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_mutex_destroy.3p' command

$ man pthread_mutex_destroy.3p

PTHREAD_MUTEX_DESTROY(3P) POSIX Programmer's Manual PTHREAD_MUTEX_DESTROY(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_mutex_destroy, pthread_mutex_init ? destroy and initialize a mutex

SYNOPSIS

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

The pthread_mutex_destroy() function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitial? ized. An implementation may cause pthread_mutex_destroy() to set the object referenced by mutex to an invalid value.

A destroyed mutex object can be reinitialized using pthread_mu? tex_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. At? tempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a pthread_cond_timedwait() or pthread_cond_wait() call by another thread, results in undefined behavior.

The pthread_mutex_init() function shall initialize the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the de? fault mutex attributes are used; the effect shall be the same as pass? ing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and un? locked.

See Section 2.9.9, Synchronization Object Copies and Alternative Map? pings for further requirements.

Attempting to initialize an already initialized mutex results in unde? fined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes. The effect shall be equivalent to dynamic initialization by a call to pthread_mu? tex_init() with parameter attr specified as NULL, except that no error checks are performed.

The behavior is undefined if the value specified by the mutex argument to pthread_mutex_destroy() does not refer to an initialized mutex.

The behavior is undefined if the value specified by the attr argument to pthread_mutex_init() does not refer to an initialized mutex at? tributes object.

RETURN VALUE

If successful, the pthread_mutex_destroy() and pthread_mutex_init() functions shall return zero; otherwise, an error number shall be re? turned to indicate the error.

ERRORS

The pthread_mutex_init() function shall fail if:

EAGAIN The system lacked the necessary resources (other than memory) to initialize another mutex.

ENOMEM Insufficient memory exists to initialize the mutex.

EPERM The caller does not have the privilege to perform the operation.

The pthread_mutex_init() function may fail if:

EINVAL The attributes object referenced by attr has the robust mutex attribute set without the process-shared attribute being set.

These functions shall not return an error code of [EINTR].

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

If an implementation detects that the value specified by the mutex ar? gument to pthread_mutex_destroy() does not refer to an initialized mu? tex, it is recommended that the function should fail and report an [EINVAL] error.

If an implementation detects that the value specified by the mutex ar? gument to pthread_mutex_destroy() or pthread_mutex_init() refers to a locked mutex or a mutex that is referenced (for example, while being used in a pthread_cond_timedwait() or pthread_cond_wait()) by another thread, or detects that the value specified by the mutex argument to pthread_mutex_init() refers to an already initialized mutex, it is rec? ommended that the function should fail and report an [EBUSY] error.

If an implementation detects that the value specified by the attr argu? ment to pthread_mutex_init() does not refer to an initialized mutex at? tributes object, it is recommended that the function should fail and report an [EINVAL] error.

Alternate Implementations Possible

This volume of POSIX.1?2017 supports several alternative implementa? tions of mutexes. An implementation may store the lock directly in the object of type pthread_mutex_t. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or

may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice, this volume of POSIX.1?2017 does not define assignment or equality for this type, and it uses the term ``initialize'' to reinforce the (more restrictive) no? tion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opaqueness of the type.

An implementation is permitted, but not required, to have pthread_mu? tex_destroy() store an illegal value into the mutex. This may help de? tect erroneous programs that try to lock (or otherwise reference) a mu? tex that has already been destroyed.

Tradeoff Between Error Checks and Performance Supported

Many error conditions that can occur are not required to be detected by the implementation in order to let implementations trade off perfor? mance versus degree of error checking according to the needs of their specific applications and execution environment. As a general rule, conditions caused by the system (such as insufficient memory) are re? quired to be detected, but conditions caused by an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are specified to re? sult in undefined behavior.

A wide range of implementations is thus made possible. For example, an implementation intended for application debugging may implement all of the error checks, but an implementation running a single, provably cor? rect application under very tight performance constraints in an embed? ded computer might implement minimal checks. An implementation might even be provided in two versions, similar to the options that compilers provide: a full-checking, but slower version; and a limited-checking, but faster version. To forbid this optionality would be a disservice to users.

By carefully limiting the use of ``undefined behavior'' only to things that an erroneous (badly coded) application might do, and by defining that resource-not-available errors are mandatory, this volume of

POSIX.1?2017 ensures that a fully-conforming application is portable across the full range of implementations, while not forcing all imple? mentations to add overhead to check for numerous things that a correct program never does. When the behavior is undefined, no error number is specified to be returned on implementations that do detect the condi? tion. This is because undefined behavior means anything can happen, which includes returning with any value (which might happen to be a valid, but different, error number). However, since the error number might be useful to application developers when diagnosing problems dur? ing application development, a recommendation is made in rationale that implementors should return a particular error number if their implemen? tation does detect the condition.

