NAME

ptrace - process tracing and debugging

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <sys/types.h>
#include <sys/ptrace.h>

int
ptrace(*int request*, *pid_t pid*, *caddr_t addr*, *int data*);

DESCRIPTION

The **ptrace**() system call provides tracing and debugging facilities. It allows one process (the *tracing* process) to control another (the *traced* process). The tracing process must first attach to the traced process, and then issue a series of **ptrace**() system calls to control the execution of the process, as well as access process memory and register state. For the duration of the tracing session, the traced process will be "re-parented", with its parent process ID (and resulting behavior) changed to the tracing process. It is permissible for a tracing process to attach to more than one other process at a time. When the tracing process has completed its work, it must detach the traced process; if a tracing process exits without first detaching all processes it has attached, those processes will be killed.

Most of the time, the traced process runs normally, but when it receives a signal (see sigaction(2)), it stops. The tracing process is expected to notice this via wait(2) or the delivery of a SIGCHLD signal, examine the state of the stopped process, and cause it to terminate or continue as appropriate. The signal may be a normal process signal, generated as a result of traced process behavior, or use of the kill(2) system call; alternatively, it may be generated by the tracing facility as a result of attaching, stepping by the tracing process, or an event in the traced process. The tracing process may choose to intercept the signal, using it to observe process behavior (such as SIGTRAP), or forward the signal to the process if appropriate. The **ptrace**() system call is the mechanism by which all this happens.

A traced process may report additional signal stops corresponding to events in the traced process. These additional signal stops are reported as SIGTRAP or SIGSTOP signals. The tracing process can use the PT_LWPINFO request to determine which events are associated with a SIGTRAP or SIGSTOP signal. Note that multiple events may be associated with a single signal. For example, events indicated by the PL_FLAG_BORN, PL_FLAG_FORKED, and PL_FLAG_EXEC flags are also reported as a system call exit event (PL_FLAG_SCX). The signal stop for a new child process enabled via PTRACE_FORK will report a SIGSTOP signal. All other additional signal stops use SIGTRAP.

DETACH AND TERMINATION

Normally, exiting tracing process should wait for all pending debugging events and then detach from all alive traced processes before exiting using PT_DETACH request. If tracing process exits without detaching, for instance due to abnormal termination, the destiny of the traced children processes is determined by the kern.kill_on_debugger_exit sysctl control.

If the control is set to the default value 1, such traced processes are terminated. If set to zero, kernel implicitly detaches traced processes. Traced processes are un-stopped if needed, and then continue the execution without tracing. Kernel drops any SIGTRAP signals queued to the traced children, which could be either generated by not yet consumed debug events, or sent by other means, the later should not be done anyway.

SELECTING THE TARGET

The *pid* argument of the call specifies the target on which to perform the requested operation. For operations affecting the global process state, the process ID is typically passed there. Similarly, for operations affecting only a thread, the thread ID needs to be passed.

Still, for global operations, the ID of any thread can be used as the target, and system will perform the request on the process owning that thread. If a thread operation got the process ID as *pid*, the system randomly selects a thread from among the threads owned by the process. For single-threaded processes there is no difference between specifying process or thread ID as the target.

DISABLING PTRACE

The **ptrace** subsystem provides rich facilities to manipulate other processes state. Sometimes it may be desirable to disallow it either completely, or limit its scope. The following controls are provided for this:

security.bsd.allow_ptrace	Setting this sysctl to zero makes **ptrace** return ENOSYS always as if the syscall is not implemented by the kernel.
security.bsd.unprivileged_proc_debug	Setting this sysctl to zero disallows the use of **ptrace**() by unprivileged processes.
security.bsd.see_other_uids	Setting this sysctl to zero prevents **ptrace**() requests from targeting processes with a real user identifier different from the caller’s. These requests will fail with error ESRCH.
security.bsd.see_other_gids	Setting this sysctl to zero disallows **ptrace**() requests from processes that have no groups in common with the target process, considering their sets of real and supplementary groups.

