, **remove**(*path*) is the equivalent of **rmdir**(*path*). Otherwise, it is the equivalent of **unlink**(*path*).

RETURN VALUES

The **remove**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **remove**() function may fail and set *errno* for any of the errors specified for the routines lstat(2), rmdir(2) or unlink(2).

SEE ALSO

rmdir(2), unlink(2)

STANDARDS

The **remove**() function conforms to ISO/IEC 9899:1990 ("ISO C90") and X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

NAME

rename - change the name of a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
rename(*const char *from*, *const char *to*);

int
renameat(*int fromfd*, *const char *from*, *int tofd*, *const char *to*);

DESCRIPTION

The **rename**() system call causes the link named *from* to be renamed as *to*. If *to* exists, it is first removed. Both *from* and *to* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system.

The **rename**() system call guarantees that if *to* already exists, an instance of *to* will always exist, even if the system should crash in the middle of the operation.

If the final component of *from* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

If *from* and *to* resolve to the same directory entry, or to different directory entries for the same existing file, **rename**() returns success without taking any further action.

The **renameat**() system call is equivalent to **rename**() except in the case where either *from* or *to* specifies a relative path. If *from* is a relative path, the file to be renamed is located relative to the directory associated with the file descriptor *fromfd* instead of the current working directory. If the *to* is a relative path, the same happens only relative to the directory associated with *tofd*. If the **renameat**() is passed the special value AT_FDCWD in the *fromfd* or *tofd* parameter, the current working directory is used in the determination of the file for the respective path parameter.

RETURN VALUES

The **rename**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **rename**() system call will fail and neither of the argument files will be affected if:

[ENAMETOOLONG] A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

[ENOENT] A component of the *from* path does not exist, or a path prefix of *to* does not exist.

[EACCES] A component of either path prefix denies search permission.

[EACCES] The requested link requires writing in a directory with a mode that denies write permission.

[EACCES] The directory pointed at by the *from* argument denies write permission, and the operation would move it to another parent directory.

[EPERM] The file pointed at by the *from* argument has its immutable, undeletable or append-only flag set, see the chflags(2) manual page for more information.

[EPERM] The parent directory of the file pointed at by the *from* argument has its immutable or append-only flag set.

[EPERM] The parent directory of the file pointed at by the *to* argument has its immutable flag set.

[EPERM] The directory containing *from* is marked sticky, and neither the containing directory nor *from* are owned by the effective user ID.

[EPERM] The file pointed at by the *to* argument exists, the directory containing *to* is marked sticky, and neither the containing directory nor *to* are owned by the effective user ID.

[ELOOP] Too many symbolic links were encountered in translating either pathname.

[ENOTDIR] A component of either path prefix is not a directory.

[ENOTDIR] The *from* argument is a directory, but *to* is not a directory.

[EISDIR] The *to* argument is a directory, but *from* is not a directory.

[EXDEV] The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

[ENOSPC] The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.

[EDQUOT] The directory in which the entry for the new name is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EIO] An I/O error occurred while making or updating a directory entry.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EFAULT] Path points outside the process's allocated address space.

[EINVAL] The *from* argument is a parent directory of *to*, or an attempt is made to rename '.' or '..'.

[ENOTEMPTY] The *to* argument is a directory and is not empty.

[ECAPMODE] **rename**() was called and the process is in capability mode.

In addition to the errors returned by the **rename**(), the **renameat**() may fail if:

[EBADF] The *from* argument does not specify an absolute path and the *fromfd* argument is neither AT_FDCWD nor a valid file descriptor open for searching, or the *to* argument does not specify an absolute path and the *tofd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *from* argument is not an absolute path and *fromfd* is neither AT_FDCWD nor a file descriptor associated with a directory, or the *to* argument is not an absolute path and *tofd* is neither AT_FDCWD nor a file descriptor associated with a directory.

[ECAPMODE] AT_FDCWD is specified and the process is in capability mode.

[ENOTCAPABLE] *path* is an absolute path or contained a ".." component leading to a directory outside of the directory hierarchy specified by *fromfd* or *tofd*.

[ENOTCAPABLE] The *fromfd* file descriptor lacks the CAP_RENAMEAT_SOURCE right, or the *tofd* file descriptor lacks the CAP_RENAMEAT_TARGET right.

SEE ALSO

chflags(2), open(2), symlink(7)

STANDARDS

The **rename**() system call is expected to conform to ISO/IEC 9945-1:1996 ("POSIX.1"). The **renameat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **renameat**() system call appeared in FreeBSD 8.0.

NAME

revoke - revoke file access

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
revoke(*const char *path*);

DESCRIPTION

The **revoke**() system call invalidates all current open file descriptors in the system for the file named by *path*. Subsequent operations on any such descriptors fail, with the exceptions that a **read**() from a character device file which has been revoked returns a count of zero (end of file), and a **close**() system call will succeed. If the file is a special file for a device which is open, the device close function is called as if all open references to the file had been closed using a special close method which does not block.

Access to a file may be revoked only by its owner or the super user. The **revoke**() system call is currently supported only for block and character special device files. It is normally used to prepare a terminal device for a new login session, preventing any access by a previous user of the terminal.

RETURN VALUES

The **revoke**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Access to the named file is revoked unless one of the following:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1024 characters.

[ENOENT] The named file or a component of the path name does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EFAULT] The *path* argument points outside the process's allocated address space.

[EINVAL] The implementation does not support the **revoke**() operation on the named file.

[EPERM] The caller is neither the owner of the file nor the super user.

SEE ALSO

revoke(1), close(2)

HISTORY

The **revoke**() system call first appeared in 4.3BSD-Reno.

BUGS

The non-blocking close method is only correctly implemented for terminal devices.

NAME

rexec - return stream to a remote command

LIBRARY

Compatibility Library (libcompat, -lcompat)

SYNOPSIS

int
rexec(*char **ahost*, *int inport*, *char *user*, *char *passwd*, *char *cmd*, *int *fd2p*);

DESCRIPTION

This interface is obsoleted by rcmd(3).

The **rexec**() function looks up the host **ahost* using gethostbyname(3), returning -1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; the call **getservbyname**(*"exec"*, *"tcp"*) (see getservent(3)) will return a pointer to a structure, which contains the necessary port. The protocol for connection is described in detail in rexecd(8).

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as stdin and stdout. If *fd2p* is non-zero, then an auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. The diagnostic information returned does not include remote authorization failure, as the secondary connection is set up after authorization has been verified. If *fd2p* is 0, then the stderr (unit 2 of the remote command) will be made the same as the stdout and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3), rexecd(8)

HISTORY

The **rexec**() function appeared in 4.2BSD.

BUGS

The **rexec**() function sends the unencrypted password across the network.

The underlying service is considered a big security hole and therefore not enabled on many sites, see rexecd(8) for explanations.

NAME

rfork_thread - create a rfork-based process thread

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
rfork_thread(*int flags*, *void *stack*, *int (*func)(void *arg)*, *void *arg*);

DESCRIPTION

The rfork_thread() function has been deprecated in favor of pthread_create(3).

The **rfork_thread**() function is a helper function for rfork(2). It arranges for a new process to be created and the child process will call the specified function with the specified argument, while running on the supplied stack.

Using this function should avoid the need to implement complex stack swap code.

RETURN VALUES

Upon successful completion, **rfork_thread**() returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

The child process context is not aware of a return from the **rfork_thread**() function as it begins executing directly with the supplied function.

ERRORS

See rfork(2) for error return codes.

SEE ALSO

fork(2), intro(2), minherit(2), rfork(2), vfork(2), pthread_create(3)

HISTORY

The **rfork_thread**() function first appeared in FreeBSD 4.3.

NAME

rfork - manipulate process resources

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
rfork(*int flags*);

DESCRIPTION

Forking, vforking or rforking are the only ways new processes are created. The *flags* argument to **rfork**() selects which resources of the invoking process (parent) are shared by the new process (child) or initialized to their default values. The resources include the open file descriptor table (which, when shared, permits processes to open and close files for other processes), and open files. The *flags* argument is the logical OR of some subset of:

RFPROC — If set a new process is created; otherwise changes affect the current process.

RFNOWAIT — If set, the child process will be dissociated from the parent. Upon exit the child will not leave a status for the parent to collect. See wait(2).

RFFDG — If set, the invoker's file descriptor table (see intro(2)) is copied; otherwise the two processes share a single table.

RFCFDG — If set, the new process starts with a clean file descriptor table. Is mutually exclusive with RFFDG.

RFTHREAD — If set, the new process shares file descriptor to process leaders table with its parent. Only applies when neither RFFDG nor RFCFDG are set.

RFMEM — If set, the kernel will force sharing of the entire address space, typically by sharing the hardware page table directly. The child will thus inherit and share all the segments the parent process owns, whether they are normally shareable or not. The stack segment is not split (both the parent and child return on the same stack) and thus **rfork**() with the RFMEM flag may not generally be called directly from high level languages including C. May be set only with RFPROC. A helper function is provided to assist with this problem and will cause the new process to

run on the provided stack. See rfork_thread(3) for information. Note that a lot of code will not run correctly in such an environment.

RFSIGSHARE If set, the kernel will force sharing the sigacts structure between the child and the parent.

RFTSIGZMB If set, the kernel will deliver a specified signal to the parent upon the child exit, instead of default SIGCHLD. The signal number signum is specified by oring the RFTSIGFLAGS(signum) expression into *flags*. Specifying signal number 0 disables signal delivery upon the child exit.

RFLINUXTHPN If set, the kernel will deliver SIGUSR1 instead of SIGCHLD upon thread exit for the child. This is intended to mimic certain Linux clone behaviour.

File descriptors in a shared file descriptor table are kept open until either they are explicitly closed or all processes sharing the table exit.

If RFPROC is set, the value returned in the parent process is the process id of the child process; the value returned in the child is zero. Without RFPROC, the return value is zero. Process id's range from 1 to the maximum integer (*int*) value. The **rfork**() system call will sleep, if necessary, until required process resources are available.

The **fork**() system call can be implemented as a call to **rfork**(*RFFDG* | *RFPROC*) but is not for backwards compatibility.

RETURN VALUES

Upon successful completion, **rfork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

ERRORS

The **rfork**() system call will fail and no child process will be created if:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. The limit is given by the sysctl(3) MIB variable KERN_MAXPROC. (The limit is actually ten less than this except for the super user).

[EAGAIN] The user is not the super user, and the system-imposed limit on the total number of processes under execution by a single user would be exceeded. The limit is

given by the sysctl(3) MIB variable KERN_MAXPROCPERUID.

[EAGAIN] The user is not the super user, and the soft resource limit corresponding to the *resource* argument RLIMIT_NOFILE would be exceeded (see getrlimit(2)).

[EINVAL] Both the RFFDG and the RFCFDG flags were specified.

[EINVAL] Any flags not listed above were specified.

[EINVAL] An invalid signal number was specified.

[ENOMEM] There is insufficient swap space for the new process.

SEE ALSO

fork(2), intro(2), minherit(2), vfork(2), pthread_create(3), rfork_thread(3)

HISTORY

The **rfork**() function first appeared in Plan9.

NAME

stpcpy, **strcat**, **strncat**, **strchr**, **strrchr**, **strcmp**, **strncmp**, **strcasecmp**, **strncasecmp**, **strcpy**, **strncpy**, **strerror**, **strlen**, **strpbrk**, **strsep**, **strspn**, **strcspn**, **strstr**, **strtok**, **index**, **rindex** - string specific functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

*char **
stpcpy(*char *dst*, *const char *src*);

*char **
strcat(*char *s*, *const char * append*);

*char **
strncat(*char *s*, *const char *append*, *size_t count*);

*char **
strchr(*const char *s*, *int c*);

*char **
strrchr(*const char *s*, *int c*);

int
strcmp(*const char *s1*, *const char *s2*);

int
strncmp(*const char *s1*, *const char *s2*, *size_t count*);

int
strcasecmp(*const char *s1*, *const char *s2*);

int
strncasecmp(*const char *s1*, *const char *s2*, *size_t count*);

*char **
strcpy(*char *dst*, *const char *src*);

*char **
strncpy(*char *dst*, *const char *src*, *size_t count*);

*char **
strerror(*int errno*);

size_t
strlen(*const char *s*);

*char **
strpbrk(*const char *s*, *const char *charset*);

*char **
strsep(*char **stringp*, *const char *delim*);

size_t
strspn(*const char *s*, *const char *charset*);

size_t
strcspn(*const char *s*, *const char *charset*);

*char **
strstr(*const char *big*, *const char *little*);

*char **
strtok(*char *s*, *const char *delim*);

*char **
index(*const char *s*, *int c*);

*char **
rindex(*const char *s*, *int c*);

DESCRIPTION

The string functions manipulate strings terminated by a null byte.

See the specific manual pages for more information. For manipulating variable length generic objects as byte strings (without the null byte check), see bstring(3).

Except as noted in their specific manual pages, the string functions do not test the destination for size

limitations.

SEE ALSO

bstring(3), index(3), rindex(3), stpcpy(3), strcasecmp(3), strcat(3), strchr(3), strcmp(3), strcpy(3), strcspn(3), strerror(3), strlen(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strstr(3), strtok(3)

STANDARDS

The **strcat**(), **strncat**(), **strchr**(), **strrchr**(), **strcmp**(), **strncmp**(), **strcpy**(), **strncpy**(), **strerror**(), **strlen**(), **strpbrk**(), **strspn**(), **strcspn**(), **strstr**(), and **strtok**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

rmdir - remove a directory file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
rmdir(*const char *path*);

DESCRIPTION

The **rmdir**() system call removes a directory file whose name is given by *path*. The directory must not have any entries other than '.' and '..'.

RETURN VALUES

The **rmdir**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The named file is removed unless:

[ENOTDIR] A component of the path is not a directory.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named directory does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[ENOTEMPTY] The named directory contains files other than '.' and '..' in it.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EPERM] The directory to be removed has its immutable, undeletable or append-only flag

set, see the chflags(2) manual page for more information.

[EPERM] The parent directory of the directory to be removed has its immutable or append-only flag set.

[EPERM] The directory containing the directory to be removed is marked sticky, and neither the containing directory nor the directory to be removed are owned by the effective user ID.

[EINVAL] The last component of the path is '.' or '..'.

[EBUSY] The directory to be removed is the mount point for a mounted file system.

[EIO] An I/O error occurred while deleting the directory entry or deallocating the inode.

[EROFS] The directory entry to be removed resides on a read-only file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

SEE ALSO

mkdir(2), unlink(2)

HISTORY

The **rmdir**() system call appeared in 4.2BSD.

NAME

rpc_clnt_calls, **clnt_call**, **clnt_freeres**, **clnt_geterr**, **clnt_perrno**, **clnt_perror**, **clnt_sperrno**, **clnt_sperror**, **rpc_broadcast**, **rpc_broadcast_exp**, **rpc_call** - library routines for client side calls

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

enum clnt_stat
clnt_call(*CLIENT *clnt*, *const rpcproc_t procnum*, *const xdrproc_t inproc*, *const caddr_t in*, *const xdrproc_t outproc*, *caddr_t out*, *const struct timeval tout*);

bool_t
clnt_freeres(*CLIENT *clnt*, *const xdrproc_t outproc*, *caddr_t out*);

void
clnt_geterr(*const CLIENT * clnt*, *struct rpc_err * errp*);

void
clnt_perrno(*const enum clnt_stat stat*);

void
clnt_perror(*CLIENT *clnt*, *const char *s*);

*char **
clnt_sperrno(*const enum clnt_stat stat*);

*char **
clnt_sperror(*CLIENT *clnt*, *const char * s*);

enum clnt_stat
rpc_broadcast(*const rpcprog_t prognum*, *const rpcvers_t versnum*, *const rpcproc_t procnum*, *const xdrproc_t inproc*, *const caddr_t in*, *const xdrproc_t outproc*, *caddr_t out*, *const resultproc_t eachresult*, *const char *nettype*);

enum clnt_stat
rpc_broadcast_exp(*const rpcprog_t prognum*, *const rpcvers_t versnum*, *const rpcproc_t procnum*, *const xdrproc_t xargs*, *caddr_t argsp*, *const xdrproc_t xresults*, *caddr_t resultsp*,

const resultproc_t eachresult, *const int inittime*, *const int waittime*, *const char * nettype*);

enum clnt_stat
rpc_call(*const char *host*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *const rpcproc_t procnum*, *const xdrproc_t inproc*, *const char *in*, *const xdrproc_t outproc*, *char *out*, *const char *nettype*);

DESCRIPTION

RPC library routines allow C language programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a request to the server. Upon receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

The **clnt_call**(), **rpc_call**(), and **rpc_broadcast**() routines handle the client side of the procedure call. The remaining routines deal with error handling in the case of errors.

Some of the routines take a *CLIENT* handle as one of the arguments. A *CLIENT* handle can be created by an RPC creation routine such as **clnt_create**() (see rpc_clnt_create(3)).

These routines are safe for use in multithreaded applications. *CLIENT* handles can be shared between threads, however in this implementation requests by different threads are serialized (that is, the first request will receive its results before the second request is sent).

Routines

See rpc(3) for the definition of the *CLIENT* data structure.

clnt_call()

A function macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create**() (see rpc_clnt_create(3)). The *inproc* argument is the XDR function used to encode the procedure's arguments, and *outproc* is the XDR function used to decode the procedure's results; *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). The *tout* argument is the time allowed for results to be returned, which is overridden by a time-out set explicitly through **clnt_control**(), see rpc_clnt_create(3). If the remote call succeeds, the status returned is RPC_SUCCESS, otherwise an appropriate status is returned.

clnt_freeres()

A function macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The *out* argument is the address of the results, and *outproc* is the XDR routine describing the results. This routine returns 1 if the results were successfully freed, and 0 otherwise.

clnt_geterr()

A function macro that copies the error structure out of the client handle to the structure at address *errp*.

clnt_perrno()

Print a message to standard error corresponding to the condition indicated by *stat*. A newline is appended. Normally used after a procedure call fails for a routine for which a client handle is not needed, for instance **rpc_call**().

clnt_perror()

Print a message to the standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. A newline is appended. Normally used after a remote procedure call fails for a routine which requires a client handle, for instance **clnt_call**().

clnt_sperrno()

Take the same arguments as **clnt_perrno**(), but instead of sending a message to the standard error indicating why an RPC call failed, return a pointer to a string which contains the message. The **clnt_sperrno**() function is normally used instead of **clnt_perrno**() when the program does not have a standard error (as a program running as a server quite likely does not), or if the programmer does not want the message to be output with **printf**() (see printf(3)), or if a message format different than that supported by **clnt_perrno**() is to be used. Note: unlike **clnt_sperror**() and **clnt_spcreateerror**() (see rpc_clnt_create(3)), **clnt_sperrno**() does not return pointer to static data so the result will not get overwritten on each call.

clnt_sperror()

Like **clnt_perror**(), except that (like **clnt_sperrno**()) it returns a string instead of printing to standard error. However, **clnt_sperror**() does not append a newline at the end of the message. Warning: returns pointer to a buffer that is overwritten on each call.

rpc_broadcast()

Like **rpc_call**(), except the call message is broadcast to all the connectionless transports specified by *nettype*. If *nettype* is NULL, it defaults to "netpath". Each time it receives a response, this routine calls **eachresult**(), whose form is: *bool_t* **eachresult**(*caddr_t out*, *const struct netbuf * addr*, *const struct netconfig * netconf*) where *out* is the same as *out* passed to **rpc_broadcast**(), except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results, and *netconf* is the netconfig structure of the transport on which the remote server responded. If **eachresult**() returns 0, **rpc_broadcast**() waits for more replies; otherwise it returns with appropriate status. Warning: broadcast file descriptors are limited in size to the maximum transfer size of that transport. For Ethernet,

this value is 1500 bytes. The **rpc_broadcast**() function uses AUTH_SYS credentials by default (see rpc_clnt_auth(3)).

rpc_broadcast_exp()

Like **rpc_broadcast**(), except that the initial timeout, *inittime* and the maximum timeout, *waittime* are specified in milliseconds. The *inittime* argument is the initial time that **rpc_broadcast_exp**() waits before resending the request. After the first resend, the retransmission interval increases exponentially until it exceeds *waittime*.

rpc_call()

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The *inproc* argument is used to encode the procedure's arguments, and *outproc* is used to decode the procedure's results; *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). The *nettype* argument can be any of the values listed on rpc(3). This routine returns RPC_SUCCESS if it succeeds, or an appropriate status is returned. Use the **clnt_perrno**() routine to translate failure status into error messages. Warning: **rpc_call**() uses the first available transport belonging to the class *nettype*, on which it can create a connection. You do not have control of timeouts or authentication using this routine.

SEE ALSO

printf(3), rpc(3), rpc_clnt_auth(3), rpc_clnt_create(3)

NAME

rpc_svc_reg, **rpc_reg**, **svc_reg**, **svc_unreg**, **svc_auth_reg**, **xprt_register**, **xprt_unregister** - library routines for registering servers

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

int
rpc_reg(*rpcprog_t prognum*, *rpcvers_t versnum*, *rpcproc_t procnum*, *char *(*procname)()*, *xdrproc_t inproc*, *xdrproc_t outproc*, *char *nettype*);

bool_t
svc_reg(*SVCXPRT *xprt*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *void (*dispatch)(struct svc_req *, SVCXPRT *)*, *const struct netconfig *netconf*);

void
svc_unreg(*const rpcprog_t prognum*, *const rpcvers_t versnum*);

int
svc_auth_reg(*int cred_flavor*, *enum auth_stat (*handler)(struct svc_req *, struct rpc_msg *)*);

void
xprt_register(*SVCXPRT *xprt*);

void
xprt_unregister(*SVCXPRT *xprt*);

DESCRIPTION

These routines are a part of the RPC library which allows the RPC servers to register themselves with rpcbind (see rpcbind(8)), and associate the given program and version number with the dispatch function. When the RPC server receives a RPC request, the library invokes the dispatch routine with the appropriate arguments.

Routines

See rpc(3) for the definition of the *SVCXPRT* data structure.

rpc_reg()

Register program *prognum*, procedure *procname*, and version *versnum* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its argument(s); *procname* should return a pointer to its static result(s); *inproc* is the XDR function used to decode the arguments while *outproc* is the XDR function used to encode the results. Procedures are registered on all available transports of the class *nettype*. See rpc(3). This routine returns 0 if the registration succeeded, -1 otherwise.

svc_reg()

Associates *prognum* and *versnum* with the service dispatch procedure, *dispatch*. If *netconf* is NULL, the service is not registered with the rpcbind(8) service. If *netconf* is non-zero, then a mapping of the triple [*prognum*, *versnum*, *netconf->nc_netid*] to *xprt->xp_ltaddr* is established with the local rpcbind service.

The **svc_reg**() routine returns 1 if it succeeds, and 0 otherwise.

svc_unreg()

Remove from the rpcbind service, all mappings of the triple [*prognum*, *versnum*, all-transports] to network address and all mappings within the RPC service package of the double [*prognum*, *versnum*] to dispatch routines.

svc_auth_reg()

Registers the service authentication routine *handler* with the dispatch mechanism so that it can be invoked to authenticate RPC requests received with authentication type *cred_flavor*. This interface allows developers to add new authentication types to their RPC applications without needing to modify the libraries. Service implementors usually do not need this routine.

Typical service application would call **svc_auth_reg**() after registering the service and prior to calling **svc_run**(). When needed to process an RPC credential of type *cred_flavor*, the *handler* procedure will be called with two arguments, *struct svc_req *rqst* and *struct rpc_msg *msg*, and is expected to return a valid *enum auth_stat* value. There is no provision to change or delete an authentication handler once registered.

The **svc_auth_reg**() routine returns 0 if the registration is successful, 1 if *cred_flavor* already has an authentication handler registered for it, and -1 otherwise.

xprt_register()

After RPC service transport handle *xprt* is created, it is registered with the RPC service package. This routine modifies the global variable *svc_fdset* (see rpc_svc_calls(3)). Service implementors usually do not need this routine.

xprt_unregister()

Before an RPC service transport handle *xprt* is destroyed, it unregisters itself with the RPC service package. This routine modifies the global variable *svc_fdset* (see rpc_svc_calls(3)). Service implementors usually do not need this routine.

SEE ALSO

select(2), rpc(3), rpc_svc_calls(3), rpc_svc_create(3), rpc_svc_err(3), rpcbind(3), rpcbind(8)

NAME

rpc_svc_create, **svc_control**, **svc_create**, **svc_destroy**, **svc_dg_create**, **svc_fd_create**, **svc_raw_create**, **svc_tli_create**, **svc_tp_create**, **svc_vc_create** - library routines for the creation of server handles

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

bool_t
svc_control(*SVCXPRT *svc*, *const u_int req*, *void *info*);

int
svc_create(*void (*dispatch)(struct svc_req *, SVCXPRT *)*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *const char *nettype*);

*SVCXPRT **
svc_dg_create(*const int fildes*, *const u_int sendsz*, *const u_int recvsz*);

void
svc_destroy(*SVCXPRT *xprt*);

*SVCXPRT **
svc_fd_create(*const int fildes*, *const u_int sendsz*, *const u_int recvsz*);

*SVCXPRT **
svc_raw_create(*void*);

*SVCXPRT **
svc_tli_create(*const int fildes*, *const struct netconfig *netconf*, *const struct t_bind *bindaddr*, *const u_int sendsz*, *const u_int recvsz*);

*SVCXPRT **
svc_tp_create(*void (*dispatch)(struct svc_req *, SVCXPRT *)*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *const struct netconfig *netconf*);

*SVCXPRT **
svc_vc_create(*const int fildes*, *const u_int sendsz*, *const u_int recvsz*);

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on servers across the network. These routines deal with the creation of service handles. Once the handle is created, the server can be invoked by calling **svc_run**().

Routines

See rpc(3) for the definition of the *SVCXPRT* data structure.

svc_control()

A function to change or retrieve various information about a service object. The *req* argument indicates the type of operation and *info* is a pointer to the information. The supported values of *req*, their argument types, and what they do are:

SVCGET_VERSQUIET

If a request is received for a program number served by this server but the version number is outside the range registered with the server, an RPC_PROGVERSMISMATCH error will normally be returned. The *info* argument should be a pointer to an integer. Upon successful completion of the SVCGET_VERSQUIET request, **info* contains an integer which describes the server's current behavior: 0 indicates normal server behavior (that is, an RPC_PROGVERSMISMATCH error will be returned); 1 indicates that the out of range request will be silently ignored.

SVCSET_VERSQUIET

If a request is received for a program number served by this server but the version number is outside the range registered with the server, an RPC_PROGVERSMISMATCH error will normally be returned. It is sometimes desirable to change this behavior. The *info* argument should be a pointer to an integer which is either 0 (indicating normal server behavior - an RPC_PROGVERSMISMATCH error will be returned), or 1 (indicating that the out of range request should be silently ignored).

svc_create()

The **svc_create**() function creates server handles for all the transports belonging to the class *nettype*. The *nettype* argument defines a class of transports which can be used for a particular application. The transports are tried in left to right order in NETPATH variable or in top to bottom order in the netconfig database. If *nettype* is NULL, it defaults to "netpath".

The **svc_create**() function registers itself with the rpcbind service (see rpcbind(8)). The *dispatch* function is called when there is a remote procedure call for the given *prognum* and

versnum; this requires calling **svc_run**() (see **svc_run**() in rpc_svc_reg(3)). If **svc_create**() succeeds, it returns the number of server handles it created, otherwise it returns 0 and an error message is logged.

svc_destroy()

A function macro that destroys the RPC service handle *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

svc_dg_create()

This routine creates a connectionless RPC service handle, and returns a pointer to it. This routine returns NULL if it fails, and an error message is logged. The *sendsz* and *recvsz* arguments are arguments used to specify the size of the buffers. If they are 0, suitable defaults are chosen. The file descriptor *fildes* should be open and bound. The server is not registered with rpcbind(8).

Warning: since connectionless-based RPC messages can only hold limited amount of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

svc_fd_create()

This routine creates a service on top of an open and bound file descriptor, and returns the handle to it. Typically, this descriptor is a connected file descriptor for a connection-oriented transport. The *sendsz* and *recvsz* arguments indicate sizes for the send and receive buffers. If they are 0, reasonable defaults are chosen. This routine returns NULL if it fails, and an error message is logged.

svc_raw_create()

This routine creates an RPC service handle and returns a pointer to it. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; (see **clnt_raw_create**() in rpc_clnt_create(3)). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel and networking interference. This routine returns NULL if it fails, and an error message is logged.

Note: **svc_run**() should not be called when the raw interface is being used.

svc_tli_create()

This routine creates an RPC server handle, and returns a pointer to it. The *fildes* argument is the file descriptor on which the service is listening. If *fildes* is RPC_ANYFD, it opens a file

descriptor on the transport specified by *netconf*. If the file descriptor is unbound and *bindaddr* is not NULL, *fildes* is bound to the address specified by *bindaddr*, otherwise *fildes* is bound to a default address chosen by the transport.

Note: the *t_bind* structure comes from the TLI/XTI SysV interface, which NetBSD does not use. The structure is defined in <*rpc/types.h*> for compatibility as:

```
struct t_bind {
   struct netbuf addr;          /* network address, see rpc(3) */
   unsigned int  qlen;          /* queue length (for listen(2)) */
};
```

In the case where the default address is chosen, the number of outstanding connect requests is set to 8 for connection-oriented transports. The user may specify the size of the send and receive buffers with the arguments *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails, and an error message is logged. The server is not registered with the rpcbind(8) service.

svc_tp_create()

The **svc_tp_create**() function creates a server handle for the network specified by *netconf*, and registers itself with the rpcbind service. The *dispatch* function is called when there is a remote procedure call for the given *prognum* and *versnum*; this requires calling **svc_run**(). The **svc_tp_create**() function returns the service handle if it succeeds, otherwise a NULL is returned and an error message is logged.

svc_vc_create()

This routine creates a connection-oriented RPC service and returns a pointer to it. This routine returns NULL if it fails, and an error message is logged. The users may specify the size of the send and receive buffers with the arguments *sendsz* and *recvsz*; values of 0 choose suitable defaults. The file descriptor *fildes* should be open and bound. The server is not registered with the rpcbind(8) service.

SEE ALSO

rpc(3), rpc_svc_calls(3), rpc_svc_err(3), rpc_svc_reg(3), rpcbind(8)

NAME

rpc_svc_err, **svcerr_auth**, **svcerr_decode**, **svcerr_noproc**, **svcerr_noprog**, **svcerr_progvers**, **svcerr_systemerr**, **svcerr_weakauth** - library routines for server side remote procedure call errors

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

void
svcerr_auth(*SVCXPRT *xprt*, *enum auth_stat why*);

void
svcerr_decode(*SVCXPRT *xprt*);

void
svcerr_noproc(*SVCXPRT *xprt*);

void
svcerr_noprog(*SVCXPRT *xprt*);

void
svcerr_progvers(*SVCXPRT *xprt*, *rpcvers_t low_vers*, *rpcvers_t high_vers*);

void
svcerr_systemerr(*SVCXPRT *xprt*);

void
svcerr_weakauth(*SVCXPRT *xprt*);

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines can be called by the server side dispatch function if there is any error in the transaction with the client.

Routines

See rpc(3) for the definition of the *SVCXPRT* data structure.

svcerr_auth()
Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode()
Called by a service dispatch routine that cannot successfully decode the remote arguments (see **svc_getargs**() in rpc_svc_reg(3)).

svcerr_noproc()
Called by a service dispatch routine that does not implement the procedure number that the caller requests.

svcerr_noprog()
Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

svcerr_progvers()
Called when the desired version of a program is not registered with the RPC package. The *low_vers* argument is the lowest version number, and *high_vers* is the highest version number. Service implementors usually do not need this routine.

svcerr_systemerr()
Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()
Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication arguments. The routine calls **svcerr_auth**(*xprt*, *AUTH_TOOWEAK*).

SEE ALSO

rpc(3), rpc_svc_calls(3), rpc_svc_create(3), rpc_svc_reg(3)

NAME

rpc - library routines for remote procedure calls

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>
#include <netconfig.h>

DESCRIPTION

These routines allow C language programs to make procedure calls on other machines across a network. First, the client sends a request to the server. On receipt of the request, the server calls a dispatch routine to perform the requested service, and then sends back a reply.

All RPC routines require the header <*rpc/rpc.h*>. Routines that take a *struct netconfig* also require that <*netconfig.h*> be included.

Nettype

Some of the high-level RPC interface routines take a *nettype* string as one of the arguments (for example, **clnt_create**(), **svc_create**(), **rpc_reg**(), **rpc_call**()). This string defines a class of transports which can be used for a particular application.

The *nettype* argument can be one of the following:

netpath Choose from the transports which have been indicated by their token names in the NETPATH environment variable. NETPATH is unset or NULL, it defaults to "visible". "netpath" is the default *nettype*.

visible Choose the transports which have the visible flag (v) set in the */etc/netconfig* file.

circuit_v This is same as "visible" except that it chooses only the connection oriented transports (semantics "tpi_cots" or "tpi_cots_ord") from the entries in the */etc/netconfig* file.

datagram_v This is same as "visible" except that it chooses only the connectionless datagram transports (semantics "tpi_clts") from the entries in the */etc/netconfig* file.

circuit_n This is same as "netpath" except that it chooses only the connection oriented datagram transports (semantics "tpi_cots" or "tpi_cots_ord").

datagram_n This is same as "netpath" except that it chooses only the connectionless datagram transports (semantics "tpi_clts").

udp This refers to Internet UDP, both version 4 and 6.

tcp This refers to Internet TCP, both version 4 and 6.

If *nettype* is NULL, it defaults to "netpath". The transports are tried in left to right order in the NETPATH variable or in top to down order in the */etc/netconfig* file.

Derived Types

The derived types used in the RPC interfaces are defined as follows:

```
typedef uint32_t rpcprog_t;
typedef uint32_t rpcvers_t;
typedef uint32_t rpcproc_t;
typedef uint32_t rpcprot_t;
typedef uint32_t rpcport_t;
typedef int32_t  rpc_inline_t;
```

Data Structures

Some of the data structures used by the RPC package are shown below.

The AUTH Structure

```
/*
 * Authentication info. Opaque to client.
 */
struct opaque_auth {
   enum_t    oa_flavor;    /* flavor of auth */
   caddr_t    oa_base;    /* address of more auth stuff */
   u_int    oa_length;    /* not to exceed MAX_AUTH_BYTES */
};

/*
 * Auth handle, interface to client side authenticators.
 */
typedef struct {
   struct    opaque_auth    ah_cred;
   struct    opaque_auth    ah_verf;
   struct auth_ops {
```

```
        void    (*ah_nextverf)();
        int     (*ah_marshal)();    /* nextverf & serialize */
        int     (*ah_validate)();   /* validate verifier */
        int     (*ah_refresh)();    /* refresh credentials */
        void    (*ah_destroy)();    /* destroy this structure */
      } *ah_ops;
      caddr_t ah_private;
    } AUTH;
```

The CLIENT Structure

```
    /*
     * Client rpc handle.
     * Created by individual implementations.
     * Client is responsible for initializing auth.
     */

    typedef struct {
      AUTH    *cl_auth;    /* authenticator */
      struct clnt_ops {
        enum clnt_stat    (*cl_call)();    /* call remote procedure */
        void    (*cl_abort)();        /* abort a call */
        void    (*cl_geterr)();        /* get specific error code */
        bool_t    (*cl_freeres)();    /* frees results */
        void    (*cl_destroy)();    /* destroy this structure */
        bool_t    (*cl_control)();    /* the ioctl() of rpc */
      } *cl_ops;
      caddr_t    cl_private;    /* private stuff */
      char    *cl_netid;    /* network identifier */
      char    *cl_tp;        /* device name */
    } CLIENT;
```

The SVCXPRT structure

```
    enum xprt_stat {
      XPRT_DIED,
      XPRT_MOREREQS,
      XPRT_IDLE
    };

    /*
     * Server side transport handle
```

```
 */
typedef struct {
   int    xp_fd;    /* file descriptor for the server handle */
   u_short    xp_port;    /* obsolete */
   const struct xp_ops {
      bool_t    (*xp_recv)();    /* receive incoming requests */
      enum xprt_stat    (*xp_stat)();    /* get transport status */
      bool_t    (*xp_getargs)();    /* get arguments */
      bool_t    (*xp_reply)();      /* send reply */
      bool_t    (*xp_freeargs)(); /* free mem allocated for args */
      void    (*xp_destroy)();    /* destroy this struct */
   } *xp_ops;
   int    xp_addrlen;    /* length of remote addr.  Obsolete */
   struct sockaddr_in    xp_raddr; /* Obsolete */
   const struct xp_ops2 {
      bool_t    (*xp_control)();    /* catch-all function */
   } *xp_ops2;
   char    *xp_tp;    /* transport provider device name */
   char    *xp_netid;    /* network identifier */
   struct netbuf    xp_ltaddr;    /* local transport address */
   struct netbuf    xp_rtaddr;    /* remote transport address */
   struct opaque_auth    xp_verf;    /* raw response verifier */
   caddr_t    xp_p1;    /* private: for use by svc ops */
   caddr_t    xp_p2;    /* private: for use by svc ops */
   caddr_t    xp_p3;    /* private: for use by svc lib */
   int    xp_type    /* transport type */
} SVCXPRT;
```

The svc_reg structure

```
struct svc_req {
   rpcprog_t    rq_prog;    /* service program number */
   rpcvers_t    rq_vers;    /* service protocol version */
   rpcproc_t    rq_proc;    /* the desired procedure */
   struct opaque_auth    rq_cred;    /* raw creds from the wire */
   caddr_t    rq_clntcred;    /* read only cooked cred */
   SVCXPRT    *rq_xprt;    /* associated transport */
};
```

The XDR structure

```
/*
```

```
 * XDR operations.
 * XDR_ENCODE causes the type to be encoded into the stream.
 * XDR_DECODE causes the type to be extracted from the stream.
 * XDR_FREE can be used to release the space allocated by an XDR_DECODE
 * request.
 */
enum xdr_op {
   XDR_ENCODE=0,
   XDR_DECODE=1,
   XDR_FREE=2
};
/*
 * This is the number of bytes per unit of external data.
 */
#define BYTES_PER_XDR_UNIT    (4)
#define RNDUP(x)  ((((x) + BYTES_PER_XDR_UNIT - 1) /
              BYTES_PER_XDR_UNIT) \ * BYTES_PER_XDR_UNIT)

/*
 * A xdrproc_t exists for each data type which is to be encoded or
 * decoded.  The second argument to the xdrproc_t is a pointer to
 * an opaque pointer.  The opaque pointer generally points to a
 * structure of the data type to be decoded.  If this points to 0,
 * then the type routines should allocate dynamic storage of the
 * appropriate size and return it.
 * bool_t  (*xdrproc_t)(XDR *, caddr_t *);
 */
typedef  bool_t (*xdrproc_t)();

/*
 * The XDR handle.
 * Contains operation which is being applied to the stream,
 * an operations vector for the particular implementation
 */
typedef struct {
   enum xdr_op    x_op;    /* operation; fast additional param */
   struct xdr_ops {
      bool_t    (*x_getlong)();    /* get a long from underlying stream */
      bool_t    (*x_putlong)();    /* put a long to underlying stream */
      bool_t    (*x_getbytes)(); /* get bytes from underlying stream */
```

```
        bool_t    (*x_putbytes)(); /* put bytes to underlying stream */
        u_int    (*x_getpostn)(); /* returns bytes off from beginning */
        bool_t    (*x_setpostn)(); /* lets you reposition the stream */
        long *    (*x_inline)();    /* buf quick ptr to buffered data */
        void    (*x_destroy)();    /* free privates of this xdr_stream */
      } *x_ops;
      caddr_t    x_public;    /* users' data */
      caddr_t    x_private;    /* pointer to private data */
      caddr_t    x_base;    /* private used for position info */
      u_int    x_handy;    /* extra private word */
    } XDR;

    /*
     * The netbuf structure. This structure is defined in <xti.h> on SysV
     * systems, but NetBSD / FreeBSD do not use XTI.
     *
     * Usually, buf will point to a struct sockaddr, and len and maxlen
     * will contain the length and maximum length of that socket address,
     * respectively.
     */
    struct netbuf {
            unsigned int maxlen;
            unsigned int len;
            void *buf;
    };

    /*
     * The format of the address and options arguments of the XTI t_bind call.
     * Only provided for compatibility, it should not be used other than
     * as an argument to svc_tli_create().
     */

    struct t_bind {
            struct netbuf   addr;
            unsigned int    qlen;
    };
```

Index to Routines

The following table lists RPC routines and the manual reference pages on which they are described:

RPC Routine	*Manual Reference Page*
auth_destroy()	rpc_clnt_auth(3)
authdes_create()	rpc_soc(3)
authnone_create()	rpc_clnt_auth(3)
authsys_create()	rpc_clnt_auth(3)
authsys_create_default()	rpc_clnt_auth(3)
authunix_create()	rpc_soc(3)
authunix_create_default()	rpc_soc(3)
callrpc()	rpc_soc(3)
clnt_broadcast()	rpc_soc(3)
clnt_call()	rpc_clnt_calls(3)
clnt_control()	rpc_clnt_create(3)
clnt_create()	rpc_clnt_create(3)
clnt_create_timed()	rpc_clnt_create(3)
clnt_create_vers()	rpc_clnt_create(3)
clnt_create_vers_timed()	rpc_clnt_create(3)
clnt_destroy()	rpc_clnt_create(3)
clnt_dg_create()	rpc_clnt_create(3)
clnt_freeres()	rpc_clnt_calls(3)
clnt_geterr()	rpc_clnt_calls(3)
clnt_pcreateerror()	rpc_clnt_create(3)
clnt_perrno()	rpc_clnt_calls(3)
clnt_perror()	rpc_clnt_calls(3)
clnt_raw_create()	rpc_clnt_create(3)
clnt_spcreateerror()	rpc_clnt_create(3)
clnt_sperrno()	rpc_clnt_calls(3)
clnt_sperror()	rpc_clnt_calls(3)
clnt_tli_create()	rpc_clnt_create(3)
clnt_tp_create()	rpc_clnt_create(3)
clnt_tp_create_timed()	rpc_clnt_create(3)
clnt_udpcreate()	rpc_soc(3)
clnt_vc_create()	rpc_clnt_create(3)
clntraw_create()	rpc_soc(3)
clnttcp_create()	rpc_soc(3)
clntudp_bufcreate()	rpc_soc(3)
get_myaddress()	rpc_soc(3)
pmap_getmaps()	rpc_soc(3)
pmap_getport()	rpc_soc(3)
pmap_rmtcall()	rpc_soc(3)

pmap_set() rpc_soc(3)
pmap_unset() rpc_soc(3)
registerrpc() rpc_soc(3)
rpc_broadcast() rpc_clnt_calls(3)
rpc_broadcast_exp() rpc_clnt_calls(3)
rpc_call() rpc_clnt_calls(3)
rpc_reg() rpc_svc_calls(3)
svc_create() rpc_svc_create(3)
svc_destroy() rpc_svc_create(3)
svc_dg_create() rpc_svc_create(3)
svc_dg_enablecache() rpc_svc_calls(3)
svc_fd_create() rpc_svc_create(3)
svc_fds() rpc_soc(3)
svc_freeargs() rpc_svc_reg(3)
svc_getargs() rpc_svc_reg(3)
svc_getcaller() rpc_soc(3)
svc_getreq() rpc_soc(3)
svc_getreqset() rpc_svc_calls(3)
svc_getrpccaller() rpc_svc_calls(3)
svc_kerb_reg() kerberos_rpc(3)
svc_raw_create() rpc_svc_create(3)
svc_reg() rpc_svc_calls(3)
svc_register() rpc_soc(3)
svc_run() rpc_svc_reg(3)
svc_sendreply() rpc_svc_reg(3)
svc_tli_create() rpc_svc_create(3)
svc_tp_create() rpc_svc_create(3)
svc_unreg() rpc_svc_calls(3)
svc_unregister() rpc_soc(3)
svc_vc_create() rpc_svc_create(3)
svcerr_auth() rpc_svc_err(3)
svcerr_decode() rpc_svc_err(3)
svcerr_noproc() rpc_svc_err(3)
svcerr_noprog() rpc_svc_err(3)
svcerr_progvers() rpc_svc_err(3)
svcerr_systemerr() rpc_svc_err(3)
svcerr_weakauth() rpc_svc_err(3)
svcfd_create() rpc_soc(3)
svcraw_create() rpc_soc(3)
svctcp_create() rpc_soc(3)

svcudp_bufcreate()	rpc_soc(3)
svcudp_create()	rpc_soc(3)
xdr_accepted_reply()	rpc_xdr(3)
xdr_authsys_parms()	rpc_xdr(3)
xdr_authunix_parms()	rpc_soc(3)
xdr_callhdr()	rpc_xdr(3)
xdr_callmsg()	rpc_xdr(3)
xdr_opaque_auth()	rpc_xdr(3)
xdr_rejected_reply()	rpc_xdr(3)
xdr_replymsg()	rpc_xdr(3)
xprt_register()	rpc_svc_calls(3)
xprt_unregister()	rpc_svc_calls(3)

FILES

/etc/netconfig

SEE ALSO

getnetconfig(3), getnetpath(3), rpc_clnt_auth(3), rpc_clnt_calls(3), rpc_clnt_create(3), rpc_svc_calls(3), rpc_svc_create(3), rpc_svc_err(3), rpc_svc_reg(3), rpc_xdr(3), rpcbind(3), xdr(3), netconfig(5)

NAME

rpcb_getmaps, **rpcb_getaddr**, **rpcb_gettime**, **rpcb_rmtcall**, **rpcb_set**, **rpcb_unset** - library routines for RPC bind service

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

*rpcblist **
rpcb_getmaps(*const struct netconfig *netconf*, *const char *host*);

bool_t
rpcb_getaddr(*const rpcprog_t prognum*, *const rpcvers_t versnum*, *const struct netconfig *netconf*, *struct netbuf *svcaddr*, *const char *host*);

bool_t
rpcb_gettime(*const char *host*, *time_t * timep*);

enum clnt_stat
rpcb_rmtcall(*const struct netconfig *netconf*, *const char *host*, *const rpcprog_t prognum*, *const rpcvers_t versnum*, *const rpcproc_t procnum*, *const xdrproc_t inproc*, *const caddr_t in*, *const xdrproc_t outproc*, *const caddr_t out*, *const struct timeval tout*, *const struct netbuf *svcaddr*);

bool_t
rpcb_set(*const rpcprog_t prognum*, *const rpcvers_t versnum*, *const struct netconfig *netconf*, *const struct netbuf *svcaddr*);

bool_t
rpcb_unset(*const rpcprog_t prognum*, *const rpcvers_t versnum*, *const struct netconfig *netconf*);

DESCRIPTION

These routines allow client C programs to make procedure calls to the RPC binder service. (see rpcbind(8)) maintains a list of mappings between programs and their universal addresses.

Routines

rpcb_getmaps()

An interface to the rpcbind service, which returns a list of the current RPC program-to-address mappings on *host*. It uses the transport specified through *netconf* to contact the remote

rpcbind service on *host*. This routine will return NULL, if the remote rpcbind could not be contacted.

rpcb_getaddr()

An interface to the rpcbind service, which finds the address of the service on *host* that is registered with program number *prognum*, version *versnum*, and speaks the transport protocol associated with *netconf*. The address found is returned in *svcaddr*. The *svcaddr* argument should be preallocated. This routine returns TRUE if it succeeds. A return value of FALSE means that the mapping does not exist or that the RPC system failed to contact the remote rpcbind service. In the latter case, the global variable *rpc_createerr* (see rpc_clnt_create(3)) contains the RPC status.

rpcb_gettime()

This routine returns the time on *host* in *timep*. If *host* is NULL, **rpcb_gettime**() returns the time on its own machine. This routine returns TRUE if it succeeds, FALSE if it fails. The **rpcb_gettime**() function can be used to synchronize the time between the client and the remote server.

rpcb_rmtcall()

An interface to the rpcbind service, which instructs rpcbind on *host* to make an RPC call on your behalf to a procedure on that host. The **netconfig**() structure should correspond to a connectionless transport. The *svcaddr* argument will be modified to the server's address if the procedure succeeds (see **rpc_call**() and **clnt_call**() in rpc_clnt_calls(3) for the definitions of other arguments).

This procedure should normally be used for a "ping" and nothing else. This routine allows programs to do lookup and call, all in one step.

Note: Even if the server is not running **rpcb_rmtcall**() does not return any error messages to the caller. In such a case, the caller times out.

Note: **rpcb_rmtcall**() is only available for connectionless transports.

rpcb_set()

An interface to the rpcbind service, which establishes a mapping between the triple [*prognum*, *versnum*, *netconf->nc_netid*] and *svcaddr* on the machine's rpcbind service. The value of *nc_netid* must correspond to a network identifier that is defined by the netconfig database. This routine returns TRUE if it succeeds, FALSE otherwise. (See also **svc_reg**() in rpc_svc_calls(3).) If there already exists such an entry with rpcbind, **rpcb_set**() will fail.

rpcb_unset()

An interface to the rpcbind service, which destroys the mapping between the triple [*prognum*, *versnum*, *netconf->nc_netid*] and the address on the machine's rpcbind service. If *netconf* is NULL, **rpcb_unset**() destroys all mapping between the triple [*prognum*, *versnum*, all-transports] and the addresses on the machine's rpcbind service. This routine returns TRUE if it succeeds, FALSE otherwise. Only the owner of the service or the super-user can destroy the mapping. (See also **svc_unreg**() in rpc_svc_calls(3).)

SEE ALSO

rpc_clnt_calls(3), rpc_svc_calls(3), rpcbind(8), rpcinfo(8)

NAME

rpmatch - determine whether the response to a question is affirmative or negative

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
rpmatch(*const char *response*);

DESCRIPTION

The **rpmatch**() function determines whether the *response* argument is an affirmative or negative response to a question according to the current locale.

