Berkeley Software Architecture Manual
4.3BSD Edition

William Joy, Robert Fabry,

Samuel Leffler, M. Kirk McKusick,

Michael Karels

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 facilities provided by the 4.3BSD version of the UNIX * 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 UNIX process: process naming and protection, memory management, software interrupts, object references (descriptors), time and statistics functions, 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 file systems, communication, terminal handling, and process control and debugging. These facilities are implemented by the operating system or by network server processes.

---

* UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

Introduction.

0. Notation and types

1. Kernel primitives

1.1. Processes and protection
1.1.1. Host and process identifiers
1.1.2. Process creation and termination
1.1.3. User and group ids
1.1.4. Process groups

1.2. Memory management
1.2.1. Text, data and stack
1.2.2. Mapping pages
1.2.3. Page protection control
1.2.4. Giving and getting advice
1.2.5. Protection primitives

1.3. Signals
1.3.1. Overview
1.3.2. Signal types
1.3.3. Signal handlers
1.3.4. Sending signals
1.3.5. Protecting critical sections
1.3.6. Signal stacks

1.4. Timing and statistics
1.4.1. Real time
1.4.2. Interval time

1.5. Descriptors
1.5.1. The reference table
1.5.2. Descriptor properties
1.5.3. Managing descriptor references
1.5.4. Multiplexing requests
1.5.5. Descriptor wrapping

1.6. Resource controls
1.6.1. Process priorities
1.6.2. Resource utilization
1.6.3. Resource limits

1.7. System operation support
1.7.1. Bootstrap operations
1.7.2. Shutdown operations
1.7.3. Accounting

2. System facilities

2.1. Generic operations
2.1.1. Read and write
2.1.2. Input/output control
2.1.3. Non-blocking and asynchronous operations

2.2. File system
2.2.1 Overview
2.2.2. Naming
2.2.3. Creation and removal
2.2.3.1. Directory creation and removal
2.2.3.2. File creation
2.2.3.3. Creating references to devices
2.2.3.4. Portal creation
2.2.3.6. File, device, and portal removal
2.2.4. Reading and modifying file attributes
2.2.5. Links and renaming
2.2.6. Extension and truncation
2.2.7. Checking accessibility
2.2.8. Locking
2.2.9. Disc quotas

2.3. Interprocess communication
2.3.1. Interprocess communication primitives
2.3.1.1. Communication domains
2.3.1.2. Socket types and protocols
2.3.1.3. Socket creation, naming and service establishment
2.3.1.4. Accepting connections
2.3.1.5. Making connections
2.3.1.6. Sending and receiving data
2.3.1.7. Scatter/gather and exchanging access rights
2.3.1.8. Using read and write with sockets
2.3.1.9. Shutting down halves of full-duplex connections
2.3.1.10. Socket and protocol options
2.3.2. UNIX domain
2.3.2.1. Types of sockets
2.3.2.2. Naming
2.3.2.3. Access rights transmission
2.3.3. INTERNET domain
2.3.3.1. Socket types and protocols
2.3.3.2. Socket naming
2.3.3.3. Access rights transmission
2.3.3.4. Raw access

2.4. Terminals and devices
2.4.1. Terminals
2.4.1.1. Terminal input
2.4.1.1.1 Input modes
2.4.1.1.2 Interrupt characters
2.4.1.1.3 Line editing
2.4.1.2. Terminal output
2.4.1.3. Terminal control operations
2.4.1.4. Terminal hardware support
2.4.2. Structured devices

2.4.3. Unstructured devices

2.5. Process control and debugging

I. Summary of facilities

0. Notation and types

The notation used to describe system calls is a variant of a C language call, consisting of a prototype call followed by declaration of parameters and results. An additional keyword **result**, not part of the normal C language, is used to indicate which of the declared entities receive results. As an example, consider the *read* call, as described in section 2.1:

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

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

Description of all error conditions arising from each system call is not provided here; they appear in the programmer's 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 in the include file <*sys/types.h*> and used in the specifications here and in many C programs. These include **caddr_t** giving a memory address (typically as a character pointer), **off_t** giving a file offset (typically as a long integer), and a set of unsigned types **u_char**, **u_short**, **u_int** and **u_long**, shorthand names for **unsigned char**, **unsigned short**, etc.

