NAME

/proc – process file system

DESCRIPTION

/proc is a file system that provides access to the image of each process in the system. The name of each entry in the **/proc** directory is a decimal number corresponding to the process-ID. The owner of each ‘‘file’’ is determined by the process’s user-ID.

Standard system call interfaces are used to access */proc* files: *open*(2), *close*(2), *read*(2), *write*(2), and *ioctl*(2). An open for reading and writing enables process control; a read-only open allows inspection but not control. As with ordinary files, more than one process can open the same **/proc** file at the same time. *Exclusive open* is provided to allow controlling processes to avoid collisions: an *open*(2) for writing that specifies O_EXCL fails if the file is already open for writing; if such an exclusive open succeeds, subsequent attempts to open the file for writing, with or without the O_EXCL flag, fail until the exclusively-opened file descriptor is closed. (Exception: a super-user *open*(2) that does not specify O_EXCL succeeds even if the file is exclusively opened.) There can be any number of read-only opens, even when an exclusive write open is in effect on the file.

Data may be transferred from or to any locations in the traced process’s address space by applying *lseek*(2) to position the file at the virtual address of interest followed by *read*(2) or *write*(2). The PIOCMAP operation can be applied to determine the accessible areas (mappings) of the address space. I/O transfers may span contiguous mappings. An I/O request extending into an unmapped area is truncated at the boundary. An I/O request beginning at an unmapped virtual address fails with EIO.

Information and control operations are provided through *ioctl*(2). These have the form:

```
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/fault.h>
#include <sys/syscall.h>
#include <sys/procfs.h>
void *p;
retval = ioctl(fildes, code, p);
```

The argument *p* is a generic pointer whose type depends on the specific *ioctl* code. Where not specifically mentioned below, its value should be zero. **<sys/procfs.h>** contains definitions of *ioctl* codes and data structures used by the operations.

Every active process contains at least one *light-weight process*, or *lwp*. Each lwp represents a flow of execution that is independently scheduled by the operating system. The PIOCOPENLWP operation can be applied to the process file descriptor to obtain a specific lwp file descriptor. I/O operations produce identical results whether applied to the process file descriptor or to an lwp file descriptor. All **/proc** *ioctl* operations can be applied to either type of file descriptor and, where not stated otherwise, produce identical results.

Process information and control operations involve the use of sets of flags. The set types **sigset_t**, **fltset_t**, and **sysset_t** correspond, respectively, to signal, fault, and system call enumerations defined in **<sys/signal.h>**, **<sys/fault.h>**, and **<sys/syscall.h>**. Each set type is large enough to hold flags for its own enumeration. Although they are of different sizes, they have a common structure and can be manipulated by these macros:

```
prfillset(&set);                /* turn on all flags in set */
premptyset(&set);               /* turn off all flags in set */
praddset(&set, flag);           /* turn on the specified flag */
prdelset(&set, flag);           /* turn off the specified flag */
r = prismember(&set, flag);     /* != 0 iff flag is turned on */
```

One of **prfillset**() or **premptyset**() must be used to initialize **set** before it is used in any other operation. **flag** must be a member of the enumeration corresponding to **set**.

IOCTL CODES

The allowable *ioctl* codes follow. Certain of these can be used only if the process or lwp file descriptor is open for writing; these include all operations that affect process control. Those requiring write access are marked with an asterisk (*). Except where noted, an **ioctl** to a process or lwp that has terminated elicits the error ENOENT.

PIOCSTATUS

PIOCSTATUS returns status information for the process and one of its lwps; *p* is a pointer to a **prstatus** structure containing at least the following fields:

```
typedef struct prstatus {
        long                pr_flags;           /* Flags */
        short               pr_why;             /* Reason for stop (if stopped) */
        short               pr_what;            /* More detailed reason */
        lwpid_t             pr_who;             /* Specific lwp identifier */
        u_short             pr_nlwp;            /* Number of lwps in the process */
        short               pr_cursig;          /* Current signal */
        sigset_t            pr_sigpend;         /* Set of process pending signals */
        sigset_t            pr_lwppend;         /* Set of lwp pending signals */
        sigset_t            pr_sighold;         /* Set of lwp held signals */
        struct siginfo      pr_info;            /* Info associated with signal or fault */
        struct sigaltstack  pr_altstack;        /* Alternate signal stack info */
        struct sigaction    pr_action;          /* Signal action for current signal */
        struct ucontext     *pr_oldcontext;     /* Previous context on lwp stack */
        caddr_t             pr_brkbase;         /* Address of the process heap */
        u_long              pr_brksize;         /* Size of the process heap, in bytes */
        caddr_t             pr_stkbase;         /* Address of the lwp stack */
        u_long              pr_stksize;         /* Size of the lwp stack, in bytes */
        short               pr_syscall;         /* System call number (if in syscall) */
        short               pr_nsysarg;         /* Number of arguments to this syscall */
        long                pr_sysarg[PRSYSARGS];   /* Arguments to this syscall */
        pid_t               pr_pid;             /* Process id */
        pid_t               pr_ppid;            /* Parent process id */
        pid_t               pr_pgrp;            /* Process group id */
        pid_t               pr_sid;             /* Session id */
        timestruc_t         pr_utime;           /* Process user cpu time */
        timestruc_t         pr_stime;           /* Process system cpu time */
        timestruc_t         pr_cutime;          /* Sum of children's user times */
        timestruc_t         pr_cstime;          /* Sum of children's system times */
```

```
        char            pr_clname[8];   /* Scheduling class name */
        long            pr_instr;       /* Current instruction */
        prgregset_t     pr_reg;         /* General registers */
} prstatus_t;
```