Why No Limits are Defined

Defining symbols for the maximum number of mutexes and condition vari? ables was considered but rejected because the number of these objects may change dynamically. Furthermore, many implementations place these objects into application memory; thus, there is no explicit maximum.

Static Initializers for Mutexes and Condition Variables

Providing for static initialization of statically allocated synchro? nization objects allows modules with private static synchronization variables to avoid runtime initialization tests and overhead. Further? more, it simplifies the coding of self-initializing modules. Such mod? ules are common in C libraries, where for various reasons the design calls for self-initialization instead of requiring an explicit module initialization function to be called. An example use of static initial? ization follows.

Without static initialization, a self-initializing routine foo() might look as follows:

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;
void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
```

```
}
void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

With static initialization, the same routine could be coded as follows:

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;
void foo()
{
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

Note that the static initialization both eliminates the need for the initialization test inside pthread_once() and the fetch of &foo_mutex to learn the address to be passed to pthread_mutex_lock() or pthread_mutex_unlock().

Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a large class of systems; those where the (entire) synchronization object can be stored in application memory.

Yet the locking performance question is likely to be raised for machines that require mutexes to be allocated out of special memory. Such machines actually have to have mutexes and possibly condition variables contain pointers to the actual hardware locks. For static initialization to work on such machines, pthread_mutex_lock() also has to test whether or not the pointer to the actual lock has been allocated. If it has not, pthread_mutex_lock() has to initialize it before use. The reservation of such resources can be made when the program is loaded, and hence return codes have not been added to mutex locking and condition variable waiting to indicate failure to complete initializa?

tion.

This runtime test in pthread_mutex_lock() would at first seem to be ex? tra work; an extra test is required to see whether the pointer has been initialized. On most machines this would actually be implemented as a fetch of the pointer, testing the pointer against zero, and then using the pointer if it has already been initialized. While the test might seem to add extra work, the extra effort of testing a register is usu? ally negligible since no extra memory references are actually done. As more and more machines provide caches, the real expenses are memory references, not instructions executed.

Alternatively, depending on the machine architecture, there are often ways to eliminate all overhead in the most important case: on the lock operations that occur after the lock has been initialized. This can be done by shifting more overhead to the less frequent operation: initial? ization. Since out-of-line mutex allocation also means that an address has to be dereferenced to find the actual lock, one technique that is widely applicable is to have static initialization store a bogus value for that address; in particular, an address that causes a machine fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity checks can be done, and then the correct address for the actual lock can be filled in. Subsequent lock operations incur no extra overhead since they do not ``fault''. This is merely one tech? nique that can be used to support static initialization, while not ad? versely affecting the performance of lock acquisition. No doubt there are other techniques that are highly machine-dependent.

The locking overhead for machines doing out-of-line mutex allocation is thus similar for modules being implicitly initialized, where it is im? proved for those doing mutex allocation entirely inline. The inline case is thus made much faster, and the out-of-line case is not signifi? cantly worse.

Besides the issue of locking performance for such machines, a concern is raised that it is possible that threads would serialize contending for initialization locks when attempting to finish initializing stati?

cally allocated mutexes. (Such finishing would typically involve taking an internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing the internal lock.) First, many implementations would reduce such serialization by hashing on the mutex address. Second, such serialization can only occur a bounded number of times. In particular, it can happen at most as many times as there are statically allocated synchronization objects. Dynamically allocated objects would still be initialized via pthread_mutex_init() or pthread_cond_init().

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient performance for an application on some implementation, the application can avoid static initialization altogether by explicitly initializing all synchronization objects with the corresponding pthread_*_init() functions, which are supported by all implementations. An implementation can also document the tradeoffs and advise which initialization technique is more efficient for that particular implementation.

Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. However, since attempting to destroy a locked mutex, or a mutex that another thread is attempting to lock, or a mutex that is being used in a pthread_cond_timedwait() or pthread_cond_wait() call by another thread, results in undefined behavior, care must be taken to ensure that no other thread may be referencing the mutex.

Robust Mutexes

Implementations are required to provide robust mutexes for mutexes with the process-shared attribute set to PTHREAD_PROCESS_SHARED. Implementations are allowed, but not required, to provide robust mutexes when the process-shared attribute is set to PTHREAD_PROCESS_PRIVATE.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_getprioceiling(), pthread_mutexattr_getrobust(),

pthread_mutex_lock(), pthread_mutex_timedlock(), pthread_mutex?

attr_getpshared()

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

COPYRIGHT