NAME

prctl − operations on a process or thread

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/prctl.h>

int prctl(int *option***, unsigned long** *arg2***, unsigned long** *arg3***,**
unsigned long *arg4***, unsigned long** *arg5***);**

DESCRIPTION

prctl() manipulates various aspects of the behavior of the calling thread or process.

Note that careless use of some **prctl**() operations can confuse the user-space run-time environment, so these operations should be used with care.

prctl() is called with a first argument describing what to do (with values defined in *<linux/prctl.h>*), and further arguments with a significance depending on the first one. The first argument can be:

PR_CAP_AMBIENT (since Linux 4.3)

Reads or changes the ambient capability set of the calling thread, according to the value of *arg2*, which must be one of the following:

PR_CAP_AMBIENT_RAISE

The capability specified in *arg3* is added to the ambient set. The specified capability must already be present in both the permitted and the inheritable sets of the process. This operation is not permitted if the **SECBIT_NO_CAP_AMBIENT_RAISE** securebit is set.

PR_CAP_AMBIENT_LOWER

The capability specified in *arg3* is removed from the ambient set.

PR_CAP_AMBIENT_IS_SET

The **prctl**() call returns 1 if the capability in *arg3* is in the ambient set and 0 if it is not.

PR_CAP_AMBIENT_CLEAR_ALL

All capabilities will be removed from the ambient set. This operation requires setting *arg3* to zero.

In all of the above operations, *arg4* and *arg5* must be specified as 0.

Higher-level interfaces layered on top of the above operations are provided in the **libcap**(3) library in the form of **cap_get_ambient**(3), **cap_set_ambient**(3), and **cap_reset_ambient**(3).

PR_CAPBSET_READ (since Linux 2.6.25)

Return (as the function result) 1 if the capability specified in *arg2* is in the calling thread's capability bounding set, or 0 if it is not. (The capability constants are defined in*<linux/capability .h>*.) The capability bounding set dictates whether the process can receive the capability through a file's permitted capability set on a subsequent call to **execve**(2).

If the capability specified in *arg2* is not valid, then the call fails with the error **EINVAL**.

A higher-level interface layered on top of this operation is provided in the **libcap**(3) library in the form of **cap_get_bound**(3).

PR_CAPBSET_DROP (since Linux 2.6.25)

If the calling thread has the **CAP_SETPCAP** capability within its user namespace, then drop the capability specified by *arg2* from the calling thread's capability bounding set. Any children of the calling thread will inherit the newly reduced bounding set.

The call fails with the error: **EPERM** if the calling thread does not have the **CAP_SETPCAP**; **EINVAL** if *arg2* does not represent a valid capability; or **EINVAL** if file capabilities are not enabled in the kernel, in which case bounding sets are not supported.

A higher-level interface layered on top of this operation is provided in the **libcap**(3) library in the form of **cap_drop_bound**(3).

PR_SET_CHILD_SUBREAPER (since Linux 3.4)
If *arg2* is nonzero, set the "child subreaper" attribute of the calling process; if *arg2* is zero, unset the attribute.

A subreaper fulfills the role of **init**(1) for its descendant processes. When a process becomes orphaned (i.e., its immediate parent terminates), then that process will be reparented to the nearest still living ancestor subreaper. Subsequently, calls to **getppid**(2) in the orphaned process will now return the PID of the subreaper process, and when the orphan terminates, it is the subreaper process that will receive a **SIGCHLD** signal and will be able to **wait**(2) on the process to discover its termination status.

The setting of the "child subreaper" attribute is not inherited by children created by **fork**(2) and **clone**(2). The setting is preserved across **execve**(2).

Establishing a subreaper process is useful in session management frameworks where a hierarchical group of processes is managed by a subreaper process that needs to be informed when one of the processes—for example, a double-forked daemon—terminates (perhaps so that it can restart that process). Some**init**(1) frame works (e.g., **systemd**(1)) employ a subreaper process for similar reasons.

PR_GET_CHILD_SUBREAPER (since Linux 3.4)
Return the "child subreaper" setting of the caller, in the location pointed to by *(int *) arg2*.

PR_SET_DUMPABLE (since Linux 2.3.20)
Set the state of the "dumpable" attribute, which determines whether core dumps are produced for the calling process upon delivery of a signal whose default behavior is to produce a core dump.

Up to and including Linux 2.6.12, *arg2* must be either 0 (**SUID_DUMP_DISABLE**, process is not dumpable) or 1 (**SUID_DUMP_USER**, process is dumpable). Between Linux 2.6.13 and Linux 2.6.17, the value 2 was also permitted, which caused any binary which normally would not be dumped to be dumped readable by root only; for security reasons, this feature has been removed. (See also the description of*/pr oc/sys/fs/suid_dumpable* in **proc**(5).)

Normally, the "dumpable" attribute is set to 1. However, it is reset to the current value contained in the file */proc/sys/fs/suid_dumpable* (which by default has the value 0), in the following circumstances:

- The process's effective user or group ID is changed.
- The process's filesystem user or group ID is changed (see **credentials**(7)).
- The process executes (**execve**(2)) a set-user-ID or set-group-ID program, resulting in a change of either the effective user ID or the effective group ID.
- The process executes (**execve**(2)) a program that has file capabilities (see **capabilities**(7)), but only if the permitted capabilities gained exceed those already permitted for the process.

Processes that are not dumpable can not be attached via **ptrace**(2) **PTRACE_ATTACH**; see **ptrace**(2) for further details.

If a process is not dumpable, the ownership of files in the process's */proc/*pid directory is affected as described in **proc**(5).

PR_GET_DUMPABLE (since Linux 2.3.20)
Return (as the function result) the current state of the calling process's dumpable attribute.

PR_SET_ENDIAN (since Linux 2.6.18, PowerPC only)
Set the endian-ness of the calling process to the value given in *arg2*, which should be one of the following: **PR_ENDIAN_BIG**, **PR_ENDIAN_LITTLE**, or **PR_ENDIAN_PPC_LITTLE** (PowerPC pseudo little endian).

PR_GET_ENDIAN (since Linux 2.6.18, PowerPC only)
Return the endian-ness of the calling process, in the location pointed to by *(int *) arg2*.

PR_SET_FP_MODE (since Linux 4.0, only on MIPS)
On the MIPS architecture, user-space code can be built using an ABI which permits linking with code that has more restrictive floating-point (FP) requirements. For example, user-space code may be built to target the O32 FPXX ABI and linked with code built for either one of the more restrictive FP32 or FP64 ABIs. When more restrictive code is linked in, the overall requirement for the process is to use the more restrictive floating-point mode.

Because the kernel has no means of knowing in advance which mode the process should be executed in, and because these restrictions can change over the lifetime of the process, the **PR_SET_FP_MODE** operation is provided to allow control of the floating-point mode from user space.