1. Kernel primitives

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

The facilities implemented in the kernel are those which define the *UNIX 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 UNIX virtual machine also allows access to files and other objects through a set of *descriptors*. Each descriptor resembles a device controller, and supports a set of operations. Like devices on real machines, some of which are internal to the machine and some of which are external, parts of the descriptor machinery are built-in to the operating system, while other parts are often implemented in server processes on other machines. The facilities provided through the descriptor machinery are described in section 2.

1.1. Processes and protection

1.1.1. Host and process identifiers

Each UNIX host has associated with it a 32-bit host id, and a host name of up to 64 characters (as defined by MAXHOSTNAMELEN in <*sys/param.h*>). These are set (by a privileged user) and returned by the calls:

```
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;
```

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

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

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

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

1.1.2. Process creation and termination

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

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

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

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

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

returning 8 bits of exit status to its parent.

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

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

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

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

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

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

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

1.1.3. User and group ids

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

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

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

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

the real and effective accounting 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 int gidset[gidsetsize];
```

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

---

* The type of the gidset array in getgroups and setgroups remains integer for compatibility with 4.2BSD. It may change to **gid_t** in future releases.

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

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

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

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

1.1.4. Process groups

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

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

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 terminals among several different jobs.

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

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

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

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

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

changes the size of the data area by *incr* bytes and returns the new end of the data area, while

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

changes the size of the stack area. The stack area is also automatically extended as needed. On the VAX the text and data areas are adjacent in the P0 region, while the stack section is in the P1 region, and grows downward.

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 */
#define  PROT_READ              0x04    /* pages can be read */
#define  PROT_WRITE             0x02    /* pages can be written */
#define  PROT_EXEC              0x01    /* pages can be executed */

/* flags contain mapping type, sharing type and options */
/* mapping type; choose one */
#define MAP_FILE                0x0001  /* mapped from a file or device */
#define MAP_ANON                0x0002  /* allocated from memory, swap space */
#define MAP_TYPE                0x000f  /* mask for type field */

/* sharing types; choose one */
#define  MAP_SHARED             0x0010  /* share changes */
#define  MAP_PRIVATE            0x0000  /* changes are private */

/* other flags */
#define MAP_FIXED               0x0020  /* map addr must be exactly as requested */
#define MAP_INHERIT             0x0040  /* region is retained after exec */
#define MAP_HASSEMAPHORE        0x0080  /* region may contain semaphores */
#define MAP_NOPREALLOC          0x0100  /* do not preallocate space */
```

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

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

The call:

---

† This section represents the interface planned for later releases of the system. Of the calls described in this section, only *sbrk* and *getpagesize* are included in 4.3BSD.

```
maddr = mmap(addr, len, prot, flags, fd, pos);
result caddr_t maddr; caddr_t addr; int *len, 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*. 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 actual amount mapped is returned in *len*. The *addr*, *len*, and *pos* parameters must all be multiples of the pagesize. 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_FILE, mapping a regular file or character-special device memory, and MAP_ANON, which maps memory not associated with any specific file. The file descriptor used for creating MAP_ANON regions is used only for naming, and may be given as −1 if no name is associated with the region.‡ The MAP_INHERIT flag allows a region to be inherited after an *exec*. The MAP_HASSEMAPHORE flag allows special handling for regions that may contain semaphores. The MAP_NOPREALLOC 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 *unmap* so that when they return from the signal the page fault could be successfully completed.

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

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

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; 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 unpredictable except after an *msync*.

A mapping can be removed by the call

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

---

‡ The current design does not allow a process to specify the location of swap space. In the future we may define an additional mapping type, MAP_SWAP, in which the file descriptor argument specifies a file or device to which swapping should be done.

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

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

