NAME

ptrace − process trace

SYNOPSIS

#include <sys/ptrace.h>

long ptrace(enum __ptrace_request *request***, pid_t** *pid***,**
void * *addr***, void *** *data***);**

DESCRIPTION

The **ptrace**() system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.

The parent can initiate a trace by calling **fork**(2) and having the resulting child do a **PTRACE_TRACEME**, followed (typically) by an **exec**(3). Alternatively, the parent may commence trace of an existing process using **PTRACE_ATTACH**.

While being traced, the child will stop each time a signal is delivered, even if the signal is being ignored. (The exception is **SIGKILL**, which has its usual effect.) The parent will be notified at its next **wait**(2) and may inspect and modify the child process while it is stopped. The parent then causes the child to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

When the parent is finished tracing, it can terminate the child with **PTRACE_KILL** or cause it to continue executing in a normal, untraced mode via **PTRACE_DETACH**.

The value of *request* determines the action to be performed:

PTRACE_TRACEME
Indicates that this process is to be traced by its parent. Any signal (except **SIGKILL**) delivered to this process will cause it to stop and its parent to be notified via **wait**(2). Also, all subsequent calls to **execve**(2) by this process will cause a **SIGTRAP** to be sent to it, giving the parent a chance to gain control before the new program begins execution. A process probably shouldn't make this request if its parent isn't expecting to trace it. (*pid*, *addr*, and *data* are ignored.)

The above request is used only by the child process; the rest are used only by the parent. In the following requests, *pid* specifies the child process to be acted on. For requests other than **PTRACE_KILL**, the child process must be stopped.

PTRACE_PEEKTEXT, **PTRACE_PEEKDATA**
Reads a word at the location *addr* in the child's memory, returning the word as the result of the **ptrace**() call. Linux does not have separate text and data address spaces, so the two requests are currently equivalent. (The argument *data* is ignored.)

PTRACE_PEEKUSER
Reads a word at offset *addr* in the child's USER area, which holds the registers and other information about the process (see *<sys/user.h>*). The word is returned as the result of the **ptrace**() call. Typically the offset must be word-aligned, though this might vary by architecture. See NOTES. (*data* is ignored.)

PTRACE_POKETEXT, **PTRACE_POKEDATA**
Copies the word *data* to location *addr* in the child's memory. As above, the two requests are currently equivalent.

PTRACE_POKEUSER
Copies the word *data* to offset *addr* in the child's USER area. As above, the offset must typically be word-aligned. In order to maintain the integrity of the kernel, some modifications to the USER area are disallowed.

PTRACE_GETREGS, **PTRACE_GETFPREGS**
Copies the child's general purpose or floating-point registers, respectively, to location *data* in the parent. See *<sys/user.h>* for information on the format of this data. (*addr* is ignored.)

PTRACE_GETSIGINFO (since Linux 2.3.99-pre6)
Retrieve information about the signal that caused the stop. Copies a *siginfo_t* structure (see **sigaction**(2)) from the child to location *data* in the parent. (*addr* is ignored.)

PTRACE_SETREGS, **PTRACE_SETFPREGS**
Copies the child's general purpose or floating-point registers, respectively, from location *data* in the parent. As for **PTRACE_POKEUSER**, some general purpose register modifications may be disallowed. (*addr* is ignored.)

PTRACE_SETSIGINFO (since Linux 2.3.99-pre6)
Set signal information. Copies a *siginfo_t* structure from location *data* in the parent to the child. This will only affect signals that would normally be delivered to the child and were caught by the tracer. It may be difficult to tell these normal signals from synthetic signals generated by **ptrace**() itself. (*addr* is ignored.)

PTRACE_SETOPTIONS (since Linux 2.4.6; see BUGS for caveats)
Sets ptrace options from *data* in the parent. (*addr* is ignored.) *data* is interpreted as a bit mask of options, which are specified by the following flags:

PTRACE_O_TRACESYSGOOD (since Linux 2.4.6)
When delivering syscall traps, set bit 7 in the signal number (i.e., deliver *(SIGTRAP | 0x80)* This makes it easy for the tracer to tell the difference between normal traps and those caused by a syscall. (**PTRACE_O_TRACESYSGOOD** may not work on all architectures.)

PTRACE_O_TRACEFORK (since Linux 2.5.46)
Stop the child at the next **fork**(2) call with *SIGTRAP | PTRACE_EVENT_FORK << 8* and automatically start tracing the newly forked process, which will start with a **SIGSTOP**. The PID for the new process can be retrieved with **PTRACE_GETEVENTMSG**.

PTRACE_O_TRACEVFORK (since Linux 2.5.46)
Stop the child at the next **vfork**(2) call with *SIGTRAP | PTRACE_EVENT_VFORK << 8* and automatically start tracing the newly vforked process, which will start with a **SIGSTOP**. The PID for the new process can be retrieved with **PTRACE_GETEVENTMSG**.

PTRACE_O_TRACECLONE (since Linux 2.5.46)
Stop the child at the next **clone**(2) call with *SIGTRAP | PTRACE_EVENT_CLONE << 8* and automatically start tracing the newly cloned process, which will start with a **SIGSTOP**. The PID for the new process can be retrieved with **PTRACE_GETEVENTMSG**. This option may not catch **clone**(2) calls in all cases. If the child calls **clone**(2) with the **CLONE_VFORK** flag, **PTRACE_EVENT_VFORK** will be delivered instead if **PTRACE_O_TRACEVFORK** is set; otherwise if the child calls **clone**(2) with the exit signal set to **SIGCHLD**, **PTRACE_EVENT_FORK** will be delivered if **PTRACE_O_TRACEFORK** is set.

PTRACE_O_TRACEEXEC (since Linux 2.5.46)
Stop the child at the next **execve**(2) call with *SIGTRAP | PTRACE_EVENT_EXEC << 8*.

PTRACE_O_TRACEVFORKDONE (since Linux 2.5.60)
Stop the child at the completion of the next **vfork**(2) call with *SIGTRAP | PTRACE_EVENT_VFORK_DONE << 8*.

PTRACE_O_TRACEEXIT (since Linux 2.5.60)
Stop the child at exit with *SIGTRAP | PTRACE_EVENT_EXIT << 8*. The child's exit status can be retrieved with **PTRACE_GETEVENTMSG**. This stop will be done early during process exit when registers are still available, allowing the tracer to see where the exit occurred, whereas the normal exit notification is done after the process is finished exiting. Even though context is available, the tracer cannot prevent the exit from

happening at this point.

PTRACE_GETEVENTMSG (since Linux 2.5.46)
Retrieve a message (as an *unsigned long*) about the ptrace event that just happened, placing it in the location *data* in the parent. For **PTRACE_EVENT_EXIT** this is the child's exit status. For **PTRACE_EVENT_FORK**, **PTRACE_EVENT_VFORK** and **PTRACE_EVENT_CLONE** this is the PID of the new process. Since Linux 2.6.18, the PID of the new process is also available for **PTRACE_EVENT_VFORK_DONE**. (*addr* is ignored.)

PTRACE_CONT
Restarts the stopped child process. If *data* is non-zero and not **SIGSTOP**, it is interpreted as a signal to be delivered to the child; otherwise, no signal is delivered. Thus, for example, the parent can control whether a signal sent to the child is delivered or not. (*addr* is ignored.)

PTRACE_SYSCALL, **PTRACE_SINGLESTEP**
Restarts the stopped child as for **PTRACE_CONT**, but arranges for the child to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively. (The child will also, as usual, be stopped upon receipt of a signal.) From the parent's perspective, the child will appear to have been stopped by receipt of a **SIGTRAP**. So, for **PTRACE_SYSCALL**, for example, the idea is to inspect the arguments to the system call at the first stop, then do another **PTRACE_SYSCALL** and inspect the return value of the system call at the second stop. The *data* argument is treated as for **PTRACE_CONT**. (*addr* is ignored.)

PTRACE_SYSEMU, **PTRACE_SYSEMU_SINGLESTEP** (since Linux 2.6.14)
For **PTRACE_SYSEMU**, continue and stop on entry to the next syscall, which will not be executed. For **PTRACE_SYSEMU_SINGLESTEP**, do the same but also singlestep if not a syscall. This call is used by programs like User Mode Linux that want to emulate all the child's system calls. The *data* argument is treated as for **PTRACE_CONT**. (*addr* is ignored; not supported on all architectures.)

PTRACE_KILL
Sends the child a **SIGKILL** to terminate it. (*addr* and *data* are ignored.)

