sun® microsystems

UNIX Interface Overview

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

[illegible]
R[illegible] February 1986

Credits and Acknowledgements

This manual is derived directly from the *System Interface Overview* from the 4.2BSD documentation.

Trademarks

Contents

Preface

This document summarizes the facilities provided by the 1.1, 2.0, 3.0, and later releases of the UNIX† operating system for the Sun Workstation. It does not attempt to act as a tutorial for use of the system nor does it attempt to explain or justify the design of the system facilities. It gives neither motivation nor implementation details, in favor of brevity. Discussions of system facilities in this document is in two major parts:

Part I describes the basic kernel functions provided to a UNIX process: process naming and protection in chapter 2, memory management in chapter 3, software interrupts in chapter 4, object references (descriptors) in chapter 6, time and statistics functions in chapter 5, and resource controls in chapter 7. These facilities, as well as facilities for bootstrap, shutdown and process accounting described in 8, are provided solely by the kernel.

Part II describes the standard system abstractions for files and file systems in chapter 9, 10 and 12, communication and terminal handling in chapter 11, and process control and debugging in chapter 13. These facilities are implemented by the operating system or by network server processes.

Appendix A provides a quick-reference list of facilities.

† UNIX is a trademark of AT&T Bell Laboratories.

1

Introduction

1

Introduction

Facilities available to a UNIX user process are logically divided into two parts: kernel facilities directly implemented by UNIX code running in the operating system, and system facilities implemented either by the system, or in cooperation with a *server process*.

Facilities implemented in the kernel are those which define the *UNIX virtual machine* which each process runs in. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The UNIX virtual machine also allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines.

System abstractions described are:

Directory Contexts
: A directory context is a position in the UNIX file system name space. Operations on files and other named objects in a file system are always specified relative to such a context.

Files
: Files are used to store uninterpreted sequence of bytes on which random access *reads* and *writes* may occur. Pages from files or devices may also be mapped into process address space. A directory may be read as a file[1].

Communications Domains
: A communications domain represents an interprocess communications environment, such as the communications facilities of the UNIX system, communications in the INTERNET, or the resource sharing protocols and access rights of a resource sharing system on a local network.

Sockets
: A socket is an endpoint of communication and the focal point for IPC in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain,

[1] Support for mapping files is not included in this release.

accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

Terminals and other devices
Devices include terminals, providing input editing and interrupt generation and output flow control and editing, magnetic tapes, disks and other peripherals. They often support the generic *read* and *write* operations as well as a number of *ioctl* s.

Processes
Process descriptors provide facilities for control and debugging of other processes.

1.1. Notation and Types

The notation used to describe system calls is a variant of a C language call, consisting of a prototype call followed by declaration of parameters and results. An additional keyword `result`, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the `read` call, as described in section 9.1.

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result char *buf; int nbytes;
```

The first line shows how the `read` routine is called, with three parameters. As shown on the second line `cc` is an integer and `read` also returns information in the parameter `buf`.

Description of all error conditions arising from each system call is not provided here; they appear in the *intro* (2) manual page of the *System Interface Manual*. All error codes also appear in the index to the *System Interface Manual*. In particular, when accessed from the C language, many calls return a characteristic −1 value when an error occurs, returning the error code in the global variable `errno`. Since some calls return −1 as a legitimate value, you may have to check `errno` to determine if the return value is genuine or an error. Other languages may present errors in different ways.

A number of system standard types are defined in the `<sys/types.h>` include file and used in the specifications here and in many C programs. These include `caddr_t` giving a memory address (typically as a character pointer), `off_t` giving a file offset (typically as a long integer), and a set of unsigned types `u_char`, `u_short`, `u_int` and `u_long`, shorthand names for `unsigned char`, `unsigned short`, and so on.

2

Processes and Protection

2

Processes and Protection

2.1. Host and Process Identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 255 characters. These are set (by a privileged user) and returned by the calls:

```
getdomainname(name, namelength);
char *name;
int namelength;

setdomainname(name, namelength);
char *name;
int namelength;

hostid = gethostid();
result long hostid;

sethostname(name, len);
char *name; int len;

gethostname(buf, buflen);
result char *buf; int buflen;
```

`getdomainname` returns the name of the domain for the current processor. `setdomainname` sets the domain of the current processor to `name`.

The host id is not used in this release of the system. The *buf* containing the host name returned by `gethostname` is null-terminated (if space allows).

On each host runs a set of *processes*. Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process id*. This number is in the range 1-30000 and is returned by the `getpid` routine:

```
pid = getpid();
result int pid;
```

On each UNIX host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process id) pairs are guaranteed unique.

2.2. Creating and Terminating Processes

A new process is created by making a logical duplicate of an existing process:

```
pid = fork();
result int pid;
```

The `fork` call returns twice, once in the parent process, where *pid* is the process identifier of the child, and once in the child process where *pid* is 0. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

A process may terminate by executing an `exit` call:

```
exit(status);
int status;
```

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

```
#include <sys/wait.h>

pid = wait(astatus);
result int pid; result union wait *astatus;

pid = wait3(astatus, options, arusage);
result int pid; result union waitstatus *astatus;
int options; result struct rusage *arusage;
```

A process can overlay itself with the memory image of another process, passing the newly created process a set of parameters, using the call:

```
execve(name, argv, envp)
char *name, **argv, **envp;
```

The specified *name* must be a file which is in a format recognized by the system, either a binary executable file or a file which causes the execution of a specified interpreter program to process its contents.

2.3. User and Group Ids

Each process in the system has associated with it two user-id's: a *real user id* and a *effective user id*, both non-negative 16 bit integers. Each process has an *real accounting group id* and an *effective accounting group id* and a set of *access group id's*. The group id's are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant `NGROUPS` in the file <sys/param.h>, guaranteed to be at least 8.

The real and effective user ids associated with a process are returned by:

```
ruid = getuid();
result int ruid;

euid = geteuid();
result int euid;
```

the real and effective accounting group ids by:

```
rgid = getgid();
result int rgid;

egid = getegid();
result int egid;
```

and the access group id set is returned by a `getgroups` call:

```
ngroups = getgroups(gidsetsize, gidset);
result int ngroups;
int gidsetsize;
result int gidset[gidsetsize];
```

The user and group id's are assigned at login time using the `setreuid`, `setregid`, and `setgroups` calls:

```
setreuid(ruid, euid);
int ruid, euid;

setregid(rgid, egid);
int rgid, egid;

setgroups(gidsetsize, gidset);
int gidsetsize; int gidset[gidsetsize];
```

The `setreuid` call sets both the real and effective user-id's, while the `setregid` call sets both the real and effective accounting group id's. Unless the caller is the super-user, *ruid* must be equal to either the current real or effective user-id, and *rgid* equal to either the current real or effective accounting group id. The `setgroups` call is restricted to the super-user.

2.4. Process Groups and System Terminals

Each process in the system is also normally associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the `getpgrp` call:

```
pgrp = getpgrp(pid);
result int pgrp; int pid;
```

The process group associated with a process may be changed by the `setpgrp` call:

```
setpgrp(pid, pgrp);
int pid, pgrp;
```

Newly created processes are assigned process id's distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process id. A privileged process may set the process group of any process to any value.

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, every system terminal has a process group and only processes which are in the process group of a terminal may read from the terminal, allowing arbitration of terminals among several different jobs. A process can examine the process group of a terminal via the `ioctl` call:

```
ioctl(fd, TIOCGPGRP, pgrp);
int fd; result int *pgrp;
```

A process may change the process group of any terminal which it can write by the `ioctl` call:

```
ioctl(fd, TIOCSPGRP, pgrp);
int fd; int *pgrp;
```

The terminal's process group may be set to any value. Thus, more than one terminal may be in a process group.

Control Terminal

Each process in the system is usually associated with a *control terminal*, accessible through the file **/dev/tty**. A newly created process inherits the control terminal of its parent. A process may be in a different process group than its control terminal, in which case the process does not receive software interrupts affecting the control terminal's process group.

You can arrange for a process to be detached from the control terminal, via this code sequence:

```
if ((i = open("/dev/tty"), O_RDONLY) >= 0)
    (void)ioctl(i, TIOCNOTTY, (char *)0);
```

3

Memory Management

3

Memory Management

This section represents the interface planned for later releases of the system. Of the calls described in this section, only `sbrk`, `getpagesize`, and *mmap* are included in the current release. Note that *mmap* is restricted in that it only works with certain character devices such as the framebuffer and devices like `mbmem`.

3.1. Text, Data, and Stack

Each process begins execution with three logical areas of memory called text, data and stack. The text area is read-only and shared, while the data and stack areas are private to the process. Both the data and stack areas may be extended and contracted on program request. The call

```
addr = sbrk(incr);
result caddr_t addr; int incr;
```

changes the size of the data area by *incr* bytes and returns the new end of the data area. The stack area is automatically extended as needed.

On the Sun system, the program text and data are at the low end of the address space, and the stack is at the high end of the address space. The area between the two is not accessible. The stack expands as necessary to accommodate the process's stack usage. The data area can be expanded by explicit system calls. The data area is usually called the 'heap'.

On the VAX the text and data areas are adjacent in the P0 region, while the stack section is in the P1 region, and grows downward.

3.2. Mapping Pages

The system supports sharing of data between processes by allowing pages to be mapped into memory. These mapped pages may be *shared* with other processes or *private* to the process. Protection and sharing options are defined in <mman.h> as:

```
/* protections are chosen from these bits, or-ed together */
#define PROT_READ   0x4 /* pages can be read */
#define PROT_WRITE  0x2 /* pages can be written */
#define PROT_EXEC   0x1 /* pages can be executed */

/* sharing types; choose either SHARED or PRIVATE */
#define MAP_SHARED  1   /* share changes */
#define MAP_PRIVATE 2   /* changes are private */
```

The cpu-dependent size of a page is returned by the `getpagesize` system call:

```
pagesize = getpagesize();
result int pagesize;
```

The call:

```
mmap(addr, len, prot, share, fd, pos);
caddr_t addr; int len, prot, share, fd; off_t pos;
```

maps the pages starting at *addr* and continuing for *len* bytes from the object represented by descriptor *fd*, at absolute position *pos*. The parameter *share* specifies whether modifications made to this mapped copy of the page, are to be kept *private*, or are to be *shared* with other references. The parameter *prot* specifies the accessibility of the mapped pages. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize.

A mapping can be removed by the call

```
munmap(addr, len);
caddr_t addr; int len;
```

Further references to these pages will refer to private pages initialized to zero.

NOTE mmap *and* munmap *are not implemented in 4.2 BSD. They are implemented in the Sun system, but in a limited way — they are used for mapping frame buffers into a process's address space.*

4

Signals

4 Signals

The system defines a set of *signals* that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored.* A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a *core* image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

4.1. Signal Types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file `<signal.h>`.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include `SIGFPE` representing floating point and other arithmetic exceptions, `SIGILL` for illegal instruction execution, `SIGSEGV` for addresses outside the currently assigned area of memory, and `SIGBUS` for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as those for the various customer-reserved instructions on the VAX (`SIGIOT`, `SIGEMT`, and `SIGTRAP`).

Software signals reflect interrupts generated by user request: `SIGINT` for the normal interrupt signal; `SIGQUIT` for the more powerful *quit* signal, that normally causes a core image to be generated; `SIGHUP` and `SIGTERM` that cause graceful process termination, either because a user has ''hung up'', or by user or program request; and `SIGKILL`, a more powerful termination signal which a process cannot catch or ignore. Other software signals (SIGALRM, `SIGVTALRM`, `SIGPROF`) indicate the expiration of interval timers.

A process can request notification via a SIGIO signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a SIGURG signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason the process is being stopped. A SIGCONT signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a SIGCHLD signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. SIGXCPU occurs when a process nears its CPU time limit and SIGXFSZ warns that the limit on file size creation has been reached.

4.2. Signal Handlers

A process has a handler associated with each signal that controls the way the signal is delivered. The call