These requests will fail with error ESRCH.

security.bsd.see_jail_proc — Setting this sysctl to zero disallows **ptrace**() requests from processes belonging to a different jail than that of the target process, even if the requesting process' jail is an ancestor of the target process'. These requests will fail with error ESRCH.

securelevel and init — The init(1) process can only be traced with **ptrace** if securelevel is zero.

procctl(2) PROC_TRACE_CTL — Process can deny attempts to trace itself with procctl(2) PROC_TRACE_CTL request. In this case requests return EPERM error.

TRACING EVENTS

Each traced process has a tracing event mask. An event in the traced process only reports a signal stop if the corresponding flag is set in the tracing event mask. The current set of tracing event flags include:

PTRACE_EXEC — Report a stop for a successful invocation of execve(2). This event is indicated by the PL_FLAG_EXEC flag in the *pl_flags* member of *struct ptrace_lwpinfo*.

PTRACE_SCE — Report a stop on each system call entry. This event is indicated by the PL_FLAG_SCE flag in the *pl_flags* member of *struct ptrace_lwpinfo*.

PTRACE_SCX — Report a stop on each system call exit. This event is indicated by the PL_FLAG_SCX flag in the *pl_flags* member of *struct ptrace_lwpinfo*.

PTRACE_SYSCALL — Report stops for both system call entry and exit.

PTRACE_FORK — This event flag controls tracing for new child processes of a traced process.

When this event flag is enabled, new child processes will enable tracing and stop before executing their first instruction. The new child process will include the PL_FLAG_CHILD flag in the *pl_flags* member of *struct ptrace_lwpinfo*. The traced process will report a stop that includes the PL_FLAG_FORKED flag. The process ID of the new child process will also be present in the *pl_child_pid* member of *struct ptrace_lwpinfo*. If the new child process was created via vfork(2), the traced process's stop will also include the PL_FLAG_VFORKED flag. Note that new child processes will

be attached with the default tracing event mask; they do not inherit the event mask of the traced process.

When this event flag is not enabled, new child processes will execute without tracing enabled.

PTRACE_LWP This event flag controls tracing of LWP (kernel thread) creation and destruction. When this event is enabled, new LWPs will stop and report an event with PL_FLAG_BORN set before executing their first instruction, and exiting LWPs will stop and report an event with PL_FLAG_EXITED set before completing their termination.

Note that new processes do not report an event for the creation of their initial thread, and exiting processes do not report an event for the termination of the last thread.

PTRACE_VFORK Report a stop event when a parent process resumes after a vfork(2).

When a thread in the traced process creates a new child process via vfork(2), the stop that reports PL_FLAG_FORKED and PL_FLAG_SCX occurs just after the child process is created, but before the thread waits for the child process to stop sharing process memory. If a debugger is not tracing the new child process, it must ensure that no breakpoints are enabled in the shared process memory before detaching from the new child process. This means that no breakpoints are enabled in the parent process either.

The PTRACE_VFORK flag enables a new stop that indicates when the new child process stops sharing the process memory of the parent process. A debugger can reinsert breakpoints in the parent process and resume it in response to this event. This event is indicated by setting the PL_FLAG_VFORK_DONE flag.

The default tracing event mask when attaching to a process via PT_ATTACH, PT_TRACE_ME, or PTRACE_FORK includes only PTRACE_EXEC events. All other event flags are disabled.

