Berkeley Software Architecture Manual
4.4BSD Edition

M. Kirk McKusick, Michael Karels

Samuel Leffler, William Joy

Robert Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

This document summarizes the system calls provided by the 4.4BSD operating system. 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.

The first section describes the basic kernel functions provided to a process: process naming and protection, memory management, software interrupts, time and statistics functions, object references (descriptors), and resource controls. These facilities, as well as facilities for bootstrap, shutdown and process accounting, are provided solely by the kernel.

The second section describes the standard system abstractions for files and filesystems, communication, terminal handling, and process control and debugging. These facilities are implemented by the operating system or by network server processes.

Notation and Types

The notation used to describe system calls is a variant of a C language function call, consisting of a prototype call followed by the 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 2.1.1:

cc = read(fd, buf, nbytes);
result ssize_t cc; int fd; result void *buf; size_t nbytes;

The first line shows how the *read* routine is called, with three parameters. As shown on the second line, the return value *cc* is a size_t and *read* also returns information in the parameter *buf*.

The descriptions of error conditions arising from each system call are not provided here; they appear in section 2 of the Programmer's Reference 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*. Other languages may present errors in different ways.

A number of system standard types are defined by the include file <*sys/types.h*> and used in the specifications here and in many C programs.

Type	Value	
caddr_t	char *	/* a memory address */
clock_t	unsigned long	/* count of CLK_TCK's */
gid_t	unsigned long	/* group ID */
int16_t	short	/* 16-bit integer */
int32_t	int	/* 32-bit integer */
int64_t	long long	/* 64-bit integer */
int8_t	signed char	/* 8-bit integer */
mode_t	unsigned short	/* file permissions */
off_t	quad_t	/* file offset */
pid_t	long	/* process ID */
qaddr_t	quad_t *	
quad_t	long long	
size_t	unsigned int	/* count of bytes */
ssize_t	int	/* signed size_t */
time_t	long	/* seconds since the Epoch */
u_char	unsigned char	
u_int	unsigned int	
u_int16_t	unsigned short	/* unsigned 16-bit integer */
u_int32_t	unsigned int	/* unsigned 32-bit integer */
u_int64_t	unsigned long long	/* unsigned 64-bit integer */
u_int8_t	unsigned char	/* unsigned 8-bit integer */
u_long	unsigned long	
u_quad_t	unsigned long long	
u_short	unsigned short	
uid_t	unsigned long	/* user ID */
uint	unsigned int	/* System V compatibility */
ushort	unsigned short	/* System V compatibility */

1. Kernel primitives

The facilities available to a user process are logically divided into two parts: kernel facilities directly implemented by code running in the operating system, and system facilities implemented either by the system, or in cooperation with a *server process*. The kernel facilities are described in section 1.

The facilities implemented in the kernel are those which define the *4.4BSD virtual machine* in which each process runs. Like many real machines, this virtual machine has memory management hardware, an interrupt facility, timers and counters. The 4.4BSD virtual machine 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 implemented in server processes on other machines. The facilities provided through the descriptor machinery are described in section 2.

1.1. Processes and protection

1.1.1. Host identifiers

Each host has associated with it an integer host ID, and a host name of up to MAXHOSTNAMELEN (256) characters (as defined in <*sys/param.h*>). These identifiers are set (by a privileged user) and retrieved using the *sysctl* interface described in section 1.7.1. The host ID is seldom used (or set), and is deprecated. For convenience and backward compatibility, the following library routines are provided:

```
sethostid(hostid);
long hostid;

hostid = gethostid();
result long hostid;

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

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

1.1.2. Process identifiers

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 pid_t pid;
```

On each host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, process ID) pairs are guaranteed unique. The parent process identifier can be obtained using the *getppid* routine:

```
pid = getppid();
result pid_t pid;
```

1.1.3. Process creation and termination

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

```
pid = fork();
result pid_t 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 imposes a hierarchical structure on the set of processes in the system.

For processes that are forking solely for the purpose of *execve*'ing another program, the *vfork* system call provides a faster interface:

```
pid = vfork();
result pid_t pid;
```

Like *fork*, the *vfork* 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 process is suspended until the child process calls either *execve* or *exit*.

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

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

The lower 8 bits of exit status are available to its parent.

When a child process exits or terminates abnormally, the parent process receives information about the event which caused termination of the child process. The interface allows the parent to wait for a particular process, process group, or any direct descendent and to retrieve information about resources consumed by the process during its lifetime. The request may be done either synchronously (waiting for one of the requested processes to exit), or asynchronously (polling to see if any of the requested processes have exited):

```
pid = wait4(wpid, astatus, options, arusage);
result pid_t pid; pid_t wpid; result int *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. If the set-user-ID mode bit is set, the effective user ID is set to the owner of the file; if the set-group-ID mode bit is set, the effective group ID is set to the group of the file. Whether changed or not, the effective user ID is then copied to the saved user ID, and the effective group ID is copied to the saved group ID.

1.1.4. User and group IDs

Each process in the system has associated with it three user IDs: a *real user ID*, an *effective user ID*, and a *saved user ID*, all unsigned integral types (**uid_t**). Each process has a *real group ID* and a set of *access group IDs*, the first of which is the *effective group ID*. The group IDs are unsigned integral types (**gid_t**). Each process may be in multiple access groups. The maximum concurrent number of access groups is a system compilation parameter, represented by the constant NGROUPS in the file *<sys/param.h>*. It is guaranteed to be at least 16.

The real group ID is used in process accounting and in testing whether the effective group ID may be changed; it is not otherwise used for access control. The members of the access group ID set are used for access control. Because the first member of the set is the effective group ID, which is changed when executing a set-group-ID program, that element is normally duplicated in the set so that access privileges for the original group are not lost when using a set-group-ID program.

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

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

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

the real and effective group IDs by:

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

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

The access group ID set is returned by a *getgroups* call:

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

The user and group IDs are assigned at login time using the *setuid*, *setgid*, and *setgroups* calls:

```
setuid(uid);
uid_t uid;

setgid(gid);
gid_t gid;

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

The *setuid* call sets the real, effective, and saved user IDs, and is permitted only if the specified *uid* is the current real user ID or if the caller is the super-user. The *setgid* call sets the real, effective, and saved group IDs; it is permitted only if the specified *gid* is the current real group ID or if the caller is the super-user. The *setgroups* call sets the access group ID set, and is restricted to the super-user.

The *seteuid* routine allows any process to set its effective user ID to either its real or saved user ID:

```
seteuid(uid);
uid_t uid;
```

The *setegid* routine allows any process to set its effective group ID to either its real or saved group ID:

```
setegid(gid);
gid_t gid;
```

1.1.5. Sessions

When a user first logs onto the system, they are put into a session with a controlling process (usually a shell). The session is created with the call:

```
pid = setsid();
result pid_t pid;
```

All subsequent processes created by the user (that do not call *setsid*) will be part of the session. The session also has a login name associated with it which is set using the privileged call:

```
setlogin(name);
char *name;
```

The login name can be retrieved using the call:

```
name = getlogin();
result char *name;
```

Unlike historic systems, the value returned by *getlogin* is stored in the kernel and can be trusted.

1.1.6. Process groups

Each process in the system is also 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). All members of a process group are members of the same session. The current process group of a process is returned by the *getpgrp* call:

```
pgrp = getpgrp();
result pid_t pgrp;
```

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, a system terminal has a process group and only processes which are in the process group of the terminal may read from the terminal, allowing arbitration of a terminal among several different jobs.

The process group associated with a process may be changed by the *setpgid* call:

