NAME

pthread_create – create a new thread / pthread_exit – terminate the calling thread

SYNOPSIS

#include <pthread.h>

int pthread_create(pthread_t * *thread*, **pthread_attr_t** * *attr*, **void** * (**start_routine*)(**void** *), **void** * *arg*);

void pthread_exit(void **retval*);

DESCRIPTION

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function *start_routine* passing it *arg* as first argument. The new thread terminates either explicitly, by calling **pthread_exit**(3), or implicitly, by returning from the *start_routine* function. The latter case is equivalent to calling **pthread_exit**(3) with the result returned by *start_routine* as exit code.

The *attr* argument specifies thread attributes to be applied to the new thread. See **pthread_attr_init**(3) for a complete list of thread attributes. The *attr* argument can also be **NULL**, in which case default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

pthread_exit terminates the execution of the calling thread. All cleanup handlers that have been set for the calling thread with **pthread_cleanup_push**(3) are executed in reverse order (the most recently pushed handler is executed first). Finalization functions for thread-specific data are then called for all keys that have non- **NULL** values associated with them in the calling thread (see **pthread_key_create**(3)). Finally, execution of the calling thread is stopped.

The *retval* argument is the return value of the thread. It can be consulted from another thread using **pthread_join**(3).

RETURN VALUE

On success, the identifier of the newly created thread is stored in the location pointed by the *thread* argument, and a 0 is returned. On error, a non-zero error code is returned.

The **pthread_exit** function never returns.

ERRORS

EAGAIN
not enough system resources to create a process for the new thread.

EAGAIN
more than **PTHREAD_THREADS_MAX** threads are already active.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_join(3), **pthread_detach**(3), **pthread_attr_init**(3).

NAME

pthread_join – join with a terminated thread

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t *thread*, **void** ***retval*);

Compile and link with *–pthread.*

DESCRIPTION

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to **pthread_exit**(3)) into the location pointed to by *retval*. If the target thread was canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE

On success, **pthread_join**() returns 0; on error, it returns an error number.

ERRORS

EDEADLK
A deadlock was detected (e.g., two threads tried to join with each other); or *thread* specifies the calling thread.

EINVAL
thread is not a joinable thread.

EINVAL
Another thread is already waiting to join with this thread.

ESRCH
No thread with the ID *thread* could be found.

NOTES

After a successful call to **pthread_join**(), the caller is guaranteed that the target thread has terminated. The caller may then choose to do any clean-up that is required after termination of the thread (e.g., freeing memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thread". Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

There is no pthreads analog of *waitpid(-1, &status, 0)*, that is, "join with any terminated thread". If you believe you need this functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with any other thread in the process.

EXAMPLE

See **pthread_create**(3).

SEE ALSO

pthread_cancel(3), **pthread_create**(3), **pthread_detach**(3), **pthread_exit**(3), **pthreads**(7)

NAME

scanf, fscanf, sscanf – input format conversion

SYNOPSIS

#include <stdio.h>

int scanf(const char * *format*, **...);**
int fscanf(FILE * *stream*, **const char *** *format*, **...);**
int sscanf(const char * *str*, **const char *** *format*, **...);**

DESCRIPTION

The **scanf**() family of functions scans input according to *format* as described below. This format may contain *conversion specifications*; the results from such conversions, if any, are stored in the locations pointed to by the *pointer* arguments that follow *format*. Each *pointer* argument must be of a type that is appropriate for the value returned by the corresponding conversion specification.

If the number of conversion specifications in *format* exceeds the number of *pointer* arguments, the results are undefined. If the number of *pointer* arguments exceeds the number of conversion specifications, then the excess *pointer* arguments are evaluated, but are otherwise ignored.

The **scanf**() function reads input from the standard input stream *stdin*, **fscanf**() reads input from the stream pointer *stream*, and **sscanf**() reads its input from the character string pointed to by *str*.

The *format* string consists of a sequence of *directives* which describe how to process the sequence of input characters. If processing of a directive fails, no further input is read, and **scanf**() returns. A "failure" can be either of the following: *input failure*, meaning that input characters were unavailable, or *matching failure*, meaning that the input was inappropriate (see below).

