SPARC COMPLIANCE DEFINITION 2.3

Interface Semantics

SCD 2.3 IS

SPARC INTERNATIONAL

The SPARC Compliance Interface Definition 2.3 is published and printed by SPARC International.

Any comments relating to the material contained herein may be submitted to:

SPARC International Inc.

535 Middlefield Road, Suite 210

Menlo Park, California 94025

ATTN: Ghassan Abbas (abbas@sparc.com)

Trademarks

SPARC COMPLIANCE DEFINITION 2.3

TABLE OF CONTENTS

Introduction

Introduction 1-1

libaio

aiocancel 2-1
aioread 2-2
aiowrite 2-2
aiowait 2-4

libc

_cleanup 3-1
addseverity 3-2
crypt 3-3
encrypt 3-3
setkey 3-3
setlabel 3-5
sysinfo 3-6
___errno 3-8
asctime_r 3-8
ctime_r 3-8
flockfile 3-8
funlockfile 3-8
getc_unlocked 3-8
getchar_unlocked 3-8
gmtime_r 3-8
localtime_r 3-8
putc_unlocked 3-8
putchar_unlocked 3-8
rand_r 3-8
strtok_r 3-8

libdl

Introduction 4-1
dlclose 4-2
dlerror 4-4
dlopen 4-5
dlsym 4-8

liblf

Introduction 5-1
lf_fcntl 5-2
lf_fpathconf 5-2
lf_fseek 5-2
lf_fstat 5-2
lf_fstatvfs 5-2
lf_ftell 5-2
lf_getrlimit 5-2
lf_lseek 5-2
lf_lstat 5-2
lf_mmap 5-2

lf_pathconf 5-2
lf_setrlimit 5-2
lf_stat 5-2
lf_statvfs 5-2
lf_tell 5-2

libnsl

inet_addr 6-1
inet_netof 6-1
inet_ntoa 6-1
rpc_broadcast_exp 6-3

libsocket

accept 7-1
bind 7-3
connect 7-4
gethostbyname 7-6
gethostbyaddr 7-6
getpeername 7-7
getprotobyname 7-8
getprotobynumber 7-8
getprotoent 7-8
getservbyname 7-9
getservbyport 7-9
getsockname 7-10
inet_lnaof 7-11
inet_makeaddr 7-11
inet_network 7-11
listen 7-13
recv 7-14
recvfrom 7-14
recvmsg 7-14
send 7-16
sendto 7-16
sendmsg 7-16
getsockopt 7-18
setsockopt 7-18
shutdown 7-20
socket 7-21

libsys

__div64 8-1
__dtoll 8-2
__dtoull 8-3
__ftoll 8-4
__ftoull 8-5
__mul64 8-6
__rem64 8-7
__udiv64 8-8
__umul64 8-9
__urem64 8-10
_Q_lltoq 8-11

_Q_qtoll 8-12
_Q_qtoull 8-13
_Q_ulltoq 8-14
fgetgrent_r 8-15
fgetpwent_r 8-16
fork 8-18
getgrent_r 8-21
getlogin_r 8-22
getpwent_r 8-23
getgrgid_r 8-25
getgrnam_r 8-25
getpwnam_r 8-25
getpwuid_r 8-25
readdir_r 8-25
makecontext 8-27
swapcontext 8-27
sbrk 8-28
ttyname 8-29
ttyname_r 8-29

libthread

cond_broadcast 9-1
cond_destroy 9-1
cond_init 9-1
cond_timedwait 9-1
cond_signal 9-1
cond_wait 9-1
fork1 9-4
mutex_destroy 9-5
mutex_init 9-5
mutex_lock 9-5
mutex_trylock 9-5
mutex_unlock 9-5
rwlock_destroy 9-8
rwlock_init 9-8
rw_rdlock 9-8
rw_tryrdlock 9-8
rw_trywrlock 9-8
rw_unlock 9-8
rw_wrlock 9-8
sema_destroy 9-10
sema_init 9-10
sema_post 9-10
sema_trywait 9-10
sema_wait 9-10
thr_continue 9-12
thr_suspend 9-12
thr_create 9-13
thr_exit 9-15
thr_getconcurrency 9-16
thr_setconcurrency 9-16
thr_getprio 9-17
thr_setprio 9-17

thr_getspecific 9-18
thr_keycreate 9-18
thr_setspecific 9-18
thr_join 9-20
thr_kill 9-21
thr_min_stack 9-22
thr_self 9-23
thr_sigsetmask 9-24
thr_main 9-25
thr_yield 9-26
sigwait 9-27

Execution Environment

/dev/zero 10-1

INDEX

SPARC COMPLIANCE DEFINITION 2.3

Introduction

Introduction

This is the SPARC Compliance Definitions 2.3 Interface Semantics

This book is a companion volume to the SCD 2.3. It defines the interface semantics for those interfaces that are required by the SCD but are not specified in any other normative reference or whose semantics are different for SCD from that of a normative reference.

It is expected that many of these semantic definitions will eventually be adopted by the committees responsible for the SCD normative references. The definitions here will be deleted when and as they are added to the normative references.

SPARC COMPLIANCE DEFINITION 2.3

libaio

aiocancel

NAME

aiocancel - cancel an asynchronous operation

SYNOPSIS

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

int aiocancel (aio_result_t *resultp);
```

DESCRIPTION

aiocancel() cancels the asynchronous operation associated with the result buffer pointed to by *resultp*. It may not be possible to immediately cancel an operation which is in progress and in this case, aiocancel() will not wait to cancel it.

Upon successful completion, aiocancel() returns 0 and the requested operation is cancelled. The application will not receive the SIGIO completion signal for an asynchronous operation that is successfully cancelled.

RETURN VALUE

aiocancel() returns 0 on success, and -1 on failure and sets *errno* to indicate the error.

ERRORS

aiocancel() will fail if any of the following are true:

EACCES	The parameter *resultp* does not correspond to any outstanding asynchronous operation, although there is at least one currently outstanding.
EINVAL	There are not any outstanding requests to cancel.

aioread
aiowrite

NAME

aioread, aiowrite - asynchronous I/O operations.

SYNOPSIS

#include <sys/asynch.h>

int aioread (int fildes, char *bufp, size_t bufs, off_t offset, int whence, aio_result_t *resultp);

int aiowrite (int fildes, const char *bufp, size_t bufs, off_t offset, int whence, aio_result_t *resultp);

DESCRIPTION

aioread() initiates one asynchronous read(BA_OS) and returns control to the calling program. The read() continues concurrently with other activity of the process. An attempt is made to read *bufs* bytes of data from the object referenced by the descriptor *fildes* into the buffer pointed to by *bufp*.

aiowrite() initiates one asynchronous write(BA_OS) and returns control to the calling program. The write() continues concurrently with other activity of the process. An attempt is made to write *bufs* bytes of data from the buffer pointed to by *bufp* to the object referenced by the descriptor *fildes*.

On objects capable of seeking, the I/O operation starts at the position specified by *whence* and *offset*. These parameters have the same meaning as the corresponding parameters to the lseek (BA_OS) function. On objects not capable of seeking the I/O operation always start from the current position and the parameters *whence* and *offset* are ignored. The seek pointer for objects capable of seeking is not updated by aioread() or aiowrite(). Sequential asynchronous operations on these devices must be managed by the application using the *whence* and *offset* parameters.

The result of the asynchronous operation is stored in the structure pointed to by *resultp*:

```
int aio_return; /* return value of read() or write() */
int aio_errno; /* value of errno for read() or write() */
```

Upon completion of the operation both aio_return and aio_errno are set to reflect the result of the operation. AIO_INPROGRESS is not a value used by the system so the client may detect a change in state by initializing aio_return to this value.

The application supplied buffer *bufp* should not be referenced by the application until after the operation has completed. While the operation is in progress, this buffer is in use by the operating system.

Notification of the completion of an asynchronous I/O operation may be obtained synchronously through the aiowait function, or asynchronously by installing a signal handler for the SIGIO signal. Asynchronous notification is accomplished by sending the process a SIGIO signal. If a signal handler is not installed for the SIGIO signal, asynchronous notification is disabled. The delivery of this instance of the SIGIO signal is reliable in that a signal delivered while the handler is executing is not lost. If the client ensures that aiowait returns nothing (using a polling timeout) before returning from the signal handler, no asynchronous I/O notifications are lost. The aiowait function is the only way to dequeue an asynchronous notification. Note: SIGIO may have several meanings simultaneously: for example, that a descriptor generated SIGIO and an asynchronous operation completed. Further, issuing an asynchronous request successfully guarantees that space exists to queue the completion notification.

close(BA_OS), exit(BA_OS) and execve() (see exec(BA_OS)) will block until all pending asynchronous I/O operations can be canceled by the system.

It is an error to use the same result buffer in more than one outstanding request. These structures may only be reused after the system has completed the operation.

RETURN VALUE

aioread() and aiowrite() return 0 on success, and -1 on failure and set *errno* to indicate the error.

ERRORS

EAGAIN	The number of asynchronous requests that the system can handle at any one time has been exceeded
EBADF	*fildes* is not a valid file descriptor open for reading.
EINVAL	The parameter *resultp* is currently being used by an outstanding asynchronous request.
ENOMEM	Memory resources are unavailable to initiate request.

aiowait

NAME

aiowait - wait for completion of asynchronous I/O operation

SYNOPSIS

```
#include <sys/asynch.h>
#include <sys/time.h>
aio_result_t *aiowait (const struct timeval *timeout);
```

DESCRIPTION

aiowait() suspends the calling process until one of its outstanding asynchronous I/O operations completes. This provides a synchronous method of notification.

If *timeout* is a non-NULL pointer, it specifies a maximum interval to wait for the completion of an asynchronous I/O operation. If *timeout* is a NULL pointer, then aiowait() blocks indefinitely. To effect a poll, the *timeout* parameter should be non-zero, pointing to a zero-valued timeval structure.

The timeval structure is defined in <sys/time.h> and contains the following members:

```
long tv_sec; /* seconds */
long tv_usec; /* and microseconds */
```

The value of tv_usec is restricted to the range [0:1000000].

RETURN VALUE

On success, aiowait() returns a pointer to the result structure used when the completed asynchronous I/O operation was requested, or a NULL pointer if the time limit expires. On failure, it returns (aio_result_t *)-1 and sets *errno* to indicate the error.

ERRORS

EINTR	A signal was delivered before an asynchronous I/O operation completed.
EINVAL	There are no outstanding asynchronous I/O requests (or, all outstanding asynchronous EINVAL. There are no outstanding asynchronous I/O requests (or, all outstanding /O requests were cancelled via aiocancel.); or tv_usec is outside of the range [0:1000000].

NOTES

aiowait() is the only way to dequeue an asynchronous notification. It may be used either inside a SIGIO signal handler or in the main program. One SIGIO signal may represent several queued events.

SPARC COMPLIANCE DEFINITION 2.3

libc

_cleanup

NAME

_cleanup - flush all open files for writing

SYNOPSIS

```
void _cleanup();
```

DESCRIPTION

_cleanup is used to flush all open files for writing, functionally it is equivalent to fflush(NULL).

SEE ALSO

fflush(BA_OS)

addseverity

NAME

addseverity - build a list of severity levels for an application for use with fmtmsg

SYNOPSIS

int addseverity(int *value*, const char **string*)

DESCRIPTION

The function addseverity adds a new severity level of *value*. *value* must be greater than 4.

The function associates string with the level *value* so that *string* is produced with messages of that *value* yielded by fmtmsg().

If a severity of *value* already exists it is replaced by the new description.

If *string* is (char *)0 then the severity level is deleted.

DIAGNOSTICS

Under the following conditions, addseverity fail by returning -1, and setting errno to:

EINVAL	Using a value smaller or equal to 4.
EIVAL	If an attempt is made to delete a currently undefined severity level.

crypt
encrypt
setkey

NAME

crypt, setkey, encrypt - generate string encoding

SYNOPSIS

char *crypt (char **key*, char **salt*);

void setkey (char **key*);

void encrypt (char **block*, int *edflag*);

DESCRIPTION

The function crypt is a string-encoding function.

The argument *key* is a string to be encoded. The argument *salt* is a two-character string chosen from the set [a-zA-Z0-9./]; this string is used to perturb the encoding algorithm, after which the string that *key* points to is used as the key to repeatedly encode a constant string. The returned value points to the encoded string. The first two characters are the *salt* itself, the remaining characters shall not be identical to the original value of *key*.

The functions setkey and encrypt provide (rather primitive) access to the encoding algorithm. The argument to setkey is a 64-bit string represented by a character array of length 64 containing only the characters with numerical value 0 and 1. The string is divided into groups of 8 and the low-order bit in each group is ignored; this gives a 56-bit key. This is the key that may be used with the above mentioned algorithm to encode the string *block* with the function encrypt; the encryption algorithm provided by the system may not actually use *key*.

The argument *block* to encrypt is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the encoding algorithm using the key set by setkey.

If the argument *edflag* is zero, the string *block* is encoded. If the *edflag* is non-zero and the implementation supports decryption then the string *block* is decoded. If the *edflag* is non-zero and the implementation does not support decryption then errno is set to ENOSYS.

DIAGNOSTICS

Under the following conditions, these functions fail, and set errno to:

ENOSYS	encrypt was called with a non-zero value for *edflag* on a system that does not support decryption.

USAGE

The return value of the function crypt points to static data that are overwritten by each call. A portable application shall not depend on portability of encrypted data, nor assume that decryption is supported on all SCD conforming platforms. Also, portable applications must set errno to zero before calling any of the functions since there are no function return values for setkey or encrypt.

RATIONALE

Encryption capability is often needed by an application that wants to provide some of its own license protection. The application needs to be able to depend on the system to provide an encryption service to do this even if the system does not provide a mechanism for decryption.

This standard does not require any particular underlying encryption algorithm, but only requires that the crypt function return a value that is not identical to the original. This leaves it to the system vendors to chose whatever algorithm they find to be appropriate, and alleviates any requirement for a system vendor to choose one that has export restrictions.

setlabel

NAME

setlabel - define the label for standard format messages.

SYNOPSIS

#include <pfmt.h>

int setlabel (const char *label);

DESCRIPTION

The routine setlabel() defines the label for messages produced in standard format.

label is a character string no more than 25 characters in length.

No label is defined before setlabel() is called. A NULL pointer or an empty string passed as argument will reset the definition of the label.

RETURN VALUE

setlabel() returns 0 in case of success, non-zero otherwise.

USAGE

The label should be set once at the beginning of a utility and remain constant.

SEE ALSO

getopt(BA_LIB)

sysinfo

NAME

sysinfo - get system information strings

SYNOPSIS

#include <sys/systeminfo.h>

long sysinfo (int *command*, char **buf*, long *count*);

DESCRIPTION

sysinfo copies information relating to the UNIX system on which the process is executing into the buffer pointed to by *buf*. *count* is the size of the buffer.

The *command*s available are:

SI_SYSNAME — Copy into the array pointed to by *buf* the string that would be returned by uname [see uname(BA_OS)] in the *sysname* field. This is the name of the implementation of the operating system, for example, UNIX_SV.

SI_HOSTNAME — Copy into the array pointed to by *buf* a string that names the present host machine. This is the string that would be returned by uname in the *nodename* field. This hostname or nodename is often the name the machine is known by locally.

The *hostname* is the name of this machine as a node in some network; different networks may have different names for the node, but presenting the nodename to the appropriate network Directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified.

Internet host names may be up to 256 bytes in length (plus the terminating null).

SI_RELEASE — Copy into the array pointed to by *buf* the string that would be returned by uname in the *release* field. Typical values might be 4.2, 4.0, 3.2.

SI_VERSION — Copy into the array pointed to by *buf* the string that would be returned by uname in the *version* field. The syntax and semantics of this string are defined by the system provider.

SI_MACHINE — Copy into the array pointed to by *buf* the string that would be returned by uname in the *machine* field.

SI_ARCHITECTURE — Copy into the array pointed to by *buf* a string describing the instruction set architecture of the current system, for example, sparc. These names may not match predefined names in the C language compilation system.

SI_HW_PROVIDER — Copies the name of the hardware manufacturer into the array pointed to by *buf*.

SI_HW_SERIAL — Copy into the array pointed to by buf a string which is the ASCII representation of the hardware-specific serial number of the physical machine on which the system call is executed. Note that this may be implemented in Read-Only Memory, via software constants set when building the operating system, or by other means, and may contain non-numeric characters. It is anticipated that manufacturers will not issue the

same "serial number" to more than one physical machine. The pair of strings returned by SI_HW_PROVIDER and SI_HW_SERIAL is likely to be unique across all vendors' System V implementations.

SI_SRPC_DOMAIN — Copies the Secure Remote Procedure Call domain name into the array pointed to by *buf*.

DIAGNOSTICS

Upon successful completion, the value returned indicates the buffer size in bytes required to hold the complete value and the terminating null character. If this value is no greater than the value passed in *count*, the entire string was copied; if this value is greater than *count*, the string copied into *buf* has been truncated to *count*-1 bytes plus a terminating null character.

Otherwise, a value of -1 is returned and errno is set to indicate the error.

RATIONALE

The commands included for sysinfo in SCD 2.3 are values that have been determined to be uniformly implemented on systems that have been presented for testing at the SCD 2.1 level. Also, the commands that require that the effective user-id be superuser are omitted.

___errno
asctime_r
ctime_r
flockfile
funlockfile
getc_unlocked
getchar_unlocked
gmtime_r
localtime_r
putc_unlocked
putchar_unlocked
rand_r
strtok_r

NAME

___errno, asctime_r, ctime_r, gmtime_r, localtime_r, flockfile, funlockfile, getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked, rand_r, strtok_r - Support routines for multithreading added to libsys and libc.

SYNOPSIS