RETURN VALUES

The **rpmatch**() functions returns:

1 The response is affirmative.

0 The response is negative.

-1 The response is not recognized.

SEE ALSO

nl_langinfo(3), setlocale(3)

HISTORY

The **rpmatch**() function appeared in FreeBSD 6.0.

NAME

rtime - get remote time

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/time.h>
#include <netinet/in.h>

int
rtime(*struct sockaddr_in *addrp*, *struct timeval *timep*, *struct timeval *timeout*);

DESCRIPTION

The **rtime**() function consults the Internet Time Server at the address pointed to by *addrp* and returns the remote time in the *timeval* struct pointed to by *timep*. Normally, the UDP protocol is used when consulting the Time Server. The *timeout* argument specifies how long the routine should wait before giving up when waiting for a reply. If *timeout* is specified as NULL, however, the routine will instead use TCP and block until a reply is received from the time server.

RETURN VALUES

The **rtime**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

SEE ALSO

timed(8)

NAME

rtprio, **rtprio_thread** - examine or modify realtime or idle priority

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/rtprio.h>

int
rtprio(*int function*, *pid_t pid*, *struct rtprio *rtp*);

int
rtprio_thread(*int function*, *lwpid_t lwpid*, *struct rtprio *rtp*);

DESCRIPTION

The **rtprio**() system call is used to lookup or change the realtime or idle priority of a process, or the calling thread. The **rtprio_thread**() system call is used to lookup or change the realtime or idle priority of a thread.

The *function* argument specifies the operation to be performed. RTP_LOOKUP to lookup the current priority, and RTP_SET to set the priority.

For the **rtprio**() system call, the *pid* argument specifies the process to operate on, 0 for the calling thread. When *pid* is non-zero, the system call reports the highest priority in the process, or sets all threads' priority in the process, depending on value of the *function* argument.

For the **rtprio_thread**() system call, the *lwpid* specifies the thread to operate on, 0 for the calling thread.

The **rtp* argument is a pointer to a struct rtprio which is used to specify the priority and priority type. This structure has the following form:

```
struct rtprio {
        u_short    type;
        u_short prio;
};
```

The value of the *type* field may be RTP_PRIO_REALTIME for realtime priorities, RTP_PRIO_NORMAL for normal priorities, and RTP_PRIO_IDLE for idle priorities. The priority

specified by the *prio* field ranges between 0 and RTP_PRIO_MAX (usually 31). 0 is the highest possible priority.

Realtime and idle priority is inherited through fork() and exec().

A realtime thread can only be preempted by a thread of equal or higher priority, or by an interrupt; idle priority threads will run only when no other real/normal priority thread is runnable. Higher real/idle priority threads preempt lower real/idle priority threads. Threads of equal real/idle priority are run round-robin.

RETURN VALUES

The **rtprio**() and **rtprio_thread**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **rtprio**() and **rtprio_thread**() system calls will fail if:

[EFAULT] The rtp pointer passed to **rtprio**() or **rtprio_thread**() was invalid.

[EINVAL] The specified *prio* was out of range.

[EPERM] The calling thread is not allowed to set the realtime priority. Only root is allowed to change the realtime priority of any thread, and non-root may only change the idle priority of threads the user owns, when the sysctl(8) variable *security.bsd.unprivileged_idprio* is set to non-zero.

[ESRCH] The specified process or thread was not found or visible.

SEE ALSO

nice(1), ps(1), rtprio(1), setpriority(2), nice(3), renice(8), p_cansee(9)

AUTHORS

The original author was Henrik Vestergaard Draboel <*hvd@terry.ping.dk*>. This implementation in FreeBSD was substantially rewritten by David Greenman. The **rtprio_thread**() system call was implemented by David Xu.

NAME

sched_get_priority_max, **sched_get_priority_min**, **sched_rr_get_interval** - get scheduling parameter limits

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sched.h>

int
sched_get_priority_max(*int policy*);

int
sched_get_priority_min(*int policy*);

int
sched_rr_get_interval(*pid_t pid*, *struct timespec *interval*);

DESCRIPTION

The **sched_get_priority_max**() and **sched_get_priority_min**() system calls return the appropriate maximum or minimum, respectively, for the scheduling policy specified by *policy*. The **sched_rr_get_interval**() system call updates the *timespec* structure referenced by the *interval* argument to contain the current execution time limit (i.e., time quantum) for the process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process is returned.

The value of *policy* should be one of the scheduling policy values defined in <*sched.h*>:

[SCHED_FIFO] First-in-first-out fixed priority scheduling with no round robin scheduling;

[SCHED_OTHER] The standard time sharing scheduler;

[SCHED_RR] Round-robin scheduling across same priority processes.

RETURN VALUES

If successful, the **sched_get_priority_max**() and **sched_get_priority_min**() system calls shall return the appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a value of -1 and set *errno* to indicate the error.

The **sched_rr_get_interval**() function returns the value 0 if successful; otherwise the value -1 is returned

and the global variable *errno* is set to indicate the error.

ERRORS

On failure *errno* will be set to the corresponding value:

[EINVAL] The value of the *policy* argument does not represent a defined scheduling policy.

[ENOSYS] The **sched_get_priority_max**(), **sched_get_priority_min**(), and **sched_rr_get_interval**() system calls are not supported by the implementation.

[ESRCH] No process can be found corresponding to that specified by *pid*.

SEE ALSO

sched_getparam(2), sched_getscheduler(2), sched_setparam(2), sched_setscheduler(2)

STANDARDS

The **sched_get_priority_max**(), **sched_get_priority_min**(), and **sched_rr_get_interval**() system calls conform to IEEE Std 1003.1b-1993 ("POSIX.1b").

NAME

sched_setparam, **sched_getparam** - set/get scheduling parameters

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sched.h>

int
sched_setparam(*pid_t pid*, *const struct sched_param *param*);

int
sched_getparam(*pid_t pid*, *struct sched_param *param*);

DESCRIPTION

The **sched_setparam**() system call sets the scheduling parameters of the process specified by *pid* to the values specified by the *sched_param* structure pointed to by *param*. The value of the *sched_priority* member in the *param* structure must be any integer within the inclusive priority range for the current scheduling policy of the process specified by *pid*. Higher numerical values for the priority represent higher priorities.

In this implementation, if the value of *pid* is negative the system call will fail.

If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters are set for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling parameters are set for the calling process.

In this implementation, the policy of when a process can affect the scheduling parameters of another process is specified in IEEE Std 1003.1b-1993 ("POSIX.1b") as a write-style operation.

The target process, whether it is running or not running, will resume execution after all other runnable processes of equal or greater priority have been scheduled to run.

If the priority of the process specified by the *pid* argument is set higher than that of the lowest priority running process and if the specified process is ready to run, the process specified by the *pid* argument will preempt a lowest priority running process. Similarly, if the process calling **sched_setparam**() sets its own priority lower than that of one or more other nonempty process lists, then the process that is the head of the highest priority list will also preempt the calling process. Thus, in either case, the

originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed.

In this implementation, when the current scheduling policy for the process specified by *pid* is normal timesharing (SCHED_OTHER, aka SCHED_NORMAL when not POSIX-source) or the idle policy (SCHED_IDLE when not POSIX-source) then the behavior is as if the process had been running under SCHED_RR with a priority lower than any actual realtime priority.

The **sched_getparam**() system call will return the scheduling parameters of a process specified by *pid* in the *sched_param* structure pointed to by *param*.

If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to *pid* are returned.

In this implementation, the policy of when a process can obtain the scheduling parameters of another process are detailed in IEEE Std 1003.1b-1993 ("POSIX.1b") as a read-style operation.

If *pid* is zero, the scheduling parameters for the calling process will be returned. In this implementation, the *sched_getparam* system call will fail if *pid* is negative.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

On failure *errno* will be set to the corresponding value:

[ENOSYS]	The system is not configured to support this functionality.
[EPERM]	The requesting process doesn not have permission as detailed in IEEE Std 1003.1b-1993 ("POSIX.1b").
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[EINVAL]	For **sched_setparam**(): one or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.

SEE ALSO

sched_get_priority_max(2), sched_get_priority_min(2), sched_getscheduler(2), sched_rr_get_interval(2), sched_setscheduler(2), sched_yield(2)

STANDARDS

The **sched_setparam**() and **sched_getparam**() system calls conform to IEEE Std 1003.1b-1993 ("POSIX.1b").

NAME

sched_setscheduler, **sched_getscheduler** - set/get scheduling policy and scheduler parameters

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sched.h>

int
sched_setscheduler(*pid_t pid*, *int policy*, *const struct sched_param *param*);

int
sched_getscheduler(*pid_t pid*);

DESCRIPTION

The **sched_setscheduler**() system call sets the scheduling policy and scheduling parameters of the process specified by *pid* to *policy* and the parameters specified in the *sched_param* structure pointed to by *param*, respectively. The value of the *sched_priority* member in the *param* structure must be any integer within the inclusive priority range for the scheduling policy specified by *policy*.

In this implementation, if the value of *pid* is negative the system call will fail.

If a process specified by *pid* exists and if the calling process has permission, the scheduling policy and scheduling parameters will be set for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling policy and scheduling parameters are set for the calling process.

In this implementation, the policy of when a process can affect the scheduling parameters of another process is specified in IEEE Std 1003.1b-1993 ("POSIX.1b") as a write-style operation.

The scheduling policies are in <*sched.h*>:

[SCHED_FIFO] First-in-first-out fixed priority scheduling with no round robin scheduling;

[SCHED_OTHER] The standard time sharing scheduler;

[SCHED_RR] Round-robin scheduling across same priority processes.

The *sched_param* structure is defined in <*sched.h*>:

```
struct sched_param {
        int sched_priority;  /* scheduling priority */
};
```

The **sched_getscheduler**() system call returns the scheduling policy of the process specified by *pid*.

If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to *pid* are returned.

In this implementation, the policy of when a process can obtain the scheduling parameters of another process are detailed in IEEE Std 1003.1b-1993 ("POSIX.1b") as a read-style operation.

If *pid* is zero, the scheduling parameters for the calling process will be returned. In this implementation, the *sched_getscheduler* system call will fail if *pid* is negative.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

On failure *errno* will be set to the corresponding value:

[ENOSYS]	The system is not configured to support this functionality.
[EPERM]	The requesting process doesn not have permission as detailed in IEEE Std 1003.1b-1993 ("POSIX.1b").
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[EINVAL]	The value of the *policy* argument is invalid, or one or more of the parameters contained in *param* is outside the valid range for the specified scheduling policy.

SEE ALSO

sched_get_priority_max(2), sched_get_priority_min(2), sched_getparam(2), sched_rr_get_interval(2), sched_setparam(2), sched_yield(2)

STANDARDS

The **sched_setscheduler**() and **sched_getscheduler**() system calls conform to IEEE Std 1003.1b-1993 ("POSIX.1b").

NAME

sched_yield - yield processor

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sched.h>

int
sched_yield(*void*);

DESCRIPTION

The **sched_yield**() system call forces the running process to relinquish the processor until it again becomes the head of its process list. It takes no arguments.

RETURN VALUES

The **sched_yield**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

On failure *errno* will be set to the corresponding value:

[ENOSYS] The system is not configured to support this functionality.

STANDARDS

The **sched_yield**() system call conforms to IEEE Std 1003.1b-1993 ("POSIX.1b").

NAME

sctp_bindx - bind or unbind an SCTP socket to a list of addresses

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_bindx(*int s*, *struct sockaddr *addrs*, *int num*, *int type*);

DESCRIPTION

The **sctp_bindx**() call binds or unbinds a address or a list of addresses to an SCTP endpoint. This allows a user to bind a subset of addresses. The **sctp_bindx**() call operates similarly to **bind**() but allows a list of addresses and also allows a bind or an unbind. The argument *s* must be a valid SCTP socket descriptor. The argument *addrs* is a list of addresses (where the list may be only 1 in length) that the user wishes to bind or unbind to the socket. The argument *type* must be one of the following values.

SCTP_BINDX_ADD_ADDR This value indicates that the listed address(es) need to be added to the endpoint.

SCTP_BINDX_REM_ADDR This value indicates that the listed address(es) need to be removed from the endpoint.

Note that when a user adds or deletes an address to an association if the dynamic address flag *net.inet.sctp.auto_asconf* is enabled any associations in the endpoint will attempt to have the address(es) added dynamically to the existing association.

RETURN VALUES

The call returns 0 on success and -1 upon failure.

ERRORS

The **sctp_bindx**() function can return the following errors:

[EINVAL] This value is returned if the *type* field is not one of the allowed values (see above).

[ENOMEM] This value is returned if the number of addresses being added causes a memory allocation failure in the call.

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

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

bind(2), sctp(4)

NAME

sctp_connectx - connect an SCTP socket with multiple destination addresses

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_connectx(*int sd*, *struct sockaddr *addrs*, *int addrcnt*, *sctp_assoc_t *id*);

DESCRIPTION

The **sctp_connectx**() call attempts to initiate an association to a peer SCTP endpoint. The call operates similarly to **connect**() but it also provides the ability to specify multiple destination addresses for the peer. This allows a fault tolerant method of initiating an association. When one of the peers addresses is unreachable, the subsequent listed addresses will also be used to set up the association with the peer.

The user also needs to consider that any address listed in an **sctp_connectx**() call is also considered "confirmed". A confirmed address is one in which the SCTP transport will trust is a part of the association and it will not send a confirmation heartbeat to it with a random nonce.

If the peer SCTP stack does not list one or more of the provided addresses in its response message then the extra addresses sent in the **sctp_connectx**() call will be silently discarded from the association. On successful completion the provided *id* will be filled in with the association identification of the newly forming association.

RETURN VALUES

The call returns 0 on success and -1 upon failure.

ERRORS

The **sctp_connectx**() function can return the following errors:

[EINVAL]	An address listed has an invalid family or no addresses were provided.
[E2BIG]	The size of the address list exceeds the amount of data provided.
[EBADF]	The argument *s* is not a valid descriptor.

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

connect(2), sctp(4)

NAME

sctp_freepaddrs, **sctp_freeladdrs** - release the memory returned from a previous call

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

void
sctp_freepaddrs(*struct sockaddr **);

void
sctp_freeladdrs(*struct sockaddr **);

DESCRIPTION

The **sctp_freepaddrs**() and **sctp_freeladdrs**() functions are used to release the memory allocated by previous calls to **sctp_getpaddrs**() or **sctp_getladdrs**() respectively.

RETURN VALUES

none.

SEE ALSO

sctp_getladdrs(3), sctp_getpaddrs(3), sctp(4)

NAME

sctp_generic_recvmsg - receive data from a peer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_generic_recvmsg(*int s*, *struct iovec *iov*, *int iovlen*, *struct sockaddr *from*, *socklen_t *fromlen*, *struct sctp_sndrcvinfo *sinfo*, *int *msgflags*);

DESCRIPTION

sctp_generic_recvmsg() is the true system call used by the sctp_recvmsg(3) function call. This call is more efficient since it is a true system call but it is specific to FreeBSD and can be expected *not* to be present on any other operating system. For detailed usage please see the sctp_recvmsg(3) function call.

RETURN VALUES

The call returns the number of bytes read on success and -1 upon failure.

ERRORS

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

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

sctp_recvmsg(3), sctp(4)

NAME

sctp_generic_sendmsg sctp_generic_sendmsg_iov - send data to a peer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_generic_sendmsg(*int s*, *void *msg*, *int msglen*, *struct sockaddr *to*, *socklen_t len*,
*struct sctp_sndrcvinfo *sinfo*, *int flags*);

int
sctp_generic_sendmsg_iov(*int s*, *struct iovec *iov*, *int iovlen*, *struct sockaddr *to*,
*struct sctp_sndrcvinfo *sinfo*, *int flags*);

DESCRIPTION

sctp_generic_sendmsg() and **sctp_generic_sendmsg_iov**() are the true system calls used by the sctp_sendmsg(3) and sctp_send(3) function calls. These are more efficient since they are true system calls but they are specific to FreeBSD and can be expected *not* to be present on any other operating system. For detailed usage please see either the sctp_send(3) or sctp_sendmsg(3) function calls.

RETURN VALUES

The call returns the number of bytes written on success and -1 upon failure.

ERRORS

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

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

sctp_send(3), sctp_sendmsg(3), sctp_sendmsgx(3), sctp_sendx(3), sctp(4)

NAME

sctp_getaddrlen - return the address length of an address family

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_getaddrlen(*sa_family_t family*);

DESCRIPTION

The **sctp_getaddrlen**() function returns the size of a specific address family. This function is provided for application binary compatibility since it provides the application with the size the operating system thinks the specific address family is. Note that the function will actually create an SCTP socket and then gather the information via a **getsockopt**() system calls. If for some reason a SCTP socket cannot be created or the **getsockopt**() call fails, an error will be returned with *errno* set as specified in the **socket**() or **getsockopt**() system call.

RETURN VALUES

The call returns the number of bytes that the operating system expects for the specific address family or -1.

ERRORS

The **sctp_getaddrlen**() function can return the following errors:

[EINVAL] The address family specified does NOT exist.

SEE ALSO

getsockopt(2), socket(2), sctp(4)

NAME

sctp_getassocid - return an association id for a specified socket address

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

sctp_assoc_t
sctp_getassocid(*int s*, *struct sockaddr *addr*);

DESCRIPTION

The **sctp_getassocid**() call attempts to look up the specified socket address *addr* and find the respective association identification.

RETURN VALUES

The call returns the association id upon success and 0 is returned upon failure.

ERRORS

The **sctp_getassocid**() function can return the following errors:

[ENOENT] The address does not have an association setup to it.

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

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

sctp(4)

NAME

sctp_getpaddrs, **sctp_getladdrs** - return a list of addresses to the caller

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_getpaddrs(*int s*, *sctp_assoc_t asocid*, *struct sockaddr **addrs*);

int
sctp_getladdrs(*int s*, *sctp_assoc_t asocid*, *struct sockaddr **addrs*);

DESCRIPTION

The **sctp_getpaddrs**() function is used to get the list of the peers addresses. The **sctp_getladdrs**() function is used to get the list of the local addresses. The association of interest is identified by the *asocid* argument. The addresses are returned in a newly allocated array of socket addresses returned in the argument *addrs* upon success.

After the caller is finished, the function **sctp_freepaddrs**() or **sctp_freeladdrs**() should be used to release the memory allocated by these calls.

RETURN VALUES

The call returns -1 upon failure and a count of the number of addresses returned in *addrs* upon success.

ERRORS

The functions can return the following errors:

[EINVAL] An address listed has an invalid family or no addresses were provided.

[ENOMEM] The call cannot allocate memory to hold the socket addresses.

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

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

getsockopt(2), sctp_freeladdrs(3), sctp_freepaddrs(3), sctp(4)

NAME

sctp_opt_info - get SCTP socket information

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_opt_info(*int sd*, *sctp_assoc_t id*, *int opt*, *void *arg*, *socklen_t *size*);

DESCRIPTION

The **sctp_opt_info**() call provides a multi-os compatible method for getting specific **getsockopt**() data where an association identification needs to be passed into the operating system. For FreeBSD a direct **getsockopt**() may be used, since FreeBSD has the ability to pass information into the operating system on a **getsockopt**() call. Other operating systems may not have this ability. For those who wish to write portable code amongst multiple operating systems this call should be used for the following SCTP socket options.

SCTP_RTOINFO

SCTP_ASSOCINFO

SCTP_PRIMARY_ADDR

SCTP_PEER_ADDR_PARAMS

SCTP_DEFAULT_SEND_PARAM

SCTP_MAX_SEG

SCTP_AUTH_ACTIVE_KEY

SCTP_DELAYED_SACK

SCTP_MAX_BURST

SCTP_CONTEXT

SCTP_EVENT

SCTP_DEFAULT_SNDINFO

SCTP_DEFAULT_PRINFO

SCTP_STATUS

SCTP_GET_PEER_ADDR_INFO

SCTP_PEER_AUTH_CHUNKS

SCTP_LOCAL_AUTH_CHUNKS

RETURN VALUES

The call returns 0 on success and -1 upon error.

ERRORS

The **sctp_opt_info**() function can return the following errors:

[EINVAL] The argument *arg* value was invalid.

[EOPNOTSUPP] The argument *opt* was not one of the above listed SCTP socket options.

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

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

getsockopt(2), sctp(4)

BUGS

Because the structure used for *arg* of the SCTP_MAX_BURST socket option has changed in FreeBSD 9.0 and higher, using SCTP_MAX_BURST as *opt* is only supported in FreeBSD 9.0 and higher.

NAME

sctp_peeloff - detach an association from a one-to-many socket to its own fd

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

int
sctp_peeloff(*int s*, *sctp_assoc_t id*);

DESCRIPTION

The **sctp_peeloff**() system call attempts detach the association specified by *id* into its own separate socket.

RETURN VALUES

The call returns -1 on failure and the new socket descriptor upon success.

ERRORS

The **sctp_peeloff**() system call can return the following errors:

[ENOTCONN] The *id* given to the call does not map to a valid association.

[E2BIG] The size of the address list exceeds the amount of data provided.

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

[ENOTSOCK] The argument *s* is not a socket.

SEE ALSO

sctp(4)

NAME

sctp_recvmsg - receive a message from an SCTP socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t
sctp_recvmsg(*int s*, *void *msg*, *size_t len*, *struct sockaddr * restrict from*, *socklen_t * restrict fromlen*, *struct sctp_sndrcvinfo *sinfo*, *int *flags*);

DESCRIPTION

The **sctp_recvmsg**() system call is used to receive a message from another SCTP endpoint. The **sctp_recvmsg**() call is used by one-to-one (SOCK_STREAM) type sockets after a successful **connect**() call or after the application has performed a **listen**() followed by a successful **accept**(). For a one-to-many (SOCK_SEQPACKET) type socket, an endpoint may call **sctp_recvmsg**() after having implicitly started an association via one of the send calls including **sctp_sendmsg**(), **sendto**() and **sendmsg**(). Or, an application may also receive a message after having called **listen**() with a positive backlog to enable the reception of new associations.

The address of the sender is held in the *from* argument with *fromlen* specifying its size. At the completion of a successful **sctp_recvmsg**() call *from* will hold the address of the peer and *fromlen* will hold the length of that address. Note that the address is bounded by the initial value of *fromlen* which is used as an in/out variable.

The length of the message *msg* to be received is bounded by *len*. If the message is too long to fit in the users receive buffer, then the *flags* argument will *not* have the MSG_EOR flag applied. If the message is a complete message then the *flags* argument will have MSG_EOR set. Locally detected errors are indicated by a return value of -1 with *errno* set accordingly. The *flags* argument may also hold the value MSG_NOTIFICATION. When this occurs it indicates that the message received is *not* from the peer endpoint, but instead is a notification from the SCTP stack (see sctp(4) for more details). Note that no notifications are ever given unless the user subscribes to such notifications using the SCTP_EVENTS socket option.

If no messages are available at the socket then **sctp_recvmsg**() normally blocks on the reception of a message or NOTIFICATION, unless the socket has been placed in non-blocking I/O mode. The

select(2) system call may be used to determine when it is possible to receive a message.

The *sinfo* argument is defined as follows.

```
struct sctp_sndrcvinfo {
        uint16_t sinfo_stream;  /* Stream arriving on */
        uint16_t sinfo_ssn;     /* Stream Sequence Number */
        uint16_t sinfo_flags;   /* Flags on the incoming message */
        uint32_t sinfo_ppid;    /* The ppid field */
        uint32_t sinfo_context; /* context field */
        uint32_t sinfo_timetolive; /* not used by sctp_recvmsg */
        uint32_t sinfo_tsn;       /* The transport sequence number */
        uint32_t sinfo_cumtsn;    /* The cumulative acknowledgment point  */
        sctp_assoc_t sinfo_assoc_id; /* The association id of the peer */
};
```

The *sinfo->sinfo_ppid* field is an opaque 32 bit value that is passed transparently through the stack from the peer endpoint. Note that the stack passes this value without regard to byte order.

The *sinfo->sinfo_flags* field may include the following:

```
#define SCTP_UNORDERED              0x0400  /* Message is un-ordered */
```

The SCTP_UNORDERED flag is used to specify that the message arrived with no specific order and was delivered to the peer application as soon as possible. When this flag is absent the message was delivered in order within the stream it was received.

The *sinfo->sinfo_stream* field is the SCTP stream that the message was received on. Streams in SCTP are reliable (or partially reliable) flows of ordered messages.

The *sinfo->sinfo_context* field is used only if the local application set an association level context with the SCTP_CONTEXT socket option. Optionally a user process can use this value to index some application specific data structure for all data coming from a specific association.

The *sinfo->sinfo_ssn* field will hold the stream sequence number assigned by the peer endpoint if the message is *not* unordered. For unordered messages this field holds an undefined value.

The *sinfo->sinfo_tsn* field holds a transport sequence number (TSN) that was assigned to this message by the peer endpoint. For messages that fit in or less than the path MTU this will be the only TSN assigned. Note that for messages that span multiple TSNs this value will be one of the TSNs that was

used on the message.

The *sinfo->sinfo_cumtsn* field holds the current cumulative acknowledgment point of the transport association. Note that this may be larger or smaller than the TSN assigned to the message itself.

The *sinfo->sinfo_assoc_id* is the unique association identification that was assigned to the association. For one-to-many (SOCK_SEQPACKET) type sockets this value can be used to send data to the peer without the use of an address field. It is also quite useful in setting various socket options on the specific association (see sctp(4)).

The *sinfo->info_timetolive* field is not used by **sctp_recvmsg**().

RETURN VALUES

The call returns the number of bytes received, or -1 if an error occurred.

ERRORS

The **sctp_recvmsg**() system call fails if:

[EBADF] An invalid descriptor was specified.

[ENOTSOCK] The argument *s* is not a socket.

[EFAULT] An invalid user space address was specified for an argument.

[EMSGSIZE] The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EAGAIN] The socket is marked non-blocking and the requested operation would block.

[ENOBUFS] The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[EHOSTUNREACH] The remote host was unreachable.

[ENOTCONN] On a one-to-one style socket no association exists.

[ECONNRESET] An abort was received by the stack while the user was attempting to send data to the peer.

[ENOENT] On a one to many style socket no address is specified so that the association cannot be located or the SCTP_ABORT flag was specified on a non-existing association.

[EPIPE] The socket is unable to send anymore data (SBS_CANTSENDMORE has been set on the socket). This typically means that the socket is not connected and is a one-to-one style socket.

SEE ALSO

getsockopt(2), recv(2), select(2), sendmsg(2), setsockopt(2), socket(2), write(2), sctp_send(3), sctp_sendmsg(3), sctp(4)

NAME

sctp_send, **sctp_sendx** - send a message from an SCTP socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t
sctp_send(*int sd*, *const void *msg*, *size_t len*, *const struct sctp_sndrcvinfo *sinfo*, *int flags*);

ssize_t
sctp_sendx(*int sd*, *const void *msg*, *size_t len*, *struct sockaddr *addrs*, *int addrcnt*,
*const struct sctp_sndrcvinfo *sinfo*, *int flags*);

DESCRIPTION

The **sctp_send**() system call is used to transmit a message to another SCTP endpoint. **sctp_send**() may be used to send data to an existing association for both one-to-many (SOCK_SEQPACKET) and one-to-one (SOCK_STREAM) socket types. The length of the message *msg* is given by *len*. If the message is too long to pass atomically through the underlying protocol, *errno* is set to EMSGSIZE, -1 is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a **sctp_send**(). Locally detected errors are indicated by a return value of -1.

If no space is available at the socket to hold the message to be transmitted, then **sctp_send**() normally blocks, unless the socket has been placed in non-blocking I/O mode. The select(2) system call may be used to determine when it is possible to send more data on one-to-one type (SOCK_STREAM) sockets.

The *sinfo* structure is used to control various SCTP features and has the following format:

```
struct sctp_sndrcvinfo {
        uint16_t sinfo_stream;  /* Stream sending to */
        uint16_t sinfo_ssn;     /* valid for recv only */
        uint16_t sinfo_flags;   /* flags to control sending */
        uint32_t sinfo_ppid;    /* ppid field */
        uint32_t sinfo_context; /* context field */
```

```
        uint32_t sinfo_timetolive; /* timetolive for PR-SCTP */
        uint32_t sinfo_tsn;       /* valid for recv only */
        uint32_t sinfo_cumtsn;    /* valid for recv only */
        sctp_assoc_t sinfo_assoc_id; /* The association id */
};
```

The *sinfo->sinfo_ppid* argument is an opaque 32 bit value that is passed transparently through the stack to the peer endpoint. It will be available on reception of a message (see sctp_recvmsg(3)). Note that the stack passes this value without regard to byte order.

The *sinfo->sinfo_flags* argument may include one or more of the following:

```
#define SCTP_EOF              0x0100  /* Start a shutdown procedures */
#define SCTP_ABORT            0x0200  /* Send an ABORT to peer */
#define SCTP_UNORDERED                0x0400  /* Message is un-ordered */
#define SCTP_ADDR_OVER                0x0800  /* Override the primary-address */
#define SCTP_SENDALL      0x1000   /* Send this on all associations */
                                              /* for the endpoint */
/* The lower byte is an enumeration of PR-SCTP policies */
#define SCTP_PR_SCTP_TTL  0x0001            /* Time based PR-SCTP */
#define SCTP_PR_SCTP_BUF  0x0002            /* Buffer based PR-SCTP */
#define SCTP_PR_SCTP_RTX  0x0003            /* Number of retransmissions based PR-SCTP */
```

The flag SCTP_EOF is used to instruct the SCTP stack to queue this message and then start a graceful shutdown of the association. All remaining data in queue will be sent after which the association will be shut down.

SCTP_ABORT is used to immediately terminate an association. An abort is sent to the peer and the local TCB is destroyed.

SCTP_UNORDERED is used to specify that the message being sent has no specific order and should be delivered to the peer application as soon as possible. When this flag is absent messages are delivered in order within the stream they are sent, but without respect to order to peer streams.

The flag SCTP_ADDR_OVER is used to specify that a specific address should be used. Normally SCTP will use only one of a multi-homed peers addresses as the primary address to send to. By default, no matter what the *to* argument is, this primary address is used to send data. By specifying this flag, the user is asking the stack to ignore the primary address and instead use the specified address not only as a lookup mechanism to find the association but also as the actual address to send to.

For a one-to-many type (SOCK_SEQPACKET) socket the flag SCTP_SENDALL can be used as a convenient way to make one send call and have all associations that are under the socket get a copy of the message. Note that this mechanism is quite efficient and makes only one actual copy of the data which is shared by all the associations for sending.

The remaining flags are used for the partial reliability extension (RFC3758) and will only be effective if the peer endpoint supports this extension. This option specifies what local policy the local endpoint should use in skipping data. If none of these options are set, then data is never skipped over.

SCTP_PR_SCTP_TTL is used to indicate that a time based lifetime is being applied to the data. The *sinfo->sinfo_timetolive* argument is then a number of milliseconds for which the data is attempted to be transmitted. If that many milliseconds elapse and the peer has not acknowledged the data, the data will be skipped and no longer transmitted. Note that this policy does not even assure that the data will ever be sent. In times of a congestion with large amounts of data being queued, the *sinfo->sinfo_timetolive* may expire before the first transmission is ever made.

The SCTP_PR_SCTP_BUF based policy transforms the *sinfo->sinfo_timetolive* field into a total number of bytes allowed on the outbound send queue. If that number or more bytes are in queue, then other buffer-based sends are looked to be removed and skipped. Note that this policy may also result in the data never being sent if no buffer based sends are in queue and the maximum specified by *timetolive* bytes is in queue.

The SCTP_PR_SCTP_RTX policy transforms the *sinfo->sinfo_timetolive* into a number of retransmissions to allow. This policy always assures that at a minimum one send attempt is made of the data. After which no more than *sinfo->sinfo_timetolive* retransmissions will be made before the data is skipped.

sinfo->sinfo_stream is the SCTP stream that you wish to send the message on. Streams in SCTP are reliable (or partially reliable) flows of ordered messages.

The *sinfo->sinfo_assoc_id* field is used to select the association to send to on a one-to-many socket. For a one-to-one socket, this field is ignored.

The *sinfo->sinfo_context* field is used only in the event the message cannot be sent. This is an opaque value that the stack retains and will give to the user when a failed send is given if that notification is enabled (see sctp(4)). Normally a user process can use this value to index some application specific data structure when a send cannot be fulfilled.

The *flags* argument holds the same meaning and values as those found in sendmsg(2) but is generally ignored by SCTP.

The fields *sinfo->sinfo_ssn*, *sinfo->sinfo_tsn*, and *sinfo->sinfo_cumtsn* are used only when receiving messages and are thus ignored by **sctp_send**(). The function **sctp_sendx**() has the same properties as **sctp_send**() with the additional arguments of an array of sockaddr structures passed in. With the *addrs* argument being given as an array of addresses to be sent to and the *addrcnt* argument indicating how many socket addresses are in the passed in array. Note that all of the addresses will only be used when an implicit association is being set up. This allows the user the equivalent behavior as doing a **sctp_connectx**() followed by a **sctp_send**() to the association. Note that if the *sinfo->sinfo_assoc_id* field is 0, then the first address will be used to look up the association in place of the association id. If both an address and an association id are specified, the association id has priority.

RETURN VALUES

The call returns the number of characters sent, or -1 if an error occurred.

ERRORS

The **sctp_send**() system call fails if:

[EBADF] An invalid descriptor was specified.

[ENOTSOCK] The argument *s* is not a socket.

[EFAULT] An invalid user space address was specified for an argument.

[EMSGSIZE] The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EAGAIN] The socket is marked non-blocking and the requested operation would block.

[ENOBUFS] The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[EHOSTUNREACH] The remote host was unreachable.

[ENOTCONN] On a one-to-one style socket no association exists.

[ECONNRESET] An abort was received by the stack while the user was attempting to send data to the peer.

[ENOENT] On a one-to-many style socket no address is specified so that the association cannot be located or the SCTP_ABORT flag was specified on a non-existing association.

[EPIPE] The socket is unable to send anymore data (SBS_CANTSENDMORE has been set on the socket). This typically means that the socket is not connected and is a one-to-one style socket.

SEE ALSO

getsockopt(2), recv(2), select(2), sendmsg(2), socket(2), write(2), sctp_connectx(3), sctp_recvmsg(3), sctp_sendmsg(3), sctp(4)

BUGS

Because **sctp_send**() may have multiple associations under one endpoint, a select on write will only work for a one-to-one style socket.

NAME

sctp_sendmsg, **sctp_sendmsgx** - send a message from an SCTP socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/sctp.h>

ssize_t
sctp_sendmsg(*int s*, *const void *msg*, *size_t len*, *const struct sockaddr *to*, *socklen_t tolen*, *uint32_t ppid*, *uint32_t flags*, *uint16_t stream_no*, *uint32_t timetolive*, *uint32_t context*);

ssize_t
sctp_sendmsgx(*int s*, *const void *msg*, *size_t len*, *const struct sockaddr *to*, *int addrcnt*, *uint32_t ppid*, *uint32_t flags*, *uint16_t stream_no*, *uint32_t timetolive*, *uint32_t context*);

DESCRIPTION

The **sctp_sendmsg**() system call is used to transmit a message to another SCTP endpoint. The **sctp_sendmsg**() may be used at any time. If the socket is a one-to-many type (SOCK_SEQPACKET) socket then an attempt to send to an address that no association exists to will implicitly create a new association. Data sent in such an instance will result in the data being sent on the third leg of the SCTP four-way handshake. Note that if the socket is a one-to-one type (SOCK_STREAM) socket then an association must be in existence (by use of the connect(2) system call). Calling **sctp_sendmsg**() or **sctp_sendmsgx**() on a non-connected one-to-one socket will result in *errno* being set to ENOTCONN, -1 being returned, and the message not being transmitted.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message *msg* is given by *len*. If the message is too long to pass atomically through the underlying protocol, *errno* is set to EMSGSIZE, -1 is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a sctp_sendmsg(3) call. Locally detected errors are indicated by a return value of -1.

If no space is available at the socket to hold the message to be transmitted, then sctp_sendmsg(3) normally blocks, unless the socket has been placed in non-blocking I/O mode. The select(2) system call may be used to determine when it is possible to send more data on one-to-one type (SOCK_STREAM) sockets.

The *ppid* argument is an opaque 32 bit value that is passed transparently through the stack to the peer endpoint. It will be available on reception of a message (see sctp_recvmsg(3)). Note that the stack passes this value without regard to byte order.

The *flags* argument may include one or more of the following:

```
#define SCTP_EOF              0x0100  /* Start a shutdown procedures */
#define SCTP_ABORT            0x0200  /* Send an ABORT to peer */
#define SCTP_UNORDERED                0x0400  /* Message is un-ordered */
#define SCTP_ADDR_OVER                0x0800  /* Override the primary-address */
#define SCTP_SENDALL      0x1000   /* Send this on all associations */
                                              /* for the endpoint */
/* The lower byte is an enumeration of PR-SCTP policies */
#define SCTP_PR_SCTP_TTL  0x0001              /* Time based PR-SCTP */
#define SCTP_PR_SCTP_BUF  0x0002              /* Buffer based PR-SCTP */
#define SCTP_PR_SCTP_RTX  0x0003              /* Number of retransmissions based PR-SCTP */
```

The flag SCTP_EOF is used to instruct the SCTP stack to queue this message and then start a graceful shutdown of the association. All remaining data in queue will be sent after which the association will be shut down.

SCTP_ABORT is used to immediately terminate an association. An abort is sent to the peer and the local TCB is destroyed.

SCTP_UNORDERED is used to specify that the message being sent has no specific order and should be delivered to the peer application as soon as possible. When this flag is absent messages are delivered in order within the stream they are sent, but without respect to order to peer streams.

The flag SCTP_ADDR_OVER is used to specify that an specific address should be used. Normally SCTP will use only one of a multi-homed peers addresses as the primary address to send to. By default, no matter what the *to* argument is, this primary address is used to send data. By specifying this flag, the user is asking the stack to ignore the primary address and instead use the specified address not only as a lookup mechanism to find the association but also as the actual address to send to.

For a one-to-many type (SOCK_SEQPACKET) socket the flag SCTP_SENDALL can be used as a convenient way to make one send call and have all associations that are under the socket get a copy of the message. Note that this mechanism is quite efficient and makes only one actual copy of the data which is shared by all the associations for sending.

The remaining flags are used for the partial reliability extension (RFC3758) and will only be effective if

the peer endpoint supports this extension. This option specifies what local policy the local endpoint should use in skipping data. If none of these options are set, then data is never skipped over.

SCTP_PR_SCTP_TTL is used to indicate that a time based lifetime is being applied to the data. The *timetolive* argument is then a number of milliseconds for which the data is attempted to be transmitted. If that many milliseconds elapse and the peer has not acknowledged the data, the data will be skipped and no longer transmitted. Note that this policy does not even assure that the data will ever be sent. In times of a congestion with large amounts of data being queued, the *timetolive* may expire before the first transmission is ever made.

The SCTP_PR_SCTP_BUF based policy transforms the *timetolive* field into a total number of bytes allowed on the outbound send queue. If that number or more bytes are in queue, then other buffer based sends are looked to be removed and skipped. Note that this policy may also result in the data never being sent if no buffer based sends are in queue and the maximum specified by *timetolive* bytes is in queue.

The SCTP_PR_SCTP_RTX policy transforms the *timetolive* into a number of retransmissions to allow. This policy always assures that at a minimum one send attempt is made of the data. After which no more than *timetolive* retransmissions will be made before the data is skipped.

stream_no is the SCTP stream that you wish to send the message on. Streams in SCTP are reliable (or partially reliable) flows of ordered messages. The *context* field is used only in the event the message cannot be sent. This is an opaque value that the stack retains and will give to the user when a failed send is given if that notification is enabled (see sctp(4)). Normally a user process can use this value to index some application specific data structure when a send cannot be fulfilled. **sctp_sendmsgx**() is identical to **sctp_sendmsg**() with the exception that it takes an array of sockaddr structures in the argument *to* and adds the additional argument *addrcnt* which specifies how many addresses are in the array. This allows a caller to implicitly set up an association passing multiple addresses as if **sctp_connectx**() had been called to set up the association.

RETURN VALUES

The call returns the number of characters sent, or -1 if an error occurred.

ERRORS

The **sctp_sendmsg**() system call fails if:

[EBADF] An invalid descriptor was specified.

[ENOTSOCK] The argument *s* is not a socket.

[EFAULT] An invalid user space address was specified for an argument.

[EMSGSIZE] The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EAGAIN] The socket is marked non-blocking and the requested operation would block.

[ENOBUFS] The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[EHOSTUNREACH]
The remote host was unreachable.

[ENOTCONN] On a one-to-one style socket no association exists.

[ECONNRESET] An abort was received by the stack while the user was attempting to send data to the peer.

[ENOENT] On a one-to-many style socket no address is specified so that the association cannot be located or the SCTP_ABORT flag was specified on a non-existing association.

[EPIPE] The socket is unable to send anymore data (SBS_CANTSENDMORE has been set on the socket). This typically means that the socket is not connected and is a one-to-one style socket.

SEE ALSO

connect(2), getsockopt(2), recv(2), select(2), sendmsg(2), socket(2), write(2), sctp_connectx(3), sctp(4)

BUGS

Because in the one-to-many style socket **sctp_sendmsg**() or **sctp_sendmsgx**() may have multiple associations under one endpoint, a select on write will only work for a one-to-one style socket.

NAME

select - synchronous I/O multiplexing

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/select.h>

int
select(*int nfds*, *fd_set *readfds*, *fd_set *writefds*, *fd_set *exceptfds*, *struct timeval *timeout*);

FD_SET(*fd*, *&fdset*);

FD_CLR(*fd*, *&fdset*);

FD_ISSET(*fd*, *&fdset*);

FD_ZERO(*&fdset*);

DESCRIPTION

The **select**() system call examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The only exceptional condition detectable is out-of-band data received on a socket. The first *nfds* descriptors are checked in each set; i.e., the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, **select**() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The **select**() system call returns the total number of ready descriptors in all the sets.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: **FD_ZERO**(*&fdset*) initializes a descriptor set *fdset* to the null set. **FD_SET**(*fd*, *&fdset*) includes a particular descriptor *fd* in *fdset*. **FD_CLR**(*fd*, *&fdset*) removes *fd* from *fdset*. **FD_ISSET**(*fd*, *&fdset*) is non-zero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is not a null pointer, it specifies the maximum interval to wait for the selection to complete. System activity can lengthen the interval by an indeterminate amount.

If *timeout* is a null pointer, the select blocks indefinitely.

To effect a poll, the *timeout* argument should not be a null pointer, but it should point to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as null pointers if no descriptors are of interest.

RETURN VALUES

The **select**() system call returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires, **select**() returns 0. If **select**() returns with an error, including one due to an interrupted system call, the descriptor sets will be unmodified.

ERRORS

An error return from **select**() indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EFAULT]	One of the arguments *readfds*, *writefds*, *exceptfds*, or *timeout* points to an invalid address.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.
[EINVAL]	The *nfds* argument was invalid.

SEE ALSO

accept(2), connect(2), getdtablesize(2), gettimeofday(2), kqueue(2), poll(2), read(2), recv(2), send(2), write(2), clocks(7)

NOTES

The default size of FD_SETSIZE is currently 1024. In order to accommodate programs which might potentially use a larger number of open files with **select**(), it is possible to increase this size by having the program define FD_SETSIZE before the inclusion of any header which includes <*sys/types.h*>.

If *nfds* is greater than the number of open files, **select**() is not guaranteed to examine the unused file descriptors. For historical reasons, **select**() will always examine the first 256 descriptors.

STANDARDS

The **select**() system call and **FD_CLR**(), **FD_ISSET**(), **FD_SET**(), and **FD_ZERO**() macros conform with IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **select**() system call appeared in 4.2BSD.

BUGS

Version 2 of the Single UNIX Specification ("SUSv2") allows systems to modify the original timeout in place. Thus, it is unwise to assume that the timeout value will be unmodified by the **select**() system call. FreeBSD does not modify the return value, which can cause problems for applications ported from other systems.