PTRACE REQUESTS

The *request* argument specifies what operation is being performed; the meaning of the rest of the arguments depends on the operation, but except for one special case noted below, all **ptrace**() calls are made by the tracing process, and the *pid* argument specifies the process ID of the traced process or a corresponding thread ID. The *request* argument can be:

PT_TRACE_ME
: This request is the only one used by the traced process; it declares that the process expects to be traced by its parent. All the other arguments are ignored. (If the parent process does not expect to trace the child, it will probably be rather confused by the results; once the traced process stops, it cannot be made to continue except via **ptrace**().) When a process has used this request and calls execve(2) or any of the routines built on it (such as execv(3)), it will stop before executing the first instruction of the new image. Also, any setuid or setgid bits on the executable being executed will be ignored. If the child was created by vfork(2) system call or rfork(2) call with the RFMEM flag specified, the debugging events are reported to the parent only after the execve(2) is executed.

PT_READ_I, PT_READ_D
: These requests read a single *int* of data from the traced process's address space. Traditionally, **ptrace**() has allowed for machines with distinct address spaces for instruction and data, which is why there are two requests: conceptually, PT_READ_I reads from the instruction space and PT_READ_D reads from the data space. In the current FreeBSD implementation, these two requests are completely identical. The *addr* argument specifies the address (in the traced process's virtual address space) at which the read is to be done. This address does not have to meet any alignment constraints. The value read is returned as the return value from **ptrace**().

PT_WRITE_I, PT_WRITE_D
: These requests parallel PT_READ_I and PT_READ_D, except that they write rather than read. The *data* argument supplies the value to be written.

PT_IO
: This request allows reading and writing arbitrary amounts of data in the traced process's address space. The *addr* argument specifies a pointer to a *struct ptrace_io_desc*, which is defined as follows:

```
struct ptrace_io_desc {
        int     piod_op;  /* I/O operation */
        void    *piod_offs;         /* child offset */
        void    *piod_addr;         /* parent offset */
        size_t  piod_len; /* request length */
};

/*
 * Operations in piod_op.
```

```
 */
#define PIOD_READ_D     1       /* Read from D space */
#define PIOD_WRITE_D    2       /* Write to D space */
#define PIOD_READ_I     3       /* Read from I space */
#define PIOD_WRITE_I    4       /* Write to I space */
```

The *data* argument is ignored. The actual number of bytes read or written is stored in *piod_len* upon return.

PT_CONTINUE
: The traced process continues execution. The *addr* argument is an address specifying the place where execution is to be resumed (a new value for the program counter), or (*caddr_t*)1 to indicate that execution is to pick up where it left off. The *data* argument provides a signal number to be delivered to the traced process as it resumes execution, or 0 if no signal is to be sent.

PT_STEP
: The traced process is single stepped one instruction. The *addr* argument should be passed (*caddr_t*)1. The *data* argument provides a signal number to be delivered to the traced process as it resumes execution, or 0 if no signal is to be sent.

PT_KILL
: The traced process terminates, as if PT_CONTINUE had been used with SIGKILL given as the signal to be delivered.

PT_ATTACH
: This request allows a process to gain control of an otherwise unrelated process and begin tracing it. It does not need any cooperation from the process to trace. In this case, *pid* specifies the process ID of the process to trace, and the other two arguments are ignored. This request requires that the target process must have the same real UID as the tracing process, and that it must not be executing a setuid or setgid executable. (If the tracing process is running as root, these restrictions do not apply.) The tracing process will see the newly-traced process stop and may then control it as if it had been traced all along.

PT_DETACH
: This request is like PT_CONTINUE, except that it does not allow specifying an alternate place to continue execution, and after it succeeds, the traced process is no longer traced and continues execution normally.

PT_GETREGS
: This request reads the traced process’s machine registers into the "*struct reg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_SETREGS — This request is the converse of PT_GETREGS; it loads the traced process's machine registers from the "*struct reg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_GETFPREGS — This request reads the traced process's floating-point registers into the "*struct fpreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_SETFPREGS — This request is the converse of PT_GETFPREGS; it loads the traced process's floating-point registers from the "*struct fpreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_GETDBREGS — This request reads the traced process's debug registers into the "*struct dbreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_SETDBREGS — This request is the converse of PT_GETDBREGS; it loads the traced process's debug registers from the "*struct dbreg*" (defined in <*machine/reg.h*>) pointed to by *addr*.