```
#include <signal.h>

struct sigvec {
    int (*sv_handler)();
    int sv_mask;
    int sv_onstack;
};

sigvec(signo, sv, osv)
int signo; struct sigvec *sv; result struct sigvec *osv;
```

assigns interrupt handler address *sv_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DFL used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv_mask* and *sv_onstack* values specify the signal mask to be used when the handler is invoked and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sv_mask* to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

The mask of *blocked* signals is independent of handlers for signals. It prevents signals from being delivered much as a raised hardware interrupt priority level prevents hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine *sv_handler* is called by a C call of the form

```
(*sv_handler)(signo, code, scp);
int signo; long code; struct sigcontext *scp;
```

The *signo* gives the number of the signal that occurred, and the *code*, a word of information supplied by the hardware. The *scp* parameter is a pointer to a machine-dependent structure containing the information for restoring the context before the signal.

4.3. Sending Signals

A process can send a signal to another process or group of processes with the calls:

```
kill(pid, signo);
int pid, signo;

killpgrp(pgrp, signo);
int pgrp, signo;
```

Unless the process sending the signal is privileged, it and the process receiving the signal must have the same effective user id.

Signals are also sent implicitly from a terminal device to the process group associated with the terminal when certain input characters are typed.

4.4. Protecting Critical Sections

To block a section of code against one or more signals, a *sigblock* call may be used to add a set of signals to the existing mask, returning the old mask:

```
oldmask = sigblock(mask);
result long oldmask; long mask;
```

The old mask can then be restored later with *sigsetmask*,

```
oldmask = sigsetmask(mask);
result long oldmask; long mask;
```

The *sigblock* call can be used to read the current mask by specifying an empty *mask*.

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
sigpause(mask);
long mask;
```

4.5. Signal Stacks

Applications that maintain complex or fixed size stacks can use the call

```
struct sigstack {
    caddr_t ss_sp;
    int ss_onstack;
};

sigstack(ss, oss)
struct sigstack *ss; result struct sigstack *oss;
```

to provide the system with a stack based at *ss_sp* for delivery of signals. The value *ss_onstack* indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a *sigstack* call should be used to reset the signal stack.

5

Timers

5

Timers

5.1. Real Time

The system's notion of the current Greenwich time and the current time zone is set and returned by the calls:

```
#include <sys/time.h>

settimeofday(tvp, tzp);
struct timeval *tp;
struct timezone *tzp;

gettimeofday(tp, tzp);
result struct timeval *tp;
result struct timezone *tzp;
```

where the structures are defined in `<sys/time.h>` as:

```
struct timeval {
    long    tv_sec; /* seconds since Jan 1, 1970 */
    long    tv_usec;    /* and microseconds */
};

struct timezone {
    int tz_minuteswest; /* of Greenwich */
    int tz_dsttime; /* type of dst correction to apply */
};
```

Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvp)
result long *tvp;
```

or

```
tv = time((long *)0);
result long tv;
```

returning only the `tv_sec` field from the `gettimeofday` call.

5.2. Interval Time

The system provides each process with three interval timers, defined in `<sys/time.h>`:

```
#define ITIMER_REAL 0   /* real time intervals */
#define ITIMER_VIRTUAL  1   /* virtual time intervals */
#define ITIMER_PROF 2   /* user and system virtual time */
```

The `ITIMER_REAL` timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. A `SIGALRM` signal is delivered when this timer expires.

The `ITIMER_VIRTUAL` timer decrements in process virtual time. It runs only when the process is executing. A `SIGVTALRM` signal is delivered when it expires.

The `ITIMER_PROF` timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by processes to statistically profile their execution. A `SIGPROF` signal is delivered when it expires.

A timer value is defined by the `itimerval` structure:

```
struct itimerval {
    struct  timeval it_interval;    /* timer interval */
    struct  timeval it_value;   /* current value */
};
```

and a timer is set or read by the call:

```
getitimer(which, value);
int which; result struct itimerval *value;

setitimer(which, value, ovalue);
int which; struct itimerval *value;
result struct itimerval *ovalue;
```

The third argument to `setitimer` specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The `alarm` system call of earlier versions of UNIX is provided as a library routine using the `ITIMER_REAL` timer. The process profiling facilities of earlier versions of UNIX remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal.

```
profil(buf, bufsize, offset, scale);
result char *buf; int bufsize, offset, scale;
```

6

Descriptors

6 Descriptors

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

6.1. The Reference Table

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

Each process has a fixed size descriptor reference table, where the size is returned by the `getdtablesize` call:

```
nds = getdtablesize();
result int nds;
```

and guaranteed to be at least 20. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

6.2. Descriptor Properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. Generic operations applying to many of these types are described in 9. Naming contexts, files and directories are described in 10. Section 11. describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in 12.

6.3. Managing Descriptor References

A duplicate of a descriptor reference may be made by doing

```
new = dup(old);
result int new; int old;
```

returning a copy of descriptor reference *old* indistinguishable from the original. The *new* chosen by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
dup2(old, new);
int old, new;
```

The `dup2` call causes the system to deallocate the descriptor reference current occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
close(old);
int old;
```

6.4. Multiplexing Requests

The system provides a standard way to do synchronous and asynchronous multiplexing of operations.

Synchronous multiplexing is performed by using the `select` call:

```
nds = select(nd, in, out, except, tvp);
result int nds; int nd; result *in, *out, *except;
struct timeval *tvp;
```

The `select` call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the specified bit masks by the subsets that select for input, output, and exceptional conditions respectively (*nd* indicates the size, in bytes, of the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned in *nds* and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as `read` or `receive` is possible, or if a connection request may be accepted (see *Accepting Connections* in section 11.1.
- A descriptor selects for output if an output oriented operation such as `write` or `send` is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section 9.3.
- A descriptor selects for an exceptional condition if a condition that would cause a `SIGURG` signal to be generated exists (see section 4.1.

If none of the specified conditions is true, the operation blocks for at most the amount of time specified by *tvp*, or waits for one of the conditions to arise if *tvp* is given as 0.

Options affecting I/O on a descriptor may be read and set by the call:

```
dopt = fcntl(d, cmd, arg);
result int dopt; int d, cmd, arg;