NAME

sem_timedwait, **sem_clockwait_np** - lock a semaphore

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

int
sem_timedwait(sem_t * restrict sem, const struct timespec * restrict abs_timeout);

int
sem_clockwait_np(sem_t * restrict sem, clockid_t clock_id, int flags, const struct timespec * rqtp,
    struct timespec * rmtp);
```

DESCRIPTION

The **sem_timedwait**() function locks the semaphore referenced by *sem*, as in the sem_wait(3) function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a sem_post(3) function, this wait will be terminated when the specified timeout expires.

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

Note that the timeout is based on the CLOCK_REALTIME clock.

The validity of the *abs_timeout* is not checked if the semaphore can be locked immediately.

The **sem_clockwait_np**() function is a more flexible variant of **sem_timedwait**(). The *clock_id* parameter specifies the reference clock. If the *flags* parameter contains TIMER_ABSTIME, then the requested timeout (*rqtp*) is an absolute timeout; otherwise, the timeout is relative. If this function fails with EINTR and the timeout is relative, a non-NULL *rmtp* will be updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). An absolute timeout has no effect on *rmtp*. A single structure can be used for both *rqtp* and *rmtp*.

RETURN VALUES

These functions return zero if the calling process successfully performed the semaphore lock operation

on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore is unchanged, and the function returns a value of -1 and sets the global variable *errno* to indicate the error.

ERRORS

These functions will fail if:

[EINVAL] The *sem* argument does not refer to a valid semaphore, or the process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1000 million.

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

[EINTR] A signal interrupted this function.

SEE ALSO

sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_timedwait**() function conforms to IEEE Std 1003.1-2004 ("POSIX.1"). The **sem_clockwait_np**() function is not specified by any standard; it exists only on FreeBSD at the time of this writing.

HISTORY

The **sem_timedwait**() function first appeared in FreeBSD 5.0. The **sem_clockwait_np**() function first appeared in FreeBSD 11.1.

NAME

sem_open, **sem_close**, **sem_unlink** - named semaphore operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <semaphore.h>

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

int
sem_close(*sem_t *sem*);

int
sem_unlink(*const char *name*);

DESCRIPTION

The **sem_open**() function creates or opens the named semaphore specified by *name*. The returned semaphore may be used in subsequent calls to sem_getvalue(3), sem_wait(3), sem_trywait(3), sem_post(3), and **sem_close**().

This implementation places strict requirements on the value of *name*: it must begin with a slash (‘/’) and contain no other slash characters.

The following bits may be set in the *oflag* argument:

O_CREAT Create the semaphore if it does not already exist.

The third argument to the call to **sem_open**() must be of type *mode_t* and specifies the mode for the semaphore. Only the S_IWUSR, S_IWGRP, and S_IWOTH bits are examined; it is not possible to grant only "read" permission on a semaphore. The mode is modified according to the process’s file creation mask; see umask(2).

The fourth argument must be an *unsigned int* and specifies the initial value for the semaphore, and must be no greater than SEM_VALUE_MAX.

O_EXCL Create the semaphore if it does not exist. If the semaphore already exists, **sem_open**() will fail. This flag is ignored unless O_CREAT is also specified.

The **sem_close**() function closes a named semaphore that was opened by a call to **sem_open**().

The **sem_unlink**() function removes the semaphore named *name*. Resources allocated to the semaphore are only deallocated when all processes that have the semaphore open close it.

RETURN VALUES

If successful, the **sem_open**() function returns the address of the opened semaphore. If the same *name* argument is given to multiple calls to **sem_open**() by the same process without an intervening call to **sem_close**(), the same address is returned each time. If the semaphore cannot be opened, **sem_open**() returns SEM_FAILED and the global variable *errno* is set to indicate the error.

The **sem_close**() and **sem_unlink**() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sem_open**() function will fail if:

[EACCES] The semaphore exists and the permissions specified by *oflag* at the time it was created deny access to this process.

[EACCES] The semaphore does not exist, but permission to create it is denied.

[EEXIST] O_CREAT and O_EXCL are set but the semaphore already exists.

[EINTR] The call was interrupted by a signal.

[EINVAL] The **sem_open**() operation is not supported for the given *name*.

[EINVAL] The *value* argument is greater than SEM_VALUE_MAX.

[ENAMETOOLONG]
The *name* argument is too long.

[ENFILE] The system limit on semaphores has been reached.

[ENOENT] O_CREAT is not set but the named semaphore does not exist.

[ENOSPC] There is not enough space to create the semaphore.

The **sem_close**() function will fail if:

[EINVAL] The *sem* argument is not a valid semaphore.

The **sem_unlink**() function will fail if:

[EACCES] Permission is denied to unlink the semaphore.

[ENAMETOOLONG]
The specified *name* is too long.

[ENOENT] The named semaphore does not exist.

SEE ALSO

close(2), open(2), umask(2), unlink(2), sem_getvalue(3), sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_open**(), **sem_close**(), and **sem_unlink**() functions conform to ISO/IEC 9945-1:1996 ("POSIX.1").

HISTORY

Support for named semaphores first appeared in FreeBSD 5.0.

NAME

sem_destroy - destroy an unnamed semaphore

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <semaphore.h>

int
sem_destroy(*sem_t *sem*);

DESCRIPTION

The **sem_destroy**() function destroys the unnamed semaphore pointed to by *sem*. After a successful call to **sem_destroy**(), *sem* is unusable until re-initialized by another call to sem_init(3).

RETURN VALUES

The **sem_destroy**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sem_destroy**() function will fail if:

[EINVAL] The *sem* argument points to an invalid semaphore.

[EBUSY] There are currently threads blocked on the semaphore that *sem* points to.

SEE ALSO

sem_init(3)

STANDARDS

The **sem_destroy**() function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

POSIX does not define the behavior of **sem_destroy**() if called while there are threads blocked on *sem*, but this implementation is guaranteed to return -1 and set *errno* to EBUSY if there are threads blocked on *sem*.

NAME

sem_getvalue - get the value of a semaphore

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <semaphore.h>

int
sem_getvalue(*sem_t * restrict sem*, *int * restrict sval*);

DESCRIPTION

The **sem_getvalue**() function sets the variable pointed to by *sval* to the current value of the semaphore pointed to by *sem*, as of the time that the call to **sem_getvalue**() is actually run.

RETURN VALUES

The **sem_getvalue**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sem_getvalue**() function will fail if:

[EINVAL] The *sem* argument points to an invalid semaphore.

SEE ALSO

sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_getvalue**() function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

The value of the semaphore is never negative, even if there are threads blocked on the semaphore. POSIX is somewhat ambiguous in its wording with regard to what the value of the semaphore should be if there are blocked waiting threads, but this behavior is conformant, given the wording of the specification.

NAME

sem_init - initialize an unnamed semaphore

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <semaphore.h>

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

DESCRIPTION

The **sem_init**() function initializes the unnamed semaphore pointed to by *sem* to have the value *value*.

A non-zero value for *pshared* specifies a shared semaphore that can be used by multiple processes, the semaphore should be located in shared memory region (see mmap(2), shm_open(2), and shmget(2)), any process having read and write access to address *sem* can perform semaphore operations on *sem*.

Following a successful call to **sem_init**(), *sem* can be used as an argument in subsequent calls to sem_wait(3), sem_trywait(3), sem_post(3), and sem_destroy(3). The *sem* argument is no longer valid after a successful call to sem_destroy(3).

RETURN VALUES

The **sem_init**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sem_init**() function will fail if:

[EINVAL] The *value* argument exceeds SEM_VALUE_MAX.

[ENOSPC] Memory allocation error.

SEE ALSO

sem_destroy(3), sem_getvalue(3), sem_post(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_init**() function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

sem_post - increment (unlock) a semaphore

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <semaphore.h>

int
sem_post(*sem_t *sem*);

DESCRIPTION

The **sem_post**() function increments (unlocks) the semaphore pointed to by *sem*. If there are threads blocked on the semaphore when **sem_post**() is called, then the highest priority thread that has been blocked the longest on the semaphore will be allowed to return from **sem_wait**().

The **sem_post**() function is signal-reentrant and may be called within signal handlers.

RETURN VALUES

The **sem_post**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sem_post**() function will fail if:

[EINVAL] The *sem* argument points to an invalid semaphore.

[EOVERFLOW] The semaphore value would exceed SEM_VALUE_MAX.

SEE ALSO

sem_getvalue(3), sem_trywait(3), sem_wait(3)

STANDARDS

The **sem_post**() function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

sem_wait, **sem_trywait** - decrement (lock) a semaphore

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <semaphore.h>

int
sem_wait(*sem_t *sem*);

int
sem_trywait(*sem_t *sem*);

DESCRIPTION

The **sem_wait**() function decrements (locks) the semaphore pointed to by *sem*, but blocks if the value of *sem* is zero, until the value is non-zero and the value can be decremented.

The **sem_trywait**() function decrements (locks) the semaphore pointed to by *sem* only if the value is non-zero. Otherwise, the semaphore is not decremented and an error is returned.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sem_wait**() and **sem_trywait**() functions will fail if:

[EINVAL] The *sem* argument points to an invalid semaphore.

Additionally, **sem_wait**() will fail if:

[EINTR] A signal interrupted this function.

Additionally, **sem_trywait**() will fail if:

[EAGAIN] The semaphore value was zero, and thus could not be decremented.

SEE ALSO

sem_getvalue(3), sem_post(3), sem_timedwait(3)

STANDARDS

The **sem_wait**() and **sem_trywait**() functions conform to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

semctl - control operations on a semaphore set

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int
semctl(*int semid*, *int semnum*, *int cmd*, ...);

DESCRIPTION

The **semctl**() system call performs the operation indicated by *cmd* on the semaphore set indicated by *semid*. A fourth argument, a *union semun arg*, is required for certain values of *cmd*. For the commands that use the *arg* argument, *union semun* must be defined as follows:

```
union semun {
    int     val;          /* value for SETVAL */
    struct  semid_ds *buf;  /* buffer for IPC_STAT & IPC_SET */
    u_short *array;       /* array for GETALL & SETALL */
};
```

Non-portable software may define _WANT_SEMUN before including *sys/sem.h* to use the system definition of *union semun*.

Commands are performed as follows:

IPC_STAT — Fetch the semaphore set's *struct semid_ds*, storing it in the memory pointed to by *arg.buf*.

IPC_SET — Changes the *sem_perm.uid*, *sem_perm.gid*, and *sem_perm.mode* members of the semaphore set's *struct semid_ds* to match those of the struct pointed to by *arg.buf*. The calling process's effective uid must match either *sem_perm.uid* or *sem_perm.cuid*, or it must have superuser privileges.

IPC_RMID — Immediately removes the semaphore set from the system. The calling process's effective uid must equal the semaphore set's *sem_perm.uid* or *sem_perm.cuid*, or the process must have superuser privileges.

GETVAL Return the value of semaphore number *semnum*.

SETVAL Set the value of semaphore number *semnum* to *arg.val*. Outstanding adjust on exit values for this semaphore in any process are cleared.

GETPID Return the pid of the last process to perform an operation on semaphore number *semnum*.

GETNCNT Return the number of processes waiting for semaphore number *semnum*'s value to become greater than its current value.

GETZCNT Return the number of processes waiting for semaphore number *semnum*'s value to become 0.

GETALL Fetch the value of all of the semaphores in the set into the array pointed to by *arg.array*.

SETALL Set the values of all of the semaphores in the set to the values in the array pointed to by *arg.array*. Outstanding adjust on exit values for all semaphores in this set, in any process are cleared.

The *struct semid_ds* is defined as follows:

```
struct semid_ds {
        struct  ipc_perm sem_perm;      /* operation permission struct */
        struct  sem *__sem_base;  /* kernel data, don't use */
        u_short sem_nsems;      /* number of sems in set */
        time_t  sem_otime;      /* last operation time */
        time_t  sem_ctime;      /* last change time */
                        /* Times measured in secs since */
                        /* 00:00:00 GMT, Jan. 1, 1970 */
};
```

RETURN VALUES

On success, when *cmd* is one of GETVAL, GETPID, GETNCNT or GETZCNT, **semctl**() returns the corresponding value; otherwise, 0 is returned. On failure, -1 is returned, and *errno* is set to indicate the error.

ERRORS

The **semctl**() system call will fail if:

[EINVAL] No semaphore set corresponds to *semid*.

[EINVAL] The *semnum* argument is not in the range of valid semaphores for given semaphore set.

[EPERM] The calling process's effective uid does not match the uid of the semaphore set's owner or creator.

[EACCES] Permission denied due to mismatch between operation and mode of semaphore set.

[ERANGE] SETVAL or SETALL attempted to set a semaphore outside the allowable range [0 .. SEMVMX].

SEE ALSO

semget(2), semop(2)

BUGS

SETALL may update some semaphore elements before returning an error.

NAME

semget - obtain a semaphore id

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/sem.h>

int
semget(*key_t key*, *int nsems*, *int flag*);

DESCRIPTION

Based on the values of *key* and *flag*, **semget**() returns the identifier of a newly created or previously existing set of semaphores. The key is analogous to a filename: it provides a handle that names an IPC object. There are three ways to specify a key:

- IPC_PRIVATE may be specified, in which case a new IPC object will be created.
- An integer constant may be specified. If no IPC object corresponding to *key* is specified and the IPC_CREAT bit is set in *flag*, a new one will be created.
- The ftok(3) function may be used to generate a key from a pathname.

The mode of a newly created IPC object is determined by ORing these constants into the *flag* argument:

0400 Read access for user.

0200 Alter access for user.

0040 Read access for group.

0020 Alter access for group.

0004 Read access for other.

0002 Alter access for other.

If a new set of semaphores is being created, *nsems* is used to indicate the number of semaphores the set should contain. Otherwise, *nsems* may be specified as 0.

RETURN VALUES

The **semget**() system call returns the id of a semaphore set if successful; otherwise, -1 is returned and *errno* is set to indicate the error.

ERRORS

The **semget**() system call will fail if:

[EACCES]	Access permission failure.
[EEXIST]	IPC_CREAT and IPC_EXCL were specified, and a semaphore set corresponding to *key* already exists.
[EINVAL]	The number of semaphores requested exceeds the system imposed maximum per set.
[EINVAL]	A semaphore set corresponding to *key* already exists and contains fewer semaphores than *nsems*.
[EINVAL]	A semaphore set corresponding to *key* does not exist and *nsems* is 0 or negative.
[ENOSPC]	Insufficiently many semaphores are available.
[ENOSPC]	The kernel could not allocate a *struct semid_ds*.
[ENOENT]	No semaphore set was found corresponding to *key*, and IPC_CREAT was not specified.

SEE ALSO

semctl(2), semop(2), ftok(3)

NAME

semop - atomic array of operations on a semaphore set

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int
semop(int semid, struct sembuf *array, size_t nops);
```

DESCRIPTION

The **semop**() system call atomically performs the array of operations indicated by *array* on the semaphore set indicated by *semid*. The length of *array* is indicated by *nops*. Each operation is encoded in a *struct sembuf*, which is defined as follows:

```
struct sembuf {
        u_short sem_num;        /* semaphore # */
        short   sem_op;         /* semaphore operation */
        short   sem_flg;        /* operation flags */
};
```

For each element in *array*, *sem_op* and *sem_flg* determine an operation to be performed on semaphore number *sem_num* in the set. The values SEM_UNDO and IPC_NOWAIT may be *OR*'ed into the *sem_flg* member in order to modify the behavior of the given operation.

The operation performed depends as follows on the value of *sem_op*:

- When *sem_op* is positive and the process has alter permission, the semaphore's value is incremented by *sem_op*'s value. If SEM_UNDO is specified, the semaphore's adjust on exit value is decremented by *sem_op*'s value. A positive value for *sem_op* generally corresponds to a process releasing a resource associated with the semaphore.

- The behavior when *sem_op* is negative and the process has alter permission, depends on the current value of the semaphore:

    - If the current value of the semaphore is greater than or equal to the absolute value of *sem_op*,

then the value is decremented by the absolute value of *sem_op*. If SEM_UNDO is specified, the semaphore's adjust on exit value is incremented by the absolute value of *sem_op*.

- If the current value of the semaphore is less than the absolute value of *sem_op*, one of the following happens:
  - If IPC_NOWAIT was specified, then **semop**() returns immediately with a return value of EAGAIN.
  - Otherwise, the calling process is put to sleep until one of the following conditions is satisfied:
    - Some other process removes the semaphore with the IPC_RMID option of semctl(2). In this case, **semop**() returns immediately with a return value of EIDRM.
    - The process receives a signal that is to be caught. In this case, the process will resume execution as defined by sigaction(2).
    - The semaphore's value is greater than or equal to the absolute value of *sem_op*. When this condition becomes true, the semaphore's value is decremented by the absolute value of *sem_op*, the semaphore's adjust on exit value is incremented by the absolute value of *sem_op*.

  A negative value for *sem_op* generally means that a process is waiting for a resource to become available.

- When *sem_op* is zero and the process has read permission, one of the following will occur:
  - If the current value of the semaphore is equal to zero then **semop**() can return immediately.
  - If IPC_NOWAIT was specified, then **semop**() returns immediately with a return value of EAGAIN.
  - Otherwise, the calling process is put to sleep until one of the following conditions is satisfied:
    - Some other process removes the semaphore with the IPC_RMID option of semctl(2). In this case, **semop**() returns immediately with a return value of EIDRM.
    - The process receives a signal that is to be caught. In this case, the process will resume execution as defined by sigaction(2).

- The semaphore's value becomes zero.

For each semaphore a process has in use, the kernel maintains an "adjust on exit" value, as alluded to earlier. When a process exits, either voluntarily or involuntarily, the adjust on exit value for each semaphore is added to the semaphore's value. This can be used to ensure that a resource is released if a process terminates unexpectedly.

RETURN VALUES

The **semop**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **semop**() system call will fail if:

[EINVAL]	No semaphore set corresponds to *semid*, or the process would exceed the system-defined limit for the number of per-process SEM_UNDO structures.
[EACCES]	Permission denied due to mismatch between operation and mode of semaphore set.
[EAGAIN]	The semaphore's value would have resulted in the process being put to sleep and IPC_NOWAIT was specified.
[E2BIG]	Too many operations were specified. [SEMOPM]
[EFBIG]	*sem_num* was not in the range of valid semaphores for the set.
[EIDRM]	The semaphore set was removed from the system.
[EINTR]	The **semop**() system call was interrupted by a signal.
[ENOSPC]	The system SEM_UNDO pool [SEMMNU] is full.
[ERANGE]	The requested operation would cause either the semaphore's current value [SEMVMX] or its adjust on exit value [SEMAEM] to exceed the system-imposed limits.

SEE ALSO

semctl(2), semget(2), sigaction(2)

BUGS

The **semop**() system call may block waiting for memory even if IPC_NOWAIT was specified.

NAME

send, **sendto**, **sendmsg**, **sendmmsg** - send message(s) from a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

ssize_t
send(*int s*, *const void *msg*, *size_t len*, *int flags*);

ssize_t
sendto(*int s*, *const void *msg*, *size_t len*, *int flags*, *const struct sockaddr *to*, *socklen_t tolen*);

ssize_t
sendmsg(*int s*, *const struct msghdr *msg*, *int flags*);

ssize_t
sendmmsg(*int s*, *struct mmsghdr * restrict msgvec*, *size_t vlen*, *int flags*);

DESCRIPTION

The **send**() and **sendmmsg**() functions, and **sendto**() and **sendmsg**() system calls are used to transmit one or more messages (with the **sendmmsg**() call) to another socket. The **send**() function may be used only when the socket is in a *connected* state, while **sendto**(), **sendmsg**() and **sendmmsg**() may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

The **sendmmsg**() function sends multiple messages at a call. They are given by the *msgvec* vector along with *vlen* specifying the vector size. The number of octets sent per each message is placed in the *msg_len* field of each processed element of the vector after transmission.

No indication of failure to deliver is implicit in a **send**(). Locally detected errors are indicated by a return value of -1.

If no messages space is available at the socket to hold the message to be transmitted, then **send**() normally blocks, unless the socket has been placed in non-blocking I/O mode. The select(2) system call

may be used to determine when it is possible to send more data.

The *flags* argument may include one or more of the following:

```
#define    MSG_OOB                 0x00001 /* process out-of-band data */
#define    MSG_DONTROUTE           0x00004 /* bypass routing, use direct interface */
#define MSG_EOR             0x00008 /* data completes record */
#define    MSG_EOF                 0x00100 /* data completes transaction */
#define    MSG_NOSIGNAL 0x20000 /* do not generate SIGPIPE on EOF */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_EOR is used to indicate a record mark for protocols which support the concept. MSG_EOF requests that the sender side of a socket be shut down, and that an appropriate indication be sent at the end of the specified data; this flag is only implemented for SOCK_STREAM sockets in the PF_INET protocol family. MSG_DONTROUTE is usually used only by diagnostic or routing programs. MSG_NOSIGNAL is used to prevent SIGPIPE generation when writing a socket that may be closed.

See recv(2) for a description of the *msghdr* structure and the *mmsghdr* structure.

RETURN VALUES

The **send**(), **sendto**() and **sendmsg**() calls return the number of octets sent. The **sendmmsg**() call returns the number of messages sent. If an error occurred a value of -1 is returned.

ERRORS

The **send**() and **sendmmsg**() functions and **sendto**() and **sendmsg**() system calls fail if:

[EBADF] An invalid descriptor was specified.

[EACCES] The destination address is a broadcast address, and SO_BROADCAST has not been set on the socket.

[ENOTSOCK] The argument *s* is not a socket.

[EFAULT] An invalid user space address was specified for an argument.

[EMSGSIZE] The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EAGAIN] The socket is marked non-blocking and the requested operation would block.

[ENOBUFS] The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS] The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

[EHOSTUNREACH]
The remote host was unreachable.

[EISCONN] A destination address was specified and the socket is already connected.

[ECONNREFUSED]
The socket received an ICMP destination unreachable message from the last message sent. This typically means that the receiver is not listening on the remote port.

[EHOSTDOWN] The remote host was down.

[ENETDOWN] The remote network was down.

[EADDRNOTAVAIL]
The process using a SOCK_RAW socket was jailed and the source address specified in the IP header did not match the IP address bound to the prison.

[EPIPE] The socket is unable to send anymore data (SBS_CANTSENDMORE has been set on the socket). This typically means that the socket is not connected.

SEE ALSO

fcntl(2), getsockopt(2), recv(2), select(2), socket(2), write(2), CMSG_DATA(3)

HISTORY

The **send**() function appeared in 4.2BSD. The **sendmmsg**() function appeared in FreeBSD 11.0.

BUGS

Because **sendmsg**() does not necessarily block until the data has been transferred, it is possible to transfer an open file descriptor across an AF_UNIX domain socket (see recv(2)), then **close**() it before it has actually been sent, the result being that the receiver gets a closed file descriptor. It is left to the application to implement an acknowledgment mechanism to prevent this from happening.

NAME

sendfile - send a file to a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>

int
sendfile(*int fd*, *int s*, *off_t offset*, *size_t nbytes*, *struct sf_hdtr *hdtr*, *off_t *sbytes*, *int flags*);

DESCRIPTION

The **sendfile**() system call sends a regular file or shared memory object specified by descriptor *fd* out a stream socket specified by descriptor *s*.

The *offset* argument specifies where to begin in the file. Should *offset* fall beyond the end of file, the system will return success and report 0 bytes sent as described below. The *nbytes* argument specifies how many bytes of the file should be sent, with 0 having the special meaning of send until the end of file has been reached.

An optional header and/or trailer can be sent before and after the file data by specifying a pointer to a *struct sf_hdtr*, which has the following structure:

```
struct sf_hdtr {
        struct iovec *headers;          /* pointer to header iovecs */
        int hdr_cnt;                    /* number of header iovecs */
        struct iovec *trailers;         /* pointer to trailer iovecs */
        int trl_cnt;                    /* number of trailer iovecs */
};
```

The *headers* and *trailers* pointers, if non-NULL, point to arrays of *struct iovec* structures. See the **writev**() system call for information on the iovec structure. The number of iovecs in these arrays is specified by *hdr_cnt* and *trl_cnt*.

If non-NULL, the system will write the total number of bytes sent on the socket to the variable pointed to by *sbytes*.

The least significant 16 bits of *flags* argument is a bitmap of these values:

SF_NODISKIO
: This flag causes **sendfile** to return EBUSY instead of blocking when a busy page is encountered. This rare situation can happen if some other process is now working with the same region of the file. It is advised to retry the operation after a short period.

  Note that in older FreeBSD versions the SF_NODISKIO had slightly different notion. The flag prevented **sendfile** to run I/O operations in case if an invalid (not cached) page is encountered, thus avoiding blocking on I/O. Starting with FreeBSD 11 **sendfile** sending files off the ffs(7) filesystem does not block on I/O (see *IMPLEMENTATION NOTES*), so the condition no longer applies. However, it is safe if an application utilizes SF_NODISKIO and on EBUSY performs the same action as it did in older FreeBSD versions, e.g., aio_read(2), read(2) or **sendfile** in a different context.

SF_NOCACHE
: The data sent to socket will not be cached by the virtual memory system, and will be freed directly to the pool of free pages.

SF_SYNC
: **sendfile** sleeps until the network stack no longer references the VM pages of the file, making subsequent modifications to it safe. Please note that this is not a guarantee that the data has actually been sent.

SF_USER_READAHEAD
: **sendfile** has some internal heuristics to do readahead when sending data. This flag forces **sendfile** to override any heuristically calculated readahead and use exactly the application specified readahead. See *SETTING READAHEAD* for more details on readahead.

When using a socket marked for non-blocking I/O, **sendfile**() may send fewer bytes than requested. In this case, the number of bytes successfully written is returned in **sbytes* (if specified), and the error EAGAIN is returned.

SETTING READAHEAD

sendfile uses internal heuristics based on request size and file system layout to do readahead. Additionally application may request extra readahead. The most significant 16 bits of *flags* specify amount of pages that **sendfile** may read ahead when reading the file. A macro **SF_FLAGS**() is provided to combine readahead amount and flags. An example showing specifying readahead of 16 pages and SF_NOCACHE flag:

SF_FLAGS(16, SF_NOCACHE)

sendfile will use either application specified readahead or internally calculated, whichever is bigger. Setting flag SF_USER_READAHEAD would turn off any heuristics and set maximum possible readahead length to the number of pages specified via flags.

IMPLEMENTATION NOTES

The FreeBSD implementation of **sendfile**() does not block on disk I/O when it sends a file off the ffs(7) filesystem. The syscall returns success before the actual I/O completes, and data is put into the socket later unattended. However, the order of data in the socket is preserved, so it is safe to do further writes to the socket.

The FreeBSD implementation of **sendfile**() is "zero-copy", meaning that it has been optimized so that copying of the file data is avoided.

TUNING

On some architectures, this system call internally uses a special **sendfile**() buffer (*struct sf_buf*) to handle sending file data to the client. If the sending socket is blocking, and there are not enough **sendfile**() buffers available, **sendfile**() will block and report a state of "sfbufa". If the sending socket is non-blocking and there are not enough **sendfile**() buffers available, the call will block and wait for the necessary buffers to become available before finishing the call.

The number of *sf_buf*'s allocated should be proportional to the number of nmbclusters used to send data to a client via **sendfile**(). Tune accordingly to avoid blocking! Busy installations that make extensive use of **sendfile**() may want to increase these values to be inline with their *kern.ipc.nmbclusters* (see tuning(7) for details).

The number of **sendfile**() buffers available is determined at boot time by either the *kern.ipc.nsfbufs* loader.conf(5) variable or the NSFBUFS kernel configuration tunable. The number of **sendfile**() buffers scales with *kern.maxusers*. The *kern.ipc.nsfbufsused* and *kern.ipc.nsfbufspeak* read-only sysctl(8) variables show current and peak **sendfile**() buffers usage respectively. These values may also be viewed through **netstat -m**.

If a value of zero is reported for *kern.ipc.nsfbufs*, your architecture does not need to use **sendfile**() buffers because their task can be efficiently performed by the generic virtual memory structures.

RETURN VALUES

The **sendfile**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EAGAIN] The socket is marked for non-blocking I/O and not all data was sent due to the socket buffer being filled. If specified, the number of bytes successfully sent will be returned in **sbytes*.

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

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

[EBUSY] A busy page was encountered and SF_NODISKIO had been specified. Partial data may have been sent.

[EFAULT] An invalid address was specified for an argument.

[EINTR] A signal interrupted **sendfile**() before it could be completed. If specified, the number of bytes successfully sent will be returned in **sbytes*.

[EINVAL] The *fd* argument is not a regular file.

[EINVAL] The *s* argument is not a SOCK_STREAM type socket.

[EINVAL] The *offset* argument is negative.

[EIO] An error occurred while reading from *fd*.

[ENOTCAPABLE] The *fd* or the *s* argument has insufficient rights.

[ENOBUFS] The system was unable to allocate an internal buffer.

[ENOTCONN] The *s* argument points to an unconnected socket.

[ENOTSOCK] The *s* argument is not a socket.

[EOPNOTSUPP] The file system for descriptor *fd* does not support **sendfile**().

[EPIPE] The socket peer has closed the connection.

SEE ALSO

netstat(1), open(2), send(2), socket(2), writev(2), tuning(7)

K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel, "A Portable Kernel Abstraction for Low-Overhead Ephemeral Mapping Management", *The Proceedings of the 2005 USENIX Annual Technical Conference*, pp 223-236, 2005.

HISTORY

The **sendfile**() system call first appeared in FreeBSD 3.0. This manual page first appeared in FreeBSD 3.1. In FreeBSD 10 support for sending shared memory descriptors had been introduced. In FreeBSD 11 a non-blocking implementation had been introduced.

AUTHORS

The initial implementation of **sendfile**() system call and this manual page were written by David G. Lawrence <*dg@dglawrence.com*>. The FreeBSD 11 implementation was written by Gleb Smirnoff <*glebius@FreeBSD.org*>.

NAME

setbuf, **setbuffer**, **setlinebuf**, **setvbuf** - stream buffering operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

void
setbuf(*FILE * restrict stream*, *char * restrict buf*);

void
setbuffer(*FILE *stream*, *char *buf*, *int size*);

int
setlinebuf(*FILE *stream*);

int
setvbuf(*FILE * restrict stream*, *char * restrict buf*, *int mode*, *size_t size*);

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered many characters are saved up and written as a block; when it is line buffered characters are saved up until a newline is output or input is read from any stream attached to a terminal device (typically stdin). The function fflush(3) may be used to force the block out early. (See fclose(3).)

Normally all files are block buffered. When the first I/O operation occurs on a file, malloc(3) is called, and an optimally-sized buffer is obtained. If a stream refers to a terminal (as stdout normally does) it is line buffered. The standard error stream stderr is always unbuffered. Note that these defaults may be altered using the stdbuf(1) utility.

The **setvbuf**() function may be used to alter the buffering behavior of a stream. The *mode* argument must be one of the following three macros:

_IONBF
: unbuffered

_IOLBF

line buffered

_IOFBF fully buffered

The *size* argument may be given as zero to obtain deferred optimal-size buffer allocation as usual. If it is not zero, then except for unbuffered files, the *buf* argument should point to a buffer at least *size* bytes long; this buffer will be used instead of the current buffer. If *buf* is not NULL, it is the caller's responsibility to free(3) this buffer after closing the stream. (If the *size* argument is not zero but *buf* is NULL, a buffer of the given size will be allocated immediately, and released on close. This is an extension to ANSI C; portable code should use a size of 0 with any NULL buffer.)

The **setvbuf**() function may be used at any time, but may have peculiar side effects (such as discarding input or flushing output) if the stream is ''active''. Portable applications should call it only once on any given stream, and before any I/O is performed.

The other three calls are, in effect, simply aliases for calls to **setvbuf**(). Except for the lack of a return value, the **setbuf**() function is exactly equivalent to the call

```
setvbuf(stream, buf, buf ? _IOFBF : _IONBF, BUFSIZ);
```

The **setbuffer**() function is the same, except that the size of the buffer is up to the caller, rather than being determined by the default BUFSIZ. The **setlinebuf**() function is exactly equivalent to the call:

```
setvbuf(stream, (char *)NULL, _IOLBF, 0);
```

RETURN VALUES

The **setvbuf**() function returns 0 on success, or EOF if the request cannot be honored (note that the stream is still functional in this case).

The **setlinebuf**() function returns what the equivalent **setvbuf**() would have returned.

SEE ALSO

stdbuf(1), fclose(3), fopen(3), fread(3), malloc(3), printf(3), puts(3)

STANDARDS

The **setbuf**() and **setvbuf**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

BUGS

setbuf() usually uses a suboptimal buffer size and should be avoided.

NAME

setuid, **seteuid**, **setgid**, **setegid** - set user and group ID

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
setuid(*uid_t uid*);

int
seteuid(*uid_t euid*);

int
setgid(*gid_t gid*);

int
setegid(*gid_t egid*);

DESCRIPTION

The **setuid**() system call sets the real and effective user IDs and the saved set-user-ID of the current process to the specified value. The **setuid**() system call is permitted if the specified ID is equal to the real user ID or the effective user ID of the process, or if the effective user ID is that of the super user.

The **setgid**() system call sets the real and effective group IDs and the saved set-group-ID of the current process to the specified value. The **setgid**() system call is permitted if the specified ID is equal to the real group ID or the effective group ID of the process, or if the effective user ID is that of the super user.

The **seteuid**() system call (**setegid**()) sets the effective user ID (group ID) of the current process. The effective user ID may be set to the value of the real user ID or the saved set-user-ID (see intro(2) and execve(2)); in this way, the effective user ID of a set-user-ID executable may be toggled by switching to the real user ID, then re-enabled by reverting to the set-user-ID value. Similarly, the effective group ID may be set to the value of the real group ID or the saved set-group-ID.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The system calls will fail if:

[EPERM] The user is not the super user and the ID specified is not the real, effective ID, or saved ID.

SEE ALSO

getgid(2), getuid(2), issetugid(2), setregid(2), setreuid(2)

STANDARDS

The **setuid**() and **setgid**() system calls are compliant with the IEEE Std 1003.1-1990 ("POSIX.1") specification with _POSIX_SAVED_IDS not defined with the permitted extensions from Appendix B.4.2.2. The **seteuid**() and **setegid**() system calls are extensions based on the POSIX concept of _POSIX_SAVED_IDS, and have been proposed for a future revision of the standard.

HISTORY

The **setuid**() function appeared in Version 1 AT&T UNIX. The **setgid**() function appeared in Version 4 AT&T UNIX.

SECURITY CONSIDERATIONS

Read and write permissions to files are determined upon a call to open(2). Once a file descriptor is open, dropping privilege does not affect the process's read/write permissions, even if the user ID specified has no read or write permissions to the file. These files normally remain open in any new process executed, resulting in a user being able to read or modify potentially sensitive data.

To prevent these files from remaining open after an exec(3) call, be sure to set the close-on-exec flag:

```
void
pseudocode(void)
{
        int fd;
        /* ... */

        fd = open("/path/to/sensitive/data", O_RDWR | O_CLOEXEC);
        if (fd == -1)
                err(1, "open");

        /* ... */
        execve(path, argv, environ);
}
```

NAME

setfib - set the default FIB (routing table) for the calling process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

int
setfib(*int fib*);

DESCRIPTION

The **setfib**() system call sets the associated fib for all sockets opened subsequent to the call, to be that of the argument *fib*. The *fib* argument must be greater than or equal to 0 and less than the current system maximum which may be retrieved by the *net.fibs* sysctl. The system maximum is set in the kernel configuration file with

options ROUTETABLES=*N*

or in */boot/loader.conf* with

net.fibs="*N*"

where *N* is an integer. This maximum is capped at 65536 due to the implementation storing the fib number in a 16-bit field in the mbuf(9) packet header, however it is not suggested that one use such a large number as memory is allocated for every FIB regardless of whether it is used, and there are places where all FIBs are iterated over.

The default fib of the process will be applied to all protocol families that support multiple fibs, and ignored by those that do not. The default fib for a process may be overridden for a socket with the use of the SO_SETFIB socket option.

RETURN VALUES

The **setfib**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **setfib**() system call will fail and no action will be taken and return EINVAL if the *fib* argument is greater than the current system maximum.

SEE ALSO

setfib(1), setsockopt(2)

STANDARDS

The **setfib**() system call is a FreeBSD extension however similar extensions have been added to many other UNIX style kernels.

HISTORY

The **setfib**() function appeared in FreeBSD 7.1.

NAME

setgroups - set group access list

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <unistd.h>

int
setgroups(*int ngroups*, *const gid_t *gidset*);

DESCRIPTION

The **setgroups**() system call sets the group access list of the current user process according to the array *gidset*. The *ngroups* argument indicates the number of entries in the array and must be no more than {NGROUPS_MAX}+1.

Only the super-user may set a new group list.

The first entry of the group array (*gidset[0]*) is used as the effective group-ID for the process. This entry is over-written when a setgid program is run. To avoid losing access to the privileges of the *gidset[0]* entry, it should be duplicated later in the group array. By convention, this happens because the group value indicated in the password file also appears in */etc/group*. The group value in the password file is placed in *gidset[0]* and that value then gets added a second time when the */etc/group* file is scanned to create the group set.

RETURN VALUES

The **setgroups**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **setgroups**() system call will fail if:

[EPERM]	The caller is not the super-user.
[EINVAL]	The number specified in the *ngroups* argument is larger than the {NGROUPS_MAX}+1 limit.
[EFAULT]	The address specified for *gidset* is outside the process address space.

SEE ALSO

getgroups(2), initgroups(3)

HISTORY

The **setgroups**() system call appeared in 4.2BSD.

NAME

setlocale - natural language formatting for C

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <locale.h>

*char **
setlocale(*int category*, *const char *locale*);

DESCRIPTION

The **setlocale**() function sets the C library's notion of natural language formatting style for particular sets of routines. Each such style is called a 'locale' and is invoked using an appropriate name passed as a C string.

The **setlocale**() function recognizes several categories of routines. These are the categories and the sets of routines they select:

LC_ALL	Set the entire locale generically.
LC_COLLATE	Set a locale for string collation routines. This controls alphabetic ordering in **strcoll**() and **strxfrm**().
LC_CTYPE	Set a locale for the ctype(3) and multibyte(3) functions. This controls recognition of upper and lower case, alphabetic or non-alphabetic characters, and so on.
LC_MESSAGES	Set a locale for message catalogs, see catopen(3) function.
LC_MONETARY	Set a locale for formatting monetary values; this affects the **localeconv**() function.
LC_NUMERIC	Set a locale for formatting numbers. This controls the formatting of decimal points in input and output of floating point numbers in functions such as **printf**() and **scanf**(), as well as values returned by **localeconv**().
LC_TIME	Set a locale for formatting dates and times using the **strftime**() function.

Only three locales are defined by default, the empty string "" which denotes the native environment, and the "C" and "POSIX" locales, which denote the C language environment. A *locale* argument of NULL

causes **setlocale**() to return the current locale. By default, C programs start in the "C" locale. The only function in the library that sets the locale is **setlocale**(); the locale is never changed as a side effect of some other routine.

RETURN VALUES

Upon successful completion, **setlocale**() returns the string associated with the specified *category* for the requested *locale*. The **setlocale**() function returns NULL and fails to change the locale if the given combination of *category* and *locale* makes no sense.

FILES

$PATH_LOCALE/locale/category
/usr/share/locale/locale/category locale file for the locale *locale* and the category *category*.

ERRORS

No errors are defined.

SEE ALSO

colldef(1), mklocale(1), catopen(3), ctype(3), localeconv(3), multibyte(3), strcoll(3), strxfrm(3), euc(5), utf8(5), environ(7)

STANDARDS

The **setlocale**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

HISTORY

The **setlocale**() function first appeared in 4.4BSD.

NAME

setpgid, **setpgrp** - set process group

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
setpgid(*pid_t pid*, *pid_t pgrp*);

int
setpgrp(*pid_t pid*, *pid_t pgrp*);

DESCRIPTION

The **setpgid**() system call sets the process group of the specified process *pid* to the specified *pgrp*. If *pid* is zero, then the call applies to the current process. If *pgrp* is zero, then the process id of the process specified by *pid* is used instead.

If the affected process is not the invoking process, then it must be a child of the invoking process, it must not have performed an exec(3) operation, and both processes must be in the same session. The requested process group ID must already exist in the session of the caller, or it must be equal to the target process ID.

RETURN VALUES

The **setpgid**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

COMPATIBILITY

The **setpgrp**() system call is identical to **setpgid**(), and is retained for calling convention compatibility with historical versions of BSD.

ERRORS

The **setpgid**() system call will fail and the process group will not be altered if:

[EINVAL] The requested process group ID is not legal.

[ESRCH] The requested process does not exist.

[ESRCH] The target process is not the calling process or a child of the calling process.

[EACCES] The requested process is a child of the calling process, but it has performed an exec(3) operation.

[EPERM] The target process is a session leader.

[EPERM] The requested process group ID is not in the session of the caller, and it is not equal to the process ID of the target process.

SEE ALSO

getpgrp(2)

STANDARDS

The **setpgid**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

NAME

setregid - set real and effective group ID

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
setregid(*gid_t rgid*, *gid_t egid*);

DESCRIPTION

The real and effective group ID's of the current process are set to the arguments. If the real group ID is changed, the saved group ID is changed to the new value of the effective group ID.

Unprivileged users may change the real group ID to the effective group ID and vice-versa; only the super-user may make other changes.

Supplying a value of -1 for either the real or effective group ID forces the system to substitute the current ID in place of the -1 argument.

The **setregid**() system call was intended to allow swapping the real and effective group IDs in set-group-ID programs to temporarily relinquish the set-group-ID value. This system call did not work correctly, and its purpose is now better served by the use of the setegid(2) system call.

When setting the real and effective group IDs to the same value, the standard **setgid**() system call is preferred.

RETURN VALUES

The **setregid**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the super-user and a change other than changing the effective group-id to the real group-id was specified.

SEE ALSO

getgid(2), issetugid(2), setegid(2), setgid(2), setuid(2)

HISTORY

The **setregid**() system call appeared in 4.2BSD.

NAME

setreuid - set real and effective user ID's

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
setreuid(*uid_t ruid*, *uid_t euid*);

DESCRIPTION

The real and effective user IDs of the current process are set according to the arguments. If *ruid* or *euid* is -1, the current uid is filled in by the system. Unprivileged users may change the real user ID to the effective user ID and vice-versa; only the super-user may make other changes.

If the real user ID is changed (i.e. *ruid* is not -1) or the effective user ID is changed to something other than the real user ID, then the saved user ID will be set to the effective user ID.

The **setreuid**() system call has been used to swap the real and effective user IDs in set-user-ID programs to temporarily relinquish the set-user-ID value. This purpose is now better served by the use of the seteuid(2) system call.

When setting the real and effective user IDs to the same value, the standard **setuid**() system call is preferred.

RETURN VALUES

The **setreuid**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EPERM] The current process is not the super-user and a change other than changing the effective user-id to the real user-id was specified.

SEE ALSO

getuid(2), issetugid(2), seteuid(2), setuid(2)

HISTORY

The **setreuid**() system call appeared in 4.2BSD.

NAME

setruid, **setrgid** - set user and group ID

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
setruid(*uid_t ruid*);

int
setrgid(*gid_t rgid*);

DESCRIPTION

The **setruid**() function (**setrgid**()) sets the real user ID (group ID) of the current process.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

COMPATIBILITY

The use of these calls is not portable. Their use is discouraged; they will be removed in the future.

ERRORS

The functions fail if:

[EPERM] The user is not the super user and the ID specified is not the real or effective ID.

SEE ALSO

getgid(2), getuid(2), setegid(2), seteuid(2), setgid(2), setuid(2)

HISTORY

The **setruid**() and **setrgid**() syscalls appeared in 4.2BSD and were dropped in 4.4BSD.

NAME

setsid - create session and set process group ID

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
setsid(*void*);

DESCRIPTION

The **setsid**() system call creates a new session. The calling process is the session leader of the new session, is the process group leader of a new process group and has no controlling terminal. The calling process is the only process in either the session or the process group.

RETURN VALUES

Upon successful completion, the **setsid**() system call returns the value of the process group ID of the new process group, which is the same as the process ID of the calling process. If an error occurs, **setsid**() returns -1 and the global variable *errno* is set to indicate the error.

ERRORS

The **setsid**() system call will fail if:

[EPERM] The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

SEE ALSO

setpgid(2), tcgetpgrp(3), tcsetpgrp(3)

STANDARDS

The **setsid**() system call is expected to be compliant with the IEEE Std 1003.1-1990 ("POSIX.1") specification.

NAME

shm_open, **shm_unlink** - shared memory object operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>

int
shm_open(const char *path, int flags, mode_t mode);

int
shm_unlink(const char *path);
```

DESCRIPTION

The **shm_open**() system call opens (or optionally creates) a POSIX shared memory object named *path*. The *flags* argument contains a subset of the flags used by open(2). An access mode of either O_RDONLY or O_RDWR must be included in *flags*. The optional flags O_CREAT, O_EXCL, and O_TRUNC may also be specified.

If O_CREAT is specified, then a new shared memory object named *path* will be created if it does not exist. In this case, the shared memory object is created with mode *mode* subject to the process' umask value. If both the O_CREAT and O_EXCL flags are specified and a shared memory object named *path* already exists, then **shm_open**() will fail with EEXIST.

Newly created objects start off with a size of zero. If an existing shared memory object is opened with O_RDWR and the O_TRUNC flag is specified, then the shared memory object will be truncated to a size of zero. The size of the object can be adjusted via ftruncate(2) and queried via fstat(2).

The new descriptor is set to close during execve(2) system calls; see close(2) and fcntl(2).

As a FreeBSD extension, the constant SHM_ANON may be used for the *path* argument to **shm_open**(). In this case, an anonymous, unnamed shared memory object is created. Since the object has no name, it cannot be removed via a subsequent call to **shm_unlink**(). Instead, the shared memory object will be garbage collected when the last reference to the shared memory object is removed. The shared memory object may be shared with other processes by sharing the file descriptor via fork(2) or sendmsg(2). Attempting to open an anonymous shared memory object with O_RDONLY will fail with EINVAL.

All other flags are ignored.

The **shm_unlink**() system call removes a shared memory object named *path*.

RETURN VALUES

If successful, **shm_open**() returns a non-negative integer, and **shm_unlink**() returns zero. Both functions return -1 on failure, and set *errno* to indicate the error.

COMPATIBILITY

The *path* argument does not necessarily represent a pathname (although it does in most other implementations). Two processes opening the same *path* are guaranteed to access the same shared memory object if and only if *path* begins with a slash ('/') character.

Only the O_RDONLY, O_RDWR, O_CREAT, O_EXCL, and O_TRUNC flags may be used in portable programs.

POSIX specifications state that the result of using open(2), read(2), or write(2) on a shared memory object, or on the descriptor returned by **shm_open**(), is undefined. However, the FreeBSD kernel implementation explicitly includes support for read(2) and write(2).

FreeBSD also supports zero-copy transmission of data from shared memory objects with sendfile(2).

Neither shared memory objects nor their contents persist across reboots.

Writes do not extend shared memory objects, so ftruncate(2) must be called before any data can be written. See *EXAMPLES*.

EXAMPLES

This example fails without the call to ftruncate(2):

```
uint8_t buffer[getpagesize()];
ssize_t len;
int fd;

fd = shm_open(SHM_ANON, O_RDWR | O_CREAT, 0600);
if (fd < 0)
	err(EX_OSERR, "%s: shm_open", __func__);
if (ftruncate(fd, getpagesize()) < 0)
	err(EX_IOERR, "%s: ftruncate", __func__);
len = pwrite(fd, buffer, getpagesize(), 0);
```

```
        if (len < 0)
                err(EX_IOERR, "%s: pwrite", __func__);
        if (len != getpagesize())
                errx(EX_IOERR, "%s: pwrite length mismatch", __func__);
```

ERRORS

shm_open() fails with these error codes for these conditions:

[EINVAL] A flag other than O_RDONLY, O_RDWR, O_CREAT, O_EXCL, or O_TRUNC was included in *flags*.

[EMFILE] The process has already reached its limit for open file descriptors.

[ENFILE] The system file table is full.

[EINVAL] O_RDONLY was specified while creating an anonymous shared memory object via SHM_ANON.

[EFAULT] The *path* argument points outside the process' allocated address space.

[ENAMETOOLONG]
The entire pathname exceeded 1023 characters.

[EINVAL] The *path* does not begin with a slash ('/') character.

[ENOENT] O_CREAT is specified and the named shared memory object does not exist.

[EEXIST] O_CREAT and O_EXCL are specified and the named shared memory object does exist.

[EACCES] The required permissions (for reading or reading and writing) are denied.

shm_unlink() fails with these error codes for these conditions:

[EFAULT] The *path* argument points outside the process' allocated address space.

[ENAMETOOLONG]
The entire pathname exceeded 1023 characters.

[ENOENT] The named shared memory object does not exist.

[EACCES] The required permissions are denied. **shm_unlink**() requires write permission to the shared memory object.

SEE ALSO

close(2), fstat(2), ftruncate(2), mmap(2), munmap(2), sendfile(2)

STANDARDS

The **shm_open**() and **shm_unlink**() functions are believed to conform to IEEE Std 1003.1b-1993 ("POSIX.1b").

HISTORY

The **shm_open**() and **shm_unlink**() functions first appeared in FreeBSD 4.3. The functions were reimplemented as system calls using shared memory objects directly rather than files in FreeBSD 8.0.

AUTHORS

Garrett A. Wollman <*wollman@FreeBSD.org*> (C library support and this manual page)

Matthew Dillon <*dillon@FreeBSD.org*> (MAP_NOSYNC)

NAME

shmat, **shmdt** - attach or detach shared memory

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

*void **
shmat(*int shmid*, *const void *addr*, *int flag*);

int
shmdt(*const void *addr*);

DESCRIPTION

The **shmat**() system call attaches the shared memory segment identified by *shmid* to the calling process's address space. The address where the segment is attached is determined as follows:

- If *addr* is 0, the segment is attached at an address selected by the kernel.
- If *addr* is nonzero and SHM_RND is not specified in *flag*, the segment is attached the specified address.
- If *addr* is specified and SHM_RND is specified, *addr* is rounded down to the nearest multiple of SHMLBA.

The **shmdt**() system call detaches the shared memory segment at the address specified by *addr* from the calling process's address space.

RETURN VALUES

Upon success, **shmat**() returns the address where the segment is attached; otherwise, -1 is returned and *errno* is set to indicate the error.

The **shmdt**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **shmat**() system call will fail if:

[EINVAL] No shared memory segment was found corresponding to *shmid*.

[EINVAL] The *addr* argument was not an acceptable address.

[EMFILE] Failed to attach the shared memory segment because the per-process *kern.ipc.shmseg* sysctl(3) limit was reached.

The **shmdt**() system call will fail if:

[EINVAL] The *addr* argument does not point to a shared memory segment.

SEE ALSO

shmctl(2), shmget(2)

NAME

shmctl - shared memory control

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int
shmctl(*int shmid*, *int cmd*, *struct shmid_ds *buf*);

DESCRIPTION

Performs the action specified by *cmd* on the shared memory segment identified by *shmid*:

IPC_STAT — Fetch the segment's *struct shmid_ds*, storing it in the memory pointed to by *buf*.

IPC_SET — Changes the *shm_perm.uid*, *shm_perm.gid*, and *shm_perm.mode* members of the segment's *struct shmid_ds* to match those of the struct pointed to by *buf*. The calling process's effective uid must match either *shm_perm.uid* or *shm_perm.cuid*, or it must have superuser privileges.

IPC_RMID — Removes the segment from the system. The removal will not take effect until all processes having attached the segment have exited. For the operation to succeed, the calling process's effective uid must match *shm_perm.uid* or *shm_perm.cuid*, or the process must have superuser privileges. If the *kern.ipc.shm_allow_removed* sysctl(3) variable is set to 0, once the IPC_RMID operation has taken place, no further processes will be allowed to attach the segment.

The *shmid_ds* structure is defined as follows:

```
struct shmid_ds {
   struct ipc_perm shm_perm;   /* operation permission structure */
   size_t          shm_segsz;  /* size of segment in bytes */
   pid_t           shm_lpid;   /* process ID of last shared memory op */
   pid_t           shm_cpid;   /* process ID of creator */
   int             shm_nattch; /* number of current attaches */
   time_t          shm_atime;  /* time of last shmat() */
```

```
    time_t        shm_dtime;  /* time of last shmdt() */
    time_t        shm_ctime;  /* time of last change by shmctl() */
};
```

RETURN VALUES

The **shmctl**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **shmctl**() system call will fail if:

[EINVAL] Invalid operation, or no shared memory segment was found corresponding to *shmid*.

[EPERM] The calling process's effective uid does not match the uid of the shared memory segment's owner or creator.

[EACCES] Permission denied due to mismatch between operation and mode of shared memory segment.

SEE ALSO

shmat(2), shmdt(2), shmget(2), ftok(3)

NAME

shmget - obtain a shared memory identifier

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/shm.h>

int
shmget(*key_t key*, *size_t size*, *int flag*);

DESCRIPTION

Based on the values of *key* and *flag*, **shmget**() returns the identifier of a newly created or previously existing shared memory segment. The key is analogous to a filename: it provides a handle that names an IPC object. There are three ways to specify a key:

- IPC_PRIVATE may be specified, in which case a new IPC object will be created.
- An integer constant may be specified. If no IPC object corresponding to *key* is specified and the IPC_CREAT bit is set in *flag*, a new one will be created.
- The ftok(3) may be used to generate a key from a pathname.

The mode of a newly created IPC object is determined by which are set by ORing these constants into the *flag* argument:

0400 Read access for owner.

0200 Write access for owner.

0040 Read access for group.

0020 Write access for group.

0004 Read access for other.

0002 Write access for other.

When creating a new shared memory segment, *size* indicates the desired size of the new segment in

bytes. The size of the segment may be rounded up to a multiple convenient to the kernel (i.e., the page size).

RETURN VALUES

Upon successful completion, **shmget**() returns the positive integer identifier of a shared memory segment. Otherwise, -1 is returned and *errno* set to indicate the error.

ERRORS

The **shmget**() system call will fail if:

[EINVAL]	Size specified is greater than the size of the previously existing segment. Size specified is less than the system imposed minimum, or greater than the system imposed maximum.
[ENOENT]	No shared memory segment was found matching *key*, and IPC_CREAT was not specified.
[ENOSPC]	The kernel was unable to allocate enough memory to satisfy the request.
[EEXIST]	IPC_CREAT and IPC_EXCL were specified, and a shared memory segment corresponding to *key* already exists.

SEE ALSO

shmat(2), shmctl(2), shmdt(2), ftok(3)

NAME

shutdown - disable sends and/or receives on a socket

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

int
shutdown(*int s*, *int how*);

DESCRIPTION

The **shutdown**() system call disables sends or receives on a socket. The *how* argument specifies the type of shutdown. Possible values are:

SHUT_RD — Further receives will be disallowed.

SHUT_WR — Further sends will be disallowed. This may cause actions specific to the protocol family of the socket *s* to happen; see *IMPLEMENTATION NOTES*.

SHUT_RDWR — Further sends and receives will be disallowed. Implies SHUT_WR.

If the file descriptor *s* is associated with a SOCK_STREAM socket, all or part of the full-duplex connection will be shut down.

IMPLEMENTATION NOTES

The following protocol specific actions apply to the use of SHUT_WR (and potentially also SHUT_RDWR), based on the properties of the socket associated with the file descriptor *s*.

