The OpenBSD C particularist

Angelo Rossi <angelo.rossi.homelab@gmail.com>

July, 23 2024

Contents

Revision #	**Comment**	**Author**	**Date**
0.1	Initial Release	Angelo Rossi	28/07/2024
0.2	Adding BSD 3 clause license	Angelo Rossi	30/07/2024

Chapter 1

Preface.

This book is intended for the person who wants to become a system programmer for the UNIX-like operating system OpenBSD[1]. The most important system calls and library routines provided by this operating system are discussed and numerous examples of *real world* applications have been provided. The main focus of the discussion is on the 7.5 release of OpenBSD which is a 4.4BSD UNIX derivative. The chapters have been organized in a *bottom up* fashion, presenting first the methods and routines for performing simple tasks with basic informations fetched from the specific man page and then moving on to complex operations that build on the earlier information. At the end of the chapter or an important section code examples are presented.

- Chapter 1, Licensing., presents some introductory concept and terminology. It also briefly describes the error handling mechanism used by routines in the Standard I/O Library;

- Chapter 2, The `errno` global variable., and Chapter 3, Moving Around in Files., present the high- and low-level input and output mechanism provided for the programmer. Methods of manipulating ordinary files and directories are described in Chapter 4, Converting File Descriptors to File Pointers. and operations on special device files are presented in Chapter 5, The umask Value.;

- Chapter 6, The `select` System Call., describes how to obtain information about the users of the system.

- Chapter 7, Reading the */var/run/utmp* and */var/log/wtmp* Files., describes the method for obtaining the time of the day, as well as how to time various events;

- Chapter 8, Interval Timers., describes both the Berkeley and System V signal and interrupt mechanism;

- Chapter 9, The Signal Stack., describes methods for executing other programs, including setting up pipes, and Chapter 10, Creating Pipes Directly., describes job control mechanism for controlling those programs;

- Chapter 11, Important Points., describes sockets, shared memory, message queues and semaphore mechanisms;

[1] OpenBSD

- Chapter 12, The `shmat` and `shmdt` System Calls., describes the mechanisms for intermachine communication using TCP/IP;
- Chapter 13, Networking System Calls., provides information on the internal organization of the OpenBSD Fast File System;
- Chapter 14, Inodes., covers a variety of miscellaneous shorter topics, including reading and setting resource limits, access to environment variables, and use of `perror` for error handling.

The appendices provide information on some specialized topics that are not often used by the systems programmer, but are nevertheless good to know. Appendix A presents information on how to call FORTRAN 90 subroutines from a C program, and vice-versa. Appendix B describes the use of Workstation Console Access aka wscons. A modest background is required to understand the material in this book. The reader is expected to be fluent in C programming language ([1]) including the more advanced concepts such as structures and pointers. Good familiarity with the organization and use of the UNIX operating system is also a must. Although not necessary, familiarity with data structures and algorithms such as those used for sorting and searching will be useful. The examples in the book are really all complete, working programs that should be entered and experimented with to gain a complete understanding of the material[2]The reader should know how to use gcc or clang compiler suites as well as to use the ld linker and the lldb debugger. For the FORTRAN 90 part in Appendix A we recomend g95 fortran compiler.

1.1 Documentation Conventions.

For the most part the conventions followed in this book should be obvious, but for the sake of clarity, we'll review them here. This handbook use *Italics*, `Constant-Width` and *`Constant-Italic`* text to emphasize special words:

Italics	are used for the names of all UNIX utilities, directories and filenames, and to emphasize new terms and concepts when they are first introduced.
`Constant Width`	is used for system calls, library routines, sample code fragments and examples. A referenze in explanatory text to a word or item used in an example or code fragment is also shown in constant width font.
`Constant Italics`	are used in code fragments to represent general terms that requires context-dependent substitution.
function(n)	is a reference to a man page[3] in section n of the *OpenBSD Manual Page*. The command is `man -s function`. For example, *tty*(4) refers to a page called *tty* in Section 4: man -s 4 tty.

[2] If you have an internet access, you need not type in the examples. As you can use git program in your system to download the material, using the command:

[3] The man command can show

1.2 Notes on man on OpenBSD.

The man utility displays the *manual page* entitled name. Pages may be selected according to a specific category or section or machine architecture or subsection. Only select manuals from the specified section. The currently available sections are:

1. general commands such as tools and utilities;
2. system calls and error numbers;
3. library functions
    - 3p *perl*(1) programmer's reference guide;
4. device drivers;
5. file formats;
6. games;
7. miscellaneous information;
8. system maintenance and operation commands;
9. kernel internals.

1.3 Acknowledgements.

As reference we used material from the books on the bibliography. We would like to thank the following people:

- Dennis M. Ritchie;
- Brian W. Kernighan;
- Ken Thompson;
- Theo De Raadt;

without them the adventure would never start. Last but not least we would like to thank the reader, whose patience we hope will be rewarded by learning the material exposed in this work. I would like to thank the people at ircnow and especially jrmu who never stopped encouraging the release of this humble work.

1.4 Licensing.

Chapter 2

Introduction.

System Calls vs. Library Routines.
Versions of BSD and OpenBSD.
Error Handling.

Over the past several years, the use of the UNIX and UNIX-like operating systems and specifically OpenBSD has become widespread for server, workstation and personal computers. This is due mainly to the following factors:

1. the UNIX philosophy emphasizes building simple, compact, clear, modular, and extensible code that can be easily maintained and repurposed by developers other than its creators. It favors composability as opposed to monolithic design;

2. cheaper hardware with massive computational capabilities well suits the UNIX philosophy stated at the previous point; UNIX can be easily adapted to a variety of cpu architectures. OpenBSD can run on the following cpus[1]:

   a) alpha — Digital Alpha-based systems;
   b) amd64 — AMD64-based systems;
   c) arm64 — 64-bit ARM systems;
   d) armv7 — ARM based devices, such as BeagleBone, PandaBoard, CuBox-i, SABRE Lite, Nitrogen6x and Wandboard;
   e) hppa — Hewlett-Packard Precision Architecture (PA-RISC) systems;
   f) i386 — Standard PC and clones based on the Intel i386 architecture and compatible processors;
   g) landisk — IO-DATA Landisk systems (such as USL-5P) based on the SH4 cpu loongson Loongson 2E- and 2F-based systems, such as the Lemote Fuloong and Yeeloong, Gdium Liberty, etc.;
   h) luna88k — Omron LUNA-88K and LUNA-88K2 workstations;
   i) macppc — Apple New World PowerPC-based machines, from the iMac onwards;
   j) octeon — Cavium Octeon-based;
   k) MIPS64 — systems;
   l) powerpc64 — IBM POWER-based;
   m) PowerNV — systems;

[1] See OpenBSD Platforms

n) riscv64 — 64-bit RISC-V systems

o) sparc64 — Sun SPARC (Scalable Processor ARChitecture).

3. the opensource initiative which delivered and still is delivering, outstanding high quality software;

4. embedded computing and mobile devices are used to build cheaper and smarter portable computers. Operating Systems like UNIX and then OpenBSD are playing a central role in provide development platforms and services for this kind of computing devices[2].

In particular OpenBSD offers a lot of features such as: *code correctness* and *advanced security* for server or client use. Most of the open source projects such like OpenSMTPD, OpenSSH, etc are part of this project. Several books have been published on the use of OpenBSD and on the use of the C programming language, which is the primary language used in UNIX and of course in OpenBSD[3]. As a result, those wanting to write system programs under this operating system have had to learn the hard way: examining the source code of existing system utilities[4]. That is a good way to understand and discover things since one of the main strength of OpenBSD is the code correctness. This leads to acquire the way to do things within the system itself, leaving less space to bad implementations and bugs. The system provide, by design, the source code for ports, system utilities and programs like Xenocara[5] and the *kernel*. This book is an attempt to enhance the learning process for a beginner user: it discusses in details the use of most of the system calls and library routines available to the C programmer on the OpenBSD operating system. It is not intended as an introduction to the C programming language, nor can it really be considered an *advanced C programming guide*. Rather, it has been written for the person interested in learning to become a *system programmer* for the OpenBSD operating system. The student who wishes to work for a university computer center, a system programmer unfamiliar with UNIX and OpenBSD who must now write a program for such system and finally the researcher interested in writing his own tool to perform his work will find material presented in this book useful. The reader is expected to be fluent in C programming, including the more advanced concepts such as structures and pointers. The ideal reader will have been programming C for at least one year, and will have had at least a minimal introduction to data structures and computer algorithms such as those used for sorting and searching. Additionally the reader should know how:

1. to install programs and utilities using the package manager or ports;

2. to compile C programs using gcc, clang, the linker ld;

3. to debug programs using gdb or lldb;

4. to use a text editor such as GNU Emacs[6];

on OpenBSD. A junior in a college-level computer sciences curriculum should have no trouble with the concepts presented here. Throughout this book small, heavily commented example have been provided to demonstrate how the various routines being discussed are actually used. The reader will benefit by actually typing these examples in, compiling them, executing them and then experimenting with them in order to observe first-hand how they operate.

[2] The Reader could consider the success of the TI BeagleBone or the Raspberry PI. Unlike the Arduino, these embedded computing solutions can run a complete OS such as OpenBSD.

[3] The kernel is largely written in C and assembly language for different computing architectures.

[4] Usually OpenBSD installs source code in */usr/src* for the system utilities and kernel and under */usr/ports* the program and utilities ported from other projects.

[5] Xenocara is the X11 graphical server.

[6] GNU Emacs

2.1 System Calls vs. Library Routines.

Before discussing the *library routines* and *system calls* provided by OpenBSD system, a few words must be said. First the difference between a system call and a library routine needs to been explained. These terms are often used incorrectly: a system call is just what its name implies - **a request for the operating system to do something on behalf of the user's program**. For example `read` is a system call which ask the operating system to fill a buffer with data stored on a disk drive or other device. Since great chaos would result if everyone were able to access devices whenever they pleased, this service must be requested dealing which each device. A library routine, on the other hand, **does not usually need the operating system to perform its work**. An example of a library routine is the `sin` function, which computes the sine of an angle expressed in radians. Since this is done simply by summing a finite series, the operating system is not needed. In order to avoid confusion when the difference is unimportant, this book will use the word *routine* to describe either a system call or a library routine.

2.2 Versions of BSD and OpenBSD.

The main focus of the book is on the OpenBSD 7.5 release which is a derivative of the 4.4BSD UNIX from University of California at Berkeley[7]. The most influential of the non-Bell Laboratories and not-AT&T UNIX development groups was the University of California at Berkeley [2]. Software from Berkeley is released in **Berkeley Software Distribution** (**BSD**) - for example, as 4.3BSD. The first Berkeley VAX UNIX work was the addition to 32V of virtual memory, demand paging, and page replacement in 1979 by William Joy and Ozalp Babaoglu, to produce 3BSD. The reason for the large virtual memory space of 3BSD was the development of what at the time were large programs, such as Berkeley's Franz LISP. This memory management work convinced the Defence Advanced Research Projects Agency (DARPA) to fund the Berkeley team for the later development of a standard system (4BSD) for DARPA's contractors to use. A goal of the 4BSD project was to provide support for the DARPA Internet networking protocols, TCP/IP. The networking implementation was general enough to communicate among diverse network facilities, ranging from local networks, such as Ethernets and token rings, to long-haul networks, such as DARPA's ARPANET. The 4BSD work for DARPA was guided by a steering committe that included many notable people from both commercial and academic institutions. The culmination of the original Berkeley DARPA UNIX project was the release of 4.2BSD in 1983; further research at Berkeley produced 4.3BSD in mid-1986, The next releases included the 4.3BSD Tahoe release of June 1988 and the 4.3BSD Reno release of June 1990. These releases were primarily ports to the Computer Consoles Incorporated hardware platform. Interleaved with these releases were two unencumbered networking releases: 4.3BSD Net1 release of March 1989 and the 4.3BSD Net2 release of the June 1991. These releases extracted nonproprietary code from 4.3BSD; they could be redistributed freely in source and binary form to companies that and individuals who were not covered by a UNIX source license. The final CSRG release was to have been two version of 4.4BSD, released in June 1993. One was to have been a traditional full source and binary distribution, called 4.4BSD-Encumbered, that required the recipient to have a UNIX source license. The other was to have been a subset of the source, called 4.4BSD-Lite, that contained no licensed code and did not require the recipient to have a UNIX source license. We arrive to the first version of NetBSD (0.8) which dates back to 1993 and comes from the 4.3BSD-Lite operating system and from the 386BSD system, the first BSD port to the Intel 386 CPU. In the following years, 1994, modifications from the 4.4BSD-Lite release, the last release from the Berkeley group, were integrated into the system. The BSD branch of UNIX has had a great importance and influence on the history of UNIX-like operating systems, to which it has contributed many tools, ideas and improvements which are now standard: the vi editor, the C shell, job control, the Berkeley Fast File System, reliable signals, support for virtual

[7] OpenBSD was forked by NetBSD which is a 4.4 BSD UNIX derivative.

memory and TCP/IP, just to name a few. This tradition of research and development survives today in the BSD systems and, in particular, in OpenBSD. In December 1994, Theo de Raadt, a founding member of the NetBSD project, resigned from the core team and in October 1995, he founded OpenBSD, a new project forked from NetBSD 1.0. The initial release, OpenBSD 1.2, was made in July 1996, followed by OpenBSD 2.0 in October of the same year. Since then, the project has issued a release every six months, each of which is supported for one year. The OpenBSD project produces a freely available, multi-platform 4.4BSD-based UNIX-like operating system. It places emphasis on correctness, security, standardization, and portability. OpenBSD runs on many different hardware platforms and is thought of as the most secure UNIX-like operating system by many security professionals, as a result of the never-ending comprehensive source code audit. OpenBSD is a full-featured UNIX-like operating system available in source and binary form at no charge. It integrates cutting-edge security technology suitable for building firewalls and private network services in a distributed environment. OpenBSD sources and binary are free and all its parts have reasonable copyright terms permitting free redistribution. The current, May 2024, version of OpenBSD is the 7.5.

2.3 Error Handling.

Error conditions appear when the program perform one or more actions which are not allowed by the system. Programs are in fact lists of instructions which are sequentially executed[8]. Those instructions must comply to rules, within the operating system, to perform the task for which they are designed. For example let's consider the following example: a program wants to open a file on the disk for writing and store in it 256 MB of data in it, but the disk is full, which is a condition not depending by the program itself, we can say that it is an external factor. As the program tries to write those data, the operating system signals the problem and generates an error, forcing the program to abort the operation. Usually the operating system signals the error, after aborting the requested operation, by placing a numerical code in memory by storing it in a system-wide accessible variable and/or make the called routine returns that code to the program. All of the routines in the Standard I/O Library[9] return one of the predefined constants `EOF` or `NULL` when an error occurs. Other library routines usually return either -1 or 0 on error, depending on what the type of their return value is, altough some routines may return different values indicating one of several different errors. Unlike library routines, system calls are identical in the way they indicate that an error has occurred. Every system call returns the value -1 when an error occurs, and most return 0 on successfull completion, unless they are returning some other integer value. Further, the external integer `errno`[10] is set to a number indicating exactly which error occurred. The *values* of these errors are defined in the include file `<errno.h>` and may be easily printed out using the `perror` library routine, described in 15.8.1. On OpenBSD when a system call detects an error, it returns an integer value indicating failure, usually -1, and sets the variable `errno` accordingly. This allows interpretation of the failure on receiving a -1 and to take action accordingly. Successful calls never set `errno`; once set, it remains until another error occurs. It should only be examined after an error. Note that a number of system calls overload the meanings of these error numbers and that the meanings must be interpreted according to the type and circumstances of the call. Errors are important. Good programs are those that do not die a horrible death in the face of an unexpected error. According to Murphy's Law: if anything could go wrong, it will. Programs should be prepared for this inevitability by checking the return codes from all systems calls and library routines whose failure will cause problems and act accordingly. Nonetheless, in order to save space and emphasize the important parts of the code, many of the examples in this book do

[8] We can apply this even to a multitasking system in which threads could be considered standalone programs which execute their own instructions lists.

[9] See *stdio*(3).

[10] See *errno* (2)

not always check return codes as should be done in real life. The examples should be taken as demonstrations of the concepts being discussed, not as complete tools.

2.3.1 The errno global variable.

In the <errno.h> include file are defined the following possible values for the errno variable:

Table 2.1: List of errors.

Name	Value	strerror output	Explanation
Undefined Error	0	Undefined Error	-
EPERM	1	Operation not permitted	An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resources
ENOENT	2	No such file or directory	A component of a specified pathname did not exist, or the pathname was an empty string
ESRCH	3	No such process	No process could be found which corresponds to the given process ID
EINTR	4	Interrupted system call	An asynchronous signal, such as SIGINT or SIGQUIT, was caught by the thread during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call will seem to have returned the error condition;
EIO	5	Input/output error	Some physical input or output error occurred. This error will not be reported until a subsequent operation on the same file descriptor and may be lost (overwritten) by any subsequent errors;
ENXIO	6	Device not configured	Input or output on a special file referred to a device that did not exist, or made a request beyond the limits of the device. This error may also occur when, for example, a tape drive is not online or no disk pack is loaded on a drive
E2BIG	7	Argument list too long	The number of bytes used for the argument and environment list of the new process exceeded the limit ARG_MAX

Table 2.1: List of errors.

Name	Value	`strerror` output	Explanation
`ENOEXEC`	8	Exec format error	A request was made to execute a file that, although it has the appropriate permissions, was not in the format required for an executable file
`EBADF`	9	Bad file descriptor	A file descriptor argument was out of range, referred to no open file, or a read (write) request was made to a file that was only open for writing (reading)
`ECHILD`	10	No child processes	A `wait`, `waitid`, or `waitpid` function was executed by a process that had no existing or unwaited-for child processes
`EDEADLK`	11	Resource deadlock avoided	An attempt was made to lock a system resource that would have resulted in a deadlock situation
`ENOMEM`	12	Cannot allocate memory	The new process image required more memory than was allowed by the hardware or by system-imposed memory management constraints. A lack of swap space is normally temporary; however, a lack of core is not. Soft limits may be increased to their corresponding hard limits
`EACCES`	13	Permission denied	An attempt was made to access a file in a way forbidden by its file access permissions
`EFAULT`	14	Bad address	The system detected an invalid address in attempting to use an argument of a call
`ENOTBLK`	15	Block device required	A block device operation was attempted on a non-block device or file
`EBUSY`	16	Device busy	An attempt to use a system resource which was in use at the time in a manner which would have conflicted with the request
`EEXIST`	17	File exists	An existing file was mentioned in an inappropriate context, for instance, as the new link name in a *link*(2) function

Table 2.1: List of errors.

Name	**Value**	`strerror` **output**	**Explanation**
`EXDEV`	18	Cross-device link	A hard link to a file on another file system was attempted
`ENODEV`	19	Operation not supported by device	An attempt was made to apply an inappropriate function to a device, for example, trying to read a write-only device such as a printer
`ENOTDIR`	20	Not a directory	A component of the specified pathname existed, but it was not a directory, when a directory was expected
`EISDIR`	21	Is a directory	An attempt was made to open a directory with write mode specified
`EINVAL`	22	Invalid argument	Some invalid argument was supplied. For example, specifying an undefined signal to a *signal*(3) or *kill*(2) function
`ENFILE`	23	Too many open files in system	Maximum number of file descriptors allowable on the system has been reached and a request for an open cannot be satisfied until at least one has been closed. The *sysctl*(2) variable kern.maxfiles contains the current limit
`EMFILE`	24	Too many open files	The maximum number of file descriptors allowable for this process has been reached and a request for an open cannot be satisfied until at least one has been closed. *getdtablesize*(3) will obtain the current limit
`ENOTTY`	25	Inappropriate `ioctl` for device	A control function, see `ioctl`, was attempted for a file or special device for which the operation was inappropriate
`ETXTBSY`	26	Text file busy	An attempt was made either to execute a pure procedure, shared text, file which was open for writing by another process, or to open with write access a pure procedure file that is currently being executed

Table 2.1: List of errors.

Name	Value	strerror output	Explanation
EFBIG	27	File too large	The size of a file exceeded the maximum. The system-wide maximum file size is 2^{63} bytes. Each file system may impose a lower limit for files contained within it
ENOSPC	28	No space left on device	A write to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks were available on the file system, or the allocation of an i-node for a newly created file failed because no more i-nodes were available on the file system
ESPIPE	29	Illegal seek	An lseek function was issued on a socket, pipe or FIFO
EROFS	30	Read-only file system	An attempt was made to modify a file or create a directory on a file system that was read-only at the time
EMLINK	31	Too many links	The maximum allowable number of hard links to a single file has been exceeded, see pathconf for how to obtain this value
EPIPE	32	Broken pipe	A write on a pipe, socket or FIFO for which there is no process to read the data
EDOM	33	Numerical argument out of domain	A numerical input argument was outside the defined domain of the mathematical function
ERANGE	34	Result too large	A result of the function was too large to fit in the available space, perhaps exceeded precision
EAGAIN	35	Resource temporarily unavailable	This is a temporary condition and later calls to the same routine may complete normally
EINPROGRESS	36	Operation now in progress	An operation that takes a long time to complete, such as a connect, was attempted on a non-blocking object, see fcntl
EALREADY	37	Operation already in progress	An operation was attempted on a non-blocking object that already had an operation in progress

Table 2.1: List of errors.

Name	Value	strerror output	Explanation
ENOTSOCK	38	Socket operation on non-socket	Self-explanatory
EDESTADDRREQ	39	Destination address required	A required address was omitted from an operation on a socket
EMSGSIZE	40	Message too long	A message sent on a socket was larger than the internal message buffer or some other network limit
EPROTOTYPE	41	Protocol wrong type for socket	A protocol was specified that does not support the semantics of the socket type requested. For example, you cannot use the Internet UDP protocol with type SOCK_STREAM
ENOPROTOOPT	42	Protocol not available	A bad option or level was specified in a getsockopt or setsockopt call
EPROTONOSUPPORT	43	Protocol not supported	The protocol has not been configured into the system or no implementation for it exists
ESOCKTNOSUPPORT	44	Socket type not supported	The support for the socket type has not been configured into the system or no implementation for it exists
EOPNOTSUPP	45	Operation not supported	The attempted operation is not supported for the type of object referenced. Usually this occurs when a file descriptor refers to a file or socket that cannot support this operation, for example, trying to accept a connection on a datagram socket
EPFNOSUPPORT	46	Protocol family not supported	The protocol family has not been configured into the system or no implementation for it exists
EAFNOSUPPORT	47	Address family not supported by protocol family	An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use NS addresses with Internet protocols
EADDRINUSE	48	Address already in use	Only one usage of each address is normally permitted
EADDRNOTAVAIL	49	Can't assign requested address	Normally results from an attempt to create a socket with an address not on this machine

Table 2.1: List of errors.

Name	Value	`strerror` output	Explanation
`ENETDOWN`	50	Network is down	A socket operation encountered a dead network
`ENETUNREACH`	51	Network is unreachable	A socket operation was attempted to an unreachable network
`ENETRESET`	52	Network dropped connection on reset	The host you were connected to crashed and rebooted
`ECONNABORTED`	53	Software caused connection abort	A connection abort was caused internal to your host machine
`ECONNRESET`	54	Connection reset by peer	A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot
`ENOBUFS`	55	No buffer space available	An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full
`EISCONN`	56	Socket is already connected	A `connect` request was made on an already connected socket; or, a `sendto` or `sendmsg` request on a connected socket specified a destination when already connected
`ENOTCONN`	57	Socket is not connected	A request to send or receive data was disallowed because the socket was not connected and, when sending on a datagram socket, no address was supplied
`ESHUTDOWN`	58	Can't send after socket shutdown	A request to send data was disallowed because the socket had already been shut down with a previous `shutdown` call
`ETOOMANYREFS`	59	Too many references: can't splice	Not used in OpenBSD
`ETIMEDOUT`	60	Operation timed out	A `connect` or `send` request failed because the connected party did not properly respond after a period of time. The timeout period is dependent on the communication protocol
`ECONNREFUSED`	61	Connection refused	No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host

Table 2.1: List of errors.

Name	Value	`strerror` output	Explanation
`ELOOP`	62	Too many levels of symbolic links	A pathname lookup involved more than 32 (`SYMLOOP_MAX`) symbolic links
`ENAMETOOLONG`	63	File name too long	A component of a pathname exceeded 255 (`NAME_MAX`) characters, or an entire pathname (including the terminating `NUL`) exceeded 1024 (`PATH_MAX`) bytes
`EHOSTDOWN`	64	Host is down	A socket operation failed because the destination host was down
`EHOSTUNREACH`	65	No route to host	A socket operation was attempted to an unreachable host
`ENOTEMPTY`	66	Directory not empty	A directory with entries other than '.' and '..' was supplied to a remove directory or `rename` call
`EPROCLIM`	67	Too many processes	Self-explanatory
`EUSERS`	68	Too many users	The quota system ran out of table entries
`EDQUOT`	69	Disk quota exceeded	A `write` to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted, or the allocation of an i-node for a newly created file failed because the user's quota of i-nodes was exhausted
`ESTALE`	70	Stale NFS file handle	An attempt was made to access an open file on an NFS file system which is now unavailable as referenced by the file descriptor. This may indicate the file was deleted on the NFS server or some other catastrophic event occurred
`EREMOTE`	71	Too many levels of remote in path	Self-explanatory
`EBADRPC`	72	RPC struct is bad	Exchange of `rpc` information was unsuccessful
`ERPCMISMATCH`	73	RPC version wrong	The version of `rpc` on the remote peer is not compatible with the local version
`EPROGUNAVAIL`	74	RPC program not available	The requested `rpc` program is not registered on the remote host

Table 2.1: List of errors.

Name	Value	strerror output	Explanation
EPROGMISMATCH	75	Program version wrong	The requested version of the rpc program is not available on the remote host
EPROCUNAVAIL	76	Bad procedure for program	An rpc call was attempted for a procedure which doesn't exist in the remote program
ENOLCK	77	No locks available	A system-imposed limit on the number of simultaneous file locks was reached
ENOSYS	78	Function not implemented	Attempted a system call that is not available on this system
EFTYPE	79	Inappropriate file type or format	The file contains invalid data or set to invalid modes
EAUTH	80	Authentication error	Attempted to use an invalid authentication ticket to mount a NFS filesystem
ENEEDAUTH	81	Need authenticator	An authentication ticket must be obtained before the given NFS file system may be mounted
EIPSEC	82	IPsec processing failure	IPsec subsystem error. Not used in OpenBSD
ENOATTR	83	Attribute not found	A UFS Extended Attribute is not found for the specified pathname
EILSEQ	84	Illegal byte sequence	An illegal sequence of bytes was used when using wide characters
ENOMEDIUM	85	No medium found	Attempted to use a removable media device with no medium present
EMEDIUMTYPE	86	Wrong medium type	Attempted to use a removable media device with incorrect or incompatible medium
EOVERFLOW	87	Value too large to be stored in data type	A numerical result of the function was too large to be stored in the caller provided space
ECANCELED	88	Operation canceled	The requested operation was canceled
EIDRM	89	Identifier removed	An IPC identifier was removed while the current thread was waiting on it
ENOMSG	90	No message of desired type	An ipc message queue does not contain a message of the desired type, or a message catalog does not contain the requested message

Table 2.1: List of errors.

Name	Value	`strerror` output	Explanation
`ENOTSUP`	91	Not supported	The operation has requested an unsupported value
`EBADMSG`	92	Bad message	A corrupted message was detected
`ENOTRECOVERABLE`	93	State not recoverable	The state protected by a robust mutex is not recoverable
`EOWNERDEAD`	94	Previous owner died	The owner of a robust mutex terminated while holding the mutex lock
`EPROTO`	95	Protocol error	A device-specific protocol error occurred

In the 15.8.3 subsection it is showed a way to print out the errors names along their codes.

Chapter 3

The Standard I/O Library.

A programmer learning C is forced to use the routines in the Standard I/O Library, called stdio, to perform simple input and output from console, in order to write programs that can interact with a user. In fact these are the first routines that we would learn reading the book from Brian W. Kernighan and Dennis M. Ritchie on C programming language ([1]). These routines perform three important functions:

- *buffering* – is performed automatically. Rather than reading or writing data a few bytes at a time, the routines perform the actual input or output in large *chunks* of several thousand bytes at time. The size of the buffer is generally specified by the constant `BUFSIZ`, defined in the include file `<stdio.h>`. The routines seem to read or write in a small units, but the data is actually saved in a buffer. This buffering is internal to the routines, and is invisible to the programmer;

- *input and output conversions* – are perfomed. For example, when using the `printf`[1] routine to print an integer, with `%d`, the character representation of that integer is actually printed. Similarly, when using `scanf`[2], the character representation of an integer is converted into its numeric value;

- *input and output are automatically formatted* – that is, it is possible to use field widths and the like to print numbers and strings in any desired format.

This chapter provides a review of the more commonly used routines contained in the Standard I/O Library.

3.1 File Pointers.

In the Standard I/O Library, a file is called a *stream*, it is described by a pointer to an object of type `FILE`, called a *file pointer*. The `FILE` data type is defined in the the file `<stdio.h>`, which has to

[1] See *printf*(3).

[2] See *scanf*(3).

be included[3] before using any of the stdio routines. There are three predefined file pointers: `stdin`, `stdout` and `stderr`. These refer to the *standard input*, the console, the *standard output* which is the terminal screen and the *standard error* stream respectively their documentation could be found in *stdin*(3). Most of the stdio routines require that a file pointer referring to an open stream be passed to them. However, when reading from the standard input or writing to the standard output, stdio provides *shorthand* routines that assume one of these streams rather than requiring them to be specified. Table 3.1 shows these routines.

Shorthand	**Equivalent**
`getchar()`	`fgetc(stdin)`,`getc(stdin)`
`gets(`*buf*`)`	`fgets(`*buf*,`BUFSIZ`,`stdin)`
`printf(`*args*`)`	`fprintf(stdout,`*args*`)`
`putchar(`*c*`)`	`fputc(`*c*`,stdout)`,`putc(`*c*`,stdout)`
`puts(`*buf*`)`	`fputs(`*buf*, `stdout)`
`scanf(`*args*`)`	`fscanf(stdin,`*args*`)`

Table 3.1: Shorthand routines for standard input and output.

3.2 Opening and Creating Files.

In order to read from or write to a stream, this must first be opened for the desired operation. The `fopen`[4] routine is used for this purpose. It takes two arguments: a character string containing the complete path name of the file to open and a character string describing how that file should be opened. It returns a pointer to and open stream of type `FILE` or the constant `NULL` if the stream could not be opened. The second argument to `fopen` may take on one of the following string values:

`"r"` or `"rb"` open file for reading;

`"r+"` or `"rb+"`, `"r+b"` open for reading and writing;

`"w"` or `"wb"` open for writing. The file is created if it does not exist, otherwise it is truncated;

`"w+"` or `"wb+"`, `"w+b"` open for reading and writing. The file is created if it does not exist, otherwise it is truncated;

`"a"` or `"ab"` open for writing. The file is created if it does not exist;

`"a+"` or `"ab+"`, `"a+b"` open for reading and writing. The file is created if it does not exist.

The letter `"b"` in the mode strings above is strictly for compatibility with ANSI X3.159-1989 ("ANSI C89") and has no effect; the `"b"` is ignored. After any of the above prefixes, the mode string can also include zero or more of the following:

`"e"` the *close-on-exec* flag is set on the underlying file descriptor of the new file;

`"x"` if the mode string starts with `"w"` or `"a"` then the function shall fail if the file specified by path already exists, as if the `O_EXCL` flag was passed to the *open*(2) function. It has no effect if used with *fdopen*() or the mode string begins with `"r"`.

[3] using the directive `#include <stdio.h>` at the top of the C program.

[4] See *fopen*(3).

3.3 Flushing files.

Sometimes it is important to *flush* data from the buffer especially during critical code execution or errors. To force a flush of data present in the stdio buffer two routines could be used: `fflush` and `fpurge`. The function `fflush` forces a write of all buffered data for the given output or update stream via the stream's underlying write function. The open status of the stream is unaffected. If the stream argument is NULL, `fflush` flushes all open output streams. The function `fpurge` erases any input or output buffered in the given stream. For output streams this discards any unwritten output. For input streams this discards any input read from the underlying object but not yet obtained via *getc*(3); this includes any text pushed back via *ungetc*(3).

3.4 Closing files.

The `fclose`[5] routine is used to close an open stream. `fclose` takes a single argument, the file pointer referring to the stream to be closed. When called, this routine flushes the buffers for the stream and performs some other internal cleanup functions. 0 is returned on success; the constant `EOF` is returned if an error occurs.

3.5 Reading and Writing Files.

The Standard I/O Library provides several ways to read and write data to and from a file.

3.5.1 The `getc` and `putc` Routines.

The simplest way to read and write data is one character or byte at a time. This is done by using the `getc`[6] and `putc`[7] routines. `getc` accepts a single argument, a file pointer referring to a stream open for reading. It returns the next character read from the stream, or the constant `EOF` when the end of file has been reached. `putc` accepts two arguments, a character to be written and a file pointer referring a stream open for writing. It places that character onto the stream and returns 0 if it succeeds or `EOF` if an error occurs. Listing 3.1 shows a small program that appends one file onto another. The first argument specifies the name of the file to be copied, and the second file specifies the name of a file to be appended to. If the file to be appended to does not exist, it will be created.

