

NAME

pthread_create – create a new thread

LIBRARY

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

SYNOPSIS

```
#include <pthread.h>
```

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

DESCRIPTION

The **pthread_create()** function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine()*; *arg* is passed as the sole argument of *start_routine()*.

The new thread terminates in one of the following ways:

- It calls **pthread_exit(3)**, specifying an exit status value that is available to another thread in the same process that calls **pthread_join(3)**.
- It returns from *start_routine()*. This is equivalent to calling **pthread_exit(3)** with the value supplied in the *return* statement.
- It is canceled (see **pthread_cancel(3)**).
- Any of the threads in the process calls **exit(3)**, or the main thread performs a return from *main()*. This causes the termination of all threads in the process.

The *attr* argument points to a *pthread_attr_t* structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using **pthread_attr_init(3)** and related functions. If *attr* is NULL, then the thread is created with default attributes.

Before returning, a successful call to **pthread_create()** stores the ID of the new thread in the buffer pointed to by *thread*; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

The new thread inherits a copy of the creating thread's signal mask (**pthread_sigmask(3)**). The set of pending signals for the new thread is empty (**sigpending(2)**). The new thread does not inherit the creating thread's alternate signal stack (**sigaltstack(2)**).

The new thread inherits the calling thread's floating-point environment (**fenv(3)**).

The initial value of the new thread's CPU-time clock is 0 (see **pthread_getcpuclockid(3)**).

Linux-specific details

The new thread inherits copies of the calling thread's capability sets (see **capabilities(7)**) and CPU affinity mask (see **sched_setaffinity(2)**).

RETURN VALUE

On success, **pthread_create()** returns 0; on error, it returns an error number, and the contents of **thread* are undefined.

ERRORS**EAGAIN**

Insufficient resources to create another thread.

EAGAIN

A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error: the **RLIMIT_NPROC** soft resource limit (set via **setrlimit(2)**), which limits the number of processes and threads for a real user ID, was reached; the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads-max*, was reached (see **proc(5)**); or the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see **proc(5)**).

EINVAL

Invalid settings in *attr*.

EPERM

No permission to set the scheduling policy and parameters specified in *attr*.

ATTRIBUTES

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

Interface	Attribute	Value
pthread_create()	Thread safety	MT-Safe

STANDARDS

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

NOTES

See **pthread_self(3)** for further information on the thread ID returned in **thread* by **pthread_create()**. Unless real-time scheduling policies are being employed, after a call to **pthread_create()**, it is indeterminate which thread—the caller or the new thread—will next execute.

A thread may either be *joinable* or *detached*. If a thread is joinable, then another thread can call **pthread_join(3)** to wait for the thread to terminate and fetch its exit status. Only when a terminated joinable thread has been joined are the last of its resources released back to the system. When a detached thread terminates, its resources are automatically released back to the system: it is not possible to join with the thread in order to obtain its exit status. Making a thread detached is useful for some types of daemon threads whose exit status the application does not need to care about. By default, a new thread is created in a joinable state, unless *attr* was set to create the thread in a detached state (using **pthread_attr_setdetachstate(3)**).

Under the NPTL threading implementation, if the **RLIMIT_STACK** soft resource limit *at the time the program started* has any value other than "unlimited", then it determines the default stack size of new threads. Using **pthread_attr_setstacksize(3)**, the stack size attribute can be explicitly set in the *attr* argument used to create a thread, in order to obtain a stack size other than the default. If the **RLIMIT_STACK** resource limit is set to "unlimited", a per-architecture value is used for the stack size. Here is the value for a few architectures:

Architecture	Default stack size
i386	2 MB
IA-64	32 MB
PowerPC	4 MB
S/390	2 MB
Sparc-32	2 MB
Sparc-64	4 MB
x86_64	2 MB

BUGS

In the obsolete LinuxThreads implementation, each of the threads in a process has a different process ID. This is in violation of the POSIX threads specification, and is the source of many other nonconformances to the standard; see **pthreads(7)**.

EXAMPLES

The program below demonstrates the use of **pthread_create()**, as well as a number of other functions in the pthreads API.