Domain	Type	Protocol	Action
PF_INET	SOCK_DGRAM	IPPROTO_SCTP	Failure, as socket is not connected.
PF_INET	SOCK_DGRAM	IPPROTO_UDP	Failure, as socket is not connected.
PF_INET	SOCK_STREAM	IPPROTO_SCTP	Send queued data and tear down association.
PF_INET	SOCK_STREAM	IPPROTO_TCP	Send queued data, wait for ACK, then send FIN.
PF_INET6	SOCK_DGRAM	IPPROTO_SCTP	Failure, as socket is not connected.
PF_INET6	SOCK_DGRAM	IPPROTO_UDP	Failure, as socket is not connected.

PF_INET6	SOCK_STREAM	IPPROTO_SCTP	Send queued data and tear down association.
PF_INET6	SOCK_STREAM	IPPROTO_TCP	Send queued data, wait for ACK, then send FIN.

RETURN VALUES

The **shutdown**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **shutdown**() system call fails if:

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

[EINVAL] The *how* argument is invalid.

[ENOTCONN] The *s* argument specifies a socket which is not connected.

[ENOTSOCK] The *s* argument does not refer to a socket.

SEE ALSO

connect(2), socket(2), inet(4), inet6(4)

STANDARDS

The **shutdown**() system call is expected to comply with IEEE Std 1003.1g-2000 ("POSIX.1g"), when finalized.

HISTORY

The **shutdown**() system call appeared in 4.2BSD. The SHUT_RD, SHUT_WR, and SHUT_RDWR constants appeared in IEEE Std 1003.1g-2000 ("POSIX.1g").

AUTHORS

This manual page was updated by Bruce M. Simpson <*bms@FreeBSD.org*> to reflect how **shutdown**() behaves with PF_INET and PF_INET6 sockets.

BUGS

The ICMP "port unreachable" message should be generated in response to datagrams received on a local port to which *s* is bound after **shutdown**() is called.

NAME

sigaction - software signal facilities

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <signal.h>

struct  sigaction {
        void    (*sa_handler)(int);
        void    (*sa_sigaction)(int, siginfo_t *, void *);
        int     sa_flags;               /* see signal options below */
        sigset_t sa_mask;               /* signal mask to apply */
};

int
sigaction(int sig, const struct sigaction * restrict act, struct sigaction * restrict oact);
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is normally blocked from further occurrence, the current thread context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. A signal may also be *blocked* for a thread, in which case it will not be delivered to that thread until it is *unblocked*. The action to be taken on delivery is determined at the time of delivery. Normally, signal handlers execute on the current stack of the thread. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

Signal routines normally execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a thread. The signal mask for a thread is initialized from that of its parent (normally empty). It may be changed with a sigprocmask(2) or pthread_sigmask(3) call, or when a signal is delivered to the thread.

When a signal condition arises for a process or thread, the signal is added to a set of signals pending for the process or thread. Whether the signal is directed at the process in general or at a specific thread depends on how it is generated. For signals directed at a specific thread, if the signal is not currently *blocked* by the thread then it is delivered to the thread. For signals directed at the process, if the signal is not currently *blocked* by all threads then it is delivered to one thread that does not have it blocked (the selection of which is unspecified). Signals may be delivered any time a thread enters the operating

system (e.g., during a system call, page fault or trap, or clock interrupt). If multiple signals are ready to be delivered at the same time, any signals that could be caused by traps are delivered first. Additional signals may be processed at the same time, with each appearing to interrupt the handlers for the previous signals before their first instructions. The set of pending signals is returned by the sigpending(2) system call. When a caught signal is delivered, the current state of the thread is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the thread will resume execution in the context from before the signal's delivery. If the thread wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a thread a new signal mask is installed for the duration of the process' signal handler (or until a sigprocmask(2) system call is made). This mask is formed by taking the union of the current signal mask set, the signal to be delivered, and the signal mask associated with the handler to be invoked.

The **sigaction**() system call assigns an action for a signal specified by *sig*. If *act* is non-NULL, it specifies an action (SIG_DFL, SIG_IGN, or a handler routine) and mask to be used when delivering the specified signal. If *oact* is non-NULL, the previous handling information for the signal is returned to the user.

The above declaration of *struct sigaction* is not literal. It is provided only to list the accessible members. See <*sys/signal.h*> for the actual definition. In particular, the storage occupied by *sa_handler* and *sa_sigaction* overlaps, and it is nonsensical for an application to attempt to use both simultaneously.

Once a signal handler is installed, it normally remains installed until another **sigaction**() system call is made, or an execve(2) is performed. A signal-specific default action may be reset by setting *sa_handler* to SIG_DFL. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. See the signal list below for each signal's default action. If *sa_handler* is SIG_DFL, the default action for the signal is to discard the signal, and if a signal is pending, the pending signal is discarded even if the signal is masked. If *sa_handler* is set to SIG_IGN current and pending instances of the signal are ignored and discarded.

Options may be specified by setting *sa_flags*. The meaning of the various bits is as follows:

SA_NOCLDSTOP If this bit is set when installing a catching function for the SIGCHLD signal, the SIGCHLD signal will be generated only when a child process exits, not when a child process stops.

SA_NOCLDWAIT If this bit is set when calling **sigaction**() for the SIGCHLD signal, the system will not create zombie processes when children of the calling

process exit. If the calling process subsequently issues a wait(2) (or equivalent), it blocks until all of the calling process's child processes terminate, and then returns a value of -1 with *errno* set to ECHILD. The same effect of avoiding zombie creation can also be achieved by setting *sa_handler* for SIGCHLD to SIG_IGN.

SA_ONSTACK — If this bit is set, the system will deliver the signal to the process on a *signal stack*, specified by each thread with sigaltstack(2).

SA_NODEFER — If this bit is set, further occurrences of the delivered signal are not masked during the execution of the handler.

SA_RESETHAND — If this bit is set, the handler is reset back to SIG_DFL at the moment the signal is delivered.

SA_RESTART — See paragraph below.

SA_SIGINFO — If this bit is set, the handler function is assumed to be pointed to by the *sa_sigaction* member of *struct sigaction* and should match the prototype shown above or as below in *EXAMPLES*. This bit should not be set when assigning SIG_DFL or SIG_IGN.

If a signal is caught during the system calls listed below, the call may be forced to terminate with the error EINTR, the call may return with a data transfer shorter than requested, or the call may be restarted. Restart of pending calls is requested by setting the SA_RESTART bit in *sa_flags*. The affected system calls include open(2), read(2), write(2), sendto(2), recvfrom(2), sendmsg(2) and recvmsg(2) on a communications channel or a slow device (such as a terminal, but not a regular file) and during a wait(2) or ioctl(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count).

After a pthread_create(3) the signal mask is inherited by the new thread and the set of pending signals and the signal stack for the new thread are empty.

After a fork(2) or vfork(2) all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child.

The execve(2) system call reinstates the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that restart pending system calls continue to do so.

The following is a list of all signals with names as in the include file <*signal.h*>:

NAME	Default Action	Description
SIGHUP	terminate process	terminal line hangup
SIGINT	terminate process	interrupt program
SIGQUIT	create core image	quit program
SIGILL	create core image	illegal instruction
SIGTRAP	create core image	trace trap
SIGABRT	create core image	abort(3) call (formerly SIGIOT)
SIGEMT	create core image	emulate instruction executed
SIGFPE	create core image	floating-point exception
SIGKILL	terminate process	kill program
SIGBUS	create core image	bus error
SIGSEGV	create core image	segmentation violation
SIGSYS	create core image	non-existent system call invoked
SIGPIPE	terminate process	write on a pipe with no reader
SIGALRM	terminate process	real-time timer expired
SIGTERM	terminate process	software termination signal
SIGURG	discard signal	urgent condition present on socket
SIGSTOP	stop process	stop (cannot be caught or ignored)
SIGTSTP	stop process	stop signal generated from keyboard
SIGCONT	discard signal	continue after stop
SIGCHLD	discard signal	child status has changed
SIGTTIN	stop process	background read attempted from control terminal
SIGTTOU	stop process	background write attempted to control terminal
SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
SIGWINCH	discard signal	window size change
SIGINFO	discard signal	status request from keyboard
SIGUSR1	terminate process	user defined signal 1
SIGUSR2	terminate process	user defined signal 2

NOTE

The *sa_mask* field specified in *act* is not allowed to block SIGKILL or SIGSTOP. Any attempt to do so will be silently ignored.

The following functions are either reentrant or not interruptible by signals and are async-signal safe.

Therefore applications may invoke them, without restriction, from signal-catching functions or from a child process after calling fork(2) in a multi-threaded process:

Base Interfaces:

_Exit(), **_exit**(), **accept**(), **access**(), **alarm**(), **bind**(), **cfgetispeed**(), **cfgetospeed**(), **cfsetispeed**(), **cfsetospeed**(), **chdir**(), **chmod**(), **chown**(), **close**(), **connect**(), **creat**(), **dup**(), **dup2**(), **execl**(), **execle**(), **execv**(), **execve**(), **faccessat**(), **fchdir**(), **fchmod**(), **fchmodat**(), **fchown**(), **fchownat**(), **fcntl**(), **fork**(), **fstat**(), **fstatat**(), **fsync**(), **ftruncate**(), **getegid**(), **geteuid**(), **getgid**(), **getgroups**(), **getpeername**(), **getpgrp**(), **getpid**(), **getppid**(), **getsockname**(), **getsockopt**(), **getuid**(), **kill**(), **link**(), **linkat**(), **listen**(), **lseek**(), **lstat**(), **mkdir**(), **mkdirat**(), **mkfifo**(), **mkfifoat**(), **mknod**(), **mknodat**(), **open**(), **openat**(), **pause**(), **pipe**(), **poll**(), **pselect**(), **pthread_sigmask**(), **raise**(), **read**(), **readlink**(), **readlinkat**(), **recv**(), **recvfrom**(), **recvmsg**(), **rename**(), **renameat**(), **rmdir**(), **select**(), **send**(), **sendmsg**(), **sendto**(), **setgid**(), **setpgid**(), **setsid**(), **setsockopt**(), **setuid**(), **shutdown**(), **sigaction**(), **sigaddset**(), **sigdelset**(), **sigemptyset**(), **sigfillset**(), **sigismember**(), **signal**(), **sigpending**(), **sigprocmask**(), **sigsuspend**(), **sleep**(), **sockatmark**(), **socket**(), **socketpair**(), **stat**(), **symlink**(), **symlinkat**(), **tcdrain**(), **tcflow**(), **tcflush**(), **tcgetattr**(), **tcgetpgrp**(), **tcsendbreak**(), **tcsetattr**(), **tcsetpgrp**(), **time**(), **times**(), **umask**(), **uname**(), **unlink**(), **unlinkat**(), **utime**(), **wait**(), **waitpid**(), **write**().

X/Open Systems Interfaces:

sigpause(), **sigset**(), **utimes**().

Realtime Interfaces:

aio_error(), **clock_gettime**(), **timer_getoverrun**(), **aio_return**(), **fdatasync**(), **sigqueue**(), **timer_gettime**(), **aio_suspend**(), **sem_post**(), **timer_settime**().

Base Interfaces not specified as async-signal safe by POSIX:

fpathconf(), **pathconf**(), **sysconf**().

Base Interfaces not specified as async-signal safe by POSIX, but planned to be:

ffs(), **htonl**(), **htons**(), **memccpy**(), **memchr**(), **memcmp**(), **memcpy**(), **memmove**(), **memset**(), **ntohl**(), **ntohs**(), **stpcpy**(), **stpncpy**(), **strcat**(), **strchr**(), **strcmp**(), **strcpy**(), **strcspn**(), **strlen**(), **strncat**(), **strncmp**(), **strncpy**(), **strnlen**(), **strpbrk**(), **strrchr**(), **strspn**(), **strstr**(), **strtok_r**(), **wcpcpy**(), **wcpncpy**(), **wcscat**(), **wcschr**(), **wcscmp**(), **wcscpy**(), **wcscspn**(), **wcslen**(), **wcsncat**(), **wcsncmp**(), **wcsncpy**(), **wcsnlen**(), **wcspbrk**(), **wcsrchr**(), **wcsspn**(), **wcsstr**(), **wcstok**(), **wmemchr**(), **wmemcmp**(), **wmemcpy**(), **wmemmove**(), **wmemset**().

Extension Interfaces:

accept4(), **bindat**(), **closefrom**(), **connectat**(), **eaccess**(), **ffsl**(), **ffsll**(), **flock**(), **fls**(), **flsl**(), **flsll**(), **futimesat**(), **pipe2**(), **strlcat**(). **strlcpy**(), **strsep**().

In addition, reading or writing *errno* is async-signal safe.

All functions not in the above lists are considered to be unsafe with respect to signals. That is to say, the behaviour of such functions is undefined when they are called from a signal handler that interrupted an unsafe function. In general though, signal handlers should do little more than set a flag; most other actions are not safe.

Also, it is good practice to make a copy of the global variable *errno* and restore it before returning from the signal handler. This protects against the side effect of *errno* being set by functions called from inside the signal handler.

RETURN VALUES

The **sigaction**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

EXAMPLES

There are three possible prototypes the handler may match:

ANSI C:

void **handler**(*int*);

Traditional BSD style:

void **handler**(*int*, *int code*, *struct sigcontext *scp*);

POSIX SA_SIGINFO:

void **handler**(*int*, *siginfo_t *info*, *ucontext_t *uap*);

The handler function should match the SA_SIGINFO prototype if the SA_SIGINFO bit is set in *sa_flags*. It then should be pointed to by the *sa_sigaction* member of *struct sigaction*. Note that you should not assign SIG_DFL or SIG_IGN this way.

If the SA_SIGINFO flag is not set, the handler function should match either the ANSI C or traditional BSD prototype and be pointed to by the *sa_handler* member of *struct sigaction*. In practice, FreeBSD always sends the three arguments of the latter and since the ANSI C prototype is a subset, both will work. The *sa_handler* member declaration in FreeBSD include files is that of ANSI C (as required by

POSIX), so a function pointer of a BSD-style function needs to be casted to compile without warning. The traditional BSD style is not portable and since its capabilities are a full subset of a SA_SIGINFO handler, its use is deprecated.

The *sig* argument is the signal number, one of the SIG... values from <*signal.h*>.

The *code* argument of the BSD-style handler and the *si_code* member of the *info* argument to a SA_SIGINFO handler contain a numeric code explaining the cause of the signal, usually one of the SI_... values from <*sys/signal.h*> or codes specific to a signal, i.e., one of the FPE_... values for SIGFPE.

The *scp* argument to a BSD-style handler points to an instance of *struct sigcontext*.

The *uap* argument to a POSIX SA_SIGINFO handler points to an instance of ucontext_t.

ERRORS

The **sigaction**() system call will fail and no new signal handler will be installed if one of the following occurs:

[EINVAL] The *sig* argument is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

kill(1), kill(2), ptrace(2), setitimer(2), setrlimit(2), sigaltstack(2), sigpending(2), sigprocmask(2), sigsuspend(2), wait(2), fpsetmask(3), setjmp(3), siginfo(3), siginterrupt(3), sigsetops(3), ucontext(3), tty(4)

STANDARDS

The **sigaction**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1"). The SA_ONSTACK and SA_RESTART flags are Berkeley extensions, as are the signals, SIGTRAP, SIGEMT, SIGBUS, SIGSYS, SIGURG, SIGIO, SIGXCPU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGWINCH, and SIGINFO. Those signals are available on most BSD-derived systems. The SA_NODEFER and SA_RESETHAND flags are intended for backwards compatibility with other operating systems. The SA_NOCLDSTOP, and SA_NOCLDWAIT flags are featuring options commonly found in other operating systems. The flags are approved by Version 2 of the Single UNIX Specification ("SUSv2"), along with the option to avoid zombie creation by ignoring SIGCHLD.

NAME

sigemptyset, **sigfillset**, **sigaddset**, **sigdelset**, **sigismember** - manipulate signal sets

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigemptyset(*sigset_t *set*);

int
sigfillset(*sigset_t *set*);

int
sigaddset(*sigset_t *set*, *int signo*);

int
sigdelset(*sigset_t *set*, *int signo*);

int
sigismember(*const sigset_t *set*, *int signo*);

DESCRIPTION

These functions manipulate signal sets stored in a *sigset_t*. Either **sigemptyset**() or **sigfillset**() must be called for every object of type *sigset_t* before any other use of the object.

The **sigemptyset**() function initializes a signal set to be empty.

The **sigfillset**() function initializes a signal set to contain all signals.

The **sigaddset**() function adds the specified signal *signo* to the signal set.

The **sigdelset**() function deletes the specified signal *signo* from the signal set.

The **sigismember**() function returns whether a specified signal *signo* is contained in the signal set.

RETURN VALUES

The **sigismember**() function returns 1 if the signal is a member of the set, 0 otherwise. The other

functions return 0 upon success. A -1 return value indicates an error occurred and the global variable *errno* is set to indicate the reason.

ERRORS

These functions could fail if one of the following occurs:

[EINVAL] *signo* has an invalid value.

SEE ALSO

kill(2), sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2)

STANDARDS

These functions are defined by IEEE Std 1003.1-1988 ("POSIX.1").

NAME

sigaltstack - set and/or get signal stack context

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <signal.h>

typedef struct {
        char    *ss_sp;
        size_t  ss_size;
        int     ss_flags;
} stack_t;
int
sigaltstack(const stack_t * restrict ss, stack_t * restrict oss);
```

DESCRIPTION

The **sigaltstack**() system call allows defining an alternate stack on which signals are to be processed for the current thread. If *ss* is non-zero, it specifies a pointer to and the size of a *signal stack* on which to deliver signals. When a signal's action indicates its handler should execute on the signal stack (specified with a sigaction(2) system call), the system checks to see if the thread is currently executing on that stack. If the thread is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution.

An active stack cannot be modified.

If SS_DISABLE is set in *ss_flags*, *ss_sp* and *ss_size* are ignored and the signal stack will be disabled. A disabled stack will cause all signals to be taken on the regular user stack. If the stack is later re-enabled then all signals that were specified to be processed on an alternate stack will resume doing so.

If *oss* is non-zero, the current signal stack state is returned. The *ss_flags* field will contain the value SS_ONSTACK if the thread is currently on a signal stack and SS_DISABLE if the signal stack is currently disabled.

NOTES

The value SIGSTKSZ is defined to be the number of bytes/chars that would be used to cover the usual case when allocating an alternate stack area. The following code fragment is typically used to allocate an alternate stack.

```
if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
        /* error return */
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, NULL) < 0)
        perror("sigaltstack");
```

An alternative approach is provided for programs with signal handlers that require a specific amount of stack space other than the default size. The value MINSIGSTKSZ is defined to be the number of bytes/chars that is required by the operating system to implement the alternate stack feature. In computing an alternate stack size, programs should add MINSIGSTKSZ to their stack requirements to allow for the operating system overhead.

Signal stacks are automatically adjusted for the direction of stack growth and alignment requirements. Signal stacks may or may not be protected by the hardware and are not ‘‘grown’’ automatically as is done for the normal stack. If the stack overflows and this space is not protected unpredictable results may occur.

RETURN VALUES

The **sigaltstack**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sigaltstack**() system call will fail and the signal stack context will remain unchanged if one of the following occurs.

[EFAULT]	Either *ss* or *oss* points to memory that is not a valid part of the process address space.
[EPERM]	An attempt was made to modify an active stack.
[EINVAL]	The *ss_flags* field was invalid.
[ENOMEM]	Size of alternate stack area is less than or equal to MINSIGSTKSZ.

SEE ALSO

sigaction(2), setjmp(3)

HISTORY

The predecessor to **sigaltstack**(), the **sigstack**() system call, appeared in 4.2BSD.

NAME

sigsetmask, **sigblock** - manipulate current signal mask

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigsetmask(*int mask*);

int
sigblock(*int mask*);

int
sigmask(*int signum*);

DESCRIPTION

This interface is made obsolete by: sigprocmask(2).

The **sigsetmask**() function sets the current signal mask to the specified *mask*. Signals are blocked from delivery if the corresponding bit in *mask* is a 1. The **sigblock**() function adds the signals in the specified *mask* to the current signal mask, rather than overwriting it as **sigsetmask**() does. The macro **sigmask**() is provided to construct the mask for a given *signum*.

The system quietly disallows SIGKILL or SIGSTOP to be blocked.

RETURN VALUES

The **sigblock**() and **sigsetmask**() functions return the previous set of masked signals.

SEE ALSO

kill(2), sigaction(2), sigprocmask(2), sigsuspend(2), sigvec(2), sigsetops(3)

HISTORY

The **sigsetmask**() and **sigblock**() functions first appeared in 4.2BSD and have been deprecated.

NAME

sighold, **sigignore**, **sigpause**, **sigrelse**, **sigset** - legacy interface for signal management

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sighold(*int sig*);

int
sigignore(*int sig*);

int
xsi_sigpause(*int sigmask*);

int
sigrelse(*int sig*);

void (*)(int)
sigset(*int*, *void (*disp)(int)*);

int
sigpause(*int sigmask*);

DESCRIPTION

This interface is made obsolete by sigsuspend(2) **and** sigaction(2).

The **sigset**() function modifies signal dispositions. The *sig* argument specifies the signal, which may be any signal except SIGKILL and SIGSTOP. The *disp* argument specifies the signal's disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If **sigset**() is used, and *disp* is the address of a signal handler, the system adds *sig* to the signal mask of the calling process before executing the signal handler; when the signal handler returns, the system restores the signal mask of the calling process to its state prior to the delivery of the signal. In addition, if **sigset**() is used, and *disp* is equal to SIG_HOLD, *sig* is added to the signal mask of the calling process and *sig 's* disposition remains unchanged. If **sigset**() is used, and *disp* is not equal to SIG_HOLD, *sig* is removed from the signal mask of the calling process.

The **sighold**() function adds *sig* to the signal mask of the calling process.

The **sigrelse**() function removes *sig* from the signal mask of the calling process.

The **sigignore**() function sets the disposition of *sig* to SIG_IGN.

The **xsi_sigpause**() function removes *sig* from the signal mask of the calling process and suspend the calling process until a signal is received. The **xsi_sigpause**() function restores the signal mask of the process to its original state before returning.

The **sigpause**() function assigns *sigmask* to the set of masked signals and then waits for a signal to arrive; on return the set of masked signals is restored. The *sigmask* argument is usually 0 to indicate that no signals are to be blocked.

RETURN VALUES

The **sigpause**() and **xsi_sigpause**() functions always terminate by being interrupted, returning -1 with *errno* set to EINTR.

Upon successful completion, **sigset**() returns SIG_HOLD if the signal had been blocked and the signal's previous disposition if it had not been blocked. Otherwise, SIG_ERR is returned and *errno* set to indicate the error.

For all other functions, upon successful completion, 0 is returned. Otherwise, -1 is returned and *errno* is set to indicate the error:

[EINVAL] The *sig* argument is not a valid signal number.

[EINVAL] For **sigset**() and **sigignore**() functions, an attempt was made to catch or ignore SIGKILL or SIGSTOP.

SEE ALSO

kill(2), sigaction(2), sigblock(2), sigprocmask(2), sigsuspend(2), sigvec(2)

STANDARDS

The **sigpause**() function is implemented for compatibility with historic 4.3BSD applications. An incompatible interface by the same name, which used a single signal number rather than a mask, was present in AT&T System V UNIX, and was copied from there into the **X/Open System Interfaces** (XSI) option of IEEE Std 1003.1-2001 ("POSIX.1"). FreeBSD implements it under the name **xsi_sigpause**(). The **sighold**(), **sigignore**(), **sigrelse**() and **sigset**() functions are implemented for compatibility with **System V** and **XSI** interfaces.

HISTORY

The **sigpause**() function appeared in 4.2BSD and has been deprecated. All other functions appeared in FreeBSD 8.1 and were deprecated before being implemented.

NAME

siginterrupt - allow signals to interrupt system calls

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
siginterrupt(*int sig*, *int flag*);

DESCRIPTION

The **siginterrupt**() function is used to change the system call restart behavior when a system call is interrupted by the specified signal. If the flag is false (0), then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart has been the default behavior since 4.2BSD, and is the default behaviour for signal(3) on FreeBSD.

If the flag is true (1), then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return -1 with the global variable *errno* set to EINTR. Interrupted system calls that have started transferring data will return the amount of data actually transferred. System call interrupt is the signal behavior found on 4.1BSD and AT&T System V UNIX systems.

Note that the new 4.2BSD signal handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent sigaction(2) call, and the signal mask operates as documented in sigaction(2). Programs may switch between restartable and interruptible system call operation as often as desired in the execution of a program.

Issuing a **siginterrupt**(*3*) call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

NOTES

This library routine uses an extension of the sigaction(2) system call that is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

RETURN VALUES

The **siginterrupt**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **siginterrupt**() call fails if:

[EINVAL] The *sig* argument is not a valid signal number.

SEE ALSO

sigaction(2), sigprocmask(2), sigsuspend(2), signal(3)

HISTORY

The **siginterrupt**() function appeared in 4.3BSD.

NAME

signal - simplified software signal facilities

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

void
(*signal(*int sig*, *void (*func)(int)))(*int*);

or in FreeBSD's equivalent but easier to read typedef'd version:
*typedef void (*sig_t) (int)*;

sig_t
signal(*int sig*, *sig_t func*);

DESCRIPTION

This **signal**() facility is a simplified interface to the more general sigaction(2) facility.

Signals allow the manipulation of a process from outside its domain as well as allowing the process to manipulate itself or copies of itself (children). There are two general types of signals: those that cause termination of a process and those that do not. Signals which cause termination of a program might result from an irrecoverable error or might be the result of a user at a terminal typing the 'interrupt' character. Signals are used when a process is stopped because it wishes to access its control terminal while in the background (see tty(4)). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals result in the termination of the process receiving them if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the **signal**() function allows for a signal to be caught, to be ignored, or to generate an interrupt. These signals are defined in the file <*signal.h*>:

Num

	Name	**Default Action**	**Description**
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction

5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2
32	SIGTHR	terminate process	thread interrupt
33	SIGLIBRT	terminate process	real-time library interrupt

The *sig* argument specifies which signal was received. The *func* procedure allows a user to choose the action upon receipt of a signal. To set the default action of the signal to occur as listed above, *func* should be SIG_DFL. A SIG_DFL resets the default action. To ignore the signal *func* should be SIG_IGN. This will cause subsequent instances of the signal to be ignored and pending instances to be discarded. If SIG_IGN is not used, further occurrences of the signal are automatically blocked and *func* is called.

The handled signal is unblocked when the function returns and the process continues from where it left off when the signal occurred. **Unlike previous signal facilities, the handler func() remains installed after a signal has been delivered.**

For some system calls, if a signal is caught while the call is executing and the call is prematurely terminated, the call is automatically restarted. Any handler installed with signal(3) will have the SA_RESTART flag set, meaning that any restartable system call will not return on receipt of a signal. The affected system calls include read(2), write(2), sendto(2), recvfrom(2), sendmsg(2) and recvmsg(2) on a communications channel or a low speed device and during a ioctl(2) or wait(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count). These semantics could be changed with siginterrupt(3).

When a process which has installed signal handlers forks, the child process inherits the signals. All caught signals may be reset to their default action by a call to the execve(2) function; ignored signals remain ignored.

If a process explicitly specifies SIG_IGN as the action for the signal SIGCHLD, the system will not create zombie processes when children of the calling process exit. As a consequence, the system will discard the exit status from the child processes. If the calling process subsequently issues a call to wait(2) or equivalent, it will block until all of the calling process's children terminate, and then return a value of -1 with *errno* set to ECHILD.

See sigaction(2) for a list of functions that are considered safe for use in signal handlers.

RETURN VALUES

The previous action is returned on a successful call. Otherwise, SIG_ERR is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **signal**() function will fail and no action will take place if one of the following occur:

[EINVAL] The *sig* argument is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

kill(1), kill(2), ptrace(2), sigaction(2), sigaltstack(2), sigprocmask(2), sigsuspend(2), wait(2), fpsetmask(3), setjmp(3), siginterrupt(3), tty(4)

HISTORY

The **signal**() function appeared in Version 4 AT&T UNIX. The current **signal** facility appeared in 4.0BSD. The option to avoid the creation of child zombies through ignoring SIGCHLD appeared in FreeBSD 5.0.

NAME

sigpending - get pending signals

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigpending(*sigset_t *set*);

DESCRIPTION

The **sigpending**() system call returns a mask of the signals pending for delivery to the calling thread or the calling process in the location indicated by *set*. Signals may be pending because they are currently masked, or transiently before delivery (although the latter case is not normally detectable).

RETURN VALUES

The **sigpending**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sigpending**() system call will fail if:

[EFAULT] The *set* argument specified an invalid address.

SEE ALSO

sigaction(2), sigprocmask(2), sigsuspend(2), sigsetops(3)

STANDARDS

The **sigpending**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

NAME

sigprocmask - manipulate current signal mask

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigprocmask(*int how*, *const sigset_t * restrict set*, *sigset_t * restrict oset*);

DESCRIPTION

The **sigprocmask**() system call examines and/or changes the current signal mask (those signals that are blocked from delivery). Signals are blocked if they are members of the current signal mask set.

If *set* is not null, the action of **sigprocmask**() depends on the value of the *how* argument. The signal mask is changed as a function of the specified *set* and the current mask. The function is specified by *how* using one of the following values from <*signal.h*>:

SIG_BLOCK The new mask is the union of the current mask and the specified *set*.

SIG_UNBLOCK The new mask is the intersection of the current mask and the complement of the specified *set*.

SIG_SETMASK The current mask is replaced by the specified *set*.

If *oset* is not null, it is set to the previous value of the signal mask. When *set* is null, the value of *how* is insignificant and the mask remains unset providing a way to examine the signal mask without modification.

The system quietly disallows SIGKILL or SIGSTOP to be blocked.

In threaded applications, pthread_sigmask(3) must be used instead of **sigprocmask**().

RETURN VALUES

The **sigprocmask**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sigprocmask**() system call will fail and the signal mask will be unchanged if one of the following occurs:

[EINVAL] The *how* argument has a value other than those listed here.

SEE ALSO

kill(2), sigaction(2), sigpending(2), sigsuspend(2), fpsetmask(3), pthread_sigmask(3), sigsetops(3)

STANDARDS

The **sigprocmask**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

NAME

sigqueue - queue a signal to a process (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigqueue(*pid_t pid*, *int signo*, *const union sigval value*);

DESCRIPTION

The **sigqueue**() system call causes the signal specified by *signo* to be sent with the value specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of PID.

The conditions required for a process to have permission to queue a signal to another process are the same as for the kill(2) system call. The **sigqueue**() system call queues a signal to a single process specified by the *pid* argument.

The **sigqueue**() system call returns immediately. If the resources were available to queue the signal, the signal will be queued and sent to the receiving process.

If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a **sigwait**() system call for *signo*, either *signo* or at least the pending, unblocked signal will be delivered to the calling thread before **sigqueue**() returns. Should any multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected for delivery, it is the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sigqueue**() system call will fail if:

[EAGAIN] No resources are available to queue the signal. The process has already queued {SIGQUEUE_MAX} signals that are still pending at the receiver(s), or a system-

wide resource limit has been exceeded.

[EINVAL] The value of the *signo* argument is an invalid or unsupported signal number.

[EPERM] The process does not have the appropriate privilege to send the signal to the receiving process.

[ESRCH] The process *pid* does not exist.

SEE ALSO

kill(2), sigaction(2), sigpending(2), sigsuspend(2), sigtimedwait(2), sigwait(2), sigwaitinfo(2), pause(3), pthread_sigmask(3), siginfo(3)

STANDARDS

The **sigqueue**() system call conforms to IEEE Std 1003.1-2004 ("POSIX.1").

HISTORY

Support for POSIX realtime signal queue first appeared in FreeBSD 7.0.

CAVEATS

When using **sigqueue** to send signals to a process which might have a different ABI (for instance, one is 32-bit and the other 64-bit), the *sival_int* member of *value* can be delivered reliably, but the *sival_ptr* may be truncated in endian dependent ways and must not be relied on. Further, many pointer integrity schemes disallow sending pointers to other processes, and this technique should not be used in programs intended to be portable.

NAME

sigreturn - return from signal

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigreturn(*const ucontext_t *scp*);

DESCRIPTION

The **sigreturn**() system call allows users to atomically unmask, switch stacks, and return from a signal context. The thread's signal mask and stack status are restored from the context structure pointed to by *scp*. The system call does not return; the users stack pointer, frame pointer, argument pointer, and processor status longword are restored from the context. Execution resumes at the specified pc. This system call is used by the trampoline code and longjmp(3) when returning from a signal to the previously executing program.

RETURN VALUES

If successful, the system call does not return. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The **sigreturn**() system call will fail and the thread context will remain unchanged if one of the following occurs.

[EFAULT]	The *scp* argument points to memory that is not a valid part of the process address space.
[EINVAL]	The process status longword is invalid or would improperly raise the privilege level of the process.

SEE ALSO

sigaction(2), setjmp(3), ucontext(3)

HISTORY

The **sigreturn**() system call appeared in 4.3BSD.

NAME

sigstack - set and/or get signal stack context

LIBRARY

Standard C Library (libc, -lc)

DESCRIPTION

The **sigstack**() function has been deprecated in favor of the interface described in sigaltstack(2).

SEE ALSO

sigaltstack(2)

HISTORY

The **sigstack**() system call appeared in 4.2BSD.

NAME

sigsuspend - atomically release blocked signals and wait for interrupt

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigsuspend(*const sigset_t *sigmask*);

DESCRIPTION

The **sigsuspend**() system call temporarily changes the blocked signal mask to the set to which *sigmask* points, and then waits for a signal to arrive; on return the previous set of masked signals is restored. The signal mask set is usually empty to indicate that all signals are to be unblocked for the duration of the call.

In normal usage, a signal is blocked using sigprocmask(2) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using **sigsuspend**() with the previous mask returned by sigprocmask(2).

RETURN VALUES

The **sigsuspend**() system call always terminates by being interrupted, returning -1 with *errno* set to EINTR.

SEE ALSO

pselect(2), sigaction(2), sigpending(2), sigprocmask(2), sigtimedwait(2), sigwait(2), sigwaitinfo(2), sigsetops(3)

STANDARDS

The **sigsuspend**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

NAME

sigtimedwait, **sigwaitinfo** - wait for queued signals (REALTIME)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigtimedwait(*const sigset_t *restrict set*, *siginfo_t *restrict info*, *const struct timespec *restrict timeout*);

int
sigwaitinfo(*const sigset_t * restrict set*, *siginfo_t * restrict info*);

DESCRIPTION

The **sigtimedwait**() system call is equivalent to **sigwaitinfo**() except that if none of the signals specified by *set* are pending, **sigtimedwait**() waits for the time interval specified in the *timespec* structure referenced by *timeout*. If the *timespec* structure pointed to by *timeout* is zero-valued and if none of the signals specified by *set* are pending, then **sigtimedwait**() returns immediately with an error. If *timeout* is the NULL pointer, the behavior is unspecified. CLOCK_MONOTONIC clock is used to measure the time interval specified by the *timeout* argument.

The **sigwaitinfo**() system call selects the pending signal from the set specified by *set*. Should any of multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at the time of the call, the calling thread is suspended until one or more signals in *set* become pending or until it is interrupted by an unblocked, caught signal.

The **sigwaitinfo**() system call is equivalent to the **sigwait**() system call if the *info* argument is NULL. If the *info* argument is non-NULL, the **sigwaitinfo**() function is equivalent to **sigwait**(), except that the selected signal number shall be stored in the *si_signo* member, and the cause of the signal shall be stored in the *si_code* member. Besides this, the **sigwaitinfo**() and **sigtimedwait**() system calls may return EINTR if interrupted by signal, which is not allowed for the **sigwait**() function.

If any value is queued to the selected signal, the first such queued value is dequeued and, if the info argument is non-NULL, the value is stored in the *si_value* member of *info*. The system resource used to queue the signal is released and returned to the system for other use. If no value is queued, the content of the *si_value* member is zero-valued. If no further signals are queued for the selected signal, the

pending indication for that signal is reset.

RETURN VALUES

Upon successful completion (that is, one of the signals specified by *set* is pending or is generated) **sigwaitinfo**() and **sigtimedwait**() return the selected signal number. Otherwise, the functions return a value of -1 and set the global variable *errno* to indicate the error.

ERRORS

The **sigtimedwait**() system call will fail if:

[EAGAIN] No signal specified by set was generated within the specified timeout period.

The **sigtimedwait**() and **sigwaitinfo**() system calls fail if:

[EINTR] The wait was interrupted by an unblocked, caught signal.

The **sigtimedwait**() system call may also fail if:

[EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than or equal to 1000 million. Kernel only checks for this error if no signal is pending in set and it is necessary to wait.

SEE ALSO

sigaction(2), sigpending(2), sigqueue(2), sigsuspend(2), sigwait(2), pause(3), pthread_sigmask(3), siginfo(3)

STANDARDS

The **sigtimedwait**() and **sigwaitinfo**() system calls conform to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

sigvec - software signal facilities

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <signal.h>

struct sigvec {
        void    (*sv_handler)();
                int     sv_mask;
                int     sv_flags;
};
int
sigvec(int sig, struct sigvec *vec, struct sigvec *ovec);
```

DESCRIPTION

This interface is made obsolete by sigaction(2).

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a sigblock(2) or sigsetmask(2) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore

the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a sigblock(2) or sigsetmask(2) call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and *or*'ing in the signal mask associated with the handler to be invoked.

The **sigvec**() function assigns a handler for a specific signal. If *vec* is non-zero, it specifies a handler routine and mask to be used when delivering the specified signal. Further, if the SV_ONSTACK bit is set in *sv_flags*, the system will deliver the signal to the process on a *signal stack*, specified with sigaltstack(2). If *ovec* is non-zero, the previous handling information for the signal is returned to the user.

The following is a list of all signals with names as in the include file <*signal.h*>:

NAME	Default Action	Description
SIGHUP	terminate process	terminal line hangup
SIGINT	terminate process	interrupt program
SIGQUIT	create core image	quit program
SIGILL	create core image	illegal instruction
SIGTRAP	create core image	trace trap
SIGABRT	create core image	abort(3) call (formerly SIGIOT)
SIGEMT	create core image	emulate instruction executed
SIGFPE	create core image	floating-point exception
SIGKILL	terminate process	kill program
SIGBUS	create core image	bus error
SIGSEGV	create core image	segmentation violation
SIGSYS	create core image	non-existent system call invoked
SIGPIPE	terminate process	write on a pipe with no reader
SIGALRM	terminate process	real-time timer expired
SIGTERM	terminate process	software termination signal
SIGURG	discard signal	urgent condition present on socket
SIGSTOP	stop process	stop (cannot be caught or ignored)
SIGTSTP	stop process	stop signal generated from keyboard
SIGCONT	discard signal	continue after stop
SIGCHLD	discard signal	child status has changed
SIGTTIN	stop process	background read attempted from control terminal
SIGTTOU	stop process	background write attempted to control terminal
SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
Dv SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))

Dv SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
Dv SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
Dv SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
Dv SIGWINCH	discard signal	Window size change
SIGINFO	discard signal	status request from keyboard
SIGUSR1	terminate process	User defined signal 1
SIGUSR2	terminate process	User defined signal 2

Once a signal handler is installed, it remains installed until another **sigvec**() call is made, or an execve(2) is performed. A signal-specific default action may be reset by setting *sv_handler* to SIG_DFL. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. See the above signal list for each signal's default action. If *sv_handler* is SIG_IGN current and pending instances of the signal are ignored and discarded.

If a signal is caught during the system calls listed below, the call is normally restarted. The call can be forced to terminate prematurely with an EINTR error return by setting the SV_INTERRUPT bit in *sv_flags*. The affected system calls include read(2), write(2), sendto(2), recvfrom(2), sendmsg(2) and recvmsg(2) on a communications channel or a slow device (such as a terminal, but not a regular file) and during a wait(2) or ioctl(2). However, calls that have already committed are not restarted, but instead return a partial success (for example, a short read count).

After a fork(2) or vfork(2) all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child.

The execve(2) system call reinstates the default action for all signals which were caught and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that interrupt system calls continue to do so.

NOTES

The mask specified in *vec* is not allowed to block SIGKILL or SIGSTOP. This is done silently by the system.

The SV_INTERRUPT flag is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

RETURN VALUES

The **sigvec**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

EXAMPLES

On the VAX-11 The handler routine can be declared:

```
void handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. The *code* argument is either a constant as given below or, for compatibility mode faults, the code provided by the hardware (Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl). The *scp* argument is a pointer to the *sigcontext* structure (defined in <*signal.h*>), used to restore the context from before the signal.

ERRORS

The **sigvec**() function will fail and no new signal handler will be installed if one of the following occurs:

[EFAULT]	Either *vec* or *ovec* points to memory that is not a valid part of the process address space.
[EINVAL]	The *sig* argument is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

kill(1), kill(2), ptrace(2), sigaction(2), sigaltstack(2), sigblock(2), sigpause(2), sigprocmask(2), sigsetmask(2), sigsuspend(2), setjmp(3), siginterrupt(3), signal(3), sigsetops(3), tty(4)

BUGS

This manual page is still confusing.

NAME

sigwait - select a set of signals

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <signal.h>

int
sigwait(*const sigset_t * restrict set*, *int * restrict sig*);

DESCRIPTION

The **sigwait**() system call selects a set of signals, specified by *set*. If none of the selected signals are pending, **sigwait**() waits until one or more of the selected signals has been generated. Then **sigwait**() atomically clears one of the selected signals from the set of pending signals (for the process or for the current thread) and sets the location pointed to by *sig* to the signal number that was cleared.

The signals specified by *set* should be blocked at the time of the call to **sigwait**().

If more than one thread is using **sigwait**() to wait for the same signal, no more than one of these threads will return from **sigwait**() with the signal number. If more than a single thread is blocked in **sigwait**() for a signal when that signal is generated for the process, it is unspecified which of the waiting threads returns from **sigwait**(). If the signal is generated for a specific thread, as by **pthread_kill**(), only that thread will return.

Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it will be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

IMPLEMENTATION NOTES

The **sigwait**() function is implemented as a wrapper around the **__sys_sigwait**() system call, which retries the call on EINTR error.

RETURN VALUES

If successful, **sigwait**() returns 0 and sets the location pointed to by *sig* to the cleared signal number. Otherwise, an error number is returned.

ERRORS

The **sigwait**() system call will fail if:

[EINVAL] The *set* argument specifies one or more invalid signal numbers.

SEE ALSO

sigaction(2), sigpending(2), sigqueue(2), sigsuspend(2), sigtimedwait(2), sigwaitinfo(2), pause(3), pthread_sigmask(3)

STANDARDS

The **sigwait**() function conforms to ISO/IEC 9945-1:1996 ("POSIX.1").

NAME

stringlist, **sl_init**, **sl_add**, **sl_free**, **sl_find** - stringlist manipulation functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stringlist.h>

*StringList **
sl_init();

int
sl_add(*StringList *sl*, *char *item*);

void
sl_free(*StringList *sl*, *int freeall*);

*char **
sl_find(*StringList *sl*, *const char *item*);

DESCRIPTION

The **stringlist** functions manipulate stringlists, which are lists of strings that extend automatically if necessary.

The *StringList* structure has the following definition:

```
typedef struct _stringlist {
        char        **sl_str;
        size_t        sl_max;
        size_t        sl_cur;
} StringList;
```

sl_str a pointer to the base of the array containing the list.

sl_max
the size of *sl_str*.

sl_cur
the offset in *sl_str* of the current element.

The following stringlist manipulation functions are available:

sl_init() Create a stringlist. Returns a pointer to a *StringList*, or NULL in case of failure.

sl_free()

Releases memory occupied by *sl* and the *sl->sl_str* array. If *freeall* is non-zero, then each of the items within *sl->sl_str* is released as well.

sl_add()

Add *item* to *sl->sl_str* at *sl->sl_cur*, extending the size of *sl->sl_str*. Returns zero upon success, -1 upon failure.

sl_find()

Find *item* in *sl*, returning NULL if it is not found.

SEE ALSO

free(3), malloc(3)

HISTORY

The **stringlist** functions appeared in FreeBSD 2.2.6 and NetBSD 1.3.

NAME

sleep - suspend thread execution for an interval measured in seconds

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

unsigned int
sleep(*unsigned int seconds*);

DESCRIPTION

The **sleep**() function suspends execution of the calling thread until either *seconds* seconds have elapsed or a signal is delivered to the thread and its action is to invoke a signal-catching function or to terminate the thread or process. System activity may lengthen the sleep by an indeterminate amount.

This function is implemented using nanosleep(2) by pausing for *seconds* seconds or until a signal occurs. Consequently, in this implementation, sleeping has no effect on the state of process timers, and there is no special handling for SIGALRM.

RETURN VALUES

If the **sleep**() function returns because the requested time has elapsed, the value returned will be zero. If the **sleep**() function returns due to the delivery of a signal, the value returned will be the unslept amount (the requested time minus the time actually slept) in seconds.

SEE ALSO

nanosleep(2), usleep(3)

STANDARDS

The **sleep**() function conforms to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

A **sleep**() function appeared in Version 7 AT&T UNIX.

NAME

sockatmark - determine whether the read pointer is at the OOB mark

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

int
sockatmark(*int s*);

DESCRIPTION

To find out if the read pointer is currently pointing at the mark in the data stream, the **sockatmark**() function is provided. If **sockatmark**() returns 1, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown below. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

```
#include <sys/socket.h>
...
oob()
{
        int out = FWRITE, mark;
        char waste[BUFSIZ];

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if ((mark = sockatmark(rem)) < 0) {
                        perror("sockatmark");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
```

```
                ...
        }
        ...
}
```

RETURN VALUES

Upon successful completion, the **sockatmark**() function returns the value 1 if the read pointer is pointing at the OOB mark, 0 if it is not. Otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **sockatmark**() call fails if:

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

[ENOTTY] The *s* argument is a descriptor for a file, not a socket.

SEE ALSO

recv(2), send(2)

HISTORY

The **sockatmark**() function was introduced by IEEE Std 1003.1-2001 ("POSIX.1"), to standardize the historical SIOCATMARK ioctl(2). The ENOTTY error instead of the usual ENOTSOCK is to match the historical behavior of SIOCATMARK.

NAME

socket - create an endpoint for communication

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/socket.h>

int
socket(*int domain*, *int type*, *int protocol*);

DESCRIPTION

The **socket**() system call creates an endpoint for communication and returns a descriptor.

The *domain* argument specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file <*sys/socket.h*>. The currently understood formats are:

PF_LOCAL	Host-internal protocols (alias for PF_UNIX),
PF_UNIX	Host-internal protocols,
PF_INET	Internet version 4 protocols,
PF_INET6	Internet version 6 protocols,
PF_ROUTE	Internal routing protocol,
PF_LINK	Link layer interface,
PF_KEY	Internal key-management function,
PF_NATM	Asynchronous transfer mode protocols,
PF_NETGRAPH	Netgraph sockets,
PF_IEEE80211	IEEE 802.11 wireless link-layer protocols (WiFi),
PF_BLUETOOTH	Bluetooth protocols,
PF_INET_SDP	OFED socket direct protocol (IPv4),
PF_INET6_SDP	OFED socket direct protocol (IPv6)

Each protocol family is connected to an address family, which has the same name except that the prefix is "AF_" in place of "PF_". Other protocol families may be also defined, beginning with "PF_", with corresponding address families.

The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM Stream socket,
SOCK_DGRAM Datagram socket,
SOCK_RAW Raw-protocol interface,
SOCK_RDM Reliably-delivered packet,
SOCK_SEQPACKET Sequenced packet stream

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility may have protocol-specific properties. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

Additionally, the following flags are allowed in the *type* argument:

SOCK_CLOEXEC Set close-on-exec on the new descriptor,
SOCK_NONBLOCK Set non-blocking mode on the new socket

The *protocol* argument specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see protocols(5).

The *protocol* argument may be set to zero (0) to request the default implementation of a socket type for the protocol, if any.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a connect(2) system call. Once connected, data may be transferred using read(2) and write(2) calls or some variant of the send(2) and recv(2) functions. (Some protocol families, such as the Internet family, support the notion of an "implied connect", which permits data to be sent piggybacked onto a connect operation by using the sendto(2) system call.) When a session has been completed a close(2) may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or

duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable *errno*. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (e.g. 5 minutes). By default, a SIGPIPE signal is raised if a process sends on a broken stream, but this behavior may be inhibited via setsockopt(2).

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send(2) calls. Datagrams are generally received with recvfrom(2), which returns the next datagram with its return address.

An fcntl(2) system call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<sys/socket.h>*. The setsockopt(2) and getsockopt(2) system calls are used to set and get options, respectively.

RETURN VALUES

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The **socket**() system call fails if:

[EACCES] Permission to create a socket of the specified type and/or protocol is denied.

[EAFNOSUPPORT] The address family (domain) is not supported or the specified domain is not supported by this protocol family.

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[ENOBUFS] Insufficient buffer space is available. The socket cannot be created until sufficient

resources are freed.

[EPERM] User has insufficient privileges to carry out the requested operation.

[EPROTONOSUPPORT]
The protocol type or the specified protocol is not supported within this domain.

[EPROTOTYPE] The socket type is not supported by the protocol.

SEE ALSO

accept(2), bind(2), connect(2), getpeername(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2), CMSG_DATA(3), getprotoent(3), netgraph(4), protocols(5)

"An Introductory 4.3 BSD Interprocess Communication Tutorial", *PS1*, 7.

"BSD Interprocess Communication Tutorial", *PS1*, 8.

STANDARDS

The **socket**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1"). The POSIX standard specifies only the AF_INET, AF_INET6, and AF_UNIX constants for address families, and requires the use of AF_* constants for the *domain* argument of **socket**(). The SOCK_CLOEXEC flag is expected to conform to the next revision of the POSIX standard. The SOCK_RDM *type*, the PF_* constants, and other address families are FreeBSD extensions.

HISTORY

The **socket**() system call appeared in 4.2BSD.

NAME

socketpair - create a pair of connected sockets

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

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

int
socketpair(*int domain*, *int type*, *int protocol*, *int *sv*);

DESCRIPTION

The **socketpair**() system call creates an unnamed pair of connected sockets in the specified communications *domain*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

The SOCK_CLOEXEC and SOCK_NONBLOCK flags in the *type* argument apply to both descriptors.

RETURN VALUES

The **socketpair**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The call succeeds unless:

[EMFILE] Too many descriptors are in use by this process.

[EAFNOSUPPORT] The specified address family is not supported on this machine.

[EPROTONOSUPPORT]
The specified protocol is not supported on this machine.

[EOPNOTSUPP] The specified protocol does not support creation of socket pairs.

[EFAULT] The address *sv* does not specify a valid part of the process address space.

SEE ALSO

pipe(2), read(2), socket(2), write(2)

STANDARDS

The **socketpair**() system call conforms to IEEE Std 1003.1-2001 ("POSIX.1") and IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **socketpair**() system call appeared in 4.2BSD.

BUGS

This call is currently implemented only for the UNIX domain.

NAME

statfs - get file system statistics

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/mount.h>

int
statfs(*const char *path*, *struct statfs *buf*);

int
fstatfs(*int fd*, *struct statfs *buf*);

DESCRIPTION

The **statfs**() system call returns information about a mounted file system. The *path* argument is the path name of any file within the mounted file system. The *buf* argument is a pointer to a *statfs* structure defined as follows:

```
typedef struct fsid { int32_t val[2]; } fsid_t; /* file system id type */

/*
 * filesystem statistics
 */

#define    MFSNAMELEN    16                /* length of type name including null */
#define    MNAMELEN      1024              /* size of on/from name bufs */
#define    STATFS_VERSION          0x20140518          /* current version number */

struct statfs {
uint32_t f_version;             /* structure version number */
uint32_t f_type;                /* type of filesystem */
uint64_t f_flags;               /* copy of mount exported flags */
uint64_t f_bsize;               /* filesystem fragment size */
uint64_t f_iosize;              /* optimal transfer block size */
uint64_t f_blocks;              /* total data blocks in filesystem */
uint64_t f_bfree;               /* free blocks in filesystem */
int64_t     f_bavail;           /* free blocks avail to non-superuser */
```

```
uint64_t f_files;               /* total file nodes in filesystem */
int64_t     f_ffree;            /* free nodes avail to non-superuser */
uint64_t f_syncwrites;                  /* count of sync writes since mount */
uint64_t f_asyncwrites;                 /* count of async writes since mount */
uint64_t f_syncreads;                   /* count of sync reads since mount */
uint64_t f_asyncreads;                  /* count of async reads since mount */
uint64_t f_spare[10];                   /* unused spare */
uint32_t f_namemax;                     /* maximum filename length */
uid_t       f_owner;                    /* user that mounted the filesystem */
fsid_t      f_fsid;             /* filesystem id */
char        f_charspare[80];        /* spare string space */
char        f_fstypename[MFSNAMELEN]; /* filesystem type name */
char        f_mntfromname[MNAMELEN];  /* mounted filesystem */
char        f_mntonname[MNAMELEN];    /* directory on which mounted */
};
```

The flags that may be returned include:

MNT_RDONLY The file system is mounted read-only; Even the super-user may not write on it.

MNT_NOEXEC Files may not be executed from the file system.

MNT_NOSUID Setuid and setgid bits on files are not honored when they are executed.

MNT_SYNCHRONOUS All I/O to the file system is done synchronously.

MNT_ASYNC No file system I/O is done synchronously.

MNT_SOFTDEP Soft updates being done (see ffs(7)).

MNT_GJOURNAL Journaling with gjournal is enabled (see gjournal(8)).

MNT_SUIDDIR Special handling of SUID bit on directories.

MNT_UNION Union with underlying file system.

MNT_NOSYMFOLLOW
Symbolic links are not followed.

MNT_NOCLUSTERR	Read clustering is disabled.
MNT_NOCLUSTERW	Write clustering is disabled.
MNT_MULTILABEL	Mandatory Access Control (MAC) support for individual objects (see mac(4)).
MNT_ACLS	Access Control List (ACL) support enabled.
MNT_LOCAL	The file system resides locally.
MNT_QUOTA	The file system has quotas enabled on it.
MNT_ROOTFS	Identifies the root file system.
MNT_EXRDONLY	The file system is exported read-only.
MNT_NOATIME	Updating of file access times is disabled.
MNT_USER	The file system has been mounted by a user.
MNT_EXPORTED	The file system is exported for both reading and writing.
MNT_DEFEXPORTED	The file system is exported for both reading and writing to any Internet host.
MNT_EXPORTANON	The file system maps all remote accesses to the anonymous user.
MNT_EXKERB	The file system is exported with Kerberos uid mapping.
MNT_EXPUBLIC	The file system is exported publicly (WebNFS).

Fields that are undefined for a particular file system are set to -1. The **fstatfs**() system call returns the same information about an open file referenced by descriptor *fd*.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **statfs**() system call fails if one or more of the following are true:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[ENAMETOOLONG]
The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] The *buf* or *path* argument points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

The **fstatfs**() system call fails if one or more of the following are true:

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

[EFAULT] The *buf* argument points to an invalid address.

[EIO] An I/O error occurred while reading from or writing to the file system.

SEE ALSO

fhstatfs(2), getfsstat(2)

HISTORY

The **statfs**() system call first appeared in 4.4BSD.

NAME

stdio - standard input/output library functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*FILE *stdin*;
*FILE *stdout*;
*FILE *stderr*;

DESCRIPTION

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed. The functions and macros are listed below; more information is available from the individual man pages.

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded. If a file can support positioning requests (such as a disk file, as opposed to a terminal) then a *file position indicator* associated with the stream is positioned at the start of the file (byte zero), unless the file is opened with append mode. If append mode is used, the position indicator will be placed at the end-of-file. The position indicator is maintained by subsequent reads, writes and positioning requests. All input occurs as if the characters were read by successive calls to the fgetc(3) function; all output takes place as if all characters were written by successive calls to the fputc(3) function.

A file is disassociated from a stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transferred to the host environment) before the stream is disassociated from the file. The value of a pointer to a FILE object is indeterminate (garbage) after a file is closed.

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at the start). If the main function returns to its original caller, or the exit(3) function is called, all open files are closed (hence all output streams are flushed) before program termination. Other methods of program termination may not close files properly and hence buffered output may be lost. In particular, _exit(2) does not flush stdio files. Neither does an exit due to a signal. Buffers are flushed by abort(3) as required by POSIX, although previous implementations did not.

This implementation makes no distinction between "text" and "binary" streams. In effect, all streams are

binary. No translation is performed and no extra padding appears on any stream.

At program startup, three streams are predefined and need not be opened explicitly:

- *standard input* (for reading conventional input),
- *standard output* (for writing conventional output), and
- *standard error* (for writing diagnostic output).

These streams are abbreviated stdin, stdout and stderr. Initially, the standard error stream is unbuffered; the standard input and output streams are fully buffered if and only if the streams do not refer to an interactive or "terminal" device, as determined by the isatty(3) function. In fact, *all* freshly-opened streams that refer to terminal devices default to line buffering, and pending output to such streams is written automatically whenever such an input stream is read. Note that this applies only to "true reads"; if the read request can be satisfied by existing buffered data, no automatic flush will occur. In these cases, or when a large amount of computation is done after printing part of a line on an output terminal, it is necessary to fflush(3) the standard output before going off and computing so that the output will appear. Alternatively, these defaults may be modified via the setvbuf(3) function.

The **stdio** library is a part of the library **libc** and routines are automatically loaded as needed by the C compiler. The SYNOPSIS sections of the following manual pages indicate which include files are to be used, what the compiler declaration for the function looks like and which external variables are of interest.

The following are defined as macros; these names may not be re-used without first removing their current definitions with **#undef**: BUFSIZ, EOF, FILENAME_MAX, FOPEN_MAX, L_ctermid, L_cuserid, L_tmpnam, NULL, P_tmpdir, SEEK_CUR, SEEK_END, SEEK_SET, TMP_MAX, clearerr, clearerr_unlocked, feof, feof_unlocked, ferror, ferror_unlocked, fileno, fileno_unlocked, fropen, fwopen, getc, getc_unlocked, getchar, getchar_unlocked, putc, putc_unlocked, putchar, putchar_unlocked, stderr, stdin and stdout. Function versions of the macro functions clearerr, clearerr_unlocked, feof, feof_unlocked, ferror, ferror_unlocked, fileno, fileno_unlocked, getc, getc_unlocked, getchar, getchar_unlocked, putc, putc_unlocked, putchar, and putchar_unlocked exist and will be used if the macro definitions are explicitly removed.

SEE ALSO

close(2), open(2), read(2), write(2)

STANDARDS

The **stdio** library conforms to ISO/IEC 9899:1999 ("ISO C99").

LIST OF FUNCTIONS

Function	Description
asprintf	formatted output conversion

clearerr	check and reset stream status
dprintf	formatted output conversion
fclose	close a stream
fdopen	stream open functions
feof	check and reset stream status
ferror	check and reset stream status
fflush	flush a stream
fgetc	get next character or word from input stream
fgetln	get a line from a stream
fgetpos	reposition a stream
fgets	get a line from a stream
fgetwc	get next wide character from input stream
fgetws	get a line of wide characters from a stream
fileno	check and reset stream status
fopen	stream open functions
fprintf	formatted output conversion
fpurge	flush a stream
fputc	output a character or word to a stream
fputs	output a line to a stream
fputwc	output a wide character to a stream
fputws	output a line of wide characters to a stream
fread	binary stream input/output
freopen	stream open functions
fropen	open a stream
fscanf	input format conversion
fseek	reposition a stream
fsetpos	reposition a stream
ftell	reposition a stream
funopen	open a stream
fwide	set/get orientation of stream
fwopen	open a stream
fwprintf	formatted wide character output conversion
fwrite	binary stream input/output
getc	get next character or word from input stream
getchar	get next character or word from input stream
getdelim	get a line from a stream
getline	get a line from a stream
gets	get a line from a stream
getw	get next character or word from input stream
getwc	get next wide character from input stream

getwchar	get next wide character from input stream
mkdtemp	create unique temporary directory
mkstemp	create unique temporary file
mktemp	create unique temporary file
perror	system error messages
printf	formatted output conversion
putc	output a character or word to a stream
putchar	output a character or word to a stream
puts	output a line to a stream
putw	output a character or word to a stream
putwc	output a wide character to a stream
putwchar	output a wide character to a stream
remove	remove directory entry
rewind	reposition a stream
scanf	input format conversion
setbuf	stream buffering operations
setbuffer	stream buffering operations
setlinebuf	stream buffering operations
setvbuf	stream buffering operations
snprintf	formatted output conversion
sprintf	formatted output conversion
sscanf	input format conversion
strerror	system error messages
swprintf	formatted wide character output conversion
sys_errlist	system error messages
sys_nerr	system error messages
tempnam	temporary file routines
tmpfile	temporary file routines
tmpnam	temporary file routines
ungetc	un-get character from input stream
ungetwc	un-get wide character from input stream
vasprintf	formatted output conversion
vdprintf	formatted output conversion
vfprintf	formatted output conversion
vfscanf	input format conversion
vfwprintf	formatted wide character output conversion
vprintf	formatted output conversion
vscanf	input format conversion
vsnprintf	formatted output conversion
vsprintf	formatted output conversion

vsscanf	input format conversion
vswprintf	formatted wide character output conversion
vwprintf	formatted wide character output conversion
wprintf	formatted wide character output conversion

BUGS

The standard buffered functions do not interact well with certain other library and system functions, especially vfork(2).

NAME

stpcpy, **stpncpy**, **strcpy**, **strncpy** - copy strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

*char **
stpcpy(*char * restrict dst*, *const char * restrict src*);

*char **
stpncpy(*char * restrict dst*, *const char * restrict src*, *size_t len*);

*char **
strcpy(*char * restrict dst*, *const char * restrict src*);

*char **
strncpy(*char * restrict dst*, *const char * restrict src*, *size_t len*);

DESCRIPTION

The **strcpy**() and **stpcpy**() functions copy the string *src* to *dst* (including the terminating '\0' character.)

The **strncpy**() and **stpncpy**() functions copy at most *len* characters from *src* into *dst*. **If** *src* **is less than** *len* **characters long, the remainder of** *dst* **is filled with '\0' characters.** Otherwise, *dst* is *not* terminated.

For all of **strcpy**(), **strncpy**(), **stpcpy**(), and **stpncpy**(), the result is undefined if *src* and *dst* overlap.

RETURN VALUES

The **strcpy**() and **strncpy**() functions return *dst*. The **stpcpy**() and **stpncpy**() functions return a pointer to the terminating '\0' character of *dst*. If **stpncpy**() does not terminate *dst* with a NUL character, it instead returns a pointer to dst[n] (which does not necessarily refer to a valid memory location.)

EXAMPLES

The following sets *chararray* to "abc\0\0\0":

```
char chararray[6];

(void)strncpy(chararray, "abc", sizeof(chararray));
```

The following sets *chararray* to "abcdef":

```
char chararray[6];

(void)strncpy(chararray, "abcdefgh", sizeof(chararray));
```

Note that it does *not* NUL terminate *chararray* because the length of the source string is greater than or equal to the length argument.

The following copies as many characters from *input* to *buf* as will fit and NUL terminates the result. Because **strncpy**() does *not* guarantee to NUL terminate the string itself, this must be done explicitly.

```
char buf[1024];

(void)strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

This could be better achieved using strlcpy(3), as shown in the following example:

```
(void)strlcpy(buf, input, sizeof(buf));
```

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strlcpy(3), wcscpy(3)

STANDARDS

The **strcpy**() and **strncpy**() functions conform to ISO/IEC 9899:1990 ("ISO C90"). The **stpcpy**() and **stpncpy**() functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **stpcpy**() function first appeared in FreeBSD 4.4, and **stpncpy**() was added in FreeBSD 8.0.

SECURITY CONSIDERATIONS

All of the functions documented in this manual page are easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack.

It is strongly suggested that the **strlcpy**() function be used in almost all cases.

For some, but not all, fixed-length records, non-terminated strings may be both valid and desirable. In that specific case, the **strncpy**() function may be most sensible.

NAME

strcasecmp, **strncasecmp** - compare strings, ignoring case

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <strings.h>

int
strcasecmp(*const char *s1*, *const char *s2*);

int
strncasecmp(*const char *s1*, *const char *s2*, *size_t len*);

#include <strings.h>
#include <xlocale.h>

int
strcasecmp_l(*const char *s1*, *const char *s2*, *locale_t loc*);

int
strncasecmp_l(*const char *s1*, *const char *s2*, *size_t len*, *locale_t loc*);

DESCRIPTION

The **strcasecmp**() and **strncasecmp**() functions compare the null-terminated strings *s1* and *s2*.

The **strncasecmp**() function compares at most *len* characters. The **strcasecmp_l**() and **strncasecmp_l**() functions do the same as their non-locale versions above, but take an explicit locale rather than using the current locale.

RETURN VALUES

The functions **strcasecmp**() and **strncasecmp**() return an integer greater than, equal to, or less than 0, depending on whether *s1* is lexicographically greater than, equal to, or less than *s2* after translation of each corresponding character to lower-case. The strings themselves are not modified. The comparison is done using unsigned characters, so that '\200' is greater than '\0'. The functions **strcasecmp_l**() and **strncasecmp_l**() do the same but take explicit locales.

SEE ALSO

bcmp(3), memcmp(3), strcmp(3), strcoll(3), strxfrm(3), tolower(3), wcscasecmp(3)

HISTORY

The **strcasecmp**() and **strncasecmp**() functions first appeared in 4.4BSD. Their prototypes existed previously in <*string.h*> before they were moved to <*strings.h*> for IEEE Std 1003.1-2001 ("POSIX.1") compliance.

NAME

strcat, **strncat** - concatenate strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

*char **
strcat(*char * restrict s*, *const char * restrict append*);

*char **
strncat(*char * restrict s*, *const char * restrict append*, *size_t count*);

DESCRIPTION

The **strcat**() and **strncat**() functions append a copy of the null-terminated string *append* to the end of the null-terminated string *s*, then add a terminating '\0'. The string *s* must have sufficient space to hold the result. If *s* and *append* overlap, the results are undefined.

The **strncat**() function appends not more than *count* characters from *append*, and then adds a terminating '\0'. If *s* and *append* overlap, the results are undefined.

RETURN VALUES

The **strcat**() and **strncat**() functions return the pointer *s*.

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strcpy(3), strlcat(3), strlcpy(3), wcscat(3)

STANDARDS

The **strcat**() and **strncat**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

SECURITY CONSIDERATIONS

The **strcat**() function is easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack.

Avoid using **strcat**(). Instead, use **strncat**() or **strlcat**() and ensure that no more characters are copied to the destination buffer than it can hold.

Note that **strncat**() can also be problematic. It may be a security concern for a string to be truncated at

all. Since the truncated string will not be as long as the original, it may refer to a completely different resource and usage of the truncated resource could result in very incorrect behavior. Example:

```
void
foo(const char *arbitrary_string)
{
        char onstack[8];

#if defined(BAD)
        /*
         * This first strcat is bad behavior.  Do not use strcat!
         */
        (void)strcat(onstack, arbitrary_string);   /* BAD! */
#elif defined(BETTER)
        /*
         * The following two lines demonstrate better use of
         * strncat().
         */
        (void)strncat(onstack, arbitrary_string,
           sizeof(onstack) - strlen(onstack) - 1);
#elif defined(BEST)
        /*
         * These lines are even more robust due to testing for
         * truncation.
         */
        if (strlen(arbitrary_string) + 1 >
           sizeof(onstack) - strlen(onstack))
                err(1, "onstack would be truncated");
        (void)strncat(onstack, arbitrary_string,
           sizeof(onstack) - strlen(onstack) - 1);
#endif
}
```

NAME

strchr, **strrchr**, **strchrnul** - locate character in string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

*char **
strchr(*const char *s*, *int c*);

*char **
strrchr(*const char *s*, *int c*);

*char **
strchrnul(*const char *s*, *int c*);

DESCRIPTION

The **strchr**() function locates the first occurrence of *c* (converted to a *char*) in the string pointed to by *s*. The terminating null character is considered part of the string; therefore if *c* is '\0', the functions locate the terminating '\0'.

The **strrchr**() function is identical to **strchr**() except it locates the last occurrence of *c*.

The **strchrnul**() function is identical to **strchr**() except that if *c* is not found in *s* a pointer to the terminating '\0' is returned.

RETURN VALUES

The functions **strchr**() and **strrchr**() return a pointer to the located character, or NULL if the character does not appear in the string.

strchrnul() returns a pointer to the terminating '\0' if the character does not appear in the string.

SEE ALSO

memchr(3), memmem(3), strcspn(3), strpbrk(3), strsep(3), strspn(3), strstr(3), strtok(3), wcschr(3)

STANDARDS

The functions **strchr**() and **strrchr**() conform to ISO/IEC 9899:1990 ("ISO C90"). The function **strchrnul**() is a GNU extension.

HISTORY

The **strchrnul**() function first appeared in glibc 2.1.1 and was added in FreeBSD 10.0.

NAME

strcmp, **strncmp** - compare strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

int
strcmp(*const char *s1*, *const char *s2*);

int
strncmp(*const char *s1*, *const char *s2*, *size_t len*);

DESCRIPTION

The **strcmp**() and **strncmp**() functions lexicographically compare the null-terminated strings *s1* and *s2*.

The **strncmp**() function compares not more than *len* characters. Because **strncmp**() is designed for comparing strings rather than binary data, characters that appear after a ‘\0’ character are not compared.

RETURN VALUES

The **strcmp**() and **strncmp**() functions return an integer greater than, equal to, or less than 0, according as the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that ‘\200’ is greater than ‘\0’.

SEE ALSO

bcmp(3), memcmp(3), strcasecmp(3), strcoll(3), strxfrm(3), wcscmp(3)

STANDARDS

The **strcmp**() and **strncmp**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

strcoll - compare strings according to current collation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

int
strcoll(*const char *s1*, *const char *s2*);

int
strcoll_l(*const char *s1*, *const char *s2*, *locale_t loc*);

DESCRIPTION

The **strcoll**() function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, depending on whether *s1* is greater than, equal to, or less than *s2*. If information about the current locale collation is not available, the value of **strcmp**(*s1*, *s2*) is returned. The **strcoll_l**() function uses an explicit locale argument rather than the system locale.

SEE ALSO

setlocale(3), strcmp(3), strxfrm(3), wcscoll(3)

STANDARDS

The **strcoll**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **strcoll_l**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

strspn, **strcspn** - span a string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

size_t
strspn(*const char *s*, *const char *charset*);

size_t
strcspn(*const char *s*, *const char *charset*);

DESCRIPTION

The **strspn**() function spans the initial part of the null-terminated string *s* as long as the characters from *s* occur in the null-terminated string *charset*. In other words, it computes the string array index of the first character of *s* which is not in *charset*, else the index of the first null character.

The **strcspn**() function spans the initial part of the null-terminated string *s* as long as the characters from *s* **do not** occur in the null-terminated string *charset* (it spans the **complement** of *charset*). In other words, it computes the string array index of the first character of *s* which is also in *charset*, else the index of the first null character.

RETURN VALUES

The **strspn**() and **strcspn**() functions return the number of characters spanned.

SEE ALSO

memchr(3), strchr(3), strpbrk(3), strrchr(3), strsep(3), strstr(3), strtok(3), wcsspn(3)

STANDARDS

The **strspn**() and **strcspn**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

NAME

strdup, **strndup** - save a copy of a string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

*char **
strdup(*const char *str*);

*char **
strndup(*const char *str*, *size_t len*);

DESCRIPTION

The **strdup**() function allocates sufficient memory for a copy of the string *str*, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3).

If insufficient memory is available, NULL is returned and *errno* is set to ENOMEM.

The **strndup**() function copies at most *len* characters from the string *str* always NUL terminating the copied string.

SEE ALSO

free(3), malloc(3), wcsdup(3)

STANDARDS

The **strdup**() function is specified by IEEE Std 1003.1-2001 ("POSIX.1"). The **strndup**() function is specified by IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The **strdup**() function first appeared in 4.4BSD. The **strndup**() function was added in FreeBSD 7.2.

NAME

strfmon - convert monetary value to string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <monetary.h>

ssize_t
strfmon(*char * restrict s*, *size_t maxsize*, *const char * restrict format*, ...);

ssize_t
strfmon_l(*char * restrict s*, *size_t maxsize*, *locale_t loc*, *const char * restrict format*, ...);

DESCRIPTION

The **strfmon**() function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The **strfmon_l**() function does the same as **strfmon**() but takes an explicit locale rather than using the current locale.

The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. After the **%**, the following appear in sequence:

- Zero or more of the following flags:

  =*f* A '**=**' character followed by another character *f* which is used as the numeric fill character.

  ^ Do not use grouping characters, regardless of the current locale default.

  + Represent positive values by prefixing them with a positive sign, and negative values by prefixing them with a negative sign. This is the default.

  (Enclose negative values in parentheses.

  ! Do not include a currency symbol in the output.

- Left justify the result. Only valid when a field width is specified.

- An optional minimum field width as a decimal number. By default, there is no minimum width.
- A '**#**' sign followed by a decimal number specifying the maximum expected number of digits after the radix character.
- A '**.**' character followed by a decimal number specifying the number the number of digits after the radix character.
- One of the following conversion specifiers:

i The *double* argument is formatted as an international monetary amount.

n The *double* argument is formatted as a national monetary amount.

% A '%' character is written.

RETURN VALUES

If the total number of resulting bytes including the terminating NUL byte is not more than *maxsize*, **strfmon**() returns the number of bytes placed into the array pointed to by *s*, not including the terminating NUL byte. Otherwise, -1 is returned, the contents of the array are indeterminate, and *errno* is set to indicate the error.

The **strfmon_l**() function returns the same values as **strfmon**().

ERRORS

The **strfmon**() function will fail if:

[E2BIG] Conversion stopped due to lack of space in the buffer.

[EINVAL] The format string is invalid.

[ENOMEM] Not enough memory for temporary buffers.

SEE ALSO

localeconv(3)

STANDARDS

The **strfmon**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1"). The **strfmon_l**() function

conforms to IEEE Std 1003.1-2008 ("POSIX.1").

AUTHORS

The **strfmon**() function was implemented by Alexey Zelkin <*phantom@FreeBSD.org*>.

This manual page was written by Jeroen Ruigrok van der Werven <*asmodai@FreeBSD.org*> based on the standards' text.

BUGS

The **strfmon**() function does not correctly handle multibyte characters in the *format* argument.

NAME

strftime - format date and time

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

size_t
strftime(*char * restrict buf*, *size_t maxsize*, *const char * restrict format*,
*const struct tm * restrict timeptr*);

size_t
strftime_l(*char *restrict buf*, *size_t maxsize*, *const char * restrict format*, *const struct tm *restrict timeptr*,
locale_t loc);

DESCRIPTION

The **strftime**() function formats the information from *timeptr* into the buffer *buf* according to the string pointed to by *format*. The function **strftime_l**() does the same as **strftime**() but takes an explicit locale rather than using the current locale.

The *format* string consists of zero or more conversion specifications and ordinary characters. All ordinary characters are copied directly into the buffer. A conversion specification consists of a percent sign "‘%’" and one other character.

No more than *maxsize* characters will be placed into the array. If the total number of resulting characters, including the terminating NUL character, is not more than *maxsize*, **strftime**() returns the number of characters in the array, not counting the terminating NUL. Otherwise, zero is returned and the buffer contents are indeterminate.

The conversion specifications are copied to the buffer after expansion as follows:-

%A is replaced by national representation of the full weekday name.

%a is replaced by national representation of the abbreviated weekday name.

%B is replaced by national representation of the full month name.

%b is replaced by national representation of the abbreviated month name.

%C is replaced by (year / 100) as decimal number; single digits are preceded by a zero.

%c is replaced by national representation of time and date.

%D is equivalent to "%m/%d/%y".

%d is replaced by the day of the month as a decimal number (01-31).

%E* %O*
POSIX locale extensions. The sequences %Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy are supposed to provide alternate representations.

Additionally %OB implemented to represent alternative months names (used standalone, without day mentioned).

%e is replaced by the day of the month as a decimal number (1-31); single digits are preceded by a blank.

%F is equivalent to "%Y-%m-%d".

%G is replaced by a year as a decimal number with century. This year is the one that contains the greater part of the week (Monday as the first day of the week).

%g is replaced by the same year as in "%G", but as a decimal number without century (00-99).

%H is replaced by the hour (24-hour clock) as a decimal number (00-23).

%h the same as **%b**.

%I is replaced by the hour (12-hour clock) as a decimal number (01-12).

%j is replaced by the day of the year as a decimal number (001-366).

%k is replaced by the hour (24-hour clock) as a decimal number (0-23); single digits are preceded by a blank.

%l is replaced by the hour (12-hour clock) as a decimal number (1-12); single digits are preceded by a blank.

%M is replaced by the minute as a decimal number (00-59).

%m is replaced by the month as a decimal number (01-12).

%n is replaced by a newline.

%O*
the same as **%E***.

%p is replaced by national representation of either "ante meridiem" (a.m.) or "post meridiem" (p.m.) as appropriate.

%R is equivalent to "%H:%M".

%r is equivalent to "%I:%M:%S %p".

%S is replaced by the second as a decimal number (00-60).

%s is replaced by the number of seconds since the Epoch, UTC (see mktime(3)).

%T is equivalent to "%H:%M:%S".

%t is replaced by a tab.

%U is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).

%u is replaced by the weekday (Monday as the first day of the week) as a decimal number (1-7).

%V is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (01-53). If the week containing January 1 has four or more days in the new year, then it is week 1; otherwise it is the last week of the previous year, and the next week is week 1.

%v is equivalent to "%e-%b-%Y".

%W is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00-53).

%w is replaced by the weekday (Sunday as the first day of the week) as a decimal number (0-6).

%X is replaced by national representation of the time.

%x is replaced by national representation of the date.

%Y is replaced by the year with century as a decimal number.

%y is replaced by the year without century as a decimal number (00-99).

%Z is replaced by the time zone name.

%z is replaced by the time zone offset from UTC; a leading plus sign stands for east of UTC, a minus sign for west of UTC, hours and minutes follow with two digits each and no delimiter between them (common form for RFC 822 date headers).

%+ is replaced by national representation of the date and time (the format is similar to that produced by date(1)).

%-* GNU libc extension. Do not do any padding when performing numerical outputs.

%_* GNU libc extension. Explicitly specify space for padding.

%0* GNU libc extension. Explicitly specify zero for padding.

%% is replaced by '%'.

SEE ALSO

date(1), printf(1), ctime(3), printf(3), strptime(3), wcsftime(3)

STANDARDS

The **strftime**() function conforms to ISO/IEC 9899:1990 ("ISO C90") with a lot of extensions including '%C', '%D', '%E*', '%e', '%G', '%g', '%h', '%k', '%l', '%n', '%O*', '%R', '%r', '%s', '%T', '%t', '%u', '%V', '%z', '%+'.

The peculiar week number and year in the replacements of '%G', '%g' and '%V' are defined in ISO 8601: 1988. The **strftime_l**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

BUGS

There is no conversion specification for the phase of the moon.

The **strftime**() function does not correctly handle multibyte characters in the *format* argument.

NAME

strlcpy, **strlcat** - size-bounded string copying and concatenation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

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

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

DESCRIPTION

The **strlcpy**() and **strlcat**() functions copy and concatenate strings with the same input parameters and output result as snprintf(3). They are designed to be safer, more consistent, and less error prone replacements for the easily misused functions strncpy(3) and strncat(3).

strlcpy() and **strlcat**() take the full size of the destination buffer and guarantee NUL-termination if there is room. Note that room for the NUL should be included in *dstsize*.

strlcpy() copies up to *dstsize* - 1 characters from the string *src* to *dst*, NUL-terminating the result if *dstsize* is not 0.

strlcat() appends string *src* to the end of *dst*. It will append at most *dstsize* - strlen(dst) - 1 characters. It will then NUL-terminate, unless *dstsize* is 0 or the original *dst* string was longer than *dstsize* (in practice this should not happen as it means that either *dstsize* is incorrect or that *dst* is not a proper string).

If the *src* and *dst* strings overlap, the behavior is undefined.

RETURN VALUES

Besides quibbles over the return type (*size_t* versus *int*) and signal handler safety (snprintf(3) is not entirely safe on some systems), the following two are equivalent:

```
n = strlcpy(dst, src, len);
n = snprintf(dst, len, "%s", src);
```

Like snprintf(3), the **strlcpy**() and **strlcat**() functions return the total length of the string they tried to

create. For **strlcpy**() that means the length of *src*. For **strlcat**() that means the initial length of *dst* plus the length of *src*.

If the return value is >= *dstsize*, the output string has been truncated. It is the caller's responsibility to handle this.

EXAMPLES

The following code fragment illustrates the simple case:

```
char *s, *p, buf[BUFSIZ];

...

(void)strlcpy(buf, s, sizeof(buf));
(void)strlcat(buf, p, sizeof(buf));
```

To detect truncation, perhaps while building a pathname, something like the following might be used:

```
char *dir, *file, pname[MAXPATHLEN];

...

if (strlcpy(pname, dir, sizeof(pname)) >= sizeof(pname))
        goto toolong;
if (strlcat(pname, file, sizeof(pname)) >= sizeof(pname))
        goto toolong;
```

Since it is known how many characters were copied the first time, things can be sped up a bit by using a copy instead of an append:

```
char *dir, *file, pname[MAXPATHLEN];
size_t n;

...

n = strlcpy(pname, dir, sizeof(pname));
if (n >= sizeof(pname))
        goto toolong;
if (strlcpy(pname + n, file, sizeof(pname) - n) >= sizeof(pname) - n)
        goto toolong;
```

However, one may question the validity of such optimizations, as they defeat the whole purpose of **strlcpy**() and **strlcat**(). As a matter of fact, the first version of this manual page got it wrong.

SEE ALSO

snprintf(3), strncat(3), strncpy(3), wcslcpy(3)

HISTORY

The **strlcpy**() and **strlcat**() functions first appeared in OpenBSD 2.4, and FreeBSD 3.3.

NAME

strmode - convert inode status information into a symbolic string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

void
strmode(*mode_t mode*, *char *bp*);

DESCRIPTION

The **strmode**() function converts a file *mode* (the type and permission information associated with an inode, see stat(2)) into a symbolic string which is stored in the location referenced by *bp*. This stored string is eleven characters in length plus a trailing NUL.

The first character is the inode type, and will be one of the following:

-	regular file
b	block special
c	character special
d	directory
l	symbolic link
p	fifo
s	socket
w	whiteout
?	unknown inode type

The next nine characters encode three sets of permissions, in three characters each. The first three characters are the permissions for the owner of the file, the second three for the group the file belongs to, and the third for the ‘‘other’’, or default, set of users.

Permission checking is done as specifically as possible. If read permission is denied to the owner of a file in the first set of permissions, the owner of the file will not be able to read the file. This is true even if the owner is in the file’s group and the group permissions allow reading or the ‘‘other’’ permissions allow reading.

If the first character of the three character set is an ‘‘r’’, the file is readable for that set of users; if a dash ‘‘-’’, it is not readable.

If the second character of the three character set is a ‘‘w’’, the file is writable for that set of users; if a dash ‘‘-’’, it is not writable.

The third character is the first of the following characters that apply:

S If the character is part of the owner permissions and the file is not executable or the directory is not searchable by the owner, and the set-user-id bit is set.

S If the character is part of the group permissions and the file is not executable or the directory is not searchable by the group, and the set-group-id bit is set.

T If the character is part of the other permissions and the file is not executable or the directory is not searchable by others, and the ‘‘sticky’’ (S_ISVTX) bit is set.

s If the character is part of the owner permissions and the file is executable or the directory searchable by the owner, and the set-user-id bit is set.

s If the character is part of the group permissions and the file is executable or the directory searchable by the group, and the set-group-id bit is set.

t If the character is part of the other permissions and the file is executable or the directory searchable by others, and the ‘‘sticky’’ (S_ISVTX) bit is set.

x The file is executable or the directory is searchable.

- None of the above apply.

The last character will always be a space.

SEE ALSO

chmod(1), find(1), stat(2), getmode(3), setmode(3)

HISTORY

The **strmode**() function first appeared in 4.4BSD.

NAME

strlen, **strnlen** - find length of string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

size_t
strlen(*const char *s*);

size_t
strnlen(*const char *s*, *size_t maxlen*);

DESCRIPTION

The **strlen**() function computes the length of the string *s*. The **strnlen**() function attempts to compute the length of *s*, but never scans beyond the first *maxlen* bytes of *s*.

RETURN VALUES

The **strlen**() function returns the number of characters that precede the terminating NUL character. The **strnlen**() function returns either the same result as **strlen**() or *maxlen*, whichever is smaller.

SEE ALSO

string(3), wcslen(3), wcswidth(3)

STANDARDS

The **strlen**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **strnlen**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

unvis, **strunvis**, **strnunvis**, **strunvisx**, **strnunvisx** - decode a visual representation of characters

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <vis.h>

int
unvis(*char *cp*, *int c*, *int *astate*, *int flag*);

int
strunvis(*char *dst*, *const char *src*);

int
strnunvis(*char *dst*, *size_t dlen*, *const char *src*);

int
strunvisx(*char *dst*, *const char *src*, *int flag*);

int
strnunvisx(*char *dst*, *size_t dlen*, *const char *src*, *int flag*);

DESCRIPTION

The **unvis**(), **strunvis**() and **strunvisx**() functions are used to decode a visual representation of characters, as produced by the vis(3) function, back into the original form.

The **unvis**() function is called with successive characters in *c* until a valid sequence is recognized, at which time the decoded character is available at the character pointed to by *cp*.

The **strunvis**() function decodes the characters pointed to by *src* into the buffer pointed to by *dst*. The **strunvis**() function simply copies *src* to *dst*, decoding any escape sequences along the way, and returns the number of characters placed into *dst*, or -1 if an invalid escape sequence was detected. The size of *dst* should be equal to the size of *src* (that is, no expansion takes place during decoding).

The **strunvisx**() function does the same as the **strunvis**() function, but it allows you to add a flag that specifies the style the string *src* is encoded with. Currently, the supported flags are: VIS_HTTPSTYLE and VIS_MIMESTYLE.

The **unvis**() function implements a state machine that can be used to decode an arbitrary stream of bytes. All state associated with the bytes being decoded is stored outside the **unvis**() function (that is, a pointer to the state is passed in), so calls decoding different streams can be freely intermixed. To start decoding a stream of bytes, first initialize an integer to zero. Call **unvis**() with each successive byte, along with a pointer to this integer, and a pointer to a destination character. The **unvis**() function has several return codes that must be handled properly. They are:

0 (zero)	Another character is necessary; nothing has been recognized yet.
UNVIS_VALID	A valid character has been recognized and is available at the location pointed to by *cp*.
UNVIS_VALIDPUSH	A valid character has been recognized and is available at the location pointed to by *cp*; however, the character currently passed in should be passed in again.
UNVIS_NOCHAR	A valid sequence was detected, but no character was produced. This return code is necessary to indicate a logical break between characters.
UNVIS_SYNBAD	An invalid escape sequence was detected, or the decoder is in an unknown state. The decoder is placed into the starting state.

When all bytes in the stream have been processed, call **unvis**() one more time with flag set to UNVIS_END to extract any remaining character (the character passed in is ignored).

The *flag* argument is also used to specify the encoding style of the source. If set to VIS_HTTPSTYLE or VIS_HTTP1808, **unvis**() will decode URI strings as specified in RFC 1808. If set to VIS_HTTP1866, **unvis**() will decode entity references and numeric character references as specified in RFC 1866. If set to VIS_MIMESTYLE, **unvis**() will decode MIME Quoted-Printable strings as specified in RFC 2045. If set to VIS_NOESCAPE, **unvis**() will not decode '\' quoted characters.

The following code fragment illustrates a proper use of **unvis**().

```
int state = 0;
char out;

while ((ch = getchar()) != EOF) {
again:
        switch(unvis(&out, ch, &state, 0)) {
        case 0:
        case UNVIS_NOCHAR:
```

```
                        break;
                case UNVIS_VALID:
                        (void)putchar(out);
                        break;
                case UNVIS_VALIDPUSH:
                        (void)putchar(out);
                        goto again;
                case UNVIS_SYNBAD:
                        errx(EXIT_FAILURE, "Bad character sequence!");
                }
        }
        if (unvis(&out, '\0', &state, UNVIS_END) == UNVIS_VALID)
                (void)putchar(out);
```

ERRORS

The functions **strunvis**(), **strnunvis**(), **strunvisx**(), and **strnunvisx**() will return -1 on error and set *errno* to:

[EINVAL] An invalid escape sequence was detected, or the decoder is in an unknown state.

In addition the functions **strnunvis**() and **strnunvisx**() will can also set *errno* on error to:

[ENOSPC] Not enough space to perform the conversion.

SEE ALSO

unvis(1), vis(1), vis(3)

R. Fielding, *Relative Uniform Resource Locators*, RFC1808.

HISTORY

The **unvis**() function first appeared in 4.4BSD. The **strnunvis**() and **strnunvisx**() functions appeared in NetBSD 6.0 and FreeBSD 9.2.

BUGS

The names VIS_HTTP1808 and VIS_HTTP1866 are wrong. Percent-encoding was defined in RFC 1738, the original RFC for URL. RFC 1866 defines HTML 2.0, an application of SGML, from which it inherits concepts of numeric character references and entity references.

NAME

strptime - parse date and time string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

*char **
strptime(*const char * restrict buf*, *const char * restrict format*, *struct tm * restrict timeptr*);

#include <time.h>
#include <xlocale.h>

*char **
strptime_l(*const char * restrict buf*, *const char * restrict format*, *struct tm * restrict timeptr*, *locale_t loc*);

DESCRIPTION

The **strptime**() function parses the string in the buffer *buf* according to the string pointed to by *format*, and fills in the elements of the structure pointed to by *timeptr*. The resulting values will be relative to the local time zone. Thus, it can be considered the reverse operation of strftime(3). The **strptime_l**() function does the same as **strptime**(), but takes an explicit locale rather than using the current locale.

The *format* string consists of zero or more conversion specifications and ordinary characters. All ordinary characters are matched exactly with the buffer, where white space in the format string will match any amount of white space in the buffer. All conversion specifications are identical to those described in strftime(3).

Two-digit year values, including formats *%y* and *%D*, are now interpreted as beginning at 1969 per POSIX requirements. Years 69-00 are interpreted in the 20th century (1969-2000), years 01-68 in the 21st century (2001-2068). The *%U* and *%W* format specifiers accept any value within the range 00 to 53.

If the *format* string does not contain enough conversion specifications to completely specify the resulting *struct tm*, the unspecified members of *timeptr* are left untouched. For example, if *format* is "%H:%M:%S", only *tm_hour*, *tm_sec* and *tm_min* will be modified. If time relative to today is desired, initialize the *timeptr* structure with today's date before passing it to **strptime**().

RETURN VALUES

Upon successful completion, **strptime**() returns the pointer to the first character in *buf* that has not been required to satisfy the specified conversions in *format*. It returns NULL if one of the conversions failed. **strptime_l**() returns the same values as **strptime**().

SEE ALSO

date(1), scanf(3), strftime(3)

HISTORY

The **strptime**() function appeared in FreeBSD 3.0.

AUTHORS

The **strptime**() function has been contributed by Powerdog Industries.

This man page was written by Jörg Wunsch.

BUGS

Both the *%e* and *%l* format specifiers may incorrectly scan one too many digits if the intended values comprise only a single digit and that digit is followed immediately by another digit. Both specifiers accept zero-padded values, even though they are both defined as taking unpadded values.

The *%p* format specifier has no effect unless it is parsed *after* hour-related specifiers. Specifying *%l* without *%p* will produce undefined results. Note that 12AM (ante meridiem) is taken as midnight and 12PM (post meridiem) is taken as noon.

The *%Z* format specifier only accepts time zone abbreviations of the local time zone, or the value "GMT". This limitation is because of ambiguity due to of the over loading of time zone abbreviations. One such example is *EST* which is both Eastern Standard Time and Eastern Australia Summer Time.

The **strptime**() function does not correctly handle multibyte characters in the *format* argument.

NAME

strstr, **strcasestr**, **strnstr** - locate a substring in a string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

*char **
strstr(*const char *big*, *const char *little*);

*char **
strcasestr(*const char *big*, *const char *little*);

*char **
strnstr(*const char *big*, *const char *little*, *size_t len*);

#include <string.h>
#include <xlocale.h>

*char **
strcasestr_l(*const char *big*, *const char *little*, *locale_t loc*);

DESCRIPTION

The **strstr**() function locates the first occurrence of the null-terminated string *little* in the null-terminated string *big*.

The **strcasestr**() function is similar to **strstr**(), but ignores the case of both strings.

The **strcasestr_l**() function does the same as **strcasestr**() but takes an explicit locale rather than using the current locale.

The **strnstr**() function locates the first occurrence of the null-terminated string *little* in the string *big*, where not more than *len* characters are searched. Characters that appear after a '\0' character are not searched. Since the **strnstr**() function is a FreeBSD specific API, it should only be used when portability is not a concern.

RETURN VALUES

If *little* is an empty string, *big* is returned; if *little* occurs nowhere in *big*, NULL is returned; otherwise a

pointer to the first character of the first occurrence of *little* is returned.

EXAMPLES

The following sets the pointer *ptr* to the "Bar Baz" portion of *largestring*:

```
const char *largestring = "Foo Bar Baz";
const char *smallstring = "Bar";
char *ptr;

ptr = strstr(largestring, smallstring);
```

The following sets the pointer *ptr* to NULL, because only the first 4 characters of *largestring* are searched:

```
const char *largestring = "Foo Bar Baz";
const char *smallstring = "Bar";
char *ptr;

ptr = strnstr(largestring, smallstring, 4);
```

SEE ALSO

memchr(3), memmem(3), strchr(3), strcspn(3), strpbrk(3), strrchr(3), strsep(3), strspn(3), strtok(3), wcsstr(3)

STANDARDS

The **strstr**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

NAME

strtod, **strtof**, **strtold** - convert ASCII string to floating point

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

double
strtod(*const char * restrict nptr*, *char ** restrict endptr*);

float
strtof(*const char * restrict nptr*, *char ** restrict endptr*);

long double
strtold(*const char * restrict nptr*, *char ** restrict endptr*);

DESCRIPTION

These conversion functions convert the initial portion of the string pointed to by *nptr* to *double*, *float*, and *long double* representation, respectively.

The expected form of the string is an optional plus (‘‘+’’) or minus sign (‘‘-’’) followed by either:

- a decimal significand consisting of a sequence of decimal digits optionally containing a decimal-point character, or

- a hexadecimal significand consisting of a ‘‘0X’’ or ‘‘0x’’ followed by a sequence of hexadecimal digits optionally containing a decimal-point character.

In both cases, the significand may be optionally followed by an exponent. An exponent consists of an ‘‘E’’ or ‘‘e’’ (for decimal constants) or a ‘‘P’’ or ‘‘p’’ (for hexadecimal constants), followed by an optional plus or minus sign, followed by a sequence of decimal digits. For decimal constants, the exponent indicates the power of 10 by which the significand should be scaled. For hexadecimal constants, the scaling is instead done by powers of 2.

Alternatively, if the portion of the string following the optional plus or minus sign begins with "INFINITY" or "NAN", ignoring case, it is interpreted as an infinity or a quiet NaN, respectively. The syntax "NAN(*s*)", where *s* is an alphanumeric string, produces the same value as the call **nan**("*s*") (respectively, **nanf**("*s*") and **nanl**("*s*").)

In any of the above cases, leading white-space characters in the string (as defined by the isspace(3) function) are skipped. The decimal point character is defined in the program's locale (category LC_NUMERIC).

RETURN VALUES

The **strtod**(), **strtof**(), and **strtold**() functions return the converted value, if any.

If *endptr* is not NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*.

If no conversion is performed, zero is returned and the value of *nptr* is stored in the location referenced by *endptr*.

If the correct value would cause overflow, plus or minus HUGE_VAL, HUGE_VALF, or HUGE_VALL is returned (according to the sign and type of the return value), and ERANGE is stored in *errno*. If the correct value would cause underflow, zero is returned and ERANGE is stored in *errno*.

ERRORS

[ERANGE] Overflow or underflow occurred.

SEE ALSO

atof(3), atoi(3), atol(3), nan(3), strtol(3), strtoul(3), wcstod(3)

STANDARDS

The **strtod**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

AUTHORS

The author of this software is David M. Gay.

permission.

LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

NAME

strtol, **strtoll**, **strtoimax**, **strtoq** - convert a string value to a *long*, *long long*, *intmax_t* or *quad_t* integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>
#include <limits.h>

long
strtol(*const char * restrict nptr*, *char ** restrict endptr*, *int base*);

long long
strtoll(*const char * restrict nptr*, *char ** restrict endptr*, *int base*);

#include <inttypes.h>

intmax_t
strtoimax(*const char * restrict nptr*, *char ** restrict endptr*, *int base*);

#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>

quad_t
strtoq(*const char *nptr*, *char **endptr*, *int base*);

DESCRIPTION

The **strtol**() function converts the string in *nptr* to a *long* value. The **strtoll**() function converts the string in *nptr* to a *long long* value. The **strtoimax**() function converts the string in *nptr* to an *intmax_t* value. The **strtoq**() function converts the string in *nptr* to a *quad_t* value. The conversion is done according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace(3)) followed by a single optional ‘+’ or ‘-’ sign. If *base* is zero or 16, the string may then include a "0x" prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is ‘0’, in which case it is taken as 8 (octal).

The remainder of the string is converted to a *long*, *long long*, *intmax_t* or *quad_t* value in the obvious

manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is not NULL, **strtol**() stores the address of the first invalid character in **endptr*. If there were no digits at all, however, **strtol**() stores the original value of *nptr* in **endptr*. (Thus, if **nptr* is not '\0' but ***endptr* is '\0' on return, the entire string was valid.)

RETURN VALUES

The **strtol**(), **strtoll**(), **strtoimax**() and **strtoq**() functions return the result of the conversion, unless the value would underflow or overflow. If no conversion could be performed, 0 is returned and the global variable *errno* is set to EINVAL (the last feature is not portable across all platforms). If an overflow or underflow occurs, *errno* is set to ERANGE and the function return value is clamped according to the following table.

Function	**underflow**	**overflow**
strtol()	LONG_MIN	LONG_MAX
strtoll()	LLONG_MIN	LLONG_MAX
strtoimax()	INTMAX_MIN	INTMAX_MAX
strtoq()	LLONG_MIN	LLONG_MAX

ERRORS

[EINVAL] The value of *base* is not supported or no conversion could be performed (the last feature is not portable across all platforms).

[ERANGE] The given string was out of range; the value converted has been clamped.

SEE ALSO

atof(3), atoi(3), atol(3), strtod(3), strtonum(3), strtoul(3), wcstol(3)

STANDARDS

The **strtol**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **strtoll**() and **strtoimax**() functions conform to ISO/IEC 9899:1999 ("ISO C99"). The BSD **strtoq**() function is deprecated.