```
setpgid(pid, pgrp);
pid_t pid, pgrp;
```

Newly created processes are assigned process IDs distinct from all processes and process groups, and the same process group as their parent. Any process may set its process group equal to its process ID or to the value of any process group within its session.

1.2. Memory management

1.2.1. 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 writable and private to the process. Both the data and stack areas may be extended and contracted on program request. The call:

```
brk(addr);
caddr_t addr;
```

sets the end of the data segment to the specified address. More conveniently, the end can be extended by *incr* bytes, and the base of the new area returned with the call:

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

Application programs normally use the library routines *malloc* and *free*, which provide a more convenient interface than *brk* and *sbrk*.

There is no call for extending the stack, as it is automatically extended as needed.

1.2.2. 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 <*sys/mman.h*> as:

Protections are chosen from these bits, or-ed together:

```
PROT_READ     /* pages can be read */
PROT_WRITE    /* pages can be written */
PROT_EXEC     /* pages can be executed */
```

Flags contain sharing type and options. Sharing options, choose one:

```
MAP_SHARED      /* share changes */
MAP_PRIVATE     /* changes are private */
```

Option flags†:

```
MAP_ANON             /* allocated from virtual memory; fd ignored */
MAP_FIXED            /* map addr must be exactly as requested */
MAP_NORESERVE        /* don't reserve needed swap area */
MAP_INHERIT          /* region is retained after exec */
MAP_HASSEMAPHORE     /* region may contain semaphores */
```

The size of a page is cpu-dependent, and is returned by the *sysctl* interface described in section 1.7.1. The *getpagesize* library routine is provided for convenience and backward compatibility:

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

The call:

```
maddr = mmap(addr, len, prot, flags, fd, pos);
result caddr_t maddr; caddr_t addr; size_t len; int prot, flags, fd; off_t pos;
```

causes the pages starting at *addr* and continuing for at most *len* bytes to be mapped from the object represented by descriptor *fd*, starting at byte offset *pos*. If *addr* is NULL, the system picks an unused address for the region. The starting address of the region is returned; for the convenience of the system, it may differ from that supplied unless the MAP_FIXED flag is given, in which case the exact address will be used or the call will fail. The *addr* parameter must be a multiple of the pagesize (if MAP_FIXED is given). If *pos* and *len* are not a multiple of pagesize, they will be rounded (down and up respectively) to a page boundary by the system; the rounding will cause the mapped region to extend past the specified range. A successful *mmap* will delete any previous mapping in the allocated address range. The parameter *prot* specifies the accessibility of the mapped pages. The parameter *flags* specifies the type of object to be mapped, mapping options, and whether modifications made to this mapped copy of the page are to be kept *private*, or are to be *shared* with other references. Possible types include MAP_SHARED or MAP_PRIVATE that map a regular file or character-special device memory, and MAP_ANON, which maps memory not associated with any specific file. The file descriptor used when creating MAP_ANON regions is not used and should be −1. The MAP_INHERIT flag allows a region to be inherited after an *execve*. The MAP_HASSEMAPHORE flag allows special handling for regions that may contain semaphores. The MAP_NORESERVE flag allows processes to allocate regions whose virtual address space, if fully allocated, would exceed the available memory plus swap resources. Such regions may get a SIGSEGV signal if they page fault and resources are not available to service their request; typically they would free up some resources via *munmap* so that when they return from the signal the page fault could be completed successfully.

A facility is provided to synchronize a mapped region with the file it maps; the call:

```
msync(addr, len);
caddr_t addr; size_t len;
```

causes any modified pages in the specified region to be synchronized with their source and other mappings. If necessary, it writes any modified pages back to the filesystem, and updates the file modification time. If *len* is 0, all modified pages within the region containing *addr* will be flushed; this usage is provisional, and

† In 4.4BSD, only MAP_ANON and MAP_FIXED are implemented.

may be withdrawn. If *len* is non-zero, only the pages containing *addr* and *len* succeeding locations will be examined. Any required synchronization of memory caches will also take place at this time.

Filesystem operations on a file that is mapped for shared modifications are currently unpredictable except after an *msync*.

A mapping can be removed by the call

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

This call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

1.2.3. Page protection control

A process can control the protection of pages using the call:

```
mprotect(addr, len, prot);
caddr_t addr; size_t len; int prot;
```

This call changes the specified pages to have protection *prot*. Not all implementations will guarantee protection on a page basis; the granularity of protection changes may be as large as an entire region.

1.2.4. Giving and getting advice

A process that has knowledge of its memory behavior may use the *madvise*† call:

```
madvise(addr, len, behav);
caddr_t addr; size_t len; int behav;
```

Behav describes expected behavior, as given in <*sys/mman.h*>:

```
MADV_NORMAL      /* no further special treatment */
MADV_RANDOM      /* expect random page references */
MADV_SEQUENTIAL  /* expect sequential references */
MADV_WILLNEED    /* will need these pages */
MADV_DONTNEED    /* don't need these pages */
```

The *mincore*† function allows a process to obtain information about whether pages are memory resident:

```
mincore(addr, len, vec);
caddr_t addr; size_t len; result char *vec;
```

Here the current memory residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-memory. *Mincore* provides only transient information about page residency. Real-time processes that need guaranteed residence over time can use the call:

```
mlock(addr, len);
caddr_t addr; size_t len;
```

This call locks the pages for the specified address range into memory (paging them in if necessary) ensuring that further references to addresses within the range will never generate page faults. The amount of memory that may be locked is controlled by a resource limit, see section 1.6.3. When the memory is no longer critical it can be unlocked using:

```
munlock(addr, len);
caddr_t addr; size_t len;
```

† The entry point for this system call is defined, but is not implemented, so currently always returns with the error ‘‘Operation not supported.’’

After the *munlock* call, the pages in the specified address range are still accessible but may be paged out if memory is needed and they are not accessed.

1.2.5. Synchronization primitives

Primitives are provided for synchronization using semaphores in shared memory.‡ These primitives are expected to be superseded by the semaphore interface being specified by the POSIX 1003 Pthread standard. They are provided as an efficient interim solution. Application programmers are encouraged to use the Pthread interface when it becomes available.

Semaphores must lie within a MAP_SHARED region with at least modes PROT_READ and PROT_WRITE. The MAP_HASSEMAPHORE flag must have been specified when the region was created. To acquire a lock a process calls:

```
value = mset(sem, wait);
result int value; semaphore *sem; int wait;
```

Mset indivisibly tests and sets the semaphore *sem*. If the previous value is zero, the process has acquired the lock and *mset* returns true immediately. Otherwise, if the *wait* flag is zero, failure is returned. If *wait* is true and the previous value is non-zero, *mset* relinquishes the processor until notified that it should retry.

To release a lock a process calls:

```
mclear(sem);
semaphore *sem;
```

Mclear indivisibly tests and clears the semaphore *sem*. If the ‘‘WANT’’ flag is zero in the previous value, *mclear* returns immediately. If the ‘‘WANT’’ flag is non-zero in the previous value, *mclear* arranges for waiting processes to retry before returning.

Two routines provide services analogous to the kernel *sleep* and *wakeup* functions interpreted in the domain of shared memory. A process may relinquish the processor by calling *msleep* with a set semaphore:

```
msleep(sem);
semaphore *sem;
```

If the semaphore is still set when it is checked by the kernel, the process will be put in a sleeping state until some other process issues an *mwakeup* for the same semaphore within the region using the call:

```
mwakeup(sem);
semaphore *sem;
```

An *mwakeup* may awaken all sleepers on the semaphore, or may awaken only the next sleeper on a queue.

1.3. Signals

1.3.1. Overview

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 a *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 if 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 also choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

‡ All currently unimplemented, no entry points exists.

All signals have the same *priority*. If multiple signals are pending, signals that may be generated by the program's action are delivered first; the order in which other signals are delivered to a process is not specified. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may occur. Multiple signals may be delivered on a single entry to the system, as if signal handlers were interrupted by other signal handlers. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

1.3.2. 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 by 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 attempts to access addresses outside the currently assigned area of memory, and SIGBUS for accesses that violate memory access constraints.

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, which 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. Programs may define their own asynchronous events using SIGUSR1 and SIGUSR2. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers. When a window changes size, a SIGWINCH is sent to the controlling terminal process group.

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 for stopping the process. 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 when a process reaches the limit on file size.

1.3.3. Signal handlers

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

```
struct sigaction {
        void        (*sa_handler)();
        sigset_t    sa_mask;
        int         sa_flags;
};

sigaction(signo, sa, osa);
int signo; struct sigaction *sa; result struct sigaction *osa;
```

assigns interrupt handler address *sa_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 *sa_handler* cause ignoring or defaulting of a condition, respectively. The *sa_mask* value specifies the signal mask to be used when the handler is invoked; it implicitly includes the signal which invoked the handler. Signal masks include one bit for each signal. The following macros, defined in *signal.h*, create an empty mask, then put *signo* into it:

```
sigemptyset(set);
sigaddset(set, signo);
result sigset_t *set; int signo;
```

Sa_flags specifies whether pending system calls should be restarted if the signal handler returns (SA_RESTART) and whether the handler should operate on the normal run-time stack or a special signal stack (SA_ONSTACK; see below). If *osa* is non-zero, the previous signal handler information 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 it then will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's *sa_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 returns normally, the signal mask will be restored and the process will resume execution in the original context.

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

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

```
(*sa_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 signal-specific 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. Normally this context will be restored when the signal handler returns. However, a process may do so at any time by using the call:

```
sigreturn(scp);
struct sigcontext *scp;
```

If the signal handler makes a call to *longjmp*, the signal mask at the time of the corresponding *setjmp* is restored.

1.3.4. Sending signals

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

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

For compatibility with old systems, a compatibility routine is provided to send a signal to a process group:

```
killpg(pgrp, signo)
pid_t pgrp; int signo;
```

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

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

1.3.5. Protecting critical sections

The *sigprocmask* system call is used to manipulate the mask of blocked signals:

```
sigprocmask(how, newmask, oldmask);
int how; sigset_t *newmask; result sigset_t *oldmask;
```

The actions done by *sigprocmask* are to add to the list of masked signals (SIG_BLOCK), delete from the list of masked signals (SIG_UNBLOCK), and block a specific set of signals (SIG_SETMASK). The *sigprocmask* call can be used to read the current mask by specifying SIG_BLOCK with an empty *newmask*.

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

```
sigsuspend(mask);
sigset_t *mask;
```

It is also possible to find out which blocked signals are pending delivery using the call:

```
sigpending(mask);
result sigset_t *mask;
```

1.3.6. Signal stacks

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

```
struct sigaltstack {
	caddr_t	ss_sp;
	long	ss_size;
	int	ss_flags;
};

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

to provide the system with a stack based at *ss_sp* of size *ss_size* for delivery of signals. The value *ss_flags* 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 to a handler for which the SA_ONSTACK flag was set, 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 doing a *sigreturn* to restore the previous stack. If the process takes a non-local exit from the signal routine, *longjmp* will do a *sigreturn* call to switch back to the run-time stack.

1.4. Timers

1.4.1. Real time

The system's notion of the current time is in Coordinated Universal Time (UTC, previously GMT) and the current time zone is set and returned by the calls:

```
settimeofday(tp, 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 */
};
```

The timezone information is present only for historical reasons and is unused by the current system.

The precision of the system clock is hardware dependent. Earlier versions of UNIX contained only a 1-second resolution version of this call, which remains as a library routine:

```
time(tvsec);
result time_t *tvsec;
```

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

The *adjtime* system calls allows for small changes in time without abrupt changes by skewing the rate at which time advances:

```
adjtime(delta, olddelta);
struct timeval *delta; result struct timeval *olddelta;
```

1.4.2. Interval time

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

```
ITIMER_REAL       /* real time intervals */
ITIMER_VIRTUAL    /* virtual time intervals */
ITIMER_PROF       /* 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:

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

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

The *it_value* specifies the time until the next signal; the *it_interval* specifies a new interval that should be loaded into the timer on each expiration. The third argument to *setitimer* specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by setting *it_value* and *it_interval* to 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. The *profil* call arranges for the kernel to begin gathering execution statistics for a process:

```
profil(samples, size, offset, scale);
result char *samples; int size, offset, scale;
```

This call begins sampling the program counter, with statistics maintained in the user-provided buffer.

1.5. Descriptors

1.5.1. The reference table

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

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 therefore may have multiple references, and are reference counted by the system.

Each process has a limited size descriptor reference table, where the current size is returned by the *getdtablesize* call:

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

and guaranteed to be at least 64. The maximum number of descriptors is a resource limit (see section 1.6.3). The entries in the descriptor reference table are referred to by small integers; for example if there are 64 slots they are numbered 0 to 63.

1.5.2. 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. For those types that support random access, the current file offset is stored in the descriptor. The generic operations applying to many of these types are described in section 2.1. Naming contexts, files and directories are described in section 2.2. Section 2.3 describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in section 2.4.

1.5.3. 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* which is indistinguishable from the original. The value of *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.

Descriptors are deallocated by:

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

1.5.4. 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 to examine the state of multiple descriptors simultaneously, and to wait for state changes on those descriptors. Sets of descriptors of interest are specified as bit masks, as follows:

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

FD_CLR(fd, &fdset);
FD_COPY(&fdset, &fdset2);
FD_ISSET(fd, &fdset);
FD_SET(fd, &fdset);
FD_ZERO(&fdset);
int fd; fs_set fdset, fdset2;
```

The *select* call examines the descriptors specified by the sets *in*, *out* and *except*, replacing the specified bit masks by the subsets that select true for input, output, and exceptional conditions respectively (*nd* indicates the number of file descriptors specified by 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 sections 2.1.3 and 2.3.1.4).
- 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 sections 2.1.3 and 2.3.1.5).
- A descriptor selects for an exceptional condition if a condition that would cause a SIGURG signal to be generated exists (see section 1.3.2), or other device-specific events have occurred.

For these tests, an operation is considered to be possible if a call to the operation would return without blocking (even if the O_NONBLOCK flag were not set). For example, a descriptor would test as ready for reading if a read call would return immediately with data, an end-of-file indication, or an error other than EWOULDBLOCK.

If none of the specified conditions is true, the operation waits for one of the conditions to arise, blocking at most the amount of time specified by *tvp*. If *tvp* is given as NULL, the *select* waits indefinitely.

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;
```

```
/* command values */

F_DUPFD     /* return a new descriptor */
F_GETFD     /* get file descriptor flags */
F_SETFD     /* set file descriptor flags */
F_GETFL     /* get file status flags */
F_SETFL     /* set file status flags */
F_GETOWN    /* get SIGIO/SIGURG proc/pgrp */
F_SETOWN    /* set SIGIO/SIGURG proc/pgrp */
F_GETLK     /* get blocking lock */
F_SETLK     /* set or clear lock */
F_SETLKW    /* set lock with wait */
```

The F_DUPFD *cmd* provides identical functionality to *dup2*; it is provided solely for POSIX compatibility. The F_SETFD *cmd* can be used to set the close-on-exec flag for a file descriptor. The F_SETFL *cmd* may be used to set a descriptor in non-blocking I/O mode and/or enable signaling when I/O is possible. F_SETOWN may be used to specify a process or process group to be signaled when using the latter mode of operation or when urgent indications arise. The *fcntl* system call also provides POSIX-compliant byte-range locking on files. However the semantics of unlocking on every *close* rather than last close makes them useless. Much better semantics and faster locking are provided by the *flock* system call (see section 2.2.7). The *fcntl* and *flock* locks can be used concurrently; they will serialize against each other properly.