pr_flags is a bit-mask holding these flags:

PR_STOPPED	lwp is stopped
PR_ISTOP	lwp is stopped on an event of interest (see PIOCSTOP)
PR_DSTOP	lwp has a stop directive in effect (see PIOCSTOP)
PR_STEP	lwp has a single-step directive in effect (see PIOCRUN)
PR_ASLEEP	lwp is in an interruptible sleep within a system call
PR_PCINVAL	lwp's current instruction (**pr_instr**) is undefined
PR_ISSYS	process is a system process (see PIOCSTOP)
PR_FORK	process has its inherit-on-fork flag set (see PIOCSET)
PR_RLC	process has its run-on-last-close flag set (see PIOCSET)
PR_KLC	process has its kill-on-last-close flag set (see PIOCSET)
PR_ASYNC	process has its asynchronous-stop flag set (see PIOCSET)
PR_PCOMPAT	process has its ptrace-compatibility flag set (see PIOCSET)

pr_why and **pr_what** together describe, for a stopped lwp, the reason for the stop. Possible values of **pr_why** are:

PR_REQUESTED indicates that the stop occurred in response to a stop directive, normally because PIOCSTOP was applied or because another lwp stopped on an event of interest and the asynchronous-stop flag (see PIOCSET) was not set for the process. **pr_what** is unused in this case.

PR_SIGNALLED indicates that the lwp stopped on receipt of a signal (see PIOCSTRACE); **pr_what** holds the signal number that caused the stop (for a newly-stopped lwp, the same value is in **pr_cursig**).

PR_FAULTED indicates that the lwp stopped on incurring a hardware fault (see PIOCSFAULT); **pr_what** holds the fault number that caused the stop.

PR_SYSENTRY and **PR_SYSEXIT** indicate a stop on entry to or exit from a system call (see PIOCSENTRY and PIOCSEXIT); **pr_what** holds the system call number.

PR_JOBCONTROL indicates that the lwp stopped due to the default action of a job control stop signal (see *sigaction*(2)); **pr_what** holds the stopping signal number.

PR_SUSPENDED indicates that the lwp is stopped due to internal synchronization within the process: *lwp_stop*() was applied to the lwp or the LWP_STOP option was used with *lwp_create*() to create the lwp.

pr_who names the specific lwp. **pr_nlwp** is the total number of lwps in the process.

pr_cursig names the current signal, that is, the next signal to be delivered to the lwp. **pr_sigpend** identifies any other signals pending for the process. **pr_lwppend** identifies any synchronous or directed signals pending for the lwp. **pr_sighold** identifies those signals whose delivery is being delayed if sent to the lwp.

pr_info, when the lwp is in a PR_SIGNALLED or PR_FAULTED stop, contains additional information pertinent to the particular signal or fault (see <**sys/siginfo.h**>).

pr_altstack contains the alternate signal stack information for the lwp (see *sigaltstack*(2)). **pr_action** contains the signal action information pertaining to the current signal (see *sigaction*(2)); it is undefined if **pr_cursig** is zero.

pr_oldcontext, if not NULL, contains the address on the lwp stack of a **ucontext** structure describing the previous user-level context (see *ucontext*(5)). It is non-NULL only if the lwp is executing in the context of a signal handler.

pr_brkbase is the virtual address of the process heap and **pr_brksize** is its size in bytes. The address formed by the sum of these values is the process *break* (see *brk*(2)). **pr_stkbase** and **pr_stksize** are, respectively, the virtual address of the lwp stack and its size in bytes.

pr_syscall is the number of the system call, if any, being executed by the lwp; it is non-zero only if the lwp is stopped on PR_SYSENTRY or PR_SYSEXIT or is asleep within a system call (PR_ASLEEP is set). If **pr_syscall** is non-zero, **pr_nsysarg** is the number of arguments to the system call and the **pr_sysarg** array contains the actual arguments.

pr_pid, **pr_ppid**, **pr_pgrp**, and **pr_sid** are, respectively, the process id, the id of the process's parent, the process's process group id, and the process's session id.

pr_utime, **pr_stime**, **pr_cutime**, and **pr_cstime** are, respectively, the user CPU and system CPU time consumed by the process, and the cumulative user CPU and system CPU time consumed by the process's children, in seconds and nanoseconds.