The *(unsigned int) arg2* argument is a bit mask describing the floating-point mode used:

PR_FP_MODE_FR
When this bit is *unset* (so called **FR=0** or **FR0** mode), the 32 floating-point registers are 32 bits wide, and 64-bit registers are represented as a pair of registers (even- and odd-numbered, with the even-numbered register containing the lower 32 bits, and the odd-numbered register containing the higher 32 bits).

When this bit is *set* (on supported hardware), the 32 floating-point registers are 64 bits wide (so called **FR=1** or **FR1** mode). Note that modern MIPS implementations (MIPS R6 and newer) support **FR=1** mode only.

Applications that use the O32 FP32 ABI can operate only when this bit is *unset* (**FR=0**; or they can be used with FRE enabled, see below). Applications that use the O32 FP64 ABI (and the O32 FP64A ABI, which exists to provide the ability to operate with existing FP32 code; see below) can operate only when this bit is *set* (**FR=1**). Applications that use the O32 FPXX ABI can operate with either **FR=0** or **FR=1 .**

PR_FP_MODE_FRE
Enable emulation of 32-bit floating-point mode. When this mode is enabled, it emulates 32-bit floating-point operations by raising a reserved-instruction exception on every instruction that uses 32-bit formats and the kernel then handles the instruction in software. (The problem lies in the discrepancy of handling odd-numbered registers which are the high 32 bits of 64-bit registers with even numbers in **FR=0** mode and the lower 32-bit parts of odd-numbered 64-bit registers in **FR=1** mode.) Enabling this bit is necessary when code with the O32 FP32 ABI should operate with code with compatible the O32 FPXX or O32 FP64A ABIs (which require **FR=1** FPU mode) or when it is executed on newer hardware (MIPS R6 onwards) which lacks **FR=0** mode support when a binary with the FP32 ABI is used.

Note that this mode makes sense only when the FPU is in 64-bit mode (**FR=1**).

Note that the use of emulation inherently has a significant performance hit and should be avoided if possible.

In the N32/N64 ABI, 64-bit floating-point mode is always used, so FPU emulation is not required and the FPU always operates in **FR=1** mode.

This option is mainly intended for use by the dynamic linker (**ld.so**(8)).

The arguments *arg3*, *arg4*, and *arg5* are ignored.

PR_GET_FP_MODE (since Linux 4.0, only on MIPS)
Return (as the function result) the current floating-point mode (see the description of **PR_SET_FP_MODE** for details).

On success, the call returns a bit mask which represents the current floating-point mode.

The arguments *arg2*, *arg3*, *arg4*, and *arg5* are ignored.

PR_SET_FPEMU (since Linux 2.4.18, 2.5.9, only on ia64)
Set floating-point emulation control bits to *arg2*. Pass **PR_FPEMU_NOPRINT** to silently emulate floating-point operation accesses, or **PR_FPEMU_SIGFPE** to not emulate floating-point operations and send **SIGFPE** instead.

PR_GET_FPEMU (since Linux 2.4.18, 2.5.9, only on ia64)
Return floating-point emulation control bits, in the location pointed to by *(int *) arg2*.

PR_SET_FPEXC (since Linux 2.4.21, 2.5.32, only on PowerPC)
Set floating-point exception mode to *arg2*. Pass **PR_FP_EXC_SW_ENABLE** to use FPEXC for FP exception enables, **PR_FP_EXC_DIV** for floating-point divide by zero, **PR_FP_EXC_OVF** for floating-point overflow, **PR_FP_EXC_UND** for floating-point underflow, **PR_FP_EXC_RES** for floating-point inexact result, **PR_FP_EXC_INV** for floating-point invalid operation, **PR_FP_EXC_DISABLED** for FP exceptions disabled, **PR_FP_EXC_NONRECOV** for async nonrecoverable exception mode, **PR_FP_EXC_ASYNC** for async recoverable exception mode, **PR_FP_EXC_PRECISE** for precise exception mode.

PR_GET_FPEXC (since Linux 2.4.21, 2.5.32, only on PowerPC)
Return floating-point exception mode, in the location pointed to by *(int *) arg2*.

PR_SET_IO_FLUSHER (since Linux 5.6)
If a user process is involved in the block layer or filesystem I/O path, and can allocate memory while processing I/O requests it must set *arg2* to 1. This will put the process in the IO_FLUSHER state, which allows it special treatment to make progress when allocating memory. If *arg2* is 0, the process will clear the IO_FLUSHER state, and the default behavior will be used.

The calling process must have the **CAP_SYS_RESOURCE** capability.

arg3, *arg4*, and *arg5* must be zero.

The IO_FLUSHER state is inherited by a child process created via **fork**(2) and is preserved across **execve**(2).

Examples of IO_FLUSHER applications are FUSE daemons, SCSI device emulation daemons, and daemons that perform error handling like multipath path recovery applications.

PR_GET_IO_FLUSHER (Since Linux 5.6)
Return (as the function result) the IO_FLUSHER state of the caller. A value of 1 indicates that the caller is in the IO_FLUSHER state; 0 indicates that the caller is not in the IO_FLUSHER state.

The calling process must have the **CAP_SYS_RESOURCE** capability.

arg2, *arg3*, *arg4*, and *arg5* must be zero.

PR_SET_KEEPCAPS (since Linux 2.2.18)
Set the state of the calling thread's "keep capabilities" flag. The effect of this flag is described in **capabilities**(7). *arg2* must be either 0 (clear the flag) or 1 (set the flag). The "keep capabilities" value will be reset to 0 on subsequent calls to **execve**(2).

PR_GET_KEEPCAPS (since Linux 2.2.18)
Return (as the function result) the current state of the calling thread's "keep capabilities" flag. See **capabilities**(7) for a description of this flag.

PR_MCE_KILL (since Linux 2.6.32)
Set the machine check memory corruption kill policy for the calling thread. If *arg2* is **PR_MCE_KILL_CLEAR**, clear the thread memory corruption kill policy and use the system-wide default. (The system-wide default is defined by */proc/sys/vm/memory_failure_early_kill*; see **proc**(5).) If *arg2* is **PR_MCE_KILL_SET**, use a thread-specific memory corruption kill policy. In this case, *arg3* defines whether the policy is *early kill* (**PR_MCE_KILL_EARLY**), *late kill* (**PR_MCE_KILL_LATE**), or the system-wide default (**PR_MCE_KILL_DEFAULT**). Early kill means that the thread receives a **SIGBUS** signal as soon as hardware memory corruption is

detected inside its address space. In late kill mode, the process is killed only when it accesses a corrupted page. See **sigaction**(2) for more information on the **SIGBUS** signal. The policy is inherited by children. The remaining unused **prctl**() arguments must be zero for future compatibility.

PR_MCE_KILL_GET (since Linux 2.6.32)
Return (as the function result) the current per-process machine check kill policy. All unused **prctl**() arguments must be zero.