/* interesting values for cmd */
#define F_DUPFD     0   /* return new descriptor */
#define F_SETFD     1   /* get close-on-exec flag */
#define F_GETFD     2   /* set close-on-exec flag */
#define F_SETFL     3   /* set descriptor options */
#define F_GETFL     4   /* get descriptor options */
#define F_SETOWN    5   /* set descriptor owner (pid/pgrp) *
#define F_GETOWN    6   /* get descriptor owner (pid/pgrp) *
```

The `F_SETFL` *cmd* may be used to set a descriptor in non-blocking i/o mode and/or enable signalling when i/o is possible. `F_SETOWN` may be used to specify a process or process group to be signalled when using the latter mode of operation.

Operations on non-blocking descriptors will either complete immediately, note an error EWOULDBLOCK, partially complete an input or output operation returning a partial count, or return an error EINPROGRESS noting that the requested operation is in progress. A descriptor which has signalling enabled will cause the specified process and/or process group be signaled, with a SIGIO for input, output, or in-progress operation complete, or a SIGURG for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is ``in progress''. The select facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

7

Resource Controls

7

Resource Controls

7.1. Process Priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

```
#define PRIO_PROCESS     0    /* process */
#define PRIO_PGRP        1    /* process group */
#define PRIO_USER        2    /* user id */

prio = getpriority(which, who);
result int prio; int which, who;

setpriority(which, who, prio);
int which, who, prio;
```

The value *prio* is in the range –20 to 20. The default priority is 0; lower priorities cause more favorable execution. The `getpriority` call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The `setpriority` call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

7.2. Resource Utilization

The resources used by a process are returned by a `getrusage` call, returning information in a structure defined in `<sys/resource.h>`:

```
#define RUSAGE_SELF 0        /* usage by this process */
#define RUSAGE_CHILDREN -1       /* usage by all children */

getrusage(who, rusage);
int who; result struct rusage *rusage;

struct rusage {
    struct  timeval ru_utime;    /* user time used */
    struct  timeval ru_stime;    /* system time used */
    int ru_maxrss;  /* maximum core resident set size: kbyte
    int ru_ixrss;   /* integral shared memory size (kbytes*s
    int ru_idrss;   /* unshared data " */
    int ru_isrss;   /* unshared stack " */
    int ru_minflt;  /* page-reclaims */
    int ru_majflt;  /* page faults */
```

```
        int ru_nswap;    /* swaps */
        int ru_inblock; /* block input operations */
        int ru_oublock; /* block output " */
        int ru_msgsnd;  /* messages sent */
        int ru_msgrcv;  /* messages received */
        int ru_nsignals;    /* signals received */
        int ru_nvcsw;   /* voluntary context switches */
        int ru_nivcsw;  /* involuntary " */
};
```

The *who* parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

7.3. Resource Limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the getrlimit and setrlimit calls:

```
#define RLIMIT_CPU  0   /* cpu time in milliseconds */
#define RLIMIT_FSIZE    1   /* maximum file size */
#define RLIMIT_DATA 2   /* maximum data segment size */
#define RLIMIT_STACK    3   /* maximum stack segment size */
#define RLIMIT_CORE 4   /* maximum core file size */
#define RLIMIT_RSS  5   /* maximum resident set size */

#define RLIM_NLIMITS    6

#define RLIM_INFINITY   0x7fffffff

struct rlimit {
    int rlim_cur;   /* current (soft) limit */
    int rlim_max;   /* hard limit */
};

getrlimit(resource, rlp);
int resource; result struct rlimit *rlp;

setrlimit(resource, rlp);
int resource; struct rlimit *rlp;
```

Only the super-user can raise the maximum limits. Other users may only alter *rlim_cur* within the range from 0 to *rlim_max* or (irreversibly) lower *rlim_max*.

8

System Operation Support

8

System Operation Support

The calls in this section are permitted only to a privileged user.

8.1. Bootstrap Operations

The call

```
mount(type, dir, flags, data);
char *type, *dir;
int flags;
caddr_t data;
```

extends the UNIX name space. The `mount` call specifies a block device *type* containing a UNIX file system to be made available starting at *dir*. If *flags* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. *Data* is a pointer to a structure which contains the type specific arguments to mount.

The call

```
swapon(blkdev, size);
char *blkdev; int size;
```

specifies a device to be made available for paging and swapping.

8.2. Shutdown Operations

The call

```
unmount(dir);
char *dir;
```

unmounts the file system mounted on *dir*. This call will succeed only if the file system is not currently being used.

The call

```
fsync(fd);
int fd;
```

moves all modified data and attributes of the file referenced by *fd* to a permanent storage device. When the `fsync` call returns, all in-memory modified copies of buffers for the associated file have been written to disk. This call is different from the `sync` call below.

The call

```
sync();
```

schedules input/output to clean all system buffer caches.

The call

```
reboot(how);
int how;
```

halts or reboots a machine. The call may request a reboot by specifying *how* as `RB_AUTOBOOT`, or that the machine be halted with `RB_HALT`. These constants are defined in `<sys/reboot.h>`.

8.3. Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. The accounting may be enabled to a file *name* by doing

```
acct(path);
char *path;
```

If *path* is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

9

Generic Operations

9

Generic Operations

Many system abstractions support the operations `read`, `write` and `ioctl`. We describe the basics of these common primitives here. Similarly, the mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions and are described here.

9.1. Read and Write

The `read` and `write` system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
cc = read(fd, buf, nbytes);
result int cc; int fd; result caddr_t buf; int nbytes;

cc = write(fd, buf, nbytes);
result int cc; int fd; caddr_t buf; int nbytes;
```

The `read` call transfers as much data as possible from the object defined by *fd* to the buffer at address *buf* of size *nbytes*. The number of bytes transferred is returned in *cc*, which is –1 if a return occurred before any data was transferred because of an error or use of non-blocking operations.