Listing 3.1: append-char - append one file to another character by character.

```c
/* -*- mode: c-mode; -*- */

/* append-char.c file. */
#include <stdio.h>
#include <stdlib.h>

/* append-char program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
```

[5] See *fclose*(3).

[6] See *getc*(3).

[7] See *putc*(3).

```
  int c;
  long int ret = EXIT_FAILURE;
  FILE *from, *to;

  /* Check our arguments. */
  if(argc == 3) {

    /* Open the from-file for reading. */
    if((from = fopen(argv[ 1 ], "r")) != NULL) {

      /*
       * Open the to-file for appending.  If to-file does
       * not exist, fopen will create it.
       */
      if((to = fopen(argv[ 2 ], "a")) != NULL) {

        /*
         * Now read characters from from-file until we
         * hit end-of-file, and put  them onto to-file.
         */
        while((c = getc(from)) != EOF)
          putc(c, to);

        /* Now close the output file. */
        if(fclose(to) == 0)
          ret = EXIT_SUCCESS;
        else
          perror("Error␣closing␣output␣file");
      } else
        perror(argv[ 2 ]);

      /* Now close the input file. */
      fclose(from);
    } else
        perror(argv[ 1 ]);
  } else
    fprintf(stderr, "usage:␣%s␣from-file␣to-file\n", *argv);
  exit(ret);
}

/* End of append-char.c file. */
```

For brevity and to emphasize the information being discussed in this chapter, Listing 3.1 and the following examples, violates one of the more important UNIX conventions. This convention dictates that **in any program where it makes sense, the program should operate on both named files or on its standard input and output**. The text formatting programs *cat*, *egrep*, *tbl* and *eqn*, to name a few, are good examples of programs that do this. Given a list of file names, these programs will open the files and process the data in them. However, if no file names are given, these programs will read data from their standard input. This allows the programs to operate as filters, so they can be invoked individually or as a part pipeline, see Chapter 9, The Signal Stack..

3.5.2 The fgets and fputs Routines.

Another way to read and write files provided by the Standard I/O Library allows the programmer to process data a *line* at the time. A line is defined by a string of zero or more characters terminated by a new-line character '\n'[8]. The fgets[9] function accept three arguments: a pointer to a character buffer to be filled, an integer specifying the size of the buffer and a file pointer referring to a stream open for reading. A pointer to the filled buffer is returned on success or the constant NULL is returned when end-of-file, EOF, is reached. The buffer will be filled with one line of characters, including the new-line, '\n', character and will be terminated with a null character, '\0'. fputs[10] accepts two arguments, a pointer to a null-terminated string of characters and a file pointer referring to a stream open for writing. It returns 0 on success, or the constant EOF if an error occurs. Listing 3.2 shows another version of our program to append one file to another; this version does it a line at a time. The constant BUFSIZ is defined in the include file <stdio.h> and is configured to be an optimum size for the system. Unless you need a particular size, this is a good value to use whenever you are working with stdio.

Listing 3.2: append-line - append one file to another line by line.

```
/* -*- mode: c-mode; -*- */

/* append-line.c file. */
#include <stdio.h>
#include <stdlib.h>

/* append-line program. */
/* Functions prototype. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  FILE *from, *to;
  char line[ BUFSIZ ];
  long int ret = EXIT_FAILURE;

  /* Check our arguments. */
  if(argc == 3) {

    /* Open the from-file for reading. */
    if((from = fopen(argv[ 1 ], "r")) != NULL) {

      /*
      * Open the to-file for appending.  If to-file does
      * not exist, fopen will create it.
      */
      if((to = fopen(argv[ 2 ], "a")) != NULL) {

        /*
         * Now read a line at a time from from-file
         * and write it to the to-file.
```

[8] A typical example of such a file is */usr/share/dict/words* which holds an english words dictionary. See the example code in listing 3.9.

[9] See *fgets*(3).

[10] See *fputs*(3).

```
         */
        while(fgets(line, BUFSIZ, from) != NULL)
          fputs(line, to);

        /* Now close output file. */
        fclose(to);

        /* Signal no errors to the shell. */
        ret = EXIT_SUCCESS;
      } else
        perror(argv[ 2 ]);

      /* Now close input file. */
      fclose(from);
    } else
      perror(argv[ 1 ]);
  } else
    fprintf(stderr, "usage:␣%s␣from-file␣to-file\n", *argv);
  exit(ret);
}

/* End of append-line.c file. */
```

3.5.3 The `fread` and `fwrite` Routines.

The Standard I/O Library also provides a method to read and write data without dividing it up into characters or lines. This is usually desirable when working with files that do not consist only of text, but also include arbitrary binary data. The `fread`[11] function accepts four arguments: a pointer to an array of some data type[12], an integer indicating the size of one array element in bytes, an integer indicating the number of array elements to read and a file pointer referring to a stream open for reading. It returns the number of array elements actually read in, or 0 on end-of-file. The `fwrite`[13] function also accepts four arguments, as described above for `fread`. It returns the number of array elements actually written, or 0 on error. The advantage to using a routine like `fread` and `fwrite` lies primarily in the ability to impose a structure on the input or output stream not provided by the stdio routines themselvers. For example, if a file contains 100 binary floating-point numbers, the easiest way to read these in would be to use something like the code segment shown below:

```
FILE *fp;
float numbers[ 100 ];

......

fread(numbers, sizeof(float), 100, fp);

......
```

Listing 3.3 shows still another version of our file appending program; this version copies the data a buffer-full of characters at a time.

[11] See *fread*(3).

[12] Could be an array of characters, integers, structures and so on.

[13] See *fwrite*(3).

Listing 3.3: append-buf - append one file to another buffer-full at a time.

```c
/* -*- mode: c-mode; -*- */

/* append-buf.c file. */
#include <stdio.h>
#include <stdlib.h>

/* append-buf program. */
/* Functions prototypes. */
int main(int, char *[]);

/* The main function. */
int main(int argc, char *argv[])
{
  int n;
  FILE *from, *to;
  char buf[ BUFSIZ ];
  long int ret = EXIT_FAILURE;

  /* Check our arguments. */
  if(argc == 3) {

    /* Open the from-file for reading. */
    if((from = fopen(argv[ 1 ], "r")) != NULL) {

      /*
       * Open the to-file for appending.  If to-file does
       * not exist, fopen will create it.
       */
      if((to = fopen(argv[ 2 ], "a")) != NULL) {

        /*
         * Now read a buffer-full at a time from from-file
         * and write it to the to-file.
         */
        while((n = fread(buf, sizeof(char), BUFSIZ, from)) > 0)
          fwrite(buf, sizeof(char), n, to);

        /* Now close the output file. */
        fclose(to);

        /* Signal no errors to the shell. */
        ret = EXIT_SUCCESS;
      } else
        perror(argv[ 2 ]);

      /* Now close the input file. */
      fclose(from);
    } else
      perror(argv[ 1 ]);
  } else
    fprintf(stderr, "usage:␣%s␣from-file␣to-file\n", *argv);
    exit(ret);
}

/* End of append-buf.c file. */
```

3.5.4 The `fscanf` and `fprintf` Routines.

Other than dividing data into units of characters or lines, the routines described in the previous sections do not interpret the data they manipulate. Sometimes however, more interpretation of the data is necessary. As the reader probably knows, the internal representation of data in the computer is not generally human-readable. For example, the number 10 is represented internally as binary value:

$$n_{10} = 10_{10} = 00001010_2 = 0b00001010$$

However, when this number is to be printed on a line printer or terminal screen, it must be converted to the two ASCII characters `'1'` and `'0'`, which have the following bit patterns:

$$'1' : \qquad 0b00110001$$

$$'0' : \qquad 0b00110000$$

Likewise, in order to read in a number from the console, the characters that represent that number to a human must be converted into the internal representation of that number in order for the computer to deal with it. The `fscanf`[14] routine accepts a variable number of arguments. The first argument is a file pointer referring to a stream open for reading, in the case of the console the programmer have to use the *standard input stream*, called `stdin`. The second argument is a character string that specifies the format of the input data. The rest of arguments are pointers to the data objects that are to be filled. `fscanf` reads character from the stream, converts them into various internal representation as specified by the format string and stores them in the data objects. The format string may contain:

- blanks, tabs and new-line characters, which match optional white space in the input;
- an ordinary character, other than `'%'`, which must match the next input character;
- a conversion specification, consisting of a `'%'` character followed by a conversion character.

A conversion specification indicates how the next input field is to be interpreted; the result is placed in the corresponding argument. Some of the more common conversion characters are:

`d` decimal integer is expected; the corresponding argument should be a pointer to an integer;

`f` floating-point number is expected; the corresponding argument should be a pointer to an object of type `float`;

`l` indicates either that the conversion will be one of dioux or n and the next pointer is a pointer to a `long int`, rather than `int`, or that the conversion will be one of efg and the next pointer is a pointer to `double`, rather than `float`, or that the conversion will be one of sc[. If the conversion is one of sc[, the expected conversion input is a multibyte character sequence. Each multibyte character in the sequence is converted with a call to the `mbrtowc` function. The field width specifies the maximum amount of bytes read from the multibyte character sequence and passed to `mbrtowc` for conversion. The next pointer is a pointer to a `wchar_t` wide-character buffer large enough to accept the converted input sequence including the terminating NUL wide character which will be added automatically;

[14] See *fscanf*(3).

ll Indicates that the conversion will be one of dioux or n and the next pointer is a pointer to a long long int, rather than int;

p matches a pointer value (as printed by ‘%p’ in printf(3)); the next pointer must be a pointer to void;

s character string is expected; the corresponding argument should point to a character array or a character buffer large enough to hold the string plus a terminating null character. The input field is terminated by a space or a new-line character.

For example to read in the string:

```
123 Hello 45.678
```

the call:

```
fscanf(stdin, "%d␣%s␣%f", &intvar, stringvar, &floatvar);
```

could be used. fscanf returns the number of input items matched or the constant EOF when end-of-file has been reached. The fprintf[15] routine also accepts a variable number of arguments. The first argument is a file pointer to a stream open for writing, the second is again a format string and the following arguments are the objects to be printed. Ordinary , non-'%', characters in the format string are copied to the output stream. A '%' character specifies that the corresponding argument is to be converted; the conversion characters are the same as those described for fscanf. Listing 3.4 shows a small program that asks the reader to enter an integer number and then computes the factorial[16] of that positive integer number and prints it out. This example uses the printf and scanf routines, which assume the use of streams stdout and stdin, rather than requiring the streams to be passed as arguments.

Listing 3.4: factorial - compute the factorial of an integer number.

```
/* -*- mode: c-mode; -*- */

/* factorial.c file. */
#include <stdio.h>
#include <stdlib.h>

/* factorial program. */
/* Function prototypes. */
int main(int, char *[]);
unsigned long int factorial(unsigned long int);
```

[15] See *fprintf*(3).

[16] The factorial of a positive integer number n is:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

where:

$$0! = 1$$

by definition. The problem with such computation is that an unsigned long int can hold a value:

$$0 \leq n \leq 2^{64} - 1 = 18446744073709551615UL$$

let’s compare this number with the nearest factorials:

$$20! < 18446744073709551615 < 21!$$

$$2432902008176640000 < 18446744073709551615 < 51090942171709440000$$

we can see that the number on the right is far beyond the unsigned long int capacity but the number on the left is not. The program practically can compute a factorial of a positive integer number that span from 0 to 20.

```

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  unsigned long int n, m;

  /*
   * Messaging the user to enter the integer
   * number.
   */
  printf("Enter␣an␣integer␣number:␣");
  scanf("%lu", &n);
  if(n <= 20ul) {
    m = factorial(n);
    printf("The␣factorial␣of␣%lu␣is␣%lu.\n", n, m);
    ret = EXIT_SUCCESS;
  } else
    perror("n␣must␣be␣a␣positive␣integer␣between␣0␣and␣20");
  exit(ret);
}

/*
 * The factorial function which computes
 * n! = 1 * 2 * 3 * ... * n
 */
unsigned long int factorial(unsigned long int n)
{
  /* computes n! */
  if((n == 0ul) || (n == 1ul))
    return 1ul;
  else
    return (n * factorial(n - 1));
}

/* End of factorial.c file. */
```

Note the use of the two constants EXIT_SUCCESS which is set to 0 and EXIT_FAILURE which is set to non zero value. They are specifically defined in <stdlib.h> to be used in the exit function.

3.5.5 The sscanf and sprintf Routines.

stdio also provides the ability to *print* formatted data into a character string and to *read* formatted data from a character string. The sscanf[17] and sprintf[18] routines are identical to fscanf and fprintf, except that instead of taking a file pointer to a stream as their first argument, they take a character string or a character buffer. sscanf will copy characters from the character string or buffer, converting them according to its second argument. sprintf will place a formatted copy of its argument into the character string or buffer. However sprintf function should be used carefully: let's consider the following example:

```
char buf[ 2 ];
```

[17] See *sscanf*(3).

[18] See *sprintf*(3).

```
unsigned int longvar = 65535;

......

sprintf(buf, "%d", longvar);

......
```

the string array is not big enough to hold the characters needed to print "65535". In this case executing the routine could lead to a catastrophic data corruption as memory is overlapped by `sprintf`. This behaviour is understandable since those routines have no idea of the length of the buffer to write in. To avoid this problem, stdio, provides a safe variant to `sprintf`: `snprintf`. It takes a variable number of arguments just as `sprintf`, but the first argument is the character string or the buffer, the second is the character string or buffer size, the rest are the same as the `sprintf` routine. `snprintf` composes a string with the same text that would be printed if the format string was used on `printf`, but instead of being printed, the content is stored as a character string in the buffer pointed by the first argument, taking the second argument as the maximum buffer capacity to fill. If the resulting string would be longer than the second argument value, the remaining characters are discarded and not stored, but counted for the value returned by the function. A terminating null character is automatically appended after the content written. After the format string argument, the function expects at least as many additional arguments as needed for format string.

Listing 3.5: snprintf - snprintf test program.

```c
/* -*- mode: c-mode; -*- */

/* snprintf.c file. */
#include <stdio.h>
#include <stdlib.h>

/* snprintf test program. */
#define MAXLENGTH 16

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  char divina[ MAXLENGTH + 1 ];
  char commedia[] = "Nel␣mezzo␣del␣cammin␣di␣nostra␣vita␣mi␣
      ritrovai␣per␣una␣selva␣oscura...";
  long int ret = EXIT_FAILURE;

  snprintf(divina, MAXLENGTH, "%s", commedia);
  printf("Source␣string:␣%s\n", commedia);
  printf("Destination␣string:␣%s\n", divina);
  exit(EXIT_SUCCESS);
}

/* End of snprintf.c file. */
```

3.6 Moving Around in Files.

It is often necessary to move to a specific location in a file before reading or writing data. For example, if a file contains several fixed-size items indexed by number, it may be easier to skip over unwanted records to read or write the desired record, rather than reading and processing all the records preceding the desired one. The Standard I/O Library routine for moving around in a file is called `fseek`[19]. It accepts three arguments: a file pointer to an open stream, a long integer specifying the number of bytes to move, called an *offset*, and an integer indicating from where in the file the offset is to be taken. If the third argument is `SEEK_SET`, the offset is taken to the beginning of the file. If it is `SEEK_CUR`, the offset is taken from the current location in the file. If the third argument is `SEEK_END`, the offset is taken from the end of the file. To move at the end of a file, the call:

```
FILE *fp;

......

fseek(fp, 0L, SEEK_END);
```

should be used. To move at the beginning of the file, the call:

```
......

fseek(fp, 0L, SEEK_SET);
```

may be used or equivalently, the `rewind` routine may be used. `rewind` takes a single argument, a file pointer to an open stream. To find out the current location in a file, the `ftell` routine should be used. `ftell` accepts a single argument, a file pointer to an open stream and returns a long integer indicating the offset from the beginning of the file.

Listing 3.6 shows a small program that creates a data file with one record for each of five users. In order to demonstrate the use of `fseek`, the program writes the file backwards; that is, the last record is written first and the first record is written last. This is somewhat pointless in practice, but serves to demonstrate the appropriate concepts. The reader should enter this program and execute it. Then try to write a program that will read the records from the file in the order 3, 0, 2, 1, 4 and print them out:

Listing 3.6: fseekdemo - demontrate the use of the fseek routine.

```
/* -*- mode: c-mode; -*- */

/* fseekdemo.c file. */
#include <stdio.h>
#include <stdlib.h>

/* fseekdemo program. */
/* structure and type definition. */
struct tagRecord {
  int uid;
  char login[ 9 ];
};

typedef struct tagRecord record_t;

/* Global variables. */
```

[19] See *fseek*(3).

```
record_t records[ 5 ] = {
  { 1, "user1" },
  { 2, "user2" },
  { 3, "user3" },
  { 4, "user4" },
  { 5, "user5" }
};

/* Function prototypes. */
int putRecord(FILE *, int, record_t *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i;
  long int ret = EXIT_FAILURE;
  FILE *fp;

  /* Open the data file for writing. */
  if((fp = fopen("datafile.dat", "w")) != NULL) {

    /* For each user, going backwards... */
    for(i = 4; i >= 0; i--) {
      printf("writing␣record␣#%d\n", i);

      /*
       * Output the record.  Notice we pass the address
       * of the structure.
       */
      if(putRecord(fp, i, &records[ i ]) == EXIT_FAILURE) {
        perror("Could␣not␣write␣record.\n");
        break;
      }
    }
    if(i == 0)
      ret = EXIT_SUCCESS;
    fclose(fp);
  } else
    perror("Could␣not␣open␣datafile.dat␣for␣writing.\n");
  exit(ret);
}

int putRecord(FILE *fp, int i, record_t *r)
{
  int ret = EXIT_FAILURE;

  /*
   * Seek to the i-th position from the beginning
   * of the file.
   */
  if(fp) {
    if(r) {
      if(fseek(fp, (long) (i * sizeof(record_t)), SEEK_SET) == 0)

        /*
         * Write the record.  We want to write  one
         * object the size of a record structure.
         */
        if(fwrite((char *) r, sizeof(record_t), 1, fp) == 1)
          ret = EXIT_SUCCESS;
    }
  }
  return ret;
}

/* End of fseekdemo.c file. */
```

To read back records from file just add to the previous program some code. In particular we have the following. Listing 3.7 reads back records in specified order.

Listing 3.7: fseekreadback - demontrate the use of the fseek routine to read back records.

```
/* -*- mode: c-mode; -*- */

/* fseekreadback.c file. */
#include <stdio.h>
#include <stdlib.h>

/* fseekreadback program. */
/* structure and type definitions. */
struct tagRecord {
  int uid;
  char login[ 9 ];
};

typedef struct tagRecord record_t;

/* Global variables. */
int positions[ 5 ] = { 3, 0, 2, 1, 4 };
record_t records[ 5 ];

/* Function prototypes. */
int getRecord(FILE *, int, record_t *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i;
  long int ret = EXIT_FAILURE;
  FILE *fp;
  record_t rec;

  /* Open the data file for reading. */
  if((fp = fopen("datafile.dat", "r")) != NULL) {
    /* For each position read back the corresponding user. */

    for(i = 0; i < 5; i++) {

      /*
       * Output the record.  Notice we pass the address
       * of the structure.
       */
      if(getRecord(fp, positions[ i ], &rec) != EXIT_FAILURE)
        printf("position:␣%d,␣uid:␣%d,␣login:␣%s\n", positions[ i ], rec.uid, rec.login);
      else {
        perror("Could␣not␣read␣record.\n");
        break;
      }
      if(i == 5)
        ret = EXIT_SUCCESS;
    }

    /* Now close the output file. */
    fclose(fp);
  } else
    perror("Could␣not␣open␣datafile.dat␣for␣reading.\n");
  exit(ret);
}

/*
 * getRecord -- get a record from a file.
 */
int getRecord(FILE *fp, int i, record_t *r)
{
  int ret = EXIT_FAILURE;

  /*
   * Seek to the i-th position from the beginning
   * of the file.
   */
  if(fp) {
    if(r) {
      if(fseek(fp, (long) (i * sizeof(record_t)), SEEK_SET) == 0) {

        /*
         * Write the record.  We want to write  one
         * object the size of a record structure.
         */
        if(fread((void *) r, sizeof(record_t), 1, fp) == 1) {
          ret = EXIT_SUCCESS;
        }
      }
    }
  }
  return ret;
}

/* End of fseekreadback.c file. */
```

As the reader may check, both putRecord and getRecord routines return a value. This is necessary to tell the program the outcome of the operation that should be performed. In fact this is a good way to tell to the calling program if something went wrong. The possible return values for both routines are EXIT_SUCCESS on succesful operation and EXIT_FAILURE on error. The following listings 3.8 and 3.9 belongs to the same program. The .h file contains the definitions and prototypes for the .c source code.

Listing 3.8: find-word - program to show the usage of fgets and /usr/share/dict/words file (include file).

```
/* -*- mode: c-mode; -*- */

/* find-word.h file. */
#ifndef __FIND_WORD_H
#define __FIND_WORD_H

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

#include "list.h"

#define FOREVER for(;;)
#define MAXINT 20
#define DEFAULT_DICTIONARY_PATH "/usr/share/dict/words"
#define min(a, b) ((a) < (b) ? (a) : (b))

/* Types. */

/* Functions prototype. */
void lowerize(char *, size_t);
void printArray(char *, void *, size_t);
size_t factorial(size_t);
size_t binomial(size_t, size_t);
void combinations(size_t *, size_t, size_t, size_t **);
long int getCombString(char *, char *, size_t *, size_t);
size_t **allocateCombs(size_t, size_t);
long int deallocateCombs(size_t **);
char *intersect(char *, char *, size_t);
bool cmp(void *, ...);
int main(int, char *[]);

#endif /* __FIND_WORD_H */

/* End if find-word.h file. */
```

Listing 3.9: find-word - program to show the usage of fgets and /usr/share/dict/words (source code file).

```
/* -*- mode: c-mode; -*- */

/* find-word.c file. */
#include "find-word.h"

/* find-word program. */

/* Main function. */
int main(int argc, char *argv[])
{
  char ch, dictionary_path[ BUFSIZ ], combstr[ BUFSIZ ];
  char line[ BUFSIZ ], *letters, *word;
  bool found, *combstbl;
  long int ret = EXIT_FAILURE;
```

```c
  FILE *dict_file;
  size_t i, j, k, l, m, n;
  size_t combs_count, count, letters_count, chars_count;
  size_t *indices, **combs;
  list_t *words_list = NULL;

  /* Check our arguments. */
  switch(argc) {
  case 2:
    strncpy(dictionary_path, DEFAULT_DICTIONARY_PATH, BUFSIZ);
    letters = argv[ 1 ];
    break;

  case 3:
    strncpy(dictionary_path, argv[ 1 ], BUFSIZ);
    letters = argv[ 2 ];
    break;

  default:
    letters = NULL;
    fprintf(stderr, "usage:␣find-word␣<dictionary-file>␣<letters>,␣find-word␣<letters>\n");
    break;
  }
  if(letters) {

    /* force all characters in the string to be lower case. */
    lowerize(letters, MAXINT);
    letters_count = strnlen(letters, MAXINT);
    if(letters_count < 21) {

      /* Open system dictionary file. */
      if((dict_file = fopen(dictionary_path, "r")) != NULL) {

        /*
         * loop starting from same length for words as
         * the entered sets of characters.
         */
        for(count = letters_count; count >= 3; count--) {

          /* computes the number of the character combinations. */
          combs_count = binomial(letters_count, count);
          printf("combinations␣count:␣%ld\n", combs_count);
          combs = allocateCombs(combs_count, count);
          if(combs) {
            indices = (size_t *) calloc(count, sizeof(size_t));
            if(indices) {

              /* generate all character combinations without repetition. */
              combinations(indices, letters_count, count, combs);
              for(i = 0; combs[ i ] != NULL; i++) {
                if(getCombString(combstr, letters, combs[ i ], count) == EXIT_SUCCESS) {

                  /* reset the file pointer to the start of the file. */
                  fseek(dict_file, 0, SEEK_SET);

                  /* loop the dictionary words database. */
                  while(fgets(line, BUFSIZ, dict_file) != NULL) {
                    line[ strcspn(line, "\n") ] = '\0';
                    lowerize(line, BUFSIZ);
                    m = strnlen(line, BUFSIZ);
                    if(m == count) {
                      word = intersect(line, combstr, count);
                      if(strnlen(word, count) == m) {
                        if(unique((void *) word, words_list, cmp) == true)
                          words_list = push(word, &words_list);
                      }
                    }
                  }
                }
              }
              free(indices);
            }
            deallocateCombs(combs);
          }
        }
        fclose(dict_file);
        FOREVER {
          word = pop(&words_list);
          if(word) {
            printf("word:␣%s\n", word);
            free(word);
          } else
            break;
        }
      } else
        fprintf(stderr, "could␣not␣open␣dictionary␣file:␣%s\n", dictionary_path);
    } else
      perror("too␣much␣letters␣given:␣>␣20!");
  }
  exit(ret);
}

/*
 * allocateCombs -- allocate combinations arrays.
 */
size_t **allocateCombs(size_t n, size_t k)
{
  size_t **ret;
  size_t i;

  /* check parameters. */
  if(n > 0) {
    if(k > 0) {
      ret = calloc(n + 1, sizeof(size_t *));
      if(ret) {
        for(i = 0; i < n; i++) {
          ret[ i ] = (size_t *) calloc(k, sizeof(size_t));
          if(!ret[ i ])
            break;
        }
        ret[ i ] = NULL;
      }
    }
  }
  return ret;
}

/*
 * deallocateCombs -- deallocate combinations arrays.
 */
long int deallocateCombs(size_t **c)
{
  long int ret = EXIT_FAILURE;
  size_t i;

  /* check parameters. */
  if(c) {
    for(i = 0; c[ i ] != NULL; i++)
      free(c[ i ]);
    free(c);
    ret = EXIT_SUCCESS;
  }
  return ret;
}

/*
 * lowerize -- tolower every characters in a string.
 */
void lowerize(char *s, size_t l)
{
  char *p = NULL;

  /* check parameters. */
  if(s) {
    p = s;
    while((*p != '\0') && ((p - s) <= l)) {
      *p = tolower(*p);
      ++p;
    }
  }
}

/*
 * getCombString -- return the string from characters
 *                   and indices sets.
 */
long int getCombString(char *comb, char *charset, size_t *indices, size_t count)
{
  long int ret = EXIT_FAILURE;
  size_t i;

  /* chack parameters. */
  if(comb) {
    if(charset) {
      if(indices) {
        if(count > 0) {
          for(i = 0; i < count; i++)
            comb[ i ] = charset[ indices[ i ] - 1 ];
          comb[ i ] = '\0';
          ret = EXIT_SUCCESS;
        }
      }
    }
  }
  return ret;
}

/*
 * printArray -- print array
 */
void printArray(char *s, void *a, size_t c)
{
  char arg[ BUFSIZ ];
  size_t i;

  /* check arguments. */
  if(s) {
    if(a) {
      snprintf(arg, BUFSIZ, "%s", s);
      for(i = 0; i < c; i++) {
        if(i == 0)
          printf("[␣");
        if(strncmp(s, "%c", BUFSIZ) == 0)
          printf(arg, ((char *) a)[i ]);
        if(strncmp(s, "%d", BUFSIZ) == 0)
          printf(arg, ((unsigned char *) a)[ i ]);
        else if(strncmp(s, "%ld", BUFSIZ) == 0)
          printf(arg, ((long *) a)[ i ]);
        if(i < (c - 1))
          printf("␣");
        else
          printf("␣]");
      }
    }
  }
}

/*
 * factorial -- compute n!
 */
size_t factorial(size_t n)
{
  if((n == 0) || (n == 1))
    return 1;
  else
    return (n * factorial(n - 1));
}

/*
 * binomial -- return the number of combinations
 *             without repetitions:
 *             c = n! / (k! (n - k)!)
 */
size_t binomial(size_t n, size_t k)
{
  return (factorial(n) / (factorial(k) * factorial(n - k)));
}

/*
 * combinations -- generates the combinations without
 *                 repetitions and with no order.
 */
void combinations(size_t *s, size_t m, size_t n, size_t **c)
{
  size_t i, j;

  /* Set the base combination: 1, 2, 3, ..., n */
  for (i = 0; i < n; i++)
    s[ i ] = n - i;
  j = 0;
  FOREVER  {
    if(c[ j ])
      memcpy(c[ j++ ], s, sizeof(size_t) * n);

    /*
     * this check is not strictly necessary,
     * but if m is not close to n,
     * it makes the whole thing quite a bit faster
     */
    i = 0;
    if(s[ i ]++ < m)
      continue;
    for(; s[ i ] >= m - i;)
      if(++i >= n)
        return;
    for(s[ i ]++; i; i--)
      s[ i - 1 ] = s[ i ] + 1;
  }
}

/*
 * intersect -- compute the intersection set from two strings.
 */
char *intersect(char *a, char *b, size_t l)
{
  char *tempa, *tempb, *ret;
  size_t i, j, k, la, lb;

  /* check parameters. */
  if(a) {
    if(b) {
      if(l > 0) {
        la = strnlen(a, l);
        tempa = calloc(la, sizeof(char));
        if(tempa) {
          strncpy(tempa, a, la);
          lb = strnlen(b, l);
          tempb = calloc(lb, sizeof(char));
          if(tempb) {
            strncpy(tempb, b, lb);
            ret = (char *) calloc(min(la, lb) + 1, sizeof(char));
            if(ret) {
              k = 0;
              for(i = 0; i < la; i++) {
                for(j = 0; j < lb; j++) {
                  if((tempa[ i ] == tempb[ j ]) &&
                    (tempa[ i ] != 0) &&
                    (tempb[ j ] != 0)) {
                    ret[ k++ ] = tempa[ i ];
                    tempa[ i ] = 0;
                    tempb[ j ] = 0;
                    break;
                  }
                }
              }
              ret[ k ] = 0;
            }
            free(tempb);
          }
          free(tempa);
        }
      }
    }
  }
  return ret;
}

/*
```

```
 * cmp -- comparing callback handler.
 */
bool cmp(void *a, ...)
{
  bool ret = false;
  char *b;   va_list aplist;

  /* chack parameters. */
  if(a) {
    if(b) {
      va_start(aplist, a);
      b = va_arg(aplist, char *);
      ret = strncmp((char *) a, (char *) b, BUFSIZ) == 0 ? true : false;
      va_end(aplist);
    }
  }
  return ret;
}

/* End of find-word.c file. */
```

Other source files are relative to the small list handling code:

```
/* -*- mode: c-mode; -*- */

/* list.h file. */
#ifndef __LIST_H
#define __LIST_H

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

/* Types. */
struct tagList {
  void *l_data;
  struct tagList *l_next;
};

typedef struct tagList list_t;

/* Types. */

/* Functions prototype. */
list_t *push(void *, list_t **);
void *pop(list_t **);
bool unique(void *, list_t *, bool (*cmp)(void *, ...));

#endif /* __LIST_H */

/* End if list.h file. */
```

```
/* -*- mode: c-mode; -*- */

/* File list.c */
#include "list.h"

/*
 * push -- push data on the head of the list.
 */
list_t *push(void *d, list_t **l)
{
  list_t *ret = *l, *temp;

  if(d) {
    temp = (list_t *) calloc(1, sizeof(list_t));
```

```c
    if(temp) {
      temp -> l_next = *l;
      temp -> l_data = d;
      ret = temp;
    }
  }
  return ret;
}

/*
 * pop -- remove data from the tail of the list.
 */
void *pop(list_t **l)
{
  void *ret = NULL;
  list_t *temp = *l;

  if(temp) {
    if(temp -> l_next) {
      while(temp -> l_next -> l_next)
        temp = temp -> l_next;
      ret = temp -> l_next -> l_data;
      free(temp -> l_next);
      temp -> l_next = NULL;
    } else {
      ret = temp -> l_data;
      free(temp);
      *l = NULL;
    }
  }
  return ret;
}

/*
 * unique -- check for other element in the list.
 *           The element to test is provided to
 *           the function itself.
 */
bool unique(void *d, list_t *l, bool (*cmp)(void *, ...))
{
  bool ret = true;
  list_t *p;

  if(d) {
    if(cmp) {
      p = l;
      while(p) {
        if(cmp(d, p -> l_data) == true) {
          ret = false;
          break;
        } else
          p = p -> l_next;
      }
    }
  }
  return ret;
}

/* End of list.c file. */
```

Chapter 4

Low-level I/O.

File Descriptors.
Opening and Creating Files.
Closing Files.
Reading and Writing Files.
Moving Around in Files.
Converting File Descriptors to File Pointers.

As discussed in the previous chapter, the Standard I/O Library provides different methods for reading and writing data efficiently and easily. However, the task performed by these routines, namely buffering and input/output conversion, are not always desirable. For example, when performing input and output directly to and from a device such a tape drive, the programmer needs to be able to determine the buffer sizes to be used, rather than letting the stdio routines do it. Of course, routines do exist that provide that level of control. The Standard I/O Library is simply a user-friendly interface to the system calls described in this chapter, which will call the *low-level* interface.