Operations on non-blocking descriptors will either complete immediately, return the 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, then 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.

1.6. Resource controls

1.6.1. 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. The instantaneous scheduling priority is a function of CPU usage and a settable priority value used in adjusting the instantaneous priority with CPU usage or inactivity. It is possible to determine the settable priority factor currently assigned to a process (PRIO_PROCESS), process group (PRIO_PGRP), or the processes of a specified user (PRIO_USER), or to alter this priority using the calls:

```
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 the specified processes to the specified value. Only the super-user may lower priorities.

1.6.2. Resource utilization

The *getrusage* call returns information describing the resources utilized by the current process (RUSAGE_SELF), or all its terminated descendent processes (RUSAGE_CHILDREN):

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

The information is returned in a structure defined in <*sys/resource.h*>:

```
struct rusage {
	struct	timeval ru_utime;	/* user time used */
	struct	timeval ru_stime;	/* system time used */
	int	ru_maxrss;	/* maximum core resident set size: kbytes */
	int	ru_ixrss;	/* integral shared memory size (kbytes*sec) */
	int	ru_idrss;	/* unshared data memory size */
	int	ru_isrss;	/* unshared stack memory size */
	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 operations */
	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 context switches */
};
```

1.6.3. 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:

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

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

The resources that may currently be controlled include:

```
RLIMIT_CPU	/* cpu time in milliseconds */
RLIMIT_FSIZE	/* maximum file size */
RLIMIT_DATA	/* data size */
RLIMIT_STACK	/* stack size */
RLIMIT_CORE	/* core file size */
RLIMIT_RSS	/* resident set size */
RLIMIT_MEMLOCK	/* locked-in-memory address space */
RLIMIT_NPROC	/* number of processes */
RLIMIT_NOFILE	/* number of open files */
```

Each limit has a current value and a maximum defined by the *rlimit* structure:

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

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*. To remove a limit on a resource, the value is set to RLIM_INFINITY.

1.7. System operation support

Unless noted otherwise, the calls in this section are permitted only to a privileged user.

1.7.1. Monitoring system operation

The *sysctl* function allows any process to retrieve system information and allows processes with appropriate privileges to set system configurations.

```
sysctl(name, namelen, oldp, oldlenp, newp, newlen);
int *name; u_int namelen; result void *oldp; result size_t *oldlenp;
void *newp; size_t newlen;
```

The information available from *sysctl* consists of integers, strings, and tables. *Sysctl* returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to *sysctl* are serialized to avoid deadlock.

The object to be interrogated or set is named using a ‘‘Management Information Base’’ (MIB) style name, listed in *name*, which is a *namelen* length array of integers. This name is from a hierarchical name space, with the most significant component in the first element of the array. It is analogous to a file pathname, but with integers as components rather than slash-separated strings.

The information is copied into the buffer specified by *oldp*. The size of the buffer is given by the location specified by *oldlenp* before the call, and that location is filled in with the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns an error.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to NULL and *newlen* set to 0.

The top level names (those used in the first element of the *name* array) are defined with a CTL_ prefix in <*sys/sysctl.h*>, and are as follows. The next and subsequent levels down are found in the include files listed here:

Name	Next Level Names	Description
CTL_DEBUG	sys/sysctl.h	Debugging
CTL_FS	sys/sysctl.h	Filesystem
CTL_HW	sys/sysctl.h	Generic CPU, I/O
CTL_KERN	sys/sysctl.h	High kernel limits
CTL_MACHDEP	sys/sysctl.h	Machine dependent
CTL_NET	sys/socket.h	Networking
CTL_USER	sys/sysctl.h	User-level
CTL_VM	vm/vm_param.h	Virtual memory

1.7.2. Bootstrap operations

The call:

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

extends the name space. The *mount* call grafts a filesystem object onto the system file tree at the point specified in *dir*. The argument *type* specifies the type of filesystem to be mounted. The argument *data* describes the filesystem object to be mounted according to the *type*. The contents of the filesystem become available through the new mount point *dir*. Any files in or below *dir* at the time of a successful mount disappear from the name space until the filesystem is unmounted. The *flags* value specifies generic properties, such as a request to mount the filesystem read-only.

Information about all mounted filesystems can be obtained with the call:

```
getfsstat(buf, bufsize, flags);
result struct statfs *buf; long bufsize, int flags;
```

The call:

```
swapon(blkdev);
char *blkdev;
```

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

1.7.3. Shutdown operations

The call:

```
unmount(dir, flags);
char *dir; int flags;
```

unmounts the filesystem mounted on *dir*. This call will succeed only if the filesystem is not currently being used or if the MNT_FORCE flag is specified.

The call:

```
sync();
```

schedules I/O to flush all modified disk blocks resident in the kernel. (This call does not require privileged status.) Files can be selectively flushed to disk using the *fsync* call (see section 2.2.6).

The call:

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

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

1.7.4. 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.

2. System facilities

The system abstractions described are:

Directory contexts

A directory context is a position in the filesystem name space. Operations on files and other named objects in a filesystem are always specified relative to such a context.

Files

Files are used to store uninterpreted sequences of bytes, which may be *read* and *written* randomly. Pages from files may also be mapped into the process address space. A directory may be read as a file if permitted by the underlying storage facility, though it is usually accessed using *getdirentries* (see section 2.2.3.1). (Local filesystems permit directories to be read, although most NFS implementations do not allow reading of directories.)

Communications domains

A communications domain represents an interprocess communications environment, such as the communications facilities of the 4.4BSD 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, 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, interrupt generation, output flow control, and editing), magnetic tapes, disks, and other peripherals. They normally 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.

2.1. Generic operations

Many system abstractions support the *read*, *write*, and *ioctl* operations. 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.

2.1.1. 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 ssize_t cc; int fd; result void *buf; size_t nbytes;

cc = write(fd, buf, nbytes);
result ssize_t cc; int fd; void *buf; size_t 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. A return value of 0 is used to indicate an end-of-file condition.

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 only a portion of the provided bytes; the user should resubmit the other bytes in a later request. Error returns because of interrupted or otherwise incomplete operations are possible, in which case no data will have been transferred.

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 {
        char      *iov_base;    /* base of a component */
        size_t    iov_len;      /* length of a component */
};
```

The *iov_base* field should be treated as if its type were ‘‘void *’’ as POSIX and other versions of the structure may use that type. Thus, pointer arithmetic should not use this value without a cast.