```
#define  MADV_NORMAL      0   /* no further special treatment */
#define  MADV_RANDOM      1   /* expect random page references */
#define  MADV_SEQUENTIAL  2   /* expect sequential references */
#define  MADV_WILLNEED    3   /* will need these pages */
#define  MADV_DONTNEED    4   /* don't need these pages */
#define  MADV_SPACEAVAIL  5   /* insure that resources are reserved */
```

Finally, a process may obtain information about whether pages are core resident by using the call

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

Here the current core residency of the pages is returned in the character array *vec*, with a value of 1 meaning that the page is in-core.

1.2.5. Synchronization primitives

Primitives are provided for synchronization using semaphores in shared memory. 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 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 the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

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

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

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 in the file *<signal.h>*.

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

Software signals reflect interrupts generated by user request: SIGINT for the normal interrupt signal; SIGQUIT for the more powerful *quit* signal, that normally causes a core image to be generated; SIGHUP and SIGTERM that cause graceful process termination, either because a user has ‘‘hung up’’, or by user or program request; and SIGKILL, a more powerful termination signal which a process cannot catch or ignore. Programs may define their own asynchronous events using SIGUSR1 and SIGUSR2. Other software signals (SIGALRM, SIGVTALRM, SIGPROF) indicate the expiration of interval timers.

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

A process may be *stopped* by a signal sent to it or the members of its process group. The SIGSTOP signal is a powerful stop signal, because it cannot be caught. Other stop signals SIGTSTP, SIGTTIN, and SIGTTOU are used when a user request, input request, or output request respectively is the reason 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 warns that the limit on file size creation has been reached.

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

```
#include <signal.h>

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

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

assigns interrupt handler address *sv_handler* to signal *signo*. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants SIG_IGN and SIG_DEF used as values for *sv_handler* cause ignoring or defaulting of a condition. The *sv_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 mask for a signal *signo* is provided by the macro *sigmask*(*signo*), from *<signal.h>*. *Sv_flags* specifies whether system calls should be restarted if the signal handler returns and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If *osv* is non-zero, the previous signal vector is returned.

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

The mask of *blocked* signals is independent of handlers for signals. It 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 *sv_handler* is called by a C call of the form

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

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

1.3.4. Sending signals

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

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

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

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

Signals are also 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

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

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

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

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

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

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

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

1.3.6. Signal stacks

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

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

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

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

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

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

1.4. Timers

1.4.1. Real time

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

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

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

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

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

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

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

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 long *tvsec;
```

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

1.4.2. Interval time

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

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

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

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

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

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

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

and a timer is set or read by the call:

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

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

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

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

The *alarm* system call of earlier versions of UNIX is provided as a library routine using the ITIMER_REAL timer. The process profiling facilities of earlier versions of UNIX remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal. The *profil* call arranges for the kernel to begin gathering execution statistics for a process:

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

This begins sampling of 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 the process 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 thus have multiple references, and are reference counted by the system.

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

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

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

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

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

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

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

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:

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

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

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

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 section 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 section 2.1.3).
- 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.

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 0, 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;

/* interesting values for cmd */
#define    F_SETFL          3       /* set descriptor options */
#define    F_GETFL          4       /* get descriptor options */
#define    F_SETOWN         5       /* set descriptor owner (pid/pgrp) */
#define    F_GETOWN         6       /* get descriptor owner (pid/pgrp) */
```

The F_SETFL *cmd* may be used to set a descriptor in non-blocking I/O mode and/or enable 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.

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

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

1.5.5. Descriptor wrapping.†

A user process may build descriptors of a specified type by *wrapping* a communications channel with a system supplied protocol translator:

```
new = wrap(old, proto)
result int new; int old; struct dprop *proto;
```

Operations on the descriptor *old* are then translated by the system provided protocol translator into requests on the underlying object *old* in a way defined by the protocol. The protocols supported by the kernel may vary from system to system and are described in the programmers manual.

Protocols may be based on communications multiplexing or a rights-passing style of handling multiple requests made on the same object. For instance, a protocol for implementing a file abstraction may or may not include locally generated ‘‘read-ahead’’ requests. A protocol that provides for read-ahead may provide higher performance but have a more difficult implementation.

Another example is the terminal driving facilities. Normally a terminal is associated with a communications line, and the terminal type and standard terminal access protocol are wrapped around a synchronous communications line and given to the user. If a virtual terminal is required, the terminal driver can be wrapped around a communications link, the other end of which is held by a virtual terminal protocol interpreter.

