NAME

pthread_key_create, pthread_setspecific, pthread_getspecific, pthread_key_delete, thr_keycreate, thr_setspecific, thr_getspecific – thread-specific-data functions

SYNOPSIS

POSIX

cc [*flag* ...] *file* ... **–lpthread** [*library* ...]

#include <pthread.h>
int pthread_key_create(pthread_key_t **keyp,* **void (****destructor***)(void *value));**
int pthread_setspecific(pthread_key_t *key,* **const void** **value***);**
void *pthread_getspecific(pthread_key_t *key***);**
int pthread_key_delete(pthread_key_t *key***);**

Solaris

cc [*flag* ...] *file* ... **–lthread** [*library* ...]

#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***);**

MT-LEVEL

MT-Safe

DESCRIPTION

Create Key

In general, thread key creation allocates a key that locates data specific to each thread in the process. The key is global to all threads in the process, which allows each thread to bind a value to the key once the key has been created. The key independently maintains specific values for each binding thread. **pthread_key_create( )** or **thr_keycreate( )** allocates a global *key* namespace, pointed to by *keyp*, that is visible to all threads in the process. Each thread is initially bound to a private element of this *key*, which allows access to its thread-specific data.

Upon key creation, a new key is assigned the value **NULL** for all active threads. Additionally, upon thread creation, all previously created keys in the new thread are assigned the value **NULL.**

Optionally, a destructor function, *destructor*, may be associated with each *key*. Upon thread exit, 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*. If more than one *destructor* exists for a thread when it exits, the order of destructor calls is unspecified.

Set Value

Once a key has been created, each thread may bind a new *value* to the key using **pthread_setspecific( )** or **thr_setspecific( )**. The values are unique to the binding thread and are individually maintained. These values continue for the life of the calling thread.

Proper synchronization of *key* storage and access must be ensured by the caller. The *value* argument to either **pthread_setspecific( )** or **thr_setspecific( )** is generally a pointer to a block of dynamically allocated memory reserved by the calling thread for its own use. (see "Examples" section below).

At thread exit, the *destructor* function, which is associated at time of creation, is called and it uses the specific key value as its sole argument.

POSIX Get Value

pthread_getspecific( ) returns the current value bound to the designated *key* specified by the calling thread. If the key has no value bound to it, the value NULL is returned. (see "Warnings" section below).

Solaris Get Value

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

POSIX Delete Key

pthread_key_delete( ) deletes a thread-specific data key formerly created by **pthread_key_create( )** or **thr_keycreate( )**. At the time **pthread_key_delete( )** is called, the thread-specific data values associated with *key* do not have to be **NULL.** It is the application's responsibility to perform cleanup actions related to the deleted key or associated thread-specific data in any threads. Cleanup can be done either before or after calling **pthread_key_delete( )**. **pthread_key_delete( )** does not invoke a destructor function.

Although **pthread_key_create( )**'s or **thr_keycreate( )**'s *destructor* function should cleanup the *key*'s thread-specific-data storage, **pthread_key_delete( )** needs to be used to free the storage associated with the *key*.

Solaris threads do not have a similar delete function.

RETURN VALUES

POSIX/Solaris

If successful, **pthread_key_create( )**, **pthread_setspecific( )**, **pthread_key_delete( )**, **thr_keycreate( )**, **thr_setspecific( )**, or **thr_getspecific( )** returns **0**; otherwise, an error number is returned to indicate the error. **pthread_getspecific( )** does not return any errors.

ERRORS

If the following conditions occur, **pthread_key_create( )** or **thr_keycreate( )** return the corresponding error number:

EAGAIN The system lacked the necessary resources to create another thread-specific data key, or the number of keys exceeds the pre-process limit of **PTHREAD_KEYS_MAX**.

ENOMEM Insufficient memory exists to create the key.

If the following conditions occur, **pthread_key_create( )**, **pthread_setspecific( )**, **thr_keycreate( )**, or **thr_setspecific( )** return the corresponding error number:

ENOMEM Insufficient memory exists to associate the value with the key.

For each of the following conditions, if the condition is detected, **pthread_setspecific( )**, **thr_setspecific( )**, or **pthread_key_delete( )** return the corresponding error number:

EINVAL The *key* value is invalid.

EXAMPLES

In this example, the thread-specific data in this function can be called from more than one thread without special initialization. POSIX threads are used exclusively in this example.

For each argument you pass to the executable of this example, a thread is created and privately bound to the string-value of that argument.

```
/* cc thisfile.c -lpthread */

#define _REENTRANT
#include <pthread.h>
void *thread_specific_data(), free();
#define MAX_ARGC 20
pthread_t tid[MAX_ARGC];
int num_threads;

main( int argc, char *argv[] ) {
    int i;
    num_threads = argc - 1;
    for( i = 0; i < num_threads; i++)
        pthread_create(&tid[i], NULL, thread_specific_data, argv[i+1]);
    for( i = 0; i < num_threads; i++)
        pthread_join(tid[i], NULL);
}  /* end main */

void *thread_specific_data(char private_data[])
{
  static pthread_mutex_t    keylock;  /* static ensures only one copy of keylock */
  static pthread_key_t      key;
  static int                once_per_keyname = 0;
  void *                    tsd = NULL;

  if (!once_per_keyname) {     /* see pthread_once(3T) */
        pthread_mutex_lock(&keylock);
        if (!once_per_keyname++)     /* retest with lock */
              pthread_key_create(&key, free);
        pthread_mutex_unlock(&keylock);
  }
  tsd = pthread_getspecific(key);
  if (tsd == NULL) {
        tsd = (void *)malloc(strlen(private_data) + 1);
        strcpy(tsd, private_data);
        pthread_setspecific(key, tsd);
        printf("tsd for %d = %s\n",thr_self(),(char *)pthread_getspecific(key));
        sleep(2);
        printf("tsd for %d remains %s\n",thr_self(),(char *)pthread_getspecific(key));
  }
}  /* end thread_specific_data */
```

```
void
free(void *v)  {
 /* application-specific clean-up function */
}
```

SEE ALSO

pthread_exit(3T)

WARNINGS

pthread_setspecific( ), **pthread_getspecific( )**, **thr_setspecific( )**, and **thr_getspecific( )**, may be called either explicitly, or implicitly from a thread-specific data destructor function. However, calling **pthread_setspecific( )** or **thr_setspecific( )** from a destructor may result in lost storage or infinite loops.