4.1 File Descriptors.

The reader should recall that in the Standard I/O Library, a file is referred to by a file pointer, of type `FILE *`. When using the low-level interface, a file is referred to using a *file descriptor*, which is simply a small integer. As with stdio, there are three predefined file descriptors: `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`, which refer to the standard input, standard output and standard error stream respectively. The files */dev/fd/0* through */dev/fd/#* refer to file descriptors which can be accessed through the file system. If the file descriptor is open and the mode the file is being opened with is a subset of the mode of the existing descriptor, the call:

```
fd = open("/dev/fd/0", mode);
```

and the call:

```
fd = fcntl(0, F_DUPFD, 0);
```

are equivalent. Unlike the Standard I/O Library, which provides a *shorthand* set of routines to deal with the standard input and output, all the low-level I/O routines require that a valid file descriptor be passed to them.

4.2 Opening and Creating Files.

The `open`[1] routine is used to open a file for reading and/or writing, or to create it. `open` takes a variable number of arguments: a character string containing the complete path name of the file to open, an integer specifying how the file is to be opened and an optional integer mode for use when creating a file. It returns an integer which is the file descriptor, on success or -1 if the file could not be opened. The second argument to open is made up of various constants ORed together. These constants are defined in the file `<fcntl.h>`:

`O_RDONLY` open for reading only;

`O_WRONLY` open for writing only;

`O_RDWR` open for reading and writing.

Any combination of the following flags may additionally be used:

`O_NONBLOCK` do not block on open or for data to become available;

`O_APPEND` append on each write;

`O_CREAT` create file if it does not exist. An additional argument of type `mode_t` must be supplied to the call;

`O_TRUNC` truncate size to 0;

`O_EXCL` error if `O_CREAT` is set and file exists;

`O_SYNC` perform synchronous I/O operations;

`O_SHLOCK` atomically obtain a shared lock;

`O_EXLOCK` atomically obtain an exclusive lock;

`O_NOFOLLOW` if last path element is a symlink, don't follow it;

`O_CLOEXEC` set `FD_CLOEXEC`, the close-on-exec flag, on the new file descriptor;

`O_DIRECTORY` error if path does not name a directory.

If the `O_CREAT` option is given, the optional third argument should contain the mode which the file should to be created. This mode specifies the access permissions on the file and is described in more detail in Chapter 4, Converting File Descriptors to File Pointers..

4.3 Closing Files.

The `close`[2] system call is used to close an open file. `close` takes a single argument, the file descriptor referring to the file to be closed. 0 is returned on success, -1 is returned if an error occurs.

[1] See *open*(2).

[2] See *close*(2).

4.4 Reading and Writing Files.

At this point the reader can easily open and close files, the next thing to do is read and write data from and to that file. Using the low-level interface, files can be read and written a buffer-full at a time. The size of the buffer is left up to the programmer which has to use an appropriate dimension. For example, if the programmer reads or writes characters one at a time, instead of in units of a few thousand, the operating system will access the disk, or the device, once for each character resulting in a slower program speed[3]. The `read` system call takes three arguments: the first is the file descriptor for the open file to read. The second is the pointer to the buffer which will contains data and the third is the number of bytes to read from the file and store into the buffer. If successful, the number of bytes actually read is returned. Upon reading end-of-file, 0 is returned. Otherwise, a -1 is returned and the global variable `errno` is set to indicate the error. The `write` system call takes three arguments: first is the file descriptor to an open file for write. The second argument is a pointer to the buffer containing the data to be written to the file and the third argument is the count, in bytes, of elements from the beginning of the buffer to be written in the file. Similarly to `read`[4] system call, upon successful completion the number of bytes which were written is returned. Otherwise, a -1 is returned and the global variable `errno` is set to indicate the error. Listing 4.1 shows a low-level version of our file appending program. Note that because read and write cause the system to access the disk each times they are called, it is important for the programmer to specify reasonably large buffer sizes or else his/her program, and the system, will run very slowly. Try experimenting with large and small buffer sizes to get a feel for the difference, the reader may need to use a file of one or five bilion characters to really appreciate the difference.

Listing 4.1: append2 - append one file to another using the low-level interface.

```
/* -*- mode: c-mode; -*- */

/* append2.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

/* append2.c program. */

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int n;
  int fromfd, tofd;
  char buf[ BUFSIZ ];
  long ret = EXIT_FAILURE;

  /* Check our arguments. */
  if(argc == 3) {

```

[3] This is actually not entirely true since peripherals are always buffered for I/O operations.

[4] See *read*(2).

```
    /* Open the from-file for reading. */
    if((fromfd = open(argv[ 1 ], O_RDONLY)) >= 0) {

      /*
       * Open the to-file for appending.  If to-file does
       * not exist, open will create it with mode 0644
       * -rw-r--r--.  Note that we specify the mode in octal
       * not decimal.
       */
      if((tofd = open(argv[ 2 ], O_WRONLY | O_CREAT | O_APPEND,
         0644)) >= 0) {

        /*
         * Now read a buffer-full line at a time from from-file
         * and write it to the to-file.  Note that we only
        * write the number of characters read  read in,
         * rather than always writing BUFSIZ characters.
         */
        while((n = read(fromfd, buf, sizeof(buf))) > 0)
          if(write(tofd, buf, n) != n) {
            write(STDERR_FILENO, "Could␣not␣write␣to␣to-file.\n",
                28);
            break;
          }

        /* Now close the files. */
        close(tofd);
        if(errno == 0)
          ret = EXIT_SUCCESS;
      } else {
        write(STDERR_FILENO, argv[ 2 ], strlen(argv[ 2 ]));
      }
      close(fromfd);
    } else {
      write(STDERR_FILENO, argv[ 1 ], strlen(argv[ 1 ]));
    }
  } else {
      write(STDERR_FILENO, "Usage:␣", 7);
      write(STDERR_FILENO, *argv, strlen(argv[ 0 ]));
    write(STDERR_FILENO, "␣from-file␣to-file\n", 19);
  }
  exit(ret);
}

/* End of append2.c file. */
```

In the example above one could disagree with the fact that error messages are printed using the low-level `write`[5] routine. The purpose of this is to explain the usage of the routine itself. It is clear that, `perror` would be quite a good choice, in fact that way one has not to give the size of the character string to print. Using `read` and `write` always involve to deal with buffers and their dimensions. Another problem is the usage of strlen. This routine, defined in `<string.h>`,

[5] See *write*(2).

is capable of computing the length of a nul-terminated string. The reader should try to pass a non nul-terminated string to this routine and see the effect. A safer way to get the length for a string is strnlen which thatkes to arguments: a string and a maximum length to return. In fact if, for whatever reason, the length overflow the value specified in the second argument, the routine returns it.

4.5 Moving Around in Files.

As mentioned before, it is often necessary to move to a specific location in a file before reading or writing data. The low-level routine for moving around in a file is called lseek. The function repositions the offset of the file descriptor in the first argument to the second argument of type off_t which is an offset according to the third argument the *whence* directive. The first argument must be an open file descriptor. lseek repositions the file pointer as follows:

- if whence is SEEK_SET, the offset is set to offset bytes;
- if whence is SEEK_CUR, the offset is set to its current location plus offset bytes;
- if whence is SEEK_END, the offset is set to the size of the file plus offset bytes.

The lseek function allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros, until data is actually written into the gap. Some devices are incapable of seeking and thus the value of the pointer associated with such a device is undefined. Upon successful completion, lseek returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of -1 is returned and errno is set to indicate the error.

4.5.1 Duplicating File Descriptors.

Occasionally it is necessary to have more than one file descriptor referring to the same file. This is common when forking and executing new processes. To obtain a new file descriptor which refers to the same file:

```
int fd, fd2;

......

fd2 = dup(fd);
```

fd2 will now refer to the same file as fd did. dup[6] returns -1 if an error occurs. Two alternate forms of the call allows the programmer to select which file descriptor the user wishes to refer to the file and additional flags. For example suppose that standard input should be connected to a given disk file referred by a file descriptor stored in the variable fd[7]:

```
int fd;

......

dup2(fd, 0);
```

In dup2, the value of the second argument, the new descriptor, is specified. If this descriptor is already in use, it is first deallocated as if a close call had been done first. When the second argument equals the first argument, dup2 just returns without affecting the close-on-exec flag. In

[6] See *dup*(2).

[7] This is how the shell handles the '<' redirect.

dup3, both the value of the second argument, the new descriptor and the close-on-exec flag on the second argument, the new file descriptor, are specified: the second argument specifies the value and the O_CLOEXEC bit in the third argument specifies the close-on-exec flag. Unlike dup2, if the first argument and the second argument are equal then dup3 fails. Otherwise, if the third argument is 0 then dup3 is identical to a call to dup2.

4.6 Converting File Descriptors to File Pointers.

Sometimes is desirable to convert an existing low-level file descriptor referring to an open file into something that can be used with the Standard I/O Library. For example the pipe system call, described in Chapter 9, The Signal Stack., returns a file descriptor connected to the output stream of another program. If this program prints nothing but a list of numbers, it would be useful to be able to use fscanf to read them in. The stdio routine fdopen[8] takes two arguments: a file descriptor referring to an open file and a character string indicating how the file descriptor is to be used. This second argument is identical to the second argument used with fopen. Upon successful completion, fdopen return a FILE pointer. Otherwise, NULL is returned and the global variable errno is set to indicate the error. As reference the second argument, indicating the mode, points to a string beginning with one of the following sequences, additional characters may follow these sequences:

‘‘r’’|‘‘rb’’ open file for reading;

‘‘r+’’|‘‘rb+’’|’’r+b’’ open for reading and writing;

‘‘w’’|‘‘wb’’ open for writing. The file is created if it does not exist, otherwise it is truncated;

‘‘w+’’|‘‘wb+’’|‘‘w+b’’ open for reading and writing. The file is created if it does not exist, otherwise it is truncated;

‘‘a’’|‘‘ab’’ open for writing. The file is created if it does not exist;

‘‘a+’’|‘‘ab+’’|‘‘a+b’’ open for reading and writing. The file is created if it does not exist.

The letter ‘‘b’’ in the mode strings above is strictly for compatibility with ANSI X3.159-1989 (“ANSI C89”) and has no effect; the ‘‘b’’ is ignored. After any of the above prefixes, the mode string can also include zero or more of the following:

‘‘e’’ the close-on-exec flag is set on the underlying file descriptor of the new FILE;

‘‘x’’ if the mode string starts with ‘‘w’’ or ‘‘a’’ then the function shall fail if the file specified by path already exists, as if the O_EXCL flag was passed to the open function. It has no effect if used with fdopen or the mode string begins with ‘‘r’’.

Like described for fopen in 3.2.

[8] See *fdopen*(3).

Chapter 5

Files and Directories.

Files and *directories* forms the interface the system presents to help the user to organize, retrieve and store informations. These are part of an entity called *file system*. Other parts of this interface are the system calls to perform particular operations to properly handle these informations. For example: delete, rename, move, truncate a file, reaname a directory, etc..

5.1 File System Concepts.

Before describing the many system calls and library routines available for manipulating files and directories, it is necessary to provide a brief overview of the OpenBSD *file system*: FFS the Fast File System. This is an improved version of the 4.4BSD File System sometimes referred as UFS, UNIX File System. FFS is designed to be fast, reliable and able to handle the most common situations effectively. By default, during installation, OpenBSD tunes FFS for general use, but the system administator can optimize it to fit the needs - whether one needs to store a very huge amount of tiny files or a some 30 GB files. The administrator doesn't need to know much about FFS internals, but as a programmer, the reader should understand *blocks*, *fragments* and *i-nodes*. OpenBSD can also use these file systems too:

cd9660	for iso 9660 formated cdrom;
ext2fs	ext2 linux file systems;
mfs	memory file system;
msdos	Microsoft msdos filesystem;
nfs	UNIX network filesystem;
ntfs	Microsoft Windows NT file system;
tmpfs	Temporary file system.

A file system is described by its *super-block*, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes. Addresses stored in i-nodes are capable of addressing fragments of *blocks*. File system blocks of at most size `MAXBSIZE` can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be `DEV_BSIZE`, or some multiple of a `DEV_BSIZE` unit. Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a small file is allocated only as many fragments of a large block as are necessary. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the i-node, using the `blksize(fs, ip, lbn)` macro. The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined. The *root i-node* is the root of the file system. i-node 0 can't be used for normal purposes and historically bad blocks were linked to i-node 1[1]. Thus the root i-node is 2. The `fs_minfree` element gives the minimum acceptable percentage of file system blocks that may be free. If the freelist drops below this level, only the super-user may continue to allocate blocks. The `fs_minfree` element may be set to 0 if no reserve of free blocks is deemed necessary, although severe performance degradations will be observed if the file system is run at greater than 95% full; thus the default value of `fs_minfree` is 5%. Empirically the best trade-off between block fragmentation and overall disk utilization at a loading of 95% comes with a fragmentation of 8; thus the default fragment size is an eighth of the block size. The element `fs_optim` specifies whether the file system should try to minimize the time spent allocating blocks (`FS_OPTTIME`), or if it should attempt to minimize the space fragmentation on the disk (`FS_OPTSPACE`). If the value of `fs_minfree` is less than 5%, then the file system defaults to optimizing for space to avoid running out of full sized blocks. If the value of `fs_minfree` is greater than or equal to 5%, fragmentation is unlikely to be problematical, and the file system defaults to optimizing for time.

5.1.1 FFS Versions.

The original FFS was written in 1980 and included hard-coded limits that were ample for the day. File systems could have up to 2^{30} blocks or just under a terabyte (TB). In 1983 a 1 TB file system was unthinkable. In 2024, 1 TB drives are the smaller and cheaper disk on the market. For larger file systems, we have FFS version 2. FFS2 can support file systems up to 8 zettabytes (ZB) and OpenBSD supports FFS and FFS2. The i386 and amd64 boot floppies support only FFS, not FFS2. The installation CD, however, supports both. Most machines that need to boot from floppy don't need FFS2 and probably don't have a BIOS that can support 2 TB dives anyway. The file system creation program *newfs* is smart enough to use FFS2 on file systems quite large to need it, so for most installations, the administrator doesn't nee to worry about difference between FFS and FFS2.

5.1.2 Blocks, Fragments and i-nodes.

Both FFS and FFS2 are managed through *blocks*, *fragments* and *i-nodes*. This arrangement isn't unique to FFS and FFS2; file systems such as NTFS use data blocks and index nodes too. The indexing system used by each file system is largely unique.

blocks — are sections of disk that contain data, Files are placed in one or more blocks. OpenBSD's FFS uses a default block size of 16 KB or eight times the fragment size, whichever is smaller. Not all files are even multiples of 16 KB, so leftover bits go in fragments;

[1] i-node 1 is no longer used for this purpose; however, numerous dump tapes make this assumption, so we are stuck with it.

fragments is one-eighth of the block size or 2 KB by default. A 20 KB file fills one block and two fragments;

i-nodes index nodes, contain basic data about files, such as file's size, permissions and the list of blocks that contain the file. Collectively, the data in an i-node is known as metadata or data about data.

Additionally there are other data structures:

super-blocks which are blocks that contain vital information about the file system's size and specifications. Super-blocks are so important that FFS makes many backup copies of them. If one needs to meddle with superblocks there's an high chance to lost the entire file system.

5.1.3 Ordinary Files.

A *file* contains whatever information a user, or the system itself, places in it. Unlike other operating systems, no format is imposed on a regular file, e.g. sequential, random access, etc. Instead a regular file is considered simply as a sequence of bytes and these bytes could be read and write in any way the programmer wants. Certain programs expect a file to be in a special format, so the C compiler gcc wants a source file to be in a specific format, in this case a C source file, to produce an object file or an executable. So the file format is not determined by the operating system but from the application programs that access the specific file. Directories provide the mapping between the names of files and the files themselves, thus inducing a structure on the file system as a whole. A *directory* contains a number of files; it may also contain subdirectories which in turn contain more files and more subdirectories. A directory behaves exactly like an ordinary file when read, though it may not be written by unprivileged, non super-user, programs. The operating system maintains several directories for its own use; one of these is the *root* directory named with */*. All files in the file system can be found by tracing a path through a chain of directories starting at the root / until the desired file is reached. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of file names separated by slashes. Any file name but the one following the last slash must be the name of a directory. If the sequence begins with a slash, the search begins in the *root directory*; otherwise the search begins in the program's current directory. As limiting cases we have:

- the name "/" refers to the root directory;
- a null file name, e.g. /a/b/, refers to the directory whose name precedes it;
- two slashes together, "//", are interpreted as a single slash.

Each directory always has at least two entries. The name "." in each directory refers to the directory itself. Thus a program may read its current directory, without knowing its name, by opening the file ".". By convention, the name ".." refers to the parent of the directory in which appears, that is, to the directory in which the current directory was created. A program may move from its current directory to the root directory by constantly changing its directory to "..". As a limiting case, when in the root directory the name ".." is a circular link to the root. As per *man hier* the OpenBSD file system contains more or less:

/ root directory;

/altroot/ alternate (backup) location for the root, "/", file system;

/bin/ user utilities fundamental to both single and multi-user environments. These programs are statically compiled and therefore do not depend on any system libraries to run;

/bsd pure kernel executable, the operating system loaded into memory at boot-time;

/bsd.mp pure kernel executable for multiprocessor machines;

/bsd.rd installation kernel. The built-in RAM disk contains utilities which can be run without an external file system, so this kernel is useful for limited system maintenance too;

/bsd.sp pure kernel executable for single processor machines;

/dev/ block and character device files;

/etc/ system configuration files and scripts;

/home/ default location for user home directories;

_*sysupgrade/* download location for *sysupgrade*;

/mnt/ empty directory commonly used by system administrators as a temporary mount point;

/root/ default home directory for the super-user;

/sbin/ system programs and administration utilities fundamental to both single and multiuser environments. Most of these programs are statically compiled and therefore do not depend on any system libraries to run;

/tmp/ temporary files that are not preserved between system reboots. Periodically cleaned by *daily*;

/usr/ contains the majority of user utilities and applications;

/var/ multi-purpose log, temporary, transient, and spool files.

5.1.4 Special files.

Special files are one of the most unusual aspects of the UNIX file system, and thus of OpenBSD. Each I/O device, disk drive, tape drive, serial port, terminal, etc., is associated with at least one such file. To user programs, special files look like any other file, but requests to read or write the file result in activation of the associated device. For example, a program whishing to write on a magnetic tape might open the file */dev/rst**. Requests to read and write this file will cause the tape to move and data to be read or written at the appropriate density. etc. By a long-standing UNIX convention, entries for special files reside in the directory */dev*, but there is nothing in the operating system that requires or enforces this. The amd64 OpenBSD installation supports the following devices:

Special device names

all creates special files for all devices on amd64;

ramdisk ramdisk kernel devices;

std creates the *standard* devices: console, klog, kmem, ksyms, mem, null, stderr, stdin, stdout, tty, zero. Which are absolutely necessary for the system to function properly;

local creates configuration-specific devices, by invoking the shell file MAKEDEV.local.

Disks

cd*	ATAPI and SCSI CD-ROM drives;
fd*	floppy disk drives (3 1/2", 5 1/4");
rd*	*rd* pseudo-disks;
sd*	SCSI disks, including flopticals;
vnd*	*file* pseudo-disk devices;
wd*	*winchester* disk drives (ST506, IDE, ESDI, RLL, ...).

Tapes

ch*	SCSI media changers;
st*	SCSI tape drives.

Terminal ports

tty[0-7][0-9a-f]	NS16x50 serial ports;
ttyc*	Cyclades serial ports;
ttyVI*	Virtio serial ports.

Pseudo terminals

ptm	pty master device;
pty*	set of 62 master pseudo terminals;
tty*	set of 62 slave pseudo terminals.

Console ports

ttyC-J*	wscons display devices;
wscons	minimal wscons devices;
wskbd*	wscons keyboards;
wsmux	wscons keyboard/mouse mux devices.

Pointing devices

wsmouse*	wscons mice;
Printers	lpa* Polled printer port;
lpt*	IEEE 1284 centronics printer.

USB devices

ttyU*	USB serial ports;
uall	all USB devices;
ugen*	generic USB devices;
uhid*	generic HID devices, see *uhid*(4);
fido	fido/* nodes;
ujoy	ujoy/* nodes;
ulpt*	printer devices;
usb*	bus control devices used by usbd for attach/detach.

Special purpose devices

apm	power Management Interface;
audio*	audio devices;
bio	`ioctl` tunnel pseudo-device;
bktr*	video frame grabbers;
bpf	Berkeley Packet Filter;
dt	Dynamic Tracer;
diskmap	disk mapper;
dri	Direct Rendering Infrastructure;
efi	EFI runtime services;
fd	fd/* nodes;
fuse	Userland File-system;
gpio*	General Purpose Input/Output;
hotplug	devices hot plugging;
ipmi*	IPMI BMC access;
nvram	NVRAM access;
kcov	Kernel code coverage tracing;
pci*	PCI bus devices;
pctr*	PC Performance Tuning Register access device;
pf	Packet Filter;
pppx*	PPP Multiplexer;
pppac*	PPP Access Concentrator;
radio*	FM tuner devices;

*random	in-kernel random data source;
rmidi*	Raw MIDI devices;
speaker	PC speaker;
tun*	network tunnel driver;
tap*	ethernet tunnel driver;
tuner*	tuner devices;
uk*	unknown SCSI devices;
video*	video V4L2 devices;
vmm	Virtual Machine Monitor;
vscsi*	Virtual SCSI controller;
pvbus*	paravirtual device tree root;
kstat	Kernel Statistics.

5.1.5 Removable File Systems.

In modern computing, especially in the workstation and personal computer world, it is important to use external extensions to the file system. This is useful to exchange data with other users or remote systems that could not access the internet. Everyone knows the usage of the USB sticks, which are static mass storage devices. OpenBSD provides a mean to add external file systems to the root: the system command *mount*. The *mount* command invokes a file system specific program to prepare and graft the special device or remote node (rhost:path) on to the file system tree at the point node. If either special or node are not provided, the appropriate information is taken from the */etc/fstab* file. For disk partitions, the special device is either a disklabel UID (DUID) or an entry in */dev*. If it is a DUID, it will be automatically mapped to the appropriate entry in */dev*. In either case the partition must be present in the disklabel loaded from the device. The partition name is the last letter in the entry name. For example, */dev/sd0a* and 3eb7f9da875cb9ee.a both refer to the 'a' partition. A mount point node must be an existing directory for a mount to succeed, except in the special case of /, of course. Only the super-user may mount file systems.

5.1.6 Device Numbers.

To create special file associated to a particular device, the super-user could use the script */etc/MAKEDEV* which automates this operation. This script relies on the system utility *mknod*. A special file is characterized by two numbers:

major the major device number is an integer number which tells the kernel which device driver entry point to use. To learn what major device number to use for a particular device, check the file */dev/MAKEDEV* to see if the device is known;

minor the minor device number tells the kernel which subunit the node corresponds to on the device; for example, a subunit may be a file system partition or a tty line.

These numbers are mapped inside */dev/MAKEDEV* script.

5.1.7 Hard Links and Symbolic Links.

A *hard link* to a file is indistinguishable from the original directory entry; any changes to a file are effectively independent of the name used to reference the file. Hard links may not normally refer to directories and may not span file systems. A *symbolic link* contains the name of the file to which it is linked. The referenced file is used when an open operation is performed on the link. There are three system utilities which deal with links:

- `stat` - obtains information about the file;
- `lstat` - like `stat` except when the named file is a symbolic link;
- `readlink` - when used on a symbolic link, places the target name in a string buffer.

A `stat`[2] on a symbolic link will return the linked-to file; an `lstat` must be done to obtain information about the link. The `readlink`[3] call may be used to read the contents of a symbolic link. Symbolic links may span file systems, refer to directories, and refer to non-existent files.

5.2 Determining the Accessibility of a File.

To determine if a file is accessible to a program, the `access`[4] system call may be used. This call takes two arguments. The first argument is the null terminated string relative to the path for which we want to know the permissions and the second argument is the mode argument which is either the bitwise OR of one or more of the access permissions to be checked:

`R_OK`	for read permission;
`W_OK`	for write permission;
`X_OK`	for execute/search permission;
`F_OK`	for the existence test.

These constants are defined in `<sys/unistd.h>`. All components of the pathname path are checked for access permissions, including `F_OK`. If the path cannot be found or if any of the desired access modes would not be granted, then a -1 value is returned and `errno` is set to the reason of failure; otherwise a 0 value is returned. This call is important because is answers to the question: *what are the access permissions for that file?*

5.3 Getting Information from an i-node.

The system call used for obtaining the information stored in an i-node is called `stat`. It takes two arguments. The first argument is the null terminated string holding the path of the object we want to get informations. The second argument is the pointer to an allocated struct of type stat which will hold the requested informations. This argument is defined in `<sys/stat.h>`:

Listing 5.1: The stat structure.

```
struct stat {
  dev_t st_dev;
  ino_t st_ino;
  mode_t st_mode;
  nlink_t st_nlink;
```

[2] See *stat*(2).

[3] See *readlink*(2).

[4] See *access*(2).

```
    uid_t st_uid;
    gid_t st_gid;
    dev_t st_rdev;
    struct timespec st_atim;
    struct timespec st_mtim;
    struct timespec st_ctim;
    off_t st_size;
    blkcnt_t st_blocks;
    blksize_t st_blksize;
    u_int32_t st_flags;
    u_int32_t st_gen;
};
```

single structure members are the following:

`st_dev` a signed 32 bit integer which represent the i-node's device;

`st_ino` an unsigned 64 bit integer which represent the i-node's number;

`st_mode` an unsigned 32 bit integer which represent a mask of bits:

- S_ISUID — set user id on execution;
- S_ISGID — set group id on execution;
- S_ISTXT — sticky bit;
- S_IRWXU RWX — mask for owner:
- S_IREAD, S_IRUSR — R for owner;
- S_IWRITE, S_IWUSRW — W for owner;
- S_IEXEC, S_IXUSR — X for owner;
- S_IRWXG — RWX mask for group:
- S_IRGRP — R for group;
- S_IWGRP — W for group;
- S_IXGRP — X for group;
- S_IRWXO — RWX mask for other:
- S_IROTH — R for other;
- S_IWOTH — W for other;
- S_IXOTH — X for other;
- S_IFMT — mask for the file type:
  - S_IFIFO — name pipe (fifo);
  - S_IFCHR — character special;
  - S_IFDIR — directory;
  - S_IFBLK — block special;
  - S_IFREG — regular;
  - S_IFLNK — symbolic link;
  - S_IFSOCK — socket;
  - S_ISVTX — save swapped text even after use.

`st_nlink` an unsigned 32 bit integer which represent the number of hard links;

`st_uid` an unsigned 32 bit integer which represent the user id of the file's owner;

`st_gid` an unsigned 32 bit integer which represent the group id;

`st_rdev` a signed 32 bit integer which represent the device type;

`st_atim` a structured data type object, struct timespec, which holds the time of the last access;

`st_mtim` a structured data type object, struct timespec, which holds the time of the last data modification;

`st_ctim` a structured data type object, strcut timespec, which holds the time of the last status change;

`st_size` a 64 bit signed integer which represent the file size in bytes;

`st_blocks` a 64 bit signed integer which is the number of blocks containing the file;

`st_blksize` a 32 bit signed integer which represent the optimal block size for file;

`st_flags` a 32 bit unsigned integer which holds user defined flags for the file;

`st_gen` a 32 bit unsigned integer which represent the file generation number.

5.4 Reading Directories.

A directory contains structures of type `dirent`[5], defined in `<sys/dirent.h>`:

Listing 5.2: The dirent structure.

```
#define MAXNAMLEN 255

struct dirent {
  ino_t d_fileno;
  off_t d_off;
  u_int16_t d_reclen;
  u_int8_t d_type;
  u_int8_t d_namlen;
  char d_name[ MAXNAMLEN + 1 ];
};
```

`d_fileno` Files which have been deleted will have i-numbers, `d_fileno`, equal to 0; these should in general be skipped over when reading the directory. A directory is read by simply opening it and reading structures either one at a time or all at once.;

`d_off` is the offset of next entry.

`d_reclen` is the length of this record;

`d_type` The `d_type` member could be:

- `DT_UNKNOWN`;
- `DT_FIFO`;
- `DT_CHR`;
- `DT_DIR`;

[5] `direct` is a macro defined to substitute `dirent`.

- DT_BLK;
- DT_REG;
- DT_LNK;
- DT_SOCK.

d_namlen is the current length of the name stored in d_name for which the maximum possible length is MAXNAMELEN + 1;

d_name it should be noted that the name of file, d_name, is not guaranteed to be null-terminated; programs should always be careful of this.

Listing 5.3 shows a small program that simply open the current directory and prints the names of all of the files it contains.

Listing 5.3: listfiles - list the names of the files in the current directory.

```
/* -*- mode: c-mode; -*- */

/* listfiles.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/dir.h>

/* listfiles program. */
/* Function prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  DIR *dp;
  struct dirent *dir;
  long int ret = EXIT_FAILURE;

  /* Open the current directory. */
  if((dp = opendir(".")) != NULL) {

    /*
     * Read directory entries.  Since we're reading
     * entries one at a time, we use the readdir routine,
     * which buffers them internally.  Don't use the
     * low-level read to do things this way, since
     * at a time is very inefficient.
     */
    while((dir = readdir(dp)) != NULL) {

      /* mark deleted file. */
      if(dir -> d_fileno == 0)
        printf("␣DELETED␣");

      /*
       * Make sure we print no more that DIRSIZ
       * characters.
       */
      printf("%.*s\n", DIRSIZ(dir), dir -> d_name);
    }
    closedir(dp);
    ret = EXIT_SUCCESS;
  } else {
    fprintf(stderr, "Could␣not␣read␣current␣directory\n");
  }
  exit(ret);
}

/* End of listfiles.c file. */
```

The program uses the system routines: `opendir`[6], `readdir`[7] and `closedir`[8]. opendir accepts one argument: the character string which holds the path of the directory to read. It returns a pointer to an object of type directory pointer `DIR` or `NULL` on error. `readdir` accepts one argument: the directory pointer and returns a pointer to an object of type `struct dirent` which holds one directory entry data or `NULL` on error. `closedir` accepts an object of type directory pointer. In order to consolidate the information provided in the preceding sections, Listing 5.4 shows a program similar in function to the standard UNIX program ls. This program will perform an ls -asl on each of its named arguments. If the argument is a directory, the contents of that directory will be listed. For simplicity's sake the program prints the user id and group id of the owner of each file rather than digging up the login and group names. Also, the filenames are not sorted and the directory is simply printed in order it is read. The directory reading routines of Berkeley UNIX are used in the example; the reader should be able to change this himself if necessary.

Listing 5.4: ls - an "ls"-like program.

```
/* -*- mode: c-mode; -*- */

/* ls.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/stat.h>

/* ls program. */
/* Global variables definitions. */
char *modes[] = {
  "---",
  "--x",
  "-w-",
  "-wx",
  "r--",
```

[6] See *opendir*(3).

[7] See *readdir*(3).