NAME

strtoul, **strtoull**, **strtoumax**, **strtouq** - convert a string to an *unsigned long*, *unsigned long long*, *uintmax_t*, or *u_quad_t* integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>
#include <limits.h>

unsigned long
strtoul(*const char * restrict nptr*, *char ** restrict endptr*, *int base*);

unsigned long long
strtoull(*const char * restrict nptr*, *char ** restrict endptr*, *int base*);

#include <inttypes.h>

uintmax_t
strtoumax(*const char * restrict nptr*, *char ** restrict endptr*, *int base*);

#include <sys/types.h>
#include <stdlib.h>
#include <limits.h>

u_quad_t
strtouq(*const char *nptr*, *char **endptr*, *int base*);

DESCRIPTION

The **strtoul**() function converts the string in *nptr* to an *unsigned long* value. The **strtoull**() function converts the string in *nptr* to an *unsigned long long* value. The **strtoumax**() function converts the string in *nptr* to an *uintmax_t* value. The **strtouq**() function converts the string in *nptr* to a *u_quad_t* value. The conversion is done according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by isspace(3)) followed by a single optional ‘+’ or ‘-’ sign. If *base* is zero or 16, the string may then include a "0x" prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is ‘0’, in which case it is taken as 8 (octal).

The remainder of the string is converted to an *unsigned long* value in the obvious manner, stopping at the end of the string or at the first character that does not produce a valid digit in the given base. (In bases above 10, the letter 'A' in either upper or lower case represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is not NULL, **strtoul**() stores the address of the first invalid character in **endptr*. If there were no digits at all, however, **strtoul**() stores the original value of *nptr* in **endptr*. (Thus, if **nptr* is not '\0' but ***endptr* is '\0' on return, the entire string was valid.)

RETURN VALUES

The **strtoul**(), **strtoull**(), **strtoumax**() and **strtouq**() functions return either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow; in the latter case, **strtoul**() returns ULONG_MAX, **strtoull**() returns ULLONG_MAX, **strtoumax**() returns UINTMAX_MAX, and **strtouq**() returns ULLONG_MAX. In all cases, *errno* is set to ERANGE. If no conversion could be performed, 0 is returned and the global variable *errno* is set to EINVAL (the last feature is not portable across all platforms).

ERRORS

[EINVAL] The value of *base* is not supported or no conversion could be performed (the last feature is not portable across all platforms).

[ERANGE] The given string was out of range; the value converted has been clamped.

SEE ALSO

strtol(3), strtonum(3), wcstoul(3)

STANDARDS

The **strtoul**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **strtoull**() and **strtoumax**() functions conform to ISO/IEC 9899:1999 ("ISO C99"). The BSD **strtouq**() function is deprecated.

NAME

strxfrm - transform a string under locale

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <string.h>

size_t
strxfrm(*char * restrict dst*, *const char * restrict src*, *size_t n*);

size_t
strxfrm_l(*char * restrict dst*, *const char *restrict src*, *size_t n*, *locale_t loc*);

DESCRIPTION

The **strxfrm**() function transforms a null-terminated string pointed to by *src* according to the current locale collation if any, then copies the transformed string into *dst*. Not more than *n* characters are copied into *dst*, including the terminating null character added. If *n* is set to 0 (it helps to determine an actual size needed for transformation), *dst* is permitted to be a NULL pointer.

Comparing two strings using **strcmp**() after **strxfrm**() is equal to comparing two original strings with **strcoll**().

strxfrm_l() does the same, however takes an explicit locale rather than the global locale.

RETURN VALUES

Upon successful completion, **strxfrm**() and **strxfrm_l**() return the length of the transformed string not including the terminating null character. If this value is *n* or more, the contents of *dst* are indeterminate.

SEE ALSO

setlocale(3), strcmp(3), strcoll(3), wcsxfrm(3)

STANDARDS

The **strxfrm**() function conforms to ISO/IEC 9899:1990 ("ISO C90"). The **strxfrm_l**() function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

svc_dg_enablecache, **svc_exit**, **svc_fdset**, **svc_freeargs**, **svc_getargs**, **svc_getreq_common**, **svc_getreq_poll**, **svc_getreqset**, **svc_getrpccaller**, **svc_pollset**, **svc_run**, **svc_sendreply** - library routines for RPC servers

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

int
svc_dg_enablecache(*SVCXPRT *xprt*, *const unsigned cache_size*);

void
svc_exit(*void*);

bool_t
svc_freeargs(*const SVCXPRT *xprt*, *const xdrproc_t inproc*, *caddr_t in*);

bool_t
svc_getargs(*const SVCXPRT *xprt*, *const xdrproc_t inproc*, *caddr_t in*);

void
svc_getreq_common(*const int fd*);

void
svc_getreq_poll(*struct pollfd *pfdp*, *const int pollretval*);

void
svc_getreqset(*fd_set * rdfds*);

*struct netbuf **
svc_getrpccaller(*const SVCXPRT *xprt*);

*struct cmsgcred **
__svc_getcallercreds(*const SVCXPRT *xprt*);

struct pollfd svc_pollset[FD_SETSIZE];

void
svc_run(*void*);

bool_t
svc_sendreply(*SVCXPRT *xprt*, *xdrproc_t outproc*, *void *out*);

DESCRIPTION

These routines are part of the RPC library which allows C language programs to make procedure calls on other machines across the network.

These routines are associated with the server side of the RPC mechanism. Some of them are called by the server side dispatch function, while others (such as **svc_run**()) are called when the server is initiated.

Routines

See rpc(3) for the definition of the *SVCXPRT* data structure.

svc_dg_enablecache() This function allocates a duplicate request cache for the service endpoint *xprt*, large enough to hold *cache_size* entries. Once enabled, there is no way to disable caching. This routine returns 0 if space necessary for a cache of the given size was successfully allocated, and 1 otherwise.

svc_exit() This function, when called by any of the RPC server procedure or otherwise, causes **svc_run**() to return.

As currently implemented, **svc_exit**() zeroes the *svc_fdset* global variable. If RPC server activity is to be resumed, services must be reregistered with the RPC library either through one of the rpc_svc_create(3) functions, or using **xprt_register**(). The **svc_exit**() function has global scope and ends all RPC server activity.

fd_set svc_fdset A global variable reflecting the RPC server's read file descriptor bit mask; it is suitable as an argument to the select(2) system call. This is only of interest if service implementors do not call **svc_run**(), but rather do their own asynchronous event processing. This variable is read-only (do not pass its address to select(2)!), yet it may change after calls to **svc_getreqset**() or any creation routines.

svc_freeargs() A function macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs**(). This routine returns TRUE if the results were successfully freed, and FALSE otherwise.

svc_getargs() A function macro that decodes the arguments of an RPC request associated with the RPC service transport handle *xprt*. The *in* argument is the address where the arguments will be placed; *inproc* is the XDR routine used to decode the arguments. This routine returns TRUE if decoding succeeds, and FALSE otherwise.

svc_getreq_common() This routine is called to handle a request on the given file descriptor.

svc_getreq_poll() This routine is only of interest if a service implementor does not call **svc_run**(), but instead implements custom asynchronous event processing. It is called when poll(2) has determined that an RPC request has arrived on some RPC file descriptors; *pollretval* is the return value from poll(2) and *pfdp* is the array of *pollfd* structures on which the poll(2) was done. It is assumed to be an array large enough to contain the maximal number of descriptors allowed.

svc_getreqset() This routine is only of interest if a service implementor does not call **svc_run**(), but instead implements custom asynchronous event processing. It is called when poll(2) has determined that an RPC request has arrived on some RPC file descriptors; *rdfds* is the resultant read file descriptor bit mask. The routine returns when all file descriptors associated with the value of *rdfds* have been serviced.

svc_getrpccaller() The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle *xprt*.

__svc_getcallercreds() *Warning*: this macro is specific to FreeBSD and thus not portable. This macro returns a pointer to a *cmsgcred* structure, defined in <*sys/socket.h*>, identifying the calling client. This only works if the client is calling the server over an AF_LOCAL socket.

struct pollfd svc_pollset[FD_SETSIZE];
svc_pollset is an array of *pollfd* structures derived from *svc_fdset[]*. It is suitable as an argument to the poll(2) system call. The derivation of *svc_pollset* from *svc_fdset* is made in the current implementation in **svc_run**(). Service implementors who do not call **svc_run**() and who wish to use this array must perform this derivation themselves.

svc_run() This routine never returns. It waits for RPC requests to arrive, and calls the appropriate service procedure using **svc_getreq_poll**() when one arrives. This procedure is usually waiting for the poll(2) system call to return.

svc_sendreply() Called by an RPC service's dispatch routine to send the results of a remote procedure call. The *xprt* argument is the request's associated transport handle; *outproc* is the XDR routine which is used to encode the results; and *out* is the address of the results. This routine returns TRUE if it succeeds, FALSE otherwise.

SEE ALSO

poll(2), select(2), rpc(3), rpc_svc_create(3), rpc_svc_err(3), rpc_svc_reg(3)

NAME

swab - swap adjacent bytes

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

void
swab(*const void * restrict src*, *void * restrict dst*, *ssize_t len*);

DESCRIPTION

The function **swab**() copies *len* bytes from the location referenced by *src* to the location referenced by *dst*, swapping adjacent bytes.

The argument *len* must be an even number. If *len* is less than zero, nothing will be done.

SEE ALSO

bzero(3), memset(3)

HISTORY

A **swab**() function appeared in Version 7 AT&T UNIX.

NAME

swapon, **swapoff** - control devices for interleaved paging/swapping

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
swapon(*const char *special*);

int
swapoff(*const char *special*);

DESCRIPTION

The **swapon**() system call makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

The **swapoff**() system call disables paging and swapping on the given device. All associated swap metadata are deallocated, and the device is made available for other purposes.

RETURN VALUES

If an error has occurred, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

Both **swapon**() and **swapoff**() can fail if:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named device does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The caller is not the super-user.

[EFAULT] The *special* argument points outside the process's allocated address space.

Additionally, **swapon**() can fail for the following reasons:

[ENOTBLK] The *special* argument is not a block device.

[EBUSY] The device specified by *special* has already been made available for swapping

[ENXIO] The major device number of *special* is out of range (this indicates no device driver exists for the associated hardware).

[EIO] An I/O error occurred while opening the swap device.

Lastly, **swapoff**() can fail if:

[EINVAL] The system is not currently swapping to *special*.

[ENOMEM] Not enough virtual memory is available to safely disable paging and swapping to the given device.

SEE ALSO

config(8), swapon(8), sysctl(8)

HISTORY

The **swapon**() system call appeared in 4.0BSD. The **swapoff**() system call appeared in FreeBSD 5.0.

NAME

symlink, **symlinkat** - make symbolic link to a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
symlink(*const char *name1*, *const char *name2*);

int
symlinkat(*const char *name1*, *int fd*, *const char *name2*);

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

The **symlinkat**() system call is equivalent to **symlink**() except in the case where *name2* specifies a relative path. In this case the symbolic link is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. If **symlinkat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to **symlink**().

RETURN VALUES

The **symlink**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The symbolic link succeeds unless:

[ENOTDIR] A component of the *name2* path prefix is not a directory.

[ENAMETOOLONG]
A component of the *name2* pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.

[ENOENT] A component of the *name2* path prefix does not exist.

[EACCES] A component of the *name2* path prefix denies search permission, or write permission is denied on the parent directory of the file to be created.

[ELOOP] Too many symbolic links were encountered in translating the *name2* path name.

[EEXIST] The path name pointed at by the *name2* argument already exists.

[EPERM] The parent directory of the file named by *name2* has its immutable flag set, see the chflags(2) manual page for more information.

[EIO] An I/O error occurred while making the directory entry for *name2*, or allocating the inode for *name2*, or writing out the link contents of *name2*.

[EROFS] The file *name2* would reside on a read-only file system.

[ENOSPC] The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.

[ENOSPC] The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.

[ENOSPC] There are no free inodes on the file system on which the symbolic link is being created.

[EDQUOT] The directory in which the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

[EDQUOT] The new symbolic link cannot be created because the user's quota of disk blocks on the file system that will contain the symbolic link has been exhausted.

[EDQUOT] The user's quota of inodes on the file system on which the symbolic link is being created has been exhausted.

[EIO] An I/O error occurred while making the directory entry or allocating the inode.

[EFAULT] The *name1* or *name2* argument points outside the process's allocated address space.

In addition to the errors returned by the **symlink**(), the **symlinkat**() may fail if:

[EBADF] The *name2* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTDIR] The *name2* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

ln(1), chflags(2), link(2), lstat(2), readlink(2), unlink(2), symlink(7)

STANDARDS

The **symlinkat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **symlink**() system call appeared in 4.2BSD. The **symlinkat**() system call appeared in FreeBSD 8.0.

NAME

sync - schedule file system updates

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

void
sync(*void*);

DESCRIPTION

The **sync**() system call forces a write of dirty (modified) buffers in the block buffer cache out to disk. The kernel keeps this information in core to reduce the number of disk I/O transfers required by the system. As information in the cache is lost after a system crash, a **sync**() system call is issued frequently by the kernel process syncer(4) (about every 30 seconds).

The fsync(2) system call may be used to synchronize individual file descriptor attributes.

SEE ALSO

fsync(2), syncer(4), sync(8)

HISTORY

The **sync**() function appeared in Version 3 AT&T UNIX.

BUGS

The **sync**() system call may return before the buffers are completely flushed.

NAME

sysarch - architecture-dependent system call

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <machine/sysarch.h>

int
sysarch(*int number*, *void *args*);

DESCRIPTION

The **sysarch**() system call performs the architecture-dependent function specified by *number* with the arguments specified by the *args* pointer. The *args* argument is a pointer to a structure defining the actual arguments of the function. Symbolic constants and argument structures for the architecture-dependent functions can be found in the header file <*machine/sysarch.h*>.

The **sysarch**() system call should never be called directly by user programs. Instead, they should access its functions using the architecture-dependent library.

RETURN VALUES

See the manual pages for specific architecture-dependent system calls for information about their return values.

SEE ALSO

i386_get_ioperm(2), i386_get_ldt(2), i386_vm86(2)

HISTORY

This manual page was taken from NetBSD.

NAME

sysconf - get configurable system variables

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

long
sysconf(*int name*);

DESCRIPTION

This interface is defined by IEEE Std 1003.1-1988 ("POSIX.1"). A far more complete interface is available using sysctl(3).

The **sysconf**() function provides a method for applications to determine the current value of a configurable system limit or option variable. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file <*unistd.h*>. Shell programmers who need access to these parameters should use the getconf(1) utility.

The available values are as follows:

_SC_ARG_MAX
The maximum bytes of argument to execve(2).

_SC_CHILD_MAX
The maximum number of simultaneous processes per user id.

_SC_CLK_TCK
The frequency of the statistics clock in ticks per second.

_SC_IOV_MAX
The maximum number of elements in the I/O vector used by readv(2), writev(2), recvmsg(2), and sendmsg(2).

_SC_NGROUPS_MAX
The maximum number of supplemental groups.

_SC_NPROCESSORS_CONF

The number of processors configured.

_SC_NPROCESSORS_ONLN
The number of processors currently online.

_SC_OPEN_MAX
One more than the maximum value the system may assign to a new file descriptor.

_SC_PAGESIZE
The size of a system page in bytes.

_SC_PAGE_SIZE
Equivalent to _SC_PAGESIZE.

_SC_STREAM_MAX
The minimum maximum number of streams that a process may have open at any one time.

_SC_TZNAME_MAX
The minimum maximum number of types supported for the name of a timezone.

_SC_JOB_CONTROL
Return 1 if job control is available on this system, otherwise -1.

_SC_SAVED_IDS
Returns 1 if saved set-group and saved set-user ID is available, otherwise -1.

_SC_VERSION
The version of IEEE Std 1003.1 ("POSIX.1") with which the system attempts to comply.

_SC_BC_BASE_MAX
The maximum ibase/obase values in the bc(1) utility.

_SC_BC_DIM_MAX
The maximum array size in the bc(1) utility.

_SC_BC_SCALE_MAX
The maximum scale value in the bc(1) utility.

_SC_BC_STRING_MAX
The maximum string length in the bc(1) utility.

_SC_COLL_WEIGHTS_MAX
: The maximum number of weights that can be assigned to any entry of the LC_COLLATE order keyword in the locale definition file.

_SC_EXPR_NEST_MAX
: The maximum number of expressions that can be nested within parenthesis by the expr(1) utility.

_SC_LINE_MAX
: The maximum length in bytes of a text-processing utility's input line.

_SC_RE_DUP_MAX
: The maximum number of repeated occurrences of a regular expression permitted when using interval notation.

_SC_2_VERSION
: The version of IEEE Std 1003.2 ("POSIX.2") with which the system attempts to comply.

_SC_2_C_BIND
: Return 1 if the system's C-language development facilities support the C-Language Bindings Option, otherwise -1.

_SC_2_C_DEV
: Return 1 if the system supports the C-Language Development Utilities Option, otherwise -1.

_SC_2_CHAR_TERM
: Return 1 if the system supports at least one terminal type capable of all operations described in IEEE Std 1003.2 ("POSIX.2"), otherwise -1.

_SC_2_FORT_DEV
: Return 1 if the system supports the FORTRAN Development Utilities Option, otherwise -1.

_SC_2_FORT_RUN
: Return 1 if the system supports the FORTRAN Runtime Utilities Option, otherwise -1.

_SC_2_LOCALEDEF
: Return 1 if the system supports the creation of locales, otherwise -1.

_SC_2_SW_DEV
: Return 1 if the system supports the Software Development Utilities Option, otherwise -1.

_SC_2_UPE
: Return 1 if the system supports the User Portability Utilities Option, otherwise -1.

_SC_AIO_LISTIO_MAX
: Maximum number of I/O operations in a single list I/O call supported.

_SC_AIO_MAX
: Maximum number of outstanding asynchronous I/O operations supported.

_SC_AIO_PRIO_DELTA_MAX
: The maximum amount by which a process can decrease its asynchronous I/O priority level from its own scheduling priority.

_SC_DELAYTIMER_MAX
: Maximum number of timer expiration overruns.

_SC_MQ_OPEN_MAX
: The maximum number of open message queue descriptors a process may hold.

_SC_RTSIG_MAX
: Maximum number of realtime signals reserved for application use.

_SC_SEM_NSEMS_MAX
: Maximum number of semaphores that a process may have.

_SC_SEM_VALUE_MAX
: The maximum value a semaphore may have.

_SC_SIGQUEUE_MAX
: Maximum number of queued signals that a process may send and have pending at the receiver(s) at any time.

_SC_TIMER_MAX
: Maximum number of timers per process supported.

_SC_GETGR_R_SIZE_MAX
: Suggested initial value for the size of the group entry buffer.

_SC_GETPW_R_SIZE_MAX
: Suggested initial value for the size of the password entry buffer.

_SC_HOST_NAME_MAX
: Maximum length of a host name (not including the terminating null) as returned from the **gethostname**() function.

_SC_LOGIN_NAME_MAX
: Maximum length of a login name.

_SC_THREAD_STACK_MIN
: Minimum size in bytes of thread stack storage.

_SC_THREAD_THREADS_MAX
: Maximum number of threads that can be created per process.

_SC_TTY_NAME_MAX
: Maximum length of terminal device name.

_SC_SYMLOOP_MAX
: Maximum number of symbolic links that can be reliably traversed in the resolution of a pathname in the absence of a loop.

_SC_ATEXIT_MAX
: Maximum number of functions that may be registered with **atexit**().

_SC_XOPEN_VERSION
: An integer value greater than or equal to 4, indicating the version of the X/Open Portability Guide to which this system conforms.

_SC_XOPEN_XCU_VERSION
: An integer value indicating the version of the XCU Specification to which this system conforms.

These values also exist, but may not be standard:

_SC_CPUSET_SIZE
: Size of the kernel cpuset.

_SC_PHYS_PAGES
: The number of pages of physical memory. Note that it is possible that the product of this value and the value of _SC_PAGESIZE will overflow a *long* in some configurations on a 32bit machine.

RETURN VALUES

If the call to **sysconf**() is not successful, -1 is returned and *errno* is set appropriately. Otherwise, if the variable is associated with functionality that is not supported, -1 is returned and *errno* is not modified. Otherwise, the current variable value is returned.

ERRORS

The **sysconf**() function may fail and set *errno* for any of the errors specified for the library function sysctl(3). In addition, the following error may be reported:

[EINVAL] The value of the *name* argument is invalid.

SEE ALSO

getconf(1), pathconf(2), confstr(3), sysctl(3)

STANDARDS

Except for the fact that values returned by **sysconf**() may change over the lifetime of the calling process, this function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

HISTORY

The **sysconf**() function first appeared in 4.4BSD.

BUGS

The value for _SC_STREAM_MAX is a minimum maximum, and required to be the same as ANSI C's FOPEN_MAX, so the returned value is a ridiculously small and misleading number.

NAME

sysctl, **sysctlbyname**, **sysctlnametomib** - get or set system information

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/sysctl.h>

int
sysctl(*const int *name*, *u_int namelen*, *void *oldp*, *size_t *oldlenp*, *const void *newp*, *size_t newlen*);

int
sysctlbyname(*const char *name*, *void *oldp*, *size_t *oldlenp*, *const void *newp*, *size_t newlen*);

int
sysctlnametomib(*const char *name*, *int *mibp*, *size_t *sizep*);

DESCRIPTION

The **sysctl**() function retrieves system information and allows processes with appropriate privileges to set system information. The information available from **sysctl**() consists of integers, strings, and tables. Information may be retrieved and set from the command interface using the sysctl(8) utility.

Unless explicitly noted below, **sysctl**() returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to **sysctl**() are serialized to avoid deadlock.

The state is described using a ‘‘Management Information Base’’ (MIB) style name, listed in *name*, which is a *namelen* length array of integers.

The **sysctlbyname**() function accepts an ASCII representation of the name and internally looks up the integer name vector. Apart from that, it behaves the same as the standard **sysctl**() function.

The information is copied into the buffer specified by *oldp*. The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call and after a call that returns with the error code ENOMEM. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns with the error code ENOMEM. If the old value is not desired, *oldp* and *oldlenp* should be set to NULL.

The size of the available data can be determined by calling **sysctl**() with the NULL argument for *oldp*. The size of the available data will be returned in the location pointed to by *oldlenp*. For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to NULL and *newlen* set to 0.

The **sysctlnametomib**() function accepts an ASCII representation of the name, looks up the integer name vector, and returns the numeric representation in the mib array pointed to by *mibp*. The number of elements in the mib array is given by the location specified by *sizep* before the call, and that location gives the number of entries copied after a successful call. The resulting *mib* and *size* may be used in subsequent **sysctl**() calls to get the data associated with the requested ASCII name. This interface is intended for use by applications that want to repeatedly request the same variable (the **sysctl**() function runs in about a third the time as the same request made via the **sysctlbyname**() function). The **sysctlnametomib**() function is also useful for fetching mib prefixes and then adding a final component. For example, to fetch process information for processes with pid's less than 100:

```
int i, mib[4];
size_t len;
struct kinfo_proc kp;

/* Fill out the first three components of the mib */
len = 4;
sysctlnametomib("kern.proc.pid", mib, &len);

/* Fetch and print entries for pid's < 100 */
for (i = 0; i < 100; i++) {
        mib[3] = i;
        len = sizeof(kp);
        if (sysctl(mib, 4, &kp, &len, NULL, 0) == -1)
                perror("sysctl");
        else if (len > 0)
                printkproc(&kp);
}
```

The top level names are defined with a CTL_ prefix in <*sys/sysctl.h*>, and are as follows. The next and subsequent levels down are found in the include files listed here, and described in separate sections below.

Name	Next Level Names	Description
CTL_DEBUG	<*sys/sysctl.h*>	Debugging
CTL_VFS	<*sys/mount.h*>	File system
CTL_HW	<*sys/sysctl.h*>	Generic CPU, I/O
CTL_KERN	<*sys/sysctl.h*>	High kernel limits
CTL_MACHDEP	<*sys/sysctl.h*>	Machine dependent
CTL_NET	<*sys/socket.h*>	Networking
CTL_USER	<*sys/sysctl.h*>	User-level
CTL_VM	<*vm/vm_param.h*>	Virtual memory

For example, the following retrieves the maximum number of processes allowed in the system:

```
int mib[2], maxproc;
size_t len;

mib[0] = CTL_KERN;
mib[1] = KERN_MAXPROC;
len = sizeof(maxproc);
sysctl(mib, 2, &maxproc, &len, NULL, 0);
```

To retrieve the standard search path for the system utilities:

```
int mib[2];
size_t len;
char *p;

mib[0] = CTL_USER;
mib[1] = USER_CS_PATH;
sysctl(mib, 2, NULL, &len, NULL, 0);
p = malloc(len);
sysctl(mib, 2, p, &len, NULL, 0);
```

CTL_DEBUG

The debugging variables vary from system to system. A debugging variable may be added or deleted without need to recompile **sysctl**() to know about it. Each time it runs, **sysctl**() gets the list of debugging variables from the kernel and displays their current values. The system defines twenty (*struct ctldebug*) variables named *debug0* through *debug19*. They are declared as separate variables so that they can be individually initialized at the location of their associated variable. The loader prevents multiple use of the same variable by issuing errors if a variable is initialized in more than one place. For example, to export the variable *dospecialcheck* as a debugging variable, the following declaration would be used:

```
int dospecialcheck = 1;
struct ctldebug debug5 = { "dospecialcheck", &dospecialcheck };
```

CTL_VFS

A distinguished second level name, VFS_GENERIC, is used to get general information about all file systems. One of its third level identifiers is VFS_MAXTYPENUM that gives the highest valid file system type number. Its other third level identifier is VFS_CONF that returns configuration information about the file system type given as a fourth level identifier (see getvfsbyname(3) as an example of its use). The remaining second level identifiers are the file system type number returned by a statfs(2) call or from VFS_CONF. The third level identifiers available for each file system are given in the header file that defines the mount argument structure for that file system.

CTL_HW

The string and integer information available for the CTL_HW level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second Level Name	Type	Changeable
HW_MACHINE	string	no
HW_MODEL	string	no
HW_NCPU	integer	no
HW_BYTEORDER	integer	no
HW_PHYSMEM	integer	no
HW_USERMEM	integer	no
HW_PAGESIZE	integer	no
HW_FLOATINGPT	integer	no
HW_MACHINE_ARCH	string	no
HW_REALMEM	integer	no

HW_MACHINE
: The machine class.

HW_MODEL
: The machine model

HW_NCPU
: The number of cpus.

HW_BYTEORDER
: The byteorder (4321 or 1234).

HW_PHYSMEM
The bytes of physical memory.

HW_USERMEM
The bytes of non-kernel memory.

HW_PAGESIZE
The software page size.

HW_FLOATINGPT
Nonzero if the floating point support is in hardware.

HW_MACHINE_ARCH
The machine dependent architecture type.

HW_REALMEM
The bytes of real memory.

CTL_KERN

The string and integer information available for the CTL_KERN level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value. The types of data currently available are process information, system vnodes, the open file entries, routing table entries, virtual memory statistics, load average history, and clock rate information.

Second Level Name	Type	Changeable
KERN_ARGMAX	integer	no
KERN_BOOTFILE	string	yes
KERN_BOOTTIME	struct timeval	no
KERN_CLOCKRATE	struct clockinfo	no
KERN_FILE	struct xfile	no
KERN_HOSTID	integer	yes
KERN_HOSTUUID	string	yes
KERN_HOSTNAME	string	yes
KERN_JOB_CONTROL	integer	no
KERN_MAXFILES	integer	yes
KERN_MAXFILESPERPROC	integer	yes
KERN_MAXPROC	integer	no
KERN_MAXPROCPERUID	integer	yes
KERN_MAXVNODES	integer	yes
KERN_NGROUPS	integer	no

KERN_NISDOMAINNAME	string	yes
KERN_OSRELDATE	integer	no
KERN_OSRELEASE	string	no
KERN_OSREV	integer	no
KERN_OSTYPE	string	no
KERN_POSIX1	integer	no
KERN_PROC	node	not applicable
KERN_PROF	node	not applicable
KERN_QUANTUM	integer	yes
KERN_SAVED_IDS	integer	no
KERN_SECURELVL	integer	raise only
KERN_UPDATEINTERVAL	integer	no
KERN_VERSION	string	no
KERN_VNODE	struct xvnode	no

KERN_ARGMAX
: The maximum bytes of argument to execve(2).

KERN_BOOTFILE
: The full pathname of the file from which the kernel was loaded.

KERN_BOOTTIME
: A *struct timeval* structure is returned. This structure contains the time that the system was booted.

KERN_CLOCKRATE
: A *struct clockinfo* structure is returned. This structure contains the clock, statistics clock and profiling clock frequencies, the number of micro-seconds per hz tick and the skew rate.

KERN_FILE
: Return the entire file table. The returned data consists of an array of *struct xfile*, whose size depends on the current number of such objects in the system.

KERN_HOSTID
: Get or set the host ID.

KERN_HOSTUUID
: Get or set the host's universally unique identifier (UUID).

KERN_HOSTNAME

Get or set the hostname.

KERN_JOB_CONTROL
Return 1 if job control is available on this system, otherwise 0.

KERN_MAXFILES
The maximum number of files that may be open in the system.

KERN_MAXFILESPERPROC
The maximum number of files that may be open for a single process. This limit only applies to processes with an effective uid of nonzero at the time of the open request. Files that have already been opened are not affected if the limit or the effective uid is changed.

KERN_MAXPROC
The maximum number of concurrent processes the system will allow.

KERN_MAXPROCPERUID
The maximum number of concurrent processes the system will allow for a single effective uid. This limit only applies to processes with an effective uid of nonzero at the time of a fork request. Processes that have already been started are not affected if the limit is changed.

KERN_MAXVNODES
The maximum number of vnodes available on the system.

KERN_NGROUPS
The maximum number of supplemental groups.

KERN_NISDOMAINNAME
The name of the current YP/NIS domain.

KERN_OSRELDATE
The kernel release version in the format *MmmRxx*, where *M* is the major version, *mm* is the two digit minor version, *R* is 0 if release branch, otherwise 1, and *xx* is updated when the available APIs change.

The userland release version is available from <*osreldate.h*>; parse this file if you need to get the release version of the currently installed userland.

KERN_OSRELEASE
The system release string.

KERN_OSREV
The system revision string.

KERN_OSTYPE
The system type string.

KERN_POSIX1
The version of IEEE Std 1003.1 ("POSIX.1") with which the system attempts to comply.

KERN_PROC
Return selected information about specific running processes.

For the following names, an array of *struct kinfo_proc* structures is returned, whose size depends on the current number of such objects in the system.

Third Level Name	**Fourth Level**
KERN_PROC_ALL	None
KERN_PROC_PID	A process ID
KERN_PROC_PGRP	A process group
KERN_PROC_TTY	A tty device
KERN_PROC_UID	A user ID
KERN_PROC_RUID	A real user ID

If the third level name is KERN_PROC_ARGS then the command line argument array is returned in a flattened form, i.e., zero-terminated arguments follow each other. The total size of array is returned. It is also possible for a process to set its own process title this way. If the third level name is KERN_PROC_PATHNAME, the path of the process' text file is stored. For KERN_PROC_PATHNAME, a process ID of -1 implies the current process.

Third Level Name	**Fourth Level**
KERN_PROC_ARGS	A process ID
KERN_PROC_PATHNAME	A process ID

KERN_PROF
Return profiling information about the kernel. If the kernel is not compiled for profiling, attempts to retrieve any of the KERN_PROF values will fail with ENOENT. The third level names for the string and integer profiling information is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Third Level Name	**Type**	**Changeable**

GPROF_STATE	integer	yes
GPROF_COUNT	u_short[]	yes
GPROF_FROMS	u_short[]	yes
GPROF_TOS	struct tostruct	yes
GPROF_GMONPARAM	struct gmonparam	no

The variables are as follows:

GPROF_STATE
: Returns GMON_PROF_ON or GMON_PROF_OFF to show that profiling is running or stopped.

GPROF_COUNT
: Array of statistical program counter counts.

GPROF_FROMS
: Array indexed by program counter of call-from points.

GPROF_TOS
: Array of *struct tostruct* describing destination of calls and their counts.

GPROF_GMONPARAM
: Structure giving the sizes of the above arrays.

KERN_QUANTUM
: The maximum period of time, in microseconds, for which a process is allowed to run without being preempted if other processes are in the run queue.

KERN_SAVED_IDS
: Returns 1 if saved set-group and saved set-user ID is available.

KERN_SECURELVL
: The system security level. This level may be raised by processes with appropriate privilege. It may not be lowered.

KERN_VERSION
: The system version string.

KERN_VNODE
: Return the entire vnode table. Note, the vnode table is not necessarily a consistent snapshot of

the system. The returned data consists of an array whose size depends on the current number of such objects in the system. Each element of the array consists of a *struct xvnode*.

CTL_NET

The string and integer information available for the CTL_NET level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second Level Name	Type	Changeable
PF_ROUTE	routing messages	no
PF_INET	IPv4 values	yes
PF_INET6	IPv6 values	yes

PF_ROUTE

Return the entire routing table or a subset of it. The data is returned as a sequence of routing messages (see route(4) for the header file, format and meaning). The length of each message is contained in the message header.

The third level name is a protocol number, which is currently always 0. The fourth level name is an address family, which may be set to 0 to select all address families. The fifth, sixth, and seventh level names are as follows:

Fifth level	Sixth Level	Seventh Level
NET_RT_FLAGS		
	rtflags	None
NET_RT_DUMP		
	None	None or fib number
NET_RT_IFLIST		
	0 or if_index	None
NET_RT_IFMALIST		
	0 or if_index	None
NET_RT_IFLISTL		
	0 or if_index	None

The NET_RT_IFMALIST name returns information about multicast group memberships on all interfaces if 0 is specified, or for the interface specified by *if_index*.

The NET_RT_IFLISTL is like NET_RT_IFLIST, just returning message header structs with additional fields allowing the interface to be extended without breaking binary compatibility. The NET_RT_IFLISTL uses 'l' versions of the message header structures: *struct if_msghdrl* and *struct ifa_msghdrl*.

PF_INET

Get or set various global information about the IPv4 (Internet Protocol version 4). The third level name is the protocol. The fourth level name is the variable name. The currently defined protocols and names are:

Protocol	Variable	Type	Changeable
icmp	bmcastecho	integer	yes
icmp	maskrepl	integer	yes
ip	forwarding	integer	yes
ip	redirect	integer	yes
ip	ttl	integer	yes
udp	checksum	integer	yes

The variables are as follows:

icmp.bmcastecho

Returns 1 if an ICMP echo request to a broadcast or multicast address is to be answered.

icmp.maskrepl

Returns 1 if ICMP network mask requests are to be answered.

ip.forwarding

Returns 1 when IP forwarding is enabled for the host, meaning that the host is acting as a router.

ip.redirect

Returns 1 when ICMP redirects may be sent by the host. This option is ignored unless the host is routing IP packets, and should normally be enabled on all systems.

ip.ttl The maximum time-to-live (hop count) value for an IP packet sourced by the system. This value applies to normal transport protocols, not to ICMP.

udp.checksum

Returns 1 when UDP checksums are being computed and checked. Disabling UDP checksums is strongly discouraged.

For variables net.inet.*.ipsec, please refer to ipsec(4).

PF_INET6

Get or set various global information about the IPv6 (Internet Protocol version 6). The third level name is the protocol. The fourth level name is the variable name.

For variables net.inet6.* please refer to inet6(4). For variables net.inet6.*.ipsec6, please refer to ipsec(4).

CTL_USER

The string and integer information available for the CTL_USER level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second Level Name	Type	Changeable
USER_BC_BASE_MAX	integer	no
USER_BC_DIM_MAX	integer	no
USER_BC_SCALE_MAX	integer	no
USER_BC_STRING_MAX	integer	no
USER_COLL_WEIGHTS_MAX	integer	no
USER_CS_PATH	string	no
USER_EXPR_NEST_MAX	integer	no
USER_LINE_MAX	integer	no
USER_POSIX2_CHAR_TERM	integer	no
USER_POSIX2_C_BIND	integer	no
USER_POSIX2_C_DEV	integer	no
USER_POSIX2_FORT_DEV	integer	no
USER_POSIX2_FORT_RUN	integer	no
USER_POSIX2_LOCALEDEF	integer	no
USER_POSIX2_SW_DEV	integer	no
USER_POSIX2_UPE	integer	no
USER_POSIX2_VERSION	integer	no
USER_RE_DUP_MAX	integer	no
USER_STREAM_MAX	integer	no
USER_TZNAME_MAX	integer	no

USER_BC_BASE_MAX

The maximum ibase/obase values in the bc(1) utility.

USER_BC_DIM_MAX

The maximum array size in the bc(1) utility.

USER_BC_SCALE_MAX

The maximum scale value in the bc(1) utility.

USER_BC_STRING_MAX
: The maximum string length in the bc(1) utility.

USER_COLL_WEIGHTS_MAX
: The maximum number of weights that can be assigned to any entry of the LC_COLLATE order keyword in the locale definition file.

USER_CS_PATH
: Return a value for the PATH environment variable that finds all the standard utilities.

USER_EXPR_NEST_MAX
: The maximum number of expressions that can be nested within parenthesis by the expr(1) utility.

USER_LINE_MAX
: The maximum length in bytes of a text-processing utility's input line.

USER_POSIX2_CHAR_TERM
: Return 1 if the system supports at least one terminal type capable of all operations described in IEEE Std 1003.2 ("POSIX.2"), otherwise 0.

USER_POSIX2_C_BIND
: Return 1 if the system's C-language development facilities support the C-Language Bindings Option, otherwise 0.

USER_POSIX2_C_DEV
: Return 1 if the system supports the C-Language Development Utilities Option, otherwise 0.

USER_POSIX2_FORT_DEV
: Return 1 if the system supports the FORTRAN Development Utilities Option, otherwise 0.

USER_POSIX2_FORT_RUN
: Return 1 if the system supports the FORTRAN Runtime Utilities Option, otherwise 0.

USER_POSIX2_LOCALEDEF
: Return 1 if the system supports the creation of locales, otherwise 0.

USER_POSIX2_SW_DEV
: Return 1 if the system supports the Software Development Utilities Option, otherwise 0.

USER_POSIX2_UPE
Return 1 if the system supports the User Portability Utilities Option, otherwise 0.

USER_POSIX2_VERSION
The version of IEEE Std 1003.2 ("POSIX.2") with which the system attempts to comply.

USER_RE_DUP_MAX
The maximum number of repeated occurrences of a regular expression permitted when using interval notation.

USER_STREAM_MAX
The minimum maximum number of streams that a process may have open at any one time.

USER_TZNAME_MAX
The minimum maximum number of types supported for the name of a timezone.

CTL_VM

The string and integer information available for the CTL_VM level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second Level Name	Type	Changeable
VM_LOADAVG	struct loadavg	no
VM_TOTAL	struct vmtotal	no
VM_SWAPPING_ENABLED	integer	maybe
VM_V_FREE_MIN	integer	yes
VM_V_FREE_RESERVED	integer	yes
VM_V_FREE_TARGET	integer	yes
VM_V_INACTIVE_TARGET	integer	yes
VM_V_PAGEOUT_FREE_MIN	integer	yes
VM_OVERCOMMIT	integer	yes

VM_LOADAVG
Return the load average history. The returned data consists of a *struct loadavg*.

VM_TOTAL
Return the system wide virtual memory statistics. The returned data consists of a *struct vmtotal*.

VM_SWAPPING_ENABLED

1 if process swapping is enabled or 0 if disabled. This variable is permanently set to 0 if the kernel was built with swapping disabled.

VM_V_FREE_MIN
Minimum amount of memory (cache memory plus free memory) required to be available before a process waiting on memory will be awakened.

VM_V_FREE_RESERVED
Processes will awaken the pageout daemon and wait for memory if the number of free and cached pages drops below this value.

VM_V_FREE_TARGET
The total amount of free memory (including cache memory) that the pageout daemon tries to maintain.

VM_V_INACTIVE_TARGET
The desired number of inactive pages that the pageout daemon should achieve when it runs. Inactive pages can be quickly inserted into process address space when needed.

VM_V_PAGEOUT_FREE_MIN
If the amount of free and cache memory falls below this value, the pageout daemon will enter "memory conserving mode" to avoid deadlock.

VM_OVERCOMMIT
Overcommit behaviour, as described in tuning(7).

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

FILES

<*sys/sysctl.h*>	definitions for top level identifiers, second level kernel and hardware identifiers, and user level identifiers
<*sys/socket.h*>	definitions for second level network identifiers
<*sys/gmon.h*>	definitions for third level profiling identifiers
<*vm/vm_param.h*>	definitions for second level virtual memory identifiers
<*netinet/in.h*>	definitions for third level IPv4/IPv6 identifiers and fourth level IPv4/v6 identifiers
<*netinet/icmp_var.h*>	definitions for fourth level ICMP identifiers
<*netinet/icmp6.h*>	definitions for fourth level ICMPv6 identifiers

<*netinet/udp_var.h*> definitions for fourth level UDP identifiers

ERRORS

The following errors may be reported:

[EFAULT] The buffer *name*, *oldp*, *newp*, or length pointer *oldlenp* contains an invalid address.

[EINVAL] The *name* array is less than two or greater than CTL_MAXNAME.

[EINVAL] A non-null *newp* is given and its specified length in *newlen* is too large or too small.

[ENOMEM] The length pointed to by *oldlenp* is too short to hold the requested value.

[ENOMEM] The smaller of either the length pointed to by *oldlenp* or the estimated size of the returned data exceeds the system limit on locked memory.

[ENOMEM] Locking the buffer *oldp*, or a portion of the buffer if the estimated size of the data to be returned is smaller, would cause the process to exceed its per-process locked memory limit.

[ENOTDIR] The *name* array specifies an intermediate rather than terminal name.

[EISDIR] The *name* array specifies a terminal name, but the actual name is not terminal.

[ENOENT] The *name* array specifies a value that is unknown.

[EPERM] An attempt is made to set a read-only value.

[EPERM] A process without appropriate privilege attempts to set a value.

SEE ALSO

confstr(3), kvm(3), sysconf(3), sysctl(8)

HISTORY

The **sysctl**() function first appeared in 4.4BSD.

NAME

system - pass a command to the shell

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
system(*const char *string*);

DESCRIPTION

The **system**() function hands the argument *string* to the command interpreter sh(1). The calling process waits for the shell to finish executing the command, ignoring SIGINT and SIGQUIT, and blocking SIGCHLD.

If *string* is a NULL pointer, **system**() will return non-zero if the command interpreter sh(1) is available, and zero if it is not.

RETURN VALUES

The **system**() function returns the exit status of the shell as returned by waitpid(2), or -1 if an error occurred when invoking fork(2) or waitpid(2). A return value of 127 means the execution of the shell failed.

SEE ALSO

sh(1), execve(2), fork(2), waitpid(2), popen(3), posix_spawn(3)

STANDARDS

The **system**() function conforms to ISO/IEC 9899:1990 ("ISO C90") and is expected to be IEEE Std 1003.2 ("POSIX.2") compatible.

SECURITY CONSIDERATIONS

The **system**() function is easily misused in a manner that enables a malicious user to run arbitrary command, because all meta-characters supported by sh(1) would be honored. User supplied parameters should always be carefully santized before they appear in *string*.

NAME

tcsendbreak, **tcdrain**, **tcflush**, **tcflow** - line control functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <termios.h>

int
tcdrain(*int fd*);

int
tcflow(*int fd*, *int action*);

int
tcflush(*int fd*, *int action*);

int
tcsendbreak(*int fd*, *int len*);

DESCRIPTION

The **tcdrain**() function waits until all output written to the terminal referenced by *fd* has been transmitted to the terminal.

The **tcflow**() function suspends transmission of data to or the reception of data from the terminal referenced by *fd* depending on the value of *action*. The value of *action* must be one of the following:

TCOOFF
Suspend output.

TCOON Restart suspended output.

TCIOFF Transmit a STOP character, which is intended to cause the terminal to stop transmitting data to the system. (See the description of IXOFF in the 'Input Modes' section of termios(4)).

TCION Transmit a START character, which is intended to cause the terminal to start transmitting data to the system. (See the description of IXOFF in the 'Input Modes' section of termios(4)).

The **tcflush**() function discards any data written to the terminal referenced by *fd* which has not been

transmitted to the terminal, or any data received from the terminal but not yet read, depending on the value of *action*. The value of *action* must be one of the following:

TCIFLUSH Flush data received but not read.

TCOFLUSH Flush data written but not transmitted.

TCIOFLUSH Flush both data received but not read and data written but not transmitted.

The **tcsendbreak**() function transmits a continuous stream of zero-valued bits for four-tenths of a second to the terminal referenced by *fd*. The *len* argument is ignored in this implementation.

RETURN VALUES

Upon successful completion, all of these functions return a value of zero.

ERRORS

If any error occurs, a value of -1 is returned and the global variable *errno* is set to indicate the error, as follows:

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

[EINVAL] The *action* argument is not a proper value.

[ENOTTY] The file associated with *fd* is not a terminal.

[EINTR] A signal interrupted the **tcdrain**() function.

[EWOULDBLOCK] The configured timeout expired before the **tcdrain**() function could write all buffered output.

SEE ALSO

tcsetattr(3), termios(4), tty(4), comcontrol(8)

STANDARDS

The **tcsendbreak**(), **tcflush**() and **tcflow**() functions are expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification.

The **tcdrain**() function is expected to be compliant with IEEE Std 1003.1-1988 ("POSIX.1") when the drain wait value is set to zero with comcontrol(8), or with ioctl(2) *TIOCSDRAINWAIT*, or with sysctl(8) *kern.tty_drainwait*. A non-zero drain wait value can result in **tcdrain**() returning

EWOULDBLOCK without writing all output. The default value for *kern.tty_drainwait* is 300 seconds.

NAME

tcgetpgrp - get foreground process group ID

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t
tcgetpgrp(*int fd*);

DESCRIPTION

The **tcgetpgrp**() function returns the value of the process group ID of the foreground process group associated with the terminal device. If there is no foreground process group, **tcgetpgrp**() returns an invalid process ID.

ERRORS

If an error occurs, **tcgetpgrp**() returns -1 and the global variable *errno* is set to indicate the error, as follows:

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

[ENOTTY] The calling process does not have a controlling terminal or the underlying terminal device represented by *fd* is not the controlling terminal.

SEE ALSO

setpgid(2), setsid(2), tcsetpgrp(3)

STANDARDS

The **tcgetpgrp**() function is expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification.

NAME

tcgetsid - get session ID associated with a controlling terminal

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <termios.h>

pid_t
tcgetsid(*int fd*);

DESCRIPTION

The **tcgetsid**() function returns the process group ID of the session leader for a controlling terminal specified by *fd*.

ERRORS

If an error occurs, **tcgetsid**() returns -1 and the global variable *errno* is set to indicate the error, as follows:

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

[ENOTTY] The calling process does not have a controlling terminal or the underlying terminal device represented by *fd* is not the controlling terminal.

SEE ALSO

getsid(2), setsid(2), tcgetpgrp(3), tcsetsid(3)

STANDARDS

The **tcgetsid**() function conforms to X/Open Portability Guide Issue 4, Version 2 ("XPG4.2").

NAME

tcsetpgrp - set foreground process group ID

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

int
tcsetpgrp(*int fd*, *pid_t pgrp_id*);

DESCRIPTION

If the process has a controlling terminal, the **tcsetpgrp**() function sets the foreground process group ID associated with the terminal device to *pgrp_id*. The terminal device associated with *fd* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must be the same as the process group ID of a process in the same session as the calling process.

RETURN VALUES

The **tcsetpgrp**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **tcsetpgrp**() function will fail if:

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

[EINVAL] An invalid value of *pgrp_id* was specified.

[ENOTTY] The calling process does not have a controlling terminal, or the file represented by *fd* is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

[EPERM] The *pgrp_id* argument does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

setpgid(2), setsid(2), tcgetpgrp(3)

STANDARDS

The **tcsetpgrp**() function is expected to be compliant with the IEEE Std 1003.1-1988 ("POSIX.1") specification.

NAME

tcsetsid - set session ID associated with a controlling terminal

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <termios.h>

int
tcsetsid(*int fd*, *pid_t pid*);

DESCRIPTION

The **tcsetsid**() function sets associates a session identified by *pid* with a controlling terminal specified by *fd*.

This implementation only allows the controlling terminal to be changed by the session leader itself. This implies that *pid* always has to be equal to the process ID.

It is unsupported to associate with a terminal that already has an associated session. Conversely, it is also unsupported to associate to a terminal when the session is already associated with a different terminal.

ERRORS

If an error occurs, **tcsetsid**() returns -1 and the global variable *errno* is set to indicate the error, as follows:

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

[ENOTTY] The file descriptor represented by *fd* is not a terminal.

[EINVAL] The *pid* argument is not equal to the session ID of the calling process.

[EPERM] The calling process is not a session leader.

[EPERM] The session already has an associated terminal or the terminal already has an associated session.

SEE ALSO

getsid(2), setsid(2), tcgetpgrp(3), tcgetsid(3)

HISTORY

A **tcsetsid**() function first appeared in QNX. It does not comply to any standard.

NAME

tempnam, **tmpfile**, **tmpnam** - temporary file routines

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

*FILE **
tmpfile(*void*);

*char **
tmpnam(*char *str*);

*char **
tempnam(*const char *tmpdir*, *const char *prefix*);

DESCRIPTION

The **tmpfile**() function returns a pointer to a stream associated with a file descriptor returned by the routine mkstemp(3). The created file is unlinked before **tmpfile**() returns, causing the file to be automatically deleted when the last reference to it is closed. The file is opened with the access value 'w+'. The file is created in the directory determined by the environment variable TMPDIR if set. The default location if TMPDIR is not set is */tmp*.

The **tmpnam**() function returns a pointer to a file name, in the P_tmpdir directory, which did not reference an existing file at some indeterminate point in the past. P_tmpdir is defined in the include file <*stdio.h*>. If the argument *str* is non-NULL, the file name is copied to the buffer it references. Otherwise, the file name is copied to a static buffer. In either case, **tmpnam**() returns a pointer to the file name.

The buffer referenced by *str* is expected to be at least L_tmpnam bytes in length. L_tmpnam is defined in the include file <*stdio.h*>.

