NAME

clone, __clone2 − create a child process

SYNOPSIS

```
/* Prototype for the glibc wrapper function */

#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */ );

/* For the prototype of the raw system call, see NOTES */
```

DESCRIPTION

clone() creates a new process, in a manner similar to **fork**(2).

This page describes both the glibc **clone**() wrapper function and the underlying system call on which it is based. The main text describes the wrapper function; the differences for the raw system call are described toward the end of this page.

Unlike **fork**(2), **clone**() allows the child process to share parts of its execution context with the calling process, such as the virtual address space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of **CLONE_PARENT** below.)

One use of **clone**() is to implement threads: multiple flows of control in a program that run concurrently in a shared address space.

When the child process is created with **clone**(), it commences execution by calling the function pointed to by the argument *fn*. (This differs from **fork**(2), where execution continues in the child from the point of the **fork**(2) call.) The *arg* argument is passed as the argument of the function *fn*.

When the *fn*(*arg*) function returns, the child process terminates. The integer returned by *fn* is the exit status for the child process. The child process may also terminate explicitly by calling **exit**(2) or after receiving a fatal signal.

The *child_stack* argument specifies the location of the stack used by the child process. Since the child and calling process may share memory, it is not possible for the child process to execute in the same stack as the calling process. The calling process must therefore set up memory space for the child stack and pass a pointer to this space to **clone**(). Stacks grow downward on all processors that run Linux (except the HP PA processors), so *child_stack* usually points to the topmost address of the memory space set up for the child stack.

The low byte of *flags* contains the number of the *termination signal* sent to the parent when the child dies. If this signal is specified as anything other than **SIGCHLD**, then the parent process must specify the **__WALL** or **__WCLONE** options when waiting for the child with **wait**(2). If no signal is specified, then the parent process is not signaled when the child terminates.

flags may also be bitwise-ORed with zero or more of the following constants, in order to specify what is shared between the calling process and the child process:

CLONE_CHILD_CLEARTID (since Linux 2.5.49)

Clear (zero) the child thread ID at the location *ctid* in child memory when the child exits, and do a wakeup on the futex at that address. The address involved may be changed by the **set_tid_address**(2) system call. This is used by threading libraries.

CLONE_CHILD_SETTID (since Linux 2.5.49)

Store the child thread ID at the location *ctid* in the child's memory. The store operation completes before **clone**() returns control to user space in the child process. (Note that the store operation may not have completed before **clone**() returns in the parent process, which will be relevant if the **CLONE_VM** flag is also employed.)

CLONE_FILES (since Linux 2.0)

If **CLONE_FILES** is set, the calling process and the child process share the same file descriptor table. Any file descriptor created by the calling process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags (using the **fcntl**(2) **F_SETFD** operation), the other process is also affected. If a process sharing a file descriptor table calls **execve**(2), its file descriptor table is duplicated (unshared).

If **CLONE_FILES** is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of **clone**(). Subsequent operations that open or close file descriptors, or change file descriptor flags, performed by either the calling process or the child process do not affect the other process. Note, however, that the duplicated file descriptors in the child refer to the same open file descriptions as the corresponding file descriptors in the calling process, and thus share file offsets and file status flags (see **open**(2)).

CLONE_FS (since Linux 2.0)

If **CLONE_FS** is set, the caller and the child process share the same filesystem information. This includes the root of the filesystem, the current working directory, and the umask. Any call to **chroot**(2), **chdir**(2), or **umask**(2) performed by the calling process or the child process also affects the other process.

If **CLONE_FS** is not set, the child process works on a copy of the filesystem information of the calling process at the time of the **clone**() call. Calls to **chroot**(2), **chdir**(2), or **umask**(2) performed later by one of the processes do not affect the other process.

CLONE_IO (since Linux 2.6.25)

If **CLONE_IO** is set, then the new process shares an I/O context with the calling process. If this flag is not set, then (as with **fork**(2)) the new process has its own I/O context.