In the following run, on a system providing the NPTL threading implementation, the stack size defaults to the value given by the "stack size" resource limit:

```
$ ulimit -s
8192                # The stack size limit is 8 MB (0x800000 bytes)
```

```
$ ./a.out hola salut servus
Thread 1: top of stack near 0xb7dd03b8; argv_string=hola
Thread 2: top of stack near 0xb75cf3b8; argv_string=salut
Thread 3: top of stack near 0xb6dce3b8; argv_string=servus
Joined with thread 1; returned value was HOLA
Joined with thread 2; returned value was SALUT
Joined with thread 3; returned value was SERVUS
```

In the next run, the program explicitly sets a stack size of 1 MB (using `pthread_attr_setstacksize(3)`) for the created threads:

```
$ ./a.out -s 0x100000 hola salut servus
Thread 1: top of stack near 0xb7d723b8; argv_string=hola
Thread 2: top of stack near 0xb7c713b8; argv_string=salut
Thread 3: top of stack near 0xb7b703b8; argv_string=servus
Joined with thread 1; returned value was HOLA
Joined with thread 2; returned value was SALUT
Joined with thread 3; returned value was SERVUS
```

Program source

```
#include <ctype.h>
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
    do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

struct thread_info { /* Used as argument to thread_start() */
    pthread_t thread_id; /* ID returned by pthread_create() */
    int thread_num; /* Application-defined thread # */
    char *argv_string; /* From command-line argument */
};

/* Thread start function: display address near top of our stack,
   and return upper-cased copy of argv_string. */

static void *
thread_start(void *arg)
{
    struct thread_info *tinfo = arg;
    char *uargv;

    printf("Thread %d: top of stack near %p; argv_string=%s\n",
           tinfo->thread_num, (void *) &tinfo, tinfo->argv_string);

    uargv = strdup(tinfo->argv_string);
    if (uargv == NULL)
        handle_error("strdup");
}
```

```

        for (char *p = uargv; *p != '\0'; p++)
            *p = toupper(*p);

    return uargv;
}

int
main(int argc, char *argv[])
{
    int                s, opt;
    void               *res;
    size_t             num_threads;
    ssize_t            stack_size;
    pthread_attr_t      attr;
    struct thread_info *tinfo;

    /* The "-s" option specifies a stack size for our threads. */

    stack_size = -1;
    while ((opt = getopt(argc, argv, "s:")) != -1) {
        switch (opt) {
            case 's':
                stack_size = strtoul(optarg, NULL, 0);
                break;

            default:
                fprintf(stderr, "Usage: %s [-s stack-size] arg...\n",
                    argv[0]);
                exit(EXIT_FAILURE);
        }
    }

    num_threads = argc - optind;

    /* Initialize thread creation attributes. */

    s = pthread_attr_init(&attr);
    if (s != 0)
        handle_error_en(s, "pthread_attr_init");

    if (stack_size > 0) {
        s = pthread_attr_setstacksize(&attr, stack_size);
        if (s != 0)
            handle_error_en(s, "pthread_attr_setstacksize");
    }

    /* Allocate memory for pthread_create() arguments. */

    tinfo = calloc(num_threads, sizeof(*tinfo));
    if (tinfo == NULL)
        handle_error("calloc");

    /* Create one thread for each command-line argument. */

```

```

for (size_t tnum = 0; tnum < num_threads; tnum++) {
    tinfo[tnum].thread_num = tnum + 1;
    tinfo[tnum].argv_string = argv[optind + tnum];

    /* The pthread_create() call stores the thread ID into
       corresponding element of tinfo[]. */

    s = pthread_create(&tinfo[tnum].thread_id, &attr,
                      &thread_start, &tinfo[tnum]);
    if (s != 0)
        handle_error_en(s, "pthread_create");
}

/* Destroy the thread attributes object, since it is no
   longer needed. */

s = pthread_attr_destroy(&attr);
if (s != 0)
    handle_error_en(s, "pthread_attr_destroy");

/* Now join with each thread, and display its returned value. */

for (size_t tnum = 0; tnum < num_threads; tnum++) {
    s = pthread_join(tinfo[tnum].thread_id, &res);
    if (s != 0)
        handle_error_en(s, "pthread_join");

    printf("Joined with thread %d; returned value was %s\n",
           tinfo[tnum].thread_num, (char *) res);
    free(res);      /* Free memory allocated by thread */
}

free(tinfo);
exit(EXIT_SUCCESS);
}

```

SEE ALSO

getrlimit(2), pthread_attr_init(3), pthread_cancel(3), pthread_detach(3), pthread_equal(3), pthread_exit(3), pthread_getattr_np(3), pthread_join(3), pthread_self(3), pthread_setattr_default_np(3), pthreads(7)