NAME

pthread_atfork – register fork handlers

SYNOPSIS

#include <sys/types.h>

int pthread_atfork (void (∗*prepare*)(void), void (∗*parent*)(void), void (∗*child*)(void));

MT-LEVEL

MT-Safe

DESCRIPTION

pthread_atfork( ) declares fork handlers to be called prior to and following **fork( )**, within the thread that called **fork( )**. The order of calls to **pthread_atfork( )** is important.

Before **fork( )** processing begins, the *prepare* fork handler is called. The *prepare* handler is not called if its address is **NULL.**

The *parent* fork handler is called after **fork( )** processing finishes in the parent process, and the *child* fork handler is called after **fork( )** processing finishes in the child process. If the address of *parent* or *child* is **NULL,** then its handler is not called.

The *prepare* fork handler is called in LIFO (last-in first-out) order, whereas the *parent* and *child* fork handlers are called in FIFO (first-in first-out) order. This calling order allows applications to preserve locking order.

RETURN VALUES

Upon successful completion, **pthread_atfork( )** returns **0**; otherwise, an error number is returned.

ERRORS

ENOMEM Insufficient table space exists to record the fork handler addresses.

SEE ALSO

fork(2), **atexit**(3C)

NOTES

Solaris threads do not offer this functionality, although a call to this interface may be used by a Solaris thread program since the two thread APIs are interoperable.

EXAMPLES

All multi-threaded applications that call **fork()** in a POSIX threads program, or call **fork1()** in a Solaris threads program, and which do more than simply call **exec()** in the child of the fork, should ensure that the child is protected from deadlock.

The deadlock scenario: since the "fork-one" model results in cloning only the thread that called fork, it is possible that, at the time of the call, another thread in the parent owns a lock. In the child, this thread is not cloned, and so no thread will unlock this lock in the child. Now, if the single thread in the child needs this lock, there is a deadlock.

The problem is more serious with locks in libraries. Since a library writer does not know if the application that is using the library calls **fork()** or not, the library has to protect itself, for complete correctness, from such a deadlock scenario. If the application that links with this library calls **fork()** and does not call **exec()** in the child, and needs a library lock that may be held by some other thread in the parent which is inside the library at the time of the fork, then the application deadlocks inside the library. The problem may be solved by using **pthread_atfork()**.

The following is a brief and simple description of how to make a library safe with respect to **fork1()** by using **pthread_atfork()**.

- Identify all the locks used by the library. Let's say this list is {L1,...Ln}. Also identify the locking order for these locks. Let's say that this order is also L1...Ln.
- Add a call to **pthread_atfork(f1, f2, f3)** in the library's *.init* section. f1, f2, f3 are defined as follows:

```
f1()
{
        pthread_mutex_lock(L1);               |
        pthread_mutex_lock(...);              |  --> ordered in lock order
        pthread_mutex_lock(Ln);               |
}                                             V

f2()
{
        pthread_mutex_unlock(L1);
        pthread_mutex_unlock(...);
        pthread_mutex_unlock(Ln);
}

f3()
{
        pthread_mutex_unlock(L1);
        pthread_mutex_unlock(...);
        pthread_mutex_unlock(Ln);
}
```