The calls using an array of *iovec* structures are:

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

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

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

2.1.2. Input/output control

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

```
ioctl(fd, request, buffer);
int fd; u_long 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 used, 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 by <*sys/ioctl.h*>, or more specifically by files it includes.

2.1.3. 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 1.5.4. 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 writable.

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 complete immediately. The descriptor may then be *select*’ed for *write* to find out when the operation has been completed. When *select* indicates the descriptor is writable, the operation has completed. Depending on the nature of the descriptor and the operation, additional activity may be started or the new state may be tested.

2.2. Filesystem

2.2.1. Overview

The filesystem abstraction provides access to a hierarchical filesystem structure. The filesystem contains directories (each of which may contain sub-directories) 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. If permitted by the underlying storage mechanism, 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.

2.2.2. Naming

The filesystem 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 NAME_MAX (255) characters excluding null and ‘‘/’’. Each pathname is up to PATH_MAX (1024) characters excluding null.

Each process always has two naming contexts: one for the root directory of the filesystem 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 file name ‘‘.’’ in each directory refers to that directory. The file name ‘‘..’’ in each directory refers to the parent directory of that directory. The parent directory of the root of the filesystem is itself.

The calls:

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

fchdir(fd);
int fd;

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

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

Information about a filesystem that contains a particular file can be obtained using the calls:

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

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

2.2.3. Creation and removal

The filesystem allows directories, files, special devices, and fifos to be created and removed from the filesystem.

2.2.3.1. Directory creation and removal

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

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

where the mode is defined as for files (see section 2.2.3.2). Directories are removed with the *rmdir* system call:

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

A directory must be empty (other than the entries ‘‘.’’ and ‘‘..’’) if it is to be deleted.

Although directories can be read as files, the usual interface is to use the call:

```
getdirentries(fd, buf, nbytes, basep);
int fd; char *buf; int nbytes; long *basep;
```

The *getdirentries* system call returns a canonical array of directory entries in the filesystem independent format described in <*dirent.h*>. Application programs usually use the library routines *opendir*, *readdir*, and *closedir* which provide a more convenient interface than *getdirentries*. The *fts* package is provided for recursive directory traversal.

2.2.3.2. File creation

Files are opened and/or created with the *open* system call:

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

The *path* parameter specifies the name of the file to be opened. The *oflag* parameter must include O_CREAT to cause the file to be created. Bits for *oflag* are defined in <*sys/fcntl.h*>:

```
O_RDONLY     /* open for reading only */
O_WRONLY     /* open for writing only */
O_RDWR       /* open for reading and writing */
O_NONBLOCK   /* no delay */
O_APPEND     /* set append mode */
O_SHLOCK     /* open with shared file lock */
O_EXLOCK     /* open with exclusive file lock */
O_ASYNC      /* signal pgrp when data ready */
O_FSYNC      /* synchronous writes */
O_CREAT      /* create if nonexistent */
O_TRUNC      /* truncate to zero length */
O_EXCL       /* error if already exists */
```

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be done 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 all writes to be appended to the file. Specifying O_TRUNC causes the file to be truncated when opened. The flag O_CREAT causes the file to be created if it does not exist, owned by the current user and the group of the containing directory. The permissions for the new file are specified in *mode* as the OR of the appropriate permissions as defined in <*sys/stat.h*>:

```
S_IRWXU         /* RWX for owner */
S_IRUSR         /* R for owner */
S_IWUSR         /* W for owner */
S_IXUSR         /* X for owner */
S_IRWXG         /* RWX for group */
S_IRGRP         /* R for group */
S_IWGRP         /* W for group */
S_IXGRP         /* X for group */
S_IRWXO         /* RWX for other */
S_IROTH         /* R for other */
S_IWOTH         /* W for other */
S_IXOTH         /* X for other */
S_ISUID         /* set user id */
S_ISGID /* set group id */
S_ISTXT /* sticky bit */
```

Historically, the file mode has been used as a four digit octal number. The bottom three digits encode 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. The 7000 bits encode set user ID as 4000, set group ID as 2000, and the sticky bit as 1000. The mode specified to *open* is modified by the process *umask*; permissions specified in the *umask* are cleared in the mode of the created file. The *umask* can be changed with the call:

```
oldmask = umask(newmask);
result mode_t oldmask; mode_t newmask;
```

If the O_EXCL flag is set, and the file already exists, then the *open* will fail without affecting the file in any way. This mechanism provides a simple exclusive access facility. For security reasons, if the O_EXCL flag is set and the file is a symbolic link, the open will fail regardless of the existence of the file referenced by the link. The O_SHLOCK and O_EXLOCK flags allow the file to be atomically *open*'ed and *flock*'ed; see section 2.2.7 for the semantics of *flock* style locks. The O_ASYNC flag enables the SIGIO signal to be sent to the process group of the opening process when I/O is possible, e.g., upon availability of data to be read.

2.2.3.3. Creating references to devices

The filesystem 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 either one of possibly many peripherals of that kind, or special characteristics of the peripheral. Structured devices have all operations done internally in ‘‘block’’ quantities while unstructured devices may have input and output done in varying units, and may act as a non-seekable communications channel rather than a random-access device. The *mknod* call creates special entries:

```
mknod(path, mode, dev);
char *path; mode_t mode; dev_t 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.

Fifo's can be created in the filesystem using the call:

```
mkfifo(path, mode);
char *path; mode_t mode;
```

The *mode* parameter is used solely to specify the access permissions of the newly created fifo.

2.2.3.4. Links and renaming

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

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 filesystem. Each link to a file is equivalent, referring to the file independently of any other name. Symbolic links cause string substitution during the pathname interpretation process, and refer to a file name rather than referring directly to a file.

Hard links and symbolic links have different properties. A hard link ensures that the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Unlike hard links, symbolic links can refernce directories and span filesystems boundaries. An *lstat* (see section 2.2.4) call on a hard link will return the information about the file (or directory) that the link references while an *lstat* call on a symbolic link will return information about the link itself. A symbolic link does not have an owner, group, permissions, access and modification times, etc. The only attributes returned from an *lstat* that refer to the symbolic link itself are the file type (S_IFLNK), size, blocks, and link count (always 1). The other attributes are filled in from the directory that contains the link.

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; char *path; result char *buf; int bufsize;
```

This call returns, in *buf*, the string substituted into pathnames passing through *path*. (This string is not NULL terminated.)

Atomic renaming of filesystem resident objects is possible with the *rename* call:

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

where both *oldname* and *newname* must be in the same filesystem. If either *oldname* or *newname* is a directory, then the other also must be a directory for the *rename* to succeed. If *newname* exists and is a directory, then it must be empty.

2.2.3.5. File, device, and fifo removal

A reference to a file, special device or fifo 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. When the last name for a file has been removed, the file may no longer be opened; the file itself is removed once any existing references have been closed.

All current access to a file can be revoked using the call:

```
revoke(path);
char *path;
```

Subsequent operations on any descriptors open at the time of the *revoke* fail, with the exceptions that a *close* call will succeed, and a *read* from a character device file which has been revoked returns a count of zero (end of file). If the file is a special file for a device which is open, the device close function is called as if all open references to the file had been closed. *Open*'s done after the *revoke* may succeed. This call is most useful for revoking access to a terminal line after a hangup in preparation for reuse by a new login session. Access to a controlling terminal is automatically revoked when the session leader for the session exits.

2.2.4. Reading and modifying file attributes

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

```
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 obtained 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 them and the group ID of the directory in which they were created. The ownership of a file may be changed by either of the calls:

```
chown(path, owner, group);
char *path; uid_t owner; gid_t group;

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

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and other. 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; mode_t mode;

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

where *mode* is a value indicating the new protection of the file, as listed in section 2.2.3.2.

Each file has a set of flags stored as a bit mask associated with it. These flags are returned in the *stat* structure and are set using the calls:

```
chflags(path, flags);
char *path; u_long flags;

fchflags(fd, flags);
int fd; u_long flags;
```

The flags specified are formed by or'ing the following values:

UF_NODUMP	Do not dump the file.
UF_IMMUTABLE	The file may not be changed.
UF_APPEND	The file may only be appended to.
SF_IMMUTABLE	The file may not be changed.
SF_APPEND	The file may only be appended to.

The UF_NODUMP, UF_IMMUTABLE and UF_APPEND flags may be set or unset by either the owner of a file or the super-user. The SF_IMMUTABLE and SF_APPEND flags may only be set or unset by the super-user. They may be set at any time, but normally may only be unset when the system is in single-user mode.

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 file access and modification times.

2.2.5. 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;
```

How is constructed by OR'ing the following bits, defined in <*unistd.h*>:

```
F_OK    /* file exists */
X_OK    /* file is executable/searchable */
W_OK    /* file is writable */
R_OK    /* file is readable */
```

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

The *pathconf* and *fpathconf* functions provide a method for applications to determine the current value of a configurable system limit or option variable associated with a pathname or file descriptor:

```
ans = pathconf(path, name);
result long ans; char *path; int name;

ans = fpathconf(fd, name);
result long ans; int fd, name;
```

For *pathconf*, the *path* argument is the name of a file or directory. For *fpathconf*, the *fd* argument is an open file descriptor. The *name* argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file <*unistd.h*>.

2.2.6. 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 defined by <*unistd.h*> as one of:

```
SEEK_SET    /* set file offset to offset */
SEEK_CUR    /* set file offset to current plus offset */
SEEK_END    /* set file offset to EOF plus offset */
```

The call ‘‘lseek(fd, 0, SEEK_CUR)’’ returns the current offset into the file.

Files may have ‘‘holes’’ in them. Holes are 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 extended or truncated with either of the calls:

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

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

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

Unless opened with the O_FSYNC flag, writes to files are held for an indeterminate period of time in the system buffer cache. The call:

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

ensures that the contents of a file are committed to disk before returning. This feature is used by applications such as editors that want to ensure the integrity of a new file before continuing.

2.2.7. Locking

The filesystem 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. Byte range locking is available with *fcntl*; see section 1.5.4. The system does not force processes to obey the locks; they are of an advisory nature only.

Locking can be done as part of the *open* call (see section 2.2.3.2) or 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 <*fcntl.h*>:

```
LOCK_SH    /* shared file lock */
LOCK_EX    /* exclusive file lock */
LOCK_NB    /* don’t block when locking */
LOCK_UN    /* unlock file */
```

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.

2.2.8. Disk quotas

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

The *quotactl* call enables, disables and manipulates filesystem quotas:

```
quotactl(path, cmd, id, addr);
char *path; int cmd; int id; char *addr;
```

A quota control command given by cmd operates on the given filename path for the given user ID. The address of an optional command specific data structure, addr, may be given. The supported commands include:

```
Q_QUOTAON     /* enable quotas */
Q_QUOTAOFF    /* disable quotas */
Q_GETQUOTA    /* get limits and usage */
Q_SETQUOTA    /* set limits and usage */
Q_SETUSE      /* set usage */
Q_SYNC        /* sync disk copy of a filesystems quotas */
```

2.2.9. Remote filesystems

There are two system calls intended to help support the remote filesystem implementation. The call:

```
nfssvc(flags, argstructp);
int flags, void *argstructp;
```

is used by the NFS daemons to pass information into and out of the kernel and also to enter the kernel as a server daemon. The flags argument consists of several bits that show what action is to be taken once in the kernel and *argstructp* points to one of three structures depending on which bits are set in flags.

The call:

```
getfh(path, fhp);
char *path; result fhandle_t *fhp;
```

returns a file handle for the specified file or directory in the file handle pointed to by fhp. This file handle can then be used in future calls to NFS to access the file without the need to repeat the pathname translation. This system call is restricted to the superuser.

2.2.10. Other filesystems

The kernel supports many other filesystems. These include:

- The log-structured filesystem. It provides an alternate disk layout than the fast filesystem optimized for writing rather than reading. For further information see the mount_lfs(8) manual page.
- The ISO-standard 9660 filesystem with Rock Ridge extensions used for CD-ROMs. For further information see the mount_cd9660(8) manual page.
- The file descriptor mapping filesystem. For further information see the mount_fdesc(8) manual page.
- The /proc filesystem as an alternative for debuggers. For further information see section 2.5.1 and the mount_procfs(8) manual page.

- The memory-based filesystem, used primarily for fast but ethereal uses such as /tmp. For further information see the mount_mfs(8) manual page.
- The kernel variable filesystem, used as an alternative to *sysctl*. For further information see section 1.7.1 and the mount_kernfs(8) manual page.
- The portal filesystem, used to mount processes in the filesystem. For further information see the mount_portal(8) manual page.
- The uid/gid remapping filesystem, usually layered above NFS filesystems exported to an outside administrative domain. For further information see the mount_umap(8) manual page.
- The union filesystem, used to place a writable filesystem above a read-only filesystem. This filesystem is useful for compiling sources on a CD-ROM without having to copy the CD-ROM contents to writable disk. For further information see the mount_union(8) manual page.

2.3. Interprocess communications

2.3.1. Interprocess communication primitives

2.3.1.1. Communication domains

The system provides access to an extensible set of communication *domains*. A communication domain (or protocol family) is identified by a manifest constant defined in the file <*sys/socket.h*>. Important standard domains supported by the system are the local (‘‘UNIX’’) domain (PF_LOCAL or PF_UNIX) for communication within the system, the ‘‘Internet’’ domain (PF_INET) for communication in the DARPA Internet, the ISO family of protocols (PF_ISO and PF_CCITT) for providing a check-off box on the list of your system capabilities, and the ‘‘NS’’ domain (PF_NS) for communication using the Xerox Network Systems protocols. Other domains can be added to the system.

2.3.1.2. 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 of an appropriate type 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	
SOCK_DGRAM	/* datagram */
SOCK_STREAM	/* virtual circuit */
SOCK_RAW	/* raw socket */
SOCK_RDM	/* reliably-delivered message */
SOCK_SEQPACKET	/* 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. A datagram socket may send messages to and receive messages from multiple peers. 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 or unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. Connection setup is required before data communication may begin. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, exchange preserving message boundaries; 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.

Each socket may have a specific *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and the implementation of a suitable protocol within the domain. 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.

2.3.1.3. 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;
```

The socket domain and type are as described above, and are specified using the definitions from *<sys/socket.h>*. The protocol may be given as 0, meaning any suitable protocol. One of several possible protocols may be selected using identifiers obtained from a library routine, *getprotobyname*.

An unconnected socket descriptor of a connection-oriented type 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. Datagram sockets need not establish connections before use.

To accept connections or to receive datagrams, a socket must first have a binding to a name (or address) within the communications domain. Such a binding may be established by a *bind* call:

```
bind(s, name, namelen);
int s; struct sockaddr *name; int namelen;
```

Datagram sockets may have default bindings established when first sending data if not explicitly bound earlier. In either case, a socket’s bound name may be retrieved with a *getsockname* call:

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

while the peer’s name can be retrieved with *getpeername*:

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

Domains may support sockets with several names.

2.3.1.4. Accepting connections

Once a binding is made to a connection-oriented socket, 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:

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

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*. If no new connections are queued for acceptance, the call will wait for a connection unless non-blocking I/O has been enabled (see section 1.5.4).

2.3.1.5. Making connections

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

```
connect(s, name, namelen);
int s; struct sockaddr *name; int namelen;
```

Although datagram sockets do not establish connections, the *connect* call may be used with such sockets to create an *association* with the foreign address. The address is recorded for use in future *send* calls, which then need not supply destination addresses. Datagrams will be received only from that peer, and asynchronous error reports may be received.

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†:

```
socketpair(domain, type, protocol, sv);
int domain, 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 PF_LOCAL domain, with pv[0] only writable and pv[1] only readable.