The **tempnam**() function is similar to **tmpnam**(), but provides the ability to specify the directory which will contain the temporary file and the file name prefix.

The environment variable TMPDIR (if set), the argument *tmpdir* (if non-NULL), the directory P_tmpdir, and the directory */tmp* are tried, in the listed order, as directories in which to store the temporary file.

The argument *prefix*, if non-NULL, is used to specify a file name prefix, which will be the first part of the created file name. The **tempnam**() function allocates memory in which to store the file name; the returned pointer may be used as a subsequent argument to free(3).

RETURN VALUES

The **tmpfile**() function returns a pointer to an open file stream on success, and a NULL pointer on error.

The **tmpnam**() and **tempfile**() functions return a pointer to a file name on success, and a NULL pointer on error.

ENVIRONMENT

TMPDIR

[**tempnam**() only] If set, the directory in which the temporary file is stored. TMPDIR is ignored for processes for which issetugid(2) is true.

COMPATIBILITY

These interfaces are provided from System V and ANSI compatibility only.

Most historic implementations of these functions provide only a limited number of possible temporary file names (usually 26) before file names will start being recycled. System V implementations of these functions (and of mktemp(3)) use the access(2) system call to determine whether or not the temporary file may be created. This has obvious ramifications for setuid or setgid programs, complicating the portable use of these interfaces in such programs.

The **tmpfile**() interface should not be used in software expected to be used on other systems if there is any possibility that the user does not wish the temporary file to be publicly readable and writable.

ERRORS

The **tmpfile**() function may fail and set the global variable *errno* for any of the errors specified for the library functions fdopen(3) or mkstemp(3).

The **tmpnam**() function may fail and set *errno* for any of the errors specified for the library function mktemp(3).

The **tempnam**() function may fail and set *errno* for any of the errors specified for the library functions malloc(3) or mktemp(3).

SEE ALSO

mkstemp(3), mktemp(3)

STANDARDS

The **tmpfile**() and **tmpnam**() functions conform to ISO/IEC 9899:1990 ("ISO C90").

SECURITY CONSIDERATIONS

The **tmpnam**() and **tempnam**() functions are susceptible to a race condition occurring between the selection of the file name and the creation of the file, which allows malicious users to potentially overwrite arbitrary files in the system, depending on the level of privilege of the running program. Additionally, there is no means by which file permissions may be specified. It is strongly suggested that mkstemp(3) be used in place of these functions.

NAME

thr_exit - terminate current thread

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/thr.h>

void
thr_exit(*long *state*);

DESCRIPTION

This function is intended for implementing threading. Normal applications should call pthread_exit(3) instead.

The **thr_exit**() system call terminates the current kernel-scheduled thread.

If the *state* argument is not NULL, the location pointed to by the argument is updated with an arbitrary non-zero value, and an _umtx_op(2) UMTX_OP_WAKE operation is consequently performed on the location.

Attempts to terminate the last thread in the process are silently ignored. Use _exit(2) syscall to terminate the process.

RETURN VALUES

The function does not return a value. A return from the function indicates that the calling thread was the last one in the process.

SEE ALSO

_exit(2), thr_kill(2), thr_kill2(2), thr_new(2), thr_self(2), thr_set_name(2), _umtx_op(2), pthread_exit(3)

STANDARDS

The **thr_exit**() system call is non-standard and is used by 1:1 Threading Library (libthr, -lthr) to implement IEEE Std 1003.1-2001 ("POSIX.1") pthread(3) functionality.

NAME

thr_kill - send signal to thread

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/thr.h>

int
thr_kill(*long id*, *int sig*);

int
thr_kill2(*pid_t pid*, *long id*, *int sig*);

DESCRIPTION

The **thr_kill**() and **thr_kill2**() system calls allow sending a signal, specified by the *sig* argument, to some threads in a process. For the **thr_kill**() function, signalled threads are always limited to the current process. For the **thr_kill2**() function, the *pid* argument specifies the process with threads to be signalled.

The *id* argument specifies which threads get the signal. If *id* is equal to -1, all threads in the specified process are signalled. Otherwise, only the thread with the thread identifier equal to the argument is signalled.

The *sig* argument defines the delivered signal. It must be a valid signal number or zero. In the latter case no signal is actually sent, and the call is used to verify the liveness of the thread.

The signal is delivered with siginfo si_code set to SI_LWP.

RETURN VALUES

If successful, **thr_kill**() and **thr_kill2**() will return zero, otherwise -1 is returned, and *errno* is set to indicate the error.

ERRORS

The **thr_kill**() and **thr_kill2**() operations return the following errors:

[EINVAL] The *sig* argument is not zero and does not specify valid signal.

[ESRCH] The specified process or thread was not found.

Additionally, the **thr_kill2**() may return the following errors:

[EPERM] The current process does not have sufficient privilege to check existence or send a signal to the specified process.

SEE ALSO

kill(2), thr_exit(2), thr_new(2), thr_self(2), thr_set_name(2), _umtx_op(2), pthread_kill(3), signal(3)

STANDARDS

The **thr_kill**() and **thr_kill2**() system calls are non-standard and are used by the 1:1 Threading Library (libthr, -lthr) to implement IEEE Std 1003.1-2001 ("POSIX.1") pthread(3) functionality.

NAME

thr_new - create new thread of execution

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/thr.h>

int
thr_new(*struct thr_param *param*, *int param_size*);

DESCRIPTION

This function is intended for implementing threading. Normal applications should call pthread_create(3) instead.

The **thr_new**() system call creates a new kernel-scheduled thread of execution in the context of the current process. The newly created thread shares all attributes of the process with the existing kernel-scheduled threads in the process, but has private processor execution state. The machine context for the new thread is copied from the creating thread's context, including coprocessor state. FPU state and specific machine registers are excluded from the copy. These are set according to ABI requirements and syscall parameters. The FPU state for the new thread is reinitialized to clean.

The *param* structure supplies parameters affecting the thread creation. The structure is defined in the <*sys/thr.h*> header as follows

```
struct thr_param {
    void        (*start_func)(void *);
    void        *arg;
    char        *stack_base;
    size_t      stack_size;
    char        *tls_base;
    size_t      tls_size;
    long        *child_tid;
    long        *parent_tid;
    int         flags;
    struct rtprio *rtp;
};
```

and contains the following fields:

start_func Pointer to the thread entry function. The kernel arranges for the new thread to start executing the function upon the first return to userspace.

arg Opaque argument supplied to the entry function.

stack_base Stack base address. The stack must be allocated by the caller. On some architectures, the ABI might require that the system put information on the stack to ensure the execution environment for *start_func*.

stack_size Stack size.

tls_base TLS base address. The value of TLS base is loaded into the ABI-defined machine register in the new thread context.

tls_size TLS size.

child_tid Address to store the new thread identifier, for the child's use.

parent_tid Address to store the new thread identifier, for the parent's use.

Both *child_tid* and *parent_tid* are provided, with the intent that *child_tid* is used by the new thread to get its thread identifier without issuing the thr_self(2) syscall, while *parent_tid* is used by the thread creator. The latter is separate from *child_tid* because the new thread might exit and free its thread data before the parent has a chance to execute far enough to access it.

flags Thread creation flags. The *flags* member may specify the following flags:

THR_SUSPENDED Create the new thread in the suspended state. The flag is not currently implemented.

THR_SYSTEM_SCOPE Create the system scope thread. The flag is not currently implemented.

rtp Real-time scheduling priority for the new thread. May be NULL to inherit the priority from the creating thread.

The *param_size* argument should be set to the size of the *param* structure.

After the first successful creation of an additional thread, the process is marked by the kernel as multi-

threaded. In particular, the P_HADTHREADS flag is set in the process' p_flag (visible in the ps(1) output), and several operations are executed in multi-threaded mode. For instance, the execve(2) system call terminates all threads but the calling one on successful execution.

RETURN VALUES

If successful, **thr_new**() will return zero, otherwise -1 is returned, and *errno* is set to indicate the error.

ERRORS

The **thr_new**() operation returns the following errors:

[EFAULT] The memory pointed to by the *param* argument is not valid.

[EFAULT] The memory pointed to by the *param* structure *child_tid*, *parent_tid* or *rtp* arguments is not valid.

[EFAULT] The specified stack base is invalid, or the kernel was unable to put required initial data on the stack.

[EINVAL] The *param_size* argument specifies a negative value, or the value is greater than the largest *struct param* size the kernel can interpret.

[EINVAL] The *rtp* member is not NULL and specifies invalid scheduling parameters.

[EINVAL] The specified TLS base is invalid.

[EPERM] The caller does not have permission to set the scheduling parameters or scheduling policy.

[EPROCLIM] Creation of the new thread would exceed the RACCT_NTHR limit, see racct(2).

[EPROCLIM] Creation of the new thread would exceed the kern.threads.max_threads_per_proc sysctl(2) limit.

[ENOMEM] There was not enough kernel memory to allocate the new thread structures.

SEE ALSO

ps(1), execve(2), racct(2), thr_exit(2), thr_kill(2), thr_kill2(2), thr_self(2), thr_set_name(2), _umtx_op(2), pthread_create(3)

STANDARDS

The **thr_new**() system call is non-standard and is used by the 1:1 Threading Library (libthr, -lthr) to implement IEEE Std 1003.1-2001 ("POSIX.1") pthread(3) functionality.

NAME

thr_self - return thread identifier for the calling thread

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/thr.h>

int
thr_self(*long *id*);

DESCRIPTION

The **thr_self**() system call stores the system-wide thread identifier for the current kernel-scheduled thread in the variable pointed by the argument *id*.

The thread identifier is an integer in the range from PID_MAX + 2 (10002) to INT_MAX. The thread identifier is guaranteed to be unique at any given time, for each running thread in the system. After the thread exits, the identifier may be reused.

RETURN VALUES

If successful, **thr_self**() will return zero, otherwise -1 is returned, and *errno* is set to indicate the error.

ERRORS

The **thr_self**() operation may return the following errors:

[EFAULT] The memory pointed to by the *id* argument is not valid.

SEE ALSO

thr_exit(2), thr_kill(2), thr_kill2(2), thr_new(2), thr_set_name(2), _umtx_op(2), pthread_getthreadid_np(3), pthread_self(3)

STANDARDS

The **thr_self**() system call is non-standard and is used by 1:1 Threading Library (libthr, -lthr) to implement IEEE Std 1003.1-2001 ("POSIX.1") pthread(3) functionality.

NAME

thr_set_name - set user-visible thread name

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/thr.h>

int
thr_set_name(*long id*, *const char *name*);

DESCRIPTION

The **thr_set_name**() system call sets the user-visible name for the thread with the identifier *id* in the current process to the NUL-terminated string *name*. The name will be silently truncated to fit into a buffer of MAXCOMLEN + 1 bytes. The thread name can be seen in the output of the ps(1) and top(1) commands, in the kernel debuggers and kernel tracing facility outputs, and in userland debuggers and program core files, as notes.

RETURN VALUES

If successful, **thr_set_name**() returns zero; otherwise, -1 is returned, and *errno* is set to indicate the error.

ERRORS

The **thr_set_name**() system call may return the following errors:

[EFAULT] The memory pointed to by the *name* argument is not valid.

[ESRCH] The thread with the identifier *id* does not exist in the current process.

SEE ALSO

ps(1), thr_exit(2), thr_kill(2), thr_kill2(2), thr_new(2), thr_self(2), _umtx_op(2), pthread_set_name_np(3), ddb(4), ktr(9)

STANDARDS

The **thr_set_name**() system call is non-standard and is used by the 1:1 Threading Library (libthr, -lthr).

NAME

thr_suspend - suspend the calling thread

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/thr.h>

int
thr_suspend(*struct timespec *timeout*);

DESCRIPTION

This function is intended for implementing threading. Normal applications should use pthread_cond_timedwait(3) together with pthread_cond_broadcast(3) for typical safe suspension with cooperation of the thread being suspended, or pthread_suspend_np(3) and pthread_resume_np(3) in some specific situations, instead.

The **thr_suspend**() system call puts the calling thread in a suspended state, where it is not eligible for CPU time. This state is exited by another thread calling thr_wake(2), when the time interval specified by *timeout* has elapsed, or by the delivery of a signal to the suspended thread.

If the *timeout* argument is NULL, the suspended state can be only terminated by explicit **thr_wake**() or signal.

If a wake from thr_wake(2) was delivered before the **thr_suspend** call, the thread is not put into a suspended state. Instead, the call returns immediately without an error.

If a thread previously called thr_wake(2) with its own thread identifier, which resulted in setting the internal kernel flag to immediately abort interruptible sleeps with an EINTR error (see thr_wake(2)), the flag is cleared. As with thr_wake(2) called from another thread, the next **thr_suspend** call does not result in suspension.

RETURN VALUES

The **thr_suspend**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **thr_suspend**() operation returns the following errors:

[EFAULT] The memory pointed to by the *timeout* argument is not valid.

[ETIMEDOUT] The specified timeout expired.

[ETIMEDOUT] The *timeout* argument specified a zero time interval.

[EINTR] The sleep was interrupted by a signal.

SEE ALSO

ps(1), thr_wake(2), pthread_resume_np(3), pthread_suspend_np(3)

STANDARDS

The **thr_suspend**() system call is non-standard.

NAME

thr_wake - wake up the suspended thread

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/thr.h>

int
thr_wake(*long id*);

DESCRIPTION

This function is intended for implementing threading. Normal applications should use pthread_cond_timedwait(3) together with pthread_cond_broadcast(3) for typical safe suspension with cooperation of the thread being suspended, or pthread_suspend_np(3) and pthread_resume_np(3) in some specific situations, instead.

Passing the thread identifier of the calling thread (see thr_self(2)) to **thr_wake**() sets a thread's flag to cause the next signal-interruptible sleep of that thread in the kernel to fail immediately with the EINTR error. The flag is cleared by an interruptible sleep attempt or by a call to thr_suspend(2.) This is used by the system threading library to implement cancellation.

If *id* is not equal to the current thread identifier, the specified thread is woken up if suspended by the thr_suspend system call. If the thread is not suspended at the time of the **thr_wake** call, the wake is remembered and the next attempt of the thread to suspend itself with the thr_suspend(2) results in immediate return with success. Only one wake is remembered.

RETURN VALUES

The **thr_wake**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **thr_wake**() operation returns these errors:

[ESRCH]	The specified thread was not found or does not belong to the process of the calling thread.

SEE ALSO

ps(1), thr_self(2) thr_suspend(2), pthread_cancel(3), pthread_resume_np(3), pthread_suspend_np(3)

STANDARDS

The **thr_suspend**() system call is non-standard and is used by 1:1 Threading Library (libthr, -lthr) to implement IEEE Std 1003.1-2001 ("POSIX.1") pthread(3) functionality.

NAME

time - get time of day

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

time_t
time(*time_t *tloc*);

DESCRIPTION

The **time**() function returns the value of time in seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time. If an error occurs, **time**() returns the value (*time_t*)-1.

The return value is also stored in **tloc*, provided that *tloc* is non-null.

ERRORS

The **time**() function may fail for any of the reasons described in gettimeofday(2).

SEE ALSO

clock_gettime(2), gettimeofday(2), ctime(3)

STANDARDS

The **time** function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

A **time**() function appeared in Version 6 AT&T UNIX.

BUGS

Neither ISO/IEC 9899:1999 ("ISO C99") nor IEEE Std 1003.1-2001 ("POSIX.1") requires **time**() to set *errno* on failure; thus, it is impossible for an application to distinguish the valid time value -1 (representing the last UTC second of 1969) from the error return value.

Systems conforming to earlier versions of the C and POSIX standards (including older versions of FreeBSD) did not set **tloc* in the error case.

NAME

times - process times

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/times.h>

clock_t
times(*struct tms *tp*);

DESCRIPTION

This interface is obsoleted by getrusage(2) and gettimeofday(2).

The **times**() function returns the value of time in CLK_TCK's of a second since the system startup time. The current value of CLK_TCK, the frequency of the statistics clock in ticks per second, may be obtained through the sysconf(3) interface.

It also fills in the structure pointed to by *tp* with time-accounting information.

The *tms* structure is defined as follows:

```
struct tms {
        clock_t tms_utime;
        clock_t tms_stime;
        clock_t tms_cutime;
        clock_t tms_cstime;
};
```

The elements of this structure are defined as follows:

tms_utime The CPU time charged for the execution of user instructions.

tms_stime The CPU time charged for execution by the system on behalf of the process.

tms_cutime The sum of the *tms_utime*s and *tms_cutime*s of the child processes.

tms_cstime The sum of the *tms_stime*s and *tms_cstime*s of the child processes.

All times are in CLK_TCK's of a second.

The times of a terminated child process are included in the *tms_cutime* and *tms_cstime* elements of the parent when one of the wait(2) functions returns the process ID of the terminated child to the parent. If an error occurs, **times**() returns the value ((*clock_t*)-1), and sets *errno* to indicate the error.

ERRORS

The **times**() function may fail and set the global variable *errno* for any of the errors specified for the library routines getrusage(2) and gettimeofday(2).

SEE ALSO

time(1), getrusage(2), gettimeofday(2), wait(2), sysconf(3), clocks(7)

STANDARDS

The **times**() function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

NAME

timespec_get - get current calendar time

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

int
timespec_get(*struct timespec *ts*, *int base*);

DESCRIPTION

The **timespec_get** function sets the interval pointed to by *ts* to hold the current calendar time based on the specified time base in *base*.

The base TIME_UTC returns the time since the epoch. This time is expressed in seconds and nanoseconds since midnight (0 hour), January 1, 1970. In FreeBSD, this corresponds to CLOCK_REALTIME.

RETURN VALUES

The **timespec_get** function returns the passed value of *base* if successful, otherwise 0 on failure.

SEE ALSO

gettimeofday(2), clock_gettime(2), time(3)

STANDARDS

The **timespec_get** function with a *base* of TIME_UTC conforms to ISO/IEC 9899:2011 ("ISO C11").

HISTORY

This interface first appeared in FreeBSD 12.

AUTHORS

Kamil Rytarowski <*kamil@NetBSD.org*>
Warner Losh <*imp@FreeBSD.org*>

NAME

timezone - return the timezone abbreviation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

*char **
timezone(*int zone*, *int dst*);

DESCRIPTION

This interface is for compatibility only; it is impossible to reliably map timezone's arguments to a time zone abbreviation. See ctime(3).

The **timezone**() function returns a pointer to a time zone abbreviation for the specified *zone* and *dst* values. The *zone* argument is the number of minutes west of GMT and *dst* is non-zero if daylight savings time is in effect.

SEE ALSO

ctime(3)

HISTORY

A **timezone**() function appeared in Version 7 AT&T UNIX.

NAME

towctrans, **wctrans** - wide character mapping functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wctype.h>

wint_t
towctrans(*wint_t wc*, *wctrans_t desc*);

wctrans_t
wctrans(*const char *charclass*);

DESCRIPTION

The **wctrans**() function returns a value of type *wctrans_t* which represents the requested wide character mapping operation and may be used as the second argument for calls to **towctrans**().

The following character mapping names are recognised:

tolower toupper

The **towctrans**() function transliterates the wide character *wc* according to the mapping described by *desc*.

RETURN VALUES

The **towctrans**() function returns the transliterated character if successful, otherwise it returns the character unchanged and sets *errno*.

The **wctrans**() function returns non-zero if successful, otherwise it returns zero and sets *errno*.

EXAMPLES

Reimplement **towupper**() in terms of **towctrans**() and **wctrans**():

```
wint_t
mytowupper(wint_t wc)
{
        return (towctrans(wc, wctrans("toupper")));
}
```

ERRORS

The **towctrans**() function will fail if:

[EINVAL] The supplied *desc* argument is invalid.

The **wctrans**() function will fail if:

[EINVAL] The requested mapping name is invalid.

SEE ALSO

tolower(3), toupper(3), wctype(3)

STANDARDS

The **towctrans**() and **wctrans**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **towctrans**() and **wctrans**() functions first appeared in FreeBSD 5.0.

NAME

towlower - upper case to lower case letter conversion (wide character version)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wctype.h>

wint_t
towlower(*wint_t wc*);

DESCRIPTION

The **towlower**() function converts an upper-case letter to the corresponding lower-case letter.

RETURN VALUES

If the argument is an upper-case letter, the **towlower**() function returns the corresponding lower-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

iswlower(3), tolower(3), towupper(3), wctrans(3)

STANDARDS

The **towlower**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

towupper - lower case to upper case letter conversion (wide character version)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wctype.h>

wint_t
towupper(*wint_t wc*);

DESCRIPTION

The **towupper**() function converts a lower-case letter to the corresponding upper-case letter.

RETURN VALUES

If the argument is a lower-case letter, the **towupper**() function returns the corresponding upper-case letter if there is one; otherwise the argument is returned unchanged.

SEE ALSO

iswupper(3), toupper(3), towlower(3), wctrans(3)

STANDARDS

The **towupper**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

tzset, **tzsetwall** - initialize time conversion information

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <time.h>

void
tzset(*void*);

void
tzsetwall(*void*);

DESCRIPTION

The **tzset**() function initializes time conversion information used by the library routine localtime(3). The environment variable TZ specifies how this is done.

If TZ does not appear in the environment, the best available approximation to local wall clock time, as specified by the tzfile(5)-format file */etc/localtime* is used.

If TZ appears in the environment but its value is a null string, Coordinated Universal Time (UTC) is used (without leap second correction).

If TZ appears in the environment and its value begins with a colon (':'), the rest of its value is used as a pathname of a tzfile(5)-format file from which to read the time conversion information. If the first character of the pathname is a slash ('/') it is used as an absolute pathname; otherwise, it is used as a pathname relative to the system time conversion information directory.

If its value does not begin with a colon, it is first used as the pathname of a file (as described above) from which to read the time conversion information. If that file cannot be read, the value is then interpreted as a direct specification (the format is described below) of the time conversion information.

If the TZ environment variable does not specify a tzfile(5)-format file and cannot be interpreted as a direct specification, UTC is used.

The **tzsetwall**() function sets things up so that localtime(3) returns the best available approximation of local wall clock time.

SPECIFICATION FORMAT

When TZ is used directly as a specification of the time conversion information, it must have the following syntax (spaces inserted for clarity):

std offset [*dst* [*offset*] [, *rule*]]

Where:

std and *dst* Three or more bytes that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper and lowercase letters are explicitly allowed. Any characters except a leading colon (':'), digits, comma (','), minus ('-'), plus ('+'), and ASCII NUL are allowed.

offset Indicates the value one must add to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

hh[:*mm*[:ss]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between zero and 24, and the minutes (and seconds) -- if present -- between zero and 59. If preceded by a ('-') the time zone shall be east of the Prime Meridian; otherwise it shall be west (which may be indicated by an optional preceding ('+')).

rule Indicates when to change to and back from summer time. The *rule* has the form:

date/time,date/time

where the first *date* describes when the change from standard to summer time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

J *n* The Julian day *n* (1 <= *n* <= 365). Leap days are not counted; that is, in all years -- including leap years -- February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n The zero-based Julian day (0 <= *n* <= 365) . Leap days are counted, and it is possible to refer to February 29.

M *m.n.d* The *d*'th day (0 <= *d* <= 6) of week *n* of month *m* of the year (1 <= *n* <= 5), (1 <= *m* <= 12), where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign ('-') or ('+') is allowed. The default, if *time* is not given, is **02:00:00**.

If no *rule* is present in the TZ specification, the rules specified by the tzfile(5)-format file *posixrules* in the system time conversion information directory are used, with the standard and summer time offsets from UTC replaced by those specified by the *offset* values in TZ.

For compatibility with System V Release 3.1, a semicolon (';') may be used to separate the *rule* from the rest of the specification.

FILES

/etc/localtime	local time zone file
/usr/share/zoneinfo	time zone directory
/usr/share/zoneinfo/posixrules	rules for POSIX-style TZ's
/usr/share/zoneinfo/Etc/GMT	for UTC leap seconds

If the file */usr/share/zoneinfo/UTC* does not exist, UTC leap seconds are loaded from */usr/share/zoneinfo/posixrules*.

SEE ALSO

date(1), gettimeofday(2), ctime(3), getenv(3), time(3), tzfile(5)

HISTORY

The **tzset**() and **tzsetwall**() functions first appeared in 4.4BSD.

NAME

ualarm - schedule signal after specified time

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

useconds_t
ualarm(*useconds_t microseconds*, *useconds_t interval*);

DESCRIPTION

This is a simplified interface to setitimer(2).

The **ualarm**() function waits a count of *microseconds* before asserting the terminating signal SIGALRM. System activity or time used in processing the call may cause a slight delay.

If the *interval* argument is non-zero, the SIGALRM signal will be sent to the process every *interval* microseconds after the timer expires (e.g. after *microseconds* number of microseconds have passed).

Due to setitimer(2) restriction the maximum number of *microseconds* and *interval* is limited to 100000000000000 (in case this value fits in the unsigned integer).

RETURN VALUES

When the signal has successfully been caught, **ualarm**() returns the amount of time left on the clock.

NOTES

A microsecond is 0.000001 seconds.

SEE ALSO

getitimer(2), setitimer(2), sigaction(2), sigsuspend(2), alarm(3), signal(3), sleep(3), usleep(3)

HISTORY

The **ualarm**() function appeared in 4.3BSD.

NAME

ucontext - user thread context

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <ucontext.h>

DESCRIPTION

The *ucontext_t* type is a structure type suitable for holding the context for a user thread of execution. A thread's context includes its stack, saved registers, and list of blocked signals.

The *ucontext_t* structure contains at least these fields:

*ucontext_t *uc_link*	context to assume when this one returns
sigset_t uc_sigmask	signals being blocked
stack_t uc_stack	stack area
mcontext_t uc_mcontext	saved registers

The *uc_link* field points to the context to resume when this context's entry point function returns. If *uc_link* is equal to NULL, then the process exits when this context returns.

The *uc_mcontext* field is machine-dependent and should be treated as opaque by portable applications.

The following functions are defined to manipulate *ucontext_t* structures:

int **getcontext**(*ucontext_t *);
*ucontext_t * * **getcontextx**(*void*);
int **setcontext**(*const ucontext_t *);
void **makecontext**(*ucontext_t *, void (*)(void), int, ...*);
int **swapcontext**(*ucontext_t *, const ucontext_t *);

SEE ALSO

sigaltstack(2), getcontext(3), getcontextx(3), makecontext(3)

NAME

ulimit - get and set process limits

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <ulimit.h>

long
ulimit(*int cmd*, ...);

DESCRIPTION

The **ulimit**() function will get and set process limits. Currently this is limited to the maximum file size. The *cmd* argument is one of the following:

UL_GETFSIZE will return the maximum file size in units of 512 blocks of the current process.

UL_SETFSIZE will attempt to set the maximum file size of the current process and its children with the second argument expressed as a long.

RETURN VALUES

Upon successful completion, **ulimit**() returns the value requested; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **ulimit**() function will fail if:

[EINVAL] The command specified was invalid.

[EPERM] The limit specified to **ulimit**() would have raised the maximum limit value, and the caller is not the super-user.

SEE ALSO

getrlimit(2)

STANDARDS

The **ulimit**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

The **ulimit**() function first appeared in FreeBSD 5.0.

BUGS

The **ulimit**() function provides limited precision for setting and retrieving process limits. If there is a need for greater precision than the type *long* provides, the getrlimit(2) and setrlimit(2) functions should be considered.

NAME

umask - set file creation mode mask

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/stat.h>

mode_t
umask(*mode_t numask*);

DESCRIPTION

The **umask**() routine sets the process's file mode creation mask to *numask* and returns the previous value of the mask. The 9 low-order access permission bits of *numask* are used by system calls, including open(2), mkdir(2), and mkfifo(2), to turn off corresponding bits requested in file mode. (See chmod(2)). This clearing allows each user to restrict the default access to his files.

The default mask value is S_IWGRP|S_IWOTH (022, write access for the owner only). Child processes inherit the mask of the calling process.

RETURN VALUES

The previous value of the file mode mask is returned by the call.

ERRORS

The **umask**() system call is always successful.

SEE ALSO

chmod(2), mkfifo(2), mknod(2), open(2)

STANDARDS

The **umask**() system call is expected to conform to IEEE Std 1003.1-1990 ("POSIX.1").

HISTORY

The **umask**() function appeared in Version 7 AT&T UNIX.

NAME

uname - get system identification

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/utsname.h>

int
uname(*struct utsname *name*);

DESCRIPTION

The **uname**() function stores NUL-terminated strings of information identifying the current system into the structure referenced by *name*.

The *utsname* structure is defined in the <*sys/utsname.h*> header file, and contains the following members:

sysname	Name of the operating system implementation.
nodename	Network name of this machine.
release	Release level of the operating system.
version	Version level of the operating system.
machine	Machine hardware platform.

RETURN VALUES

The **uname**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ENVIRONMENT

UNAME_s	If the environment variable UNAME_s is set, it will override the *sysname* member.
UNAME_r	If the environment variable UNAME_r is set, it will override the *release* member.
UNAME_v	If the environment variable UNAME_v is set, it will override the *version* member.

UNAME_m If the environment variable UNAME_m is set, it will override the *machine* member.

ERRORS

The **uname**() function may fail and set *errno* for any of the errors specified for the library functions sysctl(3).

SEE ALSO

uname(1), sysctl(3)

STANDARDS

The **uname**() function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

HISTORY

The **uname**() function first appeared in 4.4BSD.

NAME

undelete - attempt to recover a deleted file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
undelete(*const char *path*);

DESCRIPTION

The **undelete**() system call attempts to recover the deleted file named by *path*. Currently, this works only when the named object is a whiteout in a union file system. The system call removes the whiteout causing any objects in a lower layer of the union stack to become visible once more.

Eventually, the **undelete**() functionality may be expanded to other file systems able to recover deleted files such as the log-structured file system.

RETURN VALUES

The **undelete**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **undelete**() succeeds unless:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[EEXIST] The path does not reference a whiteout.

[ENOENT] The named whiteout does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write permission is denied on the directory containing the name to be undeleted.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The directory containing the name is marked sticky, and the containing directory is not owned by the effective user ID.

[EINVAL] The last component of the path is '..'.

[EIO] An I/O error occurred while updating the directory entry.

[EROFS] The name resides on a read-only file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

SEE ALSO

unlink(2), mount_unionfs(8)

HISTORY

The **undelete**() system call first appeared in 4.4BSD-Lite.

NAME

ungetc - un-get character from input stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
ungetc(*int c*, *FILE *stream*);

DESCRIPTION

The **ungetc**() function pushes the character *c* (converted to an unsigned char) back onto the input stream pointed to by *stream*. The pushed-back characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions (fseek(3), fsetpos(3), or rewind(3)) will discard the pushed back characters.

One character of push-back is guaranteed, but as long as there is sufficient memory, an effectively infinite amount of pushback is allowed.

If a character is successfully pushed-back, the end-of-file indicator for the stream is cleared. The file-position indicator is decremented by each successful call to **ungetc**(); if its value was 0 before a call, its value is unspecified after the call.

RETURN VALUES

The **ungetc**() function returns the character pushed-back after the conversion, or EOF if the operation fails. If the value of the argument *c* character equals EOF, the operation will fail and the stream will remain unchanged.

SEE ALSO

fseek(3), getc(3), setvbuf(3), ungetwc(3)

STANDARDS

The **ungetc**() function conforms to ISO/IEC 9899:1990 ("ISO C90").

NAME

ungetwc - un-get wide character from input stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

wint_t
ungetwc(*wint_t wc*, *FILE *stream*);

DESCRIPTION

The **ungetwc**() function pushes the wide character *wc* (converted to an *wchar_t*) back onto the input stream pointed to by *stream*. The pushed-backed wide characters will be returned by subsequent reads on the stream (in reverse order). A successful intervening call, using the same stream, to one of the file positioning functions fseek(3), fsetpos(3), or rewind(3) will discard the pushed back wide characters.

One wide character of push-back is guaranteed, but as long as there is sufficient memory, an effectively infinite amount of pushback is allowed.

If a character is successfully pushed-back, the end-of-file indicator for the stream is cleared.

RETURN VALUES

The **ungetwc**() function returns the wide character pushed-back after the conversion, or WEOF if the operation fails. If the value of the argument *c* character equals WEOF, the operation will fail and the stream will remain unchanged.

SEE ALSO

fseek(3), getwc(3)

STANDARDS

The **ungetwc**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

unlink, **unlinkat** - remove directory entry

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
unlink(*const char *path*);

int
unlinkat(*int fd*, *const char *path*, *int flag*);

DESCRIPTION

The **unlink**() system call removes the link named by *path* from its directory and decrements the link count of the file which was referenced by the link. If that decrement reduces the link count of the file to zero, and no process has the file open, then all resources associated with the file are reclaimed. If one or more process have the file open when the last link is removed, the link is removed, but the removal of the file is delayed until all references to it have been closed. The *path* argument may not be a directory.

The **unlinkat**() system call is equivalent to **unlink**() or **rmdir**() except in the case where *path* specifies a relative path. In this case the directory entry to be removed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory.

The values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>:

AT_REMOVEDIR
Remove the directory entry specified by *fd* and *path* as a directory, not a normal file.

If **unlinkat**() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior is identical to a call to *unlink* or *rmdir* respectively, depending on whether or not the AT_REMOVEDIR bit is set in flag.

RETURN VALUES

The **unlink**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **unlink**() succeeds unless:

[ENOTDIR] A component of the path prefix is not a directory.

[EISDIR] The named file is a directory.

[ENAMETOOLONG]
A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write permission is denied on the directory containing the link to be removed.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EPERM] The named file is a directory.

[EPERM] The named file has its immutable, undeletable or append-only flag set, see the chflags(2) manual page for more information.

[EPERM] The parent directory of the named file has its immutable or append-only flag set.

[EPERM] The directory containing the file is marked sticky, and neither the containing directory nor the file to be removed are owned by the effective user ID.

[EIO] An I/O error occurred while deleting the directory entry or deallocating the inode.

[EROFS] The named file resides on a read-only file system.

[EFAULT] The *path* argument points outside the process's allocated address space.

[ENOSPC] On file systems supporting copy-on-write or snapshots, there was not enough free space to record metadata for the delete operation of the file.

In addition to the errors returned by the **unlink**(), the **unlinkat**() may fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[ENOTEMPTY] The *flag* parameter has the AT_REMOVEDIR bit set and the *path* argument names a directory that is not an empty directory, or there are hard links to the directory other than dot or a single entry in dot-dot.

[ENOTDIR] The *flag* parameter has the AT_REMOVEDIR bit set and *path* does not name a directory.

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

SEE ALSO

chflags(2), close(2), link(2), rmdir(2), symlink(7)

STANDARDS

The **unlinkat**() system call follows The Open Group Extended API Set 2 specification.

HISTORY

The **unlink**() function appeared in Version 1 AT&T UNIX. The **unlinkat**() system call appeared in FreeBSD 8.0.

The **unlink**() system call traditionally allows the super-user to unlink directories which can damage the file system integrity. This implementation no longer permits it.

NAME

uselocale - Sets a thread-local locale

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <locale.h>

locale_t
uselocale(*locale_t locale*);

DESCRIPTION

Specifies the locale for this thread to use. Specifying *LC_GLOBAL_LOCALE* disables the per-thread locale, while NULL returns the current locale without setting a new one.

RETURN VALUES

Returns the previous locale, or LC_GLOBAL_LOCALE if this thread has no locale associated with it.

SEE ALSO

duplocale(3), freelocale(3), localeconv(3), newlocale(3), querylocale(3), xlocale(3)

STANDARDS

This function conforms to IEEE Std 1003.1-2008 ("POSIX.1").

NAME

usleep - suspend process execution for an interval measured in microseconds

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

int
usleep(*useconds_t microseconds*);

DESCRIPTION

The **usleep**() function suspends execution of the calling process until either *microseconds* microseconds have elapsed or a signal is delivered to the process and its action is to invoke a signal-catching function or to terminate the process. System activity may lengthen the sleep by an indeterminate amount.

This function is implemented using nanosleep(2) by pausing for *microseconds* microseconds or until a signal occurs. Consequently, in this implementation, sleeping has no effect on the state of process timers, and there is no special handling for SIGALRM.

RETURN VALUES

The **usleep**() function returns the value 0 if successful; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **usleep**() function will fail if:

[EINTR] A signal was delivered to the process and its action was to invoke a signal-catching function.

SEE ALSO

nanosleep(2), sleep(3)

HISTORY

The **usleep**() function appeared in 4.3BSD.

NAME

utime - set file times

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <utime.h>

int
utime(*const char *file*, *const struct utimbuf *timep*);

DESCRIPTION

This interface is obsoleted by utimensat(2) because it is not accurate to fractions of a second.

The **utime**() function sets the access and modification times of the named file from the *actime* and *modtime* fields of the *struct utimbuf* pointed at by *timep*.

If the times are specified (the *timep* argument is non-NULL) the caller must be the owner of the file or be the super-user.

If the times are not specified (the *timep* argument is NULL) the caller must be the owner of the file, have permission to write the file, or be the super-user.

ERRORS

The **utime**() function may fail and set *errno* for any of the errors specified for the library function utimes(2).

SEE ALSO

stat(2), utimensat(2), utimes(2)

STANDARDS

The **utime**() function conforms to IEEE Std 1003.1-1988 ("POSIX.1").

HISTORY

A **utime**() function appeared in Version 7 AT&T UNIX.

NAME

utrace - insert user record in ktrace log

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/param.h>
#include <sys/time.h>
#include <sys/uio.h>
#include <sys/ktrace.h>

int
utrace(*const void *addr*, *size_t len*);

DESCRIPTION

Adds a record to the process trace with information supplied by user. The record contains *len* bytes from memory pointed to by *addr*. This call only has an effect if the calling process is being traced.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL]	Specified data length *len* was bigger than KTR_USER_MAXLEN.
[ENOMEM]	Insufficient memory to honor the request.
[ENOSYS]	Currently running kernel was compiled without ktrace(2) support (**options KTRACE**).

SEE ALSO

kdump(1), ktrace(1), truss(1), ktrace(2), sysdecode_utrace(3)

HISTORY

The **utrace**() system call first appeared in FreeBSD 2.2.

NAME

uuid_compare, **uuid_create**, **uuid_create_nil**, **uuid_equal**, **uuid_from_string**, **uuid_hash**, **uuid_is_nil**, **uuid_to_string** - DCE 1.1 compliant UUID functions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <uuid.h>

int32_t
uuid_compare(*const uuid_t *uuid1*, *const uuid_t *uuid2*, *uint32_t *status*);

void
uuid_create(*uuid_t *uuid*, *uint32_t *status*);

void
uuid_create_nil(*uuid_t *uuid*, *uint32_t *status*);

int32_t
uuid_equal(*const uuid_t *uuid1*, *const uuid_t *uuid2*, *uint32_t *status*);

void
uuid_from_string(*const char *str*, *uuid_t *uuid*, *uint32_t *status*);

uint16_t
uuid_hash(*const uuid_t *uuid*, *uint32_t *status*);

int32_t
uuid_is_nil(*const uuid_t *uuid*, *uint32_t *status*);

void
uuid_to_string(*const uuid_t *uuid*, *char **str*, *uint32_t *status*);

void
uuid_enc_le(*void *buf*, *const uuid_t *uuid*);

void
uuid_dec_le(*const void *buf*, *uuid_t **);

void
uuid_enc_be(*void *buf*, *const uuid_t *uuid*);

void
uuid_dec_be(*const void *buf*, *uuid_t **);

DESCRIPTION

The family of DCE 1.1 compliant UUID functions allow applications to operate on universally unique identifiers, or UUIDs. The **uuid_create**() and **uuid_create_nil**() functions create UUIDs. The **uuid_compare**(), **uuid_equal**() and **uuid_is_nil**() functions can be used to test UUIDs. To convert from the binary representation to the string representation or vice versa, use **uuid_to_string**() or **uuid_from_string**() respectively. A 16-bit hash value can be obtained by calling **uuid_hash**().

The **uuid_to_string**() function set **str* to be a pointer to a buffer sufficiently large to hold the string. This pointer should be passed to free(3) to release the allocated storage when it is no longer needed.

The **uuid_enc_le**() and **uuid_enc_be**() functions encode a binary representation of a UUID into an octet stream in little-endian and big-endian byte-order, respectively. The destination buffer must be pre-allocated by the caller, and must be large enough to hold the 16-octet binary UUID. These routines are not part of the DCE RPC API. They are provided for convenience.

The **uuid_dec_le**() and **uuid_dec_be**() functions decode a UUID from an octet stream in little-endian and big-endian byte-order, respectively. These routines are not part of the DCE RPC API. They are provided for convenience.

RETURN VALUES

The successful or unsuccessful completion of the function is returned in the *status* argument. Possible values are:

uuid_s_ok	The function completed successfully.
uuid_s_bad_version	The UUID does not have a known version.
uuid_s_invalid_string_uuid	The string representation of an UUID is not valid.
uuid_s_no_memory	The function can not allocate memory to store an UUID representation.

SEE ALSO

uuidgen(1), uuidgen(2)

STANDARDS

The UUID functions conform to the DCE 1.1 RPC specification.

BUGS

This manpage can be improved.

NAME

uuidgen - generate universally unique identifiers

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/uuid.h>

int
uuidgen(*struct uuid *store*, *int count*);

DESCRIPTION

The **uuidgen**() system call generates *count* universally unique identifiers (UUIDs) and writes them to the buffer pointed to by *store*. The identifiers are generated according to the syntax and semantics of the DCE version 1 variant of universally unique identifiers. See below for a more in-depth description of the identifiers. When no IEEE 802 address is available for the node field, a random multicast address is generated for each invocation of the system call. According to the algorithm of generating time-based UUIDs, this will also force a new random clock sequence, thereby increasing the likelihood for the identifier to be unique.

When multiple identifiers are to be generated, the **uuidgen**() system call will generate a set of identifiers that is dense in such a way that there is no identifier that is larger than the smallest identifier in the set and smaller than the largest identifier in the set and that is not already in the set.

Universally unique identifiers, also known as globally unique identifiers (GUIDs), have a binary representation of 128-bits. The grouping and meaning of these bits is described by the following structure and its description of the fields that follow it:

```
struct uuid {
        uint32_t  time_low;
        uint16_t  time_mid;
        uint16_t  time_hi_and_version;
        uint8_t           clock_seq_hi_and_reserved;
        uint8_t           clock_seq_low;
        uint8_t           node[_UUID_NODE_LEN];
};
```

time_low The least significant 32 bits of a 60-bit timestamp. This field is stored in the native byte-order.

time_mid The least significant 16 bits of the most significant 28 bits of the 60-bit timestamp. This field is stored in the native byte-order.

time_hi_and_version The most significant 12 bits of the 60-bit timestamp multiplexed with a 4-bit version number. The version number is stored in the most significant 4 bits of the 16-bit field. This field is stored in the native byte-order.

clock_seq_hi_and_reserved The most significant 6 bits of a 14-bit sequence number multiplexed with a 2-bit variant value. Note that the width of the variant value is determined by the variant itself. Identifiers generated by the **uuidgen**() system call have variant value 10b. the variant value is stored in the most significant bits of the field.

clock_seq_low The least significant 8 bits of a 14-bit sequence number.

node The 6-byte IEEE 802 (MAC) address of one of the interfaces of the node. If no such interface exists, a random multi-cast address is used instead.

The binary representation is sensitive to byte ordering. Any multi-byte field is to be stored in the local or native byte-order and identifiers must be converted when transmitted to hosts that do not agree on the byte-order. The specification does not however document what this means in concrete terms and is otherwise beyond the scope of this system call.

RETURN VALUES

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The **uuidgen**() system call can fail with:

[EFAULT] The buffer pointed to by *store* could not be written to for any or all identifiers.

[EINVAL] The *count* argument is less than 1 or larger than the hard upper limit of 2048.

SEE ALSO

uuidgen(1), uuid(3)

STANDARDS

The identifiers are represented and generated in conformance with the DCE 1.1 RPC specification. The **uuidgen**() system call is itself not part of the specification.

HISTORY

The **uuidgen**() system call first appeared in FreeBSD 5.0.

NAME

valloc - aligned memory allocation function

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

*void **
valloc(*size_t size*);

DESCRIPTION

The **valloc**() function is obsoleted by posix_memalign(3), which can be used to request page-aligned allocations.

The **valloc**() function allocates *size* bytes aligned on a page boundary.

RETURN VALUES

The **valloc**() function returns a pointer to the allocated space if successful; otherwise a null pointer is returned.

SEE ALSO

posix_memalign(3)

HISTORY

The **valloc**() function appeared in 3.0BSD.

The **valloc**() function correctly allocated memory that could be deallocated via **free**() starting in FreeBSD 7.0.

NAME

vfork - create a new process without copying the address space

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
vfork(*void*);

DESCRIPTION

Since this function is hard to use correctly from application software, it is recommended to use posix_spawn(3) or fork(2) instead.

The **vfork**() system call can be used to create new processes without fully copying the address space of the old process, which is inefficient in a paged environment. It is useful when the purpose of fork(2) would have been to create a new system context for an execve(2). The **vfork**() system call differs from fork(2) in that the child borrows the parent process's address space and the calling thread's stack until a call to execve(2) or an exit (either by a call to _exit(2) or abnormally). The calling thread is suspended while the child is using its resources. Other threads continue to run.

The **vfork**() system call returns 0 in the child's context and (later) the pid of the child in the parent's context.

Many problems can occur when replacing fork(2) with **vfork**(). For example, it does not work to return while running in the child's context from the procedure that called **vfork**() since the eventual return from **vfork**() would then return to a no longer existent stack frame. Also, changing process state which is partially implemented in user space such as signal handlers with libthr(3) will corrupt the parent's state.

Be careful, also, to call _exit(2) rather than exit(3) if you cannot execve(2), since exit(3) will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. (Even with fork(2) it is wrong to call exit(3) since buffered data would then be flushed twice.)

RETURN VALUES

Same as for fork(2).

SEE ALSO

_exit(2), execve(2), fork(2), rfork(2), sigaction(2), wait(2), exit(3), posix_spawn(3)

HISTORY

The **vfork**() system call appeared in 3BSD.

BUGS

To avoid a possible deadlock situation, processes that are children in the middle of a **vfork**() are never sent SIGTTOU or SIGTTIN signals; rather, output or ioctl(2) calls are allowed and input attempts result in an end-of-file indication.

NAME

wprintf, **fwprintf**, **swprintf**, **vwprintf**, **vfwprintf**, **vswprintf** - formatted wide character output conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>
#include <wchar.h>

int
fwprintf(*FILE * restrict stream*, *const wchar_t * restrict format*, ...);

int
swprintf(*wchar_t * restrict ws*, *size_t n*, *const wchar_t * restrict format*, ...);

int
wprintf(*const wchar_t * restrict format*, ...);

#include <stdarg.h>

int
vfwprintf(*FILE * restrict stream*, *const wchar_t * restrict*, *va_list ap*);

int
vswprintf(*wchar_t * restrict ws*, *size_t n*, *const wchar_t *restrict format*, *va_list ap*);

int
vwprintf(*const wchar_t * restrict format*, *va_list ap*);

DESCRIPTION

The **wprintf**() family of functions produces output according to a *format* as described below. The **wprintf**() and **vwprintf**() functions write output to stdout, the standard output stream; **fwprintf**() and **vfwprintf**() write output to the given output *stream*; **swprintf**() and **vswprintf**() write to the wide character string *ws*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of stdarg(3)) are converted for output.

These functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

The **swprintf**() and **vswprintf**() functions will fail if *n* or more wide characters were requested to be written,

The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. The arguments must correspond properly (after type promotion) with the conversion specifier. After the **%**, the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a **$**, specifying the next argument to access. If this field is not provided, the argument following the last argument accessed will be used. Arguments are numbered starting at **1**. If unaccessed arguments in the format string are interspersed with ones that are accessed the results will be indeterminate.

- Zero or more of the following flags:

  '**#**' The value should be converted to an "alternate form". For **c**, **d**, **i**, **n**, **p**, **s**, and **u** conversions, this option has no effect. For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For **x** and **X** conversions, a non-zero result has the string '0x' (or '0X' for **X** conversions) prepended to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be.

  '**0**' (zero) Zero padding. For all conversions except **n**, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (**d**, **i**, **o**, **u**, **i**, **x**, and **X**), the **0** flag is ignored.

  '**-**' A negative field width flag; the converted value is to be left adjusted on the field boundary. Except for **n** conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A **-** overrides a **0** if both are given.

  ' ' (space) A blank should be left before a positive number produced by a signed conversion (**a**,

A, **d**, **e**, **E**, **f**, **F**, **g**, **G**, or **i**).

‘**+**’ A sign must always be placed before a number produced by a signed conversion. A **+** overrides a space if both are used.

‘’’ Decimal conversions (**d**, **u**, or **i**) or the integral portion of a floating point conversion (**f** or **F**) should be grouped and separated by thousands using the non-monetary separator returned by localeconv(3).

- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.

- An optional precision, in the form of a period **.** followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** conversions.

- An optional length modifier, that specifies the size of the argument. The following length modifiers are valid for the **d**, **i**, **n**, **o**, **u**, **x**, or **X** conversion:

Modifier	**d**, **i**	**o**, **u**, **x**, **X**	**n**
hh	*signed char*	*unsigned char*	*signed char **
h	*short*	*unsigned short*	*short **
l (ell)	*long*	*unsigned long*	*long **
ll (ell ell)	*long long*	*unsigned long long*	*long long **
j	*intmax_t*	*uintmax_t*	*intmax_t **
t	*ptrdiff_t*	(see note)	*ptrdiff_t **
z	(see note)	*size_t*	(see note)
q *(deprecated)*	*quad_t*	*u_quad_t*	*quad_t **

Note: the **t** modifier, when applied to a **o**, **u**, **x**, or **X** conversion, indicates that the argument is of an unsigned type equivalent in size to a *ptrdiff_t*. The **z** modifier, when applied to a **d** or **i** conversion, indicates that the argument is of a signed type equivalent in size to a *size_t*. Similarly, when applied to an **n** conversion, it indicates that the argument is a pointer to a signed type equivalent in size to a *size_t*.