A directive is one of the following:

- A sequence of white-space characters (space, tab, newline, etc.; see **isspace**(3)). This directive matches any amount of white space, including none, in the input.
- An ordinary character (i.e., one other than white space or '%'). This character must exactly match the next character of input.
- A conversion specification, which commences with a '%' (percent) character. A sequence of characters from the input is converted according to this specification, and the result is placed in the corresponding *pointer* argument. If the next item of input does not match the conversion specification, the conversion fails—this is a *matching failure*.

Each *conversion specification* in *format* begins with either the character '%' or the character sequence "**%***n***$**" (see below for the distinction) followed by:

- An optional '*' assignment-suppression character: **scanf**() reads input as directed by the conversion specification, but discards the input. No corresponding *pointer* argument is required, and this specification is not included in the count of successful assignments returned by **scanf**().
- For decimal conversions, an optional quote character ('). This specifies that the input number may include thousands' separators as defined by the **LC_NUMERIC** category of the current locale. (See **setlocale**(3).) The quote character may precede or follow the '*' assignment-suppression character.
- An optional decimal integer which specifies the *maximum field width*. Reading of characters stops either when this maximum is reached or when a nonmatching character is found, whichever happens first. Most conversions discard initial white space characters (the exceptions are noted below), and these discarded characters don't count toward the maximum field width. String input conversions store a terminating null byte ('\0') to mark the end of the input; the maximum field width does not include this terminator.
- An optional *type modifier character*. For example, the **l** type modifier is used with integer conversions such as **%d** to specify that the corresponding *pointer* argument refers to a *long int* rather than a pointer to an *int*.

- A *conversion specifier* that specifies the type of input conversion to be performed.

The conversion specifications in *format* are of two forms, either beginning with '%' or beginning with "**%***n***$**". The two forms should not be mixed in the same *format* string, except that a string containing "**%***n***$**" specifications can include **%%** and **%***. If *format* contains '%' specifications, then these correspond in order with successive *pointer* arguments. In the "**%***n***$**" form (which is specified in POSIX.1-2001, but not C99), *n* is a decimal integer that specifies that the converted input should be placed in the location referred to by the *n*-th *pointer* argument following *format*.

Conversions

The following *type modifier characters* can appear in a conversion specification:

l Indicates either that the conversion will be one of **d**, **i**, **o**, **u**, **x**, **X**, or **n** and the next pointer is a pointer to a *long int* or *unsigned long int* (rather than *int*), or that the conversion will be one of **e**, **f**, or **g** and the next pointer is a pointer to *double* (rather than *float*). Specifying two **l** characters is equivalent to **L**. If used with **%c** or **%s**, the corresponding parameter is considered as a pointer to a wide character or wide-character string respectively.

L Indicates that the conversion will be either **e**, **f**, or **g** and the next pointer is a pointer to *long double* or the conversion will be **d**, **i**, **o**, **u**, or **x** and the next pointer is a pointer to *long long*.

The following *conversion specifiers* are available:

% Matches a literal '%'. That is, **%%** in the format string matches a single input '%' character. No conversion is done (but initial white space characters are discarded), and assignment does not occur.

d Matches an optionally signed decimal integer; the next pointer must be a pointer to *int*.

i Matches an optionally signed integer; the next pointer must be a pointer to *int*. The integer is read in base 16 if it begins with *0x* or *0X*, in base 8 if it begins with *0*, and in base 10 otherwise. Only characters that correspond to the base are used.

u Matches an unsigned decimal integer; the next pointer must be a pointer to *unsigned int*.

x Matches an unsigned hexadecimal integer; the next pointer must be a pointer to *unsigned int*.

f Matches an optionally signed floating-point number; the next pointer must be a pointer to *float*.

s Matches a sequence of non-white-space characters; the next pointer must be a pointer to the initial element of a character array that is long enough to hold the input sequence and the terminating null byte ('\0'), which is added automatically. The input string stops at white space or at the maximum field width, whichever occurs first.

c Matches a sequence of characters whose length is specified by the *maximum field width* (default 1); the next pointer must be a pointer to *char*, and there must be enough room for all the characters (no terminating null byte is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.

p Matches a pointer value (as printed by **%p** in **printf**(3); the next pointer must be a pointer to a pointer to *void*.

RETURN VALUE

On success, these functions return the number of input items successfully matched and assigned; this can be fewer than provided for, or even zero, in the event of an early matching failure.