† The facilities described in this section are not included in 4.3BSD.

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. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

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

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

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

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

1.6.2. Resource utilization

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

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

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

struct rusage {
        struct      timeval ru_utime;   /* user time used */
        struct      timeval ru_stime;   /* system time used */
        int         ru_maxrss;          /* maximum core resident set size: 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 */
};
```

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

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:

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

#define    RLIM_NLIMITS        6

#define    RLIM_INFINITY       0x7fffffff

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

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

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

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

1.7. System operation support

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

1.7.1. Bootstrap operations

The call

```
mount(blkdev, dir, ronly);
char *blkdev, *dir; int ronly;
```

extends the UNIX name space. The *mount* call specifies a block device *blkdev* containing a UNIX file system to be made available starting at *dir*. If *ronly* is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. *Dir* is normally a name in the root directory.

The call

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

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

1.7.2. Shutdown operations

The call

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

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

The call

```
sync();
```

schedules input/output to clean all system buffer caches. (This call does not require privileged status.)

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. These constants are defined in <*sys/reboot.h*>.

1.7.3. Accounting

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

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

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

2. System facilities

This section discusses the system facilities that are not considered part of the kernel.

The system abstractions described are:

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

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

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

Sockets
: A socket is an endpoint of communication and the focal point for IPC in a communications domain. Sockets may be created in pairs, or given names and used to rendezvous with other sockets in a communications domain, accepting connections from these sockets or exchanging messages with them. These operations model a labeled or unlabeled communications graph, and can be used in a wide variety of communications domains. Sockets can have different *types* to provide different semantics of communication, increasing the flexibility of the model.

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

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

† Support for mapping files is not included in the 4.3 release.

2.1. Generic operations

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

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 int cc; int fd; result caddr_t buf; int nbytes;

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

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

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

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

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

The calls using an array of descriptors are:

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

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

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

2.1.2. Input/output control

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

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

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

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

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 writeable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot 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 writeable, 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. File system

2.2.1. Overview

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other 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. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

2.2.2. Naming

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

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

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

The file name ‘‘..’’ in each directory refers to the parent directory of that directory. The parent directory of the root of the file system is always that directory.

The calls

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

chroot(path)
char *path;
```

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

2.2.3. Creation and removal

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

2.2.3.1. Directory creation and removal

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

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

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

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

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

2.2.3.2. File creation

Files are created with the *open* system call,

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

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

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

One of O_RDONLY, O_WRONLY and O_RDWR should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the *open* to succeed. Specifying O_APPEND causes writes to automatically append to the file. The flag O_CREAT causes the file to be created if it does not exist, owned by the current user and the group of the containing directory. The protection for the new file is specified in *mode*. The file mode is used as a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 0700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 07 bits describe the access rights for other processes.

If the open specifies to create the file with O_EXCL and the file already exists, then the *open* will fail without affecting the file in any way. This provides a simple exclusive access facility. If the file exists but is a symbolic link, the open will fail regardless of the existence of the file specified by the link.

2.2.3.3. Creating references to devices

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

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

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

2.2.3.4. Portal creation†

The call

```
fd = portal(name, server, param, dtype, protocol, domain, socktype)
result int fd; char *name, *server, *param; int dtype, protocol;
int domain, socktype;
```

places a *name* in the file system name space that causes connection to a server process when the name is

† The *portal* call is not implemented in 4.3BSD.

used. The portal call returns an active portal in *fd* as though an access had occurred to activate an inactive portal, as now described.

When an inactive portal is accessed, the system sets up a socket of the specified *socktype* in the specified communications *domain* (see section 2.3), and creates the *server* process, giving it the specified *param* as argument to help it identify the portal, and also giving it the newly created socket as descriptor number 0. The accessor of the portal will create a socket in the same *domain* and *connect* to the server. The user will then *wrap* the socket in the specified *protocol* to create an object of the required descriptor type *dtype* and proceed with the operation which was in progress before the portal was encountered.

While the server process holds the socket (which it received as *fd* from the *portal* call on descriptor 0 at activation) further references will result in connections being made to the same socket.