```
#include <errno.h>
int *___errno(void);

#include <time.h>
char *asctime_r (const struct tm *tm, char *buf, int buflen);
char *ctime_r (const time_t *clock, char *buf, int buflen);
struct tm *gmtime_r (const time_t *clock, struct tm *res);
struct tm *localtime_r (const time_t *clock, struct tm *res);

#include <stdio.h>
void flockfile (FILE *stream);
void funlockfile (FILE *stream);
int getc_unlocked (FILE *stream);
int getchar_unlocked (void);
int putc_unlocked (int c, FILE *stream);
int putchar_unlocked (int c);
#include <stdlib.h>
int rand_r(unsigned int *seed);

#include <string.h>
char *strtok_r(char *s1, const char *s2, char **lasts);
```

DESCRIPTION and RETURN VALUES

These functions are "reentrant" versions of existing functions. They exist as the definition of the existing functions prevents the transparent implementation of multithreading, usually because of the use of a static storage area. In general, these functions are exactly equivalent to the non-reentrant versions in terms of function and results, but differ in providing for the implementation the necessary storage for completion of the function.

___errno returns the address of errno for the "calling thread". The location labelled errno provides the storage for the "main thread" in the process. In all references to "errno" which follow, it is implied that the storage used will be that for the thread invoking the operation.

asctime_r is equivalent to asctime, however the caller must supply a buffer *buf* in which to store the resulting string. *buflen* indicates the length which must be at least 26 bytes. The return value of asctime_r is a pointer to *buf* on success. On failure, NULL is returned and errno is set. If the operation fails because *buflen* is not large enough, errno will be set to ERANGE.

ctime_r is equivalent to ctime, however the caller must supply a buffer *buf* in which to store the resulting string. *buflen* indicates the length of *buf* which must be at least 26 bytes. If the operation fails because *buflen* is not long enough, ctime_r will return NULL and errno will be set to ERANGE.

flockfile and funlockfile are new functions which allow the caller to gain or release exclusive access, respectively, to *stream*. They can be used in conjunction with a sequence of calls to *getc et al*. so as to avoid the overhead of locking the stream on each access to the buffers managed by *stream*.

getc_unlocked, getchar_unlocked, putc_unlocked, and putchar_unlocked implement an unlocked access to *stream* (or, for getchar the standard input and for putchar the standard output).

gmtime_r is equivalent to gmtime but the caller must supply a result buffer *res*, which is the return value of the function.

localtime_r is equivalent to localtime but the caller must supply a result buffer *res*, which is the return value of the function.

rand_r is equivalent to rand except that a pointer to a seed *seed* must be supplied by the caller

strtok_r is equivalent to strtok except that a pointer to a string placeholder *lasts* must be supplied by the caller. The *lasts* pointer is to keep track of the next substring in which to search for the next token.

NOTES

asctime_r and ctime_r are designated as EXPERMIMENTAL since they have interfaces which are different from the ones in POSIX 1003.1c. The interfaces of these functions are in POSIX as following:

```
char  *asctime_r  (     const struct tm   *tm,
                        char              *buf);

char  *ctime_r    (     const time_t      *clock,
                        char              *buf);
```

SPARC COMPLIANCE DEFINITION 2.3

libdl

Introduction

The following terms are used in this specification:

For a *program* to *reference* a symbol means for the program to use the storage value associated with that symbol. To reference a data symbol means (a) to retrieve the value stored in the location associated with that symbol, or (b) to store a value into the location associated with that symbol. To reference a function symbol means to (a) use the value directly by calling that function, or (b) to obtain its value via a call to dlsym, presumably in order to call the function later.

For a *program* to *contain a reference* to a symbol means that the program has been constructed in such a way that it will reference a symbol that is not defined within it. In the C language, this is done by declaring a data or function to have the extern attribute. The *reference* that the program contains is an indication to the linker and loader of what the name of the symbol is, and the fact that it will be found in some other program. For details on how this is implemented in a SPARC executable file, see the *System V Application Binary Interface* and the *System V Application Binary Interface, SPARC Processor Supplement.*

Two kinds of objects are mentioned in these specifications. A *data object* is the storage location associated with a symbol in an application program. A *shared object* is (a) a file on disk that was created by linking a program as a shared object, or (b) such a file that has been loaded into memory and prepared for execution. When the word "object" is used without qualification in this specification, it means shared object, and usually the shared object in memory.

For an *object* to reference another *object* means that the first object has been link-edited with the second object in such a way as to create DT_NEEDED entries that cause the second object to be loaded automatically with the first object. (See Chapter 5 of the SCD 2.3 document.)

dlclose

NAME

dlclose - close a shared object

SYNOPSIS

#include <dlfcn.h>

int dlclose (void **handle*);

DESCRIPTION

The function dlclose disassociates from the current process a shared object previously opened by dlopen.

handle is a value that was returned from a previous call to dlopen. It designates the shared object whose *pathname* was specified in that previous call to dlopen.

Once an object has been dissasociated from the process using dlclose, its symbols and those of any objects that were loaded automatically as a result of opening the object designated by *handle* are no longer available to dlsym via *handle*.

In order for dlclose to dissasociate an object from a process, there must have been exactly one dlclose executed for each dlopen that was executed. Thus if a dlopen was executed once for a pathname, dlclose would have to be executed once with the handle that was returned for *pathname*. If a dlopen were executed twice for the same *pathname*, the disassociation would occur only after the second dlclose.

A successful invocation of dlclose does not guarantee that the objects associated with *handle* will actually be removed from the address space of the process, even if the object has been disassociated from the process and its symbols are no longer available through *handle*. Objects loaded by one invocation of dlopen may also be loaded by another invocation of dlopen. The same object may also be opened multiple times. An object may be removed from the address space by the system only after all references to that object through an explicit dlopen invocation have been closed and all other objects that reference that object have also been closed. Even then, however, it is unspecified in this standard whether the object will actually be removed from the address space.

When the system removes an object from the process address space, the object's termination function is executed. The termination function for each object is specified by the DT_FINI entry in that object's dynamic section. The exact timing of the execution of termination function relative to the timing of the dlclose that release the object is unspecified in this standard.

An SCD-conforming application will not have any processing dependencies upon the system's removal or non-removal of an object from the process address space following dlclose.

DIAGNOSTICS

If the referenced object was successfully closed, dlclose returns 0. If the object could not be closed, or if *handle* does not refer to an open object, dlclose returns a non-0 value. More detailed diagnostic information will be available through dlerror.

NOTES

The following notes are a consequence of that fact that this standard does not specify whether an object ever is actually removed from a process address space:

Once a program has executed a sequence of dlclose operations that would permit the system to remove an object from the process address space, the result of the program's executing any reference to symbols defined in that object are unspecified in this standard.

Once a program has executed a sequence of dlclose operations that would permit the system to remove an object from the process address space, if the program executes another dlopen for that object, it is unspecified in this standard whether the object is actually loaded again and whether the object's data will be in its initial state.

dlerror

NAME

dlerror - get diagnostic information

SYNOPSIS

```
#include <dlfcn.h>
char *dlerror (void);
```

DESCRIPTION

The function dlerror returns a null-terminated character string (with no trailing newline) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of dlerror, dlerror returns NULL. Thus, invoking dlerror a second time, immediately following a prior invocation, will result in NULL being returned.

NOTES

The messages returned by dlerror may reside in a static buffer that is overwritten on each call to dlerror. Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message.

dlopen

NAME

dlopen - open a shared object

SYNOPSIS

#include <dlfcn.h>

void *dlopen (char **pathname*, int *mode*);

DESCRIPTION

The function dlopen is one of a family of routines that give the user direct access to the dynamic linking facilities.

The function dlopen makes a shared object available to a running process. dlopen returns to the process a *handle* the process must use to identify the object on subsequent calls to dlsym and dlclose. This value must not be interpreted in any way by the process. (See Rationale)

pathname is the path name of the object to be opened; it may be an absolute path or relative to the current directory. If the value of *pathname* is 0, dlopen will make the symbols contained in the original a.out, and all of the objects that were loaded at program startup with the a.out, available through dlsym.

If the value of *pathname* is not zero, and no file specified by *pathname* has already been loaded into the address space, the file specified by *pathname* will be loaded. If the file specified by pathname contains DT_NEEDED entries for other shared objects, those objects will automatically be loaded by dlopen.

Objects whose names resolve to the same absolute or relative path name may be opened any number of times either using dlopen or automatically as a result of executing dlopen for an object that uses them. However, the object referenced is loaded only once into the address space of the current process. This means that the object only takes up space once; there is only one copy of its static data; and the static data are initialized only once, when the initial load takes place.

When a shared object is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY — Under this *mode*, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is referenced for the first time by the executing program. This *mode* should result in better performance, since a process may not reference all of the functions in any given shared object.

RTLD_NOW — Under this *mode*, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort, if relocations are performed for functions that are never referenced, but is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

The *mode* parameter only takes effect when an object is initially loaded. If RTLD_LAZY is specified in the first dlopen for an object, and RTLD_NOW is specified for the second dlopen of the same object, the second dlopen will not cause any relocations to be performed.

The *mode* parameter is required, and always overrides the value of the LD_BIND_NOW

environment variable.

When the system loads an object for the first time, the object's initialization function is executed. The initialization function for each object is specified by the DT_INIT entry in that object's dynamic section. If multiple objects are loaded as a result of dlopen, the order initialization functions are called is unspecified.

Objects loaded by a single invocation of dlopen may import symbols from one another or from any object loaded automatically with a.out during program startup, but objects loaded by one dlopen invocation may not directly reference symbols from objects loaded by a different dlopen invocation. Those symbols may, however, be referenced indirectly using dlsym.

RATIONALE

The functions dlopen and dlclose may not work in a manner consistent with the way the functions open and close work. For example, if the same file is opened twice, the open function will return unique file descriptors for each open operation. Using dlopen to open the same file multiple times may return the same file handle every time. The result is that if the first file handle for a dlopen call is used more than once as a parameter to dlclose, there may be unexpected side effects.

DIAGNOSTICS

If the file specified by *pathname* cannot be found, cannot be opened for reading, is not a shared object, or if an error occurs during the process of loading the file specified by *pathname* or relocating its symbolic references, dlopen will return NULL. More detailed diagnostic information will be available through dlerror.

NOTES

The same object referenced by different path names may be loaded multiple times. For example, given the object /usr/home/me/mylibs/mylib.so, and assuming the current working directory is /usr/home/me/workdir,

...

void *handle1;

void *handle2;

handle1 = dlopen ("/mylibs/mylib.so", RTLD_LAZY);

handle2 = dlopen ("/usr/home/me/mylibs/mylib.so", RTLD_LAZY);

...

results in mylibs.so being loaded twice for the current process. On the other hand, given the same object and current working directory, if LD_LIBRARY_PATH=/usr/home/me/mylibs, then

...

void *handle1;

void *handle2;

handle1 = dlopen ("mylib.so", RTLD_LAZY);

handle2 = dlopen ("/usr/home/me/mylibs/mylib.so", RTLD_LAZY);

...

results in mylibs.so being loaded only once.

Users who wish to gain access to the symbol table of the a.out itself using dlopen(0, *mode*) should be aware that some symbols defined in the a.out may not be available to the dynamic linker. The symbol table created by ld for use by the dynamic linker might contain only a subset of the symbols originally defined in the a.out: specifically, those referenced by the shared objects with which the a.out is linked.

dlsym

NAME

dlsym - get the address of a symbol in a shared object

SYNOPSIS

#include <dlfcn.h>

void *dlsym (void **handle*, char **name*);

DESCRIPTION

The function dlsym allows a process to obtain the address of a symbol defined within a shared object previously opened by dlopen.

handle is a value returned by a call to dlopen; the corresponding shared object must not have been disassociated from the executing process using dlclose. *name* is the symbol's name as a character string.

dlsym searches for the named symbol in the shared object designated by *handle* and in all shared objects loaded automatically as a result of loading the object referenced by *handle* [see dlopen(3X)].

EXAMPLES

The following example shows how one can use dlopen and dlsym to access either function or data objects. For simplicity, error checking has been omitted.

```
void *handle;
int i, *iptr;
int (*fptr) (int);
/* open the needed object */
handle = dlopen ("/usr/mydir/libx.so", RTLD_LAZY);
/* find address of function and data objects */
fptr = (int (*)(int)) dlsym (handle, "some_function");
iptr = (int *) dlsym (handle, "int_object");
/* invoke function, passing value of integer as a parameter */
i = (*fptr) (*iptr);
```

DIAGNOSTICS

If *handle* does not refer to a valid object opened by dlopen, or if the named symbol cannot be found within any of the objects associated with *handle*, dlsym will return NULL. More detailed diagnostic information will be available through dlerror.

SPARC COMPLIANCE DEFINITION 2.3

libIf

Introduction

At the time of the writing of this document, members of the Unix community are meeting in an attempt to agree on a standard implementation of 64-bit files on a 32-bit system. Members of SPARC International are taking part in these discussions. If agreement on these interfaces is reached by the industry, the interfaces for large file support contained in this section will be modified, if necessary, to match the agreed to interfaces.

lf_fcntl
lf_fpathconf
lf_fseek
lf_fstat
lf_fstatvfs
lf_ftell
lf_getrlimit
lf_lseek
lf_lstat
lf_mmap
lf_pathconf
lf_setrlimit
lf_stat
lf_statvfs
lf_tell

NAME

lf_fcntl,lf_fpathconf,lf_fseek,lf_fstat, lf_fstatvfs, lf_ftell, lf_getrlimit, lf_lseek, lf_lstat, lf_mmap, lf_pathconf, lf_setrlimit, lf_stat, lf_statvfs, lf_tell - Large support functions.

SYNOPSIS

