NAME

clone, __clone2 − create a child process

SYNOPSIS

#define _GNU_SOURCE
#include <sched.h>

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

DESCRIPTION

clone() creates a new process, in a manner similar to **fork**(2). It is actually a library function layered on top of the underlying **clone**() system call, hereinafter referred to as **sys_clone**. A description of **sys_clone** is given towards the end of this page.

Unlike **fork**(2), these calls allow the child process to share parts of its execution context with the calling process, such as the memory 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.)

The main use of **clone**() is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.

When the child process is created with **clone**(), it executes the function application *fn*(*arg*). (This differs from **fork**(2), where execution continues in the child from the point of the **fork**(2) call.) The *fn* argument is a pointer to a function that is called by the child process at the beginning of its execution. The *arg* argument is passed to the *fn* function.

When the *fn*(*arg*) function application returns, the child process terminates. The integer returned by *fn* is the exit code 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 downwards 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-or'ed 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)
Erase child thread ID at 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 child thread ID at location *ctid* in child memory.

CLONE_FILES
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 **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**(). (The duplicated file descriptors in the child refer to the same open file descriptions (see **open**(2)) as the corresponding file descriptors in the calling process.) 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.

CLONE_FS
If **CLONE_FS** is set, the caller and the child process share the same file system information. This includes the root of the file system, 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 file system information of the calling process at the time of the **clone**() call. Calls to **chroot**(2), **chdir**(2), **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_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 consists of the set of identifiers for System V IPC objects. (These objects are created using **msgctl**(2), **semctl**(2), and **shmctl**(2)). Objects created in an IPC namespace are visible to 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.

Use of this flag requires: a kernel configured with the **CONFIG_SYSVIPC** and **CONFIG_IPC_NS** options and that the process be privileged (**CAP_SYS_ADMIN**). This flag can't be specified in conjunction with **CLONE_SYSVSEM**.

CLONE_NEWNET (since Linux 2.6.24)
(The implementation of this flag is not yet complete, but probably will be mostly complete by about Linux 2.6.28.)

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 device ("veth") 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).

Use of this flag requires: a kernel configured with the **CONFIG_NET_NS** option and that the process be privileged (**CAP_SYS_ADMIN**).

CLONE_NEWNS (since Linux 2.4.19)
Start the child in a new mount namespace.

Every process lives in a mount namespace. The *namespace* of a process is the data (the set of mounts) describing the file hierarchy as seen by that process. After a **fork**(2) or **clone**() where the **CLONE_NEWNS** flag is not set, the child lives in the same mount namespace as the parent. The system calls **mount**(2) and **umount**(2) change the mount namespace of the calling process, and hence affect all processes that live in the same namespace, but do not affect processes in a different mount namespace.

After a **clone**() where the **CLONE_NEWNS** flag is set, the cloned child is started in a new mount namespace, initialized with a copy of the namespace of the parent.

Only a privileged process (one having the **CAP_SYS_ADMIN** capability) may specify the **CLONE_NEWNS** flag. It is not permitted to specify both **CLONE_NEWNS** and **CLONE_FS** in the same **clone**() call.

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.

A PID namespace provides an isolated environment for PIDs: PIDs in a new namespace start at 1, somewhat like a standalone system, and calls to **fork**(2), **vfork**(2), or **clone**(2) will produce processes with PIDs that are unique within the namespace.

The first process created in a new namespace (i.e., the process created using the **CLONE_NEWPID** flag) has the PID 1, and is the "init" process for the namespace. Children that are orphaned within the namespace will be reparented to this process rather than **init**(8). Unlike the traditional **init** process, the "init" process of a PID namespace can terminate, and if it does, all of the processes in the namespace are terminated.

PID namespaces form a hierarchy. When a PID new namespace is created, the processes in that namespace are visible in the PID namespace of the process that created the new namespace; analogously, if the parent PID namespace is itself the child of another PID namespace, then processes in the child and parent PID namespaces will both be visible in the grandparent PID namespace. Conversely, the processes in the "child" PID namespace do not see the processes in the parent namespace. The existence of a namespace hierarchy means that each process may now have multiple PIDs: one for each namespace in which it is visible; each of these PIDs is unique within the

corresponding namespace. (A call to **getpid**(2) always returns the PID associated with the namespace in which the process lives.)

After creating the new namespace, it is useful for the child to change its root directory and mount a new procfs instance at */proc* so that tools such as **ps**(1) work correctly. (If **CLONE_NEWNS** is also included in *flags*, then it isn't necessary to change the root directory: a new procfs instance can be mounted directly over */proc*.)