PT_GETREGSET — This request reads the registers from the traced process. The *data* argument specifies the register set to read, with the *addr* argument pointing at a *struct iovec* where the *iov_base* field points to a register set specific structure to hold the registers, and the *iov_len* field holds the length of the structure.

PT_SETREGSET — This request writes to the registers of the traced process. The *data* argument specifies the register set to write to, with the *addr* argument pointing at a *struct iovec* where the *iov_base* field points to a register set specific structure to hold the registers, and the *iov_len* field holds the length of the structure. If *iov_base* is NULL the kernel will return the expected length of the register set specific structure in the *iov_len* field and not change the target register set.

PT_LWPINFO — This request can be used to obtain information about the kernel thread, also known as light-weight process, that caused the traced process to stop. The *addr* argument specifies a pointer to a *struct ptrace_lwpinfo*, which is defined as follows:

```
struct ptrace_lwpinfo {
        lwpid_t pl_lwpid;
        int     pl_event;
```

```
        int         pl_flags;
        sigset_t    pl_sigmask;
        sigset_t    pl_siglist;
        siginfo_t   pl_siginfo;
        char        pl_tdname[MAXCOMLEN + 1];
        pid_t       pl_child_pid;
        u_int       pl_syscall_code;
        u_int       pl_syscall_narg;
};
```

The *data* argument is to be set to the size of the structure known to the caller. This allows the structure to grow without affecting older programs.

The fields in the *struct ptrace_lwpinfo* have the following meaning:

pl_lwpid
: LWP id of the thread

pl_event
: Event that caused the stop. Currently defined events are:

PL_EVENT_NONE	No reason given
PL_EVENT_SIGNAL	Thread stopped due to the pending signal

pl_flags
: Flags that specify additional details about observed stop. Currently defined flags are:

PL_FLAG_SCE
: The thread stopped due to system call entry, right after the kernel is entered. The debugger may examine syscall arguments that are stored in memory and registers according to the ABI of the current process, and modify them, if needed.

PL_FLAG_SCX
: The thread is stopped immediately before syscall is returning to the usermode. The debugger may examine system call return values in the ABI-defined registers and/or memory.

PL_FLAG_EXEC
: When PL_FLAG_SCX is set, this flag may be additionally specified to inform that the program being executed by debuggee process has been changed by

successful execution of a system call from the **execve**(*2*) family.

PL_FLAG_SI
: Indicates that *pl_siginfo* member of *struct ptrace_lwpinfo* contains valid information.

PL_FLAG_FORKED
: Indicates that the process is returning from a call to **fork**(*2*) that created a new child process. The process identifier of the new process is available in the *pl_child_pid* member of *struct ptrace_lwpinfo*.

PL_FLAG_CHILD
: The flag is set for first event reported from a new child which is automatically attached when PTRACE_FORK is enabled.

PL_FLAG_BORN
: This flag is set for the first event reported from a new LWP when PTRACE_LWP is enabled. It is reported along with PL_FLAG_SCX.

PL_FLAG_EXITED
: This flag is set for the last event reported by an exiting LWP when PTRACE_LWP is enabled. Note that this event is not reported when the last LWP in a process exits. The termination of the last thread is reported via a normal process exit event.

PL_FLAG_VFORKED
: Indicates that the thread is returning from a call to vfork(2) that created a new child process. This flag is set in addition to PL_FLAG_FORKED.