```
#include <fcntl.h>
int lf_fcntl (int fildes, int cmd,... /* arg */);
int64_t lf_fpathconf (int fildes, int name);
int lf_fseek (FILE *stream, lf_off_t offset, int ptrname);
#include <sys/stat.h>
int lf_fstat (int fildes, struct lf_stat *buf);
#include <sys/fstatvfs.h>
int lf_fstatvfs (int fildes, struct lf_statvfs *buf);
#include <sys/types.h>
lf_off_t lf_ftell (FILE *stream);
#include <sys/time.h>
#include <sys/resource.h>
int lf_getrlimit (int resource, struct lf_rlimit *rlp);
lf_off_t lf_lseek (int fildes, lf_off_t offset, int whence);
#include <sys/stat.h>
int lf_lstat (const char *path, struct lf_stat *buf);
#include <sys/mman.h>
caddr_t lf_mmap (caddr_t addr, size_t len, int prot, int flags, int fd, lf_off_t off);
#include <sys/types.h>
int64_t lf_pathconf (char *path, int name);
#include <sys/time.h>
#include <sys/resource.h>
```

int lf_setrlimit (int *resource*, struct lf_rlimit **rlp*);

#include <sys/stat.h>

int lf_stat (const char **path*, struct lf_stat **buf*);

#include <sys/statvfs.h>

int lf_statvfs (const char **path*, struct lf_statvfs **buf*);

#include <sys/types.h>

lf_off_t lf_tell (char **path*);

DESCRIPTION

Two new data types, int64_t and uint64_t are needed to define 64-bit signed and unsigned values. They must be defined by compliant systems in <sys/types.h>. <sys/types.h> must also contain

typedef int64_t lf_off_t;

The layout of 64-bit integers and how they are passed as parameters are specified in Table 3-1 of the *SPARC Architecture Manual, Version 8*.

lf_fcntl: The F_SETLK, F_SETLKW, F_RSETLK, and F_RSETLKW subcommands can lock a segment containing a byte at an offset greater than INT_MAX. The F_FREESP command can set the file size to values over INT_MAX. The F_GETLK, F_RGETLK, and F_FREESP subcommands set errno to EOVERFLOW if the filesystem protocol cannot satisfy the request due to an interface restriction, such as if the file is being accessed through a remote file system not supporting 64-bit file offsets.

lf_fpathconf and lf_pathconf: In addition to the _PC_* commands defined in <unistd.h> for fpathconf and pathconf, lf_fpathconf and lf_pathconf also recognize the _PC_MAX_FILE_SIZE command. When passed _PC_MAX_FILE_SIZE, lf_pathconf and lf_pathconf will return the largest file size supported on the given file system, without regard to disk space currently available. The value of _PC_MAX_FILE_SIZE must be defined in <unistd.h> to be 10.

lf_fstat, lf_lstat, and lf_stat: lf_fstat, lf_lstat, and lf_stat set errno to EOVERFLOW if the filesystem protocol cannot satisfy the request due to an interface restriction, such as if the file is being accessed through a remote file system not supporting 64-bit file offsets.

lf_fstatvfs and lf_statvfs: lf_fstatvfs and lf_statvfs set errno to EOVERFLOW if the filesystem protocol cannot satisfy the request due to an interface restriction, such as if the file is being accessed through a remote file system not supporting 64-bit file offsets.

lf_getrlimit and lf_setrlimit: lf_getrlimit and lf_setrlimit never set errno to EOVERFLOW, such as if the file is being accessed through a remote file system not supporting 64-bit file offsets.

DIAGNOSTICS

When lseek() attempts to position or query the file pointer beyond the point addressable in 31 bits, it will fail with errno set to EOVERFLOW.

When stat(), fstat(), or lstat() is invoked on a file whose size cannot be represented in 31 bits, it will fail with errno set to EOVERFLOW.

SEE ALSO

fcntl(BA_OS), fpathconf(BA_OS), getrlimit(BA_OS), lseek(BA_OS), mmap(KE_OS), stat(BA_OS), statvfs(BA_OS), limits(BA_ENV).

SPARC COMPLIANCE DEFINITION 2.3

libnsl

inet_addr
inet_netof
inet_ntoa

NAME

inet_addr, inet_netof, inet_ntoa - Internet address manipulation

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

unsigned long inet_addr (char *_cp_);

int inet_netof (struct in_addr _in_);

char *inet_ntoa (struct in_addr _in_);

DESCRIPTION

The inet_addr routines interpret a character string, *cp*, representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routines inet_netof breaks apart an Internet host address, *in*, returning the network number and local network address part, respectively.

The routine inet_ntoa returns a pointer to a string in the base 256 notation "d.d.d.d" described below.

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 format 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.

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 (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

RETURN VALUES

The value -1 is returned by inet_addr for malformed requests.

The routines inet_netof break apart Internet host addresses, returning the network number and local network address part, respectively.

The routine inet_ntoa returns a pointer to a string in the base 256 notation "d.d.d.d" described below.

rpc_broadcast_exp

NAME

rpc_broadcast_exp - broadcast a call message specifying timeout

SYNOPSIS

```
#include <rpc/rpc.h>

enum clnt_stat rpc_broadcast_exp (const u_long prognum,
 const u_long versnum, const u_long 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,
 char *nettype);
```

DESCRIPTION

This function is like rpc_broadcast(), except that the initial timeout, *inittime*, and the maximum timeout, *waittime*, are specified in milliseconds.

inittime is the initial time that rpc_broadcast_exp() waits before re-sending the request. After the first re-send, the re-transmission interval increases exponentially until it exceeds *waittime*.

SPARC COMPLIANCE DEFINITION 2.3

libsocket

accept

NAME

accept - accept a connection on a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

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

DESCRIPTION

The argument *s* is a socket that has been created with socket() and bound to an address with bind(), and that is listening for connections after a call to listen(). accept extracts the first connection on the queue of pending connections, creates a new socket with the properties of *s*, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, accept blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, accept returns an error as described below. accept uses the netconfig(4) file to determine the STREAMS device file name associated with *s*. This is the device on which the connect indication will be accepted. The accepted socket, *ns*, is used to read and write data to and from the socket that connected to *ns*; it is not used to accept more connections. The original socket (s) remains open for accepting further connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity as it is known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication occurs.

addrlen is a value-result parameter. Initially, it contains the amount of space pointed to by *addr*; on return it contains the length in bytes of the address returned.

accept is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to poll(BA_OS) a socket for the purpose of an accept by polling it for a read. However, this will only indicate when a connect indication is pending; it is still necessary to call accept.

RETURN VALUES

accept returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

accept will fail if:

EBADF	The descriptor is invalid.
ENODEV	The protocol family and type corresponding to *s* could not be found in the netconfig file.
ENOMEM	There was insufficient user memory available to complete the operation.
ENOSR	There were insufficient STREAMS resources available to complete the operation.

ENOTSOCK	The descriptor does not reference a socket.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EPROTO	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.
EWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

bind

NAME

bind - bind a name to a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int bind (int *s*, struct sockaddr **name*, int *namelen*);

DESCRIPTION

bind assigns a name to an unnamed socket, *s*. When a socket is created with socket(), it exists in a name space (address family) but has no *name* assigned. bind requests that the name pointed to by *name* be assigned to the socket. *namelen* specifies the size of *name*.

RETURN VALUES

If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global errno.

ERRORS

The bind call will fail if:

EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available on the local machine.
EBADF	*s* is not a valid descriptor.
EINVAL	*namelen* is not the size of a valid address for the specified address family.
EINVAL	The socket is already bound to an address.
ENOSR	There were insufficient STREAMS resources for the operation to complete.
ENOTSOCK	*s* is a descriptor for a file, not a socket.

connect

NAME

connect - initiate a connection on a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int connect(int *s*, struct sockaddr **name*, int *namelen*);

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, connect specifies the peer with which the socket is to be associated; this address is the address to which datagrams are to be sent if a receiver is not explicitly designated; it is the only address from which datagrams are to be received. If the socket *s* is of type SOCK_STREAM, connect attempts to make a connection to another socket. The other socket is specified by *name*. *name* is an address in the communication space of the socket. *namelen* specifies the size of data structure pointed to by *name*. Each communication space interprets the *name* parameter in its own way. If *s* is not bound, then it will be bound to an address selected by the underlying transport provider. 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 a null address.

RETURN VALUES

If the connection or binding succeeds, then 0 is returned. Otherwise a -1 is returned and sets errno to indicate the error.

ERRORS

The call fails if:

EADDRINUSE	The address is already in use.
EADDRNOTAVAIL	The specified address is not available on the remote machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EALREADY	The socket is non-blocking and a previous connection attempt has not yet been completed.
EBADF	*s* is not a valid descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should close(BA_OS) the socket descriptor, and issue another socket() call to obtain a new descriptor before attempting another connect call.
EINPROGRESS	The socket is non-blocking and the connection cannot be completed immediately. It is possible to poll(BA_OS) for completion by polling the socket for writing. However, this is only possible if the socket STREAMS module is the topmost module on the protocol stack with a write service procedure. This will be the normal case.
EINTR	The connection attempt was interrupted before any data arrived by the

	delivery of a signal.
EINVAL	*namelen* is not the size of a valid address for the specified address family.
EISCONN	The socket is already connected.
ENETUNREACH	The network is not reachable from this host.
ENOSR	There were insufficient STREAMS resources available to complete the operation.

gethostbyname
gethostbyaddr

NAME

gethostbyname, gethostbyaddr - get network host entry

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

struct hostent *gethostbyname (char **name*);

struct hostent *gethostbyaddr (struct in_addr **addr*,
const sizeof (struct in_addr), const int AF_INET);

DESCRIPTION

gethostbyaddr, and gethostbyname each return a host entry.

The entry comes from the system's hosts database. The lookup order is unspecified.

gethostbyname searches for a host entry with a given hostname.

gethostbyaddr searches for a host entry with a given hostaddress.

The internal representation of a host entry is a structure defined in <netdb.h> with the following members:

char *h_name;

char **h_aliases;

int h_addrtype;

int h_length;

char **h_addr_list;

Host addresses are supplied in network byte order.

RETURN VALUES

gethostbyname and gethostbyaddr return a pointer to a struct hostent if they successfully locate the requested entry; otherwise they return NULL, and set an integer h_errno to indicate one of these errors: HOST_NOT_FOUND, TRY_AGAIN, NO_RECOVERY, NO_DATA and NO_ADDRESS (see /usr/include/netdb.h).

NOTES

All information is contained in a static area so it must be copied if it is to be saved.

getpeername

NAME

getpeername - get name of connected peer

SYNOPSIS

int getpeername(int *s*, struct sockaddr **name*, int **namelen*);

DESCRIPTION

getpeername returns the name of the peer connected to socket *s*. The int pointed to by the *namelen* parameter 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

If successful, getpeername returns 0; otherwise it returns -1 and sets errno to indicate the error.

ERRORS

The call succeeds unless:

EBADF	The argument *s* is not a valid descriptor.
ENOMEM	There was insufficient user memory for the operation to complete.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTCONN	The socket is not connected.
ENOTSOCK	The argument *s* is not a socket.

getprotobyname
getprotobynumber
getprotoent

NAME

getprotobyname, getprotobynumber, getprotoent - get protocol entry

SYNOPSIS

#include <netdb.h>

struct protoent *getprotobyname (char **name*);

struct protoent *getprotobynumber (int *proto*);

struct protoent *getprotoent (void);

DESCRIPTION

getprotoent, getprotobyname, and getprotobynumber each return a protocol entry. The entry may comes from the system's protocols database. *name* is a pointer to one of the strings "tcp", "udp", or "icmp". *proto* is one of the values 6 (tcp), 17 (udp), 0 (ip), or 1 (icmp).

getprotoent enumerates protocol entries: successive calls to getprotoent will return either successive protocol entries or NULL. Enumeration may not be supported by some sources.

The internal representation of a protocol entry is a protoent structure defined in <netdb.h> with the following members:

char *p_name;

char **p_aliases;

int p_proto;

RETURN VALUES

getprotobyname and getprotobynumber return a pointer to a struct protoent if they successfully locate the requested entry; otherwise they return NULL.

getprotoent returns a pointer to a struct protoent if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

NOTES

All information is contained in a static area so it must be copied if it is to be saved.

Use of getprotoent is deprecated.

getservbyname
getservbyport

NAME

getservbyname, getservbyport - get service entry

SYNOPSIS

#include <netdb.h>

struct servent *getservbyname (char **name*, char **proto*);

struct servent *getservbyport (int *port*, char **proto*);

DESCRIPTION

getservbyname, and getservbyport each return a service entry. The entry come from the system's services database. getservbyname searches for a service entry with a given service name.

getservbyport searches for a service entry with a given port number and, if the protocol name is non-NULL, the protocol.

name is a pointer to one of the strings "tcp" or "udp". *port* is the number of a well-known port.

The internal representation of a service entry is a struct servent defined in <netdb.h> with the following members:

char *s_name;

char **s_aliases;

int s_port;

char *s_proto;

RETURN VALUES

getservbyname and getservbyport return a pointer to a struct servent if they successfully locate the requested entry; otherwise they return NULL.

NOTES

All information is contained in a static area, so it must be copied if it is to be saved.

getsockname

NAME

getsockname - get socket name

SYNOPSIS

#include <sys/types.h>

#include <sys/sockets.h>

int getsockname(int *s*, struct sockaddr **name*, int **namelen*);

DESCRIPTION

getsockname returns the current name for socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size in bytes of the *name* returned.

RETURN VALUES

If successful, getsockname returns 0; otherwise it returns -1 and sets errno to indicate the error.

ERRORS

The call succeeds unless:

EBADF	The argument *s* is not a valid file descriptor.
ENOMEM	There was insufficient memory available for the operation to complete.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTSOCK	The argument *s* is not a socket.

inet_lnaof
inet_makeaddr
inet_network

NAME

inet_network, inet_makeaddr, inet_lnaof - Internet address manipulation

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

int inet_network (char *_cp_);

struct in_addr inet_makeaddr (int _net_, int _lna_);

int inet_lnaof (struct in_addr _in_);

DESCRIPTION

The inet_network routine interprets a character string, *cp*, representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine inet_makeaddr takes an Internet network number, *net*, and a local network address, *lna*, and constructs an Internet address from it. The routine inet_lnaof break apart an Internet host address, *in*, 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 format 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.

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 (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

RETURN VALUES

The value -1 is returned by inet_network for malformed requests.

The routine inet_lnaof break apart Internet host addresses, returning the network number and local network address part, respectively.

listen

NAME

listen - listen for connections on a socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/sockets.h>

int listen(int s, int backlog);
```

DESCRIPTION

To accept connections, a socket, *s*, is first created with socket(), a *backlog* for incoming connections is specified with listen and then the connections are accepted with accept(). The listen call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, the client will receive an error with an indication of ECONNREFUSED.

RETURN VALUES

A 0 return value indicates success; -1 indicates an error.

ERRORS

The call fails if:

EBADF	The argument *s* is not a valid file descriptor.
ENOTSOCK	The argument *s* is not a socket.
EOPNOTSUPP	The socket is not of a type that supports the operation listen.

NOTES

There is currently no backlog limit.

recv
recvfrom
recvmsg

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/uio.h>

int recv (int *s*, char **buf*, int *len*, int *flags*);

int recvfrom (int *s*, char **buf*, int *len*, int *flags*, struct sockaddr **from*, int **fromlen*);

int recvmsg (int *s*, struct msghdr **msg*, int *flags*);

DESCRIPTION

recv, recvfrom, and recvmsg are used to receive messages from another socket. recv may be used only on a connected socket (see connect()), while recvfrom and recvmsg may be used to receive data on a socket whether it is in a connected state or not. *s* is a socket created with socket(). *buf* is a pointer to the buffer to receive the data and *len* is its size in bytes.

If *from* is not a NULL pointer, the source address of the message is filled in. *fromlen* is a value-result parameter, 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 length of the message is returned. 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()).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see fcntl(BA_OS)) in which case -1 is returned with the external variable errno set to EWOULDBLOCK.

The poll call may be used to determine when more data arrives.

The flags parameter is formed by ORing one or more of the following:

MSG_OOB	Read any out-of-band data present on the socket rather than the regular in-band data.
MSG_PEEK	Peek at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

The recvmsg call uses a struct msghdr, *msg*, to minimize the number of directly supplied parameters. This structure is defined in <sys/socket.h> and includes the following members:

```
caddr_t         msg_name;           /* optional address */
int             msg_namelen;        /* size of address */
struct iovec    *msg_iov;           /* scatter/gather array */
int             msg_iovlen;         /* # elements in msg_iov */
caddr_t         msg_accrights;      /* access rights sent/received */
int             msg_accrightslen;
```

Here msg_name and msg_namelen specify the destination 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 describe the scatter-gather locations, as described in read(BA_OS). A buffer to receive any access rights sent along with the message is specified in msg_accrights, which has length msg_accrightslen.

RETURN VALUES

These calls return the number of bytes received, or -1 if an error occurred.

ERRORS

The calls fail if:

EBADF	*s* is an invalid file descriptor.
EINTR	The operation was interrupted by delivery of a signal before any data was available to be received.
ENOMEM	There was insufficient user memory available for the operation to complete.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTSOCK	*s* is not a socket.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

send
sendto
sendmsg

NAME

send, sendto, sendmsg - send a message from a socket

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int send (int *s*, char **buf*, int *len*, int *flags*);

int sendto (int *s*, char **buf*, int *len*, int *flags*, struct sockaddr **to*, int *tolen*);

int sendmsg (int *s*, struct msghdr **msg*, int *flags*);

DESCRIPTION

send, sendto, and sendmsg are used to transmit a message to another transport end-point. send may be used only when the socket is in a connected state, while sendto and sendmsg may be used at any time. *s* is a socket created with socket(). *buf* points to a buffer containing the data to be sent. *len* is number of bytes to be sent.

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, then the error EMSGSIZE is returned, and the message is not transmitted.

A return value of -1 indicates locally detected errors only. It does not implicitly mean the message was not delivered.

If the socket does not have enough buffer space available to hold the message being sent, send blocks, unless the socket has been placed in non-blocking I/O mode (see fcntl(BA_OS)). The poll call may be used to determine when it is possible to send more data.

The flags parameter is formed from the bit-wise OR of zero or more of the following:

MSG_OOB	Send out-of-band data on sockets that support this notion. The underlying protocol must also support out-of-band data. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. It is used only by diagnostic or routing programs.

See recv() for a description of the msghdr structure.

RETURN VALUES

These calls return the number of bytes sent, or -1 if an error occurred.

