NAME

mutex, pthread_mutex_init, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock, pthread_mutex_destroy, mutex_init, mutex_lock, mutex_trylock, mutex_unlock, mutex_destroy – mutual exclusion locks

SYNOPSIS

POSIX

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

#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t **mp,* **const pthread_mutexattr_t** **attr***);**
pthread_mutex_t *mutex* **= PTHREAD_MUTEX_INITIALIZER;**
int pthread_mutex_lock(pthread_mutex_t **mp***);**
int pthread_mutex_trylock(pthread_mutex_t **mp***);**
int pthread_mutex_unlock(pthread_mutex_t **mp***);**
int pthread_mutex_destroy(pthread_mutex_t **mp***);**

Solaris

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

#include <thread.h>
#include <synch.h>
int mutex_init(mutex_t **mp***, int** *type***, void** * *arg***);**
int mutex_lock(mutex_t **mp***);**
int mutex_trylock(mutex_t **mp***);**
int mutex_unlock(mutex_t **mp***);**
int mutex_destroy(mutex_t **mp***);**

MT-LEVEL

MT-Safe

DESCRIPTION

Mutual exclusion locks (mutexes) prevent multiple threads from simultaneously executing critical sections of code which access shared data (i.e., mutexes are used to serialize the execution of threads). All mutexes must be global. A successful call for a mutex lock via **pthread_mutex_lock( )** or **mutex_lock( )** will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks it via **pthread_mutex_unlock( )** or **mutex_unlock( ).** Threads within the same process or within other processes can share mutexes.

Mutexes can synchronize threads within the same process or in other processes. Mutexes can be used to synchronize threads between processes if the mutexes are allocated in writable memory and shared among the cooperating processes (see **mmap**(2)), and have been initialized for this task.

Initialize

Mutexes are either intra-process or inter-process, depending upon the argument passed implicitly or explicitly to the initialization of that mutex. A statically allocated mutex does not need to be explicitly initialized; by default, a statically allocated mutex is initialized with all zeros and its scope is set to be within the calling process. For POSIX portability of statically allocated mutexes, use the **pthread_mutex_initializer** macro (see below).

For inter-process synchronization, a mutex needs to be allocated in memory shared between these processes. Since the memory for such a mutex must be allocated dynamically, the mutex needs to be explicitly initialized using **mutex_init( )** or **pthread_mutex_init( )** with the appropriate attribute that indicates inter-process use.

POSIX Initialize

POSIX mutexes, threads, and condition variables use attributes objects in the same manner; they are initialized with the configuration of an attributes object (see **pthread_mutexattr_init**(3T)). The **pthread_mutex_init( )** function initializes the mutex referenced by *mp* with attributes specified by *attr*. If *attr* is **NULL**, the default mutex attributes are used, which is the same as passing the address of a default mutex attributes object. Upon initialization, the state of the mutex is initialized and unlocked. If default mutex attributes are used, then only threads created within the same process can operate on the initialized mutex variable.

In POSIX, the attributes of a mutex may be specified via the attribute object created via **pthread_mutexattr_init( )** and modified using the **pthread_mutexattr_*( )** functions. To explicitly specify whether a mutex is or is not shared between processes, it can be initialized with an attribute object modified via **pthread_mutexattr_setpshared**(3T). The second argument to this function can be either of the following:

PTHREAD_PROCESS_PRIVATE The mutex can synchronize threads within this process. The **PTHREAD_PROCESS_PRIVATE** POSIX mutex type for process scope is equivalent to the **USYNC_THREAD** flag to **mutex_init( )** in the Solaris API (see below).

PTHREAD_PROCESS_SHARED The mutex can synchronize threads in this process and other processes. Only one process should initialize the mutex. The **PTHREAD_PROCESS_SHARED** POSIX mutex type for system-wide scope is equivalent to the **USYNC_PROCESS** flag to **mutex_init( )** in the Solaris API (see below).

Initializing mutexes can also be accomplished by allocating in zeroed memory (default), in which case, **PTHREAD_PROCESS_PRIVATE** is assumed. The same mutex must not be simultaneously initialized by multiple threads, nor should a mutex lock be re-initialized while in use by other threads.