The I/O context is the I/O scope of the disk scheduler (i.e., what the I/O scheduler uses to model scheduling of a process's I/O). If processes share the same I/O context, they are treated as one by the I/O scheduler. As a consequence, they get to share disk time. For some I/O schedulers, if two processes share an I/O context, they will be allowed to interleave their disk access. If several threads are doing I/O on behalf of the same process (**aio_read**(3), for instance), they should employ **CLONE_IO** to get better I/O performance.

If the kernel is not configured with the **CONFIG_BLOCK** option, this flag is a no-op.

CLONE_NEWCGROUP (since Linux 4.6)

Create the process in a new cgroup namespace. If this flag is not set, then (as with **fork**(2)) the process is created in the same cgroup namespaces as the calling process. This flag is intended for the implementation of containers.

For further information on cgroup namespaces, see **cgroup_namespaces**(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWCGROUP**.

CLONE_NEWIPC (since Linux 2.6.19)

If **CLONE_NEWIPC** is set, then create the process in a new IPC namespace. If this flag is not set, then (as with **fork**(2)), the process is created in the same IPC namespace as the calling process. This flag is intended for the implementation of containers.

An IPC namespace provides an isolated view of System V IPC objects (see **sysvipc**(7)) and (since Linux 2.6.30) POSIX message queues (see **mq_overview**(7)). The common characteristic of these IPC mechanisms is that IPC objects are identified by mechanisms other than filesystem pathnames.

Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces.

When an IPC namespace is destroyed (i.e., when the last process that is a member of the namespace terminates), all IPC objects in the namespace are automatically destroyed.

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWIPC**. This flag can't be specified in conjunction with **CLONE_SYSVSEM**.

For further information on IPC namespaces, see **namespaces**(7).

CLONE_NEWNET (since Linux 2.6.24)

(The implementation of this flag was completed only by about kernel version 2.6.29.)

If **CLONE_NEWNET** is set, then create the process in a new network namespace. If this flag is not set, then (as with **fork**(2)) the process is created in the same network namespace as the calling process. This flag is intended for the implementation of containers.

A network namespace provides an isolated view of the networking stack (network device interfaces, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the */proc/net* and */sys/class/net* directory trees, sockets, etc.). A physical network device can live in exactly one network namespace. A virtual network (**veth**(4)) device pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace.

When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the parent of the process). For further information on network namespaces, see **namespaces**(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWNET**.

CLONE_NEWNS (since Linux 2.4.19)

If **CLONE_NEWNS** is set, the cloned child is started in a new mount namespace, initialized with a copy of the namespace of the parent. If **CLONE_NEWNS** is not set, the child lives in the same mount namespace as the parent.

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWNS**. It is not permitted to specify both **CLONE_NEWNS** and **CLONE_FS** in the same **clone**() call.

For further information on mount namespaces, see **namespaces**(7) and **mount_namespaces**(7).

CLONE_NEWPID (since Linux 2.6.24)

If **CLONE_NEWPID** is set, then create the process in a new PID namespace. If this flag is not set, then (as with **fork**(2)) the process is created in the same PID namespace as the calling process. This flag is intended for the implementation of containers.

For further information on PID namespaces, see **namespaces**(7) and **pid_namespaces**(7).

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWPID**. This flag can't be specified in conjunction with **CLONE_THREAD** or **CLONE_PARENT**.

CLONE_NEWUSER

(This flag first became meaningful for **clone**() in Linux 2.6.23, the current **clone**() semantics were merged in Linux 3.5, and the final pieces to make the user namespaces completely usable were merged in Linux 3.8.)

If **CLONE_NEWUSER** is set, then create the process in a new user namespace. If this flag is not set, then (as with **fork**(2)) the process is created in the same user namespace as the calling process.

Before Linux 3.8, use of **CLONE_NEWUSER** required that the caller have three capabilities: **CAP_SYS_ADMIN**, **CAP_SETUID**, and **CAP_SETGID**. Starting with Linux 3.8, no privileges are needed to create a user namespace.

This flag can't be specified in conjunction with **CLONE_THREAD** or **CLONE_PARENT**. For security reasons, **CLONE_NEWUSER** cannot be specified in conjunction with **CLONE_FS**.