ERRORS

The calls fail if:

EBADF	*s* is an invalid file descriptor.
EINTR	The operation was interrupted by delivery of a signal before any data

	could be buffered to be sent.
EINVAL	*tolen* is not the size of a valid address for the specified address family.
EMSGSIZE	The socket requires that message be sent atomically, and the message was too long.
ENOMEM	There was insufficient memory available to complete the operation.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTSOCK	*s* is not a socket.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

getsockopt
setsockopt

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int getsockopt (int *s*, int *level*, int *optname*, void **optval*, int **optlen*);

int setsockopt (int *s*, int *level*, int *optname*, void **optval*, int *optlen*);

DESCRIPTION

getsockopt and setsockopt manipulate options associated with a socket, *s*. 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, *level* is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* is set to the TCP protocol number (see getprotobyname()).

The parameters *optval* and *optlen* are used to access option values for setsockopt. For getsockopt, they identify a buffer in which the value(s) for the requested option(s) are to be returned. For getsockopt, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. Use a 0 *optval* if no option value is to be supplied or returned.

optname and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file <sys/socket.h> contains definitions for the socket-level options described below. Options at other protocol levels vary in format and name.

Most socket-level options take an int for *optval*. For setsockopt, the *optval* parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a struct linger parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in <sys/socket.h>. struct linger contains the following members:

l_onoff	option on/off
l_linger	linger time

The following options are recognized at the socket level. Except as noted, each may be examined with getsockopt and set with setsockopt.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data is present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band

SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a bind() call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken and processes using the socket are notified using a SIGPIPE signal. 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 un-sent messages are queued on a socket and a close(BA_OS) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close 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 the setsockopt call when SO_LINGER is requested). If SO_LINGER is disabled and a close 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. 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 or read calls without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options that 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.

Finally, SO_TYPE and SO_ERROR are options used only with getsockopt. SO_TYPE returns the type of the socket (for example, SOCK_STREAM). It is useful for servers that inherit sockets on startup. 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.

RETURN VALUES

If successful, getsockopt returns 0; otherwise it returns -1 and sets errno to indicate the error.

ERRORS

The call succeeds unless:

EBADF	The argument *s* is not a valid file descriptor.
ENOMEM	There was insufficient memory available for the operation to complete.
ENOPROTOOPT	The option is unknown at the level indicated.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTSOCK	The argument *s* is not a socket.

shutdown

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

int shutdown (int *s*, int *how*);

DESCRIPTION

The shutdown call shuts down all or part of a full-duplex connection on the socket associated with *s*. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

RETURN VALUES

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

EBADF	*s* is not a valid file descriptor.
ENOMEM	There was insufficient user memory available for the operation to complete.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTCONN	The specified socket is not connected.
ENOTSOCK	*s* is not a socket.

NOTES

The *how* values should be defined constants.

socket

NAME

socket - create an endpoint for communication

SYNOPSIS

#include <sys/types.h>

#include <sys/socket.h>

int socket (int *domain*, int *type*, int *protocol*);

DESCRIPTION

socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <sys/socket.h>.

The only supported protocol family is PF_INET.

The socket has the indicated *type*, which specifies the communication semantics. Currently defined *type*s are:

SOCK_STREAM: A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_STREAM A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).

SOCK_SEQPACKET 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 is protocol specific, and presently not implemented for any protocol family.

protocol 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, multiple 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. If a protocol is specified by the caller, then it will be packaged into a socket level option request and sent to the underlying protocol layers.

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() call. Once connected, data may be transferred using read(BA_OS) and write(BA_OS) calls or some variant of the send() and recv() calls. When a session has been completed, a close(BA_OS) may be performed. Out-of-band data may also be transmitted as described on the send() manual page and received as described on the recv() manual page.

The communications protocols used to implement a SOCK_STREAM insure 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 a extended period (for instance 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that read calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM sockets allow datagrams to be sent to correspondents named in sendto calls. Datagrams are generally received with recvfrom, which returns the next datagram with its return address.

An ioctl(BA_OS) 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 with SIGPOLL signals.

The operation of sockets is controlled by socket level options. These options are defined in the file <sys/socket.h>. setsockopt and getsockopt() 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 call fails if:

EACCES	Permission to create a socket of the specified type and/or protocol is denied.
EMFILE	The per-process descriptor table is full.
ENOMEM	Insufficient user memory is available.
ENOSR	There were insufficient STREAMS resources available to complete the operation.
EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this domain.

SPARC COMPLIANCE DEFINITION 2.3

libsys

__div64

NAME

__div64 - 64 bit division function

SYNOPSIS

long long __div64(long long a, long long b);

DESCRIPTION

The function __div64() computes the quotient of the division of the numerator “a” by the denominator “b”, truncates any fractional part, and return the signed long long results.

This function returns 0 if “b” is 0.

Trap handling, when the divisor is zero, is intentionally not present in this specification, since it is considered SPARC architecture version dependent.

__dtoll

NAME

__dtoll - convert double to long long

SYNOPSIS

long long __dtoll (double d)

DESCRIPTION

This function converts the double precision value of “d” to a signed long long (integer result) by truncating (discarding) any fractional part and returns the signed long long value.

__dtoll() raises an invalid exception if the integer portion is outside of the range:

$$-2^{63} \leq d < 2^{63}$$

and returns the negative number in the inequality expression above if “d” is negative, otherwise returning the positive number in the inequality.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FdTOx instruction.

__dtoull

NAME

__dtoull - convert double to unsigned long long

SYNOPSIS

unsigned long long __dtoull (double d);

DESCRIPTION

This function converts the double precision value of “d” to an unsigned long long (integer result) by truncating (discarding) any fractional part and returns the unsigned long long value.

__dtoull raises an invalid exception if the integer portion of “d” is outside of the range:

$$0 \leq \text{abs}(d) < 2^{64.}$$

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FdTOx instruction.

__ftoll

NAME

__ftoll - convert float to long long

SYNOPSIS

long long __ftoll (float f);

DESCRIPTION

This function converts the single precision value of "f" to a signed long long (integer result) by truncating (discarding) any fractional part and returns the signed long long value.

__ftoll() raises an invalid exception if the integer portion is outside of the range:

$$-2^{63} \leq f < 2^{63}$$

and returns the negative number in the inequality expression above if "f" is negative, otherwise returning the positive number in the inequality.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FsTOx instruction.

__ftoull

NAME

__ftoull - convert float to unsigned long long

SYNOPSIS

unsigned long long __ftoull(float f);

DESCRIPTION

This function converts the single precision value of “f” to an unsigned long long (integer result) by truncating (discarding) any fractional part and returns the unsigned long long value.

__ftoull raises an invalid exception if the integer portion of “f” is outside of the range:

$$0 \leq \text{abs}(f) < 2^{64.}$$

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FsTOx instruction.

__mul64

NAME

__mul64 - 64 bit multiplication function

SYNOPSIS

long long __mul64(long long a, long long b);

DESCRIPTION

This function implements the multiplication of “a” and “b” (“a * b”).

This function returns $p - 2^{64}$ if $p \geq 2^{63}$; it returns “p” otherwise. Where “p” denote the mathematical product modulo 2^{64} of “a” and “b”; “p” is in the range:

$$0 \leq p < 2^{64}$$

Overflow handling is intentionally not present in this specification, since it is considered SPARC architecture version dependent.

__rem64

NAME

__rem64 - 64 bit remain function

SYNOPSIS

long long __rem64(long long a, long long b);

DESCRIPTION

The function __rem64() computes the remainder of the division of the numerator “a” by the “denominator “b” and returns the signed long long result.

This function returns 0 if “b” is 0.

Trap handling, if the divisor is zero, is intentionally not present in this specification, since it is considered SPARC architecture version dependent.

__udiv64

NAME

__udiv64 - Unsigned 64 bit division function

SYNOPSIS

unsigned long long __udiv64(unsigned long long a, unsigned long long b);

DESCRIPTION

The function __udiv64() computes the quotient of the division of the numerator “a” by the denominator “b”, truncates any fractional part, and return the unsigned long long result.

This function returns 0 if “b” is 0.

Trap handling, if the divisor is zero, is intentionally not present in this specification, since it is considered SPARC architecture version dependent.

__umul64

NAME

__umul64 - Unsigned 64 bit multiplication function

SYNOPSIS

unsigned long long __umul64(unsigned long long a, unsigned long long b);

DESCRIPTION

This function implements the multiplication of “a” and “b” (“a * b”).

It returns the product modulo 2^{64} of “a” and “b”. The result is in unsigned long long.

Overflow handling is intentionally not present in this specification, since it is considered SPARC architecture version dependent.

__urem64

NAME

__urem64 - unsigned 64 bits remain function

SYNOPSIS

unsigned long long __urem64(unsigned long long a, unsigned long long b);

DESCRIPTION

The function __urem64() computes the remainder of the division of the numerator “a” by the “denominator “b” and returns the unsigned long long result.

This function returns 0 if “b” is 0.

Trap handling is intentionally not present in this specification, since it is considered SPARC architecture version dependent.

_Q_lltoq

NAME

_Q_lltoq - Convert long long to long double

SYNOPSIS

long double _Q_lltoq (long long a);

DESCRIPTION

This function converts the long long value of “a” to quad precision (floating result) and returns the quad precision value.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FxTOq instruction.

_Q_qtoll

NAME

_Q_qtoll - convert long double to long long

SYNOPSIS

long long _Q_qtoll(long double a);

DESCRIPTION

This function converts the quad precision value of “a” to a signed long long (integer result) by truncating (discarding) any fractional part and returns the signed long long value.

__Q_qtoll() raises an invalid exception if the integer portion is outside of the range:

$$-2^{63} \leq a < 2^{63}$$

and returns the negative number in the inequality expression above if “a” is negative, otherwise returning the positive number in the inequality.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FqTOx instruction.

_Q_qtoull

NAME

_Q_qtoull - convert double to unsigned long long.

SYNOPSIS

unsigned long long _Q_qtoull (long double a);

DESCRIPTION

This function converts the quad precision value of “a” to an unsigned long long (integer result) by truncating (discarding) any fractional part and returns the unsigned long long value.

_Q_qtoull raises an invalid exception if the integer portion of “a” is outside of the range:

$$0 \leq \text{abs}(a) < 2^{64.}$$

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FqTOx instruction.

_Q_ulltoq

NAME

_Q_ulltoq - convert unsigned long long to long double

SYNOPSIS

long double _Q_ulltoq (unsigned long long a);

DESCRIPTION

This function converts the unsigned long long value of “a” to quad precision (floating result) and returns the quad precision value.

Rounding, overflow and exceptions handling are intentionally not present in this specification, since they are considered SPARC architecture version dependent. There is no guarantee that their behavior is similar to SPARC Architecture Version 9 FxTOq instruction.

fgetgrent_r

NAME

fgetgrent_r - get group entry

SYNOPSIS

#include <grp.h>
struct group *fgetgrent_r (FILE *f, struct group *result, char *buffer, int buflen);

DESCRIPTION

fgetgrent_r() reads and parses the next line from the stream "f", which is assumed to have the format of the group file, where each entry is of the form:

groupname: password: gid: user-list

The function fgetgrent_r() provides a reentrant interface for the fgetgrent() function which uses static storage that is re-used in each call. The use of static storage makes fgetgrent() unsafe for use in multithreaded applications. fgetgrent_r() performs the same operation as fgetgrent(). fgetgrent_r(), however, uses buffers supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications.

The parameter "result" must be a pointer to a "struct group" structure allocated by the caller. On successful completion, the function returns the group entry in this structure. The parameter "buffer" must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the group data. All of the pointers within the returned struct group result point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the group entry. The parameter "buflen" should give the size in bytes of the buffer indicated by buffer.

RETURN VALUES

Group entries are represented by the struct group structure defined in <grp.h>:

```
struct group {
      char        *gr_name;   /* the name of the group */
      char        *gr_passwd; /* the encrypted group password */
      gid_t       gr_gid;     /* the numerical group ID */
      char        **gr_mem;   /* vector of pointers to member names */
};
```

The function fgetgrent_r() returns a pointer to a struct group if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

When the pointer returned by fgetgrent_r() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE fgetgrent_r() will return NULL and set errno to ERANGE if the length of the buffer supplied by caller is not large enough to store the result.

NOTES Programs that use fgetgrent_r() cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

fgetpwent_r

NAME

fgetpwent_r - get password entry

SYNOPSIS

#include <pwd.h>
struct passwd *fgetpwent_r (FILE *f, struct passwd *result, char *buffer, int buflen);

DESCRIPTION

This function is used to obtain password entries. fgetpwent_r() reads and parses the next line from the stream f, which is assumed to have the format of the passwd file, where each entry is of the form:

username: password: uid: gid: gcos-field: home-dir: login-shell

The function fgetpwent_r() provides a reentrant interface for fgetpwent(). fgetpwent_r() performs the same operation as fgetpwent(). fgetpwent_r(), however, uses buffers supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications. fgetpwent() is not safe for use in multithreaded applications since it uses static storage that is re-used in each call to this routine.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter result must be a pointer to a struct passwd structure allocated by the caller. On successful completion, the function returns the password entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the password data. All of the pointers within the returned struct passwd result point to data stored within this buffer. See RETURN VALUES. The buffer must be large enough to hold all of the data associated with the password entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer.

RETURN VALUES

Password entries are represented by the struct passwd structure defined in <pwd.h>:

```
struct passwd {
      char  *pw_name;         /* user's login name */
      char  *pw_passwd;       /* no longer used */
      uid_t  pw_uid;          /* user's uid */
      gid_t  pw_gid;          /* user's gid */
      char  *pw_age;          /* not used */
      char  *pw_comment;      /* not used */
      char  *pw_gecos;        /* typically user's full name */
      char  *pw_dir;          /* user's home dir */
      char  *pw_shell;        /* user's login shell */
};
```

The function fgetpwent_r() returns a pointer to a struct passwd if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

When the pointer returned by the reentrant function fgetpwent_r() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE — fgetpwent_r() will return NULL and set errno to ERANGE if the length of the buffer supplied by caller is not large enough to store the result.

NOTES

fgetpwent_r() cannot be linked statically since, the implementations of this function employ dynamic loading and linking of shared objects at run time.

If the shell field is empty, “login” automatically assigns the default shell.

fork

NAME

fork - create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork() causes the creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

- real user ID, real group ID, effective user ID, effective group ID
- environment
- open file descriptors
- close-on-exec flags (see exec(BA_OS))
- signal handling settings (that is SIG_DFL, SIG_IGN, SIG_HOLD, Function address)
- supplementary group IDs
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on/off status
- nice value (see nice(KE_OS))
- scheduler class (see priocntl(RT_OS))
- all attached shared memory segments (see shmop(KE_OS))
- process group ID -- memory mappings (see mmap(KE_OS))
- session ID (see exit(BA_OS))
- current working directory
- root directory
- file mode creation mask (see umask(BA_OS))
- resource limits (see getrlimit(BA_OS))
- controlling terminal
- saved user ID and group ID

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may or may not be inherited according to the policy of that particular class (see priocntl(RT_OS)).

The child process differs from the parent process in the following ways:

- The child process has a unique process ID which does not match any active process group ID.

- The child process has a different parent process ID (that is, the process ID of the parent process).

- The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

- Each shared memory segment remains attached and shm_nattach is incremented by 1.

- All semadj values are cleared (see semop(KE_OS))).

- Process locks, text locks, data locks, and other memory locks are not inherited by the child (see plock(KE_OS) and memcntl(RT_OS)).

- The child process's tms structure is cleared: tms_utime, stime, cutime, and cstime are set to 0 (see times(BA_OS)).

- The child processes resource utilizations are set to 0; see getrlimit(BA_OS). The it_value and it_interval values for the ITIMER_REAL timer are reset to 0; see getitimer(RT_OS).

- The set of signals pending for the child process is initialized to the empty set.

- No asynchronous input or asynchronous output operations are inherited by the child.

Record locks set by the parent process are not inherited by the child process (see fcntl(BA_OS)).

fork() duplicates all the threads (see thr_create) and LWPs in the parent process in the child process.

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 (pid_t) - 1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

ERRORS

fork() will fail and no child process will be created if one or more of the following are true:

EAGAIN There are two conditions that will cause an EAGAIN error.

The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

The total amount of system memory available is temporarily insufficient to duplicate this process.

ENOMEM There is not enough swap space.

SEE ALSO

alarm(BA_OS), exec(BA_OS), exit(BA_OS), fcntl(BA_OS), getitimer(RT_OS), getrlimit(BA_OS), mmap(KE_OS), nice(KE_OS), plock(KE_OS), priocntl(RT_OS), ptrace(KE_OS), semop(KE_OS), shmop(KE_OS), times(BA_OS), umask(BA_OS), wait(BA_OS), memcntl(RT_OS), signal(BA_OS), system(BA_OS), thr_create

NOTES

The semantics of fork() in a multi-threaded application are designated as EXPERIMENTAL. The SCD2.3 definition of the multi-threaded semantics for fork() is that the entire process is duplicated (i.e. all its threads). This differs from the POSIX 1003.1c specification in which only the thread invoking fork() is duplicated in an MT application. MT semantics equivalent to the POSIX 1003.1c of fork() are offered by the SCD2.3 fork1() interface.

