

clone(2) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [VERSIONS](#) | [CONFORMING TO](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#) | [COLOPHON](#)



CLONE(2)

Linux Programmer's Manual

CLONE(2)

NAME [top](#)

clone, __clone2, clone3 - create a child process

SYNOPSIS [top](#)

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

#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
          /* pid_t *parent_tid, void *tls, pid_t *child_tid */ );

/* For the prototype of the raw clone() system call, see NOTES */

#include <linux/sched.h> /* Definition of struct clone_args */
#include <sched.h>       /* Definition of CLONE_* constants */
#include <sys/syscall.h> /* Definition of SYS_* constants */
#include <unistd.h>

long syscall(SYS_clone3, struct clone_args *cl_args, size_t size);

Note: glibc provides no wrapper for clone3(), necessitating the
use of syscall\(2\).
```

DESCRIPTION [top](#)

These system calls create a new ("child") process, in a manner similar to [fork\(2\)](#).

By contrast with [fork\(2\)](#), these system calls provide more precise control over what pieces of execution context are shared between the calling process and the child process. For example, using these system calls, the caller can control whether or not the two processes share the virtual address space, the table of file descriptors, and the table of signal handlers. These system calls also allow the new child process to be placed in separate [namespaces\(7\)](#).

Note that in this manual page, "calling process" normally corresponds to "parent process". But see the descriptions of `CLONE_PARENT` and `CLONE_THREAD` below.


```

    u64 exit_signal; /* Signal to deliver to parent on
                    child termination */
    u64 stack;      /* Pointer to lowest byte of stack */
    u64 stack_size; /* Size of stack */
    u64 tls;        /* Location of new TLS */
    u64 set_tid;    /* Pointer to a pid_t array
                    (since Linux 5.5) */
    u64 set_tid_size; /* Number of elements in set_tid
                     (since Linux 5.5) */
    u64 cgroup;     /* File descriptor for target cgroup
                    of child (since Linux 5.7) */
};

```

The *size* argument that is supplied to **clone3()** should be initialized to the size of this structure. (The existence of the *size* argument permits future extensions to the *clone_args* structure.)

The stack for the child process is specified via *cl_args.stack*, which points to the lowest byte of the stack area, and *cl_args.stack_size*, which specifies the size of the stack in bytes. In the case where the **CLONE_VM** flag (see below) is specified, a stack must be explicitly allocated and specified. Otherwise, these two fields can be specified as `NULL` and `0`, which causes the child to use the same stack area as the parent (in the child's own virtual address space).

The remaining fields in the *cl_args* argument are discussed below.

Equivalence between **clone()** and **clone3()** arguments

Unlike the older **clone()** interface, where arguments are passed individually, in the newer **clone3()** interface the arguments are packaged into the *clone_args* structure shown above. This structure allows for a superset of the information passed via the **clone()** arguments.

The following table shows the equivalence between the arguments of **clone()** and the fields in the *clone_args* argument supplied to **clone3()**:

clone()	clone3()	Notes
<i>flags</i> & <code>~0xff</code>	<i>cl_args</i> field <i>flags</i>	For most flags; details below
<i>parent_tid</i>	<i>pidfd</i>	See <code>CLONE_PIDFD</code>
<i>child_tid</i>	<i>child_tid</i>	See <code>CLONE_CHILD_SETTID</code>
<i>parent_tid</i>	<i>parent_tid</i>	See <code>CLONE_PARENT_SETTID</code>
<i>flags</i> & <code>0xff</code>	<i>exit_signal</i>	
<i>stack</i>	<i>stack</i>	
---	<i>stack_size</i>	
<i>tls</i>	<i>tls</i>	See <code>CLONE_SETTLS</code>
---	<i>set_tid</i>	See below for details
---	<i>set_tid_size</i>	
---	<i>cgroup</i>	See <code>CLONE_INTO_CGROUP</code>

The child termination signal

When the child process terminates, a signal may be sent to the parent. The termination signal is specified in the low byte of *flags* (**clone()**) or in *cl_args.exit_signal* (**clone3()**). 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 (i.e., zero) is specified, then the parent process is not signaled when the child

terminates.

The `set_tid` array

By default, the kernel chooses the next sequential PID for the new process in each of the PID namespaces where it is present. When creating a process with `clone3()`, the `set_tid` array (available since Linux 5.5) can be used to select specific PIDs for the process in some or all of the PID namespaces where it is present. If the PID of the newly created process should be set only for the current PID namespace or in the newly created PID namespace (if `flags` contains `CLONE_NEWPID`) then the first element in the `set_tid` array has to be the desired PID and `set_tid_size` needs to be 1.

If the PID of the newly created process should have a certain value in multiple PID namespaces, then the `set_tid` array can have multiple entries. The first entry defines the PID in the most deeply nested PID namespace and each of the following entries contains the PID in the corresponding ancestor PID namespace. The number of PID namespaces in which a PID should be set is defined by `set_tid_size` which cannot be larger than the number of currently nested PID namespaces.

To create a process with the following PIDs in a PID namespace hierarchy:

PID	NS level	Requested PID	Notes
0		31496	Outermost PID namespace
1		42	
2		7	Innermost PID namespace

Set the array to:

```
set_tid[0] = 7;
set_tid[1] = 42;
set_tid[2] = 31496;
set_tid_size = 3;
```

If only the PIDs in the two innermost PID namespaces need to be specified, set the array to:

```
set_tid[0] = 7;
set_tid[1] = 42;
set_tid_size = 2;
```

The PID in the PID namespaces outside the two innermost PID namespaces is selected the same way as any other PID is selected.

The `set_tid` feature requires `CAP_SYS_ADMIN` or (since Linux 5.9) `CAP_CHECKPOINT_RESTORE` in all owning user namespaces of the target PID namespaces.

Callers may only choose a PID greater than 1 in a given PID namespace if an `init` process (i.e., a process with PID 1) already exists in that namespace. Otherwise the PID entry for this PID namespace must be 1.

The flags mask

Both `clone()` and `clone3()` allow a flags bit mask that modifies their behavior and allows the caller to specify what is shared between the calling process and the child process. This bit mask—the `flags` argument of `clone()` or the `cl_args.flags` field

passed to `clone3()`—is referred to as the *flags* mask in the remainder of this page.

The *flags* mask is specified as a bitwise-OR of zero or more of the constants listed below. Except as noted below, these flags are available (and have the same effect) in both `clone()` and `clone3()`.

CLONE_CLEAR_SIGHAND (since Linux 5.5)

By default, signal dispositions in the child thread are the same as in the parent. If this flag is specified, then all signals that are handled in the parent are reset to their default dispositions (`SIG_DFL`) in the child.

Specifying this flag together with `CLONE_SIGHAND` is nonsensical and disallowed.

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 the clone call. 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_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.

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.

For further information on IPC namespaces, see [ipc namespaces\(7\)](#).

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

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.

For further information on network namespaces, see [network 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.

For further information on mount namespaces, see [namespaces\(7\)](#) and [mount namespaces\(7\)](#).

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.

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.

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.

For further information on user namespaces, see [namespaces\(7\)](#) and [user namespaces\(7\)](#).

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**.

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.

For further information on UTS namespaces, see [uts namespaces\(7\)](#).

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

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, is signaled.

The **CLONE_PARENT** flag can't be used in clone calls by the global init process (PID 1 in the initial PID namespace) and init processes in other PID namespaces. This restriction prevents the creation of multi-rooted process trees as well as the creation of unreachably zombies in the initial PID namespace.

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_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 of the clone call. Calls to [sigaction\(2\)](#) performed later by one of the processes have no effect on the other process.

Since Linux 2.6.0, the *flags* mask 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_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 the clone call. Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with [fork\(2\)](#).

If the **CLONE_VM** flag is specified and the **CLONE_VFORK** flag is not specified, then any alternate signal stack that was established by [sigaltstack\(2\)](#) is cleared in the child process.

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 is created, and [errno](#) is set to indicate the error.

ERRORS [top](#)

EAGAIN Too many processes are already running; see [fork\(2\)](#).

EBUSY (`clone3()` only)

CLONE_INTO_CGROUP was specified in `cl_args.flags`, but the file descriptor specified in `cl_args.cgroup` refers to a version 2 cgroup in which a domain controller is enabled.

EEXIST (`clone3()` only)

One (or more) of the PIDs specified in `set_tid` already exists in the corresponding PID namespace.

EINVAL Both **CLONE_SIGHAND** and **CLONE_CLEAR_SIGHAND** were specified in the `flags` mask.

EINVAL **CLONE_SIGHAND** was specified in the `flags` mask, but **CLONE_VM** was not. (Since Linux 2.6.0.)

EINVAL **CLONE_THREAD** was specified in the `flags` mask, but **CLONE_SIGHAND** was not. (Since Linux 2.5.35.)

EINVAL **CLONE_THREAD** was specified in the `flags` mask, 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 the `flags` mask.

EINVAL (since Linux 3.9)

Both **CLONE_NEWUSER** and **CLONE_FS** were specified in the `flags` mask.

EINVAL Both **CLONE_NEWIPC** and **CLONE_SYSVSEM** were specified in the `flags` mask.

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

EINVAL (since Linux 2.6.32)

CLONE_PARENT was specified, and the caller is an init process.

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

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

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

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

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

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

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

EINVAL (**clone3()** only)
CLONE_DETACHED was specified in the *flags* mask.

EINVAL (**clone()** only)
CLONE_PIDFD was specified together with **CLONE_DETACHED** in the *flags* mask.

EINVAL **CLONE_PIDFD** was specified together with **CLONE_THREAD** in the *flags* mask.

EINVAL (**clone()** only)
CLONE_PIDFD was specified together with **CLONE_PARENT_SETTID** in the *flags* mask.

EINVAL (**clone3()** only)
set_tid_size is greater than the number of nested PID namespaces.

EINVAL (**clone3()** only)
One of the PIDs specified in *set_tid* was an invalid.

EINVAL (AArch64 only, Linux 4.6 and earlier)
stack was not aligned to a 128-bit boundary.

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 the *flags* mask, 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 the *flags* mask, 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 the *flags* mask 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\(7\)](#).

EOPNOTSUPP (**clone3()** only)
CLONE_INTO_CGROUP was specified in *cl_args.flags*, but the file descriptor specified in *cl_args.cgroup* refers to a version 2 cgroup that is in the *domain invalid* state.

- 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 the *flags* mask, 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 the *flags* mask 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).
- EPERM** (**clone3()** only)
set_tid_size was greater than zero, and the caller lacks the **CAP_SYS_ADMIN** capability in one or more of the user namespaces that own the corresponding PID namespaces.
- 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 the *flags* mask, and the limit on the number of nested user namespaces would be exceeded. See the discussion of the **ENOSPC** error above.

VERSIONS [top](#)

The **clone3()** system call first appeared in Linux 5.3.

CONFORMING TO [top](#)

These system calls are Linux-specific and should not be used in programs intended to be portable.

NOTES [top](#)

One use of these systems calls is to implement threads: multiple flows of control in a program that run concurrently in a shared address space.

Note that the glibc **clone()** wrapper function makes some changes in the memory pointed to by *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.

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 clone call.

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).

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

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.

In contrast to the glibc wrapper, the raw **clone()** system call accepts NULL as a *stack* argument (and **clone3()** likewise allows *cl_args.stack* to be NULL). In this 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, and alpha) is:

```
long clone(unsigned long flags, void *stack,
           int *parent_tid, int *child_tid,
           unsigned long tls);
```

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 *stack,
           int *parent_tid, unsigned long tls,
           int *child_tid);
```

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

```
long clone(void *stack, unsigned long flags,
           int *parent_tid, int *child_tid,
           unsigned long tls);
```

On the microblaze architecture, an additional argument is supplied:

```
long clone(unsigned long flags, void *stack,
           int stack_size,          /* Size of stack */
           int *parent_tid, int *child_tid,
           unsigned long tls);
```

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 *stack_base, size_t stack_size,
             int flags, void *arg, ...
             /* pid_t *parent_tid, struct user_desc *tls,
              pid_t *child_tid */ );
```

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 *stack_base,
            int stack_size, /* Size of stack */
            int *parent_tid, int *child_tid,
            unsigned long tls);
```

`__clone2()` operates in the same way as `clone()`, except that `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 `stack_base`.

Linux 2.4 and earlier

In Linux 2.4 and earlier, `clone()` does not take arguments `parent_tid`, `tls`, and `child_tid`.

BUGS [top](#)

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.

EXAMPLES [top](#)

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\(2\)](#).

Within the sample program, we allocate the memory that is to be used for the child's stack using [mmap\(2\)](#) rather than [malloc\(3\)](#) for the following reasons:

- * [mmap\(2\)](#) allocates a block of memory that starts on a page boundary and is a multiple of the page size. This is useful if we want to establish a guard page (a page with protection `PROT_NONE`) at the end of the stack using [mprotect\(2\)](#).
- * We can specify the `MAP_STACK` flag to request a mapping that is suitable for a stack. For the moment, this flag is a no-op on Linux, but it exists and has effect on some other systems, so we should include it for portability.

Program source

```
#define _GNU_SOURCE
#include <sys/wait.h>
#include <sys/utsname.h>
#include <sched.h>
#include <string.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.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 memory to be used for the stack of the child. */

    stack = mmap(NULL, STACK_SIZE, PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
    if (stack == MAP_FAILED)
        errExit("mmap");

    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 %jd\n", (intmax_t) 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 [top](#)

[fork\(2\)](#), [futex\(2\)](#), [getpid\(2\)](#), [gettid\(2\)](#), [kcmp\(2\)](#), [mmap\(2\)](#),
[pidfd_open\(2\)](#), [set_thread_area\(2\)](#), [set_tid_address\(2\)](#), [setns\(2\)](#),
[tkill\(2\)](#), [unshare\(2\)](#), [wait\(2\)](#), [capabilities\(7\)](#), [namespaces\(7\)](#),
[pthreads\(7\)](#)

COLOPHON [top](#)

This page is part of release 5.13 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/>.

Linux

2021-03-22

CLONE(2)