NAME

fork - create a new process

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <unistd.h>

pid_t
fork(*void*);

pid_t
_Fork(*void*);

DESCRIPTION

The **fork**() function causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors, except for descriptors returned by kqueue(2), which are not inherited from the parent process. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read(2) or write(2) by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child process' resource utilizations are set to 0; see setrlimit(2).
- All interval timers are cleared; see setitimer(2).
- The robust mutexes list (see pthread_mutexattr_setrobust(3)) is cleared for the child.
- The atfork handlers established with the pthread_atfork(3) function are called as appropriate before fork in the parent process, and after the child is created, in parent and child.
- The child process has only one thread, corresponding to the calling thread in the parent

process. If the process has more than one thread, locks and other resources held by the other threads are not released and therefore only async-signal-safe functions (see sigaction(2)) are guaranteed to work in the child process until a call to execve(2) or a similar function. The FreeBSD implementation of **fork**() provides a usable malloc(3), and rtld(1) services in the child process.

The **fork**() function is not async-signal safe and creates a cancellation point in the parent process. It cannot be safely used from signal handlers, and the atfork handlers established by pthread_atfork(3) do not need to be async-signal safe either.

The **_Fork**() function creates a new process, similarly to **fork**(), but it is async-signal safe. **_Fork**() does not call atfork handlers, and does not create a cancellation point. It can be used safely from signal handlers, but then no userspace services (malloc(3) or rtld(1)) are available in the child if forked from multi-threaded parent. In particular, if using dynamic linking, all dynamic symbols used by the child after **_Fork**() must be pre-resolved. Note: resolving can be done globally by specifying the LD_BIND_NOW environment variable to the dynamic linker, or per-binary by passing the **-z** *now* option to the static linker ld(1), or by using each symbol before the **_Fork**() call to force the binding.

RETURN VALUES

Upon successful completion, **fork**() and **_Fork**() return a value of 0 to the child process and return the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

EXAMPLES

The following example shows a common pattern of how **fork**() is used in practice.

```
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(void)
{
        pid_t pid;

        /*
         * If child is expected to use stdio(3), state of
         * the reused io streams must be synchronized between
         * parent and child, to avoid double output and other
```

```
         * possible issues.
         */
        fflush(stdout);

        switch (pid = fork()) {
        case -1:
                err(1, "Failed to fork");
        case 0:
                printf("Hello from child process!\n");

                /*
                 * Since we wrote into stdout, child needs to use
                 * exit(3) and not _exit(2).  This causes handlers
                 * registered with atexit(3) to be called twice,
                 * once in parent, and once in the child.  If such
                 * behavior is undesirable, consider
                 * terminating child with _exit(2) or _Exit(3).
                 */
                exit(0);
        default:
                break;
        }

        printf("Hello from parent process (child's PID: %d)!\n", pid);

        return (0);
}
```

The output of such a program is along the lines of:

```
Hello from parent process (child's PID: 27804)!
Hello from child process!
```

ERRORS

The **fork**() system call will fail and no child process will be created if:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. The limit is given by the sysctl(3) MIB variable KERN_MAXPROC. (The limit is actually ten less than this except for the super user).

[EAGAIN] The user is not the super user, and the system-imposed limit on the total number of processes under execution by a single user would be exceeded. The limit is given by the sysctl(3) MIB variable KERN_MAXPROCPERUID.

[EAGAIN] The user is not the super user, and the soft resource limit corresponding to the *resource* argument RLIMIT_NPROC would be exceeded (see getrlimit(2)).

[ENOMEM] There is insufficient swap space for the new process.

SEE ALSO

execve(2), rfork(2), setitimer(2), setrlimit(2), sigaction(2), vfork(2), wait(2), pthread_atfork(3)

HISTORY

The **fork**() function appeared in Version 1 AT&T UNIX.

The **_Fork**() function was defined by Austin Group together with the removal of a requirement that the **fork**() implementation must be async-signal safe. The **_Fork**() function appeared in FreeBSD 14.0.