2.3.1.6. Sending and receiving data

Messages may be sent from a socket by:

```
cc = sendto(s, msg, len, flags, to, tolen);
result int cc; int s; void *msg; size_t len;
int flags; struct sockaddr *to; int tolen;
```

if the socket is not connected or:

```
cc = send(s, msg, len, flags);
result int cc; int s; void *msg; size_t len; int flags;
```

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

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
result int msglen; int s; result void *buf; size_t len; int flags;
result struct sockaddr *from; result int *fromlenaddr;
```

and:

```
msglen = recv(s, buf, len, flags);
result int msglen; int s; result void *buf; size_t len; int 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, sending or receiving high-priority out-of-band messages, or other special requests as follows:

† 4.4BSD supports *socketpair* creation only in the PF_LOCAL communication domain.

```
MSG_OOB           /* process out-of-band data */
MSG_PEEK          /* peek at incoming message */
MSG_DONTROUTE     /* send without using routing tables */
MSG_EOR           /* data completes record */
MSG_TRUNC         /* data discarded before delivery */
MSG_CTRUNC        /* control data lost before delivery */
MSG_WAITALL       /* wait for full request or error */
MSG_DONTWAIT      /* this message should be nonblocking */
```

2.3.1.7. 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 can be used to conveniently contain the parameters to the calls:

```
struct msghdr {
        caddr_t   msg_name;          /* optional address */
        u_int     msg_namelen;       /* size of address */
        struct    iovec *msg_iov;    /* scatter/gather array */
        u_int     msg_iovlen;        /* # elements in msg_iov */
        caddr_t   msg_control;       /* ancillary data */
        u_int     msg_controllen;    /* ancillary data buffer len */
        int       msg_flags;         /* flags on received message */
};
```

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 2.1.1. The data in the *msg_control* buffer is composed of an array of variable length messages used for additional information with or about a datagram not expressible by flags. The format is a sequence of message elements headed by *cmsghdr* structures:

```
struct cmsghdr {
        u_int     cmsg_len;        /* data byte count, including hdr */
        int       cmsg_level;      /* originating protocol */
        int       cmsg_type;       /* protocol-specific type */
        u_char    cmsg_data[ ];    /* variable length type specific data */
};
```

The following macros are provided for use with the *msg_control* buffer:

```
CMSG_FIRSTHDR(mhdr)          /* given msghdr, return first cmsghdr */
CMSG_NXTHDR(mhdr, cmsg)      /* given msghdr and cmsghdr, return next cmsghdr */
CMSG_DATA(cmsg)              /* given cmsghdr, return associated data pointer */
```

Access rights to be sent along with the message are specified in one of these *cmsghdr* structures, with level SOL_SOCKET and type SCM_RIGHTS. In the PF_LOCAL domain these are an array of integer descriptors, copied 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 flags;
```

2.3.1.8. Using read and write with sockets

The normal *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.

2.3.1.9. 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. If the underlying protocol supports unidirectional or bidirectional shutdown, this indication will be passed to the peer. For example, a shutdown for writing might produce an end-of-file condition at the remote end.

2.3.1.10. Socket and protocol options

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

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

setsockopt(s, level, optname, optval, optlen);
int s, level, optname; void *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’’ within the protocol family.

2.3.2. PF_LOCAL domain

This section describes briefly the properties of the PF_LOCAL (‘‘UNIX’’) communications domain.

2.3.2.1. Types of sockets

In the local domain, the SOCK_STREAM abstraction provides pipe-like facilities, while SOCK_DGRAM provides (usually) reliable message-style communications.

2.3.2.2. Naming

Socket names are strings and may appear in the filesystem name space.

2.3.2.3. Access rights transmission

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

2.3.3. 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 document describing the network implementation in 4.4BSD (SMM:18).

2.3.3.1. Socket types and protocols

SOCK_STREAM is supported by the Internet TCP protocol; SOCK_DGRAM by the UDP protocol. Each is layered atop the transport-level Internet Protocol (IP). The Internet Control Message Protocol is implemented atop/beside IP and is accessible via a raw socket. 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.

2.3.3.2. Socket naming

Sockets in the Internet domain have names composed of a 32-bit Internet address and a 16-bit port number. Options may be used to provide IP source routing or security options. The 32-bit address is composed of network and host parts; the network part is variable in size and is frequency encoded. The host part may optionally be interpreted as a subnet field plus the host on the subnet; this is is enabled by setting a network address mask at boot time.

2.3.3.3. Access rights transmission

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

2.3.3.4. Raw access

The Internet domain allows the super-user access to the raw facilities of IP. These interfaces are modeled as SOCK_RAW sockets. Each raw socket is associated with one IP protocol number, and receives all traffic received for that protocol. This approach allows administrative and debugging functions to occur, and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

2.4. Terminals and Devices

2.4.1. Terminals

Terminals support *read* and *write* I/O operations, as well as a collection of terminal specific *ioctl* operations, to control input character interpretation and editing, and output format and delays.

A terminal may be used as a controlling terminal (login terminal) for a login session. A controlling terminal is associated with a session (see section 1.1.4). A controlling terminal has a foreground process group, which must be a member of the session with which the terminal is associated (see section 1.1.5). Members of the foreground process group are allowed to read from and write to the terminal and change the terminal settings; other process groups from the session may be stopped upon attempts to do these operations.

A session leader allocates a terminal as the controlling terminal for its session using the ioctl

```
ioctl(fd, TIOCSCTTY, NULL);
int fd;
```

Only a session leader may acquire a controlling terminal.

2.4.1.1. Terminal input

Terminals are handled according to the underlying communication characteristics such as baud rate and required delays, and a set of software parameters. These parameters are described in the *termios* structure maintained by the kernel for each terminal line:

```
struct termios {
	tcflag_t	c_iflag;	/* input flags */
	tcflag_t	c_oflag;	/* output flags */
	tcflag_t	c_cflag;	/* control flags */
	tcflag_t	c_lflag;	/* local flags */
	cc_t	c_cc[NCCS];	/* control chars */
	long	c_ispeed;	/* input speed */
	long	c_ospeed;	/* output speed */
};
```

The *termios* structure is set and retrieved using the *tcsetattr* and *tcgetattr* functions.

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or noncanonical mode. Additionally, input characters are processed according to the *c_iflag* and *c_lflag* fields. Such processing can include echoing, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. Non-graphic ASCII input characters may be echoed as a two-character printable representation, ‘‘ˆcharacter.’’

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a newline character (NL), an end-of-file (EOF) character, or an end-of-line (EOL) character. Input is presented on a line-by-line basis. Using this mode means that a read request will not return until an entire line has been typed, or a signal has been received. Also, no matter how many bytes are requested in the read call, at most one line is returned. It is not, however, necessary to read a whole line at once; any number of bytes, even one, may be requested in a read without losing information.

When the terminal is in canonical mode, editing of an input line is performed. Editing facilities allow deletion of the previous character or word, or deletion of the current input line. In addition, a special character may be used to reprint the current input line. Certain other characters are also interpreted specially. Flow control is provided by the *stop output* and *start output* control characters. Output may be flushed with the *flush output* character; and the *literal character* may be used to force the following character into the input line, regardless of any special meaning it may have.

In noncanonical mode input processing, input bytes are not assembled into lines, and erase and kill processing does not occur. All input is passed through to the reading process immediately and without interpretation. Signals and flow control may be enabled; here the handler interprets input only by looking for characters that cause interrupts or output flow control; all other characters are made available.

When interrupt characters are being interpreted by the terminal handler they cause a software interrupt to be sent to all processes in the process group associated with the terminal. Interrupt characters exist to send SIGINT and SIGQUIT signals, and to stop a process group with the SIGTSTP signal either immediately, or when all input up to the stop character has been read.

2.4.1.2. Terminal output

On output, the terminal handler provides some simple formatting services. These include converting the carriage return character to the two character return-linefeed sequence, inserting delays after certain standard control characters, and expanding tabs.

2.4.2. Structured devices

Structured devices are typified by disks and magnetic tapes, but may represent any random-access device. The system performs read-modify-write type buffering actions on block devices to allow them to be read and written in random access fashion like ordinary files. Filesystems are normally mounted on block devices.

2.4.3. Unstructured devices

Unstructured devices are those devices which do not support block structure. Familiar unstructured devices are raw communications lines (with no terminal handler), raster plotters, magnetic tape and disks unfettered by buffering and permitting large block input/output and positioning and formatting commands.

2.5. Process debugging

2.5.1. Traditional debugging

Debuggers traditionally use the *ptrace* interface:

```
ptrace(request, pid, addr, data);
int request, pid, *addr, data;
```

This interface provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a request argument. A process being traced behaves normally until it encounters a signal (whether internally generated like ‘‘illegal instruction’’ or externally generated like ‘‘interrupt’’). Then the traced process enters a stopped state and its parent is notified via *wait*. When the child is in the stopped state, its core image can be examined and modified using *ptrace*. Another ptrace request can then cause the child either to terminate or to continue, possibly ignoring the signal.

A more general interface is also provided in 4.4BSD; the *mount_procfs* filesystem attaches an instance of the process name space to the global filesystem name space. The conventional mount point is */proc*. The root of the process filesystem contains an entry for each active process. These processes are visible as directories named by the process’ ID. In addition, the special entry *curproc* references the current process. Each directory contains several files, including a *ctl* file. The debugger finds (or creates) the process that it wants to debug and then issues an attach command via the *ctl* file. Further interaction can then be done with the process through the other files provided by the */proc* filesystem.

2.5.2. Kernel tracing

Another facility for debugging programs is provided by the *ktrace* interface:

```
ktrace(tracefile, ops, trpoints, pid);
char *tracefile; int ops, trpoints, pid;
```

Ktrace does kernel trace logging for the specified processes. The kernel operations that are traced include system calls, pathname translations, signal processing, and I/O. This facility can be particularly useful to debug programs for which you do not have the source.

3. Summary of facilities

1 Kernel primitives

1.1 Processes and protection

sethostid	set host identifier
gethostid	get host identifier
sethostname	set host name
gethostname	get host name
getpid	get process identifier
getppid	get parent process identifier
fork	create a new process
vfork	create a new process
exit	terminate a process
wait4	collect exit status of child
execve	execute a new program
getuid	get real user identifier
geteuid	get effective user identifier
getgid	get real group identifier
getegid	get effective group identifier
getgroups	get access group set
setuid	set real, effective, and saved user identifiers
setgid	set real, effective, and saved group identifiers
setgroups	set access group set
seteuid	set effective user identifier
setegid	set effective group identifier
setsid	create a new session
setlogin	set login name
getlogin	get login name
getpgrp	get process group
setpgid	set process group

1.2 Memory management

brk	set data section size
sbrk	change data section size
getpagesize	get system page size
mmap	map files or devices into memory
msync	synchronize a mapped region
munmap	remove a mapping
mprotect	control the protection of pages
madvise	give advise about use of memory
mincore	get advise about use of memory
mlock	lock physical pages in memory
munlock	unlock physical pages in memory
mset	acquire and set a semaphore
mclear	release a semaphore and awaken waiting processes
msleep	wait for a semaphore
mwakeup	awaken process(es) sleeping on a semaphore

1.3 Signals

sigaction	setup software signal handler
sigreturn	return from a signal
kill	send signal to a process
killpg	send signal to a process group
sigprocmask	manipulate current signal mask
sigsuspend	atomically release blocked signals and wait for interrupt

sigpending	get pending signals
sigaltstack	set and/or get signal stack context

1.4 Timers

settimeofday	set date and time
gettimeofday	get date and time
adjtime	synchronization of the system clock
setitimer	set value of interval timer
getitimer	get value of interval timer
profil	control process profiling

1.5 Descriptors

getdtablesize	get descriptor table size
dup	duplicate an existing file descriptor
dup2	duplicate an existing file descriptor
close	delete a descriptor
select	synchronous I/O multiplexing
fcntl	file control

1.6 Resource controls

getpriority	get program scheduling priority
setpriority	set program scheduling priority
getrusage	get information about resource utilization
getrlimit	get maximum system resource consumption
setrlimit	set maximum system resource consumption

1.7 System operation support

sysctl	get or set system information
mount	mount a filesystem
getfsstat	get list of all mounted filesystems
swapon	add a swap device for interleaved paging/swapping
unmount	dismount a filesystem
sync	force completion of pending disk writes (flush cache)
reboot	reboot system or halt processor
acct	enable or disable process accounting

2 System facilities

2.1 Generic operations

read	read input
write	write output
readv	read gathered input
writev	write scattered output
ioctl	control device

2.2 Filesystem

chdir	change current working directory
fchdir	change current working directory
chroot	change root directory
statfs	get file system statistics
fstatfs	get file system statistics
mkdir	make a directory file
rmdir	remove a directory file
getdirentries	get directory entries in a filesystem independent format
open	open or create a file for reading or writing
umask	set file creation mode mask
mknod	make a special file node
mkfifo	make a fifo file
link	make a hard file link

symlink	make a symbolic link to a file
readlink	read value of a symbolic link
rename	change the name of a file
unlink	remove directory entry
revoke	revoke file access
stat	get file status
fstat	get file status
lstat	get file status
chown	change owner and group of a file
fchown	change owner and group of a file
chmod	change mode of file
fchmod	change mode of file
chflags	set file flags
fchflags	set file flags
utimes	set file access and modification times
access	check access permissions of a file or pathname
pathconf	get configurable pathname variables
fpathconf	get configurable pathname variables
lseek	reposition read/write file offset
truncate	truncate a file to a specified length
ftruncate	truncate a file to a specified length
fsync	synchronize in-core state of a file with that on disk
flock	apply or remove an advisory lock on an open file
quotactl	manipulate filesystem quotas
nfssvc	NFS services
getfh	get file handle

2.3 Interprocess communications

socket	create an endpoint for communication
bind	bind a name to a socket
getsockname	get socket name
getpeername	get name of connected peer
listen	listen for connections on a socket
accept	accept a connection on a socket
connect	initiate a connection on a socket
socketpair	create a pair of connected sockets
pipe	create descriptor pair for interprocess communication
sendto	send a message from a socket
send	send a message from a socket
recvfrom	receive a message from a socket
recv	receive a message from a socket
sendmsg	send a message from a socket
recvmsg	receive a message from a socket
shutdown	shut down part of a full-duplex connection
getsockopt	get options on socket
setsockopt	set options on socket

2.4 Terminals and Devices

2.5 Process debugging

ptrace	process trace
ktrace	process tracing

3 Summary of facilities

Contents