PL_FLAG_VFORK_DONE
: Indicates that the thread has resumed after a child process created via vfork(2) has stopped sharing its address space with the traced process.

pl_sigmask
: The current signal mask of the LWP

pl_siglist
: The current pending set of signals for the LWP. Note that signals that are delivered to the process would not appear on an LWP siglist until the thread is selected for delivery.

pl_siginfo
: The siginfo that accompanies the signal pending. Only valid for

PL_EVENT_SIGNAL stop when PL_FLAG_SI is set in *pl_flags*.

pl_tdname
: The name of the thread.

pl_child_pid
: The process identifier of the new child process. Only valid for a PL_EVENT_SIGNAL stop when PL_FLAG_FORKED is set in *pl_flags*.

pl_syscall_code
: The ABI-specific identifier of the current system call. Note that for indirect system calls this field reports the indirected system call. Only valid when PL_FLAG_SCE or PL_FLAG_SCX is set in *pl_flags*.

pl_syscall_narg
: The number of arguments passed to the current system call not counting the system call identifier. Note that for indirect system calls this field reports the arguments passed to the indirected system call. Only valid when PL_FLAG_SCE or PL_FLAG_SCX is set in *pl_flags*.

PT_GETNUMLWPS
: This request returns the number of kernel threads associated with the traced process.

PT_GETLWPLIST
: This request can be used to get the current thread list. A pointer to an array of type *lwpid_t* should be passed in *addr*, with the array size specified by *data*. The return value from **ptrace**() is the count of array entries filled in.

PT_SETSTEP
: This request will turn on single stepping of the specified process. Stepping is automatically disabled when a single step trap is caught.

PT_CLEARSTEP
: This request will turn off single stepping of the specified process.

PT_SUSPEND
: This request will suspend the specified thread.

PT_RESUME
: This request will resume the specified thread.

PT_TO_SCE
: This request will set the PTRACE_SCE event flag to trace all future system call entries and continue the process. The *addr* and *data* arguments are used the same as for PT_CONTINUE.

PT_TO_SCX
: This request will set the PTRACE_SCX event flag to trace all future system call exits and continue the process. The *addr* and *data* arguments are used the same as for PT_CONTINUE.

PT_SYSCALL
: This request will set the PTRACE_SYSCALL event flag to trace all future system call entries and exits and continue the process. The *addr* and *data* arguments are used the same as for PT_CONTINUE.

PT_GET_SC_ARGS
: For the thread which is stopped in either PL_FLAG_SCE or PL_FLAG_SCX state, that is, on entry or exit to a syscall, this request fetches the syscall arguments.

  The arguments are copied out into the buffer pointed to by the *addr* pointer, sequentially. Each syscall argument is stored as the machine word. Kernel copies out as many arguments as the syscall accepts, see the *pl_syscall_narg* member of the *struct ptrace_lwpinfo*, but not more than the *data* bytes in total are copied.

PT_GET_SC_RET
: Fetch the system call return values on exit from a syscall. This request is only valid for threads stopped in a syscall exit (the PL_FLAG_SCX state). The *addr* argument specifies a pointer to a *struct ptrace_sc_ret*, which is defined as follows:

```
struct ptrace_sc_ret {
        register_t sr_retval[2];
        int        sr_error;
};
```

  The *data* argument is set to the size of the structure.

  If the system call completed successfully, *sr_error* is set to zero and the return values of the system call are saved in *sr_retval*. If the system call failed to execute, *sr_error* field is set to a positive errno(2) value. If the system call completed in an unusual fashion, *sr_error* is set to a negative value:

  ERESTART System call will be restarted.

  EJUSTRETURN
  : System call completed sucessfully but did not set a

return value (for example, setcontext(2) and sigreturn(2)).

PT_FOLLOW_FORK
: This request controls tracing for new child processes of a traced process. If *data* is non-zero, PTRACE_FORK is set in the traced process's event tracing mask. If *data* is zero, PTRACE_FORK is cleared from the traced process's event tracing mask.

PT_LWP_EVENTS
: This request controls tracing of LWP creation and destruction. If *data* is non-zero, PTRACE_LWP is set in the traced process's event tracing mask. If *data* is zero, PTRACE_LWP is cleared from the traced process's event tracing mask.