PTRACE_ATTACH
Attaches to the process specified in *pid*, making it a traced "child" of the calling process; the behavior of the child is as if it had done a **PTRACE_TRACEME**. The calling process actually becomes the parent of the child process for most purposes (e.g., it will receive notification of child events and appears in **ps**(1) output as the child's parent), but a **getppid**(2) by the child will still return the PID of the original parent. The child is sent a **SIGSTOP**, but will not necessarily have stopped by the completion of this call; use **wait**(2) to wait for the child to stop. (*addr* and *data* are ignored.)

PTRACE_DETACH
Restarts the stopped child as for **PTRACE_CONT**, but first detaches from the process, undoing the reparenting effect of **PTRACE_ATTACH**, and the effects of **PTRACE_TRACEME**. Although perhaps not intended, under Linux a traced child can be detached in this way regardless of which method was used to initiate tracing. (*addr* is ignored.)

RETURN VALUE

On success, **PTRACE_PEEK*** requests return the requested data, while other requests return zero. On error, all requests return −1, and *errno* is set appropriately. Since the value returned by a successful **PTRACE_PEEK*** request may be −1, the caller must check *errno* after such requests to determine whether or not an error occurred.

ERRORS

EBUSY
(i386 only) There was an error with allocating or freeing a debug register.

EFAULT
There was an attempt to read from or write to an invalid area in the parent's or child's memory, probably because the area wasn't mapped or accessible. Unfortunately, under Linux, different variations of this fault will return **EIO** or **EFAULT** more or less arbitrarily.

EINVAL
An attempt was made to set an invalid option.

EIO *request* is invalid, or an attempt was made to read from or write to an invalid area in the parent's or child's memory, or there was a word-alignment violation, or an invalid signal was specified during a restart request.

EPERM
The specified process cannot be traced. This could be because the parent has insufficient privileges (the required capability is **CAP_SYS_PTRACE**); non-root processes cannot trace processes that they cannot send signals to or those running set-user-ID/set-group-ID programs, for obvious reasons. Alternatively, the process may already be being traced, or be **init**(8) (PID 1).

ESRCH
The specified process does not exist, or is not currently being traced by the caller, or is not stopped (for requests that require that).

CONFORMING TO

SVr4, 4.3BSD.

NOTES

Although arguments to **ptrace**() are interpreted according to the prototype given, glibc currently declares **ptrace**() as a variadic function with only the *request* argument fixed. This means that unneeded trailing arguments may be omitted, though doing so makes use of undocumented **gcc**(1) behavior.

init(8), the process with PID 1, may not be traced.

The layout of the contents of memory and the USER area are quite OS- and architecture-specific. The offset supplied, and the data returned, might not entirely match with the definition of *struct user*.

The size of a "word" is determined by the OS variant (e.g., for 32-bit Linux it is 32 bits, etc.).

Tracing causes a few subtle differences in the semantics of traced processes. For example, if a process is attached to with **PTRACE_ATTACH**, its original parent can no longer receive notification via **wait**(2) when it stops, and there is no way for the new parent to effectively simulate this notification.

When the parent receives an event with **PTRACE_EVENT_*** set, the child is not in the normal signal delivery path. This means the parent cannot do **ptrace**(PTRACE_CONT) with a signal or **ptrace**(PTRACE_KILL). **kill**(2) with a **SIGKILL** signal can be used instead to kill the child process after receiving one of these messages.

This page documents the way the **ptrace**() call works currently in Linux. Its behavior differs noticeably on other flavors of Unix. In any case, use of **ptrace**() is highly OS- and architecture-specific.

The SunOS man page describes **ptrace**() as "unique and arcane", which it is. The proc-based debugging interface present in Solaris 2 implements a superset of **ptrace**() functionality in a more powerful and uniform way.

BUGS

On hosts with 2.6 kernel headers, **PTRACE_SETOPTIONS** is declared with a different value than the one for 2.4. This leads to applications compiled with such headers failing when run on 2.4 kernels. This can be worked around by redefining **PTRACE_SETOPTIONS** to **PTRACE_OLDSETOPTIONS**, if that is defined.

SEE ALSO

gdb(1), **strace**(1), **execve**(2), **fork**(2), **signal**(2), **wait**(2), **exec**(3), **capabilities**(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/.