2.2.3.5. File, device, and portal removal

A reference to a file, special device or portal 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.

2.2.4. Reading and modifying file attributes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.5. Links and renaming

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

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

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

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

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

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

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

If a file is a symbolic link, the ‘‘value’’ of the link may be read with the *readlink* call,

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

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

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

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

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

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 given in <*sys/file.h*> as one of:

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

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

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

A file may be truncated with either of the calls:

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

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

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

2.2.7. Checking accessibility

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

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

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

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

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

2.2.8. Locking

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

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

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

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

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

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

2.2.9. Disk quotas

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

To enable disk quotas on a file system the *setquota* call is used:

```
setquota(special, file)
char *special, *file;
```

where *special* refers to a structured device file where a mounted file system exists, and *file* refers to a disk quota file (residing on the file system associated with *special*) from which user quotas should be obtained. The format of the disk quota file is implementation dependent.

To manipulate disk quotas the *quota* call is provided:

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

quota(cmd, uid, arg, addr)
int cmd, uid, arg; caddr_t addr;
```

The indicated *cmd* is applied to the user ID *uid*. The parameters *arg* and *addr* are command specific. The file <*sys/quota.h*> contains definitions pertinent to the use of this call.

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 is identified by a manifest constant defined in the file <*sys/socket.h*>. Important standard domains supported by the system are the ‘‘unix’’ domain, AF_UNIX, for communication within the system, the ‘‘Internet’’ domain for communication in the DARPA Internet, AF_INET, and the ‘‘NS’’ domain, AF_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 */
#define     SOCK_DGRAM      1       /* datagram */
#define     SOCK_STREAM     2       /* virtual circuit */
#define     SOCK_RAW        3       /* raw socket */
#define     SOCK_RDM        4       /* reliably-delivered message */
#define     SOCK_SEQPACKET  5       /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. 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/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, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

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

Other socket types can be defined.

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.

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†:

† 4.3BSD supports *socketpair* creation only in the ''unix'' communication domain.

```
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 UNIX 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, buf, len, flags, to, tolen);
result int cc; int s; caddr_t buf; int len, flags; caddr_t to; int tolen;
```

if the socket is not connected or:

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

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

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

and

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

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

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

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

2.3.1.7. Scatter/gather and exchanging access rights

It is possible 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 */
        int         msg_namelen;       /* size of address */
        struct      iov *msg_iov;      /* scatter/gather array */
        int         msg_iovlen;        /* # elements in msg_iov */
        caddr_t     msg_accrights;     /* access rights sent/received */
        int         msg_accrightslen;  /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section 2.1.3. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the ‘‘unix’’ domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

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

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

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

2.3.1.8. Using read and write with sockets

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

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 caddr_t optval; result int *optlen;

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

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

2.3.2. UNIX domain

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

2.3.2.1. Types of sockets

In the UNIX 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 UNIX file system name space through portals†.

2.3.2.3. Access rights transmission

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

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 4.3BSD.

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 the 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 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 allows administrative and debugging functions to occur, and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

---

† The 4.3BSD implementation of the UNIX domain embeds bound sockets in the UNIX file system name space; this may change in future releases.

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.

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.

2.4.1.1.1. Input modes

A terminal is in one of three possible modes: *raw*, *cbreak*, or *cooked*. In raw mode all input is passed through to the reading process immediately and without interpretation. In cbreak mode, the handler interprets input only by looking for characters that cause interrupts or output flow control; all other characters are made available as in raw mode. In cooked mode, input is processed to provide standard line-oriented local editing functions, and input is presented on a line-by-line basis.

2.4.1.1.2. Interrupt characters

Interrupt characters are interpreted by the terminal handler only in cbreak and cooked modes, and 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.

2.4.1.1.3. Line editing

When the terminal is in cooked 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 after some number of editing operations have been applied.

Certain other characters are interpreted specially when a process is in cooked mode. The *end of line* character determines the end of an input record. The *end of file* character simulates an end of file occurrence on terminal input. Flow control is provided by *stop output* and *start output* control characters. Output may be flushed with the *flush output* character; and a *literal character* may be used to force literal input of the immediately following character in the input line.

Input characters may be echoed to the terminal as they are received. Non-graphic ASCII input characters may be echoed as a two-character printable representation, ‘‘^character.’’

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, expanding tabs, and providing translations for upper-case only terminals.

2.4.1.3. Terminal control operations

When a terminal is first opened it is initialized to a standard state and configured with a set of standard control, editing, and interrupt characters. A process may alter this configuration with certain control operations, specifying parameters in a standard structure:†

---

† The control interface described here is an internal interface only in 4.3BSD. Future releases will probably use a modified interface based on currently-proposed standards.

