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_key_create.3p' command

$ man pthread_key_create.3p

PTHREAD_KEY_CREATE(3P) POSIX Programmer's Manual PTHREAD_KEY_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_key_create ? thread-specific data key creation

SYNOPSIS

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

DESCRIPTION

The pthread_key_create() function shall create a thread-specific data key visible to all threads in the process. Key values provided by pthread_key_create() are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by pthread_setspecific() are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value NULL shall be associated with the new key in all active threads. Upon thread creation, the value NULL shall be associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-NULL destructor pointer, and

the thread has a non-NULL value associated with that key, the value of the key is set to NULL, and then the function pointed to is called with the previously associated value as its sole argument. The order of de? structor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all the destructors have been called for all non-NULL values with associated destructors, there are still some non-NULL values with associated destructors, then the process is repeated. If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for outstanding non-NULL values, there are still some non-NULL values with associated destructors, implementations may stop calling destruc? tors, or they may continue calling destructors until no non-NULL values with associated destructors exist, even though this might result in an infinite loop.

RETURN VALUE

If successful, the pthread_key_create() function shall store the newly created key value at *key and shall return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The pthread_key_create() function shall fail if:

EAGAIN The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the to? tal number of keys per process {PTHREAD_KEYS_MAX} has been ex? ceeded.

ENOMEM Insufficient memory exists to create the key.

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

The following sections are informative.

EXAMPLES

The following example demonstrates a function that initializes a thread-specific data key when it is first called, and associates a thread-specific object with each calling thread, initializing this ob? ject when necessary.

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;
static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}
func()
{
    void *ptr;
    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

Note that the key has to be initialized before pthread_getspecific() or pthread_setspecific() can be used. The pthread_key_create() call could either be explicitly made in a module initialization routine, or it can be done implicitly by the first call to a module as in this example. Any attempt to use the key before it is initialized is a programming error, making the code below incorrect.

```
static pthread_key_t key;
func()
{
    void *ptr;
   /* KEY NOT INITIALIZED!!!  THIS WILL NOT WORK!!! */
    if ((ptr = pthread_getspecific(key)) == NULL &&
        pthread_setspecific(key, NULL) != 0) {
        pthread_key_create(&key, NULL);
        ...
```

```
        }
    }
```

APPLICATION USAGE

None.

RATIONALE

Destructor Functions

Normally, the value bound to a key on behalf of a particular thread is a pointer to storage allocated dynamically on behalf of the calling thread. The destructor functions specified with pthread_key_create() are intended to be used to free this storage when the thread exits. Thread cancellation cleanup handlers cannot be used for this purpose because thread-specific data may persist outside the lexical scope in which the cancellation cleanup handlers operate.

If the value associated with a key needs to be updated during the life? time of the thread, it may be necessary to release the storage associ? ated with the old value before the new value is bound. Although the pthread_setspecific() function could do this automatically, this fea? ture is not needed often enough to justify the added complexity. In? stead, the programmer is responsible for freeing the stale storage:

```
pthread_getspecific(key, &old);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

Note: The above example could leak storage if run with asynchronous cancellation enabled. No such problems occur in the default cancellation state if no cancellation points occur between the get and set.

There is no notion of a destructor-safe function. If an application does not call pthread_exit() from a signal handler, or if it blocks any signal whose handler may call pthread_exit() while calling async-unsafe functions, all functions may be safely called from destructors.

Non-Idempotent Data Key Creation

There were requests to make pthread_key_create() idempotent with re?

spect to a given key address parameter. This would allow applications to call pthread_key_create() multiple times for a given key address and be guaranteed that only one key would be created. Doing so would re? quire the key value to be previously initialized (possibly at compile time) to a known null value and would require that implicit mutual-ex? clusion be performed based on the address and contents of the key pa? rameter in order to guarantee that exactly one key would be created. Unfortunately, the implicit mutual-exclusion would not be limited to only pthread_key_create(). On many implementations, implicit mutual-exclusion would also have to be performed by pthread_getspecific() and pthread_setspecific() in order to guard against using incompletely stored or not-yet-visible key values. This could significantly increase the cost of important operations, particularly pthread_getspecific(). Thus, this proposal was rejected. The pthread_key_create() function performs no implicit synchronization. It is the responsibility of the programmer to ensure that it is called exactly once per key before use of the key. Several straightforward mechanisms can already be used to accomplish this, including calling explicit module initialization func? tions, using mutexes, and using pthread_once(). This places no signif? icant burden on the programmer, introduces no possibly confusing ad hoc implicit synchronization mechanism, and potentially allows commonly used thread-specific data operations to be more efficient.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_getspecific(), pthread_key_delete()

The Base Definitions volume of POSIX.1?2017, <pthread.h>

COPYRIGHT

event of any discrepancy between this version and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at http://www.opengroup.org/unix/online.html .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.ker?nel.org/doc/man-pages/reporting_bugs.html .