# Threads and fork(): think twice before mixing them.

Submitted by daper on Mon, 06/08/2009 - 21:10

- C
- C++
- Linux Programming

When debugging a program I came across a bug that was caused by using fork(2) in a multi-threaded program. I thought it's worth to write some words about mixing POSIX threads with fork(2) because there are non-obvious problems when doing that.

## What happens after fork() in a multi-threadeed program

The fork(2) function creates a copy of the process, all memory pages are copied, open file descriptors are copied etc. All this stuff is intuitive for a UNIX programmer. One important thing that differs the child process from the parent is that the child has only one thread. Cloning the whole process with all threads would be problematic and in most cases not what the programmer wants. Just think about it: what to do with threads that are suspended executing a system call? So the fork(2) call clones just the thread which executed it.

## What are the problems

### Critical sections, mutexes

The non-obvious problem in this approach is that at the moment of the fork(2) call some threads may be in critical sections of code, doing non-atomic operations protected by mutexes. In the child process the threads just disappears and left data half-modified

without any possibility to "fix" them, there is no way to say what other threads were doing and what should be done to make the data consistent. Moreover: state of mutexes is undefined, they might be unusable and the only way to use them in the child is to call `pthread_mutex_init()` to reset them to a usable state. It's implementation dependent how mutexes behave after fork(2) was called. On my Linux machine locked mutexes are locked in the child.

## Library functions

Problem with mutexes and critical code sections implies another non-obvious issue. It's theoretically possible to write your code executed in threads so that you are sure it's safe to call fork when such threads run but in practice there is one big problem: library functions. You're never sure if a library function you are using doesn't use global data. Even if it is thread safe, it may be achieved using mutexes internally. You are never sure. Even system library functions that are thread-safe may use locks internally. One non-obvious example is the `malloc()` function which at least in multi-threaded programs on my system uses locks. So it's not safe to call fork(2) at the moment some other thread calls `malloc()`! What the standard says about it? After fork(2) in a multi-threaded program you may only call async-safe functions (listed in signal(7)). It's similar limitation to the list of functions you are allowed to call in a signal handler and the reason is similar: in both cases the thread might be "interrupted" while executing a function.

Here is a list of few functions that use locks internally on my system, just to show you that really almost nothing is safe:

- `malloc()`
- stdio functions like `printf()` - this is required by the standard.
- `syslog()`

## execve() and open file descriptors

It seems that calling execve(2) to start another program is the only sane reason you would like to call fork(2) in a multi-threaded program. But even doing that has at least

one problem. When calling execve(2) one must remember that open file descriptors remain open and the program that was executed may read and write to them. It creates a problem if you leave open file descriptor at the time you call execve(2) that was not intended to be visible by the executed program. It even creates security issues in some cases. There is a solution for that: you must set the FD_CLOEXEC flag on all file descriptors using fcntl(2) so they are automatically closed on new programs execution. Unfortunately it's not as simple in multi-threaded program. When using fcntl(2) to set the FD_CLOEXEC flag there is a race:

```
1. fd = open ("file", O_RDWR | O_CREAT | O_TRUNC, 0600);
2. if (fd < 0) {
3.         perror ("open()");
4.         return 0;
5. }
6.
7. fcntl (fd, F_SETFD, FD_CLOEXEC);
```

When another thread executes fork(2) and execve(2) just between this thread does open(2) but before fcntl(2) a new program is started having this file descriptor duplicated. This is not what we want. A solution was created with newer standards (like POSIX.1-2008) and newer Linux kernel (changes in 2.6.23 and later). We now have O_CLOEXEC flag to the open(2) function, so the whole operation of opening a file and setting the FD_CLOEXEC flag is atomic.

There are other ways to create file descriptors than using open(2): duplicating them with dup(2), creating sockets with socket(2) etc. All those functions have now a flag similar to O_CLOEXEC or a newer version that can take similar flag (some of them, like dup2(2) does not have a flags argument, so dup3(2) was created).

It's worth to mention that a similar thing may happen in a single threaded program when it does fork(2) and execve(2) in a signal handler. This operation is perfectly legal because both of the functions are async-safe and can be called from a signal handler, but the problem is the program may be interrupted between open(2) and fcntl(2).

For more information about the new API to set FD_CLOEXEC flag see Ulrich Drepper's blog: Secure File Descriptor Handling.

## Useful system functions: pthread_atfork()

One useful function that tries to solve the problem with fork(2) in multi-threaded programs is pthread_atfork(). It has the following prototype:

```
1. int pthread_atfork(void (*prepare)(void), void (*parent)(void),
   void (*child)(void));
```

It allows to set handler functions that will be automatically executed on fork call:

- **prepare** - Called just before a new process is created.
- **parent** - Called after a new process is created in the parent.
- **child** - Called after a new process is created in the child.

The purpose of this call is to deal with critical sections of a multi-threaded program at the time fork(2) is called. A typical scenario is when in the prepare handler mutexes are locked, in the parent handler unlocked and in the child handler reinitialized.

## Summary

In my opinion there are so many problems with fork(2) in multi-threaded programs that it's almost impossible to do it right. The only clear case is to call execve(2) in the child process just after fork(2). If you want to do something more, just do it some other way, really. From my experience it's not worth trying to make the fork(2) call save, even with `pthread_atfork()`. I truly hope you read this article before hitting problems described here.

## Resources

- fork() description from The Open Group Single UNIX Specification
- Ulrich Drepper's blog: Secure File Descriptor Handling
- pthread_atfork()