For further information on user namespaces, see **namespaces**(7) and **user_namespaces**(7).

CLONE_NEWUTS (since Linux 2.6.19)

If **CLONE_NEWUTS** is set, then create the process in a new UTS namespace, whose identifiers are initialized by duplicating the identifiers from the UTS namespace of the calling process. If this flag is not set, then (as with **fork**(2)) the process is created in the same UTS namespace as the calling process. This flag is intended for the implementation of containers.

A UTS namespace is the set of identifiers returned by **uname**(2); among these, the domain name and the hostname can be modified by **setdomainname**(2) and **sethostname**(2), respectively. Changes made to the identifiers in a UTS namespace are visible to all other processes in the same namespace, but are not visible to processes in other UTS namespaces.

Only a privileged process (**CAP_SYS_ADMIN**) can employ **CLONE_NEWUTS**.

For further information on UTS namespaces, see **namespaces**(7).

CLONE_PARENT (since Linux 2.3.12)

If **CLONE_PARENT** is set, then the parent of the new child (as returned by **getppid**(2)) will be the same as that of the calling process.

If **CLONE_PARENT** is not set, then (as with **fork**(2)) the child's parent is the calling process.

Note that it is the parent process, as returned by **getppid**(2), which is signaled when the child terminates, so that if **CLONE_PARENT** is set, then the parent of the calling process, rather than the calling process itself, will be signaled.

CLONE_PARENT_SETTID (since Linux 2.5.49)

Store the child thread ID at the location *ptid* in the parent's memory. (In Linux 2.5.32-2.5.48 there was a flag **CLONE_SETTID** that did this.) The store operation completes before **clone**() returns control to user space.

CLONE_PID (Linux 2.0 to 2.5.15)

If **CLONE_PID** is set, the child process is created with the same process ID as the calling process. This is good for hacking the system, but otherwise of not much use. From Linux 2.3.21 onward, this flag could be specified only by the system boot process (PID 0). The flag disappeared completely from the kernel sources in Linux 2.5.16. Since then, the kernel silently ignores this bit if it is specified in *flags*.

CLONE_PTRACE (since Linux 2.2)

If **CLONE_PTRACE** is specified, and the calling process is being traced, then trace the child also (see **ptrace**(2)).

CLONE_SETTLS (since Linux 2.5.32)

The TLS (Thread Local Storage) descriptor is set to *newtls*.

The interpretation of *newtls* and the resulting effect is architecture dependent. On x86, *newtls* is interpreted as a *struct user_desc ** (see **set_thread_area**(2)). On x86-64 it is the new value to be set for the %fs base register (see the **ARCH_SET_FS** argument to **arch_prctl**(2)). On architectures with a dedicated TLS register, it is the new value of that register.

CLONE_SIGHAND (since Linux 2.0)

If **CLONE_SIGHAND** is set, the calling process and the child process share the same table of signal handlers. If the calling process or child process calls **sigaction**(2) to change the behavior associated with a signal, the behavior is changed in the other process as well. However, the calling process and child processes still have distinct signal masks and sets of pending signals. So, one of them may block or unblock signals using **sigprocmask**(2) without affecting the other process.

If **CLONE_SIGHAND** is not set, the child process inherits a copy of the signal handlers of the calling process at the time **clone**() is called. Calls to **sigaction**(2) performed later by one of the processes have no effect on the other process.

Since Linux 2.6.0, *flags* must also include **CLONE_VM** if **CLONE_SIGHAND** is specified

CLONE_STOPPED (since Linux 2.6.0)

If **CLONE_STOPPED** is set, then the child is initially stopped (as though it was sent a **SIGSTOP** signal), and must be resumed by sending it a **SIGCONT** signal.

This flag was *deprecated* from Linux 2.6.25 onward, and was *removed* altogether in Linux 2.6.38. Since then, the kernel silently ignores it without error. Starting with Linux 4.6, the same bit was reused for the **CLONE_NEWCGROUP** flag.

