ptrace

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 shall enable a process to observe and control the execution of another process, as well as examine and change certain attributes of that process.

This function operates via requests, which act on the traced process using the other parameters in ways unique to each request type. The tracing process must initiate tracing, either via the `PTRACE_TRACEME` or `PTRACE_ATTACH` requests, before other requests may be performed. Except for `PTRACE_TRACEME` and `PTRACE_KILL`, all requests must be performed on a traced process that has been stopped.

All signals, except one, delivered to the traced process cause it to stop, irrespective of its registered signal handling, and cause an event to be delivered to the tracing process which can be detected using the wait(2) system call. The exception is the `SIGKILL` signal, which is delivered immediately and performs its usual specified behavior.

The following requests are defined:

`PTRACE_TRACEME`

> This request initates a trace from the perspective of the traced process, indicating that the parent of the current process shall be the tracing process. When this is called, a subsequent call to execve(2) shall cause the tracing process to receive a SIGTRAP signal, and shall stop the current process. This is the only request a traced process may perform, and a tracing process may not perform this request. The other parameters are ignored.

`PTRACE_ATTACH`

> This request initates a trace from the perspective of the tracing process on the process specified by *`pid`*. After this call succeeds, the traced process will appear to be a child of the tracing process, although the original parent will still be returned to the traced process via getppid(2). The traced process will receive a `SIGSTOP` signal; the tracing process should use wait(2) to ensure that the traced process has stopped. A tracing process is only guaranteed to be able to trace its child processes; the tracing of other processes may not be allowed by the system, and the process with process ID 1 may not be traced under any circumstances. The *`addr`* and *`data`* parameters are ignored.

`PTRACE_CONT`

This request restarts a traced process, given in *pid*, which has been stopped. The *data* parameter may point to a signal ID to deliver to the traced process; if it is zero or SIGSTOP, no signal is delivered to the child. The *addr* is ignored.

`PTRACE_DETACH`

This request performs the same function, in the same way, as `PTRACE_CONT`, except that the tracing relationship between the tracing and traced processes is also undone. If the trace was initiated using `PTRACE_ATTACH`, the original parent-child relationships that existed beforehand are restored.

`PTRACE_KILL`

This request causes a `SIGKILL` signal to be sent to the traced process specified in *pid*. The *addr* and *data* parameters are ignored.

`PTRACE_PEEKTEXT`

This request reads a word at the location *addr* of the traced process *pid*, and returns it to the caller. The *data* parameter is ignored.

`PTRACE_PEEKDATA`

This request performs identically to the `PTRACE_PEEKTEXT` request.

`PTRACE_PEEKUSER`

This request reads a word at offset *addr* in the USER area of the traced process *pid*. The offset must be word-aligned. The *data* parameter is ignored.

`PTRACE_POKETEXT`

This request writes the word pointed at by *data* to the location *addr* of the traced process *pid*.

`PTRACE_POKEDATA`

This request performs identically to the `PTRACE_POKETEXT` request.

`PTRACE_POKEUSER`

This request writes the word pointed at by *data* to offset *addr* in the USER area of the traced process *pid*. The offset must be word-aligned. Implementations may choose to disallow some modifications to the USER area.

`PTRACE_GETREGS`

This request copies the general purpose registers from the traced process *pid* to the tracing process at location *data*. This parameter may not be available on all architectures. The *addr* parameter is ignored.

`PTRACE_GETFPREGS`

This request copies the floating point registers from the traced process *pid* to the tracing process at location *data*. This parameter may not be available on all architectures. The *addr* parameter is ignored.

`PTRACE_SETREGS`

This request writes the general purpose registers to the traced process *pid* from the tracing process at location *data*. This parameter may not be available on all architectures. Implementations may choose to disallow some register modifications. The *addr* parameter is ignored.

`PTRACE_SETFPREGS`

This request writes the floating point registers to the traced process *`pid`* from the tracing process at location *`data`*. This parameter may not be available on all architectures. Implementations may choose to disallow some register modifications. The *`addr`* parameter is ignored.

`PTRACE_GETSIGINFO`

This request writes information about the signal which caused the traced process *`pid`* to stop to the tracing process at location *`data`*, as a siginfo_t. The *`addr`* parameter is ignored.

`PTRACE_SETSIGINFO`

This request writes signal information to the traced process *`pid`* from a siginfo_t structure pointed at by *`data`*, such that it will be used as the signal information by the traced process when it is resumed. The *`addr`* parameter is ignored.

`PTRACE_GETEVENTMSG`

This request stores information about the most recent ptrace event for the traced process *`pid`* in the unsigned long pointed at by *`data`*. For `PTRACE_EVENT_EXIT`, this is the exit status of the traced process. For `PTRACE_EVENT_FORK`, `PTRACE_EVENT_VFORK`, or `PTRACE_EVENT_CLONE`, this is the PID of the newly created process. The *`addr`* parameter is ignored.