PT_GET_EVENT_MASK
: This request reads the traced process's event tracing mask into the integer pointed to by *addr*. The size of the integer must be passed in *data*.

PT_SET_EVENT_MASK
: This request sets the traced process's event tracing mask from the integer pointed to by *addr*. The size of the integer must be passed in *data*.

PT_VM_TIMESTAMP
: This request returns the generation number or timestamp of the memory map of the traced process as the return value from **ptrace**(). This provides a low-cost way for the tracing process to determine if the VM map changed since the last time this request was made.

PT_VM_ENTRY
: This request is used to iterate over the entries of the VM map of the traced process. The *addr* argument specifies a pointer to a *struct ptrace_vm_entry*, which is defined as follows:

```
struct ptrace_vm_entry {
        int             pve_entry;
        int             pve_timestamp;
        u_long          pve_start;
        u_long          pve_end;
        u_long          pve_offset;
        u_int           pve_prot;
        u_int           pve_pathlen;
        long            pve_fileid;
        uint32_t        pve_fsid;
```

```
        char            *pve_path;
};
```

The first entry is returned by setting *pve_entry* to zero. Subsequent entries are returned by leaving *pve_entry* unmodified from the value returned by previous requests. The *pve_timestamp* field can be used to detect changes to the VM map while iterating over the entries. The tracing process can then take appropriate action, such as restarting. By setting *pve_pathlen* to a non-zero value on entry, the pathname of the backing object is returned in the buffer pointed to by *pve_path*, provided the entry is backed by a vnode. The *pve_pathlen* field is updated with the actual length of the pathname (including the terminating null character). The *pve_offset* field is the offset within the backing object at which the range starts. The range is located in the VM space at *pve_start* and extends up to *pve_end* (inclusive).

The *data* argument is ignored.

PT_COREDUMP This request creates a coredump for the stopped program. The *addr* argument specifies a pointer to a *struct ptrace_coredump*, which is defined as follows:

```
struct ptrace_coredump {
        int             pc_fd;
        uint32_t        pc_flags;
        off_t           pc_limit;
};
```

The fields of the structure are:

pc_fd File descriptor to write the dump to. It must refer to a regular file, opened for writing.

pc_flags Flags. The following flags are defined:

PC_COMPRESS Request compression of the dump.

PC_ALL Include non-dumpable entries into the dump. The dumper ignores MAP_NOCORE flag of the process map entry, but device mappings are not dumped

even with PC_ALL set.

pc_limit Maximum size of the coredump. Specify zero for no limit.

The size of *struct ptrace_coredump* must be passed in *data*.

PT_SC_REMOTE Request to execute a syscall in the context of the traced process, in the specified thread. The *addr* argument must point to the *struct ptrace_sc_remote*, which describes the requested syscall and its arguments, and receives the result. The size of *struct ptrace_sc_remote* must be passed in *data*.

```
struct ptrace_sc_remote {
        struct ptrace_sc_ret pscr_ret;
        u_int       pscr_syscall;
        u_int       pscr_nargs;
        u_long      *pscr_args;
};
```

The pscr_syscall contains the syscall number to execute, the pscr_nargs is the number of supplied arguments, which are supplied in the pscr_args array. Result of the execution is returned in the pscr_ret member. Note that the request and its result do not affect the returned value from the currently executed syscall, if any.

PT_COREDUMP and PT_SC_REMOTE usage

The process must be stopped before dumping or initiating a remote system call. A single thread in the target process is temporarily unsuspended in the kernel to perform the action. If the **ptrace** call fails before a thread is unsuspended, there is no event to waitpid(2) for. If a thread was unsuspended, it will stop again before the **ptrace** call returns, and the process must be waited upon using waitpid(2) to consume the new stop event. Since it is hard to deduce whether a thread was unsuspended before an error occurred, it is recommended to unconditionally perform waitpid(2) with WNOHANG flag after PT_COREDUMP and PT_SC_REMOTE, and silently accept zero result from it.