The `write` call transfers data from the buffer to the object defined by *fd*. Depending on the type of *fd*, it is possible that the *write* call will accept some portion of the provided bytes; the user should resubmit the other bytes in a later request in this case. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in `<sys/uio.h>` as:

```
struct iovec {
    caddr_t iov_msg;    /* base of a component */
    int iov_len;    /* length of a component */
};
```

The calls using an array of descriptors are:

```
cc = readv(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;

cc = writev(fd, iov, iovlen);
result int cc; int fd; struct iovec *iov; int iovlen;
```

Here *iovlen* is the count of elements in the *iov* array.

9.2. Input/Output Control

Control operations on an object are performed by the `ioctl` operation:

```
ioctl(fd, request, buffer);
int fd, request; caddr_t buffer;
```

This operation causes the specified *request* to be performed on the object *fd*. The *request* parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct `ioctl` requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in `<sys/ioctl.h>`.

9.3. Non-Blocking and Asynchronous Operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section 6.4. Thereafter the `read` call will return a specific EWOULDBLOCK error indication if there is no data to be `read`. The process may `select` the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a `select` call indicates the object is writeable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot return immediately. The descriptor may then be `select`'ed for `write` to find out when the operation can be retried. When `select` indicates the descriptor is writeable, a respecification of the original operation will return the result of the operation.

10

File System

10 File System

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other subdirectories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

10.1. Naming

The file system calls take *path name* arguments. These consist of a zero or more component *file names* separated by / characters, where each file name is up to 255 ASCII characters excluding null and /.

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a path name begins with a /, it is called a full path name and interpreted relative to the root directory context. If the path name does not begin with a / it is called a relative path name and interpreted relative to the current directory context.

The system limits the total length of a path name to 1024 characters.

The file name `..` in each directory refers to the parent directory of that directory.

The calls

```
chdir(path);
char *path;

chroot(path);
char *path;
```

change the current working directory and root directory context of a process. Only the super-user can change the root directory context of a process.

10.2. Creation and Removal

The file system allows directories, files and special devices, to be created and removed from the file system.

Directory Creation and Removal

A directory is created with the `mkdir` system call:

```
mkdir(path, mode);
char *path; int mode;
```

and removed with the `rmdir` system call:

```
rmdir(path);
char *path;
```

A directory must be empty if it is to be deleted.

File Creation

Files are created with the `open` system call,

```
fd = open(path, oflag, mode);
result int fd; char *path; int oflag, mode;
```

The *path* parameter specifies the name of the file to be created. The *oflag* parameter must include `O_CREAT` from below to cause the file to be created. The protection for the new file is specified in *mode*. Bits for *oflag* are defined in `<sys/file.h>`:

```
#define O_RDONLY    000 /* open for reading */
#define O_WRONLY    001 /* open for writing */
#define O_RDWR  002 /* open for read & write */
#define O_NDELAY    004     /* non-blocking open */
#define O_APPEND    010 /* append on each write */
#define O_CREAT 01000   /* open with file create */
#define O_TRUNC 02000   /* open with truncation */
#define O_EXCL  04000   /* error on create if file exists */
```

One of `O_RDONLY`, `O_WRONLY` and `O_RDWR` should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the `open` to succeed. Specifying `O_APPEND` causes writes to automatically append to the file. The flag `O_CREAT` causes the file to be created if it does not exist, with the specified *mode*, owned by the current user and the group of the containing directory.

If the open specifies to create the file with `O_EXCL` and the file already exists, then the `open` will fail without affecting the file in any way. This provides a simple exclusive access facility.

Creating References to Devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their 'major' and 'minor' device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in 'block' quantities while unstructured devices often have a number of special `ioctl` operations, and may have input and output performed in large units. The mknod call creates special entries:

```
mknod(path, mode, dev);
char *path; int mode, dev;
```

where *mode* is formed from the object type and access permissions. The parameter *dev* is a configuration dependent parameter used to identify specific character or block i/o devices.

File and Device Removal

A reference to a file or special device may be removed with the unlink call,

```
unlink(path);
char *path;
```

The caller must have write access to the directory in which the file is located for this call to be successful.

10.3. Reading and Modifying File Attributes

Detailed information about the attributes of a file system may be obtained with the calls:

```
#include <sys/vfs.h>

statfs(path, buf);
char *path;
result struct statfs *buf;

fstatfs(fd, buf);
int fd;
result struct statfs *buf;
```

The statfs structure includes the file system type, file system block size, total blocks in the file system, free blocks, free blocks available to non superuser, total file nodes in the file system, free file nodes in the file system, and the file system ID.

Directory entries can be obtained in a filesystem-independent format by using the getdirentries call:

```
cc = getdirentries(d, buf, nbytes, basep);
result int cc;
int d;
char *buf;
int nbytes;
result long *basep;
```

Detailed information about the attributes of a file may be obtained with the calls:

```
#include <sys/stat.h>

stat(path, stb);
char *path; result struct stat *stb;

fstat(fd, stb);
int fd; result struct stat *stb;
```

The stat structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the `lstat` call:

```
lstat(path, stb);
char *path; result struct stat *stb;
```

Newly created files are assigned the user id of the process that created it and the group id of the directory in which it was created. The ownership of a file may be changed by either of the calls

```
chown(path, owner, group);
char *path; int owner, group;

fchown(fd, owner, group);
int fd, owner, group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
chmod(path, mode);
char *path; int mode;

fchmod(fd, mode);
int fd, mode;
```

where *mode* is a value indicating the new protection of the file. The file mode is a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

Three additional bits exist: the 04000 'set-user-id' bit can be set on an executable file to cause the effective user-id of a process which executes the file to be set to the owner of that file; the 02000 bit has a similar effect on the effective group-id. The 01000 bit causes an image of an executable program to be saved longer than would otherwise be normal; this 'sticky' bit is a hint to the system that a program is heavily used.

Finally, the access and modify times on a file may be set by the call:

```
utimes(path, tvp);
char *path; struct timeval *tvp[2];
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

10.4. Links and Renaming

Links allow multiple names for a file to exist. Links exist independently of the file linked to.

Two types of links exist, *hard* links and *symbolic* links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named *path2*, to *path1*:

```
link(path1, path2);
char *path1, *path2;

symlink(path1, path2);
char *path1, *path2;
```

The `unlink` primitive may be used to remove either type of link.

If a file is a symbolic link, the 'value' of the link may be read with the `readlink` call,

```
len = readlink(path, buf, bufsize);
result int len; result char *path, *buf; int bufsize;
```

This call returns, in *buf*, the null-terminated string substituted into pathnames passing through *path*.

Atomic renaming of file system resident objects is possible with the `rename` call:

```
rename(oldname, newname);
char *oldname, *newname;
```

where both *oldname* and *newname* must be in the same file system. If *newname* exists and is a directory, then it must be empty.

10.5. Extension and Truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about in the file in a random access fashion. To set the current offset into a file, the `lseek` call may be used,

```
oldoffset = lseek(fd, offset, type);
result off_t oldoffset; int fd; off_t offset; int type;
```

where *type* is given in `<sys/file.h>` as one of,

```
#define L_SET   0   /* set absolute file offset */
#define L_INCR  1   /* set file offset relative to current pc
#define L_XTND  2   /* set offset relative to end-of-file */
```

The call

```
lseek(fd, 0, L_INCR)
```

returns the current offset into the file.

Files may have 'holes' in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero valued bytes.

A file may be truncated with either of the calls:

```
truncate(path, length);
char *path;
off_t length;

ftruncate(fd, length);
int fd;
off_t length;
```

reducing the size of the specified file to *length* bytes.

10.6. Checking Accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the `access` call:

```
accessible = access(path, how);
result int accessible; char *path; int how;
```

Here *how* is constructed by or'ing the following bits, defined in `<sys/file.h>`:

```
#define F_OK    0   /* file exists */
#define X_OK    1   /* file is executable */
#define W_OK    2   /* file is writable */
#define R_OK    4   /* file is readable */
```

The presence or absence of advisory locks does not affect the result of `access`.

10.7. Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory `read` or `write` lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. More granular locking can be built using the IPC facilities to provide a lock manager. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking is performed after an `open` call by applying the `flock` primitive,

```
flock(fd, how);
int fd, how;
```

where the *how* parameter is formed from bits defined in `<sys/file.h>`:

```
#define LOCK_SH 1   /* shared lock */
#define LOCK_EX 2   /* exclusive lock */
#define LOCK_NB 4   /* don't block when locking */
#define LOCK_UN 8   /* unlock */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a `flock` call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including LOCK_NB in the *how* parameter. Specifying LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

10.8. Disk Quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To manipulate disk quotas on a file system the `quotactl` call is used:

```
#include <ufs/quota.h>

quotactl(cmd, special, uid, addr);
int cmd;
char *special;
int uid;
caddr_t addr;
```

where *cmd* indicates a command to be applied to the user ID *uid*. *Special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted. *Addr* is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of *addr* is given with each command.

11

Interprocess Communications

11

Interprocess Communications

11.1. Interprocess Communication Primitives

Communication Domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file `<sys/socket.h>`. Important standard domains supported by the system are the UNIX domain, `AF_UNIX`, for communication within the system, and the ''internet'' domain for communication in the DARPA internet, `AF_INET`. Other domains can be added to the system.

Socket Types and Protocols

Within a domain, communication takes place between communication endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in `<sys/socket.h>`:

```
/* Standard socket types */
#define SOCK_DGRAM  1   /* datagram */
#define SOCK_STREAM 2   /* virtual circuit */
#define SOCK_RAW    3   /* raw socket */
#define SOCK_RDM    4   /* reliably-delivered message */
#define SOCK_SEQPACKET  5   /* sequenced packets */
```

The `SOCK_DGRAM` type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. The `SOCK_RDM` type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The `send` and `receive` operations (described below) generate reliable/unreliable datagrams. The `SOCK_STREAM` type models connection-based virtual circuits: two-way byte streams with no record boundaries. The `SOCK_SEQPACKET` type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.[2]

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

Socket Creation, Naming, and Service Establishment

Sockets may be *connected* or *unconnected.* An unconnected socket descriptor is obtained by the socket call:

```
s = socket(domain, type, protocol);
result int s; int domain, type, protocol;
```

An unconnected socket descriptor may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket.

To accept connections, a socket must first have a binding to a name within the communications domain. Such a binding is established by a bind call:

```
bind(s, name, namelen);
int s; char *name; int namelen;
```

A socket's bound name may be retrieved with a getsockname call:

```
getsockname(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

while the peer's name can be retrieved with getpeername:

```
getpeername(s, name, namelen);
int s; result caddr_t name; result int *namelen;
```

Domains may support sockets with several names.

Accepting Connections

Once a binding is made, it is possible to listen for connections:

```
listen(s, backlog);
int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An accept call:

[2] This release does not support the SOCK_RDM and SOCK_SEQPACKET types.

```
t = accept(s, name, anamelen);
result int t;
int s;
result caddr_t name;
result int *anamelen;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*.

Making Connections

An active connection to a named socket is made by the `connect` call:

```
connect(s, name, namelen);
int s; caddr_t name; int namelen;
```

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the `socketpair` call[3]:

```
socketpair(d, type, protocol, sv);
int d, type, protocol; result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with `accept` and `connect`.

The call

```
pipe(pv);
result int pv[2];
```

creates a pair of `SOCK_STREAM` sockets in the UNIX domain, with `pv[0]` only writeable and `pv[1]` only readable.

Sending and Receiving Data

Messages may be sent from a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
result int cc;
int s;
caddr_t buf;
int len, flags;
caddr_t to;
int tolen;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
result int cc; int s; caddr_t buf; int len, flags;
```

if the socket is connected. The corresponding receive primitives are:

[3] This release supports `socketpair` creation only in the "unix" communication domain.

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
result int msglen;
int s;
result caddr_t buf;
int len, flags;
result caddr_t from; result int *fromlenaddr;
```

and

```
msglen = recv(s, buf, len, flags);
result int msglen;
int s;
result caddr_t buf;
int len, flags;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

```
#define MSG_PEEK    0x1 /* peek at incoming message */
#define MSG_OOB 0x2 /* process out-of-band data */
```

Scatter/Gather and Exchanging Access Rights

It is possible to scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in `<sys/socket.h>`, which is used to contain the parameters to the calls:

```
struct msghdr {
    caddr_t msg_name;       /* optional address */
    int msg_namelen;    /* size of address */
    struct  iov *msg_iov;   /* scatter/gather array */
    int msg_iovlen;     /* # elements in msg_iov */
    caddr_t msg_accrights;  /* access rights sent/received
    int msg_accrightslen;   /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section 9.1. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the "unix" domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations `sendmsg` and `recvmsg`:

```
sendmsg(s, msg, flags);
int s; struct msghdr *msg; int flags;

msglen = recvmsg(s, msg, flags);
result int msglen; int s; result struct msghdr *msg; int flag
```

Using Read and Write with Sockets

The normal UNIX `read` and `write` calls may be applied to connected sockets and translated into `send` and `receive` calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

Shutting Down Halves of Full-Duplex Connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down.

Socket and Protocol Options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or non-standard facilities. The `getsockopt` and `setsockopt` calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen);
int s, level, optname;
result caddr_t optval;
result int *optlen;

setsockopt(s, level, optname, optval, optlen);
int s, level, optname; caddr_t optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* `SOL_SOCKET` is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a ''protocol number''.

11.2. UNIX Domain

This section describes briefly the properties of the UNIX communications domain.

Types of Sockets

In the UNIX domain, the `SOCK_STREAM` abstraction provides pipe-like facilities, while `SOCK_DGRAM` provides datagrams — unreliable message-style communications.

Naming

Socket names are strings and the current implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this is a side effect of the implementation.

Access Rights Transmission

The ability to pass UNIX descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

11.3. INTERNET Domain

This section describes briefly how the INTERNET domain is mapped to the model described in this section. More information will be found in the *Networking Implementation Notes* in *Networking on the Sun Workstation*.

Socket Types and Protocols

`SOCK_STREAM` is supported by the INTERNET TCP protocol; `SOCK_DGRAM` by the UDP protocol. The `SOCK_SEQPACKET` has no direct INTERNET family analogue; a protocol based on one from the XEROX NS family and layered on top of IP could be implemented to fill this gap.

Socket Naming

Sockets in the INTERNET domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide source routing for the address, security options, or additional addresses for subnets of INTERNET for which the basic 32 bit addresses are insufficient.

Access Rights Transmission

No access rights transmission facilities are provided in the INTERNET domain.

Raw Access

The INTERNET domain allows the super-user access to the raw facilities of the various network interfaces and the various internal layers of the protocol implementation. This allows administrative and debugging functions to occur. These interfaces are modeled as `SOCK_RAW` sockets.

12

Devices

12 Devices

The system uses a collection of device-drivers to access attached peripherals. Such devices are grouped into two classes: structured devices on which block-oriented input/output operations occur, and unstructured devices (the rest).

12.1. Structured Devices

Structured devices include disk and tape drives, and are accessed through a system buffer-caching mechanism, which permits them to be accessed as ordinary files are, performing reads and writes as necessary to allow random-access.

The *mount* command in the system allows a structured device containing a file system volume to be accessed through the UNIX file system calls.

Tape drives also typically provide a structured interface, although this is rarely used.

12.2. Unstructured Devices

Unstructured devices are those devices which do not support a randomly accessed block structure.

Communications lines, raster plotters, normal magnetic tape access (in large or variable size blocks), and access to disk drives permitting large block transfers and special operations like disk formatting and labelling all use unstructured device interfaces.

The writing of devices for unstructured devices other than communications lines is described in the *Device Driver Manual* in the System Internals Manual.

13

Debugging Support

13

Debugging Support

13.1. ptrace — Process Tracing

ptrace provides a means by which a process may control the execution of another process, and examine and change its memory image. Its primary use is for the implementation of breakpoint debugging[4].

```
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

ptrace(request, pid, addr, data, addr2)
enum ptracereq request;
int pid;
char *addr;
int data;
char *addr2;
```

There are five arguments whose interpretation depends on the *request* argument. Generally, *pid* is the process ID of the traced process. A process being traced behaves normally until it encounters some signal whether internally generated like 'illegal instruction' or externally generated like 'interrupt'. See sigvec(2) for the list. Then the traced process enters a stopped state and the tracing process is notified via wait(2). When the traced process is in the stopped state, its memory image can be examined and modified using ptrace. If desired, another ptrace request can then cause the traced process either to terminate or to continue, possibly ignoring the signal.

Note that several different values of the *request* argument can make ptrace return data values — since −1 is a possibly legitimate value, to differentiate between −1 as a legitimate value and −1 as an error code, you should clear the *errno* global error code before doing a ptrace call, and then check the value of *errno* afterwards.

The value of the *request* argument determines the precise action of the call:

PTRACE_TRACEME
: This request is the only one used by the traced process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

[4] Enhancements which would allow a descriptor-based process control facility have not been implemented.

PTRACE_PEEKTEXT, PTRACE_PEEKDATA
The word in the traced process's address space at *addr* is returned. If the instruction and data spaces are separate (for example, historically on a PDP-11), request PTRACE_PEEKTEXT indicates instruction space while PTRACE_PEEKDATA indicates data space. *Addr* must be even, the child must be stopped and the input *data* and *addr2* are ignored.

PTRACE_PEEKUSER
The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be a valid offset within the kernel's per-process data pages. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

PTRACE_POKETEXT, PTRACE_POKEDATA
The given *data* is written at the word in the process's address space corresponding to addr, which must be even. No useful value is returned. If the instruction and data spaces are separate request PTRACE_PEEKTEXT indicates instruction space while PTRACE_PEEKDATA indicates data space. The PTRACE_POKETEXT request must be used to write into a process's text space even if the instruction and data spaces are not separate. Attempts to write in a pure text space fail if another process is executing the same file.

PTRACE_POKEUSER
The process's system data is written, as it is read with request PTRACE_PEEKUSER. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

PTRACE_CONT
The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

PTRACE_KILL
The traced process terminates.

PTRACE_SINGLESTEP
Execution continues as in request PTRACE_CONT; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. On the Sun and VAX-11 the T-bit is used and just one instruction is executed. This is part of the mechanism for implementing breakpoints.

PTRACE_ATTACH
Attach to the process identified by the *pid* argument and begin tracing it. Process *pid* does not have to be a child of the requestor, but the requestor must have permission to send process *pid* a signal and the effective userids of the requesting process and process *pid* must match.

`PTRACE_DETACH`
: Detach the process being traced. Process *pid* is no longer being traced and continues its execution. The *data* argument is taken as a signal number and the process continues at location *addr* as if it had incurred that signal.

`PTRACE_GETREGS`
: The traced process's registers are returned in a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in `<machine/reg.h>` describes the data that is returned.

`PTRACE_SETREGS`
: The traced process's registers are written from a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in `<machine/reg.h>` describes the data that is set.

PTRACE_READTEXT, `PTRACE_READDATA`
: Read data from the address space of the traced process. If the instruction and data spaces are separate, request `PTRACE_READTEXT` indicates instruction space while `PTRACE_READDATA` indicates data space. The *addr* argument is the address within the traced process from where the data is read, the *data* argument is the number of bytes to read, and the *addr2* argument is the address within the requesting process where the data is written.

PTRACE_WRITETEXT, `PTRACE_WRITEDATA`
: Write data into the address space of the traced process. If the instruction and data spaces are separate, request `PTRACE_READTEXT` indicates instruction space while `PTRACE_READDATA` indicates data space. The *addr* argument is the address within the traced process where the data is written, the *data* argument is the number of bytes to write, and the *addr2* argument is the address within the requesting process from where the data is read.

As indicated, these calls (except for requests `PTRACE_TRACEME` and `PTRACE_ATTACH)` can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the 'termination' status returned by *wait* has the value `WSTOPPED` to indicate a stop rather than genuine termination.

To forestall possible fraud, `ptrace` inhibits the set-user-id and set-group-id facilities on subsequent `execve(2)` calls. If a traced process calls `execve`, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

On the Sun and VAX-11, 'word' also means a 32-bit integer.

A

Summary of Facilities

A

Summary of Facilities

A.1. Kernel Primitives

Process Naming and Protection

`gethostid`	get UNIX host id
`sethostname`	set UNIX host name
`gethostname`	get UNIX host name
`getpid`	get process id
`fork`	create new process
`exit`	terminate a process
`execve`	execute a different process
`getuid`	get user id
`geteuid`	get effective user id
`setreuid`	set real and effective user id's
`getgid`	get accounting group id
`getegid`	get effective accounting group id
`getgroups`	get access group set
`setregid`	set real and effective group id's
`setgroups`	set access group set
`getpgrp`	get process group
`setpgrp`	set process group

Memory Management

`<mman.h>`	memory management definitions
`sbrk`	change data section size
`getpagesize`	get memory page size
`mmap`†	map pages of memory
`mremap`†	remap pages in memory
`munmap`†	unmap memory

Signals

`<signal.h>`	signal definitions
`sigvec`	set handler for signal
`kill`	send signal to process
`killpgrp`	send signal to process group
`sigblock`	block set of signals
`sigsetmask`	restore set of blocked signals

[4] † These calls are supported in limited form in the 3.0 Sun release.

`sigpause`	wait for signals
`sigstack`	set software stack for signals

Timing and Statistics

`<sys/time.h>`	time-related definitions
`gettimeofday`	get current time and timezone
`settimeofday`	set current time and timezone
`getitimer`	read an interval timer
`setitimer`	get and set an interval timer
`profil`	profile process

Descriptors

`getdtablesize`	descriptor reference table size
`dup`	duplicate descriptor
`dup2`	duplicate to specified index
`close`	close descriptor
`select`	multiplex input/output
`fcntl`	control descriptor options

Resource Controls

`<sys/resource.h>`	resource-related definitions
`getpriority`	get process priority
`setpriority`	set process priority
`getrusage`	get resource usage
`getrlimit`	get resource limitations
`setrlimit`	set resource limitations

System Operation Support

`mount`	mount a device file system
`swapon`	add a swap device
`unmount`	umount a file system
`sync`	flush system caches
`reboot`	reboot a machine
`acct`	specify accounting file

A.2. System Facilities

Generic Operations

`read`	read data
`write`	write data
`<sys/uio.h>`	scatter-gather related definitions
`readv`	scattered data input
`writev`	gathered data output
`<sys/ioctl.h>`	standard control operations
`ioctl`	device control operation

[4] † Not supported in the 1.0 Sun release.

File System

Operations marked with a * exist in two forms: as shown, operating on a file name, and operating on a file descriptor, when the name is preceded with a ''f''.

`<sys/file.h>`	file system definitions
`chdir`	change directory
`chroot`	change root directory
`mkdir`	make a directory
`rmdir`	remove a directory
`open`	open a new or existing file
`mknod`	make a special file
`unlink`	remove a link
`stat*`	return status for a file
`lstat`	returned status of link
`chown*`	change owner
`chmod*`	change mode
`utimes`	change access/modify times
`link`	make a hard link
`symlink`	make a symbolic link
`readlink`	read contents of symbolic link
`rename`	change name of file
`lseek`	reposition within file
`truncate*`	truncate file
`access`	determine accessibility
`flock`	lock a file

Interprocess Communications

`<sys/socket.h>`	standard definitions
`socket`	create socket
`bind`	bind socket to name
`getsockname`	get socket name
`listen`	allow queueing of connections
`accept`	accept a connection
`connect`	connect to peer socket
`socketpair`	create pair of connected sockets
`sendto`	send data to named socket
`send`	send data to connected socket
`recvfrom`	receive data on unconnected socket
`recv`	receive data on connected socket
`sendmsg`	send gathered data and/or rights
`recvmsg`	receive scattered data and/or rights
`shutdown`	partially close full-duplex connection
`getsockopt`	get socket option
`setsockopt`	set socket option

Debugging Support

`ptrace`	trace process

Index

Revision History

Revision	*Date*	*Comments*
A	15 February 1986	First Release of this manual in this form. Extracted this manual from out of the *System Interface Manual* to form a stand-alone document.

Notes

Notes

Notes

Notes

Notes

Notes

Notes