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_atfork.3p' command

$ man pthread_atfork.3p

PTHREAD_ATFORK(3P) POSIX Programmer's Manual PTHREAD_ATFORK(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_atfork ? register fork handlers

SYNOPSIS

```
#include <pthread.h>
int pthread_atfork(void (*prepare)(void), void (*parent)(void),
    void (*child)(void));
```

DESCRIPTION

The pthread_atfork() function shall declare fork handlers to be called before and after fork(), in the context of the thread that called fork(). The prepare fork handler shall be called before fork() pro? cessing commences. The parent fork handle shall be called after fork() processing completes in the parent process. The child fork handler shall be called after fork() processing completes in the child process. If no handling is desired at one or more of these three points, the corresponding fork handler address(es) may be set to NULL.

If a fork() call in a multi-threaded process leads to a child fork han? dler calling any function that is not async-signal-safe, the behavior

is undefined.

The order of calls to pthread_atfork() is significant. The parent and child fork handlers shall be called in the order in which they were es? tablished by calls to pthread_atfork(). The prepare fork handlers shall be called in the opposite order.

RETURN VALUE

Upon successful completion, pthread_atfork() shall return a value of zero; otherwise, an error number shall be returned to indicate the er? ror.

ERRORS

The pthread_atfork() function shall fail if:

ENOMEM Insufficient table space exists to record the fork handler ad? dresses.

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

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

The original usage pattern envisaged for pthread_atfork() was for the prepare fork handler to lock mutexes and other locks, and for the par? ent and child handlers to unlock them. However, since all of the rele? vant unlocking functions, except sem_post(), are not async-signal-safe, this usage results in undefined behavior in the child process unless the only such unlocking function it calls is sem_post().

RATIONALE

There are at least two serious problems with the semantics of fork() in a multi-threaded program. One problem has to do with state (for exam? ple, memory) covered by mutexes. Consider the case where one thread has a mutex locked and the state covered by that mutex is inconsistent while another thread calls fork(). In the child, the mutex is in the locked state (locked by a nonexistent thread and thus can never be un? locked). Having the child simply reinitialize the mutex is unsatisfac?

tory since this approach does not resolve the question about how to correct or otherwise deal with the inconsistent state in the child.

It is suggested that programs that use fork() call an exec function very soon afterwards in the child process, thus resetting all states.

In the meantime, only a short list of async-signal-safe library rou? tines are promised to be available.

Unfortunately, this solution does not address the needs of multi-threaded libraries. Application programs may not be aware that a multi-threaded library is in use, and they feel free to call any number of library routines between the fork() and exec calls, just as they always have. Indeed, they may be extant single-threaded programs and cannot, therefore, be expected to obey new restrictions imposed by the threads library.

On the other hand, the multi-threaded library needs a way to protect its internal state during fork() in case it is re-entered later in the child process. The problem arises especially in multi-threaded I/O li? braries, which are almost sure to be invoked between the fork() and exec calls to effect I/O redirection. The solution may require locking mutex variables during fork(), or it may entail simply resetting the state in the child after the fork() processing completes.

The pthread_atfork() function was intended to provide multi-threaded libraries with a means to protect themselves from innocent application programs that call fork(), and to provide multi-threaded application programs with a standard mechanism for protecting themselves from fork() calls in a library routine or the application itself.

The expected usage was that the prepare handler would acquire all mutex locks and the other two fork handlers would release them.

For example, an application could have supplied a prepare routine that acquires the necessary mutexes the library maintains and supplied child and parent routines that release those mutexes, thus ensuring that the child would have got a consistent snapshot of the state of the library (and that no mutexes would have been left stranded). This is good in theory, but in reality not practical. Each and every mutex and lock in

the process must be located and locked. Every component of a program including third-party components must participate and they must agree who is responsible for which mutex or lock. This is especially problem? atic for mutexes and locks in dynamically allocated memory. All mutexes and locks internal to the implementation must be locked, too. This pos? sibly delays the thread calling fork() for a long time or even indefi? nitely since uses of these synchronization objects may not be under control of the application. A final problem to mention here is the problem of locking streams. At least the streams under control of the system (like stdin, stdout, stderr) must be protected by locking the stream with flockfile(). But the application itself could have done that, possibly in the same thread calling fork(). In this case, the process will deadlock.

Alternatively, some libraries might have been able to supply just a child routine that reinitializes the mutexes in the library and all as? sociated states to some known value (for example, what it was when the image was originally executed). This approach is not possible, though, because implementations are allowed to fail *_init() and *_destroy() calls for mutexes and locks if the mutex or lock is still locked. In this case, the child routine is not able to reinitialize the mutexes and locks.

When fork() is called, only the calling thread is duplicated in the child process. Synchronization variables remain in the same state in the child as they were in the parent at the time fork() was called. Thus, for example, mutex locks may be held by threads that no longer exist in the child process, and any associated states may be inconsis? tent. The intention was that the parent process could have avoided this by explicit code that acquires and releases locks critical to the child via pthread_atfork(). In addition, any critical threads would have needed to be recreated and reinitialized to the proper state in the child (also via pthread_atfork()).

A higher-level package may acquire locks on its own data structures be? fore invoking lower-level packages. Under this scenario, the order

specified for fork handler calls allows a simple rule of initialization for avoiding package deadlock: a package initializes all packages on which it depends before it calls the pthread_atfork() function for it? self.

As explained, there is no suitable solution for functionality which re? quires non-atomic operations to be protected through mutexes and locks. This is why the POSIX.1 standard since the 1996 release requires that the child process after fork() in a multi-threaded process only calls async-signal-safe interfaces.

FUTURE DIRECTIONS

The pthread_atfork() function may be formally deprecated (for example, by shading it OB) in a future version of this standard.

SEE ALSO

atexit(), exec, fork()

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

COPYRIGHT