Be careful to call _exit() rather than exit(BA_OS) if you cannot execve(), since exit(BA_OS) will flush and close standard I/O channels, and thereby corrupt the parent processes standard I/O data structures. Using exit(BA_OS) will flush buffered data twice. See exit(BA_OS).

In a multi-threaded process, fork() can cause blocking system calls to be interrupted and return with an error of EINTR

getgrent_r

NAME

getgrent_r - get group entry

SYNOPSIS

#include <grp.h>
struct group *getgrent_r (struct group *result, char *buffer, int buflen);

DESCRIPTION

getgrent_r() is used to obtain entries from the system's groups database. The function getgrent_r() provides a reentrant interface for the getgrent() function which uses static storage that is re-used in each call. The use of static storage makes getgrent() unsafe for use in multithreaded applications.

getgrent_r() performs the same operation as getgrent(). getgrent_r() uses a buffer supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications.

The parameter result must be a pointer to a struct group structure allocated by the caller. On successful completion, the function returns the group entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the group data. All of the pointers within the returned struct group result point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the group entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer.

RETURN VALUES

Group entries are represented by the struct group structuredefined in <grp.h>:

```
struct group {
      char  *gr_name;   /* the name of the group */
      char  *gr_passwd; /* the encrypted group password */
      gid_t gr_gid;     /* the numerical group ID */
      char  **gr_mem;   /* vector of pointers to member names */
};
```

The function getgrent_r() returns a pointer to a struct group if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

When the pointer returned by the reentrant function getgrent_r() is non-NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE — getgrent_r() will return NULL and seterrno to ERANGE if the length of the buffer supplied by caller is not large enough to store the result.

NOTES

Programs that use getgrent_r() cannot be linked statically since the implementations of this function employ dynamic loading and linking of shared objects at run time.

getlogin_r

NAME

getlogin_r - get login name

SYNOPSIS

```
#include <stdlib.h>

char *getlogin_r(char *name, int namelen);
```

DESCRIPTION

getlogin_r() returns a pointer to the login name associated with the controlling terminal. It may be used in conjunction with getpwnam_r() to locate the correct password file entry when the same user id is shared by several login names.

If getlogin_r() is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login name is to call cuserid(), or to call getlogin_r() and if it fails to call getpwuid_r().

getlogin_r() has the same functionality as getlogin() except that a buffer name with length namelen has to be supplied by the caller to store the result. name must be at least LOGNAME_MAX bytes in size (defined in limits.h).

RETURN VALUES

Returns a null pointer if the login name is not found.

ERRORS

getlogin_r() will fail if the following is true:

ERANGE The size of the buffer is smaller than the result to be returned.

NOTES

The getlogin_r() interface is different from the POSIX 1003.1c interface. The function getlogin_r is defined in POSIX as following:

```
int     getlogin_r(char *name, size_t namelen);
```

This function is designated as EXPERIMENTAL.

The return values point to static data whose content is overwritten by each call.

getlogin() is unsafe in multi-thread applications. getlogin_r() should be used instead.

getpwent_r

NAME

getpwent_r - get password entry

SYNOPSIS

#include <pwd.h>
struct passwd *getpwent_r (struct passwd *result, char *buffer, int buflen);

DESCRIPTION

This function is used to obtain password entries. The function getpwent_r() is used to enumerate password entries from the system's passwords database. Successive calls to getpwent_r() return either successive entries or NULL, indicating the end of the enumeration.

The function getpwent_r() provides a reentrant interface for getpwent(). getpwent_r() performs the same operation as getpwent(). getpwent_r() uses buffers supplied by the caller to store returned results, and is safe for use in both single-threaded and multithreaded applications. getpwent() is not safe for use in multithreaded applications since it uses static storage that is re-used in each call to this routine.

The parameter result must be a pointer to a struct passwd structure allocated by the caller. On successful completion, getpwent_r() returns the password entry in this structure. The parameter buffer must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the password data. All of the pointers within the returned struct passwd result point to data stored within this buffer. See RETURN VALUES. The buffer must be large enough to hold all of the data associated with the password entry. The parameter buflen should give the size in bytes of the buffer indicated by buffer.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. If multiple threads interleave calls to getpwent_r(), the threads will enumerate disjoint subsets of the password database.

RETURN VALUES

Password entries are represented by the struct passwd structure defined in <pwd.h>:

```
struct passwd {
      char  *pw_name;         /* user's login name */
      char  *pw_passwd;       /* no longer used */
      uid_t pw_uid;           /* user's uid */
      gid_t pw_gid;           /* user's gid */
      char  *pw_age;          /* not used */
      char  *pw_comment;      /* not used */
      char  *pw_gecos;        /* typically user's full name */
      char  *pw_dir;          /* user's home dir */
      char  *pw_shell;        /* user's login shell */
};
```

The function getpwent_r() returns a pointer to a struct passwd if it successfully enumerates an entry; otherwise it returns NULL, indicating the end of the enumeration.

The function getpwent() uses static storage, so returned data must be copied before a subsequent call to this function if the data is to be saved. When the pointer returned by getpwnam_r() is non-

NULL, it is always equal to the result pointer that was supplied by the caller.

ERRORS

ERANGE getpwent_r() will return NULL and set errno to ERANGE if the length of the buffer supplied by caller is not large enough to store the result.

NOTES

getpwent_r cannot be linked statically since, the implementations of this function employ dynamic loading and linking of shared objects at run time.

If the shell field is empty, “login” automatically assigns the default shell.

getgrgid_r
getgrnam_r
getpwnam_r
getpwuid_r
readdir_r

NAME

getgrgid_r, getgrnam_r, getpwnam_r, getpwuid_r, readdir_r - Support routines for multithreading added to libsys and libc.

SYNOPSIS

```
#include <grp.h>

struct group *getgrgid_r (gid_t gid, struct group *result, char *buffer, int buflen);

struct group *getgrnam_r(const char *name, struct group *result, char *buffer, int buflen);


#include <pwd.h>

struct passwd *getpwnam_r (const char *name, struct passwd *result, char *buffer, int buflen);

struct passwd *getpwuid_r (uid_t uid, struct passwd *result, char *buffer, int buflen);


#include <dirent.h>

struct dirent *readdir_r(DIR *dirp, struct dirent *res);
```

DESCRIPTION and RETURN VALUES

These functions are "reentrant" versions of existing functions. They exist as the definition of the existing functions prevents the transparent implementation of multithreading, usually because of the use of a static storage area. In general, these functions are exactly equivalent to the non-reentrant versions in terms of function and results, but differ in providing for the implementation the necessary storage for completion of the function.

getgrgid_r and getgrnam_r are equivalent to getgrgid and getgrnam, respectively. When these functions succeed, they return the argument *result* as their value. Otherwise they return NULL. When successful, the contents of *result* have been updated to return the group entry associated with either *name* or *gid*, respectively. *buf* is provided in order to store the strings and pointers needed to describe a group entry, and is *buflen* in length. If *buflen* is not large enough to store the resulting strings, the functions return NULL with errno set to ERANGE.

getpwuid_r and getpwnam_r are equivalent to getgrgid and getgrnam, respectively. When these functions succeed, they return the argument *result* as their value. Otherwise they return NULL. When successful, the contents of *result* have been updated to return the password entry associated with either *name* or *uid*, respectively. *buf* is provided in order to store the strings and pointers needed to describe a password entry, and is *buflen* in length. If *buflen* is not large enough to store the resulting strings, the functions return NULL with errno set to ERANGE.

readdir_r is equivalent to readdir except that *res* must be supplied by the caller to store the result. To allocate *res*, a block of storage equivalent to sizeof (struct dirent) + _POSIX_PATH_MAX should be allocated.

NOTES

These functions are designated as EXPERMIMENTAL since they have interfaces which are different from the ones in POSIX 1003.1c. The interfaces of these functions are in POSIX as following:

```
int   getgrgid_r (gid_t             gid,
                  struct group      *grp,
                  char              *buffer,
                  size_t            bufsize,
                  struct group      **result);

int   getgrnam_r (const char        *name,
                  struct group      *grp,
                  char              *buffer,
                  size_t            bufsize,
                  struct group      **result);

int   getpwnam_r (const char        *name,
                  struct passwd     *pwd,
                  char              *buffer,
                  size_t            buflen,
                  struct passwd     **result);

int   getpwuid_r (uid_t             uid,
                  struct passwd     *pwd,
                  char              *buffer,
                  size_t            bufsize,
                  struct passwd     **result);

int   readdir_r ( DIR               *dirp,
                  struct direct     *entry,
                  struct dirent     **result);
```

makecontext
swapcontext

NAME

makecontext, swapcontext - manipulate user contexts

SYNOPSIS

```
#include <ucontext.h>

void    makecontext (ucontext_t *ucp,    void(*func)(),    int argc,...);

int     swapcontext (ucontext_t *oucp,  ucontext_t *ucp);
```

DESCRIPTION

These functions are useful for implementing user-level context switching between multiple threads of control within a process.

makecontext() modifies the context specified by ucp, which has been initialized using getcontext(); when this context is resumed using swapcontext() or setcontext() (see getcontext(BA_OS)), program execution continues by calling the function func, passing it the arguments that follow argc in the makecontext() call. The integer value of argc must match the number of arguments that follow argc. Otherwise the behavior is undefined.

swapcontext() saves the current context in the context structure pointed to by oucp and sets the context to the context structure pointed to by ucp.

RETURN VALUES

On successful completion, swapcontext returns a value of zero. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

These functions will fail if either of the following is true:

EFAULT ucp or oucp points to an invalid address.

ENOMEM ucp does not have enough stack left to complete the operation.

SEE ALSO

exit(BA_OS), getcontext(BA_OS), sigaction(BA_OS), sigprocmask(BA_OS)

NOTES

The size of the ucontext_t structure may change in future releases. To remain binary compatible, users of these features must always use makecontext() or getcontext() to create new instances of them.

sbrk

NAME

sbrk - query the current break value

SYNOPSIS

#include <unistd.h>

void *sbrk (const int 0);

DESCRIPTION

The function sbrk is used to query the amount of space allocated for the calling process's data segment [see exec(BA_OS)].

STATUS

This function is DEPPRECATED effective November 1st, 1993. It may be removed from the SCD as early as November 1st, 1996.

DIAGNOSTICS

Upon successful completion, sbrk returns the current break value. Otherwise, a value of -1 is returned and errno is set to indicate the error. If sbrk is called with a non-zero value, the application is not portable.

RATIONALE

Calling sbrk(0) yields a value that, at one time, had a predictable, well defined interpretation. It has not had this property for many years, since the wide-spread usage of sparse, demand-paged address spaces. Its use is deprecated because the interpretation of the value returned is so highly variable as to be non-portable. It is more appropriately regarded as a function yielding a value relevant to one of many attributes of memory occupancy. sbrk (non-zero) is not specified in any relevant standard, as its interactions with and dependencies upon other memory allocation mechanisms (e.g., malloc) are undefined. The use of sbrk (non-zero) is non-conforming since the implementation of system supplied functions may freely use such memory allocation mechanisms.

NOTES

Applications desiring memory allocation functionality should use malloc for this purpose. Alternatively, applications may construct their own memory allocation arenas by building upon mmap and mappings to /dev/zero

ttyname
ttyname_r

NAME

ttyname, ttyname_r - find name of a terminal

SYNOPSIS

#include <stdlib.h>

char *ttyname (int fildes);

char *ttyname_r (int fildes, char *buf, int len);

DESCRIPTION

ttyname() returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor fildes.

ttyname_r() has the equivalent functionality to ttyname() except that a buffer buf with length len must be supplied by the caller to store the result. buf must be at least POSIX_PATH_MAX in size (defined in <limits.h>).

RETURN VALUES

ttyname() and ttyname_r() return a NULL pointer if fildes does not describe a terminal device in directory /dev.

ERRORS

ttyname_r() will fail if the following is true:

ERANGE The size of the buffer is smaller than the result to be returned.

NOTES

ttyname_r is designated as EXPERMIMENTAL since it has an interface which is different from the one in POSIX 1003.1c. ttyname_r interface in POSIX is as following:

```
int         ttyname_r ( int         fildes,
                        char        *name,
                        size_t      namesize);
```

SPARC COMPLIANCE DEFINITION 2.3

libthread

cond_broadcast
cond_destroy
cond_init
cond_timedwait
cond_signal
cond_wait

NAME

condition, cond_init, cond_destroy, cond_wait, cond_timedwait, cond_signal, cond_broadcast - condition variables

SYNOPSIS

```
#include <synch.h>

int cond_init (cond_t *cvp, int type, void *arg);

int cond_destroy (cond_t *cvp);

int cond_wait (cond_t *cvp, mutex_t *mp);

int cond_timedwait (cond_t *cvp, mutex_t *mp, timestruc_t *abstime);

int cond_signal (cond_t *cvp);

int cond_broadcast (cond_t *cvp);
```

DESCRIPTION

A condition variable enables threads to atomically block until a condition is satisfied. The condition is tested under the protection of a mutual exclusion lock (mutex). When the condition is false, a thread typically blocks on a condition variable and atomically releases the mutex waiting for the condition to change. When another thread changes the condition, it may signal the associated condition variable to cause one or more waiting threads to wake up, reacquire the mutex, and re-evaluate the condition.

Condition variables can be used to synchronize threads among processes if they are allocated in memory that is writable and shared by the cooperating processes (see mmap(KE_OS)) and have been initialized for this behavior.

Condition variables must be initialized before use. cond_init() initializes the condition variable pointed to by *cvp*. A condition variable can potentially have several different types of behavior, specified by *type*. No current *type* uses *arg* although a future *type* may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS — The condition variable can be used to synchronize threads in this process and other processes. Only one process should initialize the condition variable. *arg* is ignored.

USYNC_THREAD — The condition variable can be used to synchronize threads in this process, only. *arg* is ignored.

Condition variables may also be initialized by allocation in zeroed memory. In this case a type of USYNC_THREAD is assumed. Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be re-initialized while other threads may be using it.

cond_destroy() destroys any state associated with the condition variable pointed to by *cvp*. The space for storing the condition variable is not freed. A condition variable must not be destroyed while other threads may be using it.

cond_wait() atomically releases the mutex pointed to by *mp* and causes the calling thread to block on the condition variable pointed to by *cvp*. The blocked thread may be awakened by cond_signal(), cond_broadcast(), or when interrupted by delivery of a signal or a fork(). Any change in value of a condition associated with the condition variable cannot be inferred by the return of cond_wait() and any such condition must be re-evaluated.

cond_timedwait() is similar to cond_wait(), except that the calling thread will not block past the time of day specified by *abstime*. If the time of day becomes greater than *abstime* then cond_timedwait() returns with the error code ETIME.

cond_wait() and cond_timedwait() always return with the mutex locked and owned by the calling thread even when returning an error.

cond_signal() unblocks one thread that is blocked on the condition variable pointed to by *cvp*.

cond_broadcast() unblocks all threads that are blocked on the condition variable pointed to by *cvp*.

If no threads are blocked on the condition variable then cond_signal() and cond_broadcast() have no effect.

Both functions should be called under the protection of the same mutex that is used with the condition variable being signaled. Otherwise the condition variable may be signaled between the test of the associated condition and blocking in cond_wait(). This can cause an infinite wait.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL — Invalid argument. For cond_init(), *type* is not a recognized type. For cond_timedwait(), the specified number of seconds, *abstime*, is greater than some implementation dependent time that is at least the start time of the application plus 50,000,000, or the number of nanoseconds is greater than or equal to 1,000,000,000.

If any of the following conditions are detected, cond_wait() or cond_timedwait() fails and returns the corresponding value:

EINTR — The wait was interrupted by a signal or fork().

If any of the following conditions are detected, cond_timedwait() fails and returns the corresponding value:

ETIME — The time specified by *abstime* has passed.

EXAMPLES

cond_wait() is normally used in a loop testing some condition, as follows:

```
(void) mutex_lock(mp);

while (cond == FALSE) {
      (void) cond_wait (cvp, mp);
}

(void) mutex_unlock(mp);
```

cond_timedwait() is also normally used in a loop testing some condition. It uses an absolute

timeout value as follows:

```
timestruc_t to;
...
(void) mutex_lock(mp);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
      err = cond_timedwait (cvp, mp, &to);
      if (err == ETIME) {
              /* timeout, do something */
              break;
}
}
(void) mutex_unlock(mp);
```

This sets a bound on the total wait time even though cond_timedwait() may return several times due to the condition being signaled or the wait being interrupted.