The value **EOF** is returned if the end of input is reached before either the first successful conversion or a matching failure occurs. **EOF** is also returned if a read error occurs, in which case the error indicator for the stream (see **ferror**(3)) is set, and *errno* is set to indicate the error.

NAME

sem_destroy – destroy an unnamed semaphore

LIBRARY

POSIX threads library (*libpthread*, *–lpthread*)

SYNOPSIS

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

DESCRIPTION

sem_destroy() destroys the unnamed semaphore at the address pointed to by *sem*.

Only a semaphore that has been initialized by **sem_init**(3) should be destroyed using **sem_destroy**().

Destroying a semaphore that other processes or threads are currently blocked on (in **sem_wait**(3)) produces undefined behavior.

Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized using **sem_init**(3).

RETURN VALUE

sem_destroy() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
sem is not a valid semaphore.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
sem_destroy()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2001, POSIX.1-2008.

NOTES

An unnamed semaphore should be destroyed with **sem_destroy**() before the memory in which it is located is deallocated. Failure to do this can result in resource leaks on some implementations.

SEE ALSO

sem_init(3), **sem_post**(3), **sem_wait**(3), **sem_overview**(7)

NAME

sem_init – initialize an unnamed semaphore

SYNOPSIS

```
#include <semaphore.h>

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

DESCRIPTION

sem_init() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.

The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.

If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

If *pshared* is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory (see **shm_open**(3), **mmap**(2), and **shmget**(2)). (Since a child created by **fork**(2) inherits its parent's memory mappings, it can also access the semaphore.) Any process that can access the shared memory region can operate on the semaphore using **sem_post**(3), **sem_wait**(3), and so on.

Initializing a semaphore that has already been initialized results in undefined behavior.

RETURN VALUE

sem_init() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
value exceeds **SEM_VALUE_MAX**.

ENOSYS
pshared is nonzero, but the system does not support process-shared semaphores (see **sem_overview**(7)).

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
sem_init()	Thread safety	MT-Safe

SEE ALSO

sem_destroy(3), **sem_post**(3), **sem_wait**(3), **sem_overview**(7)

NAME

sem_post – unlock a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_post(sem_t *_sem_);

DESCRIPTION

sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a **sem_wait**(3) call will be woken up and proceed to lock the semaphore.

RETURN VALUE

sem_post() returns 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
sem is not a valid semaphore.

EOVERFLOW
The maximum allowable value for a semaphore would be exceeded.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
sem_post()	Thread safety	MT-Safe

SEE ALSO

sem_getvalue(3), **sem_wait**(3), **sem_overview**(7), **signal−safety**(7)

NAME

sem_wait, sem_timedwait, sem_trywait – lock a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_wait(sem_t *_sem_);
int sem_trywait(sem_t *_sem_);
int sem_timedwait(sem_t *restrict *sem*,
const struct timespec *restrict *abs_timeout*);

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

sem_timedwait():
_POSIX_C_SOURCE >= 200112L

DESCRIPTION

sem_wait() decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as **sem_wait**(), except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking.

sem_timedwait() is the same as **sem_wait**(), except that *abs_timeout* specifies a limit on the amount of time that the call should block if the decrement cannot be immediately performed. The *abs_timeout* argument points to a **timespec**(3) structure that specifies an absolute timeout in seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

If the timeout has already expired by the time of the call, and the semaphore could not be locked immediately, then **sem_timedwait**() fails with a timeout error (*errno* set to **ETIMEDOUT**).

If the operation can be performed immediately, then **sem_timedwait**() never fails with a timeout error, regardless of the value of *abs_timeout*. Furthermore, the validity of *abs_timeout* is not checked in this case.

RETURN VALUE

All of these functions return 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EAGAIN
(**sem_trywait**()) The operation could not be performed without blocking (i.e., the semaphore currently has the value zero).

EINTR
The call was interrupted by a signal handler; see **signal**(7).

EINVAL
sem is not a valid semaphore.

EINVAL
(**sem_timedwait**()) The value of *abs_timeout.tv_nsecs* is less than 0, or greater than or equal to 1000 million.

ETIMEDOUT
(**sem_timedwait**()) The call timed out before the semaphore could be locked.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
sem_wait(), **sem_trywait**(), **sem_timedwait**()	Thread safety	MT-Safe

SEE ALSO

clock_gettime(2), **sem_getvalue**(3), **sem_post**(3), **timespec**(3), **sem_overview**(7), **time**(7)