```
struct ttymode {
        short       tt_ispeed;      /* input speed */
        int         tt_iflags;      /* input flags */
        short       tt_ospeed;      /* output speed */
        int         tt_oflags;      /* output flags */
};
```

and ‘‘special characters’’ are specified with the *ttychars* structure,

```
struct ttychars {
        char        tc_erasec;      /* erase char */
        char        tc_killc;       /* erase line */
        char        tc_intrc;       /* interrupt */
        char        tc_quitc;       /* quit */
        char        tc_startc;      /* start output */
        char        tc_stopc;       /* stop output */
        char        tc_eofc;        /* end-of-file */
        char        tc_brkc;        /* input delimiter (like nl) */
        char        tc_suspc;       /* stop process signal */
        char        tc_dsuspc;      /* delayed stop process signal */
        char        tc_rprntc;      /* reprint line */
        char        tc_flushc;      /* flush output (toggles) */
        char        tc_werasc;      /* word erase */
        char        tc_lnextc;      /* literal next character */
};
```

2.4.1.4. Terminal hardware support

The terminal handler allows a user to access basic hardware related functions; e.g. line speed, modem control, parity, and stop bits. A special signal, SIGHUP, is automatically sent to processes in a terminal’s process group when a carrier transition is detected. This is normally associated with a user hanging up on a modem controlled terminal line.

2.4.2. Structured devices

Structures 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 a totally random access fashion like ordinary files. File systems are normally created in 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 and kernel descriptors

The status of the facilities in this section is still under discussion. The *ptrace* facility of earlier UNIX systems remains in 4.3BSD. Planned enhancements would allow a descriptor-based process control facility.

I. Summary of facilities

1. Kernel primitives

1.1. Process naming and protection

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

1.2 Memory management

<sys/mman.h>	memory management definitions
sbrk	change data section size
sstk†	change stack section size
getpagesize	get memory page size
mmap†	map pages of memory
msync†	flush modified mapped pages to filesystem
munmap†	unmap memory
mprotect†	change protection of pages
madvise†	give memory management advice
mincore†	determine core residency of pages
msleep†	sleep on a lock
mwakeup†	wakeup process sleeping on a lock

1.3 Signals

<signal.h>	signal definitions
sigvec	set handler for signal
kill	send signal to process
killpgrp	send signal to process group
sigblock	block set of signals
sigsetmask	restore set of blocked signals
sigpause	wait for signals
sigstack	set software stack for signals

1.4 Timing and statistics

<sys/time.h>	time-related definitions
gettimeofday	get current time and timezone
settimeofday	set current time and timezone

† Not supported in 4.3BSD.

getitimer	read an interval timer
setitimer	get and set an interval timer
profil	profile process

1.5 Descriptors

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

1.6 Resource controls

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

1.7 System operation support

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

2. System facilities

2.1 Generic operations

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

2.2 File system

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

<sys/file.h>	file system definitions
chdir	change directory
chroot	change root directory
mkdir	make a directory
rmdir	remove a directory
open	open a new or existing file
mknod	make a special file
portal†	make a portal entry

† Not supported in 4.3BSD.

unlink	remove a link
stat*	return status for a file
lstat	returned status of link
chown*	change owner
chmod*	change mode
utimes	change access/modify times
link	make a hard link
symlink	make a symbolic link
readlink	read contents of symbolic link
rename	change name of file
lseek	reposition within file
truncate*	truncate file
access	determine accessibility
flock	lock a file

2.3 Communications

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

2.4 Terminals, block and character devices

2.5 Processes and kernel hooks