If default mutex attributes are used, statically allocated mutexes can be initialized by the macro **PTHREAD_MUTEX_INITIALIZER.** The effect is the same as a dynamic initialization by a call to **pthread_mutex_init( )** with parameter *attr* specified as **NULL ,** except error checks are not performed.

Default mutex initialization (intra-process):

```
pthread_mutex_t mp;
pthread_mutexattr_t mattr;

pthread_mutex_init(&mp, NULL);
```

OR

```
pthread_mutexattr_init(&mattr);
pthread_mutex_init(&mp, &mattr);
```
OR
```
pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_PRIVATE);
pthread_mutex_init(&mp, &mattr);
```
OR
```
pthread_mutex_t  mp  =  PTHREAD_MUTEX_INITIALIZER;
```
OR
```
pthread_mutex_t mp;
mp = calloc (1, sizeof (pthread_mutex_t));
```

Customized mutex initialization (inter-process):

```
pthread_mutexattr_init(&mattr);
pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&mp, &mattr);
```

Solaris Initialize

The equivalent Solaris API used to initialize a mutex so that it has several different types of behavior is the *type* argument passed to **mutex_init( )**. No current type uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_THREAD The mutex can synchronize threads only in this process. *arg* is ignored. The **USYNC_THREAD** Solaris mutex type for process scope is equivalent to the POSIX mutex attribute setting **PTHREAD_PROCESS_PRIVATE.**

USYNC_PROCESS The mutex can synchronize threads in this process and other processes. Only one process should initialize the mutex. *arg* is ignored. The **USYNC_PROCESS** Solaris mutex type for process scope is equivalent to the POSIX mutex attribute setting **PTHREAD_PROCESS_SHARED.**

Initializing mutexes can also be accomplished by allocating in zeroed memory (default), in which case, a *type* of **USYNC_THREAD** is assumed. The same mutex must not be simultaneously initialized by multiple threads. A mutex lock must not be re-initialized while in use by other threads.

If default mutex attributes are used, the macro **DEFAULTMUTEX** can be used to initialize mutexes that are statically allocated.

Default mutex initialization (intra-process):

```
mutex_t mp;

mutex_init(&mp, NULL, NULL);
```
OR
```
mutex_init(&mp, USYNC_THREAD, NULL);
```

OR

mutex_t mp = DEFAULTMUTEX;

OR

mutex_t mp;

mp = calloc(1, sizeof (mutex_t));

OR

mutex_t mp;

mp = malloc(sizeof (mutex_t));

memset(mp, 0, sizeof (mutex_t));

Customized mutex initialization (inter-process):

mutex_init(&mp, USYNC_PROCESS, NULL);

Lock and Unlock

A critical section of code is enclosed by a the call to lock the mutex and the call to unlock the mutex to protect it from simultaneous access by multiple threads. Only one thread at a time may possess mutually exclusive access to the critical section of code that is enclosed by the mutex-locking call and the mutex-unlocking call, whether the mutex's scope is intra-process or inter-process. A thread calling to lock the mutex either gets exclusive access to the code starting from the successful locking until its call to unlock the mutex, or it waits until the mutex is unlocked by the thread that locked it.

Mutexes have ownership, unlike semaphores. Although any thread, within the scope of a mutex, can get an unlocked mutex and lock access to the same critical section of code, only the thread that locked a mutex can unlock it.

If a thread waiting for a mutex receives a signal, upon return from the signal handler, the thread resumes waiting for the mutex as if there was no interrupt. A mutex protects code, not data; therefore, strongly bind a mutex with the data by putting both within the same structure, or at least within the same procedure.

POSIX/Solaris Locking

A call to **pthread_mutex_lock( )** or **mutex_lock( )** locks the mutex object referenced by *mp*. If the mutex is already locked, the calling thread blocks until the mutex is freed; this will return with the mutex object referenced by *mp* in the locked state with the calling thread as its owner. If the current owner of a mutex tries to relock the mutex, it will result in deadlock.

pthread_mutex_trylock( ) and **mutex_trylock( )** is the same as **pthread_mutex_lock( )** and **mutex_lock( ) ,** respectively, except that if the mutex object referenced by *mp* is locked (by any thread, including the current thread), the call returns immediately with an error.

pthread_mutex_unlock( ) or **mutex_unlock( )** are called by the owner of the mutex object referenced by *mp* to release it. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). If there are threads blocked on the mutex object referenced by *mp* when **pthread_mutex_unlock( )** is called, the *mp* is freed, and the scheduling policy will determine which thread gets the mutex. If the calling thread is not the owner of the lock, no error status is returned, and the behavior of the program is undefined.