pr_clname contains the name of the lwp's scheduling class.

pr_instr contains the machine instruction to which the lwp's program counter refers. The amount of data retrieved from the process is machine-dependent; on SPARC machines, it is a 32-bit word. In general, the size is that of the machine's smallest instruction. If PR_PCINVAL is set, **pr_instr** is undefined; this occurs whenever the lwp is not stopped or when the program counter refers to an invalid virtual address.

pr_reg is an array holding the contents of a stopped lwp's general registers. On SPARC machines the predefined constants **R_G0** … **R_G7**, **R_O0** … **R_O7**, **R_L0** … **R_L7**, **R_I0** … **R_I7**, **R_PSR**, **R_PC**, **R_nPC**, and **R_Y** can be used as indices to refer to the corresponding registers; previous register windows can be read from their overflow locations on the stack. If the lwp is not stopped, all register values are undefined.

When applied to an lwp file descriptor, PIOCSTATUS returns the status for the specific lwp. When applied to the process file descriptor, an lwp is chosen by the system for the operation. The chosen lwp is a stopped lwp only if all of the process's lwps are stopped, is stopped on an event of interest only if all of the lwps are so stopped (excluding PR_SUSPENDED lwps), is in a PR_REQUESTED stop only if there are no other events of interest to be found, or failing everything else is in a PR_SUSPENDED stop (implying that the process is deadlocked). The chosen lwp remains fixed so long as all of the lwps are either stopped on events of interest or are PR_SUSPENDED and PIOCRUN is not applied to any of them.

When applied to the process file descriptor, every */proc ioctl* operation that must act on an lwp uses the same algorithm to choose which lwp to act upon. Together with synchronous stopping (see PIOCSET), this enables a debugger to control a multiple-lwp process using only the process file descriptor if it so chooses. More fine-grained control can be achieved using individual lwp file descriptors.

PIOCLSTATUS

The PIOCLSTATUS operation fills in an array of *prstatus* structures addressed by *p*, one element (one structure) for each lwp in the process, containing the status that would be returned by applying PIOCSTATUS to the corresponding lwp file descriptor, plus an additional element at the beginning containing the status that would be returned by applying PIOCSTATUS to the process file descriptor.

***PIOCSTOP**
PIOCWSTOP

When applied to the process file descriptor, PIOCSTOP directs all lwps to stop and waits for them to stop; PIOCWSTOP simply waits for all lwps to stop. When applied to an lwp file descriptor, PIOCSTOP directs the specific lwp to stop and waits until it has stopped; PIOCWSTOP simply waits for the lwp to stop. When applied to an lwp file descriptor, these operations complete when the lwp stops on an event of interest, immediately if already so stopped. When applied to the process file descriptor, they complete when every lwp has stopped on an event of interest or has come to a PR_SUSPENDED stop. If *p* is non-zero it points to a **prstatus** structure to be filled with status information for the specific or chosen stopped lwp (see PIOCSTATUS).

An ''event of interest'' is either a PR_REQUESTED stop or a stop that has been specified in the process's tracing flags (set by PIOCSTRACE, PIOCSFAULT, PIOCSENTRY, and PIOCSEXIT). PR_JOBCONTROL and PR_SUSPENDED stops are specifically not events of interest. (An lwp may stop twice due to a stop signal, first showing PR_SIGNALLED if the signal is traced and again showing PR_JOBCONTROL if the lwp is set running without clearing the signal.) If PIOCSTOP is applied to an lwp that is stopped, but not on an event of interest, the stop directive takes effect when the lwp is restarted by the competing mechanism; at that time the lwp enters a PR_REQUESTED stop before executing any user-level code.

ioctl()s are interruptible by signals so that, for example, an *alarm*(2) can be set to avoid waiting forever for a process or lwp that may never stop on an event of interest. If PIOCSTOP is interrupted, the lwp stop directives remain in effect even though the *ioctl*() returns an error.

A system process (indicated by the PR_ISSYS flag) never executes at user level, has no user-level address space visible through **/proc**, and cannot be stopped. Applying PIOCSTOP or PIOCWSTOP to a system process or any of its lwps elicits the error EBUSY.

***PIOCRUN**

An lwp is made runnable again after a stop. If *p* is non-zero it points to a **prrun** structure describing additional actions to be performed. The **prrun** structure contains at least the following fields:

```
typedef struct prrun {
        long        pr_flags;      /* Flags */
        sigset_t    pr_trace;      /* Set of signals to be traced */
```

```
        sigset_t     pr_sighold;     /* Set of signals to be held */
        fltset_t     pr_fault;       /* Set of faults to be traced */
        caddr_t      pr_vaddr;       /* Virtual address at which to resume */
} prrun_t;
```

pr_flags is a bit-mask describing optional actions; the remainder of the entries are meaningful only if the appropriate bits are set in **pr_flags**. Flag definitions:

PRCSIG clears the current signal, if any (see PIOCSSIG).