[8] See *closedir*(3).

```
  "r-x",
  "rw-",
  "rwx"
};

/* Function prototypes. */
void usage(void);
long int list(char *, uint8_t);
void printout(char *, char *, uint8_t);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int ch;
  long int ret = EXIT_FAILURE;
  struct stat st_buf;
  struct dirent *dir;
  DIR *dp;
  uint8_t flags;

  /* Check arguments count. */
  flags = 0;
  if(argc < 2) {
    ret = list(".", flags);
  } else {

      /* Process arguments. */
      while((ch = getopt(argc, argv, "als")) != -1) {
      switch(ch) {
      case ’a’:
        flags |= 0x01;
        break;

      case ’s’:
        flags |= 0x02;
        break;

      case ’l’:
        flags |= 0x04;
        break;

      default:
        usage();
        flags |= 0x08;
        break;
      }
    }
      if((flags & 0x08) == 0) {
      argc -= optind;
      argv += optind;
      if(stat(*argv, &st_buf) >= 0) {

        /*
         * If it is a directory we list it,
         * otherwise just print the info about
         * the file.
         */
        if((st_buf.st_mode & S_IFMT) == S_IFDIR)
          ret = list(*argv, flags);
        else {
          printout(".", *argv, flags);
          ret = EXIT_SUCCESS;
        }
      } else {
        fprintf(stderr, "ls␣error.\n");
      }
    }
  }
  exit(ret);
}

/*
 * list -- read a directory  and list the files it
 *         contains.
 */
long int list(char *name, uint8_t flags)
{
  long int ret = EXIT_FAILURE;
  DIR *dp;
  struct dirent *dir;

  /* Open the directory. */
  if((dp = opendir(name)) != NULL) {

    /* For each entry... */
      while((dir = readdir(dp)) != NULL) {

      /* Skip removed file. */
      if(dir -> d_fileno == 0)
        continue;

      /* Print it out. */
      printout(name, dir -> d_name, flags);
    }
      ret = EXIT_SUCCESS;
  } else
    fprintf(stderr, "%s:␣cannot␣open.\n", name);
  return ret;
}

/*
 * printout -- print out the information about
 *             a file.
 */
void printout(char *dir, char *name, uint8_t flags)
{
  int i, j;
  char perms[ 10 ];
  struct stat st_buf;
  char newname[ S_BLKSIZE ];

  /*
   * Make full path name, so
   * we have a legal path.
   */
  snprintf(newname, S_BLKSIZE, "%s/%s", dir, name);
  if((name[ 0 ] != ’.’) || ((flags & 0x01) != 0)) {

    /*
     * At this point we know the file exists,
     * so this won’t fail.
     */
    stat(newname, &st_buf);
    if((flags & 0x04) != 0) {

      /* Print size in kbytes. */
      if((flags & 02) != 0)
        printf("%5d ", (st_buf.st_size + S_BLKSIZE - 1) / S_BLKSIZE);

      /*
       * Get the file type.  For convenience (and to
       * make this example universal), we ignore the
       * other types which are version-dependent.
       */
      switch(st_buf.st_mode & S_IFMT) {
      case S_IFREG:
        putchar(’-’);
        break;

      case S_IFDIR:
        putchar(’d’);
        break;

      case S_IFCHR:
        putchar(’c’);
        break;

      case S_IFBLK:
        putchar(’b’);
        break;

      default:
        putchar(’?’);
        break;
      }

      /*
       * Get each of the three groups of permissions
       * (owner, group, other).  Since they're just
       * bits, we can count in binary and use this
       * as subscript (see the modes array, above).
       */
      *perms = '\0';
      for(i = 2; i >= 0; i--) {

        /*
         * Since we're subscripting, we don't
         * read the constants.  Just get a
         * value between 0 and 7.
         */
        j = (st_buf.st_mode >> (i * 3)) & 0x07;

        /*
         * Get the perm bits.
         */
        strncat(perms, modes[ j ], 4);
      }

      /*
       * Handle special bits which replace the 'x'
       * in places.
       */
      if((st_buf.st_mode & S_ISUID) != 0)
        perms[ 2 ] = 's';
      if((st_buf.st_mode & S_ISGID) != 0)
        perms[ 5 ] = 's';
      if((st_buf.st_mode & S_ISVTX) != 0)
        perms[ 8 ] = 't';

      /*
       * Print permissions, number of links,
       * user and group ids.
       */
      printf("%s%3d␣%5d/%-5d␣", perms, \
        st_buf.st_nlink, \
        st_buf.st_uid, \
        st_buf.st_gid);

      /*
       * Print the size of the file in bytes.
       * and the last modification time.  The
       * ctime routine converts a time to ASCII;
       * it is described in Chapter 7, Telling
       * Time and Timing Things.
       */
      if((flags & 0x02) != 0)
```

```
        printf("%7d␣", st_buf.st_size);
      printf("%.12s␣", ctime(&st_buf.st_mtime) + 4);

      /*
       * Finally, print the file name.
       */
    }
    printf("%s", name);
    putchar('\n');
  }
}

/*
 * usage -- show program usage on the shell.
 */
void usage(void)
{
  printf("Usage:␣ls␣[-asl]␣dir\n");
}

/* End of ls.c file. */
```

5.5 Modifying File Attributes.

The chmod[9] system call is used to change the modes of a file. It takes two arguments: the first argument is a character string containing the path of a file. The second argument is a value of type mode_t, the same in the stat structure (see 5.1). A similar call fchmod takes as first argument the file descriptor of an open file and as second argument the same of chmod. Both routines, upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. The chown[10] system call changes the owner and group of a file. It takes three arguments: the first argument is the character string holding the path for the file, the second argument is an integer of type uid_t which represent the new owner user id and finally the third argument of type gid_t that represent the new group id. A similar routine is fchown: its first argument is the file descriptor of an open file and the rest two arguments are the same of chown. Both routines, upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

5.6 Miscellaneous File System Routines.

The rest of this chapter is devoted to the routines that don't fit into their own section but are nevertheless important.

5.6.1 Changing Directories.

A program can change its current working directory with the chdir[11] system call. It takes a single parameter as the character string containing the new directory path. A slightly different system call is fchdir which takes the file descriptor of the directory to change to. Upon successful completion,

[9] See *chmod*(2).

[10] See *chown*(2).

[11] See *chdir*(2).

the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. All of these routines are described in Chapter 13, Networking System Calls..

5.6.2 Deleting and Truncating Files.

Files can be deleted using the unlink[12] system call. It takes one argument: the character string which represents the file path. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. To remove directory we have to use the rmdir system call. It takes one argument: the character string which represents the path to the directory that should be deleted. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. truncate[13] causes the file named by path, or referenced by file descriptor in ftruncate, to be truncated or extended to length bytes in size. If the file was larger than this size, the extra data is lost. If the file was smaller than this size, it will be extended as if by writing bytes with the value zero. With ftruncate, the file must be open for writing. Both routines, upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

5.6.3 Making Directories.

To create a directory we use the mkdir[14] system call. It takes two arguments: the first is a character string containing the path of the directory to create, the second argument an integer of type mode_t to specify the directory modes. The directory path is created with the access permissions specified by the second argument and restricted by the umask of the calling process. The directory's owner id is set to the process's effective user id. The directory's group id is set to that of the parent directory in which it is created. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

5.6.4 Linking and Renaming Files.

The link[15] system call atomically creates the specified directory entry[16]. It takes two arguments: the first is a character string which represent the path of the source object to link to. The second argument is a character string which is the path of the *hard link* to be created with the attributes of the underlying object pointed at by the first argument. If the link is successful: the link count of the underlying object is incremented; the first argument and the second argument share equal access and rights to the underlying object. If the file specified in the first argument is removed, the file specified in the second argument is not deleted and the link count of the underlying object is decremented. The file specified in the first argument must exist for the hard link to succeed and both the files must be in the same file system. As mandated by POSIX.1 the file specified in the first argument may not be a directory. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. To rename a file the rename[17] system call is used. It takes two character string arguments. The first argument is the path of the source file, the second argument is the destination file path. The rename function causes the link named as source object to be renamed as destination object. If the destination object exists, it is first removed. Both source and destination objects must be of the same type: that is, both directories or both non-directories, must reside on the same file system. rename guarantees that if the destination already exists, an instance of it will always exist, even

[12] See *unlink*(2).

[13] See *truncate*(2).

[14] See *mkdir*(2).

[15] See *link*(2).

[16] Hard link.

[17] See *rename*(2).

if the system should crash in the middle of the operation. If the final component of source object is a symbolic link, the symbolic link is renamed, not the file or directory to which it points. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

5.6.5 Symbolic Links.

In OpenBSD, *symbolic links* are simply "pointers" to files; they are not hard links. Unlike them, they may cross file system boundaries. To create a symbolic link the symlink[18] system call is used. A symbolic link provided as second argument in a character string is created to the first argument in a character string: the second argument is the name of the file created, the first argument is the string used in creating the symbolic link. Either name may be an arbitrary path name; the files need not be on the same file system, and the file specified by the first argument need not exist at all. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

5.6.6 The umask Value.

When a file is created with the system call open[19], a mode is supplied for the file to created with. Invisibly to the user, this mode is modified by the program's *umask*. The umask is a number just like the mode, except it indicates permissions to be turned off rather than on. For example, if the program's umask is 0022 and a file is created mode 0666, the actual mode of the file be computed as:

```
file_mode = create_mode & ~umask;
```

so the actual mode of this file will be:

```
0666 & ~0022 = 0666 & 0755 = 0644
```

the umask value only affects creation modes of files and directories; the modes supplied to the chmod call are not affected. Most systems have a default umask value of 0 or 022. It may be changed with the umask system call. This system routine sets the process's file mode creation mask to the value of the argument and returns the previous value of the mask. Only the read, write, and execute file permission bits of the argument are honored, all others are ignored. The file mode creation mask is used by the bind, mkdir, mkdirat, mkfifo, mkfifoat, mknod, mknodat, open and openat system calls to turn off corresponding bits requested in the file mode, see chmod. This clearing allows users to restrict the default access to their files. The default mask value is S_IWGRP|S_IWOTH, which is 022, write access for the owner only. Child processes inherit the mask of the calling process.

[18] See *symlink*(2).

[19] The creat system call is now obsolete.

Chapter 6

Device I/O Control.

The `ioctl` System Call.
Line Disciplines.
The `fcntl` System Call.
Non-blocking I/O.
The `select` System Call.

Controlling input and output devices is an important task for several reasom. Some examples include:

- when prompting fo a password, it is normally desirable to prevent the computer from echoing by printing the characters typed and thus giving the password away;
- many people like to adjust various input control characters on their terminal, such as the *erase*, *kill* and *interrupt* characters;
- programs accesing the magnetic tape device often need to rewind the tape, skip over files on the tape device off-line, etc.;
- the volume level for the audio board output;
- the motor state, on or off, for a disk or optical drive;
- the tray motor for an blueray optical drive;
- a serial port configuration: speed, number of bits, parity, stop bit, etc..

6.1 The `ioctl` System Call.

OpenBSD operating system provide one *catch-all* system call for controlling input and output at the device level. This call is `ioctl`[1]. It takes a variable number of arguments. The first argument is a file descriptor to an open file, the second argument is an unsigned long integer representing the request. This has encoded in it whether the argument is an *in* parameter or *out* parameter and the size of the third optional argument in bytes. Macros and defines used in specifying an `ioctl` request are located in the file `<sys/ioctl.h>`. The third optional argument is either an integer of type `int` or a pointer to a device-specific data structure, depending upon the given request. The following Listing 6.1 shows the usage of ioctl to plays some notes on the internal PC speaker.

[1] See *ioctl*(2).

Listing 6.1: speaker - plays some notes on the internal PC speaker.

```c
/* -*- mode: c-mode; -*- */

/* speaker.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <dev/isa/spkrio.h>

/* speaker program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd, i;
  long int ret = EXIT_FAILURE;
  tone_t tones[ 5 ] = {
    { 440, 200 },
    { 880, 200 },
    { 1660, 200 },
    { 3320, 200 },
    { 6640, 200 }
  };

  /* Call ioctl. */
  if((fd = open("/dev/speaker", O_WRONLY, 0)) >= 0) {
    for(i = 0; i < 5; i++) {
      if(ioctl(fd, SPKRTONE, &tones[ i ]) < 0) {
        perror("speaker");
        break;
      }
    }
    close(fd);
    if(i >= 5)
      ret = EXIT_SUCCESS;
  } else {
    perror("speaker");
  }
  exit(ret);
}

/* End of speaker.c file. */
```

A typical peripheral that the user often encounters is a serial type device: it could be a serial port or a terminal. The serial device is mapped to the file */dev/tty** and */dev/cua**, they as regarded as *hardware terminal*. When a user logs into the OpenBSD system on one of these hardware terminal ports, the system has already opened the associated device and prepared the line for normal interactive use[2]. There is also a special case of a terminal file that connects not to a hardware terminal port, but to another program on the other side. These special terminal devices are called *ptys* and provide the mechanism necessary to give users the same interface to the system when logging in over a network[3] for example. Even in these cases the details of how the terminal file was opened and set up is already handled by special software in the system. Thus, users do not normally need to worry about the details of how these lines are opened or used. For hardware terminal ports, dial-out is supported through matching device nodes called *calling units*. For instance, the terminal called */dev/tty03* would have a matching calling unit called */dev/cua03*. These two devices are normally differentiated by creating the calling unit device node with a minor number 128 greater than the dial-in device node. Whereas the dial-in device, the tty, normally requires a hardware signal to indicate to the system that it is active, the dial-out device, the cua, does not, and hence can communicate unimpeded with a device such as a modem, or with another system over a *serial link*. This means that a process like `getty` will wait on a dial-in device until a connection is established. Meanwhile, a dial-out connection can be established on the dial-out device[4] without disturbing anything else on the system. The `getty` process does not even notice that anything is happening on the `terminal port`. If a connecting call comes in after the dial-out connection has finished, the `getty` process will deal with it properly, without having noticed the intervening dial-out action. When an interactive user logs in, the system prepares the line to behave in a certain way[5], described in `stty` at the command level, and in `termios` at the programming level. To change settings associated with a *login terminal*, refer to the preceding `stty` and `termios` system documentation[6] for the common cases.

6.2 Line Disciplines.

A terminal file is used like any other file in the system in that it can be opened, read, and written to using standard system calls. For each existing terminal file, there is a software processing module called a *line discipline* associated with it. The line discipline essentially glues the low level device driver code with the high level generic interface routines[7] and is responsible for implementing the semantics associated with the device. When a terminal file is first opened by a program, the default line discipline called the termios line discipline is associated with the file. This is the primary line discipline that is used in most cases and provides the semantics that users normally associate with a terminal. When the termios line discipline is in effect, the terminal file behaves and is operated according to the rules described in `termios`. The operations described here generally represent features common across all line disciplines, although some of these calls may not make sense in conjunction with a line discipline other than `termios` and some may not be supported by the underlying hardware[8].

6.2.1 Terminal File Operations.

All of the following operations are invoked using the `ioctl` system call. In addition to the ioctl requests defined here, the specific line discipline in effect will define other requests specific to it[9]. The following section lists the available `ioctl` requests. The name of the request, a description of its purpose, and the typed argument parameter, if any, are listed. For example, the first entry says:

```
TIOCSETD int *ldisc
```

[2] See `getty`.

[3] Using `ssh` or `telnet`.

[4] For the very same hardware terminal port.

[5] called a line discipline.

[6] man pages.

[7] Such as `read` and `write`.

[8] Or lack thereof, as in the case of ptys.

[9] Actually *termios*(4) defines them as function calls, not ioctl requests.

and would be called on the terminal associated with file descriptor zero by the following code fragment:

```
int ldisc;

......

ldisc = TTYDISC;
ioctl(0, TIOCSETD, &ldisc);
```

6.2.2 Terminal File Request Descriptions.

These are:

- TIOCSETD int *ldisc — change to the new line discipline pointed to by ldisc. The available line disciplines currently available are:
  - TTYDISC — termios interactive line discipline;
  - PPPDISC — point-to-point protocol line discipline;
  - NMEADISC — NMEA 0183 line discipline;
  - MSTSDISC — Meinberg Standard Time String line discipline;
- TIOCGETD int *ldisc — return the current line discipline in the integer pointed to by ldisc;
- TIOCSBRK void — set the terminal hardware into BREAK condition;
- TIOCCBRK void — clear the terminal hardware BREAK condition;
- TIOCSDTR void — sssert data terminal ready (DTR);
- TIOCCDTR void — clear data terminal ready (DTR);
- TIOCGPGRP int *tpgrp — return the current process group the terminal is associated with in the integer pointed to by tpgrp. This is the underlying call that implements the *tcgetpgrp*(3) call;
- TIOCSPGRP int *tpgrp — associate the terminal with the process group, as an integer, pointed to by tpgrp. This is the underlying call that implements the *tcsetpgrp*(3) call;
- TIOCGETA struct termios *term — place the current value of the termios state associated with the device in the termios structure pointed to by term. This is the underlying call that implements the *tcgetattr*(3) call;
- TIOCSETA struct termios *term — set the termios state associated with the device immediately. This is the underlying call that implements the *tcsetattr*(3) call with the TCSANOW option;
- TIOCSETAF struct termios *term — first wait for any output to complete, clear any pending input, then set the termios state associated with the device. This is the underlying call that implements the *tcsetattr*(3) call with the TCSAFLUSH option;
- TIOCOUTQ int *num — place the current number of characters in the output queue in the integer pointed to by num;

- TIOCNOTTY void — This call is obsolete but left for compatibility. In the past, when a process that didn't have a controlling terminal[10] first opened a terminal device, it acquired that terminal as its controlling terminal. For some programs this was a hazard as they didn't want a controlling terminal in the first place, and this provided a mechanism to disassociate the controlling terminal from the calling process. It must be called by opening the file */dev/tty* and calling TIOCNOTTY on that file descriptor. The current system does not allocate a controlling terminal to a process on an *open*(2) call: there is a specific ioctl called TIOCSCTTY to make a terminal the controlling terminal. In addition, a program can *fork*(2) and call the *setsid*(2) system call which will place the process into its own session - which has the effect of disassociating it from the controlling terminal. This is the new and preferred method for programs to lose their controlling terminal;

- TIOCSETVERAUTH int *secs — indicate that the current user has successfully authenticated to this session. Future authentication checks may then be bypassed by performing a TIOCCHKVERAUTH check. The verified authentication status will expire after secs seconds. Only root may perform this operation;

- TIOCCLRVERAUTH void — clear any verified auth status associated with this session;

- TIOCCHKVERAUTH void — check the verified auth status of this session. The calling process must have the same real user ID and parent process as the process which called TIOCSETVERAUTH. A zero return indicates success;

- TIOCSTOP void — stop output on the terminal, like typing ^S at the keyboard;

- TIOCSTART void — start output on the terminal, like typing ^Q at the keyboard;

- TIOCSCTTY void — make the terminal the controlling terminal for the process, the process must not currently have a controlling terminal;

- TIOCDRAIN void — wait until all output is drained;

- TIOCEXCL void — set exclusive use on the terminal. No further opens are permitted except by root. Of course, this means that programs that are run by root, or setuid, will not obey the exclusive setting - which limits the usefulness of this feature;

- TIOCNXCL void — clear exclusive use of the terminal. Further opens are permitted.

- TIOCFLUSH int *what — if the value of the int pointed to by what contains the FREAD bit as defined in <sys/fcntl.h>, then all characters in the input queue are cleared. If it contains the FWRITE bit, then all characters in the output queue are cleared. If the value of the integer is zero, then it behaves as if both the FREAD and FWRITE bits were set, i.e., clears both queues;

- TIOCGWINSZ struct winsize *ws — put the window size information associated with the terminal in the winsize structure pointed to by ws. The window size structure contains the number of rows and columns and pixels if appropriate, of the devices attached to the terminal. It is set by user software and is the means by which most full-screen oriented programs determine the screen size;

- TIOCSWINSZ struct winsize *ws — set the window size associated with the terminal to be the value in the winsize structure pointed to by ws, see above;

[10] See The Controlling Terminal in *termios*(4).

- TIOCCONS int *on — if on points to a non-zero integer, redirect kernel console output[11] to this terminal. If on points to a zero integer, redirect kernel console output back to the normal console. This is usually used on workstations to redirect kernel messages to a particular window;

- TIOCMSET int *state — the integer pointed to by state contains bits that correspond to modem state. Following is a list of defined variables and the modem state they represent:

  - TIOCM_LE — Line Enable;
  - TIOCM_DTR — Data Terminal Ready;
  - TIOCM_RTS — Request To Send;
  - TIOCM_ST — Secondary Transmit;
  - TIOCM_SR — Secondary Receive;
  - TIOCM_CTS — Clear To Send;
  - TIOCM_CAR — Carrier Detect;
  - TIOCM_CD — Carrier Detect (synonym);
  - TIOCM_RNG — Ring Indication;
  - TIOCM_RI — Ring Indication (synonym);
  - TIOCM_DSR — Data Set Ready.

  This call sets the terminal modem state to that represented by state. Not all terminals may support this;

- TIOCMGET int *state — return the current state of the terminal modem lines as represented above in the integer pointed to by state;

- TIOCMBIS int *state — the bits in the integer pointed to by state represent modem state as described above; however, the state is OR-ed in with the current state;

- TIOCMBIC int *state — the bits in the integer pointed to by state represent modem state as described above; however, each bit which is on in state is cleared in the terminal;

- TIOCGTSTAMP struct timeval *timeval — return the, single, timestamp;

- TIOCSTSTAMP struct tstamps *tstamps — chooses the conditions which will cause the current system time to be immediately copied to the terminal timestamp storage. This is often used to determine exactly the moment at which one or more of these events occurred, though only one can be monitored. Only TIOCM_CTS and TIOCM_CAR are honoured in tstamps.ts_set and tstamps.ts_clr; these indicate which raising and lowering events on the respective lines should cause a timestamp capture;

- TIOCSFLAGS int *state — the bits in the integer pointed to by state contain bits that correspond to serial port state. Following is a list of defined variables and the serial port state they represent:

  - TIOCFLAG_SOFTCAR — ignore hardware carrier;
  - TIOCFLAG_CLOCAL — set clocal on open;
  - TIOCFLAG_CRTSCTS — set crtscts on open;
  - TIOCFLAG_MDMBUF — set mdmbuf on open.

[11] See *printf*(9).

This call sets the serial port state to that represented by state. Not all serial ports may support this;

- TIOCGFLAGS int *state — return the current state of the serial port as represented above in the integer pointed to by state;

- TIOCSTAT void — causes the kernel to write a status message to the terminal that displays the current load average, the name of the command in the foreground, its process ID, the symbolic wait channel, the number of user and system seconds used, the percentage of CPU the process is getting, and the resident set size of the process.

6.2.3 The winsize Structure.

OpenBSD supports a windowing system such as the X Window System by Xorg. This includes structure which defines the size of a window. Programs such *vim* and *less* use the information about *window size* to determine the number of rows and columns on the *screen*. These informations are stored in the kernel in order to provide a consistent interface, but is not used by the kernel itself:

Listing 6.2: The winsize structure.

```
struct winsize {
  unsigned short ws_row;
  unsigned short ws_col;
  unsigned short ws_xpixel;
  unsigned short ws_ypixel;
};
```

ws_row	member is the number of window rows in characters;
ws_col	member is the number of window columns in characters;
ws_xpixel	member is the window horizontal size in pixels;
ws_ypixel	member is the window vertical size in pixels.

The associated request is TIOCGWINSZ to read the current window size and TIOCSWINSZ to set the window size. When ws_row and ws_col are zero, the entire structure has to be ignored, as no window size has been set. When a window size is changed, either by the user, using a mouse or other device, or by a program, all programs in the terminal's process group are sent the SIGWINCH signal indicating a size change. This enables editors and the like to re-format the screen according to new size. Listing 6.3 shows the usage for the winsize structure.

Listing 6.3: winsize - returns the size of the terminal window.

```
/* -*- mode: c-mode; -*- */

/* winsize.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/ioctl.h>
#include <sys/tty.h>
#include <sys/ttycom.h>

```

```
/* winsize program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd, i;
  long int ret = EXIT_FAILURE;
  struct winsize ws;

  /* Call ioctl. */
  if((fd = open("/dev/tty", O_RDWR | O_NOCTTY)) >= 0) {
    if(ioctl(fd, TIOCGWINSZ, &ws) >= 0) {
      if((ws.ws_row == 0) && (ws.ws_col == 0))
        printf("Ignoring␣the␣winsize␣structure.\n");
      else {
        printf("terminal␣number␣of␣rows:␣%d\n", ws.ws_row);
        printf("terminal␣number␣of␣columns:␣%d\n", ws.ws_col);
        printf("terminal␣x␣pixels␣size:␣%d\n", ws.ws_xpixel);
        printf("terminal␣y␣pixels␣size:␣%d\n", ws.ws_ypixel);
      }
      ret = EXIT_SUCCESS;
    } else
      perror("winsize");
    close(fd);
  }
  exit(ret);
}

/* End of winsize.c file. */
```

6.2.4 The termios Structure.

It is the general terminal line discipline. Informations about that are stored in the termios structure defined in <termios.h>:

Listing 6.4: The termios structure.

```
#define NCCS 20

struct termios {
  tcflag_t c_iflag;
  tcflag_t c_oflag;
  tcflag_t c_cflag;
  tcflag_t c_lflag;
  cc_t c_cc[ NCCS ];
  int c_ispeed;
  int c_ospeed;
};
```

c_iflag　　is a bit mask for the input control flags which can be composed ORing the following constants:

- `IGNBRK` — ignore BREAK condition;
- `BRKINT` — map BREAK to SIGINT;
- `IGNPAR` — ignore (discard) parity errors;
- `PARMRK` — mark parity and framing errors;
- `INPCK` — enable checking of parity errors;
- `ISTRIP` — strip 8th bit off chars;
- `INLCR` — map NL into CR;
- `IGNCR` — ignore CR;
- `ICRNL` — map CR to NL (ala CRMOD);
- `IXON` — enable output flow control;
- `IXOFF` — enable input flow control;
- `IXANY` — any char will restart after stop;
- `IUCLC` — translate upper to lower case;
- `IMAXBEL` — ring bell on input queue full.

`c_oflag` is a bit mask for the output control flags which can be composed ORing the following constants:

- `OPOST` — enable following output processing;
- `ONLCR` — map NL to CR-NL (ala CRMOD);
- `TABDLY` — horizontal tab delay mask;
- `TAB0` — no tab delay or expansion;
- `TAB3` — expand tabs to spaces;
- `OXTABS` — BSD name for TAB3;
- `ONOEOT` — discard EOT's (^D) on output;
- `OCRNL` — map CR to NL;
- `OLCUC` — translate lower case to upper case;
- `ONOCR` — no CR output at column 0;
- `ONLRET` — NL performs the CR function.

`c_cflags` are the hardware control flags. This bit mask could be composed ORing the following constants:

- `CIGNORE` — ignore control flags;
- `CSIZE` — character size mask;
- `CS5` — 5 bits (pseudo);
- `CS6` — 6 bits;
- `CS7` — 7 bits;
- `CS8` — 8 bits;
- `CSTOPB` — send 2 stop bits;
- `CREAD` — enable receiver;
- `PARENB` — parity enable;
- `PARODD` — odd parity, else even;

- `HUPCL` — hang up on last close;
- `CLOCAL` — ignore modem status lines;
- `CRTSCTS` — RTS/CTS full-duplex flow control;
- `CRTS_IFLOW` — XXX compat;
- `CCTS_OFLOW` — XXX compat;
- `MDMBUF` — DTR/DCD hardware flow control;
- `CHWFLOW` — all types of hw flow control.

`c_lflag`

is a bit mask for the local flags. It is composed by ORing the following constants:

- `ECHOKE` — visual erase for line kill;
- `ECHOE` — visually erase chars;
- `ECHOK` — echo NL after line kill;
- `ECHO` — enable echoing;
- `ECHONL` — echo NL even if ECHO is off;
- `ECHOPRT` — visual erase mode for hardcopy;
- `ECHOCTL` — echo control chars as ^(Char);
- `ISIG` — enable signals INTR, QUIT, [D]SUSP;
- `ICANON` — canonicalize input lines;
- `ALTWERASE` — use alternate WERASE algorithm;
- `IEXTEN` — enable DISCARD and LNEXT;
- `EXTPROC` — external processing;
- `TOSTOP` — stop background jobs from output;
- `FLUSHO` — output being flushed (state);
- `XCASE` — canonical upper/lower case;
- `NOKERNINFO` — no kernel output from VSTATUS;
- `PENDIN` — XXX retype pending input (state);
- `NOFLSH` — don't flush after interrupt.

`c_cc`

array contains the control character defined for the terminal. Every member in this array has got a label:

- `VEOF` = 0;
- `VEOL` = 1;
- `VEOL2` = 2;
- `VERASE` = 3;
- `VWERASE` = 4;
- `VKILL` = 5;
- `VREPRINT` = 6;
- first spare = 7;
- `VINTR` = 8;
- `VQUIT` = 9;

- VSUSP = 10;
- VDSUSP = 11;
- VSTART = 12;
- VSTOP = 13;
- VLNEXT = 14;
- VDISCARD = 15;
- VMIN = 16;
- VTIME = 17;
- VSTATUS = 18;
- second spare = 19.

c_ispeed,c_ospeed are the input and output spped in baud. Standard values are:

- B0 = 0 Bd;
- B50 = 50 Bd;
- B75 = 75 Bd;
- B110 = 110 Bd;
- B134 = 134 Bd;
- B150 = 150 Bd;
- B200 = 200 Bd;
- B300 = 300 Bd;
- B600 = 600 Bd;
- B1200 = 1200 Bd;
- B1800 = 1800 Bd;
- B2400 = 2400 Bd;
- B4800 = 4800 Bd;
- B9600 = 9600 Bd;
- B19200 = 19200 Bd;
- B38400 = 38400 Bd;
- B7200 = 7200 Bd;
- B14400 = 14400 Bd;
- B28800 = 28800 Bd;
- B57600 = 57600 Bd;
- B76800 = 76800 Bd;
- B115200 = 115200 Bd;
- B230400 = 230400 Bd;
- EXTA = 19200 Bd;
- EXTB = 38400 Bd.

Listing 6.5 shows a small program that turns off on ECHO and turn on BREAK then prints screenfuls of the files named on its command line. The program pauses after each screenful and waits for the reader to type any character to continue. Because the terminal is in BREAK mode, the read will return immediately. When all files have been displayed, the program resets the terminal modes and exits. This is a primitive version of the OpenBSD *less* command.

Listing 6.5: pager - simple file paginator.

```c
/* -*- mode: c-mode; -*- */

/* pager.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <sys/tty.h>
#include <sys/ttycom.h>

/* pager program. */
#define FOREVER for(;;)

/* Functions prototypes. */
void prompt(void);
long int more(char *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd, i;
  long int ret = EXIT_FAILURE;
  struct termios old_tos, new_tos;

  /* Check arguments count. */
  if(argc >= 2) {
    if((fd = open("/dev/tty", O_RDWR | O_NOCTTY)) >= 0) {

      /* Retrieve terminal informations. */
      if(ioctl(fd, TIOCGETA, &old_tos) >= 0) {
        memcpy((void *) &new_tos, (void *) &old_tos, sizeof(
           struct termios));
        new_tos.c_iflag &= ~IGNBRK;     /* not ignore BREAK. */
        new_tos.c_lflag &= ~ECHO;       /* disable ECHO. */
        new_tos.c_lflag &= ~ISIG;       /* disable signals: INTR,
            QUIT, DSUSP, SUSP. */
        if(ioctl(fd, TIOCSETA, &new_tos) >= 0) {

          /* Printout files. */
          while(--argc)
            if(more(*++argv) == EXIT_FAILURE)
              break;

          /* Reset the terminal configuration. */
          if(ioctl(fd, TIOCSETA, &old_tos) >= 0)
            ret = EXIT_SUCCESS;
          else
            perror("pager:␣failed␣to␣set␣old␣termios");
        } else
          perror("pager:␣failed␣to␣set␣new␣termios");
      } else
        perror("pager:␣failed␣to␣get␣termios");
      close(fd);
    } else
      perror("pager:␣could␣not␣open␣tty");
  } else
    fprintf(stderr, "Usage:␣%s␣file␣[␣file1␣...␣]\n", *argv);
  exit(ret);
}

/*
 * more -- print out characters.
 */
long int more(char *name)
{
  long int ret = EXIT_FAILURE;
  FILE *fp;
  int line;
  char line_buf[ BUFSIZ ];

  /* Check arguments. */
  if(name) {

    /* Open the file to print. */
    if((fp = fopen(name, "r")) != NULL) {
      FOREVER {
        line = 1;
        while(line < 24) {

          /*
           * If end-of-file, let them hit a key one
           * more time and then go back.
           */
          if(fgets(line_buf, BUFSIZ, fp) != NULL) {
            fwrite(line_buf, 1, strlen(line_buf), stdout);
            line++;
          } else {
            fclose(fp);
            ret = EXIT_SUCCESS;
            prompt();
            return ret;
          }
        }
        prompt();
      }
    } else
      fprintf(stderr, "Could␣not␣open␣%s\n", name);
  }
  return ret;
}

/*
 * prompt -- handle interaction with user.
 */
void prompt(void)
{
  int answer;

  printf("Type␣any␣character␣for␣next␣page:␣");
  answer = getchar();
  putchar('\n');
}

/* End of pager.c file. */
```

There are many, many more things which may be done with the `ioctl` system call, including magnetic tape, network routing changes, harddisk and cdrom drives, etc.. All of the operations are described in the various manual pages contained in Section 4 of the OpenBSD Manual Page. The operations described here and used in the examples above are in *tty*(4).

6.3 The `fcntl` System Call.

The `fcntl` system call provides control over the properties of a file that is already open. It takes a variable number of arguments. The first argument is a file descriptor to an open file, the second argument is a command, described below and the third optional argument depends to the second argument: is technically a pointer to `void`, but is interpreted as an `int` by some commands, a pointer to a structure of type `flock` by others and ignored by the rest. The commands are:

`F_DUPFD` return a new descriptor as follows:

- lowest numbered available descriptor greater than or equal to arg, interpreted as an `int`;
- Same object references as the original descriptor;
- New descriptor shares the same file offset if the object was a file;
- Same access mode: read, write or read/write;
- Same file status flags, i.e., both file descriptors share the same file status flags;
- The close-on-exec flag associated with the new file descriptor is set to remain open across *execve*(2) calls.

`F_DUPFD_CLOEXEC` like `F_DUPFD`, but the `FD_CLOEXEC` flag associated with the new file descriptor is set, so the file descriptor is closed when *execve*(2) is called;

`F_GETFD` get the close-on-exec flag associated with the file descriptor fd as `FD_CLOEXEC`. If the returned value ANDed with `FD_CLOEXEC` is 0, the file will remain open across `exec`, otherwise the file will be closed upon execution of `exec` where the third optional argument is ignored;

`F_SETFD` set the close-on-exec flag associated with the file descriptor to the optional third argyment, where this, interpreted an an `int`, is either 0 or `FD_CLOEXEC`, as described above;

F_GETFL get file status flags associated with the file descriptor, as described below where the third optional argument is ignored. The flags for this commands are:

- O_NONBLOCK — non-blocking I/O; if no data is available to a *read*(2) call, or if a *write*(2) operation would block, the read or write call returns -1 with the error EAGAIN;
- O_APPEND — force each write to append at the end of file; corresponds to the O_APPEND flag of *open*(2);
- O_ASYNC — enable the SIGIO signal to be sent to the process group when I/O is possible, e.g., upon availability of data to be read;
- O_SYNC — cause writes to be synchronous. Data will be written to the physical device instead of just being stored in the buffer cache; corresponds to the O_SYNC flag of *open*(2).

F_SETFL Set file status flags associated with the file descriptor to third optional argument which is interpreted an int. For the flags in use with this command see F_GETFL in the previous item;

F_GETOWN get the process ID or process group currently receiving SIGIO and SIGURG signals; process groups are returned as negative values, the third optional argument is ignored;

F_SETOWN set the process or process group to receive SIGIO and SIGURG signals; process groups are specified by supplying the third optional argument, intrpreted as an int, as negative, otherwise it is taken as a process id.

The flock structure is described as follows:

Listing 6.6: The flock structure.

```
struct flock {
  off_t l_start;
  off_t l_len;
  pid_t l_pid;
  short l_type;
  short l_whence;
};
```

l_start is the starting offset;

l_len is the length of the file, if is equal to 0 it means until the end of file;

l_pid lock the owner;

l_type lock the read and write, etc.;

l_whence member is the type of l_start.

The flock system call apply or remove and advisory lock on open file. It takes two arguments. The first argument is the file descriptor for the open file. The second argument is one of:

- LOCK_SH — apply a shared lock;
- LOCK_EX — apply an exclusive lock;

- LOCK_UN — remove an existing lock.

LOCK_SH and LOCK_EX may be combined with the optional LOCK_NB for nonblocking mode. Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee consistency, i.e., processes may still access files without using advisory locks possibly resulting in inconsistencies. The locking mechanism allows two types of locks: shared locks and exclusive locks. At any time multiple shared locks may be applied to a file, but at no time are multiple exclusive, or both shared and exclusive, locks allowed simultaneously on a file. A shared lock may be upgraded to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; this results in the previous lock being released and the new lock applied, possibly after other processes have gained and released the lock. Requesting a lock on an object that is already locked normally causes the caller to be blocked until the lock may be acquired. If operation is the bitwise OR of LOCK_NB and LOCK_SH or LOCK_EX, then this will not happen; instead the call will fail and the error EWOULDBLOCK will be returned. Locks are on files, not file descriptors. That is, file descriptors duplicated through *dup*(2) or *fork*(2) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock. Processes blocked awaiting a lock may be awakened by signals. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

6.4 Non-blocking I/O.

Normally, when a process issued a read, that process is blocked until there is something to read. That is, the process essentially goes to sleep until the read returns either the data read in, end-of-file, or an error. This is not always desirable, however. By using the F_SETFL operation on fcntl, it is possible to make reads, and other operations on the file descriptor, return an error immediately if the operation would block. If this occurs, errno is set to EWOULDBLOCK. Examples of blocking and non-blocking I/O are present in [4], in the networking case it is desirable to create and fork to a thread for every connection in a server program. Just think to the telnet or ssh server: they are stand alone server program, but allows a number of connections to them.

6.5 The select System Call.

The select system call is used to perform *synchronous I/O multiplexing* — that is, it enables the programmer to manage reading and writing to several file descriptors at once without "blocking" indefinitely on any of the operations. select is used by the programmer to check the status of his open file descriptors before operating on them. For example, if the program continuously prints information to the screen, but should also process any input the user types, the program can use select to *poll* the terminal and when characters are present to be read, it can read them in and process them. It takes five arguments: the first argument is the number of the last file descriptors that should be processed: from 0 to this argument - 1 number. The second and third arguments are pointers to the open file descriptors to read and to write respectively. The fourth argument is a pointer to exceptional condition pending. The fifth argument is timeout, if it is a non-null pointer, it specifies a maximum interval to wait for the selection to complete. If this argument is a null pointer, the select blocks indefinitely. To effect a poll, the timeout argument should be non-null, pointing to a zero-valued timeval structure. Timeout is not changed by select and may be reused on subsequent calls; however, it is good style to re-initialize it before each invocation of select. Exceptional conditions include the presence of out-of-band data on a socket. On return, select replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. It returns the total number of ready descriptors in all the sets. The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets:

- FD_ZERO(&fdset) – initializes a descriptor set fdset to the null set;
- FD_SET(fd, &fdset) – includes a particular descriptor fd in fdset;
- FD_CLR(fd, &fdset) – removes fd from fdset;
- FD_ISSET(fd, &fdset) – is non-zero if fd is a member of fdset, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE, which is normally at least equal to the maximum number of descriptors supported by the system.

Any of the second, third and fourth arguments may be given as null pointers if no descriptors are of interest. If successful, select return the number of ready descriptors that are contained in the descriptor sets. If a descriptor is included in multiple descriptor sets, each inclusion is counted separately. If the time limit expires before any descriptors become ready, they return 0. Otherwise, if select return with an error, including one due to an interrupted call, they return -1, and the descriptor sets will be unmodified.

Listing 6.7: select - program to demonstrate the select system call.

```c
/* -*- mode: c-mode; -*- */

/* select.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ioctl.h>

/* select program. */
#define BUFFER_SIZE 32

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int n, nfds;
  char buf[ BUFFER_SIZE ];
  long int ret = EXIT_FAILURE;
  fd_set readfds;
  struct timeval tv;

  /*
   * We will be reading from standard input (file
   * descriptor 0), so we want to know when the
   * user has typed something.
   */
  FD_ZERO(&readfds);
  FD_SET(0, &readfds);

  /* Set the timeout for 10 seconds. */
  bzero((void *) &tv, sizeof(struct timeval));
  tv.tv_sec = 15;
  tv.tv_usec = 0;

  /* Prompt for input. */
  printf("Type␣a␣word;␣if␣you␣don’t␣in␣10␣");
  printf("seconds␣I’ll␣use␣\"WORD\":␣");
  fflush(stdout);

  /*
   * Now call select.  We pass NULL for
   * writefds and exceptfds, since we
   * aren’t interested in them.
   */
  nfds = select(1, &readfds, NULL, NULL, &tv);

  /*
   * Now we check the results.  If nfds is zero,
   * then we timed out and should assume the
   * default.  Otherwise, if file descriptor 0
   * is set in readfds, that means that it is
   * ready to be read and we can read something
   * from it.
   */
  if(nfds == 0)
    strncpy(buf, "WORD", 5);
  else
    if(FD_ISSET(0, &readfds)) {
      n = read(0, buf, BUFFER_SIZE);
      buf[ n > 0 ? n - 1 : 0 ] = ’\0’;
    }
  printf("\nThe␣word␣is:␣%s\n", buf);

  /*
   * This is not useful, but since we use this
   * method to return success or failure, just
   * go on.
   */
  ret = EXIT_SUCCESS;
  exit(ret);
}

/* End of select.c file. */
```

Chapter 7

Information About Users.

Several pieces of information are maintained about each user of the system. Most of this information is stored in the *password file /etc/passwd* and the *group file /etc/group*. This chapter describes each piece of information, what the operating system uses ir for and how programs can access and change it.

7.1 The Login Name.

Each user on the system is given a unique *login name*. It is recommended that login names contain only lowercase characters and digits. They may also contain uppercase characters, non-leading hyphens, periods, underscores, and a trailing '$'. Login names may not be longer than 31 characters[1]. A user uses his login name to identify himself/herself to the system when logging in. Login names are also used for the electronic mail system, to label output printed on a networked printer, etc.. OpenBSD kernel does not use the login name for anything: it is only used by user-level programs. To obtain the login name of the user executing a program, this may use the `getlogin` routine, see the *getlogin*(2) manual entry. It takes no argument and if the call to he routine succeeds, it returns a pointer to a NUL- terminated string in a static buffer. If the name has not been set, it returns `NULL`.

7.2 The User Id.

Each process in the system is associated with in two integers number called the *real user id* and the *effective user id*. These numbers are used by OpenBSD kernel to determine the process's access permissions, record accounting information, etc.. The real user id always identifies the user executing the process. Only the super-user may change his real user id, thus becoming another user. The effective user id is used to determine the process's permissions. Normally, the effective user id is equal to the real user id. By changing its effective user id, a process gains the permissions associated with the new user id and, at least temporarily, loses those associated with its real user id.

[1] See *adduser*(8).

A user id is always unique and refers to only one user of the system. The getuid function returns the real user id of the calling process. The geteuid function returns the effective user id of the calling process. The real user id is that of the user who has invoked the program. As the effective user id gives the process additional permissions during execution of set-user-ID mode processes, getuid is used to determine the real user id of the calling process. The getuid and geteuid functions are always successful, and no return value is reserved to indicate an error. The real and effective user ids are changed using setuid and seteuid system calls respectively. They take one argument of uid_t type. The setuid function sets the real and effective user ids and the saved set-user-ID of the current process to the specified value. The setuid function is permitted if the effective user id is that of the super-user, or if the specified user id is the same as the effective user id. If not, but the specified user id is the same as the real user id, setuid will set the effective user id to the real user id. The seteuid function sets the effective user id of the current process. The effective user id may be set to the value of the real user id or the saved set-user-ID, see *intro*(2) and *execve*(2); in this way, the effective user id of a set-user-ID executable may be toggled by switching to the real user id, then re-enabled by reverting to the set-user-ID value. The setuid and seteuid functions return the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

7.3 The Group Id.

In addition to the real and effective user ids, OpenBSD system associates a *real group id* and an *effective group id* with each process. These numbers are entirely analogous to the real and effective user ids, with the exception that they do not uniquely identify a specific user. Instead, several users may be members of the same group, permitting them to have access to files owned by that group while denying others access. To get the real group id and the effective group id of the calling process we use getgid and getegid respectively. They take no arguments. The real group id is specified at login time and it is the group of the user who invoked the program. As the effective group id gives the process additional permissions during the execution of set-group-ID mode processes, getgid is used to determine the real group id of the calling process. The setgid function sets the real and effective group ids and the saved set-group-ID of the current process to the specified value. The setgid function is permitted if the effective user id is that of the super-user, or if the specified group id is the same as the effective group id. If not, but the specified group id is the same as the real group id, setgid will set the effective group id to the real group id. Supplementary group ids remain unchanged. The setegid function sets the effective group id of the current process. The effective group id may be set to the value of the real group id or the saved set-group-ID; in this way, the effective group id of a set-group-ID executable may be toggled by switching to the real group id, then re-enabled by reverting to the set-group-ID value. The setgid and setegid functions return the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

7.3.1 The OpenBSD Group Mechanism.

A user could be in more groups at once and the processes he executes have the permissions associated with all those groups instead of only one at a time. In order to manipulate this mechanism, there are two system calls: getgroups and setgroups. getgroups takes two arguments and gets the current *group access list* of the current user process and stores it in the array pointed by the second argument of type gid_t. The first argument of type int indicates the number of entries that may be placed in the array pointed by the second argument. getgroups returns the actual number of groups returned in the second argument. No more than NGROUPS_MAX will ever be returned. If the first argument is 0, getgroups returns the number of groups without modifying the second argument array. A successful call returns the number of groups in the group set. A value of -1 indicates that an error occurred, and the error code is stored in the global variable errno. Likewise setgroups sets the group access list of the current user process according to the array pointed by the second argument of type gid_t. The first argument, a parameter of type int, indicates the number of entries in the the second argument array and must be no more than NGROUPS_MAX. Only the super-user may set new groups. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

7.4 Reading the Password File.

The password file contains almost all the information commonly maintained about each user of the system. A super-user accessible only file is */etc/master.passwd* consists of newline-separated records, one per user, containing ten colon-separated fields. These fields are as follows:

name	user's login name;
password	user's encrypted password;
uid	user's login user id;
gid	user's login group id;
class	user's general classification, see *login.conf*(5);
change	password change time;
expire	account expiration time;
gecos	general information about the user;
home_dir	user's home directory;
shell	user's login shell.

The publicly-readable password file is generated from the */etc/master.passwd* and resides in */etc/passwd*. Each line in the file describes a separate user. The differences between these two files are that the latter lacks class, change, expire fields removed and the password field is replaced with and asterisk '*'. To operate on the password database file which is described in *passwd*(5) there are several system calls: getpwnam and getpwuid are some of these. Each entry of this database are in the structure passwd defined in the include file <pwd.h>:

Listing 7.1: The passwd structure.

```
struct passwd {
  char *pw_name;
  char *pw_passwd;
  uid_t pw_uid;
  gid_t pw_gid;
  time_t pw_change;
  char *pw_class;
  char *pw_gecos;
  char *pw_dir;
  char *pw_shell;
  time_t pw_expire;
};
```

pw_name   is the user name string;

pw_passwd   is a string containing an encrypted password;

pw_uid   user id

pw_gid   group id;

pw_change   is the last change time;

pw_class   is the user access class;

    is the Honeywell login info string;

pw_dir pw_gecos[2]   is the user home directory path;

pw_shell   is the user shell interpreter path;

pw_expire   is the expiration date for the user account.

Several routines are provided to read the password file; all of them return a pointer to structure of type passwd, or NULL on end-of-file or error[3]. It points to static data that is overwritten at each call; programs must copy the data into another structure if it is to be saved. The getpwent routine requires no arguments and returns the next entry in the password file, reading sequentially from the beginning. getpwuid takes a numeric user id as an argument and returns the entry for that user id. getpwnam takes a pointer to a character string containing a login name as an argument and returns the entry for that login name. The routines setpwent and endpwent are used to open and close the password file respectively. These should be used to rewind the password file and "reset" the getpwent routine. Listing 7.2 shows the usage of the routines setpwent, endpwent and getpwent.

Listing 7.2: passwd - program to demonstrate the password database system calls.

```
/* -*- mode: c-mode; -*- */

/* passwd.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <pwd.h>

/* passwd program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  struct passwd *pw;

  /* Open the password database file. */
  setpwent();
  do {
```

[3] "entry not found" is considered an error.

```
    pw = getpwent();
    if(pw) {
      printf("user␣name:␣%s,␣", pw -> pw_name);
      printf("user␣id:␣%d,␣", pw -> pw_uid);
      printf("group␣id:␣%d\n", pw -> pw_gid);
    }
  } while(pw);
  ret = EXIT_SUCCESS;

  /* Close the password database file. */
  endpwent();
  exit(ret);
}

/* End of passwd.c file. */
```

7.5 Reading the Group File.

The group file, */etc/group*, also contains lines of colon-separated fields. These lines are described by the group structure, defined in the include file `<grp.h>`:

Listing 7.3: The group structure.

```
struct group {
  char *gr_name;
  char *gr_passwd;
  gid_t gr_gid;
  char **gr_mem;
};
```

The fields are:

`gr_name` the name of the group;

`gr_passwd` the encrypted password for the group. The field is almost always left blank. If non-blank, then the newgrp command prompts for a password before permitting a user to change to this group. Because of the group mechanism, this field is meaningless in OpenBSD;

`gr_gid` the numeric group id of the group;

`gr_mem` pointers to the login names of the members of the group. The list is null-terminated.

The routines to read the group file are patterned directly after those to read the password file. All the routines return a pointer to a structure of type group or NULL on error/end. The routines are called getgrent, getgrgid and getgrnam. The routines setgrent and endgrent are also available. Listing 7.4 provides an example of usage for the system calls to handle the groups database.

Listing 7.4: group - program to demonstrate the group database system calls.

```
/* -*- mode: c-mode; -*- */

/* group.c file. */
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <grp.h>

/* group program. */

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  struct group *grp;
  char **members;

  /* Open the group database file. */
  setgrent();
  do {
    grp = getgrent();
    if(grp) {
      printf("group␣name:␣%s,␣", grp -> gr_name);
      printf("group␣password:␣%s,␣", grp -> gr_passwd);
      printf("group␣id:␣%d\n", grp -> gr_gid);
      printf("group␣members:␣");
      members = grp -> gr_mem;
      while(*members) {
        printf("%s", *members++);
        if(*members)
          printf(",␣");
      }
      printf("\n");
    }
  } while(grp);
  endgrent();
  ret = EXIT_SUCCESS;
  exit(ret);
}

/* End of group.c file. */
```

7.6 Reading the */var/run/utmp* and */var/log/wtmp* Files.

The file */var/run/utmp* contains a record of all users currently logged in on the system. The `<utmp.h>` file declares the structures used to record information about current users in the utmp file, logins and logouts in the wtmp file, and last logins in the lastlog file. The timestamps of date changes, shutdowns, and reboots are also logged in the wtmp file. wtmp file can grow rapidly on busy systems, so daily or weekly rotation is recommended. If any one of these files does not exist, it is not created. They must be created manually and are maintained by *newsyslog*(8).

Listing 7.5: The lastlog and utmp structures.

```
#define _PATH_UTMP "/var/run/utmp"
#define _PATH_WTMP "/var/log/wtmp"
#define _PATH_LASTLOG"/var/log/lastlog"

#define UT_NAMESIZE 32
#define UT_LINESIZE 8
#define UT_HOSTSIZE 256

struct lastlog {
  time_t ll_time;
  char ll_line[ UT_LINESIZE ];
  char ll_host[ UT_HOSTSIZE ];
};

struct utmp {
  char ut_line[ UT_LINESIZE ];
  char ut_name[ UT_NAMESIZE ];
  char ut_host[ UT_HOSTSIZE ];
  time_t ut_time;
};
```

To read the */var/run/utmp* file just open it as showed in the previous chapters. Listing 7.6 shows how to read the utmp file.

Listing 7.6: utmp - program to read /var/run/utmp.

```
/* -*- mode: c-mode; -*- */

/* utmp.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <utmp.h>

/* utmp program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd;
  long int ret = EXIT_FAILURE;
  struct utmp record;

  /* Open the /va/run/utmp file. */
  if((fd = open(_PATH_UTMP, O_RDONLY)) >= 0) {
    while(read(fd, (void *) &record, sizeof(struct utmp)) > 0) {
      if(record.ut_name[ 0 ] != '\0') {
        printf("line:␣%.*s,␣", UT_LINESIZE, record.ut_line);
        printf("name:␣%.*s,␣", UT_NAMESIZE, record.ut_name);
        printf("name:␣%.*s,␣", UT_HOSTSIZE, record.ut_host);
        printf("time:␣%s", ctime(&record.ut_time));
      }
    }
    ret = EXIT_SUCCESS;
    close(fd);
  } else
    perror("open␣/var/run/utmp");
  exit(ret);
}

/* End of utmp.c file. */
```

Chapter 8

Time and Timing.

This chapter covers a miscellany of topics unrelated but for the fact that they have to do with time:

- how OpenBSD system keeps track of time;
- how to put processes to sleep;
- how to determine how CPU time a process uses;
- how to change file modification times.

8.1 Time.

The OpenBSD operating system keeps track of the current date and time by storing the number of seconds that have elapsed since January, 1, 1900 UTC[1]. The time is stored in a signed 64 bit integer.

8.1.1 Obtaining the Time.

In the OpenBSD operating system the `time` call may be used to obtain the time of the day. This function takes one argument and returns the number of seconds elapsed since Jan 1 1970 00:00:00 UTC. This value is also written to the memory pointed by the first argument of type `time_t`, unless now is `NULL`. The `time` function is always successful, and no return value is reserved to indicate an error. `time`, still useable, was replaced by `gettimeofday` routine. This function writes the absolute value of the system's Coordinated Universal Time (UTC) clock to the memory pointed by the first argument, unless it is `NULL`. The UTC clock's absolute value is the time elapsed since Jan 1 1970 00:00:00 +0000 - the Epoch[2]. The clock normally advances continuously, though it may jump discontinuously if a process calls `settimeofday` or *clock_settime*(2). For this reason, `gettimeofday` is not generally suitable for measuring elapsed time. Whenever possible, use *clock_gettime*(2) to measure elapsed time with one of the system's monotonic clocks instead. The `settimeofday` function sets the system's UTC clock to the absolute value now unless now

[1] Coordinated Universal Time, also known as Greenwich Mean Time.

[2] Considered to be the UNIX birthday.

is NULL. Only the super-user may set the clock. If the system *securelevel*(7) is 2 or greater, the clock may only be advanced. This limitation prevents a malicious super-user from setting arbitrary timestamps on files. Setting the clock cancels any ongoing *adjtime*(2) adjustment. The structure pointed to by the first argument is defined in the include file <sys/time.h> as:

Listing 8.1: The timeval structure.

```
struct timeval {
  time_t tv_sec;
  suseconds_t tv_usec;
};
```

tv_sec seconds elapsed from 1/1/1970;

tv_usec microseconds elapse from boot.

The second argument is historical: the system no longer maintains timezone information in the kernel. This argument should always be NULL. gettimeofday zeroes it if it is not NULL. settimeofday ignores the contents of this argument if it is not NULL. Listing 8.2 shows a program getting the time-of-the-day:

Listing 8.2: time - a program to show the time-of-the-day.

```
/* -*- mode: c-mode; -*- */

/* time.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>

/* time program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  struct timeval now;

  /* get-time-of-the-day. */
  if(gettimeofday(&now, NULL) >= 0) {
    printf("time␣in␣seconds:␣%lld,␣", now.tv_sec);
    printf("time␣in␣microseconds:␣%ld\n", now.tv_usec);
    printf("date:␣%s\n", ctime(&now.tv_sec));
    ret = EXIT_SUCCESS;
  }
  exit(ret);
}
```

```

/* End of timer.c file. */
```

8.1.2 Timezones.

On the OpenBSD operating system the *timezone* information could be retrieved by `localtime` and `gmtime` routines. They return pointers to `tm` structures, described below. `localtime` corrects for the time zone and any time zone adjustments, such as Daylight Saving Time in the United States. After filling in the `tm` structure, `localtime` sets the `tm_isdst`'th element of tzname to a pointer to an ASCII string that's the time zone abbreviation to be used with the return value of `localtime`.

Listing 8.3: The tm structure.

```
struct tm {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
  long tm_gmtoff;
  char *tm_zone;
};
```

`tm_sec`	seconds after the minute [0 - 60];
`tm_min`	minutes after the hour [0 - 59];
`tm_hour`	hours since midnight [0 - 23];
`tm_mday`	day of the month [1 - 31];
`tm_mon`	months since January [0 - 11];
`tm_year`	years since 1900;
`tm_wday`	days since Sunday [0 - 6];
`tm_yday`	days since January 1 [0 - 365];
`tm_isdst`	Daylight Saving Time flag;
`tm_gmtoff`	offset from UTC in seconds;
`tm_zone`	timezone abbreviation.

Listing 8.4 shows how to retrieve the timezone for the machine executing the program.

Listing 8.4: timezone - a program to show the time-of-the-day and timezone.

```
/* -*- mode: c-mode; -*- */

/* timezone.c file. */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>

/* timezone program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  struct timeval now;
  struct tm *tm_val;

  /* get-time-of-the-day. */
  if(gettimeofday(&now, NULL) >= 0) {
    if((tm_val = localtime(&now.tv_sec)) != NULL) {
      printf("seconds:␣%d,␣", tm_val -> tm_sec);
      printf("minutes:␣%d,␣", tm_val -> tm_min);
      printf("hours:␣%d,␣", tm_val -> tm_hour);
      printf("day␣of␣month:␣%d,␣", tm_val -> tm_mday);
      printf("month:␣%d,␣", tm_val -> tm_mon);
      printf("year:␣%d,␣", tm_val -> tm_year);
      printf("weekday:␣%d,␣", tm_val -> tm_wday);
      printf("day␣of␣year:␣%d\n", tm_val -> tm_yday);
      printf("summer␣time␣in␣effect?␣%d\n", tm_val -> tm_isdst);
      printf("offset␣from␣UTC␣in␣seconds:␣%ld\n", tm_val ->
         tm_gmtoff);
      printf("timezone␣name:␣%s\n", tm_val -> tm_zone);
      ret = EXIT_SUCCESS;
    } else
        perror("Could␣not␣get␣local␣time");
  } else
      perror("Could␣not␣get␣time-of-the-day");
  exit(ret);
}

/* End of timezone.c file. */
```

8.1.3 Time Differences.

By using gmtime, difftime and asctime routines, it is possible to convert the difference between two times to ASCII. For example, to see how long a user was logged in, his login time can be subtracted from his logout time. This difference can then be taken as UTC and converted to an ASCII string. The hours minutes and seconds fields of this result will represent the difference between the two times, modulo 24 hours. Listing 8.5 shows a program that computes the last session time for a user.

Listing 8.5: difftime - a program to compute the session time of a user.

```
/* -*- mode: c-mode; -*- */

/* difftime.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <utmp.h>
#include <sys/types.h>
#include <sys/time.h>

/* difftime program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd_wtmp;
  long int ret = EXIT_FAILURE;
  double d;
  off_t lp = 0;
  struct utmp tmp_record, login_record, logout_record;

  /* Open the /va/run/utmp file. */
  if(argc == 2) {
    if((fd_wtmp = open(_PATH_WTMP, O_RDONLY)) >= 0) {
      bzero((void *) &login_record, sizeof(struct utmp));
      while(read(fd_wtmp, (void *) &tmp_record, sizeof(struct utmp)) > 0) {
        if(tmp_record.ut_name[ 0 ] != '\0') {
          if(strncmp((const char *) argv[ 1 ], (const char *) tmp_record.ut_name, UT_NAMESIZE) == 0) {
            lp = lseek(fd_wtmp, 0, SEEK_CUR);
            memcpy((void *) &login_record, (void *) &tmp_record, sizeof(struct utmp));
          }
        }
      }
      if(lp >= 0) {
        if(login_record.ut_name[ 0 ] != '\0') {
          printf("Found␣login␣name:␣%s␣in␣position␣%d.\n", argv[ 1 ],  lp);
          if(lseek(fd_wtmp, lp, SEEK_SET) >= 0) {
            bzero((void *) &logout_record, sizeof(struct utmp));
            while(read(fd_wtmp, (void *) &tmp_record, sizeof(struct utmp)) > 0) {
              if(tmp_record.ut_name[ 0 ] == '\0') {
                if(strncmp((const char *) tmp_record.ut_line, (const char *) login_record.ut_line, UT_LINESIZE) == 0) {
                  printf("found␣the␣corresponding␣logout␣entry␣for␣%s...\n", argv[ 1 ]);
                  memcpy((void *) &logout_record, (void *)  &tmp_record,  sizeof(struct utmp));
                  break;
                }
              }
            }
            d = difftime(logout_record.ut_time, login_record.ut_time);
            printf("user␣%s␣last␣session␣time:␣%f␣s.\n", argv[ 1 ], d);
          } else
            perror("Could␣not␣seek␣in␣/var/log/wtmp");
        } else
          fprintf(stderr, "no␣such␣login:␣%s\n", argv[ 1 ]);
      } else
        perror("Could␣not␣seek␣in␣/var/log/wtmp");
      close(fd_wtmp);
    } else
      perror("Could␣not␣open␣/var/log/wtmp");
  } else
    fprintf(stderr, "Usage:␣difftime␣name\n");
  exit(ret);
}

/* End of difftime.c file. */
```

8.2 Sleeping and Alarm Clocks.

8.2.1 Sleeping.

Many times it is necessary for a program to "go to sleep" for a period of time. For example, if some condition must be checked, for example, every 20 minutes before checking things again. The simplest way to do this is to use the sleep system call. The function suspends execution of the calling thread until at least the given number of seconds have elapsed or an unmasked signal is delivered. This version of sleep is implemented with *nanosleep*(2), so delivery of any unmasked signal will terminate the sleep early, even if SA_RESTART is set with *sigaction*(2) for the interrupting signal. It takes one argument an unsigned int representing the seconds to sleep. If sleep sleeps for the full count of seconds, it returns 0. Otherwise, it returns the number of seconds remaining from the original request. The function sets errno to EINTR if it is interrupted by the delivery of a signal.

8.2.2 The Alarm Clock.

Another common need is to be advised when a given amount of time has elapsed, but to be able to continue executing. For example, if a program is waiting for something that "might" happen, it needs to know when it has waited long enough and should give up. To schedule an *alarm*, the alarm system call should be used. The function schedules the SIGALRM signal for delivery to the calling process after the given number of seconds have elapsed. If an alarm is already pending, another call to alarm will supersede the prior call. It takes one argument an unsigned int which represent the number of seconds to trigger the alarm. If this argument is zero, any pending alarm is cancelled. alarm returns the number of seconds remaining until the pending alarm would have expired. If it has already expired, it was cancelled, or no alarm was ever scheduled, it returns zero.

8.3 Process Timing.

Top obtain information about the amount of processor time used by a process, the times system call may be used. The function fills in the structure pointed to by tp with time- accounting information. The tms structure is defined as follows:

Listing 8.6: The tms structure.

```
struct tms {
  clock_t tms_utime;
  clock_t tms_stime;
  clock_t tms_cutime;
  clock_t tms_cstime;
};
```

The elements of this structure are defined as follows:

tms_utime CPU time charged for the execution of user instructions;

tms_stime CPU time charged for execution by the system on behalf of the process;

tms_cutime sum of tms_utime and tms_cutime for all of the child processes;

tms_cstime sum of tms_stime and tms_cstime for all of the child processes.

All times are in CLK_TCKs of a second. The times of a terminated child process are included in the tms_cutime and tms_cstime elements of the parent when one of the *wait*(2) functions returns the process id of the terminated child to the parent. Upon successful completion, times returns the value of real time, in CLK_TCKs of a second, elapsed since an arbitrary point in the past. This point does not change between invocations of times so two such return values constitute a real time interval. On failure, times returns (clock_t) -1 and the global variable errno is set to indicate the error. Listing 8.7 shows the proper method to calculate the amount of CPU time required by a given segment of code.

Listing 8.7: cputime - measure cpu time used by a section of code.

```
/* -*- mode: c-mode; -*- */

/* cputime.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <utmp.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/times.h>
```

```

/* cputime program. */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i, temp, prev, succ;
  long int ret = EXIT_FAILURE;
  struct tms before, after;

  /* Get current time. */
  times(&before);

  /* some code. */
  for(i = 1; i < rand(); i++) {
      prev = 1;
    succ = 2;
    do {
      printf("%d\n", prev);
      temp = prev + succ;
      prev = succ;
      succ = temp;
    } while(succ < 1836311903);
  }
  ret = EXIT_SUCCESS;

  /* Get time after computation. */
  times(&after);
  printf("User␣time:␣%lld␣seconds.\n", after.tms_utime - before.tms_utime);
  printf("System␣time:␣%lld␣seconds.\n", after.tms_stime - before.tms_stime);
  exit(ret);
}

/* End of cputime.c file. */
```

8.4 Changing File Times.

OpenBSD provides several systems call to set file access and modification times. Let's consider the `utimes` routine. It takes two arguments. The first argument is a pointer to the character string containing the path of the file. The second argument is an array of type `timeval` of size 2. This contains the new access time as first value and the modification time as second value in the array. If the second argument is `NULL`, the access and modification times are set to the current time. The caller must be the owner of the file, have permission to write the file, or be the super-user. In either case, the file status change time is set to the current time. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error.

8.5 Interval Timers.

On OpenBSD there are more general mechanism called *interval timers*. They are maintained in structures of type itimerval defined in the include file <sys/time.h>:

Listing 8.8: The itimerval structure.

```
#define ITIMER_REAL     0
#define ITIMER_VIRTUAL  1
#define ITIMER_PROF     2

struct itimerval {
  struct timeval it_interval;
  struct timeval it_value;
};
```

it_interval field specifies the number of seconds and microseconds before the timer should expire; if these values are zero the timer is disabled;

it_value specifies the values the timer should be reset to when expires; if these are zero the timer will not be reset.

The system provides each process with three interval timers, defined in <sys/time.h>. The getitimer call needs two arguments and returns the current value for the kind of timer specified in the first argument from the array pointed by the second argument. The setitimer takes three arguments. The first two arguments are the same of the getitimer system call, the third value is a pointer to the itimerval structure which update the indicated value, returning the previous value of the timer if the new value is non-null. Setting it_value to 0 disables a timer and setting it_interval to 0 causes a timer to be disabled after its next expiration, assuming it_value is non-zero. Time values smaller than the resolution of the system clock are rounded up to this resolution[3]. The ITIMER_REAL timer decrements in real time. 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 interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

[3] Typically 10 milliseconds.

Chapter 9

Processing Signals.

Signals are software interrupts that are delivered to processes to inform them of abnormal events occuring in their environment. Some signals such as *floating point exception*, have a direct counterparts in the computer hardware; other signals, such as *change in child process status*, are purely software-oriented. In OpenBSD most of the signals cause a process to terminate when they are received. Depending on the signal, the memory image of the executing process may be placed on the disk in the file *core*. This is the familiar *core dump*; it is often useful when debugging a broken program.

9.1 Overview of Signal Handling.

OpenBSD system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt:

1. the signal is normally blocked from further occurrence;

2. 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 a signal is to be ignored. A process may also specify that a default action is to be taken by the system when a signal occurs. A signal may also be blocked, in which case its delivery is postponed until it is unblocked. The action to be taken on delivery is determined at the time of delivery. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special signal stack. Signal routines normally execute with the signal that caused their invocation blocked, but other signals may yet occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent, normally empty. It may be changed with a *sigprocmask*(2) call, or when a signal is delivered to the process. 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 is delivered to the process. Signals may be delivered any time a process enters the operating system[1]. If multiple signals are ready to be delivered at the same time, any signals that

[1] E.g., during a system call, page fault or trap, or clock interrupt.

could be caused by traps are delivered first. Additional signals may be processed at the same time, with each appearing to interrupt the handlers for the previous signals before their first instructions. The set of pending signals is returned by the *sigpending*(2) function. When a caught signal is delivered, the current state of the process is saved, a new signal mask is calculated, as described below and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself. When a signal is delivered to a process, a new signal mask is installed for the duration of the process's signal handler, or until a *sigprocmask*(2) call is made. This mask is formed by taking the union of the current signal mask set, the signal to be delivered, and the signal mask sa_mask associated with the handler to be invoked, but always excluding SIGKILL and SIGSTOP.

9.1.1 The sigaction interface.

The following structure, defined in <signal.h> allow the programmer to configure the behaviour of the process in response to signals coming in.

Listing 9.1: The sigaction structure.

```
struct sigaction {
  union {
    void (*__sa_handler)(int);
    void (*__sa_sigaction)(int, siginfo_t *, void *);
  } __sigaction_u;
  sigset_t sa_mask;
  int sa_flags;
};
```

The system call sigaction assigns an action for a signal. It takes three arguments: the first argument is the signal itself. If the second argument is non-zero, it specifies an action: SIG_DFL, SIG_IGN, or a handler routine and mask to be used when delivering the specified signal. If the third argument is non-zero, the previous handling information for the signal is returned to the user. Once a signal handler is installed, it normally remains installed until another sigaction call is made, or an *execve*(2) is performed. The value of sa_handler or, if the SA_SIGINFO flag is set, the value of sa_sigaction instead, indicates what action should be performed when a signal arrives. A signal-specific default action may be reset by setting sa_handler to SIG_DFL. Alternately, if the SA_RESETHAND flag is set the default action will be reinstated when the signal is first posted. The defaults are process termination, possibly with core dump; no action; stopping the process; or continuing the process. If sa_handler is SIG_DFL, the default action for the signal is to discard the signal, and if a signal is pending, the pending signal is discarded even if the signal is masked. If sa_handler is set to SIG_IGN, current and pending instances of the signal are ignored and discarded. If the first argument of sigaction is SIGCHLD and sa_handler is set to SIG_IGN, the SA_NOCLDWAIT flag is implied. The signal mask sa_mask is typically manipulated using the *sigaddset*(3) family of functions. Options may be specified by setting sa_flags. The meaning of the various bits is as follows:

SA_NOCLDSTOP If this bit is set when installing a catching function for the SIGCHLD signal, the SIGCHLD signal will be generated only when a child process exits, not when a child process stops.

SA_NOCLDWAIT If this bit is set when calling sigaction for the SIGCHLD signal, the system will not create zombie processes when children of the calling process exit, though existing zombies will remain. If the calling process subsequently issues a *waitpid*(2), or equivalent and there are no previously existing zombie child processes that match the *waitpid*(2) criteria, it blocks until all of the calling process's child processes that would match terminate, and then returns a value of -1 with errno set to ECHILD.

SA_ONSTACK — If this bit is set, the system will deliver the signal to the process on a signal stack, specified with *sigaltstack*(2).

SA_NODEFER — If this bit is set, further occurrences of the delivered signal are not masked during the execution of the handler.

SA_RESETHAND — If this bit is set, the handler is reset back to SIG_DFL at the moment the signal is delivered.

SA_SIGINFO — If this bit is set, the second argument of the handler is set to be a pointer to a siginfo_t structure as described in <sys/siginfo.h>. It provides much more information about the causes and attributes of the signal that is being delivered.

SA_RESTART — If a signal is caught during the system calls listed below, the call may be forced to terminate with the error EINTR, the call may return with a data transfer shorter than requested, or the call may be restarted. Restarting of pending calls is requested by setting the SA_RESTART bit in sa_flags. The affected system calls include *read*(2), *write*(2), *sendto*(2), *recvfrom*(2), *sendmsg*(2) and *recvmsg*(2) on a communications channel or a slow device[2] and during a *wait*(2) or *ioctl*(2). However, calls that have already committed are not restarted, but instead return a partial success, for example, a short read count.

After a *fork*(2) or *vfork*(2), all signals, the signal mask, the signal stack, and the restart/interrupt flags are inherited by the child. *execve*(2) reinstates the default action for SIGCHLD and all signals which were caught; all other signals remain ignored. All signals are reset to be caught on the user stack and the signal mask remains the same; signals that restart pending system calls continue to do so. Upon successful completion, the value 0 is returned by sigaction; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

9.2 The Signals.

OpenBSD provides the following signals with names as in the include file <signal.h>:

Table 9.1: List of available signals.

Name	Value	Default Action	Description
SIGHUP	1	terminate process	terminal line hangup
SIGINT	2	terminate process	interrupt program
SIGQUIT	3	create core image	quit program
SIGILL	4	create core image	illegal instruction
SIGTRAP	5	create core image	trace trap

[2] Such as a terminal, but not a regular file.

Table 9.1: List of available signals.

Name	Value	Default Action	Description
SIGABRT	6	create core image	*abort*(3) call, formerly SIGIOT
SIGEMT	7	create core image	emulate instruction executed
SIGFPE	8	create core image	floating-point exception
SIGKILL	9	terminate process	kill program, cannot be caught or ignored
SIGBUS	10	create core image	bus error
SIGSEGV	11	create core image	segmentation violation
SIGSYS	12	create core image	system call given invalid argument
SIGPIPE	13	terminate process	write on a pipe with no reader
SIGALRM	14	terminate process	real-time timer expired
SIGTERM	15	terminate process	software termination signal
SIGURG	16	discard signal	urgent condition present on socket
SIGSTOP	17	stop process	stop, cannot be caught or ignored
SIGTSTP	18	stop process	stop signal generated from keyboard
SIGCONT	19	discard signal	continue after stop
SIGCHLD	20	discard signal	child status has changed
SIGTTIN	21	stop process	background read attempted from controlling terminal
SIGTTOU	22	stop process	background write attempted to controlling terminal
SIGIO	23	discard signal	I/O is possible on a descriptor[3]
SIGXCPU	24	terminate process	CPU time limit exceeded[4]

[3] See *fcntl*(2).

[4] See *setrlimit*(2).

Table 9.1: List of available signals.

Name	Value	Default Action	Description
SIGXFSZ	25	terminate process	file size limit exceeded[5]
SIGVTALRM	26	terminate process	virtual time alarm[6]
SIGPROF	27	terminate process	profiling timer alarm[7]
SIGWINCH	28	discard signal	window size change
SIGINFO	29	discard signal	status request from keyboard
SIGUSR1	30	terminate process	user-defined signal 1
SIGUSR2	31	terminate process	user-defined signal 2
SIGTHR	32	discard signal	thread AST

9.3 Sending Signals.

The *kill* function sends the specified signal to a pid. It takes two arguments: the first is the signal as listed in the previous section. The second argument is the pid of a process or a group of processes. This argument may be one of the signals specified in *sigaction*(2) or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of pid. For a process to have permission to send a signal to a process designated by pid, the real or effective user id of the receiving process must match that of the sending process or the user must have appropriate privileges, such as given by a set-user-ID program or the user is the super-user. A single exception is the signal SIGCONT, which may always be sent to any process with the same session id as the caller.

- if pid is greater than zero: sig is sent to the process whose id is equal to pid;
- if pid is zero: sig is sent to all processes whose group id is equal to the process group id of the sender, and for which the process has permission; this is a variant of *killpg*(3);
- if pid is -1: If the user has super-user privileges, the signal is sent to all processes excluding system processes and the process sending the signal. If the user is not the super-user, the signal is sent to all processes with the same uid as the user excluding the process sending the signal. No error is returned if any process could be signaled;
- if pid is negative but not -1: sig is sent to all processes whose process group id is equal to the absolute value of pid; this is a variant of *killpg*(3).

If the value of the first argument causes the signal to be sent to the calling process, either this argument or at least one pending unblocked signal will be delivered before `kill` returns unless the signal is blocked in the calling thread, the signal is unblocked in another thread, or another thread is waiting for the signal in *sigwait*(). Setuid and setgid processes are dealt with slightly differently. For the non-root user, to prevent attacks against such processes, some signal deliveries are not permitted and return the error EPERM. The following signals are allowed through to this

[5] See *setrlimit*(2).

[6] See *setitimer*(2).

[7] See *setitimer*(2).

class of processes: SIGKILL, SIGINT, SIGTERM, SIGSTOP, SIGTTIN, SIGTTOU, SIGTSTP, SIGHUP, SIGUSR1, SIGUSR2. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

9.4 Catching and Ignoring Signals.

Using the sigaction structure to configure signals and the sigaction to attach the handler function to the signal event led us to the Listing 9.2 where two signals are configured: SIGUSR1 and SIGUSR2.

Listing 9.2: sigaction - shows how to intercept/ignore signals.

```
/* -*- mode: c-mode; -*- */

/* sigaction.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

/* sigaction program. */

/* Functions prototypes. */
void handler(int);
int main(int, char *[]);

/* Global variables. */
struct sigaction sa = {
  handler,
  SIGUSR1,
  SA_SIGINFO
};

struct sigaction sb = {
  SIG_IGN,
  SIGUSR2,
};

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;

  /* Setup signal handler for this process. */
  if(sigaction(SIGUSR1, &sa, NULL) >= 0) {
    if(sigaction(SIGUSR2, &sb, NULL) >= 0) {
      ret = EXIT_SUCCESS;
      pause();
    } else
      perror("Could␣not␣setup␣SIGUSR2");
  } else
```

```
    perror("Could␣not␣setup␣SIGUSR1");
  exit(ret);
}

void handler(int si)
{
  /* Saving the current errno value. */
  int save_errno = errno;

  /* Handler code. */
  printf("Entering␣handler.\n");

  /* ... */
  printf("Signal␣passed␣to␣handler:␣%d\n", si);
  printf("Exiting␣handler.\n");

  /* Restore the old errno value. */
  errno = save_errno;
}

/* End of sigaction.c file. */
```

The first is intercepted and handled in the handler function. The second is ignored. Note the sa and sb object of type `struct sigaction`: the first configuration for sa set to intercept the signal SIGUSR1 and fill the `int` parameter passed to the handler with it. The reader should compile and execute the program that will wait until a signal is sent to it. Using *kill* command, from a different console, the user may try:

```
$ ps ax | grep sigaction
22948 p7  R+/1     0:00.00 grep sigaction
$ kill -s SIGUSR2 22948
$ ps ax | grep sigaction
22948 p7  R+/1     0:00.00 grep sigaction
$ kill -s SIGUSR1 22948
$ ps ax | grep sigaction
$
```

on the program console we can read these messages:

```
Entering handler.
Signal passed to handler: 30
Exiting handler.
```

9.4.1 Catching Signals.

A signal can be caught and handled by a user routine by supplying a pointer to that routine in the `sigaction` call. The first time the signal is received, this routine will be called to process that signal. When the routine, commonly called a *signal handler*, is executed, it will be passed a single integer argument indicating which signal was received. This integer can be compared against the constants in `<signal.h>`, enabling the programmer to write general-purpose signal handlers. Listing 9.3 shows a small program that catches the interrupt signal and prints the string "OUCH" when it is received:

Listing 9.3: ouch1 - prints "OUCH" when an interrupt is received.

```
/* -*- mode: c-mode; -*- */

/* ouch1.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

/* ouch1 program. */
#define FOREVER for(;;)

/* Functions prototypes. */
void handler(int);
int main(int, char *[]);
/* Global variables. */
struct sigaction sa = {
  handler,
  SIGINT,
  SA_SIGINFO
};

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;

  /* Setup signal handler for this process. */
  if(sigaction(SIGINT, &sa, NULL) >= 0) {
    ret = EXIT_SUCCESS;
    FOREVER
      pause();
  } else
    perror("Could␣not␣setup␣SIGINT");
  exit(ret);
}

void handler(int si)
{
  /* Handler code */
  printf("OUCH\n");
}

/* End of ouch1.c file. */
```

This program differs from the sigaction.c one since the signal SIGINT has got the SA_RESETHAND flag set. This will reset the SIGINT flag to default behaviour for the specified process after the signal is captured for the first time. The second time the signal is sent to the process again, the process will be interrupted.

9.5 Using Signals for Timeouts.

By using the `alarm` system call, a program can generate timeouts while performing various functions. For example, a program that wishes to read from a terminal, but give up after 30 seconds and take a default action, would issue an alarm request for 30 seconds immediately before starting the read. When 30 seconds elapsed, a `SIGALARM` signal would be sent to the process. Listing 9.4 shows a program using `alarm` system call to produce a timeout.

Listing 9.4: alarm - perform alarm issuing for the executing process.

```c
/* -*- mode: c-mode; -*- */

/* alarm.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>

#define FOREVER for(;;)

/* alarm program. */
/* Functions prototypes. */
void handler(int);
int main(int, char *[]);

/* Global variables. */
struct sigaction sa = {
  handler,
  SIGALRM,
  SA_SIGINFO
};

/* The main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;

  /* setup signal handler for this process. */
  if(sigaction(SIGALRM, &sa, NULL) >= 0) {
    alarm(15);
    ret = EXIT_SUCCESS;
    FOREVER {
      printf("Waiting!\n");
      sleep(5);
    }
  } else
    perror("Could␣not␣setup␣SIGINT");
  exit(ret);
}

```

```
void handler(int si)
{
  /* Handler code. */
  if(si == SIGALRM)
    printf("Alarm␣received.\n");
}

/* End of alarm.c file. */
```

9.5.1 The setjmp and longjmp Routines.

The `setjmp` function save its calling environment in its argument and it returns 0. The corresponding `longjmp` function restore the environment saved by the most recent invocation of the respective `setjmp` function. They then return so that program execution continues as if the corresponding invocation of the `setjmp` call had just returned the value specified by its second argument, instead of 0. The value specified by the second argument must be non-zero; a 0 value is treated as 1 to allow the programmer to differentiate between a direct invocation of `setjmp` and a return via `longjmp`. The `longjmp` routine may not be called after the routine which called the `setjmp` routines returns. All accessible objects have values as of the time the `longjmp` routine was called, except that the values of objects of automatic storage invocation duration that do not have the volatile type and have been changed between the `setjmp` invocation and `longjmp` call are indeterminate. The `setjmp/longjmp` function pairs save and restore the signal mask. Listing 9.5 shows a program using the setjmp/longjmp and alarm system call to produce a timeout.

Listing 9.5: timeout - program to demonstrate a timeout routine.

```
/* -*- mode: c-mode; -*- */

/* timeout.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <setjmp.h>

/* timeout.c program. */
#define FOREVER for(;;)

/* Functions prototypes. */
void timeout(int);
int main(int, char *[]);

/* Global variables. */
struct sigaction sa = {
  timeout,
  SIGALRM,
  SA_SIGINFO | SA_RESETHAND
};
jmp_buf env;

```

```
/* Main function. */
int main(int argc, char *argv[])
{
  char buff[ BUFSIZ ];
  long int ret = EXIT_FAILURE;

  /* Setup signal handler for this process. */
  if(sigaction(SIGALRM, &sa, NULL) >= 0) {

    /*
     * The code inside the if gets executed the first
     * time through setjmp, the code inside the else
     * the second time.
    */
    if(setjmp(env) == 0) {

      /*
       * Issue a request for an alarm to be
       * delivered in 15 seconds.
       */
      alarm(15);

      /* Prompt for input. */
      printf("Type␣a␣word:␣if␣you␣don't␣in␣15␣seconds␣I'll␣use␣\"WORD\":␣");
      fgets(buff, BUFSIZ, stdin);

      /* Turns off the alarm. */
      alarm(0);
      ret = EXIT_SUCCESS;
    } else {
      strncpy(buff, "WORD", BUFSIZ);
    }
    printf("\nThe␣word␣is␣%s\n", buff);
  } else
    perror("Could␣not␣setup␣SIGINT");
  exit(ret);
}

/*
 * timeout -- timeout function, executed when the alarm
 *            is issued.
 */
void timeout(int sig)
{
  /*
   * Ignore the signal for the duration of this
   * routine.
   */
  if(sig == SIGALRM) {

    /* Restore the action of the alarm signal. */
    if(sigaction(SIGALRM, &sa, NULL) >= 0) {

      /*
       * We would perform any timeout-related
       * functions here; in this case there
       * are none.
      */
      ;

      /*
       * Return to the main routine at setjmp
       * and make setjmp return 1.
       */
      ;
      longjmp(env, 1);
    }
  }
}

/* End of timeout.c file. */
```

9.6 The OpenBSD Signal Mechanism.

9.6.1 The Signal Mask.

A user-defined signal handler is called with the signal mechanism provided by OpenBSD where signals are manipulated using sigaddset, sigdelset, sigemptyset, sigfillset, sigismember, sigpending, sigprocmask and sigsuspend system calls. These functions manipulate signal sets stored in a sigset_t object. Either sigemptyset or sigfillset must be called for every object of type sigset_t before any other use of the object. sigemptyset and sigfillset are provided as macros, but actual functions are available if their names are undefined, with #undef name.

- sigemptyset function initializes a signal set to be empty;
- sigfillset initializes a signal set to contain all signals;
- sigaddset adds the specified signal as argument to the signal set;
- sigdelset deletes the specified signal as argument from the signal set;
- sigismember returns whether a specified signal as argument is contained in the signal set.

The sigismember function returns 1 if the signal is a member of the set and 0 otherwise. The other functions return 0 upon success. A -1 return value indicates an error occurred and the global variable errno is set to indicate the reason. The sigprocmask function examines and/or changes the current signal mask, those signals that are blocked from delivery. Signals are blocked if they are members of the current signal mask set. If the second argument is not NULL, the action of sigprocmask depends on the value of the parameter specified as first argument, which can be one of the following values:

SIG_BLOCK The new mask is the union of the current mask and the specified set.

SIG_UNBLOCK The new mask is the intersection of the current mask and the complement of the specified set.

SIG_SETMASK The current mask is replaced by the specified set.

If the third argument is not NULL, it is set to the previous value of the signal mask. When the second argument is NULL, the value of the first argument is insignificant and the mask remains unchanged, providing a way to examine the signal mask without modification. The system quietly disallows SIGKILL or SIGSTOP to be blocked. Only signals which are in the pending state will be blocked. Signals that are explicitly ignored or for which no handler has been installed and where the default action is to discard the signal are not held as pending and will be discarded regardless of the signal mask. Blocked signals remain in the pending state until another call to sigprocmask removes the pending signal(s) from the mask. If there are unblocked signals that are pending after the signal mask is updated, at least one will be delivered before sigprocmask returns. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. sigsuspend temporarily changes the blocked signal mask to the set pointed by the first argument and then waits for a signal to arrive; on return the previous set of masked signals is restored. The signal mask set is usually empty to indicate that all signals are to be unblocked for the duration of the call. In normal usage, a signal is blocked using *sigprocmask*(2) to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using sigsuspend with the previous mask returned by *sigprocmask*(2). The sigsuspend function always terminates by being interrupted, returning -1 with errno set to EINTR. Listing 9.6 shows a program which blocks all signals but SIGUSR1.

Listing 9.6: sigblock - program to demonstrate the blocking of signal(s).

```c
/* -*- mode: c-mode; -*- */

/* sigblock.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <setjmp.h>
#include <sys/signal.h>

/* sigblock program. */
#define FOREVER for(;;)

/* Functions prototypes. */
void handler(int, siginfo_t *, void *);
int main(int, char *[]);

/* Global variables. */
struct sigaction signals;
jmp_buf env;

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;

  /* Setup signal set for this process. */
  signals.sa_sigaction = handler;
  if(sigfillset(&signals.sa_mask) >= 0) {
    if(sigdelset(&signals.sa_mask, SIGUSR1) >= 0) {
      printf("Current␣signal␣mask␣set:␣0x%8x\n", signals.sa_mask);

      /*
       * Blocking all signals but SIGUSR1.
       * Use # `kill -s SIGUSR1 pid` to terminate the
       * process.
       */
      if(sigprocmask(SIG_BLOCK, &signals.sa_mask, NULL) >= 0) {
        if(setjmp(env) == 0) {
          FOREVER {
            ;
          }
        } else
          ret = EXIT_SUCCESS;
      }
    }
  }
  exit(ret);
}

/*
 * handler - the handler function execute when the configured
 *           signal is issued.
 */
void handler(int sig, siginfo_t *mask, void *d)
{
  longjmp(env, 1);
}

/* End of sigblock.c file. */
```

9.6.2 The Signal Stack.

It is possible for a program to specify an alternate stack on which signals should be processed. This may be necessary if receipt of the signal can occur when the process stack is invalid. For example, if a process runs out of stack space, it must be terminated: since there is no stack space available, the stack cannot be extended to catch the signal. Using the alternate signal stack, the process can take the signal on this stack, issue the appropriate requests to increse the stack size limit and then return to normal operation on the regular stack. The *alternate signal stack* is defined in <sys/signals.h> as follows:

Listing 9.7: The sigaltstack structure.

```
typedef struct sigaltstack {
  void *ss_sp;
  size_t ss_size;
  int ss_flags;
} stack_t;
```

sigaltstack allows users to define an alternate stack on which signals delivered to this thread are to be processed. If the first argument is non-zero and SS_DISABLE is set in ss_flags structure member, the signal stack will be disabled. A disabled stack will cause all signals to be taken on the regular user stack. Trying to disable an active stack will cause sigaltstack to return -1 with errno set to EPERM. Otherwise, the ss_sp structure member specifies a pointer to a space to be used as the signal stack and structure member named ss_size specifies the size of that space. When a signal's action indicates its handler should execute on the signal stack, specified with a *sigaction*(2) system call, the system checks to see if the thread is currently executing on that stack. If the thread is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution. If the third argument is non-zero, the current signal stack state is returned in the memory pointed by this argument. The ss_flags field will contain the value SS_ONSTACK if the thread is currently on a signal stack and SS_DISABLE if the signal stack is currently disabled. The value SIGSTKSZ is defined to be the number of bytes/chars that would be used to cover the usual case when allocating an alternate stack area. The following code fragment is typically used to allocate an alternate stack:

```
if((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
  /* error return */
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if(sigaltstack(&sigstk, NULL) == -1)
  perror("sigaltstack");
```

An alternative approach is provided for programs with signal handlers that require a specific amount of stack space other than the default size. The value MINSIGSTKSZ is defined to be the number of bytes/chars that is required by the operating system to implement the alternate stack feature. In computing an alternate stack size, programs should add MINSIGSTKSZ to their stack requirements to allow for the operating system overhead. Signal stacks are automatically adjusted for the direction of stack growth and alignment requirements. Signal stacks may or may not be protected by the hardware and are not "grown" automatically as is done for the normal stack. If the stack overflows and this space is not protected, unpredictable results may occur. On OpenBSD some additional restrictions prevent dangerous address space modifications. The proposed space at ss_sp is verified to be contiguously mapped for read-write permissions, no execute and incapable of syscall entry[8]. If those conditions are met, a page-aligned inner region will be freshly mapped, all zero, with MAP_STACK[9], destroying the pre-existing data in the region. Once the sigaltstack is disabled, the MAP_STACK attribute remains on the memory, so it is best to deallocate the memory via a method that results in *munmap*(2). Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. Listing 9.8 shows a program using the alternate stack feature.

Listing 9.8: sigstack - program to demonstrate the signal stack features.

```
/* -*- mode: c-mode; -*- */

/* sigstack.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
```

[8] See *msyscall*(2).

[9] See *mmap*(2).

```
#include <setjmp.h>
#include <sys/resource.h>
#include <sys/signal.h>

/* sigstack program. */
#define STACKSIZE 10240
#define FOREVER for(;;)

/* Functions prototypes. */
void fn(void);
void handler(int, siginfo_t *, void *);
int main(int, char *[]);

/* Global variables. */
char *stack;                    /* pointer to signal stack base. */
int tooksig = 0;                /* 1 after we take the signal. */
jmp_buf env;

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  struct sigaction signals;
  struct sigaltstack ss;
  struct rlimit limits;

  /* Set stack size limit to 50 kBytes. */
  if(getrlimit(RLIMIT_STACK, &limits) >= 0) {
    if(limits.rlim_cur > (50 * STACKSIZE)) {
      limits.rlim_cur = 50 * STACKSIZE;
    }
    if(setrlimit(RLIMIT_STACK, &limits) >= 0) {

      /*
       * Take illegal instruction and process it with handler,
       * on the interrupt stack.
       */
      signals.sa_mask = 0;
      signals.sa_sigaction = handler;
      signals.sa_flags = SA_ONSTACK;
      if(sigaction(SIGILL, &signals, NULL) >= 0) {

        /*
         * Allocate memory for the signal stack.  The
         * kernel assumes the addresses grow in the same
         * direction as the process stack.
         */
        if((stack = (char *) malloc(sizeof(char *) * STACKSIZE)) != NULL) {

          /*
```

```
           * Issue the call to tell the system about the
           * signal stack.  We pass the end of the signal
           * stack, no the beginning, since the stack
           * grows toward higher addresses.
           */
          ss.ss_size = STACKSIZE;
          ss.ss_sp = (void *) stack;
          if(sigaltstack(&ss, NULL) >= 0) {

            /* Start using the stack. */
            ret = EXIT_SUCCESS;
            fn();
          } else {
            fprintf(stderr, "Cannot␣configure␣alternate␣signal␣stack.\n");
          }
        } else {
          fprintf(stderr, "Out␣of␣memory!\n");
        }
      } else {
        fprintf(stderr, "Cannot␣configure␣SIGILL␣signal␣handling.\n");
      }
    } else {
      fprintf(stderr, "Could␣not␣set␣process␣current␣stack␣limit.\n");
    }
  } else {
    fprintf(stderr, "Could␣not␣get␣process␣current␣stack␣limit.\n");
  }
  exit(ret);
}

/*
 * handler - the handler function called when the signal
 *           is issued.
 */
void handler(int sig, siginfo_t *mask, void *d)
{
  struct rlimit limits;

  /* Increase the stack limit to the maximum. */
  if(getrlimit(RLIMIT_STACK, &limits) >= 0) {
    limits.rlim_cur = limits.rlim_max;
    if(setrlimit(RLIMIT_STACK, &limits) >= 0) {
      tooksig = 1;
      return;
    } else
      fprintf(stderr, "Could␣not␣set␣current␣stack␣limit.\n");
  } else
    fprintf(stderr, "Could␣not␣get␣current␣stack␣limit.\n");
  exit(EXIT_FAILURE);
}

/*
 * fn - a generic recursive test function.
 */
void fn(void)
{
  /* Take up 5 kBytes of space on stack. */
  printf("%s\n", tooksig ? "Now␣on␣extended␣stack." : "On␣50␣kBytes␣stack.");

  /* Recurse. */
  fn();
}

/* End of sigstack.c file. */
```

Signals play an important role in OpenBSD programming and it is important to understand them. This chapter has discussed several of the techniques and pitfalls associated with signal processing: Chapter 10, Creating Pipes Directly., discuss several more signals associated with OpenBSD *job control*.

Chapter 10

Executing Programs

One of the most powerful tools provided for the UNIX programmer on OpenBSD is the ability to have one program execute another. For example, the command interpreter[1] is a simple program like any other, which executes programs for the user. It is possible for anyone to write a shell if the user doesn't like the ones provided and several people have. This chapter describes the methods used to execute programs from within other programs.

10.1 The System Library Routine.

The simplest way to execute a program is by using the `system` library routine. This takes a single argument, a character string containing the command to be executed. The `system` function hands the argument string to the command interpreter *sh*(1). The calling process waits for the shell to finish executing the command, ignoring `SIGINT` and `SIGQUIT` and blocking `SIGCHLD`. If the argument string is `NULL`, `system` will return non-zero. Otherwise, it returns the termination status of the shell in the format specified by *waitpid*(2). Note that fork handlers established using *pthread_atfork*(3) are not called when a multithreaded program calls `system`. If a child process cannot be created, or the termination status of the shell cannot be obtained, `system` returns -1 and sets `errno` to indicate the error. If execution of the shell fails, `system` returns the termination status for a program that terminates with a call of `exit(127)`. There are three major problems with `system`: first, it is not versatile:

- commands may be executed, but the process executing them has no control over the subprocess;
- a lot of overhead is required. Before executing the desired command, `system` executes a shell process. Because the shell will immediately be executing something else, this is a waste of processor time;
- `system` is a security hole. In order to prevent random system cracking, the security problems presented by `system` will be not described here. Suffice to say that a set-user-id, particularly to the super-user, program should never use `system` to execute its sub processes.

[1] Called the *shell*.

10.2 Executing Programs Directly.

The alternative to using system is to create new processes and execute programs directly. There are three distinct steps to executing programs: creating new processes, making them execute other programs and waiting for them to terminate. In order to execute a program, it is first necessary to create a new process for that program to run in. A running program creates a new process by making a copy of itself. This copy is then immediately overlaid with the new program to be executed.

10.2.1 Creating Processes.

The system call to create a new process is called fork. fork causes creation of a new process: this is called *child process* which is an exact copy of the calling process, called *parent process*, except for the following:

- the child process has a unique process id, which also does not match any existing process group id;
- the child process has a different parent process id[2];
- the child process has a single thread;
- the child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an *lseek*(2) on a descriptor in the child process can affect a subsequent *read*(2) or *write*(2) by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes;
- the child process has no *fcntl*(2)-style file locks;
- the child process' resource utilizations are set to 0[3];
- all interval timers are cleared[4];
- the child process' semaphore undo values are set to 0[5];
- the child process' pending signals set is empty;
- the child process has no memory locks[6];

In general, the child process should call *_exit*(2) rather than *exit*(3). Otherwise, any stdio buffers that exist both in the parent and child will be flushed twice. Similarly, *_exit*(2) should be used to prevent *atexit*(3) routines from being called twice[7]. Upon successful completion, fork returns a value of 0 to the child process and returns the process id of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.

[2] I.e., the process id of the parent process.

[3] See *getrusage*(2).

[4] See *setitimer*(2).

[5] See *semop*(2).

[6] See *mlock*(2) and *mlockall*(2).

[7] Once in the parent and once in the child

10.2.2 Executing Programs.

The system call to execute programs is generically called exec. It exists in several forms described below, but all forms of the call share certain properties. The exec family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the *new process image file*. There shall be no return from a successful exec, because the calling process image is overlaid by the new process image. The fexecve function shall be equivalent to the execve function except that the file to be executed is determined by the file descriptor in the first argument instead of a pathname. The file offset of the first argument, the file descriptor, is ignored. When a C-language program is executed as a result of a call to one of the exec family of functions, it shall be entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. In addition, the following variable, which must be declared by the user if it is to be used directly:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The argv and environ arrays are each terminated by a null pointer. The null pointer terminating the argv array is not counted in argc. Applications can change the entire environment in a single operation by assigning the environ variable to point to an array of character pointers to the new environment strings. After assigning a new value to environ, applications should not rely on the new environment strings remaining part of the environment, as a call to getenv, putenv, setenv, unsetenv, or any function that is dependent on an environment variable may, on noticing that environ has changed, copy the environment strings to a new array and assign environ to point to it. Any application that directly modifies the pointers to which the environ variable points has undefined behavior. Conforming multi-threaded applications shall not use the environ variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable shall be considered a use of the environ variable to access that environment variable. The arguments specified by a program with one of the exec functions shall be passed on to the new process image in the corresponding main arguments. The first argument of the functions: execl, execle, execlp, execv, execve and execvp represents a pointer to the pathname string that identifies the new process image file. For the system calls: execlp and execvp the first argument is used to construct a pathname that identifies the new process image file. If the file argument contains a '/' character, the file argument shall be used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable PATH[8]. If this environment variable is not present, the results of the search are implementation-defined. There are two distinct ways in which the contents of the process image file may cause the execution to fail, distinguished by the setting of errno to either ENOEXEC or EINVAL. In the cases where the other members of the exec family of functions would fail and set errno to ENOEXEC, the execlp and execvp functions shall execute a command interpreter and the environment of the executed command shall be as if the process invoked the sh utility using execl as follows:

```
execl(<shell path>, arg0, file, arg1, ..., (char *) 0);
```

where <shell path> is an unspecified pathname for the sh utility, file is the process image file, and for execvp, where arg0, arg1, and so on correspond to the values passed to execvp in argv[0], argv[1], and so on. The arguments represented by arg0, ... are pointers to null-terminated character strings. These strings shall constitute the argument list available to the new process image. The list is terminated by a null pointer. The argument arg0 should point to

[8] See the Base Definitions volume of POSIX.1-2017, Chapter 8, Environment Variables.

a filename string that is associated with the process being started by one of the exec functions. The argument argv is an array of character pointers to null-terminated strings. The application shall ensure that the last member of this array is a null pointer. These strings shall constitute the argument list available to the new process image. The value in argv[0] should point to a filename string that is associated with the process being started by one of the exec functions. In the functions execle and fexecve the last argument is an array of character pointers to null-terminated strings. These strings shall constitute the environment for the new process image. This array is terminated by a null pointer. For those forms not containing an array for the environment: execl, execv, execlp, and execvp, the environment for the new process image shall be taken from the external variable environ in the calling process. The number of bytes available for the new process' combined argument and environment lists is {ARG_MAX}. It is implementation-defined whether null terminators, pointers, and/or any alignment bytes are included in this total. File descriptors open in the calling process image shall remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description remain unchanged. For any file descriptor that is closed for this reason, file locks are removed as a result of the close as described in close. Locks that are not removed by closing of file descriptors remain unchanged. If file descriptor 0, 1, or 2 would otherwise be closed after a successful call to one of the exec family of functions, implementations may open an unspecified file for the file descriptor in the new process image. If a standard utility or a conforming application is executed with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing, the environment in which the utility or application is executed shall be deemed non-conforming, and consequently the utility or application might not behave as described in this standard. Directory streams open in the calling process image shall be closed in the new process image. The state of the floating-point environment in the initial thread of the new process image shall be set to the default. The state of conversion descriptors and message catalog descriptors in the new process image is undefined. For the new process image, the equivalent of:

```
setlocale(LC_ALL, "C")
```

shall be executed at start-up. Signals set to the default action, SIG_DFL, in the calling process image shall be set to the default action in the new process image. Except for SIGCHLD, signals set to be ignored, SIG_IGN, by the calling process image shall be set to be ignored by the new process image. Signals set to be caught by the calling process image shall be set to the default action in the new process image[9]. If the SIGCHLD signal is set to be ignored by the calling process image, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new process image. After a successful call to any of the exec functions, alternate signal stacks are not preserved and the SA_ONSTACK flag shall be cleared for all signals. After a successful call to any of the exec functions, any functions previously registered by the atexit or pthread_atfork functions are no longer registered. If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user id, effective group id, saved set-user-id, and saved set-group-id are unchanged in the new process image. Otherwise, if the set-user-id mode bit of the new process image file is set, the effective user id of the new process image shall be set to the user id of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image shall be set to the group id of the new process image file. The real user id, real group id, and supplementary group ids of the new process image shall remain the same as those of the calling process image. The effective user id and effective group id of the new process image shall be saved, as the saved set-user-id and the saved set-group-id, for use by setuid. Any shared memory segments attached to the calling process image shall not be attached to the new process image. Any named semaphores open in the calling process shall be closed as if by appropriate calls to sem_close. Any blocks of typed memory that were mapped in the calling process are unmapped, as if munmap was implicitly called

[9] See <signal.h>.

to unmap them. Memory locks established by the calling process via calls to mlockall or mlock shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to the exec function. If the exec function fails, the effect on memory locks is unspecified. Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image. When the calling process image does not use the SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC scheduling policies, the scheduling policy and parameters of the new process image and the initial thread in that new process image are implementation-defined. When the calling process image uses the SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC scheduling policies, the process policy and scheduling parameter settings shall not be changed by a call to an exec function. The initial thread in the new process image shall inherit the process scheduling policy and parameters. It shall have the default system contention scope, but shall inherit its allocation domain from the calling process image. Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image. All open message queue descriptors in the calling process shall be closed, as described in mq_close. Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the exec function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the exec function itself blocks awaiting such I/O completion. In no event, however, shall the new process image created by the exec function be affected by the presence of outstanding asynchronous I/O operations at the time the exec function is called. Whether any I/O is canceled, and which I/O may be canceled upon exec, is implementation-defined. The new process image shall inherit the CPU-time clock of the calling process image. This inheritance means that the process CPU-time clock of the process being exec-ed shall not be reinitialized or altered as a result of the exec function other than to reflect the time spent by the process executing the exec function itself. The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero. If the calling process is being traced, the new process image shall continue to be traced into the same trace stream as the original process image, but the new process image shall not inherit the mapping of trace event names to trace event type identifiers that was defined by calls to the posix_trace_eventid_open or the posix_trace_trid_eventid_open functions in the calling process image. If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described in the posix_trace_shutdown function. The thread id of the initial thread in the new process image is unspecified. The size and location of the stack on which the initial thread in the new process image runs is unspecified. The initial thread in the new process image shall have its cancellation type set to PTHREAD_CANCEL_DEFERRED and its cancellation state set to PTHREAD_CANCEL_ENABLED. The initial thread in the new process image shall have all thread-specific data values set to NULL and all thread-specific data keys shall be removed by the call to exec without running destructors. The initial thread in the new process image shall be joinable, as if created with the detachstate attribute set to PTHREAD_CREATE_JOINABLE. The new process shall inherit at least the following attributes from the calling process image:

- nice value[10];
- semadj values[11];
- process id;
- parent process id;
- process group id;
- session membership;

[10] See nice.