Destroy

Either **pthread_mutex_destroy( )** or **mutex_destroy( )** destroys the mutex object referenced by *mp*; the mutex object becomes uninitialized. The space used by the destroyed mutex variable is not freed. It needs to be explicitly reclaimed.

RETURN VALUES

If successful, all of these functions return **0**; otherwise, an error number is returned.

pthread_mutex_trylock( ) or **mutex_trylock( )** returns **0** if a lock on the mutex object referenced by *mp* is obtained; otherwise, an error number is returned.

ERRORS

These functions fail and return the corresponding value if any of the following conditions are detected:

EFAULT *mp* or *attr* points to an illegal address.

pthread_mutex_init( ) or **mutex_init( )** fails and returns the corresponding value if any of the following conditions are detected:

EINVAL The value specified by *mp* or *attr* is invalid.

pthread_mutex_trylock( ) or **mutex_trylock( )** fails and returns the corresponding value if any of the following conditions occur:

EBUSY The mutex pointed to by *mp* was already locked.

SEE ALSO

mmap(2), **pthread_create**(3T), **pthread_mutexattr_init**(3T)

EXAMPLES

Single Gate

The following example uses one global mutex as a gate-keeper to permit each thread exclusive sequential access to the code within the user-defined function "change_global_data." This type of synchronization will protect the state of shared data, but it also prohibits parallelism.

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

#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#define NUM_THREADS 12
void *change_global_data(void *);    /*  for thr_create()   */
main(int argc,char * argv[])       {
        int i=0;
        for (i=0; i< NUM_THREADS; i++)           {
                thr_create(NULL, 0, change_global_data, NULL, 0, NULL);
        }
        while ((thr_join(NULL, NULL, NULL) == 0));
}
```

```
void * change_global_data(void *null) {
        static mutex_t   Global_mutex;
        static int       Global_data = 0;
        mutex_lock(&Global_mutex);
        Global_data++;
        sleep(1);
        printf("%d is global data\n",Global_data);
        mutex_unlock(&Global_mutex);
         return NULL;
}
```

Multiple Instruction Single Data

The previous example, the mutex, the code it owns, and the data it protects was enclosed in one function. The next example uses C++ features to accommodate many functions that use just one mutex to protect one data:

```
/* CC thisfile.c -lthread   use C++ to compile*/
#define _REENTRANT
#include <stdlib.h>
#include <stdio.h>
#include <thread.h>
#include <errno.h>
#include <iostream.h>
#define NUM_THREADS 16
void *change_global_data(void *);    /*  for thr_create()   */

class Mutected {
        private:
                static mutex_t   Global_mutex;
                static int       Global_data;
        public:
                static int       add_to_global_data(void);
                static int       subtract_from_global_data(void);
};

int Mutected::Global_data = 0;
mutex_t Mutected::Global_mutex;

int Mutected::add_to_global_data()      {
        mutex_lock(&Global_mutex);
        Global_data++;
        mutex_unlock(&Global_mutex);
        return Global_data;
}


int Mutected::subtract_from_global_data()       {
```

```
        mutex_lock(&Global_mutex);
        Global_data--;
        mutex_unlock(&Global_mutex);
        return Global_data;
}

void
main(int argc,char * argv[])    {
        int i=0;
        for (i=0;i< NUM_THREADS;i++){
                thr_create(NULL,0,change_global_data,NULL,0,NULL);
        }
        while ((thr_join(NULL,NULL,NULL) == 0));
}

void * change_global_data(void *)     {
        static int switcher = 0;
        if ((switcher++ % 3) == 0)   /* one-in-three threads subtracts */
                cout << Mutected::subtract_from_global_data() << endl;
        else
                cout << Mutected::add_to_global_data() << endl;
        return NULL;
}
```

Interprocess Locking

A mutex can protect data that is shared among processes. The mutex would need to be initialized as either **PTHREAD_PROCESS_SHARED,** for POSIX, or **USYNC_PROCESS,** for Solaris threads. (see **mutex_init**(3T) and **pthread_mutexattr_init**(3T)). One process initializes the process-shared mutex and writes it to a file to be mapped into memory by all cooperating processes (see **mmap**(2)). Afterwards, other independent processes can run the same program (whether concurrently or not) and share mutex-protected data.