For PT_SC_REMOTE, the selected thread must be stopped in the safe place, which is currently defined as a syscall exit, or a return from kernel to user mode (basically, a signal handler call place). Kernel returns EBUSY status if attempt is made to execute remote syscall at unsafe stop.

Note that neither kern.trap_enotcap sysctl setting, nor the corresponding procctl(2) flag PROC_TRAPCAP_CTL_ENABLE are obeyed during the execution of the syscall by PT_SC_REMOTE. In other words, SIGTRAP signal is not sent to a process executing in capability

mode, which violated a mode access restriction.

Note that due to the mode of execution for the remote syscall, in particular, the setting where only one thread is allowed to run, the syscall might block on resources owned by suspended threads. This might result in the target process deadlock. In this situation, the only way out is to kill the target.

ARM MACHINE-SPECIFIC REQUESTS

PT_GETVFPREGS Return the thread's VFP machine state in the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_SETVFPREGS Set the thread's VFP machine state from the buffer pointed to by *addr*.

The *data* argument is ignored.

x86 MACHINE-SPECIFIC REQUESTS

PT_GETXMMREGS Copy the XMM FPU state into the buffer pointed to by the argument *addr*. The buffer has the same layout as the 32-bit save buffer for the machine instruction FXSAVE.

This request is only valid for i386 programs, both on native 32-bit systems and on amd64 kernels. For 64-bit amd64 programs, the XMM state is reported as part of the FPU state returned by the PT_GETFPREGS request.

The *data* argument is ignored.

PT_SETXMMREGS Load the XMM FPU state for the thread from the buffer pointed to by the argument *addr*. The buffer has the same layout as the 32-bit load buffer for the machine instruction FXRSTOR.

As with PT_GETXMMREGS, this request is only valid for i386 programs.

The *data* argument is ignored.

PT_GETXSTATE_INFO Report which XSAVE FPU extensions are supported by the CPU and allowed in userspace programs. The *addr* argument must point to a variable of type *struct ptrace_xstate_info*, which contains the information on the request return. *struct ptrace_xstate_info* is defined as follows:

```
struct ptrace_xstate_info {
        uint64_t   xsave_mask;
        uint32_t   xsave_len;
};
```

The xsave_mask field is a bitmask of the currently enabled extensions. The meaning of the bits is defined in the Intel and AMD processor documentation. The xsave_len field reports the length of the XSAVE area for storing the hardware state for currently enabled extensions in the format defined by the x86 XSAVE machine instruction.

The *data* argument value must be equal to the size of the *struct ptrace_xstate_info*.

PT_GETXSTATE — Return the content of the XSAVE area for the thread. The *addr* argument points to the buffer where the content is copied, and the *data* argument specifies the size of the buffer. The kernel copies out as much content as allowed by the buffer size. The buffer layout is specified by the layout of the save area for the XSAVE machine instruction.

PT_SETXSTATE — Load the XSAVE state for the thread from the buffer specified by the *addr* pointer. The buffer size is passed in the *data* argument. The buffer must be at least as large as the *struct savefpu* (defined in *x86/fpu.h*) to allow the complete x87 FPU and XMM state load. It must not be larger than the XSAVE state length, as reported by the xsave_len field from the *struct ptrace_xstate_info* of the PT_GETXSTATE_INFO request. Layout of the buffer is identical to the layout of the load area for the XRSTOR machine instruction.

PT_GETFSBASE — Return the value of the base used when doing segmented memory addressing using the %fs segment register. The *addr* argument points to an *unsigned long* variable where the base value is stored.

The *data* argument is ignored.

PT_GETGSBASE — Like the PT_GETFSBASE request, but returns the base for the %gs segment register.

PT_SETFSBASE — Set the base for the %fs segment register to the value pointed to by the *addr* argument. *addr* must point to the *unsigned long* variable containing the new base.