[11] See semop.

- real user id;
- real group id;
- supplementary group ids;
- time left until an alarm clock signal[12];
- current working directory;
- root directory;
- file mode creation mask[13];
- file size limit[14]
- process signal mask[15];
- pending signal[16];
- tms_utime, tms_stime, tms_cutime, and tms_cstime[17];
- resource limits;
- controlling terminal;
- interval timers.

The initial thread of the new process shall inherit at least the following attributes from the calling thread:

- signal mask[18];
- pending signals[19].

All other process attributes defined in this volume of POSIX.1-2017 shall be inherited in the new process image from the old process image. All other thread attributes defined in this volume of POSIX.1-2017 shall be inherited in the initial thread in the new process image from the calling thread in the old process image. The inheritance of process or thread attributes not defined by this volume of POSIX.1-2017 is implementation-defined. A call to any exec function from a process with more than one thread shall result in all threads being terminated and the new executable image being loaded and executed. No destructor functions or cleanup handlers shall be called. Upon successful completion, the exec functions shall mark for update the last data access timestamp of the file. If an exec function failed but was able to locate the process image file, whether the last data access timestamp is marked for update is unspecified. Should the exec function succeed, the process image file shall be considered to have been opened with open. The corresponding close shall be considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the exec functions, posix_spawn or posix_spawnp. The argv[] and envp[] arrays of pointers and the strings to which those arrays point shall not be modified by a call to one of the exec functions, except as a consequence of replacing the process image. The saved resource limits in the new process image are set to be a copy of the process'

[12] See alarm.

[13] See umask.

[14] See getrlimit and setrlimit.

[15] See pthread_sigmask.

[16] See sigpending.

[17] See times.

[18] See sigprocmask and pthread_sigmask.

[19] See sigpending.

corresponding hard and soft limits. If one of the exec functions returns to the calling process image, an error has occurred; the return value shall be -1, and errno shall be set to indicate the error.

10.2.3 Waiting for Processes to Terminate.

After spawning a new process, the parent process is free to go about its business. The two processes will be executing at the same time; neither will wait on the other. This is the way the shell starts up a process in the background; it simply spawns a new process which executes the new program and the parent prints another prompt to you. Unfortunately, the above is not always desirable. Often the parent cannot continue until the program the child executes has completed its work. For this reason, the wait system call is provided. The function takes one argument, a pointer to an int which represent the status of the child process. wait suspends execution of its calling process until status information is available for a terminated child process, or a signal is received. On return from a successful wait call, the status area, if non-zero, is filled in with termination information about the process that exited. The *wait4*() call provides a more general interface for programs that need to wait for certain child processes, that need resource utilization statistics accumulated by child processes, or that require options. The other wait functions are implemented using *wait4*(). In the waitpid and wait4 system calls the first argument, the wpid parameter, specifies the set of child processes for which to wait. The following symbolic constants are currently defined in <sys/wait.h>:

```
#define WAIT_ANY (-1) /* any process */
#define WAIT_MYPGRP 0 /* any process in my process group */
```

If the first argument is set to WAIT_ANY, the call waits for any child process. If it is set to WAIT_MYPGRP, the call waits for any child process in the process group of the caller. If it is greater than zero, the call waits for the process with process id equals to the first argument. Finally if is less than -1, the call waits for any process whose process group id equals the absolute value of the first argument. The status parameter is defined below. The options argument is the bitwise OR of zero or more of the following values:

WCONTINUED Causes status to be reported for stopped child processes that have been continued by receipt of a SIGCONT signal.

WNOHANG Indicates that the call should not block if there are no processes that wish to report status.

WUNTRACED If set, children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP signal also have their status reported.

in wait3 and wait4, if the last argument is non-zero, a summary of the resources used by the terminated process and all its children is returned[20]. When the WNOHANG option is specified and no processes wish to report status, wait4 returns a process id of 0. The waitpid call is identical to wait4 with the last argument value of zero. The older wait3 call is the same as wait4 with a first argument value of -1. The following macros may be used to test the manner of exit of the process. One of the first three macros will evaluate to a non-zero (true) value:

WIFCONTINUED(status) True if the process has not terminated, and has continued after a job control stop. This macro can be true only if the wait call specified the WCONTINUED option.

WIFEXITED(status) True if the process terminated normally by a call to *_exit*(2) or *exit*(3).

[20] This information is currently not available for stopped processes.

WIFSIGNALED(status) True if the process terminated due to receipt of a signal.

WIFSTOPPED(status) True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the WUNTRACED option or if the child process is being traced[21].

Depending on the values of those macros, the following macros produce the remaining status information about the child process:

WEXITSTATUS(status) If WIFEXITED(status) is true, evaluates to the low-order 8 bits of the argument passed to *_exit*(2) or *exit*(3) by the child.

WTERMSIG(status) If WIFSIGNALED(status) is true, evaluates to the number of the signal that caused the termination of the process.

WCOREDUMP(status) If WIFSIGNALED(status) is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received.

WSTOPSIG(status) If WIFSTOPPED(status) is true, evaluates to the number of the signal that caused the process to stop. If wait returns due to a stopped or terminated child process, the process id of the child is returned to the calling process. Otherwise, a value of -1 is returned and errno is set to indicate the error.

If wait4, wait3 or waitpid returns due to a stopped or terminated child process, the process id of the child is returned to the calling process. If there are no children not previously awaited, -1 is returned with errno set to ECHILD. Otherwise, if WNOHANG is specified and there are no stopped or exited children, 0 is returned. If an error is detected or a caught signal aborts the call, a value of -1 is returned and errno is set to indicate the error.

Listing 10.1: ezshell - a simple shell program.

```
/* -*- mode: c-mode; -*- */

/* ezshell.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

/* Some general usage macros. */
#define FOREVER for(;;)
#define BUFFER_SIZE 1024
#define ARGS_SIZE 64

/* ezshell program. */
/* Functions prototypes. */
long int execute(char *[]);
void parse(char *, char *[]);
int main(int , char *[]);

/* Main function. */
```

[21] See *ptrace*(2).

```
int main(int argc, char *argv[])
{
  char buff[ BUFFER_SIZE ];
  char *args[ ARGS_SIZE ];
  long int ret = EXIT_SUCCESS;

  /* Main loop. */
  do {

    /* Prompt for read a command. */
    printf("Command:␣");
    if(fgets(buff, BUFFER_SIZE, stdin) != NULL) {

      /* Split the string into arguments. */
      parse(buff, args);
      ret = execute(args);
    } else {
      printf("\n");
      ret = EXIT_FAILURE;
    }
  } while(ret != EXIT_FAILURE);
  exit(ret);
}

/*
 * parse -- split the command in buff into
 *          individual arguments.
 */
void parse(char *buff, char *args[])
{
  while(*buff != ’\0’) {

    /*
     * Strip whitespace.  Use nulls, so
     * that the previous argument is tewrminated
     * automatically.
     */
    while((*buff == ’␣’) || (*buff == ’\t’) || (*buff == ’\n’))
      *buff++ = ’\0’;

    /* Save the argument. */
    *args++ = buff;

    /* Skip over the argument. */
    while((*buff != ’\0’) && (*buff != ’␣’) && (*buff != ’\t’) &&
          (*buff != ’\n’))
      buff++;
  }
  *args = ’\0’;
}

/*
```

```
 * execute -- spawn a child process and execute
 *            the program.
 */
long int execute(char *args[])
{
  int pid, status;
  long int ret = EXIT_FAILURE;

  /* Get a child process. */
  if((pid = fork()) >= 0) {
    if(pid == 0) {
      printf("Executing:␣%s␣with␣pid␣%d\n", *args, pid);
      if(execvp(*args, args) < 0)
        perror("execvp");
      perror(*args);
      ret = EXIT_FAILURE;
    }

    /* The parent executes the wait. */
    while(wait(&status) != pid)

      /* empty ... */
      ;
    ret = EXIT_SUCCESS;
  } else
    perror("fork");
  return ret;
}

/* End of ezshell.c file. */
```

10.3 Redirecting Input and Output.

Listing 10.1 is useful, perhaps even as a very primitive shell. It reads a command name from the standard input and then executes it. Unfortunately, there is no way to make the command read from a file, nor write to one as the real shell does. Fortunately, this is relatively easy to do. Chapter 3, Moving Around in Files., described the dup system call, which could be used to obtain a new file descriptor referring to the same file as its argument. Further, as mentioned above, files stay open across calls to exec and child processes are identical in every way to their parents. This implies that to make a process read and write files instead of the terminal, it is only necessary to open the files and issue the appropriate calls to dup in the child process. Listing 10.2 shows a modified version of the execute routine from Listing 10.1. This routine takes four arguments: the arguments to the program and file descriptors referring to the files which should be used as the new program's standard input, standard output and standard error output. If no file is to be used, the caller of execute can simply pass down 0, 1 or 2 respectively. The program must check, however that it does not inadvertently close one of these descriptors, since the call to dup would the fail, in other words, it is not possible to make dup return its argument.

Listing 10.2: The execute function.

```
/* Functions prototypes. */
long int execute(char *[], int, int, int);
```

```
/*
 * execute -- executes a command in a forked
 *            process.
 */
long int execute(cha *args[], int sin, int sout, int serr)
{
  int pid, status;
  long int ret = EXIT_FAILURE;

  /* Get a child process. */
  if((pid = fork()) >= 0) {

    /* The child executes the code inside the if. */
    if(pid == 0) {

      /*
       * For each of standard input, output,
       * and error output, set the child's
       * to the passed-down file descriptor.
       * Note that we can't just close 0, 1
       * and 2 since we might need them.
       */
      if(sin != 0) {
        close(0);
        dup(sin);
      }
      if(out != 1) {
        close(1);
        dup(sout);
      }
      if(serr != 2) {
        close(2);
        dup(serr);
      }
      if(execvp(*args, args) < 0)
        perror("execvp");
      else {
        perror(*args);
        ret = EXIT_FAILURE;
      }
    }

    /* The parent executes the wait. */
    while(wait(&status) != pid)
      ; /* empty loop... */
    ret = EXIT_SUCCESS;
  } else
      perror("fork");
  return ret;
}
```

10.4 Setting Up Pipelines.

One of the most powerful features of the UNIX operating system and OpenBSD is the ability to construct a pipeline pf commands. This pipeline is set up such that the output of the first command is sent to the input of the second, the output of the second command is sent to the input of the third and so forth. This eliminates the need to run each command separately, saving the intermediate results in temporary files.

10.4.1 The `popen` Library Routine.

One way to create a pipe is to use `popen`. The function "opens" a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the second argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only. The first argument is a pointer to a NUL-terminated string containing a shell command line. This string is passed to */bin/sh* using the -c flag; interpretation, if any, is performed by the shell. The second argument is a pointer to a NUL-terminated string which must be either `"r"` or `"re"` for reading or `"w"` or `"we"` for writing. If the letter `"e"` is present in the string then the close-on-exec flag shall be set on the file descriptor underlying the FILE that is returned. The return value from `popen` is a normal standard I/O stream in all respects except that it must be closed with `pclose` rather than *fclose*(3). Writing to such a stream writes to the standard input of the command; the command's standard output is the same as that of the process that called `popen`, unless this is altered by the command itself. Conversely, reading from a "popened" stream reads the command's standard output, and the command's standard input is the same as that of the process that called `popen`. Note that `popen` output streams are fully buffered by default. In addition, fork handlers established using *pthread_atfork*(3) are not called when a multithreaded program calls `popen`. The `pclose` function waits for the associated process to terminate and returns the exit status of the command as returned by *wait4*(2). The `popen` function returns `NULL` if the *fork*(2) or *pipe*(2) calls fail, or if it cannot allocate memory. The `pclose` function returns -1 if stream is not associated with a "popened" command, if stream already *pclosed*, or if *wait4*(2) returns an error.

10.4.2 Creating Pipes Directly.

The system call to create a pipe is called `pipe`. Which is an object allowing unidirectional data flow, and allocates a pair of file descriptors. The first argument holds an array of two file descriptors: the first connects to the read end of the pipe and the second connects to the write end, so that data written to the second value in the array appears on, i.e., can be read from, the first entry. This allows the output of one program to be sent to another program: the source's standard output is set up to be the write end of the pipe and the sink's standard input is set up to be the read end of the pipe. The pipe itself persists until all its associated descriptors are closed. A pipe whose read or write end has been closed is considered widowed. Writing on such a pipe causes the writing process to receive a `SIGPIPE` signal. Widowing a pipe is the only way to deliver end-of-file to a reader: after the reader consumes any buffered data, reading a widowed pipe returns a zero count. The `pipe2` function is identical to pipe except that the non-blocking I/O mode on both ends of the pipe is determined by the `O_NONBLOCK` flag in the flags argument and the close-on-exec flag on both the new file descriptors is determined by the `O_CLOEXEC` flag in the second argument. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. Listing 10.3 shows a program that opens a pipe to the email program mutt and sends a message to the person executing it. The fdopen function takes a low-level file descriptor and a mode as arguments and returns an stdio file pointer which refers to the same file. This enables programs to use low-level I/O routines for a time and then convert to high-level routines. Note that there is no real need for the parent to wait on the child process to terminate. In fact, deleting the wait has the advantage of making the child run in the background so that the user doesn't have to wait for it to finish. The reader is invited to modify this program to execute other programs and read from the pipe instead of writing or perhaps both.

Listing 10.3: mailer - open a pipe to the mutt command and send email.

```
/* -*- mode: c-mode; -*- */

/* mailer.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

/* Some general usage macros. */
#define FOREVER for(;;)
#define BUFFER_SIZE 1024
#define ARGS_SIZE 64

/* mailer program. */
/* Functions prototypes. */
char *getlogin(void);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  char *username;
  int pid, pipefds[ 2 ];
  long int ret = EXIT_SUCCESS;
  FILE *fp;

  /* Get user's name. */
  if((username = getlogin()) != NULL) {

    /*
     * Create the pipe.  This has to be done
     * BEFORE the fork.
     */
    if(pipe(pipefds) >= 0) {
      if((pid = fork()) >= 0) {

        /*
         * The child process executes  the stuff inside
         * the if.
         */
        if(pid == 0) {

          /*
           * Make the read  side of the pipe our
           * standard input.
           */
          close(STDIN_FILENO);
          dup(pipefds[ 0 ]);
          close(pipefds[ 0 ]);

          /*
           * Close the write  side of the pipe;
           * we'll let our output  go to the screen.
           */
          close(pipefds[ 1 ]);

          /* Execute the command "mutt username". */
          if(execl("/usr/local/bin/mutt", "-s␣\"ERROR␣Messages\"", "myemail@gmail.com", "-a␣logFile.log", NULL) >= 0) {
            ;
          } else
              perror("execl");
        } else {

          /* The parent executes this code. */
          /*
           * Close the read side of the pipe; we
           * don't need it and the child is not
           * writing on the pipe anyway.
           */
          close(pipefds[ 0 ]);

          /* Convert the write side of the pipe to stdio. */
          if((fp = fdopen(pipefds[ 1 ], "w")) != NULL) {

            /* send a message. close the pipe. */
            fprintf(fp, "Errors␣from␣your␣porgram.\n");
            fclose(fp);
            ret = EXIT_SUCCESS;
            while(wait(NULL) != pid)
              ;
          } else {
            perror("fdopen");
          }
        }
      } else {
        perror("fork");
      }
    } else
      perror("pipe");
  } else
      fprintf(stderr, "Who␣are␣you?\n");
  exit(ret);
}

/* End of mailer.c file. */
```

Chapter 11

Job Control

Each *job* is a *process group* and a *process* is a program in execution[1]. The *job control mechanism* provided in OpenBSD system enables a user to control many processes at once. Coupled with the commands provided by the *Korn shell*, called *ksh* and the tty driver, the job control mechanism enables the user to:

- suspend an executing job;
- place that job in the background;
- continue the job's execution;
- return the job to the foreground;
- cause a background job to be stopped when it attempts output to the terminal;
- cause a background job to stop when it tries to read from the terminal.

The chapter describes how the various tasks mentioned above can be performed by user programs. In order to provide a familiar framework on which to base our discussion, we will describe things in terms of ksh commands. The Korn shell, or ksh, was invented by David Korn of AT&T Bell Laboratories in the mid-1980s. It is almost entirely upwardly compatible with the Bourne shell, which means that Bourne shell users can use it right away, and all system utilities that use the Bourne shell can use the Korn shell instead. It began its public life in 1986 as part of AT&T's "Experimental Toolchest", meaning that its source code was available at very low cost to anyone who was willing to use it without technical support and with the knowledge that it might still have a few bugs. Eventually, AT&T's UNIX System Laboratories (USL) decided to give it full support as a UNIX utility. As of USL's version of UNIX called System V Release 4, SVR4 for short (1989), it was distributed with all USL UNIX systems, all third-party versions of UNIX derived from SVR4, and many other versions. The OpenBSD ksh is based on the public domain 7th edition Bourne shell clone by Charles Forsyth and parts of the BRL shell by Doug A. Gwyn, Doug Kingston, Ron Natalie, Arnold Robbins, Lou Salkind and others. The first release of pdksh was created by Eric Gisin, and it was subsequently maintained by John R. MacMillan, Simon J. Gerraty and Michael Rendell.

[1] See [2].

11.1 Preliminary Concepts.

11.1.1 The Controlling Terminal.

When a terminal file, e.g. */dev/tty12*, is opened, it causes the opening process to wait until a connection is established. In practice, user programs rarely open these file directly; they are opened by the *init process* and become a user's standard input and output files. The first terminal file open in a process becomes the *controlling terminal* for that process. The controlling terminal is inherited by a child process during a `fork`, even if the controlling terminal is closed. The file /dev/tty is, in each process, a synonym fo the controlling terminal associated with that process. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected. Certain processes in the system, usually the daemons started at system boot time, clear their controlling terminal using the `ioctl` system call, with `TIOCNOTTY` as the operation constant. The reason for this will become clear later.

11.1.2 Process Groups.

On OpenBSD systems, it is possible to place processes into any arbitrary process group using the `setpgid` system call. The Korn shell uses this call in a straight-forward way; each shell job constitutes a single process group. Each time it starts a process, ksh sets that process's group to the same number as its process id. The process group id is set in both parent and child to deal with race condition. The process group of the current process is returned by `getpgrp`. The process group of the pid process is returned by `getpgid`. If the first argument of `getpid` is zero, the function returns the process group of the current process. Process groups are used for distribution of signals and by terminals to arbitrate requests for their input: processes that have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read. These calls are thus used by programs such as *csh*(1) to create process groups in implementing job control. The `tcgetpgrp` and `tcsetpgrp` calls are used to get/set the process group of the controlling terminal. The process group associated with a terminal may be obtained using the call:

```
ioctl(fd, TIOCGPGRP, &pgrp)
```

where `pgrp` is an integer and `fd` refers to the terminal in question. The terminal's process group may be changed using the `ioctl` system call with `TIOCGPGRP` as the operation constant.

11.1.3 System Calls.

In order to write subroutines that mimic those of ksh, it is necessary to first describe a few of the system calls we will be using. Several of them have been described in detail in previous chapters and we will only mention them briefly here to describe what we plan to use them for.

`ioctl` will be used to initially set the process group of the controlling terminal to the process group of the shell. This is necessary to allow the shell to print prompts, read from the terminal and accept signals. We will also use `ioctl` to change the process group of the terminal to permit a job in another process group to access it, thus putting the job in the foreground.

`setpgid` sets the process group of the specified process pid to the specified value in the second argument. If the first argument is zero, then the call applies to the current process. If the second argument is zero, the process id of the specified process is used.

killpg
: sends the signal in the second argument to the process group specified by the first argument[2]. If the first argument is 0, killpg sends the signal to the sending process's group. The sending process and members of the process group must have the same effective user id or the sender must be the super-user. As a single special case the continue signal SIGCONT may be sent to any process with the same session id as the caller.

wait4
: This call is a much more sophisticated version of the wait system call. it is called as:pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);where wpid parameter specifies the set of child processes for which to wait. The following symbolic constants are currently defined in <sys/wait.h>:

```
#define WAIT_ANY (-1) /* any process */
#define WAIT_MYPGRP 0 /* any process in my process
    group */
```

If wpid is set to WAIT_ANY, the call waits for any child process. If wpid is set to WAIT_MYPGRP, the call waits for any child process in the process group of the caller. If wpid is greater than zero, the call waits for the process with process id wpid. If wpid is less than -1, the call waits for any process whose process group id equals the absolute value of wpid. status is a pointer to type union wait; options is an integer containing a bit mask described below and rusage is an optional pointer of type struct rusage. If non-zero, it will be filled in with resource usage statistics about the child process. The union and the options flags are defined in the include file <sys/wait.h>; the other structure is defined in the include file <sys/resource.h>. As with wait, the process id of the process whose status is being given is returned and -1 is returned when there are no processes that wish to report their status. The flags can be ORed into options:

WCONTINUED
: Causes status to be reported for stopped child processes that have been continued by receipt of a SIGCONT signal.

WHOHANG
: this flag specifies that the call should not block if there are no processes which wish to report their status. This enables a process to check for any processes whose status has changed and then go on to something else if there are none;

WUNTRACED
: if set, children of the current process that are stopped due to a SIGTTIN, SIGTTOU, SIGTSTP or SIGSTOP signal also have their status reported.

There are also four macros defined; each takes a single argument:

WIFCONTINUED(status)
: True if the process has not terminated, and has continued after a job control stop. This macro can be true only if the wait call specified the WCONTINUED option.

WIFEXITED(status)
: True if the process terminated normally by a call to _*exit*(2) or *exit*(3).

WIFSIGNALED(status)
: True if the process terminated due to receipt of a signal.

[2] See *sigaction*(2) for a list of signals.

`WIFSTOPPED(status)` True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the `WUNTRACED` option or if the child process is being traced[3].

Depending on the values of those macros, the following macros produce the remaining status information about the child process:

`WEXITSTATUS(status)` if `WIFEXITED(status)` is true, evaluates to the low-order 8 bits of the argument passed to _*exit*(2) or *exit*(3) by the child.

`WTERMSIG(status)` If `WIFSIGNALED(status)` is true, evaluates to the number of the signal that caused the termination of the process.

`WCOREDUMP(status)` If `WIFSIGNALED(status)` is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received.

`WSTOPSIG(status)` If `WIFSTOPPED(status)` is true, evaluates to the number of the signal that caused the process to stop.

11.1.4 The job and process Data Types.

In the include file `<sys/proc.h>` the `struct pgrp` is defined:

Listing 11.1: The pgrp structure.

```
struct pgrp {
  LIST_ENTRY(pgrp) pg_hash;
  LIST_HEAD(, process) pg_members;
  struct session *pg_session;
  struct sigiolst pg_sigiolst;
  pid_t pg_id;
  int pg_jobc;
};
```

The structure members are:

`pg_hash`	hash chain;
`pg_members`	pointer to pgrp members;
`pg_session`	pointer to session;
`pg_sigiolst`	list of sigio structures;
`pg_id`	pgrp id;
`pj_jobc`	procs qualifying pgrp for job control.

and the `struct process` is defined as follow:

[3] See *ptrace*(2).

Listing 11.2: The process structure.

```
struct process {
  struct proc *ps_mainproc;
  struct ucred *ps_ucred;
  LIST_ENTRY(process) ps_list;
  TAILQ_HEAD(,proc) ps_threads;
  LIST_ENTRY(process) ps_pglist;
  struct process *ps_pptr;
  LIST_ENTRY(process) ps_sibling;
  LIST_HEAD(, process) ps_children;
  LIST_ENTRY(process) ps_hash;
  LIST_ENTRY(process) ps_orphan;
  LIST_HEAD(, process) ps_orphans;
  struct sigiolst ps_sigiolst;
  struct sigacts *ps_sigacts;
  struct vnode *ps_textvp;
  struct filedesc *ps_fd;
  struct vmspace *ps_vmspace;
  pid_t ps_pid;
  struct futex_list ps_ftlist;
  struct tslpqueue ps_tslpqueue;
  struct rwlock ps_lock;
  struct mutex ps_mtx;
  struct klist ps_klist;
  u_int ps_flags;
  int ps_siglist;
  struct proc *ps_single;
  u_int ps_singlecount;
  int ps_traceflag;
  struct vnode *ps_tracevp;
  struct ucred *ps_tracecred;
  u_int ps_xexit;
  int ps_xsig;
  pid_t ps_ppid;
  pid_t ps_oppid;
  int ps_ptmask;
  struct ptrace_state *ps_ptstat;
  struct rusage *ps_ru;
  struct tusage ps_tu;
  struct rusage ps_cru;
  struct itimerspec ps_timer[ 3 ];
  struct timeout ps_rucheck_to;
  time_t ps_nextxcpu;
  u_int64_t ps_wxcounter;
  struct unveil *ps_uvpaths;
  ssize_t ps_uvvcount;
  size_t  ps_uvncount;
  int ps_uvdone;
  struct plimit *ps_limit;
  struct pgrp *ps_pgrp;
  char ps_comm[ _MAXCOMLEN ];
  vaddr_t ps_strings;
  vaddr_t ps_auxinfo;
  vaddr_t ps_timekeep;
  vaddr_t ps_sigcode;
  vaddr_t ps_sigcoderet;
  u_long  ps_sigcookie;
  u_int ps_rtableid;
  char ps_nice;
  struct uprof {
    caddr_t pr_base;
    size_t pr_size;
    u_long pr_off;
    u_int pr_scale;
  } ps_prof;
  u_int32_t ps_acflag;
  uint64_t ps_pledge;
  uint64_t ps_execpledge;
  int64_t ps_kbind_cookie;
  u_long ps_kbind_addr;
  struct pinsyscall ps_pin;
  struct pinsyscall ps_libcpin;
  u_int ps_threadcnt;
  struct timespec ps_start;
  struct timeout ps_realit_to;
};

#define ps_startzero ps_klist
#define ps_endzero ps_startcopy
#define ps_startcopy ps_limit
#define BOGO_PC (u_long) -1
#define ps_endcopy ps_threadcnt
```

ps_mainproc is the original thread in the process. It's only still special for the handling of some signal and ptrace behaviors that need to be fixed;

ps_ucred process owner's identity;

ps_list list of all processes;

ps_threads [K | S] threads in this process;

ps_pglist list of processes in pgrp;

ps_pptr pointer to parent process;

ps_sibling list of sibling processes;

ps_children pointer to list of children;

ps_hash hash chain;

ps_orphan list of orphan processes. An orphan is the child that has been re-parented to the debugger as a result of attaching to it. Need to keep track of them for parent to be able to collect the exit status of what used to be children;

ps_orphans pointer to list of orphans;

`ps_sigiolst`	list of sigio structures;
`ps_sigacts`	[I] signal actions, state;
`ps_textvp`	vnode of executable;
`ps_fd`	pointer to open files structure;
`ps_vmspace`	address space;
`ps_pid`	process identifier;
`ps_ftlist`	futexes attached to this process;
`ps_tslpqueue`	[p] queue of threads in thrsleep;
`ps_lock`	per-process rwlock;
`ps_mtx`	per-process mutex;

The following fields are all zeroed upon creation in process_new:

`ps_klist`	knotes attached to this process;
`ps_flags`	[a] PS_* flags;
`ps_siglist`	signals pending for the process;
`ps_single`	[S] thread for single-threading;
`ps_singlecount`	[a] not yet suspended threads;
`ps_traceflag`	kernel trace points;
`ps_tracevp`	trace to vnode;
`ps_tracecred`	creds for writing trace;
`ps_xexit`	exit status for wait;
`ps_xsig`	stopping or killing signal;
`ps_ppid`	[a] cached parent pid;
`ps_oppid`	[a] save parent pid during ptrace;
`ps_ptmask`	ptrace event mask;
`ps_ptstat`	ptrace state;
`ps_ru`	sum of stats for dead threads;
`ps_tu`	accumulated times;
`ps_cru`	sum of stats for reaped children;
`ps_timers`	[m] `ITIMER_REAL` timer;
`ps_rucheck_to`	[ ] resource limit check timer;
`ps_nextxcpu`	when to send next `SIGXCPU` in seconds of process runtime;
`ps_wxcounter`	—;

`ps_uvpaths` unveil vnodes and names;

`ps_uvvcount` count of unveil vnodes held;

`ps_uvncount` count of unveil names allocated;

`ps_uvdone` no more unveil is permitted;

The following fields are all copied upon creation in process_new:

`ps_limit` [m, R] process limits;

`ps_pgrp` pointer to process group;

`ps_comm` command name, incl NUL;

`ps_strings` user pointers to argv/env;

`ps_auxinfo` user pointer to auxinfo;

`ps_timekeep` user pointer to timekeep;

`ps_sigcode` [I] user pointer to signal code;

`ps_sigcoderet` [I] user ptr to sigreturn retPC;

`ps_sigcookie` [I]

`ps_rtableid` [a] process routing table/domain;

`ps_nice` process *nice* value;

`ps_prof` are the profile argument organized in a struct:

- `pr_base` — buffer base;
- `pr_size` — buffer size;
- `pr_off` — pc offset;
- `pr_scale` — pc scaling.