CLONE_SYSVSEM (since Linux 2.5.10)

If **CLONE_SYSVSEM** is set, then the child and the calling process share a single list of System V semaphore adjustment (*semadj*) values (see **semop**(2)). In this case, the shared list accumulates *semadj* values across all processes sharing the list, and semaphore adjustments are performed only when the last process that is sharing the list terminates (or ceases sharing the list using **unshare**(2)). If this flag is not set, then the child has a separate *semadj* list that is initially empty.

CLONE_THREAD (since Linux 2.4.0)

If **CLONE_THREAD** is set, the child is placed in the same thread group as the calling process. To make the remainder of the discussion of **CLONE_THREAD** more readable, the term "thread" is used to refer to the processes within a thread group.

Thread groups were a feature added in Linux 2.4 to support the POSIX threads notion of a set of threads that share a single PID. Internally, this shared PID is the so-called thread group identifier (TGID) for the thread group. Since Linux 2.4, calls to **getpid**(2) return the TGID of the caller.

The threads within a group can be distinguished by their (system-wide) unique thread IDs (TID). A new thread's TID is available as the function result returned to the caller of **clone**(), and a thread can obtain its own TID using **gettid**(2).

When a call is made to **clone**() without specifying **CLONE_THREAD**, then the resulting thread is placed in a new thread group whose TGID is the same as the thread's TID. This thread is the *leader* of the new thread group.

A new thread created with **CLONE_THREAD** has the same parent process as the caller of **clone**() (i.e., like **CLONE_PARENT**), so that calls to **getppid**(2) return the same value for all of the threads in a thread group. When a **CLONE_THREAD** thread terminates, the thread that created it using **clone**() is not sent a **SIGCHLD** (or other termination) signal; nor can the status of such a thread be obtained using **wait**(2). (The thread is said to be *detached*.)

After all of the threads in a thread group terminate the parent process of the thread group is sent a **SIGCHLD** (or other termination) signal.

If any of the threads in a thread group performs an **execve**(2), then all threads other than the thread group leader are terminated, and the new program is executed in the thread group leader.

If one of the threads in a thread group creates a child using **fork**(2), then any thread in the group can **wait**(2) for that child.

Since Linux 2.5.35, *flags* must also include **CLONE_SIGHAND** if **CLONE_THREAD** is specified (and note that, since Linux 2.6.0, **CLONE_SIGHAND** also requires **CLONE_VM** to be included).

Signal dispositions and actions are process-wide: if an unhandled signal is delivered to a thread, then it will affect (terminate, stop, continue, be ignored in) all members of the thread group.

Each thread has its own signal mask, as set by **sigprocmask**(2).

A signal may be process-directed or thread-directed. A process-directed signal is targeted at a thread group (i.e., a TGID), and is delivered to an arbitrarily selected thread from among those that are not blocking the signal. A signal may be process directed because it was generated by the kernel for reasons other than a hardware exception, or because it was sent using **kill**(2) or **sigqueue**(3). A thread-directed signal is targeted at (i.e., delivered to) a specific thread. A signal may be thread directed because it was sent using **tgkill**(2) or **pthread_sigqueue**(3), or because the thread executed a machine language instruction that triggered a hardware exception (e.g., invalid memory access triggering **SIGSEGV** or a floating-point exception triggering **SIGFPE**).

A call to **sigpending**(2) returns a signal set that is the union of the pending process-directed signals and the signals that are pending for the calling thread.

If a process-directed signal is delivered to a thread group, and the thread group has installed a handler for the signal, then the handler will be invoked in exactly one, arbitrarily selected member of the thread group that has not blocked the signal. If multiple threads in a group are waiting to accept the same signal using **sigwaitinfo**(2), the kernel will arbitrarily select one of these threads to receive the signal.

CLONE_UNTRACED (since Linux 2.5.46)

If **CLONE_UNTRACED** is specified, then a tracing process cannot force **CLONE_PTRACE** on this child process.

CLONE_VFORK (since Linux 2.2)

If **CLONE_VFORK** is set, the execution of the calling process is suspended until the child releases its virtual memory resources via a call to **execve**(2) or **_exit**(2) (as with **vfork**(2)).