```
/* cc thisfile.c -lthread */
/* To execute, run the command line "a.out 0 & a.out 1" */

#define _REENTRANT
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <thread.h>
#define INTERPROCESS_FILE "ipc-sharedfile"
#define NUM_ADDTHREADS 12
#define NUM_SUBTRACTTHREADS 10
```

```
#define INCREMENT '0'
#define DECREMENT '1'
typedef struct {
        mutex_t     Interprocess_mutex;
        int         Interprocess_data;
} buffer_t;
buffer_t *buffer;


void *add_interprocess_data(), *subtract_interprocess_data();
void create_shared_memory(), test_argv();
int zeroed[sizeof(buffer_t)];
int ipc_fd, i=0;


void
main(int argc,char * argv[])    {
    test_argv(argv[1]);

    switch (*argv[1])  {
     case INCREMENT:
        create_shared_memory();
        ipc_fd = open(INTERPROCESS_FILE, O_RDWR);
        buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
              PROT_READ | PROT_WRITE, MAP_SHARED, ipc_fd, 0);
        buffer->Interprocess_data = 0;
        mutex_init(&buffer->Interprocess_mutex, USYNC_PROCESS,0);
        for (i=0; i< NUM_ADDTHREADS; i++)
        thr_create(NULL, 0, add_interprocess_data, argv[1],
                  0, NULL);
        break;

     case DECREMENT:
        while((ipc_fd = open(INTERPROCESS_FILE, O_RDWR)) == -1)
            sleep(1);
        buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
              PROT_READ | PROT_WRITE, MAP_SHARED, ipc_fd, 0);
        for (i=0; i< NUM_SUBTRACTTHREADS; i++)
         thr_create(NULL, 0, subtract_interprocess_data, argv[1],
                0, NULL);
        break;
    } /* end switch */

    while ((thr_join(NULL,NULL,NULL) == 0));
} /* end main */
```

```
void *add_interprocess_data(char argv_1[])      {
        mutex_lock(&buffer->Interprocess_mutex);
        buffer->Interprocess_data++;
        sleep(2);
        printf("%d is add-interprocess data, and %c is argv1\n",
              buffer->Interprocess_data, argv_1[0]);
        mutex_unlock(&buffer->Interprocess_mutex);
        return NULL;
}

void *subtract_interprocess_data(char argv_1[]){
        mutex_lock(&buffer->Interprocess_mutex);
        buffer->Interprocess_data--;
        sleep(2);
        printf("%d is subtract-interprocess data, and %c is argv1\n",
              buffer->Interprocess_data, argv_1[0]);
        mutex_unlock(&buffer->Interprocess_mutex);
        return NULL;
}

void create_shared_memory()   {
        int i;
        ipc_fd = creat(INTERPROCESS_FILE, O_CREAT | O_RDWR );
        for (i=0; i<sizeof(buffer_t); i++){
                zeroed[i] = 0;
                write(ipc_fd, &zeroed[i],2);
        }
        close(ipc_fd);
        chmod(INTERPROCESS_FILE, S_IRWXU | S_IRWXG | S_IRWXO);
}

void test_argv(char argv1[])    {
 if (argv1 == NULL)  {
  printf("use 0 as arg1 for initial process\n \
  or use 1 as arg1 for the second process\n");
  exit(NULL);
 }
}
```

In this example, run the commandline

a.out 0 & a.out 1

NOTES

Currently, the only supported policy is **SCHED_OTHER.** In Solaris, under the **SCHED_OTHER** policy, there is no established order in which threads are unblocked.

In the current implementation of threads, **pthread_mutex_lock( )**, **pthread_mutex_unlock( )**, **mutex_lock( )**, **mutex_unlock( )**, **pthread_mutex_trylock( )**, and **mutex_trylock( )** do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return **EINVAL.** Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes which are allocated locally may contain junk data. Such mutexes need to be initialized using **pthread_mutex_init( )** or **mutex_init( )**.

By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined.