NAME

fork - create a child process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content. Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.

* The child's parent process ID is the same as the parent's process ID.

* The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

* Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.

* The child's set of pending signals is initially empty (sigpending(2)).

* The child does not inherit semaphore adjustments from its parent (semop(2)).

* The child does not inherit process-associated record locks from its parent (fcntl(2)). (On the other hand, it does inherit fcntl(2) open file description locks and flock(2) locks from its parent.)

* The child does not inherit timers from its parent (setitimer(2), alarm(2), timer_create(2)).

* The child does not inherit outstanding asynchronous I/O operations from its parent (aio_read(3), aio_write(3)), nor does it inherit any asynchronous I/O contexts from its parent (see io_setup(2)).

The process attributes in the preceding list are all specified in POSIX.1. The parent and child also differ with respect to the following Linux-specific process attributes:

* The child does not inherit directory change notifications (dnotify) from its parent (see the description of **F_NOTIFY** in fcntl(2)).

* The prctl(2) **PR_SET_PDEATHSIG** setting is reset so that the child does not receive a signal when its parent terminates.

* The default timer slack value is set to the parent's current timer slack value. See the description of **PR_SET_TIMERSLACK** in prctl(2).

* Memory mappings that have been marked with the madvise(2) **MADV_DONTFORK** flag are not inherited across a **fork**().

* Memory in address ranges that have been marked with the madvise(2) **MADV_WIPEONFORK** flag is zeroed in the child after a **fork**(). (The **MADV_WIPEONFORK** setting remains in place for those address ranges in the child.)

* The termination signal of the child is always **SIGCHLD** (see clone(2)).

* The port access permission bits set by ioperm(2) are not inherited by the child; the child must turn on any bits that it requires using ioperm(2).

Note the following further points:

* The child process is created with a single thread—the one that called **fork**(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of pthread_atfork(3) may be helpful for dealing with problems that this can cause.

* After a **fork**() in a multithreaded program, the child can safely call only async-signal-safe functions (see signal-safety(7)) until such time as it calls execve(2).

* The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see open(2)) as the corresponding file descriptor in the parent. This means that the two file descriptors share open file status flags, file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in fcntl(2)).

* The child inherits copies of the parent's set of open message queue descriptors (see mq_overview(7)). Each file descriptor in the child refers to the same open message queue description as the corresponding file descriptor in the parent. This means that the two file descriptors share the same flags (*mq_flags*).

* The child inherits copies of the parent's set of open directory streams (see opendir(3)). POSIX.1 says that the corresponding directory streams in the parent and child *may* share the directory stream positioning; on Linux/glibc they do not.

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and *errno* is set appropriately.

ERRORS

EAGAIN A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

* the **RLIMIT_NPROC** soft resource limit (set via setrlimit(2)), which limits the number of processes and threads for a real user ID, was reached;

* the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads-max*, was reached (see proc(5));

* the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see proc(5)); or

* the PID limit (*pids.max*) imposed by the cgroup "process number" (PIDs) controller was reached.

EAGAIN The caller is operating under the **SCHED_DEADLINE** scheduling policy and does not have the reset-on-fork flag set. See sched(7).

ENOMEM **fork**() failed to allocate the necessary kernel structures because memory is tight.

ENOMEM An attempt was made to create a child process in a PID namespace whose "init" process has terminated. See pid_namespaces(7).

ENOSYS **fork**() is not supported on this platform (for example, hardware without a Memory-Management Unit).

ERESTARTNOINTR (since Linux 2.6.17)
System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD.

NOTES

Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

C library/kernel differences

Since version 2.3.3, rather than invoking the kernel's **fork**() system call, the glibc **fork**() wrapper that is provided as part of the NPTL threading implementation invokes clone(2) with flags that provide the same effect as the traditional system call. (A call to **fork**() is equivalent to a call to clone(2) specifying *flags* as just **SIGCHLD**.) The glibc wrapper invokes any fork handlers that have been established using pthread_atfork(3).

EXAMPLE

See pipe(2) and wait(2).

SEE ALSO

clone(2), execve(2), exit(2), setrlimit(2), unshare(2), vfork(2), wait(2), daemon(3), pthread_atfork(3), capabilities(7), credentials(7)

COLOPHON

This page is part of release 5.02 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.