PR_SET_MM (since Linux 3.3)
Modify certain kernel memory map descriptor fields of the calling process. Usually these fields are set by the kernel and dynamic loader (see **ld.so**(8) for more information) and a regular application should not use this feature. However, there are cases, such as self-modifying programs, where a program might find it useful to change its own memory map.

The calling process must have the **CAP_SYS_RESOURCE** capability. The value in *arg2* is one of the options below, while *arg3* provides a new value for the option. The *arg4* and *arg5* arguments must be zero if unused.

Before Linux 3.10, this feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled.

PR_SET_MM_START_CODE
Set the address above which the program text can run. The corresponding memory area must be readable and executable, but not writable or shareable (see **mprotect**(2) and **mmap**(2) for more information).

PR_SET_MM_END_CODE
Set the address below which the program text can run. The corresponding memory area must be readable and executable, but not writable or shareable.

PR_SET_MM_START_DATA
Set the address above which initialized and uninitialized (bss) data are placed. The corresponding memory area must be readable and writable, but not executable or shareable.

PR_SET_MM_END_DATA
Set the address below which initialized and uninitialized (bss) data are placed. The corresponding memory area must be readable and writable, but not executable or shareable.

PR_SET_MM_START_STACK
Set the start address of the stack. The corresponding memory area must be readable and writable.

PR_SET_MM_START_BRK
Set the address above which the program heap can be expanded with **brk**(2) call. The address must be greater than the ending address of the current program data segment. In addition, the combined size of the resulting heap and the size of the data segment can't exceed the **RLIMIT_DATA** resource limit (see **setrlimit**(2)).

PR_SET_MM_BRK
Set the current **brk**(2) value. The requirements for the address are the same as for the **PR_SET_MM_START_BRK** option.

The following options are available since Linux 3.5.

PR_SET_MM_ARG_START
Set the address above which the program command line is placed.

PR_SET_MM_ARG_END
Set the address below which the program command line is placed.

PR_SET_MM_ENV_START
Set the address above which the program environment is placed.

PR_SET_MM_ENV_END
Set the address below which the program environment is placed.

The address passed with **PR_SET_MM_ARG_START**, **PR_SET_MM_ARG_END**, **PR_SET_MM_ENV_START**, and **PR_SET_MM_ENV_END** should belong to a process stack area. Thus, the corresponding memory area must be readable, writable, and (depending on the kernel configuration) have the **MAP_GROWSDOWN** attribute set (see **mmap**(2)).

PR_SET_MM_AUXV
Set a new auxiliary vector. The*ar g3* argument should provide the address of the vector. The *arg4* is the size of the vector.

PR_SET_MM_EXE_FILE
Supersede the */proc/*pid*/exe* symbolic link with a new one pointing to a new executable file identified by the file descriptor provided in *arg3* argument. The file descriptor should be obtained with a regular **open**(2) call.

To change the symbolic link, one needs to unmap all existing executable memory areas, including those created by the kernel itself (for example the kernel usually creates at least one executable memory area for the ELF *.text* section).

In Linux 4.9 and earlier, the **PR_SET_MM_EXE_FILE** operation can be performed only once in a process's lifetime; attempting to perform the operation a second time results in the error **EPERM**. This restriction was enforced for security reasons that were subsequently deemed specious, and the restriction was removed in Linux 4.10 because some user-space applications needed to perform this operation more than once.

The following options are available since Linux 3.18.

PR_SET_MM_MAP
Provides one-shot access to all the addresses by passing in a *struct prctl_mm_map* (as defined in <*linux/prctl.h*>). The*ar g4* argument should provide the size of the struct.

This feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled.

PR_SET_MM_MAP_SIZE
Returns the size of the *struct prctl_mm_map* the kernel expects. This allows user space to find a compatible struct. The *arg4* argument should be a pointer to an unsigned int.

This feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled.

PR_SET_VMA (since Linux 5.17)
Sets an attribute specified in *arg2* for virtual memory areas starting from the address specified in *arg3* and spanning the size specified in *arg4*. *arg5* specifies the value of the attribute to be set.

Note that assigning an attribute to a virtual memory area might prevent it from being merged with adjacent virtual memory areas due to the difference in that attribute's value.

Currently, *arg2* must be one of:

PR_SET_VMA_ANON_NAME
Set a name for anonymous virtual memory areas. *ar g5* should be a pointer to a null-terminated string containing the name. The name length including null byte cannot exceed 80 bytes. If *arg5* is NULL, the name of the appropriate anonymous virtual memory areas will be reset. The name can contain only printable ascii characters (including space), except '[', ']', '\', '$', and '`'.

PR_MPX_ENABLE_MANAGEMENT, **PR_MPX_DISABLE_MANAGEMENT** (since Linux 3.19, removed in Linux 5.4; only on x86)

Enable or disable kernel management of Memory Protection eXtensions (MPX) bounds tables. The *arg2*, *arg3*, *arg4*, and *arg5* arguments must be zero.

MPX is a hardware-assisted mechanism for performing bounds checking on pointers. It consists of a set of registers storing bounds information and a set of special instruction prefixes that tell the CPU on which instructions it should do bounds enforcement. There is a limited number of these registers and when there are more pointers than registers, their contents must be "spilled" into a set of tables. These tables are called "bounds tables" and the MPX **prctl**() operations control whether the kernel manages their allocation and freeing.

When management is enabled, the kernel will take over allocation and freeing of the bounds tables. It does this by trapping the #BR exceptions that result at first use of missing bounds tables and instead of delivering the exception to user space, it allocates the table and populates the bounds directory with the location of the new table. For freeing, the kernel checks to see if bounds tables are present for memory which is not allocated, and frees them if so.

Before enabling MPX management using **PR_MPX_ENABLE_MANAGEMENT**, the application must first have allocated a user-space buffer for the bounds directory and placed the location of that directory in the *bndcfgu* register.

These calls fail if the CPU or kernel does not support MPX. Kernel support for MPX is enabled via the **CONFIG_X86_INTEL_MPX** configuration option. You can check whether the CPU supports MPX by looking for the *mpx* CPUID bit, like with the following command:

```
cat /proc/cpuinfo | grep ' mpx '
```

A thread may not switch in or out of long (64-bit) mode while MPX is enabled.

All threads in a process are affected by these calls.

The child of a **fork**(2) inherits the state of MPX management. During **execve**(2), MPX management is reset to a state as if **PR_MPX_DISABLE_MANAGEMENT** had been called.

For further information on Intel MPX, see the kernel source file *Documentation/x86/intel_mpx.txt*.

Due to a lack of toolchain support, **PR_MPX_ENABLE_MANAGEMENT** and **PR_MPX_DISABLE_MANAGEMENT** are not supported in Linux 5.4 and later.

PR_SET_NAME (since Linux 2.6.9)

Set the name of the calling thread, using the value in the location pointed to by *(char *) arg2*. The name can be up to 16 bytes long, including the terminating null byte. (If the length of the string, including the terminating null byte, exceeds 16 bytes, the string is silently truncated.) This is the same attribute that can be set via **pthread_setname_np**(3) and retrieved using **pthread_getname_np**(3). The attribute is likewise accessible via */proc/self/task/*tid*/comm* (see **proc**(5)), where *tid* is the thread ID of the calling thread, as returned by **gettid**(2).