The *data* argument is ignored.

PT_SETGSBASE Like the PT_SETFSBASE request, but sets the base for the %gs segment register.

PowerPC MACHINE-SPECIFIC REQUESTS

PT_GETVRREGS Return the thread's ALTIVEC machine state in the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_SETVRREGS Set the thread's ALTIVEC machine state from the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_GETVSRREGS Return doubleword 1 of the thread's VSX registers VSR0-VSR31 in the buffer pointed to by *addr*.

The *data* argument is ignored.

PT_SETVSRREGS Set doubleword 1 of the thread's VSX registers VSR0-VSR31 from the buffer pointed to by *addr*.

The *data* argument is ignored.

Additionally, other machine-specific requests can exist.

RETURN VALUES

Most requests return 0 on success and -1 on error. Some requests can cause **ptrace**() to return -1 as a non-error value, among them are PT_READ_I and PT_READ_D, which return the value read from the process memory on success. To disambiguate, *errno* can be set to 0 before the call and checked afterwards.

The current **ptrace**() implementation always sets *errno* to 0 before calling into the kernel, both for historic reasons and for consistency with other operating systems. It is recommended to assign zero to *errno* explicitly for forward compatibility.

ERRORS

The **ptrace**() system call may fail if:

[ESRCH]

- No process having the specified process ID exists.

[EINVAL]

- A process attempted to use PT_ATTACH on itself.
- The *request* argument was not one of the legal requests.
- The signal number (in *data*) to PT_CONTINUE was neither 0 nor a legal signal number.
- PT_GETREGS, PT_SETREGS, PT_GETFPREGS, PT_SETFPREGS, PT_GETDBREGS, or PT_SETDBREGS was attempted on a process with no valid register set. (This is normally true only of system processes.)
- PT_VM_ENTRY was given an invalid value for *pve_entry*. This can also be caused by changes to the VM map of the process.
- The size (in *data*) provided to PT_LWPINFO was less than or equal to zero, or larger than the *ptrace_lwpinfo* structure known to the kernel.
- The size (in *data*) provided to the x86-specific PT_GETXSTATE_INFO request was not equal to the size of the *struct ptrace_xstate_info*.
- The size (in *data*) provided to the x86-specific PT_SETXSTATE request was less than the size of the x87 plus the XMM save area.
- The size (in *data*) provided to the x86-specific PT_SETXSTATE request was larger than returned in the xsave_len member of the *struct ptrace_xstate_info* from the PT_GETXSTATE_INFO request.
- The base value, provided to the amd64-specific requests PT_SETFSBASE or PT_SETGSBASE, pointed outside of the valid user address space. This error will not occur in 32-bit programs.

[EBUSY]

- PT_ATTACH was attempted on a process that was already being traced.
- A request attempted to manipulate a process that was being traced by some process other than the one making the request.
- A request (other than PT_ATTACH) specified a process that was not stopped.

[EPERM]

- A request (other than PT_ATTACH) attempted to manipulate a process that was not being traced at all.
- An attempt was made to use PT_ATTACH on a process in violation of the requirements listed under PT_ATTACH above.

[ENOENT]

- PT_VM_ENTRY previously returned the last entry of the memory map. No more entries exist.

[ENOMEM]

- A PT_READ_I, PT_READ_D, PT_WRITE_I, or PT_WRITE_D request attempted to access an invalid address, or a memory allocation failure occurred when accessing process memory.

[ENAMETOOLONG]

- PT_VM_ENTRY cannot return the pathname of the backing object because the buffer is not big enough. *pve_pathlen* holds the minimum buffer size required on return.

SEE ALSO

execve(2), sigaction(2), wait(2), execv(3), i386_clr_watch(3), i386_set_watch(3)

HISTORY

The **ptrace**() function appeared in Version 6 AT&T UNIX.