NOTES

These interfaces also available via: #include <thread.h>

By default, there is no defined order of unblocking if multiple threads are waiting for a condition variable.

fork1

NAME

fork1 - create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork1 (void);
```

DESCRIPTION

fork1()causes creation of a new process. It differs from fork() in that fork() duplicates all the threads in the parent process in the child process, while fork1() duplicates only the calling thread in the child process.

RETURN VALUES

Upon successful completion, fork1() 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 (pid_t)-1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

ERRORS

Same as fork().

NOTES

When calling fork1() the thread in the child must not depend on any resources that are held by threads that no longer exist in the child. In particular, locks held by these threads will not be released.

mutex_destroy
mutex_init
mutex_lock
mutex_trylock
mutex_unlock

NAME

mutex, mutex_init, mutex_destroy, mutex_lock, mutex_trylock, mutex_unlock - mutual exclusion locks

SYNOPSIS

```
#include <synch.h>

int mutex_init (mutex_t *mp, int type, void *arg);

int mutex_destroy (mutex_t *mp);

int mutex_lock (mutex_t *mp);

int mutex_trylock (mutex_t *mp);

int mutex_unlock (mutex_t *mp);
```

DESCRIPTION

Mutual exclusion locks (mutexes) are used to serialize the execution of threads. They are typically used to ensure that only one thread executes a critical section of code at any one time (mutual exclusion).

Mutexes can be used to synchronize threads in this process and other processes if they are allocated in memory that is writable and shared among the cooperating processes (see mmap(KE_OS)) and have been initialized for this behavior.

Mutexes must be initialized before use. mutex_init() initializes the mutex pointed to by *mp*. A mutex can potentially have several different types of behavior, specified by *type*. No current *type* uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS — The mutex can be used to synchronize threads in this process and other processes. Only one process should initialize the mutex. *arg* is ignored.

USYNC_THREAD — The mutex can be used to synchronize threads in this process, only. *arg* is ignored.

Mutexes may also be initialized by allocation in zeroed memory. In this case a type of USYNC_THREAD is assumed. Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be re-initialized while other threads may be using it.

mutex_destroy() destroys any state associated with the mutex pointed to by *mp*. The space for storing the mutex is not freed. A mutex lock must not be destroyed while other threads may be using it.

mutex_lock() locks the mutex pointed to by *mp*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. When mutex_lock() returns, the mutex is locked and the calling thread is the owner.

mutex_trylock() attempts to lock the mutex pointed to by *mp*. If the mutex is already locked it returns with an error. Otherwise the mutex is locked and the calling thread is the owner.

mutex_unlock() unlocks the mutex pointed to by *mp*. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). If any other threads are waiting for the mutex to become available, one of them is unblocked. If the calling thread is not the owner of the lock, the behavior of the program is undefined.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL Invalid argument.

If any of the following conditions are detected, mutex_trylock() fails and returns the corresponding value:

EBUSY The mutex pointed to by *mp* was already locked.

NOTES

In the current implementation, mutex_lock(), mutex_unlock(), and mutex_trylock() do not validate the mutex type. Therefore, EINVAL is not returned for an uninitialized mutex or for a mutex with an invalid *type*. The behavior of these interfaces for mutexes containing an invalid *type* is unspecified.

For example, the following call to mutex_lock() might hang. Since mutex is allocated from the stack and is not initialized, it may have junk data in it. mutex needs to be initialized using mutex_init().

int global;

```
main()
{
        mutex_t mutex;

        /*
         * The address of this mutex is passed to threads
         * created from main(). NOTE: this is not recommended
         * style.
         */

...
        if (mutex_lock(&mutex) == 0) {
                /* this call may hang */
                global++;
                mutex_unlock(&mutex);
         }
         else printf ("mutex_lock() failed\n");

...
}
```

Instead, mutex should first be initialized using mutex_init().

```
int global;

main()
{
```

```
        mutex_t mutex;
...
        mutex_init (&mutex, USYNC_THREAD, NULL);
        mutex_lock(&mutex);
        /* This call is now guaranteed to work */
        global++;
        mutex_unlock(&mutex);
}
```

By default, there is no defined order of acquisition if multiple threads are waiting for a mutex.

These interfaces are also available via: #include <thread.h>

rwlock_destroy
rwlock_init
rw_rdlock
rw_tryrdlock
rw_trywrlock
rw_unlock
rw_wrlock

NAME

rwlock, rwlock_init, rwlock_destroy, rw_rdlock, rw_wrlock, rw_tryrdlock, rw_trywrlock, rw_unlock - multiple readers, single writer locks

SYNOPSIS

```
#include <synch.h>

int rwlock_init (rwlock_t *rwlp, int type, void *arg);

int rwlock_destroy (rwlock_t *rwlp);

int rw_rdlock (rwlock_t *rwlp);

int rw_wrlock (rwlock_t *rwlp);

int rw_unlock (rwlock_t *rwlp);

int rw_tryrdlock (rwlock_t *rwlp);

int rw_trywrlock (rwlock_t *rwlp);
```

DESCRIPTION

Multiple readers, single writer (readers/writer) locks allow many threads to have simultaneous read-only access to data while allowing only one thread to have write access at any given time. They are typically used to protect data that is searched more frequently than it is changed.

Readers/writer locks can be used to synchronize threads in this process and other processes if they are allocated in memory that is writable and shared among the cooperating processes (see mmap(KE_OS)) and have been initialized for this behavior.

Readers/writer locks must be initialized before use. rwlock_init() initializes the readers/writer lock pointed to by *rwlp*. A readers/writer lock can potentially have several different types of behavior, specified by *type*. No current type uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS — The readers/writer lock can be used to synchronize threads in this process and other processes. Only one process should initialize the readers/writer lock. *arg* is ignored.

USYNC_THREAD — The readers/writer lock can be used to synchronize threads in this process, only. *arg* is ignored.

Readers/writer locks may also be initialized by allocation in zeroed memory. In this case a type of USYNC_THREAD is assumed. Multiple threads must not initialize the same readers/writer lock simultaneously. A readers/writer lock must not be re-initialized while other threads may be using it.

rwlock_destroy() destroys any state associated with the readers/writer lock pointed to by *rwlp*. The space for storing the readers/writer lock is not freed. A readers/writer lock must not be destroyed while other threads may be using it.

rw_rdlock() acquires a read lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for writing, the calling thread blocks until the write lock is released. More than one thread may hold a read lock on a readers/writer lock at any one time.

rw_tryrdlock() attempts to acquire a read lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for writing, it returns an error.

rw_wrlock() acquires a write lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for reading or writing, the calling thread blocks until all the read locks and write locks are released. Only one thread may hold a write lock on a readers/writer lock at any one time.

rw_trywrlock() attempts to acquire a write lock on the readers/writer lock pointed to by *rwlp*. If the readers/writer lock is already locked for reading or writing, it returns an error.

rw_unlock() unlocks a readers/writer lock pointed to by *rwlp*. The readers/writer lock must be locked and the calling thread must hold the lock either for reading or writing. If any other threads are waiting for the readers/writer lock to become available, one of them is unblocked. If the calling thread does not hold the lock for either reading or writing, the behavior of the program is undefined.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL Invalid argument.

If any of the following conditions are detected, rw_tryrdlock() or rw_trywrlock() fails and returns the corresponding value:

EBUSY The readers/writer lock pointed to by *rwlp* was already locked.

NOTES

These interfaces also available via: #include <thread.h>

By default, there is no defined order of acquisition if multiple threads are waiting for a readers/writer lock. However, implementations usually bias acquisition order in some way so as to avoid writer starvation.

sema_destroy
sema_init
sema_post
sema_trywait
sema_wait

NAME

semaphore, sema_init, sema_destroy, sema_wait, sema_trywait, sema_post - semaphores

SYNOPSIS

```
#include <synch.h>

int sema_init (sema_t *sp, unsigned int count, int type, void * arg);

int sema_destroy (sema_t *sp);

int sema_wait (sema_t *sp);

int sema_trywait (sema_t *sp);

int sema_post (sema_t *sp);
```

DESCRIPTION

Conceptually, a semaphore is a non-negative integer count. Semaphores are typically used to coordinate access to resources. The semaphore count is initialized to the number of free resources. Threads then atomically increment the count when resources are added and atomically decrement the count when resources are removed. When the semaphore count becomes zero, indicating no more resources are present, threads trying to decrement the semaphore will block until the count becomes greater than zero.

Semaphores can be used to synchronize threads in this process and other processes if they are allocated in memory that is writable and is shared among the cooperating processes (see mmap(KE_OS)) and have been initialized for this behavior.

Semaphores must be initialized before use. sema_init() initializes the semaphore pointed to by *sp* to *count*. A semaphore can potentially have several different types of behavior, specified by *type*. No current type uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_PROCESS — The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore. *arg* is ignored.

USYNC_THREAD — The semaphore can be used to synchronize threads in this process, only. *arg* is ignored.

Multiple threads must not initialize the same semaphore simultaneously. A semaphore must not be re-initialized while other threads may be using it.

sema_destroy() destroys any state associated with the semaphore pointed to by *sp*. The space for storing the semaphore is not freed. A semaphore must not be destroyed while other threads may be using it.

sema_wait() blocks the calling thread until the count in the semaphore pointed to by *sp* becomes greater than zero and then atomically decrements it.

sema_trywait() atomically decrements the count in the semaphore pointed to by *sp* if the count is greater than zero. Otherwise it returns an error.

sema_post() atomically increments the count semaphore pointed to by *sp*. If there are any threads blocked on the semaphore, one is unblocked.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, these functions fail and return the corresponding value:

EINVAL	Invalid argument.

If any of the following conditions are detected, sema_wait() fails and returns the corresponding value:

EINTR	The wait was interrupted by a signal.

If any of the following conditions are detected, sema_trywait() fails and returns the corresponding value:

EBUSY	The semaphore pointed to by *sp* has a zero count.

NOTES

These interfaces also available via: #include <thread.h>

By default, there is no defined order of unblocking if multiple threads are waiting for a semaphore.

thr_continue
thr_suspend

NAME

thr_suspend, thr_continue - suspend or continue thread execution

SYNOPSIS

```
#include <thread.h>

int thr_suspend (thread_t target_thread);

int thr_continue (thread_t target_thread);
```

DESCRIPTION

thr_suspend() immediately suspends the execution of the thread specified by *target_thread*. On successful return from thr_suspend(), the suspended thread is no longer executing. Once a thread is suspended, subsequent calls to thr_suspend() have no effect.

thr_continnue() resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to thr_continue() have no effect.

A suspended thread will not be awakened by a signal. The signal stays pending until the execution of the thread is resumed by thr_continue().

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, thr_suspend() or thr_continue() fails and returns the corresponding value:

ESRCH *target_thread* cannot be found in the current process.

thr_create

NAME

thr_create - create a new thread of control

SYNOPSIS

```
#include <thread.h>

int thr_create (void *stack_base, size_t stack_size,
   void *(*start_routine) (void *), void *arg,
   long flags, thread_t *new_thread);
```

DESCRIPTION

thr_create() adds a new thread of control to the current process. The new thread begins execution by calling the function specified by *start_routine* with a single argument, *arg*. If *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine* (see thr_exit).

The new thread will use the stack starting at the address specified by *stack_base* and continuing for *stack_size* bytes. *stack_size* must be greater than the value returned by *thr_min_stack*(). If *stack_base* is NULL then thr_create() allocates a stack for the new thread with at least *stack_size* bytes. If *stack_size* is zero then a default size is used. If *stack_size* is not zero then it must be greater than the value returned by thr_min_stack(). A stack of minimum size may not accommodate the stack frame for *start_function*. If a stack size is specified, it must take into account the requirements *start_function* and the functions that it may call in turn, in addition to the minimum requirement.

flags specifies additional attributes for the created thread. The value in *flags* is constructed from the bitwise inclusive OR of the following:

THR_SUSPENDED	The new thread is created suspended and will not execute *start_routine* until it is started by thr_continue().
THR_DETACHED	The new thread is created detached. Its thread ID and other resources may be reused as soon as the thread terminates. A detached thread cannot be waited for via thr_join().
THR_BOUND	The new thread is created permanently bound to an LWP (i.e. it is a bound thread).
THR_NEW_LWP	The desired concurrency level for unbound threads is increased by one. This is similar to incrementing concurrency by one via thr_setconcurrency). Typically, this adds a new LWP to the pool of LWPs running unbound threads.
THR_DAEMON	The thread is marked as a daemon. The process will exit when all non-daemon threads exit.

If both THR_BOUND and THR_NEW_LWP are specified then, typically, two LWPs are created, one for the bound thread and another for the pool of LWPs running unbound threads.

When *new_thread* is not NULL then it points to a location where the ID of the new thread is stored if thr_create() is successful. The ID is only valid within the calling process.

The new thread inherits the calling thread's signal mask and priority. Pending signals are not inherited.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, thr_create() fails and returns the corresponding value:

EAGAIN	A system limit is exceeded, e.g., too many LWPs were created.
ENOMEM	Not enough memory was available to create the new thread.
EINVAL	*stack_base* is not NULL and *stack_size* is less than the value returned by thr_min_stack().
EINVAL	*stack_base* is NULL and *stack_size* is not zero and is less than the value returned by thr_min_stack().

EXAMPLES

This example shows how to create a default thread with a new signal mask. new_mask is assumed to have a different value than the creator's signal mask (orig_mask). new_mask is set to block all signals except for SIGINT. The creator's signal mask is changed so that the new thread inherits a different mask, and is restored to its original value after thr_create() returns. This examples assumes that SIGINT is also unmasked in the creator. If it is masked by the creator, then unmasking the signal opens the creator up to this signal. The other alternative is to have the new thread set its own signal mask in its start routine.

```
thread_t tid;
sigset_t new_mask, orig_mask;
int error;

(void)sigfillset(&new_mask);
(void)sigdelset(&new_mask, SIGINT);
(void)thr_sigsetmask(SIG_SETMASK, &new_mask, &orig_mask);
error = thr_create (NULL, 0, do_func, NULL, 0, &tid);
(void)thr_sigsetmask(SIG_SETMASK, NULL, &orig_mask);
```

NOTES

Typically, thread stacks allocated by thr_create() begin on page boundaries and any specified size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows will result in a SIGSEGV signal being sent to the offending thread. Thread stacks allocated by the caller are used as is.

Using a default stack size for the new thread, instead of passing a user-specified stack size, results in much better thr_create() performance.

A thread has not terminated until thr_exit() has finished. The only way to determine this is by thr_join(). When thr_join() returns a departed thread, it means that this thread has terminated and its resources are reclaimable. For instance, if a user specified a stack to thr_create(), this stack can only be reclaimed after thr_join() has reported this thread as a departed thread. It is not possible to determine when a detached thread has terminated. A detached thread disappears without leaving a trace.

If there is no explicit synchronization, an unsuspended, detached thread can die and have its thread ID re-assigned to another new thread before its creator returns from thr_create().

thr_exit

NAME

thr_exit - thread termination

SYNOPSIS

```
#include <thread.h>
void thr_exit (void *status);
```

DESCRIPTION

thr_exit() terminates the calling thread. All thread-specific data bindings are released (see thr_keycreate). If the calling thread is not detached, then the thread's ID and the exit status specified by *status* are retained until it is waited for (see thr_join). Otherwise, *status* is ignored and the thread's ID may be reclaimed immediately.

If the calling thread is the last non-daemon thread in the process (see thr_create), then the process terminates with a status of zero (see exit(BA_OS)). If the initial thread returns from main() then the process exits with a status equal to the return value.

RETURN VALUE

thr_exit() does not return.

thr_getconcurrency
thr_setconcurrency

NAME

thr_setconcurrency, thr_getconcurrency - get/set thread concurrency level

SYNOPSIS

```
#include <thread.h>

int thr_setconcurrency (int new_level);

int thr_getconcurrency (void);
```

DESCRIPTION

Unbound threads in a process (see thr_create) may or may not be required to be simultaneously active. By default, the threads system ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency. thr_setconcurrency() permits the application to give the threads system a hint, specified by *new_level*, for the desired level of concurrency. The actual number of simultaneously active threads may be larger or smaller than this number. The value for the desired concurrency level may also be affected by creating threads with the THR_NEW_LWP flag set (see thr_create).

If *new_level* is zero, the threads system will only ensure that a sufficient number of threads are active so that the process can continue to make progress.

thr_getconcurrency() returns the current value for the desired concurrency level. The actual number of simultaneously active threads may be larger or smaller than this number.

RETURN VALUE

thr_setconcurrency() returns zero when successful. A nonzero value indicates an error code.

thr_getconcurrency() always returns the current value for the desired concurrency level.

ERRORS

If any of the following conditions are detected, thr_setconcurrency() fails and returns the corresponding value:

EAGAIN	the specified concurrency level would cause a system resource to be exceeded.
EINVAL	*new_level* is negative.

thr_getprio
thr_setprio

NAME

thr_setprio, thr_getprio - set/get a thread priority

SYNOPSIS