PR_GET_NAME (since Linux 2.6.11)

Return the name of the calling thread, in the buffer pointed to by *(char *) arg2*. The buffer should allow space for up to 16 bytes; the returned string will be null-terminated.

PR_SET_NO_NEW_PRIVS (since Linux 3.5)

Set the calling thread's *no_new_privs* attribute to the value in *arg2*. With *no_new_privs* set to 1, **execve**(2) promises not to grant privileges to do anything that could not have been done without the **execve**(2) call (for example, rendering the set-user-ID and set-group-ID mode bits, and file capabilities non-functional). Once set, the *no_new_privs* attribute cannot be unset. The setting of this attribute is inherited by children created by **fork**(2) and **clone**(2), and preserved across **execve**(2).

Since Linux 4.10, the value of a thread's *no_new_privs* attribute can be viewed via the *NoNewPrivs* field in the */proc/*pid*/status* file.

For more information, see the kernel source file *Documentation/userspace−api/no_new_privs.rst* (or *Documentation/prctl/no_new_privs.txt* before Linux 4.13). See also **seccomp**(2).

PR_GET_NO_NEW_PRIVS (since Linux 3.5)
Return (as the function result) the value of the *no_new_privs* attribute for the calling thread. A value of 0 indicates the regular **execve**(2) behavior. A value of 1 indicates **execve**(2) will operate in the privilege-restricting mode described above.

PR_PAC_RESET_KEYS (since Linux 5.0, only on arm64)
Securely reset the thread's pointer authentication keys to fresh random values generated by the kernel.

The set of keys to be reset is specified by *arg2*, which must be a logical OR of zero or more of the following:

PR_PAC_APIAKEY
instruction authentication key A

PR_PAC_APIBKEY
instruction authentication key B

PR_PAC_APDAKEY
data authentication key A

PR_PAC_APDBKEY
data authentication key B

PR_PAC_APGAKEY
generic authentication "A" key.

(Yes folks, there really is no generic B key.)

As a special case, if *arg2* is zero, then all the keys are reset. Since new keys could be added in future, this is the recommended way to completely wipe the existing keys when establishing a clean execution context. Note that there is no need to use**PR_P AC_RESET_KEYS** in preparation for calling **execve**(2), since **execve**(2) resets all the pointer authentication keys.

The remaining arguments *arg3*, *arg4*, and *arg5* must all be zero.

If the arguments are invalid, and in particular if *arg2* contains set bits that are unrecognized or that correspond to a key not available on this platform, then the call fails with error **EINVAL**.

Warning: Because the compiler or run-time environment may be using some or all of the keys, a successful **PR_PAC_RESET_KEYS** may crash the calling process. The conditions for using it safely are complex and system-dependent. Don't use it unless you know what you are doing.

For more information, see the kernel source file *Documentation/arm64/pointer−authentication.rst* (or *Documentation/arm64/pointer−authentication.txt* before Linux 5.3).

PR_SET_PDEATHSIG (since Linux 2.1.57)
Set the parent-death signal of the calling process to *arg2* (either a signal value in the range 1..**NSIG**−1**,** or 0 to clear). This is the signal that the calling process will get when its parent dies.

Warning: the "parent" in this case is considered to be the *thread* that created this process. In other words, the signal will be sent when that thread terminates (via, for example, **pthread_exit**(3)), rather than after all of the threads in the parent process terminate.

The parent-death signal is sent upon subsequent termination of the parent thread and also upon termination of each subreaper process (see the description of **PR_SET_CHILD_SUBREAPER** above) to which the caller is subsequently reparented. If the parent thread and all ancestor subreapers have already terminated by the time of the **PR_SET_PDEATHSIG** operation, then no parent-death signal is sent to the caller.

The parent-death signal is process-directed (see **signal**(7)) and, if the child installs a handler using the **sigaction**(2) **SA_SIGINFO** flag, the *si_pid* field of the *siginfo_t* argument of the handler

contains the PID of the terminating parent process.

The parent-death signal setting is cleared for the child of a **fork**(2). It is also (since Linux 2.4.36 / 2.6.23) cleared when executing a set-user-ID or set-group-ID binary, or a binary that has associated capabilities (see **capabilities**(7)); otherwise, this value is preserved across **execve**(2). The parent-death signal setting is also cleared upon changes to any of the following thread credentials: effective user ID, effective group ID, filesystem user ID, or filesystem group ID.

PR_GET_PDEATHSIG (since Linux 2.3.15)
Return the current value of the parent process death signal, in the location pointed to by *(int *) arg2*.

PR_SET_PTRACER (since Linux 3.4)
This is meaningful only when the Yama LSM is enabled and in mode 1 ("restricted ptrace", visible via */proc/sys/kernel/yama/ptrace_scope*). When a "ptracer process ID" is passed in*ar g2*, the caller is declaring that the ptracer process can **ptrace**(2) the calling process as if it were a direct process ancestor. Each**PR_SET_PTRA CER** operation replaces the previous "ptracer process ID". Employing **PR_SET_PTRACER** with *arg2* set to 0 clears the caller's "ptracer process ID". If *arg2* is **PR_SET_PTRACER_ANY**, the ptrace restrictions introduced by Yama are effectively disabled for the calling process.

For further information, see the kernel source file *Documentation/admin−guide/LSM/Yama.rst* (or *Documentation/security/Yama.txt* before Linux 4.13).

PR_SET_SECCOMP (since Linux 2.6.23)
Set the secure computing (seccomp) mode for the calling thread, to limit the available system calls. The more recent**seccomp**(2) system call pro vides a superset of the functionality of **PR_SET_SECCOMP**, and is the preferred interface for new applications.

The seccomp mode is selected via *arg2*. (The seccomp constants are defined in<*linux/seccomp.h*>.) The following values can be specified:

SECCOMP_MODE_STRICT (since Linux 2.6.23)
See the description of **SECCOMP_SET_MODE_STRICT** in **seccomp**(2).

This operation is available only if the kernel is configured with **CONFIG_SECCOMP** enabled.

SECCOMP_MODE_FILTER (since Linux 3.5)
The allowed system calls are defined by a pointer to a Berkeley Packet Filter passed in *arg3*. This argument is a pointer to *struct sock_fprog*; it can be designed to filter arbitrary system calls and system call arguments. See the description of**SECCOMP_SET_MODE_FILTER** in **seccomp**(2).

This operation is available only if the kernel is configured with **CONFIG_SECCOMP_FILTER** enabled.

For further details on seccomp filtering, see **seccomp**(2).