PRCFAULT clears the current fault, if any (see PIOCCFAULT).

PRSTRACE sets the traced signal set to **pr_trace** (see PIOCSTRACE).

PRSHOLD sets the held signal set to **pr_sighold** (see PIOCSHOLD).

PRSFAULT sets the traced fault set to **pr_fault** (see PIOCSFAULT).

PRSVADDR sets the address at which execution resumes to **pr_vaddr**.

PRSTEP directs the lwp to execute a single machine instruction. On completion of the instruction, a trace trap occurs. If FLTTRACE is being traced, the lwp stops, otherwise it is sent SIGTRAP; if SIGTRAP is being traced and not held, the lwp stops. When the lwp stops on an event of interest the single-step directive is cancelled, even if the stop occurs before the instruction is executed. This operation requires hardware and operating system support and may not be implemented on all processors. It is implemented on SPARC machines.

PRSABORT is meaningful only if the lwp is in a PR_SYSENTRY stop or is marked PR_ASLEEP; it instructs the lwp to abort execution of the system call (see PIOCSENTRY, PIOCSEXIT).

PRSTOP directs the lwp to stop again as soon as possible after resuming execution (see PIOCSTOP). In particular if the lwp is stopped on PR_SIGNALLED or PR_FAULTED, the next stop will show PR_REQUESTED, no other stop will have intervened, and the lwp will not have executed any user-level code.

When applied to an lwp file descriptor PIOCRUN makes the specific lwp runnable. The operation fails (EBUSY) if the specific lwp is not stopped on an event of interest.

When applied to the process file descriptor an lwp is chosen for the operation as described under PIOCSTATUS. The operation fails (EBUSY) if the chosen lwp is not stopped on an event of interest. Otherwise, the chosen lwp is marked PR_REQUESTED. If as a consequence all lwps are in the PR_REQUESTED or PR_SUSPENDED stop state, all lwps showing PR_REQUESTED are made runnable.

PIOCLWPIDS This returns, in an array of *lwpid_t*'s addressed by *p*, the lwp identifiers of all the lwps that exist in the process, plus an entra identifier containing zero to mark the end of the list. The number of lwps in the process can be determined from the **pr_nlwp** field of the **prstatus** structure.

PIOCOPENLWP The return value *retval* provides a **/proc** file descriptor that refers to the lwp named in the *lwpid_t* addressed by *p*. The read/write attributes of the newly-acquired file descriptor are the same as those of the file descriptor used in the operation. The new file descriptor has an independent file offset for *lseek*(2). On error (no such lwp), −1 is returned and **errno** is set to ENOENT.

∗PIOCSTRACE This defines a set of signals to be traced in the process: the receipt of one of these signals by an lwp causes the lwp to stop. The set of signals is defined via an instance of **sigset_t** addressed by *p*. Receipt of SIGKILL cannot be traced.

If a signal that is included in an lwp's held signal set is sent to the lwp, the signal is not received and does not cause a stop until it is removed from the held signal set, either by the lwp itself or by setting the held signal set with PIOCSHOLD or the PRSHOLD option of PIOCRUN.

PIOCGTRACE The current traced signal set is returned in an instance of **sigset_t** addressed by *p*.

∗PIOCSSIG The current signal and its associated signal information for the specific or chosen lwp are set according to the contents of the **siginfo** structure addressed by *p* (see **<sys/siginfo.h>**). If the specified signal number is zero or if *p* is zero, the current signal is cleared. The semantics of this operation are different from those of *kill*(2), *lwp_kill*(2) or PIOCKILL in that the signal is delivered to the lwp immediately after execution is resumed (even if it is being held) and an additional PR_SIGNALLED stop does not intervene even if the signal is traced. Setting the current signal to SIGKILL terminates the process immediately.

∗PIOCKILL If applied to the process file descriptor, a signal is sent to the process with semantics identical to those of *kill*(2). If applied to an lwp file descriptor, a signal is sent to the lwp with semantics identical to those of *lwp_kill*(2). *p* points to an *int* naming the signal. Sending SIGKILL terminates the process immediately.

∗PIOCUNKILL A signal is deleted, that is, it is removed from the set of pending signals. If applied to the process file descriptor, the signal is deleted from the process's pending signals. If applied to an lwp file descriptor, the signal is deleted from the lwp's pending signals. The current signal (if any) is unaffected. *p* points to an *int* naming the signal. It is an error to attempt to delete SIGKILL.

PIOCGHOLD
∗PIOCSHOLD PIOCGHOLD returns the set of held signals for the specific or chosen lwp (signals whose delivery will be delayed if sent to the lwp) in an instance of *sigset_t* addressed by *p*. PIOCSHOLD correspondingly sets the lwp's held signal set but does not allow SIGKILL or SIGSTOP to be held.