```
#include <thread.h>

int thr_setprio (thread_t target_thread, int pri);

int thr_getprio (thread_t target_thread, int *pri);
```

DESCRIPTION

Each thread has a priority which it inherits from its creator. thr_setprio() changes the priority of the thread, specified by *target_thread*, within the current process to the priority specified by *pri*. By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The *target_thread* will preempt lower priority threads, and will yield to higher priority threads.

The function thr_getprio() stores the current priority for the thread specified by *target_thread* in the location pointed to by *pri*.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates error.

ERRORS

If any of the following conditions are detected, thr_setprio() or thr_getprio() fails and returns the corresponding value:

ESRCH	*target_thread* cannot be found in the current process.

If any of the following conditions are detected, thr_setprio() fails and returns the corresponding value:

EINVAL	The value of *pri* makes no sense for the scheduling class associated with the *target_thread*.

thr_getspecific
thr_keycreate
thr_setspecific

NAME

thr_keycreate, thr_setspecific, thr_getspecific - thread-specific data

SYNOPSIS

```
#include <thread.h>

int thr_keycreate(thread_key_t *keyp, void (*destructor) (void *value));

int thr_setspecific(thread_key_t key, void *value);

int thr_getspecific(thread_key_t key, void **valuep);
```

DESCRIPTION

thr_keycreate() allocates a key that is used to identify data that is specific to each thread in the process. The key is global to all threads in the process. Once a key has been created each thread may bind a value to the key. The values are specific to the binding thread and are maintained for each thread independently. All threads initially have the value NULL associated with the key when it is created. When thr_keycreate() returns successfully the allocated key is stored in the location pointed to by *keyp*. The caller must ensure that storage and access to this key are properly synchronized.

An optional destructor function, specified by *destructor,* may be associated with each key. If a key has a non-NULL destructor function and the thread has a non-NULL value associated with that key, the destructor function is called with the current associated *value* when the thread exits. The order in which the destructor functions are called for all the allocated keys is unspecified.

thr_setspecific() binds *value* to the thread-specific data *key,* for the calling thread.

thr_getspecific() stores the current value bound to *key* for the calling thread into the location pointed to by *valuep*.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, thr_keycreate() fails and returns the corresponding value:

EAGAIN The key name space is exhausted.

If any of the following conditions are detected, thr_keycreate() or thr_setspecific() fails and returns the corresponding value:

ENOMEM Not enough memory is available.

If any of the following conditions are detected, thr_setspecific() or thr_getspecific() fails and returns the corresponding value:

EINVAL *key* is invalid.

EXAMPLES

This examples shows the use of thread-specific data in a function that can be called from more than one thread without special initialization.

```
static mutex_t keylock;
static thread_key_t key;
static int once = 0;

func()
{
        void *ptr;

        if (!once) {
                (void) mutex_lock(&keylock);
                if (!once) {
                        (void) thr_keycreate (&key, free);
                        once++;
             }
                (void) mutex_unlock(&keylock);
         }

        (void) thr_getspecific (key, (void *)&ptr);
        if (ptr == NULL) {
                ptr = malloc(SIZE);
                (void) thr_setspecific (key, ptr);
         }

}
```

thr_join

NAME

thr_join - wait for thread termination

SYNOPSIS

#include <thread.h>

int thr_join (thread_t wait_for, thread_t *departed, void **status);

DESCRIPTION

thr_join() blocks the calling thread until the thread specified by *wait_for* terminates. The specified thread must be in the current process and must not be detached (see thr_create). If *wait_for* is (thread_t)0, then thr_join() waits for any undetached thread in the process to terminate.

If *departed* is not NULL, it points to a location that is set to the ID of the terminated thread if thr_join() returns successfully. If *status* is not NULL, it points to a location that is set to the exit status of the terminated thread if thr_join() returns successfully.

If thr_join() is not successful, the value of the location pointed to by status is unchanged.

Multiple threads cannot wait for the same thread to terminate; one thread will return successfully and the others will fail with an error of ESRCH.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, thr_join() fails and returns the corresponding value:

ESRCH	*wait_for* is not a valid, undetached thread in the current process.
EDEADLK	*wait_for* specifies the calling thread.
EDEADLCK	wait_for is (thread_t)0 and there is no valid, undetached thread in the current process which is not the calling thread.

thr_kill

NAME

thr_kill - send a signal to a thread

SYNOPSIS

```
#include <thread.h>
#include <signal.h>

int thr_kill (thread_t target_thread, int sig);
```

DESCRIPTION

thr_kill() sends the signal, *sig*, to the thread specified by *target_thread*. *target_thread* must be a thread within the same process as the calling thread. *sig* must be one from the list given in signal (BA_ENV)or zero. If *sig* is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of *target_thread*.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions detected, thr_kill() fails and returns the corresponding value:

EINVAL	*sig* is not a valid signal number.
ESRCH	*target_thread* cannot be found in the current process.

thr_min_stack

NAME

thr_min_stack - minimum stack space for a thread

SYNOPSIS

```
#include <thread.h>

size_t thr_min_stack(void);
```

DESCRIPTION

When a thread is created with a user-supplied stack, the user must reserve enough space to run this thread. In a dynamically linked execution environment, it is very hard to know what the minimum stack requirements are for a thread. The function thr_min_stack() returns the amount of space needed to execute a null thread. This is a thread that was created to execute a null procedure. A thread that does something useful should have a stack size that is thr_min_stack() + <some increment>.

Most users should not be creating threads with user-supplied stacks. This functionality was provided to support applications that wanted complete control over their execution environment.

Typically, users should let the threads library manage stack allocation. The threads library provides default stacks which should meet the requirements of any created thread.

RETURN VALUE

thr_min_stack returns the minimum stack size for a thread.

thr_self

NAME

thr_self - get thread identifier

SYNOPSIS

```
#include <thread.h>
thread_t thr_self(void)
```

DESCRIPTION

thr_self() returns the ID of the calling thread.

thr_sigsetmask

NAME

thr_sigsetmask - change and/or examine calling thread's signal mask

SYNOPSIS

#include <thread.h>

#include <signal.h>

int thr_sigsetmask (int how, const sigset_t *set, sigset_t *oset);

DESCRIPTION

thr_sigsetmask() examines and/or changes the calling thread's signal mask. If the value of the argument *set* is not NULL, it points to a set of signals to be used to change the currently blocked set. The value of the argument how determines the manner in which the set is changed. *how* may have one of the following values:

SIG_BLOCK	*set* represent a set of signals to block. They are added to the current signal mask.
SIG_UNBLOCK	*set* represents a set of signals to unblock. These signals are deleted from the current signal mask.
SIG_SETMASK	*set* represents the new signal mask. The current signal mask is replaced by set.

If the value of *oset* is not NULL, it points to space where the previous signal mask is stored. If the value of *set* is NULL, the value of *how* is not significant and the thread's signal mask is unchanged; thus, thr_sigsetmask() can be used to enquire about the currently blocked signals.

RETURN VALUE

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, thr_sigsetmask() fails and returns the corresponding value:

EINVAL	*set* is not NULL and the value of *how* is not defined.

NOTES

It is not possible to block those signals that cannot be ignored (see sigaction(BA_OS)). In addition, if using the threads library, it is not possible to block the signal SIGLWP, reserved by the threads library, and it is not possible to unblock the signal SIGWAITING, which is always blocked on all threads. This restriction is silently imposed by the threads library.

thr_main

NAME

thr_main - identify the main thread

SYNOPSIS

```
#include <thread.h>
int thr_main(void);
```

DESCRIPTION

thr_main — identifies the calling thread as the main thread or not the main thread.

RETURN VALUES

thr_main() returns:

1	if the calling thread is the main thread.
0	if the calling thread is not the main thread.
-1	if libthread is not linked in or thread initialization has not completed.

thr_yield

NAME

thr_yield - yield execution to another thread

SYNOPSIS

```
#include <thread.h>
void thr_yield(void);
```

DESCRIPTION

thr_yield() causes the current thread to yield its execution in favor of another thread with the same or greater priority.

RETURN VALUE

No value is returned.

sigwait

NAME

sigwait - wait until a signal is posted

SYNOPSIS

#include <signal.h>

int sigwait (sigset_t *set);

DESCRIPTION

sigwait() selects a signal in set that is pending on the calling thread (see thr_create()). If no signal in set is pending, then sigwait() blocks until a signal in set becomes pending. The selected signal is cleared from the set of signals pending on the calling thread and the number of the signal is returned. The selection of a signal in set is independent of the signal mask of the calling thread. This means a thread can synchronously wait for signals that are being blocked by the signal mask of the calling thread.

If more than one thread waits for the same signal, only one is unblocked when the signal arrives.

RETURN VALUES

Upon successful completion, a signal number is returned. Otherwise, a value of -1 is returned and errno is set to indicate error.

ERRORS

If any of the following conditions are detected, sigwait() fails and returns the corresponding value:

EINVAL	set contains an unsupported signal number.
EFAULT	set points to an invalid address.

NOTES

sigwait() cannot be used to wait for signals that cannot be caught (see sigaction(BA_OS)). This restriction is silently imposed by the system.

sigwait() is designated as EXPERMIMENTAL since it has an interface which is different from the one in POSIX 1003.1c. sigwait interface in POSIX is as following:

```
int   sigwait (   const sigset_t    *setp,
                  int               *signo);
```

SPARC COMPLIANCE DEFINITION 2.3

Execution Environment

/dev/zero

NAME

/dev/zero

SYNOPSIS

/dev/zero

DESCRIPTION

The device /dev/zero is defined to be a special file which is a source of zeroed, unnamed memory. Reads from this device always return a buffer full of zeroes. The file is infinite in length. Writes to this file are always successful, but the data written is ignored. Mapping a zero special file creates a zero-initialized, unnamed memory object of a length equal to the length of the mapping rounded up to the nearest page size as returned by sysconf. Multiple processes can share such a zero special file object provided a common ancestor mapped the object MAP_SHARED

SPARC COMPLIANCE DEFINITION 2.3

INDEX

Symbols

../mylibs/mylib.so 4-6
/dev 8-29
/dev/zero 10-1
/usr/home/me/mylibs 4-6
/usr/home/me/mylibs/mylib.so 4-6
/usr/home/me/workdir 4-6
/usr/include/netdb.h 7-6
___errno 3-8, 3-9
__div64 8-1
__dtol 8-2
__dtoll 8-2, 8-4
__dtoull 8-3
__ftoll 8-4
__ftoull 8-5, 8-13
__mul64 8-6, 8-9
__rem64 8-7, 8-10
__udiv64 8-8
__umul64 8-9
__urem64 8-10
_cleanup 3-1
_exit() 8-20
_PC_MAX_FILE_SIZE 5-3
_POSIX_PATH_MAX 8-25
_Q_lltoq 8-11
_Q_qtoll 8-12
_Q_qtoull 8-13
_Q_ulltoq 8-14

Numerics

0 (ip) 7-8
1 (icmp) 7-8
128.net.host 6-1, 7-11
17 (udp) 7-8
6 (tcp) 7-8
64-bit file offsets. 5-3

A

a 2-4
accept 7-1
address family 7-3
addseverity 3-2
AF_INET 7-6, 7-16
AIO_INPROGRESS 2-2
aio_result_t 2-1, 2-2, 2-4
aiocancel 2-1
aiocancel() 2-1
aioread 2-2
aioread() 2-2, 2-3
aiowait 2-4
aiowait() 2-4
aiowrite 2-2
aiowrite() 2-3
alarm(BA_OS) 8-19
arpa/inet.h 6-1, 7-11
ASCII 3-6
asctime_r 3-8, 3-9
Asynchronous 2-2
asynchronous 2-1, 2-3, 7-22
asynchronous errors 7-19
asynchronous I/O 2-2, 2-4
atomically 7-17

B

BA_ENV 5-4, 9-21
BA_LIB 3-5
BA_OS 2-2, 3-6, 5-4, 8-27, 8-28, 9-15, 9-24
bind 7-3
bind(3N) 7-1
bytes 7-7

C

C language 6-2, 7-12
caddr_t 5-2, 6-3, 7-14
calling thread 3-9
child process 9-4
Class A network 6-1, 7-11
Class B network 6-1, 7-11
clnt_stat rpc_broadcast_exp 6-3
close(2) 7-4, 7-19
close(BA_OS) 2-2
cond_broadcast 9-1, 9-2
cond_broadcast() 9-2
cond_destroy 9-1
cond_destroy() 9-1
cond_init 9-1
cond_init() 9-1
cond_signal 9-1
cond_signal() 9-2
cond_t 9-1
cond_timedwait 9-1, 9-2, 9-3
cond_timedwait() 9-2
cond_wait 9-1, 9-2
cond_wait(9-2
cond_wait() 9-2
condition variables 9-1
connect 7-4
connect(3N) 7-14, 7-21
connected 7-14
connected peer 7-7
connections 7-13
const char 5-2, 8-25
const int 6-3
const sigset_t 9-24
const struct timeval 2-4
const time_t 3-8
const u_long 6-3
const xdrproc_t 6-3

context switching 8-27
controlling terminal 8-18
crypt 3-3
cstime 8-19
ctermid_r 3-8
ctime_r 3-8, 3-9
cutime, 8-19

D

d.d.d.d 6-1
datagram 7-19
datagrams 7-19
debugging 7-19
decryption 3-3, 3-4
DEPPRECATED 8-28
DEPRECATED 8-28
dev/zero 8-28
diagnostic 4-4
DIR 8-25
dirent.h 8-25
dlclose 4-2, 4-3, 4-6
dlerror 4-2, 4-4
dlfcn.h 4-2, 4-4, 4-5, 4-8
dlopen 4-2, 4-3, 4-5, 4-6, 4-7, 4-8
dlopen(3X) 4-8
dlsym 4-6, 4-8
double 8-2, 8-3
DT_FINI 4-2
DT_INIT 4-6
DT_NEEDED 4-1, 4-5
dynamic linker 4-7

E

EACCES 2-1, 7-22
EADDRINUSE 7-3, 7-4
EADDRNOTAVAIL 7-3, 7-4
EAFNOSUPPORT 7-4
EAGAIN 2-3, 8-19, 9-14, 9-16, 9-18
EALREADY 7-4
EBADF 2-3, 7-1, 7-3, 7-4, 7-7, 7-10, 7-13, 7-15, 7-16, 7-19, 7-20
EBUSY 9-6, 9-9, 9-11
ECONNREFUSED 7-4, 7-13
EDEADLK 9-20
EFAULT 8-27, 9-27
EINPROGRESS 7-4
EINTR 2-4, 7-4, 7-15, 7-16, 8-20, 9-2, 9-11
EINVAL 2-1, 2-3, 2-4, 3-2, 7-3, 7-5, 7-17, 9-2, 9-6, 9-9, 9-11, 9-14, 9-16, 9-17, 9-18, 9-21, 9-24, 9-27
EISCONN 7-5
EIVAL 3-2
EMFILE 7-22
EMSGSIZE 7-16, 7-17
encrypt 3-3
Encryption 3-4
ENETUNREACH 7-5
ENODEV 7-1
ENOMEM 2-3, 7-1, 7-7, 7-10, 7-15, 7-17, 7-19, 7-20, 7-22, 8-19, 8-27, 9-14, 9-18
ENOPROTOOPT 7-19
ENOSR 7-1, 7-3, 7-5, 7-7, 7-10, 7-15, 7-17, 7-19, 7-20, 7-22
ENOSYS 3-3
ENOTCONN 7-7, 7-20
ENOTSOCK 7-2, 7-3, 7-7, 7-10, 7-13, 7-15, 7-17, 7-19, 7-20
enumeration 8-23
EOPNOTSUPP 7-2, 7-13
EOVERFLOW 5-3, 5-4
EPROTO 7-2
EPROTONOSUPPORT 7-22
ERANGE 3-9, 8-15, 8-16, 8-21, 8-22, 8-24, 8-25, 8-29
errno 5-4, 7-3
errno.h 3-8
ESRCH 9-12, 9-17, 9-20, 9-21
ETIME 9-2, 9-3
ETIMEDOUT 7-22
EWOULDBLOCK 7-2, 7-14, 7-15, 7-17
exec(BA_OS) 2-2, 8-18, 8-19
execve() 2-2
exit 9-15
exit(BA_OS) 2-2, 8-18, 8-19, 8-20, 8-27
exit(BA_OS). 8-20

F

F_FREESP 5-3
F_GETLK 5-3
F_RGETLK, 5-3
F_RSETLK 5-3
F_RSETLKW 5-3
F_SETLK 5-3
F_SETLKW 5-3
FALSE 9-3
fcntl(2) 7-14, 7-16
fcntl(BA_OS) 5-4, 8-19
fcntl.h 5-2
FdTOx 8-2, 8-3
fflush(BA_OS) 3-1
fflush(NULL) 3-1
fgetgrent 8-15
fgetgrent_r 8-15
fgetpwent 8-16
fgetpwent_r 8-16
FILE 3-8, 5-2, 8-15, 8-16
file descriptor 7-15
float 8-4, 8-5
flockfile 3-8, 3-9

fmtmsg 3-2
fork 8-18, 9-2, 9-4
fork() 8-19, 9-2, 9-4
fork1 9-4
fork1() 8-19, 9-4
fpathconf(BA_OS) 5-4
FqTOx 8-12, 8-13
fractional 8-5
fstat() 5-4
FsTOx 8-4, 8-5
full-duplex 7-20
funlockfile 3-8, 3-9
FxTOq 8-11, 8-14

G

getc_unclocked 3-8
getc_unlocked 3-9
getchar_unclocked 3-8
getchar_unlocked 3-9
getcontext 8-27
getcontext(BA_OS) 8-27
getgrent 8-21
getgrent_r 8-21, 8-29
getgrgid_r 3-9, 8-25
getgrnam_r 8-25
gethostbyaddr 7-6
gethostbyname 7-6
getitimer(RT_OS) 8-19
getlogin 8-22
getlogin_r 8-22
getopt 3-5
getpeername 7-7
getprotobyname 7-8
getprotobyname(3N) 7-18
getprotobynumber 7-8
getprotoent 7-8
getpwent_r 8-23, 8-24
getpwent_r() 8-23
getpwnam_r 8-22, 8-25
getpwuid_r 8-22, 8-25
getrlimit(BA_OS) 5-4, 8-18, 8-19
getservbyname 7-9
getservbyport 7-9
getsockname 7-10
getsockopt 7-18
getsockopt(3N) 7-22
gid_t 8-15, 8-16, 8-21, 8-23
global errno 7-3
gmtime 3-9
gmtime_r 3-8, 3-9
grp.h 8-15, 8-21, 8-25

H

hardware-specific serial number 3-6
host entry 7-6
HOST_NOT_FOUND 7-6
hostaddress 7-6
hostname 3-6, 7-6
hosts 7-6

I

I/O operation 2-2
icmp 7-8
ID 9-13, 9-20, 9-23
in_addr 6-1
inet_addr 6-1, 6-2
inet_lnaof 7-11, 7-12
inet_makeaddr 7-11
inet_netof 6-1, 6-2
inet_network 7-11, 7-12
inet_ntoa 6-1, 6-2
INT_MAX 5-3
int64_t 5-2, 5-3
Internet address 6-1, 7-11
INTERNET ADDRESSES 6-1, 7-11
ioctl(2) 7-22
it_interval 8-19
it_value 8-19
ITIMER_REAL 8-19

K

KE_OS 5-4, 9-5, 9-8, 9-10
keyloc 9-19

L

l_linger 7-18
l_onoff 7-18
LD_BIND_NOW 4-5
LD_LIBRARY_PATH 4-6
lf_fcntl 5-2, 5-3
lf_fpathconf 5-2
lf_fseek 5-2
lf_fstat 5-2, 5-3
lf_fstatvfs 5-2, 5-3
lf_ftell 5-2
lf_getrlimit 5-2, 5-3
lf_lseek 5-2
lf_lstat 5-2, 5-3
lf_mmap 5-2
lf_off_t 5-2, 5-3
lf_pathconf 5-2, 5-3
lf_setrlimit 5-3
lf_setrlimitlf_statvfs 5-2
lf_stat 5-2, 5-3
lf_statvfs 5-2, 5-3
limits(BA_ENV) 5-4
limits.h 8-22, 8-29
listen 7-13

listen(3N 7-1
localtime_r 3-8, 3-9
locks 9-5
login 8-17, 8-24
LOGNAME_MAX 8-22
long 3-6
long double 8-11, 8-12, 8-13, 8-14
long long 8-1, 8-2, 8-3, 8-4, 8-5, 8-6, 8-9, 8-10, 8-11, 8-12, 8-14
lseek() 5-4
lseek(BA_OS) 2-2, 5-4
lstat() 5-4
LWP 9-13

M

main 9-15
main thread 3-9
makecontext 8-27
malloc 8-28
MAP_SHARED 10-1
memcntl(RT_OS) 8-19
Memory 2-3
memory 7-22
memory allocation 8-28
memory mappings 8-18
memory segments 8-18
mmap 8-28
mmap(KE_OS 5-4, 9-1
mmap(KE_OS) 8-18, 8-19, 9-5, 9-8, 9-10
MSG_DONTROUTE 7-16
msg_iov 7-15
msg_name 7-15
MSG_OOB 7-14, 7-16, 7-19
MSG_PEEK 7-14
Multiple processes 10-1
multiple protocol levels 7-18
multiple threads 9-7
multi-threaded 8-19
multithreaded applications 8-21, 8-23
multi-threaded process 8-20
multithreading 3-8, 3-9, 8-25
mutex 9-1, 9-5, 9-6, 9-7
mutex_destroy 9-5
mutex_init 9-5, 9-6, 9-7
mutex_lock 9-2, 9-3, 9-5, 9-6, 9-7, 9-19
mutex_t 9-1, 9-5, 9-7, 9-19
mutex_trylock 9-5, 9-6
mutex_unlock 9-2, 9-5, 9-6, 9-7, 9-19
Mutexes 9-5
mutexes 9-6
Mutual exclusion 9-5
mylib.so 4-6

N

name space 7-3
namelen 7-7
netconfig(4) 7-1
netdb.h 7-6, 7-8, 7-9
netinet/in.h 6-1, 7-11
network is not reachable 7-5
new_level 9-16
new_mask 9-14
nice(KE_OS) 8-18, 8-19
NO_ADDRESS 7-6
NO_DATA 7-6
NO_RECOVERY 7-6
non- daemon threads 9-13
non-blocking 7-1, 7-2, 7-4, 7-17
non-blocking I/O 7-22
non-NULL 9-18
non-zero 3-5, 9-11
normative references 1-1
nsigned long long 8-8
NUL 8-24
NULL 2-4, 3-9, 4-4, 4-6, 4-8, 7-8, 7-14, 7-15, 8-16, 8-23, 8-25, 9-3, 9-13, 9-14, 9-18, 9-19, 9-20, 9-24
NULL, 7-9, 9-20

O

o ERANG 8-24
off_t 2-2
orig_mask 9-14
oucp 8-27

P

password 8-23
password database 8-23
pcontext 8-27
peer 7-7
PF_INET 7-21
pfmt.h 3-5
pid_t 8-18, 8-19, 9-4
plock(KE_OS) 8-19
poll 7-16
poll(2) 7-1
poll(3C) 7-4
POSIX 1003.1c 8-19, 8-22, 8-26, 8-29, 9-27
POSIX_PATH_MAX 8-29
priocntl(RT_OS) 8-18, 8-19
priority 9-17
Procedure Call domain name 3-7
process 9-4
process group ID 8-18
process ID 9-4
protocol 7-2
protocol name 7-9

protoent 7-8
ptrace(KE_OS) 8-19
putc_unclocked 3-8
putc_unlocked 3-9
putchar_unlocked 3-8, 3-9
pwd.h 8-16, 8-25

Q

quad precision 8-11
quad precision value 8-14

R

rand_r 3-8, 8-25
read() 2-2
read(2) 7-15, 7-21
read(BA_OS) 2-2
readdir 8-25
readdir_r 8-25
readers/writer 9-8
readers/writer lock 9-8
Read-Only Memory 3-6
recv 7-14
recv(3N) 7-16, 7-21
recvfrom 7-14
recvmsg 7-14
reentrant 3-9, 8-25
resultproc_t 6-3
rpc/rpc.h 6-3
rpc_broadcast 6-3
rpc_broadcast_exp 6-3
RTLD_LAZY 4-5, 4-6
RTLD_NOW 4-5
rw_rdlock 9-8, 9-9
rw_tryrdlock 9-8, 9-9
rw_trywrlock 9-8, 9-9
rw_unlock 9-8, 9-9
rw_wrlock 9-8, 9-9
rwlock_destroy 9-8
rwlock_init 9-8
rwlock_t 9-8
rwlp 9-9

S

sbrk 8-28
sbrk(0) 8-28
SCD 2.1 3-7
SCD 2.3 1-1, 3-7
SCD2.3 8-19
SCD-conforming 4-2
scheduler class 8-18
sema_destroy 9-10
sema_init 9-10
sema_post 9-10, 9-11
sema_t 9-10
sema_trywait 9-10, 9-11
sema_wait 9-10, 9-11
semadj 8-19
semaphore 9-10
semop(KE_OS) 8-19
send 7-16
send(3N) 7-21
sendmsg 7-16
sendtol 7-16
setcontext 8-27
set-group-ID mode bit 8-18
setkey 3-3
setlabel 3-5
setlabel() 3-5
setsockopt 7-18
set-user-ID mode bit 8-18
shared object 4-2, 4-5, 4-8
shm_nattach 8-18
shmop(KE_OS) 8-18, 8-19
shutdown 7-20
SI_ARCHITECTURE 3-6
SI_HOSTNAME 3-6
SI_HW_PROVIDER 3-6, 3-7
SI_HW_SERIAL 3-6, 3-7
SI_MACHINE 3-6
SI_RELEASE 3-6
SI_SRPC_DOMAIN 3-7
SI_SYSNAME 3-6
SI_VERSION 3-6
SIG_BLOCK 9-24
SIG_DFL 8-18
SIG_HOLD 8-18
SIG_IGN 8-18
SIG_SETMASK 9-14, 9-24
SIG_UNBLOCK 9-24
sigaction 9-24, 9-27
sigaction(BA_OS) 8-27
sigdelset 9-14
sigfillset 9-14
SIGINT 9-14
SIGIO 2-1, 2-2, 2-4
signal 7-22, 9-2, 9-14
signal mask 9-24, 9-27
signal(BA_ENV) 9-21
signal(BA_OS) 8-19
signal.h 9-21, 9-24, 9-27
SIGPIPE 7-19, 7-22
SIGPOLL 7-22
sigprocmask(BA_OS) 8-27
SIGSEGV 9-14
sigset_t 9-14, 9-24, 9-27
SIGURG 7-22
sigwait 9-27
SIGWAITING 9-24

single precision 8-5
size_t 5-2, 8-22
SO_BROADCAST 7-18, 7-19
SO_DEBUG 7-18, 7-19
SO_DONTROUTE 7-16, 7-18, 7-19
SO_ERROR 7-19
SO_KEEPALIVE 7-18, 7-19
SO_LINGER 7-18, 7-19
SO_OOBINLINE 7-18, 7-19
SO_RCVBUF 7-19
SO_REUSEADDR 7-18, 7-19
SO_SNDBUF 7-19
SO_TYPE 7-19
SOCK_DGRAM 7-4, 7-21, 7-22
SOCK_SEQPACKET 7-13, 7-21, 7-22
SOCK_STREAM 7-1, 7-2, 7-4, 7-13, 7-16, 7-19, 7-21, 7-22
socket 7-1, 7-4, 7-7, 7-10, 7-13, 7-16, 7-19, 7-20, 7-21, 7-22
socket level 7-22
socket(3N) 7-1, 7-3, 7-4, 7-14, 7-16
sockets 7-18, 7-19
SOL_SOCKET 7-18
SPARC 5-1
SPARC Architecture 5-3
stack_size 9-14
stat() 5-4
stat(BA_OS) 5-4
statvfs(BA_OS) 5-4
stdio.h 3-8
stdlib.h 3-8, 8-22, 8-29
stime 8-19
stream 3-9
STREAMS 7-1, 7-2, 7-3, 7-4, 7-5, 7-7, 7-10, 7-15, 7-17, 7-19, 7-20, 7-22
string.h 3-8
strtok_r 3-8
struct dirent 8-25
struct group 8-15, 8-21, 8-25
struct hostent 7-6
struct in_addr 6-1, 7-6, 7-11
struct iovec 7-14
struct lf_rlimit 5-2, 5-3
struct lf_stat 5-2, 5-3
struct lf_statvfs 5-2, 5-3
struct msghdr 7-14, 7-16
struct passwd 8-16, 8-23, 8-25
struct protoent 7-8
struct servent 7-9
struct sockaddr 7-1, 7-3, 7-7, 7-10, 7-14, 7-16
struct tm 3-8
swapcontext 8-27
synch.h 9-1, 9-5, 9-8, 9-10
synchronize threads 9-1
synchronous 2-4
sys/asynch.h 2-1, 2-2, 2-4
sys/fstatvfs.h 5-2
sys/mman.h 5-2
sys/resource.h 5-2
sys/socket.h 6-1, 7-1, 7-3, 7-4, 7-6, 7-11, 7-14, 7-16, 7-18, 7-21, 7-22
sys/sockets.h 7-10, 7-13
sys/stat.h 5-2, 5-3
sys/statvfs.h 5-3
sys/systeminfo.h 3-6
sys/time.h 2-4, 5-2
sys/types.h 5-2, 5-3, 6-1, 7-1, 7-3, 7-4, 7-6, 7-10, 7-11, 7-13, 7-14, 7-16, 7-18, 7-21, 8-18, 9-4
sys/uio.h 7-14
sysinfo 3-6
system calls 8-20
system information 3-6
system(BA_OS) 8-19

T

target_thread 9-12, 9-17, 9-21
TCP 7-18
tcp 7-8, 7-9
TCP protocol 7-18
tell 5-2, 5-3
THR_BOUND 9-13
thr_continue 9-12, 9-13
thr_create 8-19, 9-13, 9-14, 9-15, 9-16, 9-20
thr_create) 9-16
THR_DAEMON 9-13
THR_DETACHED 9-13
thr_exit 9-13, 9-14, 9-15
thr_getconcurrency 9-16
thr_getprio 9-17
thr_getspecific 9-18, 9-19
thr_join 9-14, 9-15, 9-20
thr_keycreate 9-15, 9-18, 9-19
thr_kill 9-21
thr_kill() 9-21
thr_main 9-25
thr_min_stack 9-13, 9-14, 9-22
THR_NEW_LWP 9-13, 9-16
thr_self 9-23
thr_self() 9-23
thr_setconcurrency 9-13, 9-16
thr_setprio 9-17
thr_setspecifi 9-19
thr_setspecific 9-18
thr_sigsetmask 9-14, 9-24
thr_suspend 9-12
THR_SUSPENDED 9-13
thr_yield 9-26
thread 9-13, 9-15, 9-16, 9-20, 9-21, 9-22, 9-23, 9-24,

9-25, 9-27
thread.h 9-3, 9-7, 9-9, 9-11, 9-12, 9-13, 9-15, 9-16, 9-17, 9-18, 9-20, 9-21, 9-22, 9-23, 9-24, 9-25, 9-26
thread_key_t 9-18, 9-19
thread_t 9-12, 9-13, 9-14, 9-17, 9-20, 9-21, 9-23
thread's ID 9-15
threads 9-3, 9-10, 9-16, 9-17
thread-specific 9-18
time.h 3-8
time_t 3-8
TIMEOUT 9-3
timeout 6-3
times(BA_OS) 8-19
timestruc_t 9-1, 9-3
timeva 2-4
TLD_LAZY 4-8
tms 8-19
tms_utime 8-19
TRY_AGAIN 7-6
ttyname 8-29
ttyname() 8-29
ttyname_r 8-29
tv_sec 2-4
tv_usec 2-4

U

u_long 6-3
ucontext.h 8-27
ucontext_t 8-27
ucp 8-27
udp 7-8, 7-9
uid_t 8-16, 8-23, 8-25
uint64_t 5-3
umask(BA_OS) 8-18, 8-19
uname 3-6
unblock 9-24
unistd.h 8-18, 8-28, 9-4
UNIX 3-6
UNIX_SV 3-6
unnamed socket 7-3
unsigne 8-14
Unsigned 64 bit 8-9
unsigned int 3-8
unsigned long 6-1
unsigned long long 8-3, 8-5, 8-9, 8-10, 8-13, 8-14
USYNC_PROCESS 9-1, 9-5, 9-8, 9-10
USYNC_THREAD 9-1, 9-5, 9-7, 9-8, 9-10

W

wait(BA_OS) 8-19
waittime 6-3
write() 2-2
write(2) 7-21
write(BA_OS) 2-2

X

xdrproc_t 6-3