PR_GET_SECCOMP (since Linux 2.6.23)
Return (as the function result) the secure computing mode of the calling thread. If the caller is not in secure computing mode, this operation returns 0; if the caller is in strict secure computing mode, then the **prctl**() call will cause a **SIGKILL** signal to be sent to the process. If the caller is in filter mode, and this system call is allowed by the seccomp filters, it returns 2; otherwise, the process is killed with a **SIGKILL** signal.

This operation is available only if the kernel is configured with **CONFIG_SECCOMP** enabled.

Since Linux 3.8, the *Seccomp* field of the */proc/*pid*/status* file provides a method of obtaining the same information, without the risk that the process is killed; see **proc**(5).

PR_SET_SECUREBITS (since Linux 2.6.26)
Set the "securebits" flags of the calling thread to the value supplied in *arg2*. See**capabilities**(7).

PR_GET_SECUREBITS (since Linux 2.6.26)
Return (as the function result) the "securebits" flags of the calling thread. See **capabilities**(7).

PR_GET_SPECULATION_CTRL (since Linux 4.17)
Return (as the function result) the state of the speculation misfeature specified in *arg2*. Currently, the only permitted value for this argument is **PR_SPEC_STORE_BYPASS** (otherwise the call fails with the error **ENODEV**).

The return value uses bits 0-3 with the following meaning:

PR_SPEC_PRCTL
Mitigation can be controlled per thread by **PR_SET_SPECULATION_CTRL**.

PR_SPEC_ENABLE
The speculation feature is enabled, mitigation is disabled.

PR_SPEC_DISABLE
The speculation feature is disabled, mitigation is enabled.

PR_SPEC_FORCE_DISABLE
Same as **PR_SPEC_DISABLE** but cannot be undone.

PR_SPEC_DISABLE_NOEXEC (since Linux 5.1)
Same as **PR_SPEC_DISABLE**, but the state will be cleared on **execve**(2).

If all bits are 0, then the CPU is not affected by the speculation misfeature.

If **PR_SPEC_PRCTL** is set, then per-thread control of the mitigation is available. If not set, **prctl**() for the speculation misfeature will fail.

The *arg3*, *arg4*, and *arg5* arguments must be specified as 0; otherwise the call fails with the error **EINVAL**.

PR_SET_SPECULATION_CTRL (since Linux 4.17)
Sets the state of the speculation misfeature specified in *arg2*. The speculation-misfeature settings are per-thread attributes.

Currently, *arg2* must be one of:

PR_SPEC_STORE_BYPASS
Set the state of the speculative store bypass misfeature.

PR_SPEC_INDIRECT_BRANCH (since Linux 4.20)
Set the state of the indirect branch speculation misfeature.

If *arg2* does not have one of the above values, then the call fails with the error **ENODEV**.

The *arg3* argument is used to hand in the control value, which is one of the following:

PR_SPEC_ENABLE
The speculation feature is enabled, mitigation is disabled.

PR_SPEC_DISABLE
The speculation feature is disabled, mitigation is enabled.

PR_SPEC_FORCE_DISABLE
Same as **PR_SPEC_DISABLE**, but cannot be undone. A subsequent **prctl**(*arg2*, **PR_SPEC_ENABLE**) with the same value for *arg2* will fail with the error **EPERM**.

PR_SPEC_DISABLE_NOEXEC (since Linux 5.1)
Same as **PR_SPEC_DISABLE**, but the state will be cleared on **execve**(2). Currently only supported for *arg2* equal to **PR_SPEC_STORE_BYPASS.**

Any unsupported value in *arg3* will result in the call failing with the error **ERANGE**.

The *arg4* and *arg5* arguments must be specified as 0; otherwise the call fails with the error **EINVAL**.

The speculation feature can also be controlled by the **spec_store_bypass_disable** boot parameter. This parameter may enforce a read-only policy which will result in the **prctl**() call failing with the error **ENXIO**. For further details, see the kernel source file *Documentation/admin−guide/kernel−parameters.txt*.

PR_SVE_SET_VL (since Linux 4.15, only on arm64)
Configure the thread's SVE vector length, as specified by *(int) arg2*. Arguments *arg3*, *arg4*, and *arg5* are ignored.

The bits of *arg2* corresponding to **PR_SVE_VL_LEN_MASK** must be set to the desired vector length in bytes. This is interpreted as an upper bound: the kernel will select the greatest available vector length that does not exceed the value specified. In particular, specifying **SVE_VL_MAX** (defined in*<asm/sigconte xt.h>)* for the **PR_SVE_VL_LEN_MASK** bits requests the maximum supported vector length.

In addition, the other bits of *arg2* must be set to one of the following combinations of flags:

0
Perform the change immediately. At the next **execve**(2) in the thread, the vector length will be reset to the value configured in */proc/sys/abi/sve_default_vector_length*.

PR_SVE_VL_INHERIT
Perform the change immediately. Subsequent**execv e**(2) calls will preserve the new vector length.

PR_SVE_SET_VL_ONEXEC
Defer the change, so that it is performed at the next **execve**(2) in the thread. Further **execve**(2) calls will reset the vector length to the value configured in */proc/sys/abi/sve_default_vector_length*.

PR_SVE_SET_VL_ONEXEC | PR_SVE_VL_INHERIT
Defer the change, so that it is performed at the next **execve**(2) in the thread. Further **execve**(2) calls will preserve the new vector length.

In all cases, any previously pending deferred change is canceled.

The call fails with error **EINVAL** if SVE is not supported on the platform, if *arg2* is unrecognized or invalid, or the value in the bits of *arg2* corresponding to **PR_SVE_VL_LEN_MASK** is outside the range **SVE_VL_MIN**..**SVE_VL_MAX** or is not a multiple of 16.

On success, a nonnegative value is returned that describes the *selected* configuration. If **PR_SVE_SET_VL_ONEXEC** was included in *arg2*, then the configuration described by the return value will take effect at the next **execve**(2). Otherwise, the configuration is already in effect when the **PR_SVE_SET_VL** call returns. In either case, the value is encoded in the same way as the return value of **PR_SVE_GET_VL**. Note that there is no explicit flag in the return value corresponding to **PR_SVE_SET_VL_ONEXEC**.

The configuration (including any pending deferred change) is inherited across **fork**(2) and **clone**(2).

For more information, see the kernel source file *Documentation/arm64/sve.rst* (or *Documentation/arm64/sve.txt* before Linux 5.3).

Warning: Because the compiler or run-time environment may be using SVE, using this call without the **PR_SVE_SET_VL_ONEXEC** flag may crash the calling process. The conditions for using it safely are complex and system-dependent. Don't use it unless you really know what you are doing.

PR_SVE_GET_VL (since Linux 4.15, only on arm64)
Get the thread's current SVE vector length configuration.

Arguments *arg2*, *arg3*, *arg4*, and *arg5* are ignored.

Provided that the kernel and platform support SVE, this operation always succeeds, returning a nonnegative value that describes the *current* configuration. The bits corresponding to **PR_SVE_VL_LEN_MASK** contain the currently configured vector length in bytes. The bit corresponding to **PR_SVE_VL_INHERIT** indicates whether the vector length will be inherited across **execve**(2).

Note that there is no way to determine whether there is a pending vector length change that has not yet taken effect.

For more information, see the kernel source file *Documentation/arm64/sve.rst* (or *Documentation/arm64/sve.txt* before Linux 5.3).

PR_SET_SYSCALL_USER_DISPATCH (since Linux 5.11, x86 only)
Configure the Syscall User Dispatch mechanism for the calling thread. This mechanism allows an application to selectively intercept system calls so that they can be handled within the application itself. Interception takes the form of a thread-directed **SIGSYS** signal that is delivered to the thread when it makes a system call. If intercepted, the system call is not executed by the kernel.

To enable this mechanism, *arg2* should be set to **PR_SYS_DISPATCH_ON**. Once enabled, further system calls will be selectively intercepted, depending on a control variable provided by user space. In this case,*ar g3* and *arg4* respectively identify the *offset* and *length* of a single contiguous memory region in the process address space from where system calls are always allowed to be executed, regardless of the control variable. (Typically, this area would include the area of memory containing the C library.)

arg5 points to a char-sized variable that is a fast switch to allow/block system call execution without the overhead of doing another system call to re-configure Syscall User Dispatch. This control variable can either be set to **SYSCALL_DISPATCH_FILTER_BLOCK** to block system calls from executing or to **SYSCALL_DISPATCH_FILTER_ALLOW** to temporarily allow them to be executed. This value is checked by the kernel on every system call entry, and any unexpected value will raise an uncatchable **SIGSYS** at that time, killing the application.

When a system call is intercepted, the kernel sends a thread-directed **SIGSYS** signal to the triggering thread. Various fields will be set in the *siginfo_t* structure (see **sigaction**(2)) associated with the signal:

- *si_signo* will contain **SIGSYS**.
- *si_call_addr* will show the address of the system call instruction.
- *si_syscall* and *si_arch* will indicate which system call was attempted.
- *si_code* will contain **SYS_USER_DISPATCH**.
- *si_errno* will be set to 0.

The program counter will be as though the system call happened (i.e., the program counter will not point to the system call instruction).

When the signal handler returns to the kernel, the system call completes immediately and returns to the calling thread, without actually being executed. If necessary (i.e., when emulating the system call on user space.), the signal handler should set the system call return value to a sane value, by modifying the register context stored in the *ucontext* argument of the signal handler. See**sigaction**(2), **sigreturn**(2), and **getcontext**(3) for more information.

If *arg2* is set to **PR_SYS_DISPATCH_OFF**, Syscall User Dispatch is disabled for that thread. the remaining arguments must be set to 0.

The setting is not preserved across **fork**(2), **clone**(2), or **execve**(2).

For more information, see the kernel source file *Documentation/admin-guide/syscall−user−dispatch.rst*

PR_SET_TAGGED_ADDR_CTRL (since Linux 5.4, only on arm64)
Controls support for passing tagged user-space addresses to the kernel (i.e., addresses where bits 56—63 are not all zero).

The level of support is selected by *arg2*, which can be one of the following:

0 Addresses that are passed for the purpose of being dereferenced by the kernel must be untagged.

PR_TAGGED_ADDR_ENABLE
Addresses that are passed for the purpose of being dereferenced by the kernel may be tagged, with the exceptions summarized below.

The remaining arguments *arg3*, *arg4*, and *arg5* must all be zero.

On success, the mode specified in *arg2* is set for the calling thread and the return value is 0. If the arguments are invalid, the mode specified in *arg2* is unrecognized, or if this feature is unsupported by the kernel or disabled via */proc/sys/abi/tagged_addr_disabled*, the call fails with the error **EINVAL**.

In particular, if **prctl**(**PR_SET_TAGGED_ADDR_CTRL**, 0, 0, 0, 0) fails with **EINVAL**, then all addresses passed to the kernel must be untagged.

Irrespective of which mode is set, addresses passed to certain interfaces must always be untagged:

- **brk**(2), **mmap**(2), **shmat**(2), **shmdt**(2), and the *new_address* argument of **mremap**(2).

  (Prior to Linux 5.6 these accepted tagged addresses, but the behaviour may not be what you expect. Don't rely on it.)

- 'polymorphic' interfaces that accept pointers to arbitrary types cast to a *void* * or other generic type, specifically **prctl**(), **ioctl**(2), and in general **setsockopt**(2) (only certain specific **setsockopt**(2) options allow tagged addresses).

This list of exclusions may shrink when moving from one kernel version to a later kernel version. While the kernel may make some guarantees for backwards compatibility reasons, for the purposes of new software the effect of passing tagged addresses to these interfaces is unspecified.

The mode set by this call is inherited across **fork**(2) and **clone**(2). The mode is reset by**execv e**(2) to 0 (i.e., tagged addresses not permitted in the user/kernel ABI).

For more information, see the kernel source file *Documentation/arm64/tagged−address−abi.rst*.

Warning: This call is primarily intended for use by the run-time environment. A successful **PR_SET_TAGGED_ADDR_CTRL** call elsewhere may crash the calling process. The conditions for using it safely are complex and system-dependent. Don't use it unless you know what you are doing.

PR_GET_TAGGED_ADDR_CTRL (since Linux 5.4, only on arm64)
Returns the current tagged address mode for the calling thread.

Arguments *arg2*, *arg3*, *arg4*, and *arg5* must all be zero.

If the arguments are invalid or this feature is disabled or unsupported by the kernel, the call fails with **EINVAL**. In particular, if **prctl**(**PR_GET_TAGGED_ADDR_CTRL**, 0, 0, 0, 0) fails with **EINVAL**, then this feature is definitely either unsupported, or disabled via */proc/sys/abi/tagged_addr_disabled*. In this case, all addresses passed to the kernel must be untagged.

Otherwise, the call returns a nonnegative value describing the current tagged address mode, encoded in the same way as the *arg2* argument of **PR_SET_TAGGED_ADDR_CTRL**.

For more information, see the kernel source file *Documentation/arm64/tagged−address−abi.rst*.

PR_TASK_PERF_EVENTS_DISABLE (since Linux 2.6.31)
Disable all performance counters attached to the calling process, regardless of whether the counters were created by this process or another process. Performance counters created by the calling process for other processes are unaffected. For more information on performance counters, see the Linux kernel source file *tools/perf/design.txt*.

Originally called **PR_TASK_PERF_COUNTERS_DISABLE**; renamed (retaining the same numerical value) in Linux 2.6.32.

PR_TASK_PERF_EVENTS_ENABLE (since Linux 2.6.31)
The converse of **PR_TASK_PERF_EVENTS_DISABLE**; enable performance counters attached to the calling process.

Originally called **PR_TASK_PERF_COUNTERS_ENABLE**; renamed in Linux 2.6.32.

PR_SET_THP_DISABLE (since Linux 3.15)
Set the state of the "THP disable" flag for the calling thread. If *arg2* has a nonzero value, the flag is set, otherwise it is cleared. Setting this flag provides a method for disabling transparent huge pages for jobs where the code cannot be modified, and using a malloc hook with **madvise**(2) is not an option (i.e., statically allocated data). The setting of the "THP disable" flag is inherited by a child created via **fork**(2) and is preserved across **execve**(2).

PR_GET_THP_DISABLE (since Linux 3.15)
Return (as the function result) the current setting of the "THP disable" flag for the calling thread: either 1, if the flag is set, or 0, if it is not.

PR_GET_TID_ADDRESS (since Linux 3.5)
Return the *clear_child_tid* address set by **set_tid_address**(2) and the **clone**(2) **CLONE_CHILD_CLEARTID** flag, in the location pointed to by *(int **) arg2*. This feature is available only if the kernel is built with the **CONFIG_CHECKPOINT_RESTORE** option enabled. Note that since the**pr ctl**() system call does not have a compat implementation for the AMD64 x32 and MIPS n32 ABIs, and the kernel writes out a pointer using the kernel's pointer size, this operation expects a user-space buffer of 8 (not 4) bytes on these ABIs.

PR_SET_TIMERSLACK (since Linux 2.6.28)
Each thread has two associated timer slack values: a "default" value, and a "current" value. This operation sets the "current" timer slack value for the calling thread. *ar g2* is an unsigned long value, then maximum "current" value is ULONG_MAX and the minimum "current" value is 1. If the nanosecond value supplied in *arg2* is greater than zero, then the "current" value is set to this value. If*ar g2* is equal to zero, the "current" timer slack is reset to the thread's "default" timer slack value.

The "current" timer slack is used by the kernel to group timer expirations for the calling thread that are close to one another; as a consequence, timer expirations for the thread may be up to the specified number of nanoseconds late (but will never expire early). Grouping timer expirations can help reduce system power consumption by minimizing CPU wake-ups.

The timer expirations affected by timer slack are those set by **select**(2), **pselect**(2), **poll**(2), **ppoll**(2), **epoll_wait**(2), **epoll_pwait**(2), **clock_nanosleep**(2), **nanosleep**(2), and **futex**(2) (and thus the library functions implemented via futexes, including **pthread_cond_timedwait**(3), **pthread_mutex_timedlock**(3), **pthread_rwlock_timedrdlock**(3), **pthread_rwlock_timedwrlock**(3), and **sem_timedwait**(3)).

Timer slack is not applied to threads that are scheduled under a real-time scheduling policy (see **sched_setscheduler**(2)).

When a new thread is created, the two timer slack values are made the same as the "current" value of the creating thread. Thereafter, a thread can adjust its "current" timer slack value via **PR_SET_TIMERSLACK**. The "default" value can't be changed. The timer slack values of *init* (PID 1), the ancestor of all processes, are 50,000 nanoseconds (50 microseconds). The timer slack value is inherited by a child created via **fork**(2), and is preserved across **execve**(2).

Since Linux 4.6, the "current" timer slack value of any process can be examined and changed via the file */proc/*pid*/timerslack_ns*. See**pr oc**(5).

PR_GET_TIMERSLACK (since Linux 2.6.28)
Return (as the function result) the "current" timer slack value of the calling thread.

PR_SET_TIMING (since Linux 2.6.0)
Set whether to use (normal, traditional) statistical process timing or accurate timestamp-based process timing, by passing **PR_TIMING_STATISTICAL** or **PR_TIMING_TIMESTAMP** to *arg2*. **PR_TIMING_TIMESTAMP** is not currently implemented (attempting to set this mode will yield the error **EINVAL**).

PR_GET_TIMING (since Linux 2.6.0)
Return (as the function result) which process timing method is currently in use.

PR_SET_TSC (since Linux 2.6.26, x86 only)
Set the state of the flag determining whether the timestamp counter can be read by the process. Pass **PR_TSC_ENABLE** to *arg2* to allow it to be read, or **PR_TSC_SIGSEGV** to generate a **SIGSEGV** when the process tries to read the timestamp counter.

PR_GET_TSC (since Linux 2.6.26, x86 only)
Return the state of the flag determining whether the timestamp counter can be read, in the location pointed to by *(int *) arg2*.

PR_SET_UNALIGN
(Only on: ia64, since Linux 2.3.48; parisc, since Linux 2.6.15; PowerPC, since Linux 2.6.18; Alpha, since Linux 2.6.22; sh, since Linux 2.6.34; tile, since Linux 3.12) Set unaligned access control bits to *arg2*. Pass **PR_UNALIGN_NOPRINT** to silently fix up unaligned user accesses, or **PR_UNALIGN_SIGBUS** to generate **SIGBUS** on unaligned user access. Alpha also supports an additional flag with the value of 4 and no corresponding named constant, which instructs kernel to not fix up unaligned accesses (it is analogous to providing the **UAC_NOFIX** flag in **SSI_NVPAIRS** operation of the **setsysinfo**() system call on Tru64).

PR_GET_UNALIGN
(See **PR_SET_UNALIGN** for information on versions and architectures.) Return unaligned access control bits, in the location pointed to by *(unsigned int *) arg2*.

RETURN VALUE

On success, **PR_CAP_AMBIENT**+**PR_CAP_AMBIENT_IS_SET**, **PR_CAPBSET_READ**, **PR_GET_DUMPABLE**, **PR_GET_FP_MODE**, **PR_GET_IO_FLUSHER**, **PR_GET_KEEPCAPS**, **PR_MCE_KILL_GET**, **PR_GET_NO_NEW_PRIVS**, **PR_GET_SECUREBITS**, **PR_GET_SPECULATION_CTRL**, **PR_SVE_GET_VL**, **PR_SVE_SET_VL**, **PR_GET_TAGGED_ADDR_CTRL**, **PR_GET_THP_DISABLE**, **PR_GET_TIMING**, **PR_GET_TIMERSLACK**, and (if it returns) **PR_GET_SECCOMP** return the nonnegative values described above. All other*option* v alues return 0 on success. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EACCES
option is **PR_SET_SECCOMP** and *arg2* is **SECCOMP_MODE_FILTER**, but the process does not have the **CAP_SYS_ADMIN** capability or has not set the *no_new_privs* attribute (see the discussion of **PR_SET_NO_NEW_PRIVS** above).

EACCES
option is **PR_SET_MM**, and *arg3* is **PR_SET_MM_EXE_FILE**, the file is not executable.

EBADF
option is **PR_SET_MM**, *arg3* is **PR_SET_MM_EXE_FILE**, and the file descriptor passed in *arg4* is not valid.

EBUSY
option is **PR_SET_MM**, *arg3* is **PR_SET_MM_EXE_FILE**, and this the second attempt to change the */proc/*pid*/exe* symbolic link, which is prohibited.

EFAULT
arg2 is an invalid address.

