Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'pthread_create.3p' command

$ man pthread_create.3p

PTHREAD_CREATE(3P) POSIX Programmer's Manual PTHREAD_CREATE(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

pthread_create ? thread creation

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 shall create a new thread, with at? tributes specified by attr, within a process. If attr is NULL, the de? fault attributes shall be used. If the attributes specified by attr are modified later, the thread's attributes shall not be affected. Upon successful completion, pthread_create() shall store the ID of the cre? ated thread in the location referenced by thread.

The thread is created executing start_routine with arg as its sole ar? gument. If the start_routine returns, the effect shall be as if there was an implicit call to pthread_exit() using the return value of

start_routine as the exit status. Note that the thread in which main() was originally invoked differs from this. When it returns from main(), the effect shall be as if there was an implicit call to exit() using the return value of main() as the exit status.

The signal state of the new thread shall be initialized as follows:

* The signal mask shall be inherited from the creating thread.

* The set of signals pending for the new thread shall be empty.

The thread-local current locale and the alternate stack shall not be inherited.

The floating-point environment shall be inherited from the creating thread.

If pthread_create() fails, no new thread is created and the contents of the location referenced by thread are undefined.

If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero.

The behavior is undefined if the value specified by the attr argument to pthread_create() does not refer to an initialized thread attributes object.

RETURN VALUE

If successful, the pthread_create() function shall return zero; other? wise, an error number shall be returned to indicate the error.

ERRORS

The pthread_create() function shall fail if:

EAGAIN The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process {PTHREAD_THREADS_MAX} would be exceeded.

EPERM The caller does not have appropriate privileges to set the re? quired scheduling parameters or scheduling policy.

The pthread_create() function shall not return an error code of [EINTR].

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

There is no requirement on the implementation that the ID of the cre? ated thread be available before the newly created thread starts execut? ing. The calling thread can obtain the ID of the created thread through the thread argument of the pthread_create() function, and the newly created thread can obtain its ID by a call to pthread_self().

RATIONALE

A suggested alternative to pthread_create() would be to define two sep? arate operations: create and start. Some applications would find such behavior more natural. Ada, in particular, separates the ``creation'' of a task from its ``activation''.

Splitting the operation was rejected by the standard developers for many reasons:

* The number of calls required to start a thread would increase from one to two and thus place an additional burden on applications that do not require the additional synchronization. The second call, however, could be avoided by the additional complication of a start-up state attribute.
* An extra state would be introduced: ``created but not started''. This would require the standard to specify the behavior of the thread operations when the target has not yet started executing.
* For those applications that require such behavior, it is possible to simulate the two separate steps with the facilities that are currently provided. The start_routine() can synchronize by waiting on a condition variable that is signaled by the start operation.

An Ada implementor can choose to create the thread at either of two points in the Ada program: when the task object is created, or when the task is activated (generally at a ``begin''). If the first approach is adopted, the start_routine() needs to wait on a condition variable to receive the order to begin ``activation''. The second approach requires no such condition variable or extra synchronization. In either ap? proach, a separate Ada task control block would need to be created when

the task object is created to hold rendezvous queues, and so on.

An extension of the preceding model would be to allow the state of the thread to be modified between the create and start. This would allow the thread attributes object to be eliminated. This has been rejected because:

* All state in the thread attributes object has to be able to be set for the thread. This would require the definition of functions to modify thread attributes. There would be no reduction in the number of function calls required to set up the thread. In fact, for an application that creates all threads using identical attributes, the number of function calls required to set up the threads would be dramatically increased. Use of a thread attributes object per? mits the application to make one set of attribute setting function calls. Otherwise, the set of attribute setting function calls needs to be made for each thread creation.
* Depending on the implementation architecture, functions to set thread state would require kernel calls, or for other implementa? tion reasons would not be able to be implemented as macros, thereby increasing the cost of thread creation.
* The ability for applications to segregate threads by class would be lost.

Another suggested alternative uses a model similar to that for process creation, such as ``thread fork''. The fork semantics would provide more flexibility and the ``create'' function can be implemented simply by doing a thread fork followed immediately by a call to the desired ``start routine'' for the thread. This alternative has these problems:

* For many implementations, the entire stack of the calling thread would need to be duplicated, since in many architectures there is no way to determine the size of the calling frame.
* Efficiency is reduced since at least some part of the stack has to be copied, even though in most cases the thread never needs the copied context, since it merely calls the desired start routine.

If an implementation detects that the value specified by the attr argu?

ment to pthread_create() does not refer to an initialized thread at? tributes object, it is recommended that the function should fail and report an [EINVAL] error.

FUTURE DIRECTIONS

None.

SEE ALSO

fork(), pthread_exit(), pthread_join()

The Base Definitions volume of POSIX.1?2017, Section 4.12, Memory Syn? chronization, <pthread.h>