PIOCMAXSIG
PIOCACTION These operations provide information about the signal actions associated with the traced process (see *sigaction*(2)). PIOCMAXSIG returns, in the *int* addressed by *p*, the maximum signal number understood by the system. This can be used to allocate storage for use with the PIOCACTION operation, which returns the traced process's signal actions in an array of **sigaction** structures addressed by *p*. Signal numbers are displaced by 1 from array indices, so that the action for signal number **n** appears in position **n**-1 of the array.

∗PIOCSFAULT This defines a set of hardware faults to be traced in the process: on incurring one of these faults an lwp stops. The set is defined via an instance of *fltset_t* addressed by *p*. Fault names are defined in **<sys/fault.h>** and include the following. Some of these may not occur on all processors; there may be processor-specific faults in addition to these.

FLTILL	illegal instruction
FLTPRIV	privileged instruction
FLTBPT	breakpoint trap
FLTTRACE	trace trap
FLTACCESS	memory access fault (bus error)
FLTBOUNDS	memory bounds violation
FLTIOVF	integer overflow
FLTIZDIV	integer zero divide
FLTFPE	floating-point exception
FLTSTACK	unrecoverable stack fault
FLTPAGE	recoverable page fault

When not traced, a fault normally results in the posting of a signal to the lwp that incurred the fault. If an lwp stops on a fault, the signal is posted to the lwp when execution is resumed unless the fault is cleared by PIOCCFAULT or by the PRCFAULT option of PIOCRUN. FLTPAGE is an exception; no signal is posted. There may be additional processor-specific faults like this. **pr_info** in the **prstatus** structure identifies the signal to be sent and contains machine-specific information about the fault.

PIOCGFAULT The current traced fault set is returned in an instance of **fltset_t** addressed by *p*.

∗PIOCCFAULT The current fault (if any) is cleared; the associated signal is not sent to the specific or chosen lwp.

∗PIOCSENTRY
∗PIOCSEXIT These operations instruct the process's lwps to stop on entry to or exit from specified system calls. The set of system calls to be traced is defined via an instance of *sysset_t* addressed by *p*.

When entry to a system call is being traced, an lwp stops after having begun the call to the system but before the system call arguments have been fetched from the lwp. When exit from a system call is being traced, an lwp stops on completion of the system call just prior to checking for signals and returning to user level. At this point all return values have been stored into the lwp's registers.

If an lwp is stopped on entry to a system call (PR_SYSENTRY) or when sleeping in an interruptible system call (PR_ASLEEP is set), it may be instructed to go directly to system call exit by specifying the PRSABORT flag in a PIOCRUN request. Unless exit from the system call is being traced the lwp returns to user level showing error EINTR.

PIOCGENTRY
PIOCGEXIT These return the current traced system call entry or exit set in an instance of **sysset_t** addressed by *p*.

∗PIOCSET
∗PIOCRESET

PIOCSET sets one or more modes of operation for the traced process. PIOCRESET resets these modes. The modes to be set or reset are specified by flags in a *long* addressed by *p*:

PR_FORK (inherit-on-fork): When set, the process's tracing flags are inherited by the child of a *fork*(2) or *vfork*(2). When reset, child processes start with all tracing flags cleared.

PR_RLC (run-on-last-close): When set and the last writable **/proc** file descriptor referring to the traced process or any of its lwps is closed, all of the process's tracing flags are cleared, any outstanding stop directives are canceled, and if any lwps are stopped on events of interest, they are set running as though PIOCRUN had been applied to them. When reset, the process's tracing flags are retained and lwps are not set running on last close.

PR_KLC (kill-on-last-close): When set and the last writable **/proc** file descriptor referring to the traced process or any of its lwps is closed, the process is terminated with SIGKILL.

PR_ASYNC (asynchronous-stop): When set, a stop on an event of interest by one lwp does not directly affect any other lwp in the process. When reset and an lwp stops on an event of interest other than PR_REQUESTED, all other lwps in the process are directed to stop.

PR_PCOMPAT (ptrace-compatibility): When set, a stop on an event of interest by the traced process is reported to the parent of the traced process via *wait*(2), SIGTRAP is sent to the traced process when it executes a successful *exec*(2), setuid/setgid flags are not honored for execs performed by the traced process, any exec of an object file that the traced process cannot read fails, and the traced process dies when its parent dies. This mode is deprecated; it is provided only to allow **ptrace**() to be implemented as a library function using **/proc**.

It is an error (EINVAL) to specify flags other than those described above or to apply these operations to a system process. The current modes are reported in the **prstatus** structure (see PIOCSTATUS).

PIOCGREG
∗PIOCSREG

These operations respectively get and set the general registers for the specific or chosen lwp into or out of an array addressed by *p*; the array has type **prgregset_t**. Register contents are accessible using a set of predefined indices (see PIOCSTATUS). Only certain bits of the processor-status register (R_PS) can be modified by PIOCSREG; on SPARC machines these include only the condition-code bits. Other privileged registers cannot be modified at all. PIOCSREG fails (EBUSY) if the lwp is not stopped on an event of interest. If the lwp is not stopped, the register values returned by PIOCGREG are undefined.