`PTRACE_SYSCALL`

This request performs the same function, in the same way, as `PTRACE_CONT`, but with the additional step of causing the traced process to stop at the next entry to or exit from a system call. The usual events that would also cause the traced process to stop continue to do so.

`PTRACE_SINGLESTEP`

This request performs the same function, in the same way, as `PTRACE_CONT`, but with the additional step of causing the traced process to stop after execution of a single instruction. The usual events that would also cause the traced process to stop continue to do so.

`PTRACE_SYSEMU`

This request performs the same function, in the same way, as `PTRACE_CONT`, but with the additional step of causing the traced process to stop on entry to the next syscall, which will then not be executed.

`PTRACE_SYSEMU_SINGLESTEP`

This request performs the same function, in the same way, as `PTRACE_CONT`, but with the additional step of causing the traced process to stop on entry to the next syscall, which will then not be executed. If the next instruction is not itself a syscall, the traced process will stop after a single instruction is executed.

`PTRACE_SETOPTIONS`

This request sets `ptrace()` options for the traced process *`pid`* from the location pointed to by *`data`*. The *`addr`* is ignored. This location is interpreted as a bitmask of options, as defined by the following flags:

`PTRACE_O_TRACESYSGOOD`

This option, when set, causes syscall traps to set bit 7 in the signal number.

`PTRACE_O_TRACEFORK`

This option, when set, causes the traced process to stop when it calls fork(2). The original traced process will stop with SIGTRAP | PTRACE_EVENT_FORK << 8, and the new process will be stopped with SIGSTOP. The new process will also be traced by the tracing process, as if the tracing process had sent the PTRACE_ATTACH request for that process. The PID of the new process may be retrieved with the PTRACE_GETEVENTMSG request.

`PTRACE_O_TRACEVFORK`

This option, when set, causes the traced process to stop when it calls vfork(2). The original traced process will stop with SIGTRAP | PTRACE_EVENT_VFORK << 8, and the new process will be stopped with SIGSTOP. The new process will also be traced by the tracing process, as if the tracing process had sent the PTRACE_ATTACH request for that process. The PID of the new process may be retrieved with the PTRACE_GETEVENTMSG request.

`PTRACE_O_TRACECLONE`

This option, when set, causes the traced process to stop when it calls clone(2). The original traced process will stop with SIGTRAP | PTRACE_EVENT_CLONE << 8, and the new process will be stopped with SIGSTOP. The new process will also be traced by the tracing process, as if the tracing process had sent the PTRACE_ATTACH request for that process. The PID of the new process may be retrieved with the PTRACE_GETEVENTMSG request. Under certain circumstances, clone(2) calls by the traced process will generate events and information consistent with the PTRACE_O_TRACEVFORK or PTRACE_O_TRACEFORK options above.

`PTRACE_O_TRACEEXEC`

This option, when set, causes the traced process to stop when it calls execve(2). The traced process will stop with SIGTRAP | PTRACE_EVENT_EXEC << 8.

`PTRACE_O_TRACEVFORKDONE`

This option, when set, causes the traced process to stop at the completion of its next vfork(2) call. The traced process will stop with SIGTRAP | PTRACE_EVENT_EXEC << 8.

`PTRACE_O_TRACEEXIT`

This option, when set, causes the traced process to stop upon exit. The traced process will stop with SIGTRAP | PTRACE_EVENT_EXIT << 8, and its exit status can be retrieved with the PTRACE_GETEVENTMSG request. The stop is guaranteed to be early in the process exit process, meaning that information such as register status at exit is preserved. Upon continuing, the traced process will immediately exit.

Return Value

On success, `ptrace()` shall return the requested data for `PTRACE_PEEK` requests, or zero for all other requests. On error, all requests return -1, with `errno` set to an appropriate value. Note that -1 may be a valid return value for `PTRACE_PEEK` requests; the application is responsible for distinguishing between an error condition and a valid return value in that case.

Errors

On error, `ptrace()` shall set `errno` to one of the regular error values below:

`EBUSY`

An error occurred while allocating or freeing a debug register.

`EFAULT`	The request attempted to read from or write to an invalid area in the memory space of the tracing or traced process.
`EIO`	The request was invalid, or it attempted to read from or write to an invalid area in the memory space of the tracing or traced process, or it violated a word-alignment boundary, or an invalid signal was given to continue the traced process.
`EINVAL`	An attempt was made to set an invalid option.
`EPERM`	The request to trace a process was denied by the system.
`ESRCH`	The process requested does not exist, is not being traced by the current process, or is not stopped.

<<< Previous Home Next >>>
psignal Up putwc_unlocked