EFAULT
option is **PR_SET_SECCOMP**, *arg2* is **SECCOMP_MODE_FILTER**, the system was built with **CONFIG_SECCOMP_FILTER**, and *arg3* is an invalid address.

EFAULT
option is **PR_SET_SYSCALL_USER_DISPATCH** and *arg5* has an invalid address.

EINVAL
The value of *option* is not recognized, or not supported on this system.

EINVAL
option is **PR_MCE_KILL** or **PR_MCE_KILL_GET** or **PR_SET_MM**, and unused **prctl**() arguments were not specified as zero.

EINVAL
arg2 is not valid value for this *option*.

EINVAL
option is **PR_SET_SECCOMP** or **PR_GET_SECCOMP**, and the kernel was not configured with **CONFIG_SECCOMP**.

EINVAL
option is **PR_SET_SECCOMP**, *arg2* is **SECCOMP_MODE_FILTER**, and the kernel was not configured with **CONFIG_SECCOMP_FILTER**.

EINVAL
option is **PR_SET_MM**, and one of the following is true

- *arg4* or *arg5* is nonzero;
- *arg3* is greater than **TASK_SIZE** (the limit on the size of the user address space for this architecture);
- *arg2* is **PR_SET_MM_START_CODE**, **PR_SET_MM_END_CODE**, **PR_SET_MM_START_DATA**, **PR_SET_MM_END_DATA**, or **PR_SET_MM_START_STACK**, and the permissions of the corresponding memory area are not as required;
- *arg2* is **PR_SET_MM_START_BRK** or **PR_SET_MM_BRK**, and *arg3* is less than or equal to the end of the data segment or specifies a value that would cause the **RLIMIT_DATA** resource limit to be exceeded.

EINVAL
option is **PR_SET_PTRACER** and *arg2* is not 0, **PR_SET_PTRACER_ANY**, or the PID of an existing process.

EINVAL
option is **PR_SET_PDEATHSIG** and *arg2* is not a valid signal number.

EINVAL
option is **PR_SET_DUMPABLE** and *arg2* is neither **SUID_DUMP_DISABLE** nor **SUID_DUMP_USER**.

EINVAL
option is **PR_SET_TIMING** and *arg2* is not **PR_TIMING_STATISTICAL**.

EINVAL
option is **PR_SET_NO_NEW_PRIVS** and *arg2* is not equal to 1 or *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
option is **PR_GET_NO_NEW_PRIVS** and *arg2*, *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
option is **PR_SET_THP_DISABLE** and *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
option is **PR_GET_THP_DISABLE** and *arg2*, *arg3*, *arg4*, or *arg5* is nonzero.

EINVAL
option is **PR_CAP_AMBIENT** and an unused argument (*arg4*, *arg5*, or, in the case of **PR_CAP_AMBIENT_CLEAR_ALL**, *arg3*) is nonzero; or *arg2* has an invalid value; or *arg2* is **PR_CAP_AMBIENT_LOWER**, **PR_CAP_AMBIENT_RAISE**, or**PR_CAP_AMBIENT_IS_SET** and *arg3* does not specify a valid capability.

EINVAL
option was **PR_GET_SPECULATION_CTRL** or **PR_SET_SPECULATION_CTRL** and unused arguments to **prctl**() are not 0. **EINVAL** *option* is **PR_PAC_RESET_KEYS** and the arguments are invalid or unsupported. See the description of **PR_PAC_RESET_KEYS** above for details.

EINVAL
option is **PR_SVE_SET_VL** and the arguments are invalid or unsupported, or SVE is not available on this platform. See the description of **PR_SVE_SET_VL** above for details.

EINVAL
option is **PR_SVE_GET_VL** and SVE is not available on this platform.

EINVAL
option is **PR_SET_SYSCALL_USER_DISPATCH** and one of the following is true:

- *arg2* is **PR_SYS_DISPATCH_OFF** and the remaining arguments are not 0;
- *arg2* is **PR_SYS_DISPATCH_ON** and the memory range specified is outside the address space of the process.
- *arg2* is invalid.

EINVAL
option is **PR_SET_TAGGED_ADDR_CTRL** and the arguments are invalid or unsupported. See the description of **PR_SET_TAGGED_ADDR_CTRL** above for details.

EINVAL
option is **PR_GET_TAGGED_ADDR_CTRL** and the arguments are invalid or unsupported. See the description of **PR_GET_TAGGED_ADDR_CTRL** above for details.

ENODEV
option was **PR_SET_SPECULATION_CTRL** the kernel or CPU does not support the requested speculation misfeature.

ENXIO
option was **PR_MPX_ENABLE_MANAGEMENT** or **PR_MPX_DISABLE_MANAGEMENT** and the kernel or the CPU does not support MPX management. Check that the kernel and processor have MPX support.

ENXIO
option was **PR_SET_SPECULATION_CTRL** implies that the control of the selected speculation misfeature is not possible. See **PR_GET_SPECULATION_CTRL** for the bit fields to determine which option is available.

EOPNOTSUPP
option is **PR_SET_FP_MODE** and *arg2* has an invalid or unsupported value.

EPERM
option is **PR_SET_SECUREBITS**, and the caller does not have the **CAP_SETPCAP** capability, or tried to unset a "locked" flag, or tried to set a flag whose corresponding locked flag was set (see **capabilities**(7)).

EPERM
option is **PR_SET_SPECULATION_CTRL** wherein the speculation was disabled with **PR_SPEC_FORCE_DISABLE** and caller tried to enable it again.

EPERM
option is **PR_SET_KEEPCAPS**, and the caller's **SECBIT_KEEP_CAPS_LOCKED** flag is set (see **capabilities**(7)).

EPERM
option is **PR_CAPBSET_DROP**, and the caller does not have the **CAP_SETPCAP** capability.

EPERM
option is **PR_SET_MM**, and the caller does not have the **CAP_SYS_RESOURCE** capability.

EPERM
option is **PR_CAP_AMBIENT** and *arg2* is **PR_CAP_AMBIENT_RAISE**, but either the capability specified in *arg3* is not present in the process's permitted and inheritable capability sets, or the **PR_CAP_AMBIENT_LOWER** securebit has been set.

ERANGE
option was **PR_SET_SPECULATION_CTRL** and *arg3* is not **PR_SPEC_ENABLE**, **PR_SPEC_DISABLE**, **PR_SPEC_FORCE_DISABLE**, nor **PR_SPEC_DISABLE_NOEXEC**.

VERSIONS

The **prctl**() system call was introduced in Linux 2.1.57.

STANDARDS

This call is Linux-specific. IRIX has a **prctl**() system call (also introduced in Linux 2.1.44 as irix_prctl on the MIPS architecture), with prototype

```
ptrdiff_t prctl(intoption, intarg2, intarg3);
```

and options to get the maximum number of processes per user, get the maximum number of processors the calling process can use, find out whether a specified process is currently blocked, get or set the maximum stack size, and so on.

SEE ALSO

signal(2), **core**(5)