PIOCGFPREG
∗PIOCSFPREG

These operations respectively get and set the floating-point registers for the specific or chosen lwp into or out of a structure addressed by *p*; the structure has type **prfpregset_t**. An error (EINVAL) is returned if there is no floating-point hardware on the machine. PIOCSFPREG fails (EBUSY) if the lwp is not stopped on an event of interest. If the lwp is not stopped, the register values returned by PIOCGFPREG are undefined.

***PIOCNICE** The traced process's *nice*(2) priority is incremented by the amount contained in the *int* addressed by *p*. Only the super-user may better a process's priority in this way, but any user may make the priority worse. This operation is meaningful only when applied to a process in the time-sharing scheduling class.

PIOCPSINFO This returns miscellaneous process information such as that reported by *ps*(1). *p* is a pointer to a **prpsinfo** structure containing at least the following fields:

```
typedef struct prpsinfo {
        char          pr_state;         /* numeric process state (see pr_sname) */
        char          pr_sname;         /* printable character representing pr_state */
        char          pr_zomb;          /* !=0: process terminated but not waited for */
        char          pr_nice;          /* nice for cpu usage */
        u_long        pr_flag;          /* process flags */
        uid_t         pr_uid;           /* real user id */
        gid_t         pr_gid;           /* real group id */
        pid_t         pr_pid;           /* unique process id */
        pid_t         pr_ppid;          /* process id of parent */
        pid_t         pr_pgrp;          /* pid of process group leader */
        pid_t         pr_sid;           /* session id */
        caddr_t       pr_addr;          /* physical address of process */
        long          pr_size;          /* size of process image in pages */
        long          pr_rssize;        /* resident set size in pages */
        caddr_t       pr_wchan;         /* wait addr for sleeping process */
        short         pr_syscall;       /* system call number (if in syscall) */
        timestruc_t   pr_start;         /* process start time, sec+nsec since epoch */
        timestruc_t   pr_time;          /* usr+sys cpu time for this process */
        timestruc_t   pr_ctime;         /* usr+sys cpu time for reaped children */
        long          pr_pri;           /* priority, high value is high priority */
        char          pr_oldpri;        /* pre-SVR4, low value is high priority */
        char          pr_cpu;           /* pre-SVR4, cpu usage for scheduling */
        dev_t         pr_ttydev;        /* controlling tty device (PRNODEV if none) */
        char          pr_clname[PRCLSZ];       /* scheduling class name */
        char          pr_fname[PRFNSZ];        /* last component of exec()ed pathname */
        char          pr_psargs[PRARGSZ];      /* initial characters of arg list */
} prpsinfo_t;
```

Some of the entries in **prpsinfo**, such as **pr_state** and **pr_flag**, are system-specific and should not be expected to retain their meanings across different versions of the operating system. **pr_addr** is a vestige of the past and has no real meaning in current systems.

PIOCPSINFO can be applied to a *zombie* process (one that has terminated but whose parent has not yet performed a **wait**(2) on it).

PIOCNMAP
PIOCMAP These operations provide information about the memory mappings (virtual address ranges) associated with the traced process. PIOCNMAP returns, in the *int* addressed by *p*, the number of mappings that are currently active. This can be used to allocate storage for use with the PIOCMAP operation, which returns the list of currently active mappings. For

PIOCMAP, **p** addresses an array of elements of type **prmap_t**; one array element (one structure) is returned for each mapping, plus an additional element containing all zeros to mark the end of the list. The **prmap** structure contains at least the following fields:

```
typedef struct prmap {
        caddr_t     pr_vaddr;      /* Virtual address */
        u_long      pr_size;       /* Size of mapping in bytes */
        off_t       pr_off;        /* Offset into mapped object, if any */
        long        pr_mflags;     /* Protection and attribute flags */
} prmap_t;
```

pr_vaddr is the virtual address of the mapping within the traced process and **pr_size** is its size in bytes. **pr_off** is the offset within the mapped object (if any) to which the virtual address is mapped.

pr_mflags is a bit-mask of protection and attribute flags:

MA_READ	mapping is readable by the traced process
MA_WRITE	mapping is writable by the traced process
MA_EXEC	mapping is executable by the traced process
MA_SHARED	mapping changes are shared by the mapped object

A contiguous area of the address space having the same underlying mapped object may appear as multiple mappings due to varying read/write/execute/shared attributes. The underlying mapped object does not change over the range of a single mapping. An I/O operation to a mapping marked MA_SHARED fails if applied at a virtual address not corresponding to a valid page in the underlying mapped object. A write to a MA_SHARED mapping that is not marked MA_WRITE fails. Reads and writes to private mappings always succeed. Reads and writes to unmapped addresses always fail.