The following length modifier is valid for the **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion:

Modifier	**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**
L	*long double*

The following length modifier is valid for the **c** or **s** conversion:

Modifier	**c**	**s**
l (ell)	*wint_t*	*wchar_t **

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk '*' or an asterisk followed by one or more decimal digits and a '$' instead of a digit string. In this case, an *int* argument supplies the field width or precision. A negative field width is treated as a left adjustment flag followed by a positive field width; a negative precision is treated as though it were missing. If a single format directive mixes positional (nn$) and non-positional arguments, the results are undefined.

The conversion specifiers and their meanings are:

diouxX The *int* (or appropriate variant) argument is converted to signed decimal (**d** and **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters "abcdef" are used for **x** conversions; the letters "ABCDEF" are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.

DOU The *long int* argument is converted to signed decimal, unsigned octal, or unsigned decimal, as if the format had been **ld**, **lo**, or **lu** respectively. These conversion characters are deprecated, and will eventually disappear.

eE The *double* argument is rounded and converted in the style [-]*d.ddd*e+-*dd* where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An **E** conversion uses the letter 'E' (rather than 'e') to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, positive and negative infinity are represented as inf and -inf respectively when using the lowercase conversion character, and INF and -INF respectively when using the uppercase conversion character. Similarly, NaN is represented as nan when using the lowercase conversion, and NAN when using the uppercase conversion.

fF The *double* argument is rounded and converted to decimal notation in the style [-]*ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

gG The *double* argument is converted in style **f** or **e** (or **F** or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

aA The *double* argument is converted to hexadecimal notation in the style [-]0x*h.hhhp*[+-]*d*, where the number of digits after the hexadecimal-point character is equal to the precision specification. If the precision is missing, it is taken as enough to exactly represent the floating-point number; if the precision is explicitly zero, no hexadecimal-point character appears. This is an exact conversion of the mantissa+exponent internal floating point representation; the [-]0x*h.hhh* portion represents exactly the mantissa; only denormalized mantissas have a zero value to the left of the hexadecimal point. The **p** is a literal character ‘p’; the exponent is preceded by a positive or negative sign and is represented in decimal, using only enough characters to represent the exponent. The **A** conversion uses the prefix "0X" (rather than "0x"), the letters "ABCDEF" (rather than "abcdef") to represent the hex digits, and the letter ‘P’ (rather than ‘p’) to separate the mantissa and exponent.

C Treated as **c** with the **l** (ell) modifier.

c The *int* argument is converted to an *unsigned char*, then to a *wchar_t* as if by btowc(3), and the resulting character is written.

If the **l** (ell) modifier is used, the *wint_t* argument is converted to a *wchar_t* and written.

S Treated as **s** with the **l** (ell) modifier.

s The *char* * argument is expected to be a pointer to an array of character type (pointer to a string) containing a multibyte sequence. Characters from the array are converted to wide characters and written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.

If the **l** (ell) modifier is used, the *wchar_t* * argument is expected to be a pointer to an array of wide characters (pointer to a wide string). Each wide character in the string is written. Wide characters from the array are written up to (but not including) a terminating wide NUL character; if a precision is specified, no more than the number specified are written (including shift sequences). If a precision is given, no null character need be present; if the precision is not specified, or is greater than the number of characters in the string, the array must contain a terminating wide NUL character.

p The *void* * pointer argument is printed in hexadecimal (as if by '%#x' or '%#lx').

n The number of characters written so far is stored into the integer indicated by the *int* * (or variant) pointer argument. No argument is converted.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

The decimal point character is defined in the program's locale (category LC_NUMERIC).

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

SEE ALSO

btowc(3), fputws(3), printf(3), putwc(3), setlocale(3), wcsrtombs(3), wscanf(3)

STANDARDS

Subject to the caveats noted in the *BUGS* section of printf(3), the **wprintf**(), **fwprintf**(), **swprintf**(), **vwprintf**(), **vfwprintf**() and **vswprintf**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

SECURITY CONSIDERATIONS

Refer to printf(3).

NAME

wait, **waitid**, **waitpid**, **wait3**, **wait4**, **wait6** - wait for processes to change status

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t
wait(int *status);

pid_t
waitpid(pid_t wpid, int *status, int options);

#include <signal.h>

int
waitid(idtype_t idtype, id_t id, siginfo_t *info, int options);

#include <sys/time.h>
#include <sys/resource.h>

pid_t
wait3(int *status, int options, struct rusage *rusage);

pid_t
wait4(pid_t wpid, int *status, int options, struct rusage *rusage);

pid_t
wait6(idtype_t idtype, id_t id, int *status, int options, struct __wrusage *wrusage, siginfo_t *infop);
```

DESCRIPTION

The **wait**() function suspends execution of its calling thread until *status* information is available for a child process or a signal is received. On return from a successful **wait**() call, the *status* area contains information about the process that reported a status change as defined below.

The **wait4**() and **wait6**() system calls provide a more general interface for programs that need to wait for specific child processes, that need resource utilization statistics accumulated by child processes, or that

require options. The other wait functions are implemented using either **wait4**() or **wait6**().

The **wait6**() function is the most general function in this family and its distinct features are:

All of the desired process statuses to be waited on must be explicitly specified in *options*. The **wait**(), **waitpid**(), **wait3**(), and **wait4**() functions all implicitly wait for exited and trapped processes, but the **waitid**() and **wait6**() functions require the corresponding WEXITED and WTRAPPED flags to be explicitly specified. This allows waiting for processes which have experienced other status changes without having to also handle the exit status from terminated processes.

The **wait6**() function accepts a *wrusage* argument which points to a structure defined as:

```
struct __wrusage {
        struct rusage   wru_self;
        struct rusage   wru_children;
};
```

This allows the calling process to collect resource usage statistics from both its own child process as well as from its grand children. When no resource usage statistics are needed this pointer can be NULL.

The last argument *infop* must be either NULL or a pointer to a *siginfo_t* structure. If non-NULL, the structure is filled with the same data as for a SIGCHLD signal delivered when the process changed state.

The set of child processes to be queried is specified by the arguments *idtype* and *id*. The separate *idtype* and *id* arguments support many other types of identifiers in addition to process IDs and process group IDs.

- If *idtype* is P_PID, **waitid**() and **wait6**() wait for the child process with a process ID equal to (pid_t)id.
- If *idtype* is P_PGID, **waitid**() and **wait6**() wait for the child process with a process group ID equal to (pid_t)id.
- If *idtype* is P_ALL, **waitid**() and **wait6**() wait for any child process and the id is ignored.
- If *idtype* is P_PID or P_PGID and the id is zero, **waitid**() and **wait6**() wait for any child process in the same process group as the caller.

Non-standard identifier types supported by this implementation of **waitid**() and **wait6**() are:

P_UID Wait for processes whose effective user ID is equal to (uid_t) *id*.

P_GID Wait for processes whose effective group ID is equal to (gid_t) *id*.

P_SID Wait for processes whose session ID is equal to *id*. If the child process started its own session, its session ID will be the same as its process ID. Otherwise the session ID of a child process will match the caller's session ID.

P_JAILID Waits for processes within a jail whose jail identifier is equal to *id*.

For the **waitpid**() and **wait4**() functions, the single *wpid* argument specifies the set of child processes for which to wait.

- If *wpid* is -1, the call waits for any child process.
- If *wpid* is 0, the call waits for any child process in the process group of the caller.
- If *wpid* is greater than zero, the call waits for the process with process ID *wpid*.
- If *wpid* is less than -1, the call waits for any process whose process group ID equals the absolute value of *wpid*.

The *status* argument is defined below.

The *options* argument contains the bitwise OR of any of the following options.

WCONTINUED Report the status of selected processes that have continued from a job control stop by receiving a SIGCONT signal.

WNOHANG Do not block when there are no processes wishing to report status.

WUNTRACED Report the status of selected processes which are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal.

WSTOPPED An alias for WUNTRACED.

WTRAPPED Report the status of selected processes which are being traced via ptrace(2) and have trapped or reached a breakpoint. This flag is implicitly set for the functions **wait**(), **waitpid**(), **wait3**(), and **wait4**().
For the **waitid**() and **wait6**() functions, the flag has to be explicitly included in *options*

if status reports from trapped processes are expected.

WEXITED Report the status of selected processes which have terminated. This flag is implicitly set for the functions **wait**(), **waitpid**(), **wait3**(), and **wait4**().
For the **waitid**() and **wait6**() functions, the flag has to be explicitly included in *options* if status reports from terminated processes are expected.

WNOWAIT Keep the process whose status is returned in a waitable state. The process may be waited for again after this call completes.

For the **waitid**() and **wait6**() functions, at least one of the options WEXITED, WUNTRACED, WSTOPPED, WTRAPPED, or WCONTINUED must be specified. Otherwise there will be no events for the call to report. To avoid hanging indefinitely in such a case these functions return -1 with errno set to EINVAL.

If *rusage* is non-NULL, a summary of the resources used by the terminated process and all its children is returned.

If *wrusage* is non-NULL, separate summaries are returned for the resources used by the terminated process and the resources used by all its children.

If *infop* is non-NULL, a siginfo_t structure is returned with the *si_signo* field set to SIGCHLD and the *si_pid* field set to the process ID of the process reporting status. For the exited process, the *si_status* field of the siginfo_t structure contains the full 32 bit exit status passed to _exit(2); the *status* argument of other calls only returns 8 lowest bits of the exit status.

When the WNOHANG option is specified and no processes wish to report status, **waitid**() sets the *si_signo* and *si_pid* fields in *infop* to zero. Checking these fields is the only way to know if a status change was reported.

When the WNOHANG option is specified and no processes wish to report status, **wait4**() and **wait6**() return a process id of 0.

The **wait**() call is the same as **wait4**() with a *wpid* value of -1, with an *options* value of zero, and a *rusage* value of NULL. The **waitpid**() function is identical to **wait4**() with an *rusage* value of NULL. The older **wait3**() call is the same as **wait4**() with a *wpid* value of -1. The **wait4**() function is identical to **wait6**() with the flags WEXITED and WTRAPPED set in *options* and *infop* set to NULL.

The following macros may be used to test the current status of the process. Exactly one of the following four macros will evaluate to a non-zero (true) value:

WIFCONTINUED(*status*)
True if the process has not terminated, and has continued after a job control stop. This macro can be true only if the wait call specified the WCONTINUED option.

WIFEXITED(*status*)
True if the process terminated normally by a call to _exit(2) or exit(3).

WIFSIGNALED(*status*)
True if the process terminated due to receipt of a signal.

WIFSTOPPED(*status*)
True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the WUNTRACED option or if the child process is being traced (see ptrace(2)).

Depending on the values of those macros, the following macros produce the remaining status information about the child process:

WEXITSTATUS(*status*)
If **WIFEXITED**(*status*) is true, evaluates to the low-order 8 bits of the argument passed to _exit(2) or exit(3) by the child.

WTERMSIG(*status*)
If **WIFSIGNALED**(*status*) is true, evaluates to the number of the signal that caused the termination of the process.

WCOREDUMP(*status*)
If **WIFSIGNALED**(*status*) is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received.

WSTOPSIG(*status*)
If **WIFSTOPPED**(*status*) is true, evaluates to the number of the signal that caused the process to stop.

NOTES

See sigaction(2) for a list of termination signals. A status of 0 indicates normal termination.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned the parent process 1 ID (the init process ID).

If a signal is caught while any of the **wait**() calls are pending, the call may be interrupted or restarted when the signal-catching routine returns, depending on the options in effect for the signal; see discussion of SA_RESTART in sigaction(2).

The implementation queues one SIGCHLD signal for each child process whose status has changed; if **wait**() returns because the status of a child process is available, the pending SIGCHLD signal associated with the process ID of the child process will be discarded. Any other pending SIGCHLD signals remain pending.

If SIGCHLD is blocked and **wait**() returns because the status of a child process is available, the pending SIGCHLD signal will be cleared unless another status of the child process is available.

RETURN VALUES

If **wait**() returns due to a stopped, continued, or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

If **wait6**(), **wait4**(), **wait3**(), or **waitpid**() returns due to a stopped, continued, or terminated child process, the process ID of the child is returned to the calling process. If there are no children not previously awaited, -1 is returned with *errno* set to ECHILD. Otherwise, if WNOHANG is specified and there are no stopped, continued or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of -1 is returned and *errno* is set to indicate the error.

If **waitid**() returns because one or more processes have a state change to report, 0 is returned. If an error is detected, a value of -1 is returned and *errno* is set to indicate the error. If WNOHANG is specified and there are no stopped, continued or exited children, 0 is returned. The *si_signo* and *si_pid* fields of *infop* must be checked against zero to determine if a process reported status.

wait() called with -1 to wait for any child process will ignore a child that is referenced by a process descriptor (see pdfork(2)). Specific processes can still be waited on by specifying the process ID.

ERRORS

The **wait**() function will fail and return immediately if:

[ECHILD] The calling process has no existing unwaited-for child processes.

[ECHILD] No status from the terminated child process is available because the calling process has asked the system to discard such status by ignoring the signal SIGCHLD or setting the flag SA_NOCLDWAIT for that signal.

[EFAULT] The *status* or *rusage* argument points to an illegal address. (May not be detected

before exit of a child process.)

[EINTR] The call was interrupted by a caught signal, or the signal did not have the SA_RESTART flag set.

[EINVAL] An invalid value was specified for *options*, or *idtype* and *id* do not specify a valid set of processes.

SEE ALSO

_exit(2), ptrace(2), sigaction(2), exit(3), siginfo(3)

STANDARDS

The **wait**(), **waitpid**(), and **waitid**() functions are defined by POSIX; **wait6**(), **wait4**(), and **wait3**() are not specified by POSIX. The **WCOREDUMP**() macro is an extension to the POSIX interface.

The ability to use the WNOWAIT flag with **waitpid**() is an extension; POSIX only permits this flag with **waitid**().

HISTORY

The **wait**() function appeared in Version 1 AT&T UNIX.

NAME

wmemchr, **wmemcmp**, **wmemcpy**, **wmemmove**, **wmemset**, **wcpcpy**, **wcpncpy**, **wcscasecmp**, **wcscat**, **wcschr**, **wcscmp**, **wcscpy**, **wcscspn**, **wcsdup**, **wcslcat**, **wcslcpy**, **wcslen**, **wcsncasecmp**, **wcsncat**, **wcsncmp**, **wcsncpy**, **wcsnlen**, **wcspbrk**, **wcsrchr**, **wcsspn**, **wcsstr** - wide character string manipulation operations

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

*wchar_t **
wmemchr(*const wchar_t *s*, *wchar_t c*, *size_t n*);

int
wmemcmp(*const wchar_t *s1*, *const wchar_t *s2*, *size_t n*);

*wchar_t **
wmemcpy(*wchar_t * restrict s1*, *const wchar_t * restrict s2*, *size_t n*);

*wchar_t **
wmemmove(*wchar_t *s1*, *const wchar_t *s2*, *size_t n*);

*wchar_t **
wmemset(*wchar_t *s*, *wchar_t c*, *size_t n*);

*wchar_t **
wcpcpy(*wchar_t *s1*, *wchar_t *s2*);

*wchar_t **
wcpncpy(*wchar_t *s1*, *wchar_t *s2*, *size_t n*);

int
wcscasecmp(*const wchar_t *s1*, *const wchar_t *s2*);

*wchar_t **
wcscat(*wchar_t * restrict s1*, *const wchar_t * restrict s2*);

*wchar_t **

wcschr(*const wchar_t *s*, *wchar_t c*);

int
wcscmp(*const wchar_t *s1*, *const wchar_t *s2*);

*wchar_t **
wcscpy(*wchar_t * restrict s1*, *const wchar_t * restrict s2*);

size_t
wcscspn(*const wchar_t *s1*, *const wchar_t *s2*);

*wchar_t **
wcsdup(*const wchar_t *s*);

size_t
wcslcat(*wchar_t *s1*, *const wchar_t *s2*, *size_t n*);

size_t
wcslcpy(*wchar_t *s1*, *const wchar_t *s2*, *size_t n*);

size_t
wcslen(*const wchar_t *s*);

int
wcsncasecmp(*const wchar_t *s1*, *const wchar_t *s2*, *size_t n*);

*wchar_t **
wcsncat(*wchar_t * restrict s1*, *const wchar_t * restrict s2*, *size_t n*);

int
wcsncmp(*const wchar_t *s1*, *const wchar_t * s2*, *size_t n*);

*wchar_t **
wcsncpy(*wchar_t * restrict s1*, *const wchar_t * restrict s2*, *size_t n*);

size_t
wcsnlen(*const wchar_t *s*, *size_t maxlen*);

*wchar_t **
wcspbrk(*const wchar_t *s1*, *const wchar_t *s2*);

*wchar_t **
wcsrchr(*const wchar_t *s*, *wchar_t c*);

size_t
wcsspn(*const wchar_t *s1*, *const wchar_t *s2*);

*wchar_t **
wcsstr(*const wchar_t * restrict s1*, *const wchar_t * restrict s2*);

DESCRIPTION

The functions implement string manipulation operations over wide character strings. For a detailed description, refer to documents for the respective single-byte counterpart, such as memchr(3).

SEE ALSO

memchr(3), memcmp(3), memcpy(3), memmove(3), memset(3), stpcpy(3), stpncpy(3), strcasecmp(3), strcat(3), strchr(3), strcmp(3), strcpy(3), strcspn(3), strdup(3), strlcat(3), strlcpy(3), strlen(3), strncat(3), strncmp(3), strncpy(3), strnlen(3), strpbrk(3), strrchr(3), strspn(3), strstr(3)

STANDARDS

These functions conform to ISO/IEC 9899:1999 ("ISO C99"), with the exception of **wcpcpy**(), **wcpncpy**(), **wcscasecmp**(), **wcsdup**(), **wcsncasecmp**(), and **wcsnlen**(), which conform to IEEE Std 1003.1-2008 ("POSIX.1"); and **wcslcat**() and **wcslcpy**(), which are extensions.

NAME

wcscoll - compare wide strings according to current collation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

int
wcscoll(*const wchar_t *s1*, *const wchar_t *s2*);

DESCRIPTION

The **wcscoll**() function compares the null-terminated strings *s1* and *s2* according to the current locale collation order. In the "C" locale, **wcscoll**() is equivalent to **wcscmp**().

RETURN VALUES

The **wcscoll**() function returns an integer greater than, equal to, or less than 0, if *s1* is greater than, equal to, or less than *s2*.

No return value is reserved to indicate errors; callers should set *errno* to 0 before calling **wcscoll**(). If it is non-zero upon return from **wcscoll**(), an error has occurred.

ERRORS

The **wcscoll**() function will fail if:

[EILSEQ] An invalid wide character code was specified.

[ENOMEM] Cannot allocate enough memory for temporary buffers.

SEE ALSO

setlocale(3), strcoll(3), wcscmp(3), wcsxfrm(3)

STANDARDS

The **wcscoll**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

BUGS

The current implementation of **wcscoll**() only works in single-byte LC_CTYPE locales, and falls back to using **wcscmp**() in locales with extended character sets.

NAME

wcsftime - convert date and time to a wide-character string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

size_t
wcsftime(*wchar_t * restrict wcs*, *size_t maxsize*, *const wchar_t * restrict format*,
*const struct tm * restrict timeptr*);

DESCRIPTION

The **wcsftime**() function is equivalent to the **strftime**() function except for the types of its arguments. Refer to strftime(3) for a detailed description.

COMPATIBILITY

Some early implementations of **wcsftime**() had a *format* argument with type *const char* * instead of *const wchar_t *.

SEE ALSO

strftime(3)

STANDARDS

The **wcsftime**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

wcsrtombs, **wcsnrtombs** - convert a wide-character string to a character string (restartable)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

size_t
wcsrtombs(*char * restrict dst*, *const wchar_t ** restrict src*, *size_t len*, *mbstate_t * restrict ps*);

size_t
wcsnrtombs(*char * restrict dst*, *const wchar_t ** restrict src*, *size_t nwc*, *size_t len*,
*mbstate_t * restrict ps*);

DESCRIPTION

The **wcsrtombs**() function converts a string of wide characters indirectly pointed to by *src* to a corresponding multibyte character string stored in the array pointed to by *dst*. No more than *len* bytes are written to *dst*.

If *dst* is NULL, no characters are stored.

If *dst* is not NULL, the pointer pointed to by *src* is updated to point to the character after the one that conversion stopped at. If conversion stops because a null character is encountered, **src* is set to NULL.

The *mbstate_t* argument, *ps*, is used to keep track of the shift state. If it is NULL, **wcsrtombs**() uses an internal, static *mbstate_t* object, which is initialized to the initial conversion state at program startup.

The **wcsnrtombs**() function behaves identically to **wcsrtombs**(), except that conversion stops after reading at most *nwc* characters from the buffer pointed to by *src*.

RETURN VALUES

The **wcsrtombs**() and **wcsnrtombs**() functions return the number of bytes stored in the array pointed to by *dst* (not including any terminating null), if successful, otherwise it returns (*size_t*)-1.

ERRORS

The **wcsrtombs**() and **wcsnrtombs**() functions will fail if:

[EILSEQ] An invalid wide character was encountered.

[EINVAL] The conversion state is invalid.

SEE ALSO

mbsrtowcs(3), wcrtomb(3), wcstombs(3)

STANDARDS

The **wcsrtombs**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

The **wcsnrtombs**() function is an extension to the standard.

NAME

wcstof, **wcstod**, **wcstold** - convert string to *float*, *double* or *long double*

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

float
wcstof(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*);

long double
wcstold(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*);

double
wcstod(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*);

DESCRIPTION

The **wcstof**(), **wcstod**() and **wcstold**() functions are the wide-character versions of the **strtof**(), **strtod**() and **strtold**() functions. Refer to strtod(3) for details.

SEE ALSO

strtod(3), wcstol(3)

STANDARDS

The **wcstof**(), **wcstod**() and **wcstold**() functions conform to ISO/IEC 9899:1999 ("ISO C99").

NAME

wcstol, **wcstoul**, **wcstoll**, **wcstoull**, **wcstoimax**, **wcstoumax** - convert a wide character string value to a *long*, *unsigned long*, *long long*, *unsigned long long*, *intmax_t* or *uintmax_t* integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

long
wcstol(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*, *int base*);

unsigned long
wcstoul(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*, *int base*);

long long
wcstoll(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*, *int base*);

unsigned long long
wcstoull(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*, *int base*);

#include <inttypes.h>

intmax_t
wcstoimax(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*, *int base*);

uintmax_t
wcstoumax(*const wchar_t * restrict nptr*, *wchar_t ** restrict endptr*, *int base*);

DESCRIPTION

The **wcstol**(), **wcstoul**(), **wcstoll**(), **wcstoull**(), **wcstoimax**() and **wcstoumax**() functions are wide-character versions of the **strtol**(), **strtoul**(), **strtoll**(), **strtoull**(), **strtoimax**() and **strtoumax**() functions, respectively. Refer to their manual pages (for example strtol(3)) for details.

SEE ALSO

strtol(3), strtoul(3)

STANDARDS

The **wcstol**(), **wcstoul**(), **wcstoll**(), **wcstoull**(), **wcstoimax**() and **wcstoumax**() functions conform to

ISO/IEC 9899:1999 ("ISO C99").

NAME

wcstok - split wide-character string into tokens

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

*wchar_t **
wcstok(*wchar_t * restrict str*, *const wchar_t * restrict sep*, *wchar_t ** restrict last*);

DESCRIPTION

The **wcstok**() function is used to isolate sequential tokens in a null-terminated wide character string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The first time that **wcstok**() is called, *str* should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, *sep*, must be supplied each time, and may change between calls. The context pointer *last* must be provided on each call.

The **wcstok**() function is the wide character counterpart of the **strtok_r**() function.

RETURN VALUES

The **wcstok**() function returns a pointer to the beginning of each subsequent token in the string, after replacing the token itself with a null wide character (L'\0'). When no more tokens remain, a null pointer is returned.

EXAMPLES

The following code fragment splits a wide character string on ASCII space, tab and newline characters and writes the tokens to standard output:

```
const wchar_t *seps = L" \t\n";
wchar_t *last, *tok, text[] = L" \none\ttwo\t\tthree  \n";

for (tok = wcstok(text, seps, &last); tok != NULL;
    tok = wcstok(NULL, seps, &last))
        wprintf(L"%ls\n", tok);
```

COMPATIBILITY

Some early implementations of **wcstok**() omit the context pointer argument, *last*, and maintain state across calls in a static variable like **strtok**() does.

SEE ALSO

strtok(3), wcschr(3), wcscspn(3), wcspbrk(3), wcsrchr(3), wcsspn(3)

STANDARDS

The **wcstok**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

wcstombs - convert a wide-character string to a character string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

size_t
wcstombs(*char * restrict mbstring*, *const wchar_t * restrict wcstring*, *size_t nbytes*);

DESCRIPTION

The **wcstombs**() function converts a wide character string *wcstring* into a multibyte character string, *mbstring*, beginning in the initial conversion state. Up to *nbytes* bytes are stored in *mbstring*. Partial multibyte characters at the end of the string are not stored. The multibyte character string is null terminated if there is room.

RETURN VALUES

The **wcstombs**() function returns the number of bytes converted (not including any terminating null), if successful, otherwise it returns (*size_t*)-1.

ERRORS

The **wcstombs**() function will fail if:

[EILSEQ] An invalid wide character was encountered.

[EINVAL] The conversion state is invalid.

SEE ALSO

mbstowcs(3), multibyte(3), wcsrtombs(3), wctomb(3)

STANDARDS

The **wcstombs**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

wcswidth - number of column positions in wide-character string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

int
wcswidth(*const wchar_t *pwcs*, *size_t n*);

DESCRIPTION

The **wcswidth**() function determines the number of column positions required for the first *n* characters of *pwcs*, or until a null wide character (L'\0') is encountered.

RETURN VALUES

The **wcswidth**() function returns 0 if *pwcs* is an empty string (L""), -1 if a non-printing wide character is encountered, otherwise it returns the number of column positions occupied.

SEE ALSO

iswprint(3), wcwidth(3)

STANDARDS

The **wcswidth**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

wcsxfrm - transform a wide string under locale

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

size_t
wcsxfrm(*wchar_t * restrict dst*, *const wchar_t * restrict src*, *size_t n*);

DESCRIPTION

The **wcsxfrm**() function transforms a null-terminated wide character string pointed to by *src* according to the current locale collation order then copies the transformed string into *dst*. No more than *n* wide characters are copied into *dst*, including the terminating null character added. If *n* is set to 0 (it helps to determine an actual size needed for transformation), *dst* is permitted to be a NULL pointer.

Comparing two strings using **wcscmp**() after **wcsxfrm**() is equivalent to comparing two original strings with **wcscoll**().

RETURN VALUES

Upon successful completion, **wcsxfrm**() returns the length of the transformed string not including the terminating null character. If this value is *n* or more, the contents of *dst* are indeterminate.

SEE ALSO

setlocale(3), strxfrm(3), wcscmp(3), wcscoll(3)

STANDARDS

The **wcsxfrm**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

BUGS

The current implementation of **wcsxfrm**() only works in single-byte LC_CTYPE locales, and falls back to using **wcsncpy**() in locales with extended character sets.

Comparing two strings using **wcscmp**() after **wcsxfrm**() is *not* always equivalent to comparison with **wcscoll**(); **wcsxfrm**() only stores information about primary collation weights into *dst*, whereas **wcscoll**() compares characters using both primary and secondary weights.

NAME

wctomb - convert a wide-character code to a character

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdlib.h>

int
wctomb(*char *mbchar*, *wchar_t wchar*);

DESCRIPTION

The **wctomb**() function converts a wide character *wchar* into a multibyte character and stores the result in *mbchar*. The object pointed to by *mbchar* must be large enough to accommodate the multibyte character, which may be up to MB_LEN_MAX bytes.

A call with a null *mbchar* pointer returns nonzero if the current locale requires shift states, zero otherwise; if shift states are required, the shift state is reset to the initial state.

RETURN VALUES

If *mbchar* is NULL, the **wctomb**() function returns nonzero if shift states are supported, zero otherwise. If *mbchar* is valid, **wctomb**() returns the number of bytes processed in *mbchar*, or -1 if no multibyte character could be recognized or converted. In this case, **wctomb**()'s internal conversion state is undefined.

ERRORS

The **wctomb**() function will fail if:

[EILSEQ] An invalid multibyte sequence was detected.

[EINVAL] The internal conversion state is invalid.

SEE ALSO

mbtowc(3), wcrtomb(3), wcstombs(3), wctob(3)

STANDARDS

The **wctomb**() function conforms to ISO/IEC 9899:1999 ("ISO C99").

NAME

wcwidth - number of column positions of a wide-character code

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wchar.h>

int
wcwidth(*wchar_t wc*);

DESCRIPTION

The **wcwidth**() function determines the number of column positions required to display the wide character *wc*.

RETURN VALUES

The **wcwidth**() function returns 0 if the *wc* argument is a null wide character (L'\0'), -1 if *wc* is not printable, otherwise it returns the number of column positions the character occupies.

EXAMPLES

This code fragment reads text from standard input and breaks lines that are more than 20 column positions wide, similar to the fold(1) utility:

```
wint_t ch;
int column, w;

column = 0;
while ((ch = getwchar()) != WEOF) {
        w = wcwidth(ch);
        if (w > 0 && column + w >= 20) {
                putwchar(L'\n');
                column = 0;
        }
        putwchar(ch);
        if (ch == L'\n')
                column = 0;
        else if (w > 0)
                column += w;
}
```

SEE ALSO

iswprint(3), wcswidth(3)

STANDARDS

The **wcwidth**() function conforms to IEEE Std 1003.1-2001 ("POSIX.1").

NAME

wordexp - perform shell-style word expansions

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <wordexp.h>

int
wordexp(*const char * restrict words*, *wordexp_t * restrict we*, *int flags*);

void
wordfree(*wordexp_t *we*);

DESCRIPTION

The **wordexp**() function performs shell-style word expansion on *words* and places the list of words into the *we_wordv* member of *we*, and the number of words into *we_wordc*.

The *flags* argument is the bitwise inclusive OR of any of the following constants:

WRDE_APPEND	Append the words to those generated by a previous call to **wordexp**().
WRDE_DOOFFS	As many NULL pointers as are specified by the *we_offs* member of *we* are added to the front of *we_wordv*.
WRDE_NOCMD	Disallow command substitution in *words*. See the note in *BUGS* before using this.
WRDE_REUSE	The *we* argument was passed to a previous successful call to **wordexp**() but has not been passed to **wordfree**(). The implementation may reuse the space allocated to it.
WRDE_SHOWERR	Do not redirect shell error messages to */dev/null*.
WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.

The *wordexp_t* structure is defined in <*wordexp.h*> as:

```
typedef struct {
```

```
        size_t  we_wordc;       /* count of words matched */
        char    **we_wordv;     /* pointer to list of words */
        size_t  we_offs;  /* slots to reserve in we_wordv */
} wordexp_t;
```

The **wordfree**() function frees the memory allocated by **wordexp**().

IMPLEMENTATION NOTES

The **wordexp**() function is implemented using the undocumented **freebsd_wordexp** shell built-in command.

RETURN VALUES

The **wordexp**() function returns zero if successful, otherwise it returns one of the following error codes:

WRDE_BADCHAR — The *words* argument contains one of the following unquoted characters: <newline>, ‘|’, ‘&’, ‘;’, ‘<’, ‘>’, ‘(’, ‘)’, ‘{’, ‘}’.

WRDE_BADVAL — An error after successful parsing, such as an attempt to expand an undefined shell variable with WRDE_UNDEF set in *flags*.

WRDE_CMDSUB — An attempt was made to use command substitution and WRDE_NOCMD is set in *flags*.

WRDE_NOSPACE — Not enough memory to store the result or an error during fork(2).

WRDE_SYNTAX — Shell syntax error in *words*.

The **wordfree**() function returns no value.

ENVIRONMENT

IFS — Field separator.

EXAMPLES

Invoke the editor on all *.c* files in the current directory and */etc/motd* (error checking omitted):

```
wordexp_t we;

wordexp("${EDITOR:-vi} *.c /etc/motd", &we, 0);
execvp(we.we_wordv[0], we.we_wordv);
```

DIAGNOSTICS

Diagnostic messages from the shell are written to the standard error output if WRDE_SHOWERR is set in *flags*.

SEE ALSO

sh(1), fnmatch(3), glob(3), popen(3), system(3)

STANDARDS

The **wordexp**() and **wordfree**() functions conform to IEEE Std 1003.1-2001 ("POSIX.1").

BUGS

The current **wordexp**() implementation does not recognize multibyte characters other than UTF-8, since the shell (which it invokes to perform expansions) does not.

SECURITY CONSIDERATIONS

Pathname generation may create output that is exponentially larger than the input size.

Although this implementation detects command substitution reliably for WRDE_NOCMD, the attack surface remains fairly large. Also, some other implementations (such as older versions of this one) may execute command substitutions even if WRDE_NOCMD is set.

NAME

xdr, **xdr_array**, **xdr_bool**, **xdr_bytes**, **xdr_char**, **xdr_destroy**, **xdr_double**, **xdr_enum**, **xdr_float**, **xdr_free**, **xdr_getpos**, **xdr_hyper**, **xdr_inline**, **xdr_int**, **xdr_long**, **xdr_longlong_t**, **xdrmem_create**, **xdr_opaque**, **xdr_pointer**, **xdrrec_create**, **xdrrec_endofrecord**, **xdrrec_eof**, **xdrrec_skiprecord**, **xdr_reference**, **xdr_setpos**, **xdr_short**, **xdr_sizeof**, **xdrstdio_create**, **xdr_string**, **xdr_u_char**, **xdr_u_hyper**, **xdr_u_int**, **xdr_u_long**, **xdr_u_longlong_t**, **xdr_u_short**, **xdr_union**, **xdr_vector**, **xdr_void**, **xdr_wrapstring** - library routines for external data representation

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/types.h>
#include <rpc/xdr.h>

See *DESCRIPTION* for function declarations.

DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

int
xdr_array(*XDR *xdrs*, *char **arrp*, *u_int *sizep*, *u_int maxsize*, *u_int elsize*, *xdrproc_t elproc*)

A filter primitive that translates between variable-length arrays and their corresponding external representations. The *arrp* argument is the address of the pointer to the array, while *sizep* is the address of the element count of the array; this element count cannot exceed *maxsize*. The *elsize* argument is the **sizeof** each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

int
xdr_bool(*XDR *xdrs*, *bool_t *bp*)

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

int
xdr_bytes(*XDR *xdrs*, *char **sp*, *u_int *sizep*, *u_int maxsize*)

A filter primitive that translates between counted byte strings and their external representations. The *sp* argument is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns one if it succeeds, zero otherwise.

int
xdr_char(*XDR *xdrs*, *char *cp*)

A filter primitive that translates between C characters and their external representations. This routine returns one if it succeeds, zero otherwise. Note: encoded characters are not packed, and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider **xdr_bytes**(), **xdr_opaque**() or **xdr_string**().

void
xdr_destroy(*XDR *xdrs*)

A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking **xdr_destroy**() is undefined.

int
xdr_double(*XDR *xdrs*, *double *dp*)

A filter primitive that translates between C *double* precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_enum(*XDR *xdrs*, *enum_t *ep*)

A filter primitive that translates between C *enum*s (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_float(*XDR *xdrs*, *float *fp*)

A filter primitive that translates between C *float*s and their external representations. This routine returns one if it succeeds, zero otherwise.

void
xdr_free(*xdrproc_t proc*, *void *objp*)

Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note: the pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

u_int
xdr_getpos(*XDR *xdrs*)

A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

int
xdr_hyper(*XDR *xdrs*, *quad_t *llp*)

A filter primitive that translates between ANSI C *long long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

*long **
xdr_inline(*XDR *xdrs*, *int len*)

A macro that invokes the in-line routine associated with the XDR stream, *xdrs*. The routine returns a pointer to a contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note: pointer is cast to *long **.

Warning: **xdr_inline**() may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

int
xdr_int(*XDR *xdrs*, *int *ip*)

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_long(*XDR *xdrs*, *long *lp*)

A filter primitive that translates between C *long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

int

xdr_longlong_t(*XDR *xdrs*, *quad_t *llp*)

A filter primitive that translates between ANSI C *long long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

void
xdrmem_create(*XDR *xdrs*, *char *addr*, *u_int size*, *enum xdr_op op*)

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* argument determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

int
xdr_opaque(*XDR *xdrs*, *char *cp*, *u_int cnt*)

A filter primitive that translates between fixed size opaque data and its external representation. The *cp* argument is the address of the opaque object, and *cnt* is its size in bytes. This routine returns one if it succeeds, zero otherwise.

int
xdr_pointer(*XDR *xdrs*, *char **objpp*, *u_int objsize*, *xdrproc_t xdrobj*)

Like **xdr_reference**() except that it serializes NULL pointers, whereas **xdr_reference**() does not. Thus, **xdr_pointer**() can represent recursive data structures, such as binary trees or linked lists.

void
xdrrec_create(*XDR *xdrs*, *u_int sendsize*, *u_int recvsize*, *void *handle*, *int (*readit)()*, *int (*writeit)()*)

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsize*; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize*; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, **writeit**() is called. Similarly, when a stream's input buffer is empty, **readit**() is called. The behavior of these two routines is similar to the system calls read(2) and write(2), except that *handle* is passed to the former routines as the first argument. Note: the XDR stream's *op* field must be set by the caller.

Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

int

xdrrec_endofrecord(*XDR *xdrs*, *int sendnow*)

This routine can be invoked only on streams created by **xdrrec_create**(). The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns one if it succeeds, zero otherwise.

int
xdrrec_eof(*XDR *xdrs*)

This routine can be invoked only on streams created by **xdrrec_create**(). After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

int
xdrrec_skiprecord(*XDR *xdrs*)

This routine can be invoked only on streams created by **xdrrec_create**(). It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

int
xdr_reference(*XDR *xdrs*, *char **pp*, *u_int size*, *xdrproc_t proc*)

A primitive that provides pointer chasing within structures. The *pp* argument is the address of the pointer; *size* is the **sizeof** the structure that **pp* points to; and *proc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

Warning: this routine does not understand NULL pointers. Use **xdr_pointer**() instead.

int
xdr_setpos(*XDR *xdrs*, *u_int pos*)

A macro that invokes the set position routine associated with the XDR stream *xdrs*. The *pos* argument is a position value obtained from **xdr_getpos**(). This routine returns one if the XDR stream could be repositioned, and zero otherwise.

Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

int
xdr_short(*XDR *xdrs*, *short *sp*)

A filter primitive that translates between C *short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

unsigned long
xdr_sizeof(*xdrproc_t func*, *void *data*)

This routine returns the amount of memory required to encode *data* using filter *func*.

#ifdef _STDIO_H_
/* XDR using stdio library */
void
xdrstdio_create(*XDR *xdrs*, *FILE *file*, *enum xdr_op op*)
#endif

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the Standard I/O stream *file*. The *op* argument determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Warning: the destroy routine associated with such XDR streams calls fflush(3) on the *file* stream, but never fclose(3).

int
xdr_string(*XDR *xdrs*, *char **sp*, *u_int maxsize*)

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note: *sp* is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

int
xdr_u_char(*XDR *xdrs*, *unsigned char *ucp*)

A filter primitive that translates between *unsigned* C characters and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_u_hyper(*XDR *xdrs*, *u_quad_t *ullp*)
A filter primitive that translates between *unsigned* ANSI C *long long* integers and their external

representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_u_int(*XDR *xdrs*, *unsigned *up*)

A filter primitive that translates between C *unsigned* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_u_long(*XDR *xdrs*, *unsigned long *ulp*)

A filter primitive that translates between C *unsigned long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_u_longlong_t(*XDR *xdrs*, *u_quad_t *ullp*)
A filter primitive that translates between *unsigned* ANSI C *long long* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_u_short(*XDR *xdrs*, *unsigned short *usp*)

A filter primitive that translates between C *unsigned short* integers and their external representations. This routine returns one if it succeeds, zero otherwise.

int
xdr_union(*XDR *xdrs*, *enum_t *dscmp*, *char *unp*, *const struct xdr_discrim *choices*, *xdrproc_t defaultarm*)

A filter primitive that translates between a discriminated C *union* and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an *enum_t*. Next the union located at *unp* is translated. The *choices* argument is a pointer to an array of *xdr_discrim* structures. Each structure contains an ordered pair of [*value*, *proc*]. If the union's discriminant is equal to the associated *value*, then the **proc**() is called to translate the union. The end of the *xdr_discrim* structure array is denoted by a routine of value NULL. If the discriminant is not found in the *choices* array, then the **defaultarm**() procedure is called (if it is not NULL). Returns one if it succeeds, zero otherwise.

int
xdr_vector(*XDR *xdrs*, *char *arrp*, *u_int size*, *u_int elsize*, *xdrproc_t elproc*)

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The *arrp* argument is the address of the pointer to the array, while *size* is the element count of the array. The *elsize* argument is the **sizeof** each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

int
xdr_void(*void*)

This routine always returns one. It may be passed to RPC routines that require a function argument, where nothing is to be done.

int
xdr_wrapstring(*XDR *xdrs*, *char **sp*)

A primitive that calls **xdr_string**(*xdrs*, *sp*, *MAXUN.UNSIGNED*); where MAXUN.UNSIGNED is the maximum value of an unsigned integer. The **xdr_wrapstring**() function is handy because the RPC package passes a maximum of two XDR routines as arguments, and **xdr_string**(), one of the most frequently used primitives, requires three. Returns one if it succeeds, zero otherwise.

SEE ALSO

rpc(3)

eXternal Data Representation Standard: Protocol Specification.

eXternal Data Representation: Sun Technical Notes.

XDR: External Data Representation Standard, Sun Microsystems, Inc., USC-ISI, RFC1014.

HISTORY

The **xdr_sizeof** function first appeared in FreeBSD 9.0.

NAME

xdr_accepted_reply, **xdr_authsys_parms**, **xdr_callhdr**, **xdr_callmsg**, **xdr_opaque_auth**, **xdr_rejected_reply**, **xdr_replymsg** - XDR library routines for remote procedure calls

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <rpc/rpc.h>

bool_t
xdr_accepted_reply(*XDR *xdrs*, *struct accepted_reply *ar*);

bool_t
xdr_authsys_parms(*XDR *xdrs*, *struct authsys_parms *aupp*);

bool_t
xdr_callhdr(*XDR *xdrs*, *struct rpc_msg *chdr*);

bool_t
xdr_callmsg(*XDR *xdrs*, *struct rpc_msg *cmsg*);

bool_t
xdr_opaque_auth(*XDR *xdrs*, *struct opaque_auth *ap*);

bool_t
xdr_rejected_reply(*XDR *xdrs*, *struct rejected_reply *rr*);

bool_t
xdr_replymsg(*XDR *xdrs*, *struct rpc_msg *rmsg*);

DESCRIPTION

These routines are used for describing the RPC messages in XDR language. They should normally be used by those who do not want to use the RPC package directly. These routines return TRUE if they succeed, FALSE otherwise.

Routines

See rpc(3) for the definition of the *XDR* data structure.

xdr_accepted_reply()

Used to translate between RPC reply messages and their external representation. It includes the status of the RPC call in the XDR language format. In the case of success, it also includes the call results.

xdr_authsys_parms()
Used for describing UNIX operating system credentials. It includes machine-name, uid, gid list, etc.

xdr_callhdr()
Used for describing RPC call header messages. It encodes the static part of the call message header in the XDR language format. It includes information such as transaction ID, RPC version number, program and version number.

xdr_callmsg()
Used for describing RPC call messages. This includes all the RPC call information such as transaction ID, RPC version number, program number, version number, authentication information, etc. This is normally used by servers to determine information about the client RPC call.

xdr_opaque_auth()
Used for describing RPC opaque authentication information messages.

xdr_rejected_reply()
Used for describing RPC reply messages. It encodes the rejected RPC message in the XDR language format. The message could be rejected either because of version number mis-match or because of authentication errors.

xdr_replymsg()
Used for describing RPC reply messages. It translates between the RPC reply message and its external representation. This reply could be either an acceptance, rejection or NULL.

SEE ALSO

rpc(3), xdr(3)

NAME

xlocale - Thread-safe extended locale support

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <xlocale.h>

DESCRIPTION

The extended locale support includes a set of functions for setting thread-local locales, as well convenience functions for performing locale-aware calls with a specified locale.

The core of the xlocale API is the *locale_t* type. This is an opaque type encapsulating a locale. Instances of this can be either set as the locale for a specific thread or passed directly to the *_l* suffixed variants of various standard C functions. Two special *locale_t* values are available:

- NULL refers to the current locale for the thread, or to the global locale if no locale has been set for this thread.

- LC_GLOBAL_LOCALE refers to the global locale.

The global locale is the locale set with the setlocale(3) function.

SEE ALSO

duplocale(3), freelocale(3), localeconv(3), newlocale(3), querylocale(3), uselocale(3)

CONVENIENCE FUNCTIONS

The xlocale API includes a number of *_l* suffixed convenience functions. These are variants of standard C functions that have been modified to take an explicit *locale_t* parameter as the final argument or, in the case of variadic functions, as an additional argument directly before the format string. Each of these functions accepts either NULL or LC_GLOBAL_LOCALE. In these functions, NULL refers to the C locale, rather than the thread's current locale. If you wish to use the thread's current locale, then use the unsuffixed version of the function.

These functions are exposed by including <*xlocale.h*> *after* including the relevant headers for the standard variant. For example, the strtol_l(3) function is exposed by including <*xlocale.h*> after <*stdlib.h*>, which defines strtol(3).

For reference, a complete list of the locale-aware functions that are available in this form, along with the

headers that expose them, is provided here:

<*wctype.h*> iswalnum_l(3), iswalpha_l(3), iswcntrl_l(3), iswctype_l(3), iswdigit_l(3), iswgraph_l(3), iswlower_l(3), iswprint_l(3), iswpunct_l(3), iswspace_l(3), iswupper_l(3), iswxdigit_l(3), towlower_l(3), towupper_l(3), wctype_l(3),

<*ctype.h*> digittoint_l(3), isalnum_l(3), isalpha_l(3), isblank_l(3), iscntrl_l(3), isdigit_l(3), isgraph_l(3), ishexnumber_l(3), isideogram_l(3), islower_l(3), isnumber_l(3), isphonogram_l(3), isprint_l(3), ispunct_l(3), isrune_l(3), isspace_l(3), isspecial_l(3), isupper_l(3), isxdigit_l(3), tolower_l(3), toupper_l(3)

<*inttypes.h*> strtoimax_l(3), strtoumax_l(3), wcstoimax_l(3), wcstoumax_l(3)

<*langinfo.h*> nl_langinfo_l(3)

<*monetary.h*> strfmon_l(3)

<*stdio.h*> asprintf_l(3), fprintf_l(3), fscanf_l(3), printf_l(3), scanf_l(3), snprintf_l(3), sprintf_l(3), sscanf_l(3), vasprintf_l(3), vfprintf_l(3), vfscanf_l(3), vprintf_l(3), vscanf_l(3), vsnprintf_l(3), vsprintf_l(3), vsscanf_l(3)

<*stdlib.h*> atof_l(3), atoi_l(3), atol_l(3), atoll_l(3), mblen_l(3), mbstowcs_l(3), mbtowc_l(3), strtod_l(3), strtof_l(3), strtol_l(3), strtold_l(3), strtoll_l(3), strtoq_l(3), strtoul_l(3), strtoull_l(3), strtouq_l(3), wcstombs_l(3), wctomb_l(3)

<*string.h*> strcoll_l(3), strxfrm_l(3), strcasecmp_l(3), strcasestr_l(3), strncasecmp_l(3)

<*time.h*> strftime_l(3) strptime_l(3)

<*wchar.h*> btowc_l(3), fgetwc_l(3), fgetws_l(3), fputwc_l(3), fputws_l(3), fwprintf_l(3), fwscanf_l(3), getwc_l(3), getwchar_l(3), mbrlen_l(3), mbrtowc_l(3), mbsinit_l(3), mbsnrtowcs_l(3), mbsrtowcs_l(3), putwc_l(3), putwchar_l(3), swprintf_l(3), swscanf_l(3), ungetwc_l(3), vfwprintf_l(3), vfwscanf_l(3), vswprintf_l(3), vswscanf_l(3), vwprintf_l(3), vwscanf_l(3), wcrtomb_l(3), wcscoll_l(3), wcsftime_l(3), wcsnrtombs_l(3), wcsrtombs_l(3), wcstod_l(3), wcstof_l(3), wcstol_l(3), wcstold_l(3), wcstoll_l(3), wcstoul_l(3), wcstoull_l(3), wcswidth_l(3), wcsxfrm_l(3), wctob_l(3), wcwidth_l(3), wprintf_l(3), wscanf_l(3)

<*wctype.h*> iswblank_l(3), iswhexnumber_l(3), iswideogram_l(3), iswnumber_l(3), iswphonogram_l(3), iswrune_l(3), iswspecial_l(3), nextwctype_l(3), towctrans_l(3),

wctrans_l(3)

<*xlocale.h*> localeconv_l(3)

STANDARDS

The functions conform to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The xlocale APIs first appeared in Darwin 8.0. This implementation was written by David Chisnall, under sponsorship from the FreeBSD Foundation and first appeared in FreeBSD 9.1.

CAVEATS

The setlocale(3) function, and others in the family, refer to the global locale. Other functions that depend on the locale, however, will take the thread-local locale if one has been set. This means that the idiom of setting the locale using setlocale(3), calling a locale-dependent function, and then restoring the locale will not have the expected behavior if the current thread has had a locale set using uselocale(3). You should avoid this idiom and prefer to use the *_l* suffixed versions instead.