If **CLONE_VFORK** is not set, then both the calling process and the child are schedulable after the call, and an application should not rely on execution occurring in any particular order.

CLONE_VM (since Linux 2.0)

If **CLONE_VM** is set, the calling process and the child process run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with **mmap**(2) or **munmap**(2) by the child or calling process also affects the other process.

If **CLONE_VM** is not set, the child process runs in a separate copy of the memory space of the calling process at the time of **clone**(). Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with **fork**(2).

NOTES

Note that the glibc **clone**() wrapper function makes some changes in the memory pointed to by *child_stack* (changes required to set the stack up correctly for the child) *before* invoking the **clone**() system call. So, in cases where **clone**() is used to recursively create children, do not use the buffer employed for the parent's stack as the stack of the child.

C library/kernel differences

The raw **clone**() system call corresponds more closely to **fork**(2) in that execution in the child continues from the point of the call. As such, the *fn* and *arg* arguments of the **clone**() wrapper function are omitted.

Another difference for the raw **clone**() system call is that the *child_stack* argument may be NULL, in which case the child uses a duplicate of the parent's stack. (Copy-on-write semantics ensure that the child gets separate copies of stack pages when either process modifies the stack.) In this case, for correct operation, the **CLONE_VM** option should not be specified. (If the child *shares* the parent's memory because of the use of the **CLONE_VM** flag, then no copy-on-write duplication occurs and chaos is likely to result.)

The order of the arguments also differs in the raw system call, and there are variations in the arguments across architectures, as detailed in the following paragraphs.

The raw system call interface on x86-64 and some other architectures (including sh, tile, ia-64, and alpha) is:

```
long clone(unsigned long flags, void *child_stack,
           int *ptid, int *ctid,
           unsigned long newtls);
```

On x86-32, and several other common architectures (including score, ARM, ARM 64, PA-RISC, arc, Power PC, xtensa, and MIPS), the order of the last two arguments is reversed:

```
long clone(unsigned long flags, void *child_stack,
          int *ptid, unsigned long newtls,
          int *ctid);
```

On the cris and s390 architectures, the order of the first two arguments is reversed:

```
long clone(void *child_stack, unsigned long flags,
                      int *ptid, int *ctid,
                      unsigned long newtls);
```

On the microblaze architecture, an additional argument is supplied:

```
        long clone(unsigned long flags, void *child_stack,
                      int stack_size,         /* Size of stack */
                      int *ptid, int *ctid,
                      unsigned long newtls);
```

blackfin, m68k, and sparc

The argument-passing conventions on blackfin, m68k, and sparc are different from the descriptions above. For details, see the kernel (and glibc) source.

ia64

On ia64, a different interface is used:

```
        int __clone2(int (*fn)(void *),
                       void *child_stack_base, size_t stack_size,
                       int flags, void *arg, ...
                    /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

The prototype shown above is for the glibc wrapper function; for the system call itself, the prototype can be described as follows (it is identical to the **clone**() prototype on microblaze):