PIOCOPENM The return value *retval* provides a read-only file descriptor for a mapped object associated with the traced process. If *p* is zero the traced process's *exec*(2)ed file is found. This enables a debugger to find the object file symbol table without having to know the path name of the executable file. If **p** is non-zero it points to a **caddr_t** containing a virtual address within the traced process and the mapped object, if any, associated with that address is found; this can be used to get a file descriptor for a shared library that is attached to the process. On error (invalid address or no mapped object for the designated address), −1 is returned and *errno* is set to EINVAL.

PIOCCRED Fetch the set of credentials associated with the process. *p* points to an instance of **prcred_t** which is filled by the operation. The **prcred** structure contains at least the following fields:

```
typedef struct prcred {
        uid_t       pr_euid;       /* Effective user id */
        uid_t       pr_ruid;       /* Real user id */
        uid_t       pr_suid;       /* Saved user id (from exec) */
        gid_t       pr_egid;       /* Effective group id */
        gid_t       pr_rgid;       /* Real group id */
        gid_t       pr_sgid;       /* Saved group id (from exec) */
```

```
        u_int          pr_ngroups;      /* Number of supplementary groups */
} prcred_t;
```

PIOCGROUPS
Fetch the set of supplementary group IDs associated with the process. *p* points to an array of elements of type **gid_t**, which will be filled by the operation. PIOCCRED can be applied beforehand to determine the number of groups (**pr_ngroups**) that will be returned and the amount of storage that should be allocated to hold them.

PIOCUSAGE
When applied to the process file descriptor, PIOCUSAGE returns the process usage information; when applied to an lwp file descriptor, it returns usage information for the specific lwp. *p* points to a **prusage** structure which is filled by the operation. The **prusage** structure contains at least the following fields:

```
typedef struct prusage {
        lwpid_t        lwpid;           /* LWP identifier, 0: process or defunct */
        u_long         count;           /* number of contributing LWPs */
        timestruc_t    tstamp;          /* time stamp */
        timestruc_t    rtime;           /* real time */
        timestruc_t    utime;           /* user CPU time */
        timestruc_t    stime;           /* system CPU time */
        timestruc_t    ftime;           /* fault wait time */
        timestruc_t    otime;           /* other wait time */
        timestruc_t    ktime;           /* kernel lock wait time */
        timestruc_t    ltime;           /* user lock wait time */
        timestruc_t    itime;           /* interrupt time */
        timestruc_t    wtime;           /* wait-cpu time */
        u_long         minf;            /* minor page faults */
        u_long         majf;            /* major page faults */
        u_long         nswap;           /* swaps */
        u_long         inblk;           /* input blocks */
        u_long         oublk;           /* output blocks */
        u_long         msnd;            /* messages sent */
        u_long         mrcv;            /* messages received */
        u_long         sigs;            /* signals received */
        u_long         vctx;            /* voluntary context switches */
        u_long         ictx;            /* involuntary context switches */
        u_long         sysc;            /* system calls */
        u_long         ioch;            /* chars read and written */
} prusage_t;
```

PIOCLUSAGE
The PIOCLUSAGE operation fills in an array of *prusage* structures addressed by *p*, one element for each lwp in the process plus an additional element at the beginning that contains the summation over all defunct lwps (lwps that once existed but no longer exist in the process). Excluding the **lwpid**, **tstamp**, **rtime** and **nswap** entries, the entry-by-entry summation over all these structures is the definition of the process usage information.

PIOCOPENPD

The return value *retval* provides a read-only file descriptor for a ''page data file'', enabling tracking of address space references and modifications on a per-page basis.

A *read*(2) of the page data file descriptor returns structured page data and atomically clears the page data maintained for the file by the system. That is to say, each read returns data collected since the last read; the first read returns data collected since the file was opened. When the call completes, the read buffer contains the following structure as its header and thereafter contains a number of variable length structures that must be accessed by walking linearly through the buffer.

```
typedef struct prpageheader {
        timestruc_t   tstamp;       /* real time time stamp */
        u_long        nmap;         /* number of address space mappings */
        u_long        npage;        /* total number of pages */
} prpageheader_t;
```

The header is followed by **nmap** variable-length **prasmap** structures:

```
typedef struct prasmap {
        caddr_t       vaddr;        /* virtual address */
        u_long        npage;        /* number of pages in mapping */
        u_char        data[1];      /* referenced, modified, present flags */
} prasmap_t;
```

The **data[]** array is of variable length, with one entry for each page in the mapping, **npage** entries altogether, rounded up with empty entries at the end so that the structure size is an integral numbers of *long*'s. **data[]** entries may contain these flags:

PG_REFERENCED page has been referenced
PG_MODIFIED page has been modified

If the read buffer is not large enough to contain all of the page data, the read fails with E2BIG and the page data is not cleared. The required size of the read buffer can be determined through *fstat*(2). Application of *lseek*(2) to the page data file descriptor is ineffective. Closing the page data file descriptor terminates the system overhead associated with collecting the data.