Use of this flag requires: a kernel configured with the **CONFIG_PID_NS** option and that the process be privileged (**CAP_SYS_ADMIN**). This flag can't be specified in conjunction with **CLONE_THREAD**.

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 host name can be modified by **setdomainname**(2) and **sethostname**(2), respectively. Changes made to these identifiers in one UTS namespace are visible to other processes in the same namespace, but are not visible to processes in other UTS namespaces.

Use of this flag requires: a kernel configured with the **CONFIG_UTS_NS** option and that the process be privileged (**CAP_SYS_ADMIN**).

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 child thread ID at location *ptid* in parent and child memory. (In Linux 2.5.32-2.5.48 there was a flag **CLONE_SETTID** that did this.)

CLONE_PID (obsolete)
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. Since 2.3.21 this flag can be specified only by the system boot process (PID 0). It disappeared in Linux 2.5.16.

CLONE_PTRACE
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 *newtls* argument is the new TLS (Thread Local Storage) descriptor. (See **set_thread_area**(2).)

CLONE_SIGHAND
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 some 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-test6, *flags* must also include **CLONE_VM** if **CLONE_SIGHAND** is specified

CLONE_STOPPED (since Linux 2.6.0-test2)
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.

From Linux 2.6.25 this flag is deprecated. You probably never wanted to use it, you certainly shouldn't be using it, and soon it will go away.

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 undo values (see **semop**(2)). If this flag is not set, then the child has a separate undo list, which is initially empty.

CLONE_THREAD (since Linux 2.4.0-test8)
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.

Signals may be sent to a thread group as a whole (i.e., a TGID) using **kill**(2), or to a specific thread (i.e., TID) using **tgkill**(2).

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), but signals can be pending either: for the whole process (i.e., deliverable to any member of the thread group), when sent with **kill**(2); or for an individual thread, when sent with **tgkill**(2). A call to **sigpending**(2) returns a signal set that is the union of the signals pending for the whole process and the signals that are pending for the calling thread.

If **kill**(2) is used to send a signal 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 a signal sent using **kill**(2).

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
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
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).

sys_clone
The **sys_clone** system call corresponds more closely to **fork**(2) in that execution in the child continues from the point of the call. Thus, **sys_clone** only requires the *flags* and *child_stack* arguments, which have the same meaning as for **clone**(). (Note that the order of these arguments differs from **clone**().)

Another difference for **sys_clone** is that the *child_stack* argument may be zero, in which case 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.

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

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.

EINVAL
CLONE_SIGHAND was specified, but **CLONE_VM** was not. (Since Linux 2.6.0-test6.)

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

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

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

EINVAL
Both **CLONE_NEWPID** and **CLONE_THREAD** were specified in *flags*.

EINVAL
Returned by **clone**() when a zero value is specified for *child_stack*.

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_NEWUTS was specified in *flags*, but the kernel was not configured with the **CONFIG_UTS** option.

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.

EPERM
CLONE_NEWIPC, **CLONE_NEWNET**, **CLONE_NEWNS**, **CLONE_NEWPID**, or **CLONE_NEWUTS** was specified by a non-root process (process without **CAP_SYS_ADMIN**).

EPERM
CLONE_PID was specified by a process other than process 0.

VERSIONS

There is no entry for **clone**() in libc5. glibc2 provides **clone**() as described in this manual page.

CONFORMING TO

The **clone**() and **sys_clone** calls are Linux-specific and should not be used in programs intended to be portable.

NOTES

In the kernel 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 kernel 2.6).

For a while there was **CLONE_DETACHED** (introduced in 2.5.32): parent wants no child-exit signal. In 2.6.2 the need to give this 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*.

On ia64, a different system call 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 **__clone2**() system call 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*.

BUGS

Versions of the GNU C library that include the NPTL threading library contain a wrapper function for **getpid**(2) that performs caching of PIDs. This caching relies on support in the glibc wrapper for **clone**(), but as currently implemented, the cache may not be up to date in some circumstances. In particular, if a signal is delivered to the child immediately after the **clone**() call, then a call to **getpid**() in a handler for the signal may return the PID of the calling process ("the parent"), if the clone wrapper has 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**() *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 may be necessary to use code such as the following:

```
#include <syscall.h>

pid_t mypid;

mypid = syscall(SYS_getpid);
```

SEE ALSO

fork(2), **futex**(2), **getpid**(2), **gettid**(2), **set_thread_area**(2), **set_tid_address**(2), **tkill**(2), **unshare**(2), **wait**(2), **capabilities**(7), **pthreads**(7)

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.