```
        long clone2(unsigned long flags, void *child_stack_base,
                       int stack_size,         /* Size of stack */
                       int *ptid, int *ctid,
                       unsigned long tls);
```

__clone2() operates in the same way as **clone**(), except that *child_stack_base* points to the lowest address of the child's stack area, and *stack_size* specifies the size of the stack pointed to by *child_stack_base*.

Linux 2.4 and earlier

In Linux 2.4 and earlier, **clone**() does not take arguments *ptid*, *tls*, and *ctid*.

RETURN VALUE

On success, the thread ID of the child process is returned in the caller's thread of execution. On failure, −1 is returned in the caller's context, no child process will be created, and *errno* will be set appropriately.

ERRORS

EAGAIN
Too many processes are already running; see **fork**(2).

EINVAL
CLONE_SIGHAND was specified, but **CLONE_VM** was not. (Since Linux 2.6.0.)

EINVAL
CLONE_THREAD was specified, but **CLONE_SIGHAND** was not. (Since Linux 2.5.35.)

EINVAL
CLONE_THREAD was specified, but the current process previously called **unshare**(2) with the **CLONE_NEWPID** flag or used **setns**(2) to reassociate itself with a PID namespace.

EINVAL
Both **CLONE_FS** and **CLONE_NEWNS** were specified in *flags*.

EINVAL (since Linux 3.9)
Both **CLONE_NEWUSER** and **CLONE_FS** were specified in *flags*.

EINVAL
Both **CLONE_NEWIPC** and **CLONE_SYSVSEM** were specified in *flags*.

EINVAL
One (or both) of **CLONE_NEWPID** or **CLONE_NEWUSER** and one (or both) of **CLONE_THREAD** or **CLONE_PARENT** were specified in *flags*.

EINVAL
Returned by the glibc **clone**() wrapper function when *fn* or *child_stack* is specified as NULL.

EINVAL
CLONE_NEWIPC was specified in *flags*, but the kernel was not configured with the **CONFIG_SYSVIPC** and **CONFIG_IPC_NS** options.

EINVAL
CLONE_NEWNET was specified in *flags*, but the kernel was not configured with the **CONFIG_NET_NS** option.

EINVAL
CLONE_NEWPID was specified in *flags*, but the kernel was not configured with the **CONFIG_PID_NS** option.

EINVAL
CLONE_NEWUSER was specified in *flags*, but the kernel was not configured with the **CONFIG_USER_NS** option.

EINVAL
CLONE_NEWUTS was specified in *flags*, but the kernel was not configured with the **CONFIG_UTS_NS** option.

EINVAL
child_stack is not aligned to a suitable boundary for this architecture. For example, on aarch64, *child_stack* must be a multiple of 16.

ENOMEM
Cannot allocate sufficient memory to allocate a task structure for the child, or to copy those parts of the caller's context that need to be copied.

ENOSPC (since Linux 3.7)
CLONE_NEWPID was specified in flags, but the limit on the nesting depth of PID namespaces would have been exceeded; see **pid_namespaces**(7).

ENOSPC (since Linux 4.9; beforehand **EUSERS**)
CLONE_NEWUSER was specified in *flags*, and the call would cause the limit on the number of nested user namespaces to be exceeded. See **user_namespaces**(7).

From Linux 3.11 to Linux 4.8, the error diagnosed in this case was **EUSERS**.

ENOSPC (since Linux 4.9)
One of the values in *flags* specified the creation of a new user namespace, but doing so would have caused the limit defined by the corresponding file in */proc/sys/user* to be exceeded. For further details, see **namespaces**(7).

EPERM
CLONE_NEWCGROUP, **CLONE_NEWIPC**, **CLONE_NEWNET**, **CLONE_NEWNS**, **CLONE_NEWPID**, or **CLONE_NEWUTS** was specified by an unprivileged process (process without **CAP_SYS_ADMIN**).

EPERM
CLONE_PID was specified by a process other than process 0. (This error occurs only on Linux 2.5.15 and earlier.)

EPERM
CLONE_NEWUSER was specified in *flags*, but either the effective user ID or the effective group ID of the caller does not have a mapping in the parent namespace (see **user_namespaces**(7)).

EPERM (since Linux 3.9)
CLONE_NEWUSER was specified in *flags* and the caller is in a chroot environment (i.e., the caller's root directory does not match the root directory of the mount namespace in which it resides).

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.)

EUSERS (Linux 3.11 to Linux 4.8)
CLONE_NEWUSER was specified in *flags*, and the limit on the number of nested user namespaces would be exceeded. See the discussion of the **ENOSPC** error above.

CONFORMING TO

clone() is Linux-specific and should not be used in programs intended to be portable.

NOTES

The **kcmp**(2) system call can be used to test whether two processes share various resources such as a file descriptor table, System V semaphore undo operations, or a virtual address space.

Handlers registered using **pthread_atfork**(3) are not executed during a call to **clone**().

In the Linux 2.4.x series, **CLONE_THREAD** generally does not make the parent of the new thread the same as the parent of the calling process. However, for kernel versions 2.4.7 to 2.4.18 the **CLONE_THREAD** flag implied the **CLONE_PARENT** flag (as in Linux 2.6.0 and later).

For a while there was **CLONE_DETACHED** (introduced in 2.5.32): parent wants no child-exit signal. In Linux 2.6.2, the need to give this flag together with **CLONE_THREAD** disappeared. This flag is still defined, but has no effect.

On i386, **clone**() should not be called through vsyscall, but directly through *int $0x80*.

BUGS

GNU C library versions 2.3.4 up to and including 2.24 contained a wrapper function for **getpid**(2) that performed caching of PIDs. This caching relied on support in the glibc wrapper for **clone**(), but limitations in the implementation meant that the cache was not up to date in some circumstances. In particular, if a signal was delivered to the child immediately after the **clone**() call, then a call to **getpid**(2) in a handler for the signal could return the PID of the calling process ("the parent"), if the clone wrapper had not yet had a chance to update the PID cache in the child. (This discussion ignores the case where the child was created using **CLONE_THREAD**, when **getpid**(2) *should* return the same value in the child and in the process that called **clone**(), since the caller and the child are in the same thread group. The stale-cache problem also does not occur if the *flags* argument includes **CLONE_VM**.) To get the truth, it was sometimes necessary to use code such as the following:

```
#include <syscall.h>

pid_t mypid;

mypid = syscall(SYS_getpid);
```

Because of the stale-cache problem, as well as other problems noted in **getpid**(2), the PID caching feature was removed in glibc 2.25.

EXAMPLE

The following program demonstrates the use of **clone**() to create a child process that executes in a separate UTS namespace. The child changes the hostname in its UTS namespace. Both parent and child then display the system hostname, making it possible to see that the hostname differs in the UTS namespaces of the parent and child. For an example of the use of this program, see **setns**(2).

Program source

```
#define _GNU_SOURCE
#include <sys/wait.h>
```

```
#include <sys/utsname.h>
#include <sched.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)

static int              /* Start function for cloned child */
childFunc(void *arg)
{
    struct utsname uts;

    /* Change hostname in UTS namespace of child */

    if (sethostname(arg, strlen(arg)) == -1)
        errExit("sethostname");

    /* Retrieve and display hostname */

    if (uname(&uts) == -1)
        errExit("uname");
    printf("uts.nodename in child:  %s\n", uts.nodename);

    /* Keep the namespace open for a while, by sleeping.
       This allows some experimentation--for example, another
       process might join the namespace. */

    sleep(200);

    return 0;           /* Child terminates now */
}

#define STACK_SIZE (1024 * 1024)    /* Stack size for cloned child */

int
main(int argc, char *argv[])
{
    char *stack;                    /* Start of stack buffer */
    char *stackTop;                 /* End of stack buffer */
    pid_t pid;
    struct utsname uts;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <child-hostname>\n", argv[0]);
        exit(EXIT_SUCCESS);
    }

    /* Allocate stack for child */

    stack = malloc(STACK_SIZE);
    if (stack == NULL)
```

```
            errExit("malloc");
        stackTop = stack + STACK_SIZE;  /* Assume stack grows downward */

        /* Create child that has its own UTS namespace;
           child commences execution in childFunc() */

        pid = clone(childFunc, stackTop, CLONE_NEWUTS | SIGCHLD, argv[1]);
        if (pid == -1)
            errExit("clone");
        printf("clone() returned %ld\n", (long) pid);

        /* Parent falls through to here */

        sleep(1);           /* Give child time to change its hostname */

        /* Display hostname in parent's UTS namespace. This will be
           different from hostname in child's UTS namespace. */

        if (uname(&uts) == -1)
            errExit("uname");
        printf("uts.nodename in parent: %s\n", uts.nodename);

        if (waitpid(pid, NULL, 0) == -1)    /* Wait for child */
            errExit("waitpid");
        printf("child has terminated\n");

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

fork(2), **futex**(2), **getpid**(2), **gettid**(2), **kcmp**(2), **set_thread_area**(2), **set_tid_address**(2), **setns**(2), **tkill**(2), **unshare**(2), **wait**(2), **capabilities**(7), **namespaces**(7), **pthreads**(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/.