PIOCGETPR
PIOCGETU

These operations copy, respectively, the traced process's *proc* structure and *user* structure into the buffer addressed by *p*. They are provided for completeness but it should be unnecessary to access either of these structures directly since relevant status information is available through other control operations. Their use is discouraged because a program making use of them is tied to a particular version of the operating system.

PIOCGETPR can be applied to a *zombie* process (see PIOCPSINFO).

NOTES

Each operation (*ioctl* or I/O) is guaranteed to be atomic with respect to the traced process, except when applied to a system process. I/O to memory that is shared by another process or is the target of asynchronous I/O is not guaranteed to be atomic.

For security reasons, except for the super-user, an open of a **/proc** file fails unless both the user-ID and group-ID of the caller match those of the traced process and the process's object file is readable by the caller. Files corresponding to setuid and setgid processes can be opened only by the super-user. Even if held by the super-user, an open process or lwp file descriptor becomes invalid if the traced process performs an *exec*() of a setuid/setgid object file or an object file that it cannot read. Any operation performed on an invalid file descriptor, except *close*(2), fails with EAGAIN. In this situation, if any tracing flags are set and the process or any lwp file descriptor is open for writing, the process will have been directed to stop and its run-on-last-close flag will have been set (see PIOCSET). This enables a controlling process (if it has permission) to reopen the process file to get a new valid file descriptor, close the invalid file descriptors, and proceed. Just closing the invalid file descriptors causes the traced process to resume execution with no tracing flags set. Any process not currently open for writing via **/proc** but that has left-over tracing flags from a previous open and that *exec*s a setuid/setgid or unreadable object file will not be stopped but will have all its tracing flags cleared.

To wait for one or more of a set of processes or lwps to stop, **/proc** file descriptors can be used in a *poll*(2) system call. When requested and returned, the polling event POLLPRI indicates that the process or lwp stopped on an event of interest. Although they cannot be requested, the polling events POLLHUP, POLLERR and POLLNVAL may be returned. POLLHUP indicates that the process or lwp has terminated. POLLERR indicates that the file descriptor has become invalid. POLLNVAL is returned immediately if POLLPRI is requested on a file descriptor referring to a system process (see PIOCSTOP).

Descriptions of structures in this document include only interesting structure elements, not filler and padding fields, and may show elements out of order for descriptive clarity. The actual structure definitions are contained in **<sys/procfs.h>**.

For reasons of symmetry and efficiency there are more control operations than strictly necessary.

BUGS

The types **gregset_t** and **fpregset_t** defined in **<sys/reg.h>** are similar to but not the same as the types **prgregset_t** and **prfpregset_t** defined in **<sys/procfs.h>**.

FILES

/proc directory (list of processes)
/proc/*nnnnn* process file

SEE ALSO

open(2), *poll*(2), *sigaction*(2), *signal*(2), *signal*(5), *siginfo*(5)

DIAGNOSTICS

Errors that can occur in addition to the errors normally associated with file system access:

ENOENT The traced process or lwp has terminated after being opened.

EIO I/O was attempted at an illegal address in the traced process.

EBADF An I/O or *ioctl* operation requiring write access was attempted on a file descriptor not open for writing.

EBUSY PIOCSTOP or PIOCWSTOP was applied to a system process; an exclusive *open*(2) was attempted on a process file already already open for writing; an *open*(2) for writing was attempted and an exclusive open is in effect on the process file; PIOCRUN, PIOCSREG or PIOCSFPREG was applied to a process or lwp not stopped on an event of interest; an attempt was made to mount **/proc** when it is already mounted.

EPERM Someone other than the super-user attempted to better a process's priority by issuing PIOCNICE.

ENOSYS An attempt was made to perform an unsupported operation (such as create, remove, link, or unlink) on an entry in **/proc**.

EFAULT An I/O or *ioctl* request referred to an invalid address in the controlling process.

EINVAL In general this means that some invalid argument was supplied to a system call. The list of conditions eliciting this error includes: the *ioctl* code is undefined; an *ioctl* operation was issued on a file descriptor referring to the **/proc** directory; an out-of-range signal number was specified with PIOCSSIG, PIOCKILL, or PIOCUNKILL; SIGKILL was specified with PIOCUNKILL; an illegal virtual address was specified in a PIOCOPENM request; PIOCGFPREG or PIOCSFPREG was issued on a machine without floating-point hardware.

E2BIG Data to be returned in a *read*(2) of the page data file exceeds the size of the read buffer provided by the caller.

EINTR A signal was received by the controlling process while waiting for the traced process or lwp to stop via PIOCSTOP or PIOCWSTOP.

EAGAIN The traced process has performed an *exec* of a setuid/setgid object file or of an object file that it cannot read; all further operations on the process or lwp file descriptor (except *close*(2)) elicit this error.