## NAME

vfork − create a child process and block parent

## SYNOPSIS

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

pid_t vfork(void);
```

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

**vfork**():

Since glibc 2.12:

```
(_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
    || /* Since glibc 2.19: */ _DEFAULT_SOURCE
    || /* Glibc versions <= 2.19: */ _BSD_SOURCE
```

Before glibc 2.12:

```
_BSD_SOURCE || _XOPEN_SOURCE >= 500
```

## DESCRIPTION

### Standard description

(From POSIX.1) The **vfork**() function has the same effect as **fork**(2), except that the behavior is undefined if the process created by **vfork**() either modifies any data other than a variable of type *pid_t* used to store the return value from **vfork**(), or returns from the function in which **vfork**() was called, or calls any other function before successfully calling **_exit**(2) or one of the **exec**(3) family of functions.

### Linux description

**vfork**(), just like **fork**(2), creates a child process of the calling process. For details and return value and errors, see **fork**(2).

**vfork**() is a special case of **clone**(2). It is used to create new processes without copying the page tables of the parent process. It may be useful in performance-sensitive applications where a child is created which then immediately issues an **execve**(2).

**vfork**() differs from **fork**(2) in that the calling thread is suspended until the child terminates (either normally, by calling **_exit**(2), or abnormally, after delivery of a fatal signal), or it makes a call to **execve**(2). Until that point, the child shares all memory with its parent, including the stack. The child must not return from the current function or call **exit**(3) (which would have the effect of calling exit handlers established by the parent process and flushing the parent's **stdio**(3) buffers), but may call **_exit**(2).

As with **fork**(2), the child process created by **vfork**() inherits copies of various of the caller's process attributes (e.g., file descriptors, signal dispositions, and current working directory); the **vfork**() call differs only in the treatment of the virtual address space, as described above.

Signals sent to the parent arrive after the child releases the parent's memory (i.e., after the child terminates or calls **execve**(2)).

### Historic description

Under Linux, **fork**(2) is implemented using copy-on-write pages, so the only penalty incurred by **fork**(2) is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. However, in the bad old days a **fork**(2) would require making a complete copy of the caller's data space, often needlessly, since usually immediately afterward an **exec**(3) is done. Thus, for greater efficiency, BSD introduced the **vfork**() system call, which did not fully copy the address space of the parent process, but borrowed the parent's memory and thread of control until a call to **execve**(2) or an exit occurred. The parent process was suspended while the child was using its resources. The use of **vfork**() was tricky: for example, not modifying data in the parent process depended on knowing which variables were held in a register.

## CONFORMING TO

4.3BSD; POSIX.1-2001 (but marked OBSOLETE). POSIX.1-2008 removes the specification of **vfork**().

The requirements put on **vfork**() by the standards are weaker than those put on **fork**(2), so an

implementation where the two are synonymous is compliant. In particular, the programmer cannot rely on the parent remaining blocked until the child either terminates or calls **execve**(2), and cannot rely on any specific behavior with respect to shared memory.

## NOTES

Some consider the semantics of **vfork**() to be an architectural blemish, and the 4.2BSD man page stated: "This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of **vfork**() as it will, in that case, be made synonymous to **fork**(2)." However, even though modern memory management hardware has decreased the performance difference between **fork**(2) and **vfork**(), there are various reasons why Linux and other systems have retained **vfork**():

* Some performance-critical applications require the small performance advantage conferred by **vfork**().
* **vfork**() can be implemented on systems that lack a memory-management unit (MMU), but **fork**(2) can't be implemented on such systems. (POSIX.1-2008 removed **vfork**() from the standard; the POSIX rationale for the **posix_spawn**(3) function notes that that function, which provides functionality equivalent to **fork**(2)+**exec**(3), is designed to be implementable on systems that lack an MMU.)
* On systems where memory is constrained, **vfork**() avoids the need to temporarily commit memory (see the description of */proc/sys/vm/overcommit_memory* in **proc**(5)) in order to execute a new program. (This can be especially beneficial where a large parent process wishes to execute a small helper program in a child process.) By contrast, using **fork**(2) in this scenario requires either committing an amount of memory equal to the size of the parent process (if strict overcommitting is in force) or overcommitting memory with the risk that a process is terminated by the out-of-memory (OOM) killer.

### Caveats

The child process should take care not to modify the memory in unintended ways, since such changes will be seen by the parent process once the child terminates or executes another program. In this regard, signal handlers can be especially problematic: if a signal handler that is invoked in the child of **vfork**() changes memory, those changes may result in an inconsistent process state from the perspective of the parent process (e.g., memory changes would be visible in the parent, but changes to the state of open file descriptors would not be visible).

When **vfork**() is called in a multithreaded process, only the calling thread is suspended until the child terminates or executes a new program. This means that the child is sharing an address space with other running code. This can be dangerous if another thread in the parent process changes credentials (using **setuid**(2) or similar), since there are now two processes with different privilege levels running in the same address space. As an example of the dangers, suppose that a multithreaded program running as root creates a child using **vfork**(). After the **vfork**(), a thread in the parent process drops the process to an unprivileged user in order to run some untrusted code (e.g., perhaps via plug-in opened with **dlopen**(3)). In this case, attacks are possible where the parent process uses **mmap**(2) to map in code that will be executed by the privileged child process.

### Linux notes

Fork handlers established using **pthread_atfork**(3) are not called when a multithreaded program employing the NPTL threading library calls **vfork**(). Fork handlers are called in this case in a program using the LinuxThreads threading library. (See **pthreads**(7) for a description of Linux threading libraries.)

A call to **vfork**() is equivalent to calling **clone**(2) with *flags* specified as:

CLONE_VM | CLONE_VFORK | SIGCHLD

### History

The **vfork**() system call appeared in 3.0BSD. In 4.4BSD it was made synonymous to **fork**(2) but NetBSD introduced it again; see ⟨http://www.netbsd.org/Documentation/kernel/vfork.html⟩. In Linux, it has been equivalent to **fork**(2) until 2.2.0-pre6 or so. Since 2.2.0-pre9 (on i386, somewhat later on other architectures) it is an independent system call. Support was added in glibc 2.0.112.

## BUGS

Details of the signal handling are obscure and differ between systems. The BSD man page states: "To avoid a possible deadlock situation, processes that are children in the middle of a **vfork**() are never sent **SIGTTOU** or **SIGTTIN** signals; rather, output or *ioctl*s are allowed and input attempts result in an end-of-file indication."

## SEE ALSO

**clone**(2), **execve**(2), **_exit**(2), **fork**(2), **unshare**(2), **wait**(2)

## COLOPHON

This page is part of release 5.05 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/.