`ps_acflag` accounting flags;

`ps_pledge` [m] pledge promises;

`ps_execpledge` [m] execpledge promises;

`ps_kbind_cookie` [m];

`ps_kbind_addr` [m];

`ps_pin` static or ld.so;

`ps_libcpin` libc.so, from *pinsyscalls*(2);

`ps_threadcnt` number of threads;

`ps_start` starting uptime;

`ps_realit_to` [m] `ITIMER_REAL` timeout;

In the same file we also have the `struct proc`:

Listing 11.3: The proc structure.

```
struct proc {
  TAILQ_ENTRY(proc) p_runq;
  LIST_ENTRY(proc) p_list;
  struct  process *p_p;
  TAILQ_ENTRY(proc) p_thr_link;
  TAILQ_ENTRY(proc) p_fut_link;
  struct futex *p_futex;
  struct filedesc *p_fd;
  struct vmspace *p_vmspace;
  struct p_inentry p_spinentry;
  struct p_inentry p_pcinentry;
  int p_flag;
  u_char p_spare;
  char p_stat;
  u_char p_runpri;
  u_char p_descfd;
  pid_t p_tid;
  LIST_ENTRY(proc) p_hash;
  int p_dupfd;
  int p_cpticks;
  const volatile void *p_wchan;
  struct timeout p_sleep_to;
  const char *p_wmesg;
  fixpt_t p_pctcpu;
  u_int p_slptime;
  u_int p_uticks;
  u_int p_sticks;
  u_int p_iticks;
  struct cpu_info *volatile p_cpu;
  struct rusage p_ru;
  struct tusage p_tu;
  struct plimit *p_limit;
  struct kcov_dev *p_kd;
  struct lock_list_entry *p_sleeplocks;
  struct kqueue *p_kq;
  int p_siglist;
  sigset_t p_sigmask;
  char p_name[ _MAXCOMLEN ];
  u_char p_slppri;
  u_char p_usrpri;
  u_int p_estcpu;
  int p_pledge_syscall;
  struct ucred *p_ucred;
  struct sigaltstack p_sigstk;
  u_long p_prof_addr;
  u_long p_prof_ticks;
  struct user *p_addr;
  struct mdproc p_md;
  sigset_t p_oldmask;
  int p_sisig;
  union sigval p_sigval;
  long p_sitrapno;
  int p_sicode;
};
```

The meanings of the members of this structure are:

p_runq	[S] current run/sleep queue;
p_list	list of all threads;
p_p	[I] the process of this thread;
p_thr_link	threads in a process linkage;
p_fut_link	threads in a futex linkage;
p_futex	current sleeping futex;
p_fd	copy of p_p -> ps_fd;
p_vmspace	[I] copy of p_p -> ps_vmspace;
p_spinentry	[o] cache for SP check;
p_pcinentry	[o] cache for PC check;
p_flag	P_* flags;
p_spare	unused;
p_stat	[S] S* process status;
p_runpri	[S] runqueue priority;
p_descfd	if not 255, fdesc permits this fd;
p_tid	thread identifier;
p_hash	hash chain;
p_dupfd	sideways return value from filedescopen. XXX;
p_cpticks	ticks of cpu time;
p_wchan	[S] sleep address;
p_sleep_to	timeout for tsleep;
p_wmesg	[S] reason for sleep;
p_pctcpu	[S] %cpu for this thread;
p_slptime	[S] time since last blocked;
p_uticks	statclock hits in user mode;
p_sticks	statclock hits in system mode;
p_iticks	statclock hits processing intr;
p_cpu	[S] CPU we're running on;
p_ru	statistics;

`p_tu`	accumulated times;
`p_limit`	[l] read ref. of p_p -> ps_limit;
`p_kd`	kcov device handle;
`p_sleeplocks`	WITNESS lock tracking;
`p_kq`	[o] select/poll queue of evts;
`p_siglist`	[a] signals arrived and not delivered;
`p_sigmask`	[a] current signal mask;
`p_name`	thread name, incl NUL;
`p_slppri`	[S] sleeping priority;
`p_usrpri`	[S] priority based on p_estcpu and ps_nice;
`p_estcpu`	[S] time averaged val of p_cpticks;
`p_pledge_syscall`	cache of current syscall;
`p_ucred`	[o] cached credentials;
`p_sigstk`	sp and on stack state variable;
`p_prof_addr`	temporary storage for profiling address until AST;
`p_prof_ticks`	temporary storage for profiling ticks until AST;
`p_addr`	kernel virtual addr of u-area;
`p_md`	any machine-dependent fields;
`p_oldmask`	saved mask from before sigpause;
`p_sisig`	for core dump/debugger;
`p_sigval`	for core dump/debugger;
`p_sitrapno`	for core dump/debugger;
`p_sicode`	for core dump/debugger.

11.1.5 Using kernel to retrieve processes informations.

The `sysctl` system call is used to retrieve kernel informations about various topics. We can use it to find the processes running on the system. In the listing 11.4 is showed a program to get the list of processes:

Listing 11.4: getprocs - retrieve informations on processes.

```c
/* -*- mode: c-mode; -*- */

/* getprocs.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

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

/* getprocs program. */
#define TRUE  1
#define FALSE 0

/* Functions prototypes. */
struct kinfo_proc *getprocs(int *, int);
long int showinfo(int);
int main(int, char *[]);

/* Main function. */
int main(int argc, char * argv[])
{
  long int ret = EXIT_FAILURE;

  /* Check arguments. */
  if(argc == 1) {
    exit(showinfo(FALSE));
  } else if((argc == 2) && \
        (!strncmp(argv[1], "-t", 3) || !strncmp(argv[1], "--threads", 10))) {
        exit(showinfo(TRUE));
  } else {
    printf( "Usage:\n" ) ;
    printf( "␣␣␣␣␣␣list␣[-h]␣[-t]\n\n" ) ;
    printf( "Options:\n" ) ;
    printf( "␣␣␣␣␣␣-h,␣--help␣␣␣␣␣␣␣␣␣␣␣␣Print␣this␣information\n" ) ;
    printf( "␣␣␣␣␣␣-t,␣--threads␣␣␣␣␣␣␣␣␣Show␣threads\n\n" ) ;
    ret = EXIT_SUCCESS;
  }
  exit(ret);
}

/*
 * getprocs -- retrieve the list of processes.
 */
struct kinfo_proc *getprocs(int *count, int threads)
{
  struct kinfo_proc *procbase = NULL ;
  unsigned int maxslp ;
  size_t size = sizeof(maxslp) ;
  int maxslp_mib[] = {
    CTL_VM,
    VM_MAXSLP
  };
  int mib[ 6 ] = {
    CTL_KERN,
    KERN_PROC,
    threads ? KERN_PROC_KTHREAD | KERN_PROC_SHOW_THREADS : KERN_PROC_KTHREAD,
    0,
    sizeof(struct kinfo_proc),
    0
  };
  if(sysctl(maxslp_mib, 2, &maxslp, &size, NULL, 0) == -1) {
    perror("list");
    return NULL;
  }

  retry:
  if(sysctl(mib, 6, NULL, &size, NULL, 0 ) == -1) {
    perror("list") ;
    return NULL;
  }
  size = 5 * size / 4;           /* extra slop */
  procbase = (struct kinfo_proc *) malloc(size);
  if(procbase == NULL) {
    perror("list") ;
    return NULL;
  }
  mib[ 5 ] = (int) (size / sizeof(struct kinfo_proc));
  if(sysctl(mib, 6, procbase, &size, NULL, 0)) {
    if(errno == ENOMEM) {
      free(procbase);
      goto retry;
    }
    perror("list") ;
    return NULL;
  }
  *count = (int) (size / sizeof(struct kinfo_proc));
  return procbase;
}

/*
 * showinfo -- show informations about threads.
 */
long int showinfo(int threads)
{
  struct kinfo_proc *list, *proc;
  int count, i ;

  /* */
  if((list = getprocs(&count, threads)) == NULL) {
    return EXIT_FAILURE;
  }
  proc = list ;
  if(threads) {
    for(i = 0; i < count; ++i, ++proc) {
      if(proc -> p_tid != -1) {
        printf("%s:␣pid:␣%d␣(tid:␣%d)\n", proc -> p_comm, proc ->
             p_pid, proc -> p_tid);
      }
    }
  } else {
    for(i = 0; i < count; ++i, ++proc) {
      printf("%s:␣pid:␣%d\n", proc -> p_comm, proc->p_pid ) ;
    }
  }
  return EXIT_SUCCESS;
}

/* End of getprocs.c file. */
```

11.2 Job Control in the Shell.

This section describes the various parts of job control that are handled primarily by the shell. This includes moving processes from foreground to background and back, suspending process in mid-execution and so on.

11.2.1 Setting Up for Job Control.

In order to perform job control, it is necessary to set up the environment. This set-up is done by the shell when it is first invoked and includes setting the shell's process group and then setting the terminal's process group. Listing 11.5 shows how this might be done.

Listing 11.5: setupjc - setup for job control.

```
/* -*- mode: c-mode; -*- */

/* setupjc.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

/* Some general usage macros. */
#define FOREVER for(;;)
#define BUFFER_SIZE 1024
#define ARGS_SIZE 64

/* Global variables. */
int npid;
int npgrp;
int ntermpgrp;

/* Functions prototypes. */
void setup(void);

/* setup function. */
void setup(void)
{
  /* Obtain shell's process id. */
  npid = getpid();

```

```
    /*
     * Just use pid for process group.  This is
     * not a requirement, just convenient.  Other
     * ways of picking a process group can be used.
     */
    npgrp = npid;
    ntermpgrp = npid;

    /* Set the shell's process group. */
    if(setpgid(npid, npgrp) >= 0) {
      if(ioctl(1, TIOCSPGRP, &npgrp) >= 0) {
        ;
      } else {
        perror("ioctl");
        exit(EXIT_FAILURE);
      }
    } else {
      perror("getpgid");
      exit(EXIT_FAILURE);
    }
}

/* End of setupjc.c file. */
```

11.2.2 Executing a Program.

When executing a program, the shell performs something similar to what is done in Listing 10.2. The actual routine handles more complex things than the example; in particular, the routine is recursive after a fashion in order to handle building pipelines. The important thing about executing programs, though, is that after the first child has been spawned, the child whose process id will become the process group for this job, the terminal must be placed in this process group. If this is not done, the program will not be executing in the foreground, of course this is what is wanted if the command line contained an ampersand on the end. It is not terribly important wheter the parent or the child sets the process group, as long as it gets done. In ksh, the parent shell handles this.

11.2.3 Stopping a Job.

Job control refers to the shell's ability to monitor and control jobs, which are processes or groups of processes created for commands or pipelines. At a minimum, the shell keeps track of the status of the background[4] jobs that currently exist; this information can be displayed using the *jobs* commands. If job control is fully enabled, using *set -m* or *set -o monitor*, as it is for interactive shells, the processes of a job are placed in their own process group. Foreground jobs can be stopped by typing the suspend character from the terminal, normally ^Z, jobs can be restarted in either the foreground or background using the *fg* and *bg* commands, and the state of the terminal is saved or restored when a foreground job is stopped or restarted, respectively. When an attempt is made to exit the shell while there are jobs in the stopped state, the shell warns the user that there are stopped jobs and does not exit. If another attempt is immediately made to exit the shell, the stopped jobs are sent a SIGHUP signal and the shell exits. Similarly, if the nohup option is not set and there are running jobs when an attempt is made to exit a login shell, the shell warns the user

[4] I.e. asynchronous.

and does not exit. If another attempt is immediately made to exit the shell, the running jobs are sent a SIGHUP signal and the shell exits. Listing 11.6 shows how to make a child quit:

Listing 11.6: stopproc - make a child process quit.

```
/* -*- mode: c-mode; -*- */

/* stopproc.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/proc.h>
#include <sys/wait.h>

/* stopproc program. */
#define FOREVER for(;;)

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char * argv[])
{
  int pgrp;
  int status;
  long int ret = EXIT_FAILURE;
  pid_t pid;
  struct sigaction signal = {
    SIG_IGN,
    SIGQUIT,
  };

  /* fork */
  if((pid = fork()) == 0) {

    /* Child execute code if pid == 0. */
    printf("Child␣executed!\n");
    FOREVER {
      ;
    }
    _exit(EXIT_SUCCESS);
  } else {

    /* Parent executes otherwise. */
    if(sigaction(SIGQUIT, &signal, NULL) >= 0) {
      pgrp = getpgrp();
      printf("Parent␣waiting␣5␣seconds␣before␣make␣its␣child␣quit.\n");
      sleep(5);
      if(killpg(pgrp, SIGQUIT) >= 0) {
        printf("Parent␣make␣its␣child␣quit.\n");
        while(wait(&status) != pid)
          ;
        printf("Child␣quitted!\n");
        ret = EXIT_SUCCESS;
      } else
        perror("killpg");
    } else
      perror("sigaction");
  }
  exit(ret);
}

/* End of stopproc.c file. */
```

In fact using killpg would kill parent and child which are in the same process group. The mechanism is to make the parent ignore the SIGQUIT and then let it propagate to all child processes to kill them. In this case we have only one child. Do not use SIGKILL and SIGSTOP for this, since they cannot be ignored[5].

11.2.4 Backgrounding and Foregrounding a Job.

There are two ways to place a job in the background. The first is by placing an ampersand "&" at the end of the command string when the command is first entered. Since this case is handled when the processes are started and has little if anything to do with job control, it is not described further here. The second method, using the bg command, involves sending a "continue" signal to the job. Because the job is not in the foreground, otherwise the bg command could not have been read by the shell, no process group manipulation is necessary. Bringing a job in the foreground is more complex that putting it in the background. Because the job is not in the process group of the terminal, the terminal's process group must be changed. If the job is in the stop state it must be started first. The code fragment 11.7 is taken by the system sources at */usr/src/bin/ksh/jobs.c*.

Listing 11.7: j_resume - the ksh bg/fg commands default function.

```
/* fg and bg built-ins: called only if Flag(FMONITOR) set */
int j_resume(const char *cp, int bg)
{
  Job       *j;
  Proc      *p;
  int       ecode;
  int       running;
  int       rv = 0;
  sigset_t omask;

  sigprocmask(SIG_BLOCK, &sm_sigchld, &omask);

  if ((j = j_lookup(cp, &ecode)) == NULL) {
    sigprocmask(SIG_SETMASK, &omask, NULL);
    bi_errorf("%s:␣%s", cp, lookup_msgs[ecode]);
    return 1;
```

[5] See *sigaction*(2).

```
  }

  if (j->pgrp == 0) {
    sigprocmask(SIG_SETMASK, &omask, NULL);
    bi_errorf("job␣not␣job-controlled");
    return 1;
  }

  if (bg)
    shprintf("[%d]␣", j->job);

  running = 0;
  for (p = j->proc_list; p != NULL; p = p->next) {
    if (p->state == PSTOPPED) {
      p->state = PRUNNING;
      p->status = 0;
      running = 1;
    }
    shprintf("%s%s", p->command, p->next ? "|␣" : "");
  }
  shprintf("\n");
  shf_flush(shl_stdout);
  if (running)
    j->state = PRUNNING;

  put_job(j, PJ_PAST_STOPPED);
  if (bg)
    j_set_async(j);
  else {
    /* attach tty to job */
    if (j->state == PRUNNING) {
      if (ttypgrp_ok && (j->flags & JF_SAVEDTTY))
  tcsetattr(tty_fd, TCSADRAIN, &j->ttystate);
      /* See comment in j_waitj regarding saved_ttypgrp. */
      if (ttypgrp_ok &&
    tcsetpgrp(tty_fd, (j->flags & JF_SAVEDTTYPGRP) ?
        j->saved_ttypgrp : j->pgrp) == -1) {
  if (j->flags & JF_SAVEDTTY)
    tcsetattr(tty_fd, TCSADRAIN, &tty_state);
  sigprocmask(SIG_SETMASK, &omask, NULL);
  bi_errorf("1st␣tcsetpgrp(%d,␣%d)␣failed:␣%s",
      tty_fd,
      (int) ((j->flags & JF_SAVEDTTYPGRP) ?
       j->saved_ttypgrp : j->pgrp),
      strerror(errno));
  return 1;
      }
    }
    j->flags |= JF_FG;
    j->flags &= ~JF_KNOWN;
    if (j == async_job)
      async_job = NULL;
  }

  if (j->state == PRUNNING && killpg(j->pgrp, SIGCONT) == -1) {
    int     err = errno;

    if (!bg) {
      j->flags &= ~JF_FG;
      if (ttypgrp_ok && (j->flags & JF_SAVEDTTY))
  tcsetattr(tty_fd, TCSADRAIN, &tty_state);
      if (ttypgrp_ok && tcsetpgrp(tty_fd, our_pgrp) == -1) {
  warningf(true,
     "fg:␣2nd␣tcsetpgrp(%d,␣%d)␣failed:␣%s",
     tty_fd, (int) our_pgrp,
     strerror(errno));
      }
    }
    sigprocmask(SIG_SETMASK, &omask, NULL);
    bi_errorf("cannot␣continue␣job␣%s:␣%s",
        cp, strerror(err));
    return 1;
  }
  if (!bg) {
    if (ttypgrp_ok) {
      j->flags &= ~(JF_SAVEDTTY | JF_SAVEDTTYPGRP);
    }
    rv = j_waitj(j, JW_NONE, "jw:resume");
  }
  sigprocmask(SIG_SETMASK, &omask, NULL);
  return rv;
}
```

11.2.5 The *jobs* Command.

In the Korn shell the *jobs* command is used to print the status of all running jobs. If no jobs are specified, all jobs are displayed. The -n option causes information to be displayed only for jobs that have changed state since the last notification. If the -l option is used, the process ID of each process in a job is also listed. The -p option causes only the process group of each job to be printed. See Job control below for the format of job and the displayed job.

11.2.6 Waiting for Jobs.

The task of waiting for jobs to complete is give to the `wait4` system call. Not only do we find out about jobs that have exited, but we also find out about those that have changed their status.

11.2.7 Asynchronous Process Notification.

11.3 Job Control Outside the Shell.

As mentioned previously, processes that are not in the distinguished process group are not permitted to read from terminal. In OpenBSD the process receives a `SIGTTIN` signal which causes it to stop. The shell can then be used to place the job in the foreground and the read can be satisfied.

Processes are normally allowed to write to the terminal regardless of whether or not they are in the foreground.

11.4 Important Points.

There are several important points to notice from this chapter and its examples:

- the examples in this chapter are for demonstration purposed only. They will work well enough as a demonstration, but they would not be suitable for incorporation into real shell program. In order to do this, it would be necessary to protect several areas of the code from interruption by signals, in particular, since the `SIGCHLD` handler works on the same data structures as the other routines, `SIGCHLD` must be ignored when modifying these structures, built-in commands would have to be handled specially, such as interruption of shell procedures, stopping a process which was executed from inside a shell construct such as foreach loop causes the rest of the loop to to be aborted and so on;

- throughout the examples, whenever a process needed to be placed in the same process group as the terminal, it was always the terminal process group that was changed. An alternative method would have been to use setpgrp to change the process group of the process. There is, however, a reason for changing the terminal's process group and not the process's: if the process uses its own process group for something and obtains that information via getpgrp, the if the shell changes the process's process group that information will no longer be accurate. For this reason, it is always the terminal's process group that is changed;

- in Chapter 9, The Signal Stack., we mentioned that the shell will ignore `SIGINT` and `SIGQUIT` in processes that it places in the background. This is not desirable when in a job control environment, since there is no way, when bringing the job into the foreground, to cause these signals not to be ignored anymore. Fortunately, it is not necessary to ignore these signals in background processes when working with the tty driver. Recall that signals generated from the keyboard are sent only to the process in the process group of the terminal. Since background processes are not in this process group, they will not receive the signal anyway. However, when they are placed into the foreground, the interrupt keys will work correctly, since the background processes are not ignoring the signals themselves.

Job control is a very useful feature to have in OpenBSD system; unfortunately the implementation is rather complex. Generally speaking, there is no way to implement *part* of the job control, it's an all-or-nothing prospect.

Chapter 12

Interprocess Communication.

Sockets.
Message Queues.
Semaphores.
Shared Memory.

The *interprocess communication*, ipc, facilities of OpenBSD system allow two or more distinct processes to communicate with each other. We have already discussed one form of ipc, the pipe. This mechanism allows two related processes, one of which must be a descendant of the other, to communicate over a two-way byte stream using the read and write system calls. OpenBSD provide more powerful ipc facilities that allow two or more completely unrelated process to communicate with each other: semaphores, shared memory, messages queues and sockets. Each of these mechanism, while powerful in its own area, tends to be rather restrictive in the types of uses to which it can be put. In OpenBSD the socket is a generalization of the pipe mechanism for which is, in fact, implemented as a pair of connected sockets. The socket are described in the book [4] and the ipc in the book [3].

12.1 Sockets.

Interprocess communication beyond the scope of the pipe mechanism can normally be described using *client/server* model. In this model, one process is called the *server*; it is responsible for satisfying requests put to it by the other process, the *client*. As an example, consider a program tha manages all the printer queues on a machine. This program would be called a server. When a user prints a file, the printing program, the client, contacts the server and asks it to put the file into the queue for the specified printer. The server does this and then invokes the appropriate program to actually print the file on the printer. Normally, when a server program is invoked, it asks the operating system for a socket. When it gets one, it assigns a well-known name to that socket, so that other programs can ask the operating system to talk to that name, since they will not known the integer value of the socket itself. After naming the socket, the server listens on the socket for connection requests from client processes to come in. When a connection request arrives, the server may accept or reject the connection. If it accepts the connection, the operating system joins the client and server together at the socket and the server may read and write data to and from the socket just as if it were a pipe to the client. The client begins the process by asking that the socket be connected to some other socket having a given name. The operating system attempts to find a socket with the given name and if it does, sends the process listening to that socket a connection request. If the server accepts the connection, the operating system joins the two processes together at the socket and the client can read and write data to and from the socket just as if it were a pipe to the server.

12.1.1 The socket System Call.

socket creates an endpoint for communication and returns a descriptor. It takes three arguments: the first is an integer which is the domain, it specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file <sys/socket.h>. The currently understood formats are:

AF_UNIX UNIX internal protocols

AF_INET Internet Protocol version 4 (IPv4) protocol family

AF_INET6 Internet Protocol version 6 (IPv6) protocol family

The second argument is the socket type, which specifies the semantics of communication. Currently defined types are:

- SOCK_STREAM;
- SOCK_DGRAM;
- SOCK_RAW;
- SOCK_SEQPACKET.

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports *datagrams*, connectionless, unreliable messages of a fixed, typically small, maximum length. A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for AF_UNIX. SOCK_RAW sockets provide access to internal network protocols and interfaces and are available only to the super-user. Any combination of the following flags may additionally be used in the type argument:

SOCK_CLOEXEC Set close-on-exec flag on the new descriptor.

SOCK_NONBLOCK Set non-blocking I/O mode on the new socket.

SOCK_DNS For domains AF_INET or AF_INET6, only allow *connect*(2), *sendto*(2) or *sendmsg*(2) to the DNS port, typically 53.

The third argument is the protocol which specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. This argument specifies the protocol number to use and it is particular to the "communication domain" in which communication is to take place[1]. A value of 0 for this argument will let the system select an appropriate protocol for the requested socket type. Sockets of type SOCK_STREAM are full-duplex byte streams. A *stream socket* must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed, a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2). The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered

[1] See *protocols*(5).

broken and calls will indicate an error with -1 returns and with `ETIMEDOUT` as the specific code in the global variable `errno`. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period, e.g., 5 minutes. A `SIGPIPE` signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

`SOCK_SEQPACKET` sockets employ the same system calls as `SOCK_STREAM` sockets. The only difference is that read(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

`SOCK_DGRAM` and `SOCK_RAW` sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recvfrom*(2), which returns the next datagram with its return address.

An *fcntl*(2) call can be used to specify a process group to receive a `SIGURG` signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via `SIGIO`. The operation of sockets is controlled by socket level options. These options are defined in the file `<sys/socket.h>`. *setsockopt*(2) and *getsockopt*(2) are used to set and get options, respectively. If successful, `socket` returns a non-negative integer, the *socket file descriptor*. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

12.1.2 The `send` and `recv` System Calls.

The `send` system call.

The `send` function shall initiate transmission of a message from the specified socket to its peer and it shall send a message only when the socket is connected. If the socket is a connectionless-mode socket, the message shall be sent to the pre-specified peer address. The `send` function takes four arguments: the first is the socket file descriptor. The second is the pointer to the buffer containing the message to send. The third specifies the length of the message in bytes and the last specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

`MSG_EOR` terminates a record, if supported by the protocol;

`MSG_OOB` sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific;

`MSG_NOSIGNAL` Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The `EPIPE` error shall still be returned.

The length of the message to be sent is specified by the third argument: if the message is too long to pass through the underlying protocol, `send` shall fail and no data shall be transmitted. Successful completion of a call to `send` does not guarantee delivery of the message. A return value of -1 indicates only locally-detected errors. If space is not available at the sending socket to hold the message to be transmitted, and the socket file descriptor does not have `O_NONBLOCK` set, send shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted, and the socket file descriptor does have `O_NONBLOCK` set, send shall fail. The `select` and `poll` functions can be used to determine when it is possible to send more data. The socket in use may require the process to have appropriate privileges to use the `send` function. Upon successful completion, `send` shall return the number of bytes sent. Otherwise, -1 shall be returned and `errno` set to indicate the error.

The recv system call.

The recv function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data. The recv function takes the following arguments: the first argument is the socket which specifies the socket file descriptor. The second argument is the pointer to the buffer where the message should be stored. The third is the length which specifies the length in bytes of the buffer pointed to by the buffer argument. The fourth argument specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK peeks at an incoming message. The data is treated as unread and the next recv or similar function shall still return this data;

MSG_OOB requests out-of-band data. The significance and semantics of out-of-band data are protocol- specific;

MSG_WAITALL on SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

The recv function shall return the length of the message written to the buffer pointed to by the second argument. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the flags argument, the excess bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded. If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message. If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recv shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, recv shall fail and set errno to EAGAIN or EWOULDBLOCK. Upon successful completion, recv shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recv shall return 0. Otherwise, -1 shall be returned and errno set to indicate the error.

12.1.3 The listen System Call.

The listen function shall mark a connection-mode socket, specified by the first argument, as accepting connections. The second argument provides a hint which the implementation shall use to limit the number of outstanding connections in the socket's listen queue. Implementations may impose a limit on backlog and silently reduce the specified value. Normally, a larger backlog argument value shall result in a larger or equal length of the *listen queue*. Implementations shall support values of backlog up to SOMAXCONN, defined in <sys/socket.h>. The implementation may include incomplete connections in its listen queue. The limits on the number of incomplete connections and completed connections queued may be different. The implementation may have an upper limit on the length of the listen queue – either global or per accepting socket. If the second argument exceeds this limit, the length of the listen queue is set to the limit. If listen is called with the second argument value that is less than 0, the function behaves as if it had been called with an argument value of 0. The third argument equal to 0 may allow the socket to accept connections, in which case the length of the listen queue may be set to an implementation-defined minimum value. The socket in use may require the process to have appropriate privileges to use the listen function. Upon successful completions, listen shall return 0; otherwise, -1 shall be returned and errno set to indicate the error.

12.1.4 The shutdown System Call.

The shutdown function shall cause all or part of a full-duplex connection on the socket associated with the file descriptor socket to be shut down. The shutdown function takes the following arguments: the first argument specifies the file descriptor of the socket and the second argument specifies the type of shutdown. The values for this argument are as follows:

SHUT_RD disables further receive operations;

SHUT_WR disables further send operations;

SHUT_RDWR disables further send and receive operations.

The shutdown function disables subsequent send and/or receive operations on a socket, depending on the value of the second argument. Upon successful completion, shutdown shall return 0; otherwise, -1 shall be returned and errno set to indicate the error. The close system call could be used to close a socket. If its first argument refers to a socket, close shall cause the socket to be destroyed. If the socket is in connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time and the socket has untransmitted data, then close shall block for up to the current linger interval until all data is transmitted.

12.1.5 The accept System Call.

Server process use this function call to accept a connection on the socket. The accept function shall extract the first connection on the queue of pending connections, create a new socket with the same socket type protocol and address family as the specified socket, and allocate a new file descriptor for that socket. The accept function takes three arguments: the first argument specifies a socket that was created with socket, has been bound to an address with bind and has issued a successful call to listen. The second argument is either a null pointer or a pointer to a sockaddr structure where the address of the connecting socket shall be returned. The third argument is either a null pointer, if the second argument is a null pointer or a pointer to a socklen_t object which on input specifies the length of the supplied sockaddr structure and on output specifies the length of the stored address. If the second argument is not a null pointer, the address of the peer for the accepted connection shall be stored in the sockaddr structure pointed to by this argument and the length of this address shall be stored in the object pointed to by the third argument. If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address shall be truncated. If the protocol permits connections by unbound clients and the peer is not bound, then the value stored in the object pointed to by the second argument is unspecified. If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file descriptor for the socket, accept shall block until a connection is present. If the listen queue is empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket, accept shall fail and set errno to EAGAIN or EWOULDBLOCK. The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections. Upon successful completion, accept shall return the non-negative file descriptor of the accepted socket. Otherwise, -1 shall be returned, errno shall be set to indicate the error, and any object pointed to by the third argument shall remain unchanged.

12.1.6 The connect System Call.

connect is used by the client process to establish a connection with a server. The function shall attempt to make a connection on a connection-mode socket or to set or reset the peer address of a connectionless-mode socket. The function takes the following arguments: the first argument specifies the file descriptor associated with the socket. The second argument specifies a pointer to a sockaddr structure containing the peer address. The length and format of the address depend on the address family of the socket. The third argument specifies the length of the sockaddr structure pointed to by the second argument. If the socket has not already been bound to a local address, connect shall bind it to an address which, unless the socket's address family is AF_UNIX, is an unused local address. If the initiating socket is not connection-mode, then connect shall set the socket's peer address, and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all datagrams are sent on subsequent send functions and limits the remote sender for subsequent recv functions. If the sa_family member of the structure pointer by the second argument is AF_UNSPEC, the socket's peer address shall be reset. Note that despite no connection being made, the term "connected" is used to describe a connectionless-mode socket for which a peer address has been set. If the initiating socket is connection-mode, then connect shall attempt to establish a connection to the address specified by the address argument. If the connection cannot be established immediately and O_NONBLOCK is not set for the file descriptor for the socket, connect shall block for up to an unspecified timeout interval until the connection is established. If the timeout interval expires before the connection is established, connect shall fail and the connection attempt shall be aborted. If connect is interrupted by a signal that is caught while blocked waiting to establish a connection, connect shall fail and set errno to EINTR, but the connection request shall not be aborted, and the connection shall be established asynchronously. If the connection cannot be established immediately and O_NONBLOCK is set for the file descriptor for the socket, connect shall fail and set errno to EINPROGRESS, but the connection request shall not be aborted, and the connection shall be established asynchronously. Subsequent calls to connect for the same socket, before the connection is established, shall fail and set errno to EALREADY. When the connection has been established asynchronously, pselect, select and poll shall indicate that the file descriptor for the socket is ready for writing. The socket in use may require the process to have appropriate privileges to use the connect function. Upon successful completion, connect shall return 0; otherwise, -1 shall be returned and errno set to indicate the error.

12.1.7 Connectionless Sockets.

Sockets that use the SOCK_DGRAM method of communication do not need to be connected in order to be used. This is because modified versions of recv, sendto and recvfrom are used to send and receive datagrams.

12.1.8 The sendto System Call.

The sendto function shall send a message through a connection-mode or connectionless-mode socket. If the socket is a connectionless-mode socket, the message shall be sent to the address specified by the fifth argument if no pre-specified peer address has been set. If a peer address has been pre-specified, either the message shall be sent to the address specified by the fifth argument, overriding the pre-specified peer address, or the function shall return -1 and set errno to EISCONN. If the socket is connection-mode, fifth argument be ignored. The sendto function takes the following arguments: the first argument specifies the socket file descriptor. The second argument is a pointer to a buffer containing the message to be sent. The third argument specifies the size of the message in bytes. The fourth argument specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

MSG_EOR terminates a record, if supported by the protocol;

MSG_OOB sends out-of-band data on sockets that support out-of-band data. The significance and semantics of out-of-band data are protocol-specific;

MSG_NOSIGNAL requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The EPIPE error shall still be returned.

the fifth argument points to a sockaddr structure containing the destination address. The length and format of the address depend on the address family of the socket. The sixth argument specifies the length of the sockaddr structure pointed to by the fifth argument. If the socket protocol supports broadcast and the specified address is a broadcast address for the socket protocol, sendto shall fail if the SO_BROADCAST option is not set for the socket. The fifth argument specifies the address of the target. The third argument specifies the length of the message. Successful completion of a call to sendto does not guarantee delivery of the message. A return value of -1 indicates only locally-detected errors. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, sendto shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have O_NONBLOCK set, sendto shall fail. The socket in use may require the process to have appropriate privileges to use the sendto function. Upon successful completion, sendto shall return the number of bytes sent. Otherwise, -1 shall be returned and errno set to indicate the error.

12.1.9 The recvfrom System Call.

The recvfrom function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data. The recvfrom function takes the following arguments: the first arument specifies the socket file descriptor. The second argument is a pointer which points to the buffer where the message should be stored. The third argument specifies the length in bytes of the buffer pointed to by the buffer argument. The fourth argument specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK peeks at an incoming message. The data is treated as unread and the next recvfrom or similar function shall still return this data;

MSG_OOB requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific;

MSG_WAITALL on SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

The fifth argument is a pointer to a sockaddr structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket. The sixth argument is either a null pointer, if address is a null pointer or a pointer to a socklen_t object which on input specifies the length of the supplied sockaddr structure, and on output specifies the length of the stored address. The recvfrom function shall return the length of the message written to the buffer pointed to by the buffer argument. For message-based sockets, such as SOCK_RAW, SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffer and MSG_PEEK is not set in the fourth argument, the excess bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded. If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message. Not all protocols provide the source address for messages. If the fifth argument is not a null pointer and the protocol provides the source address of messages, the source address of the received message shall be stored in the sockaddr structure pointed to by the fifth argument and the length of this address shall be stored in the object pointed to by the sixth argument. If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address shall be truncated. If the fifth argument is not a null pointer and the protocol does not provide the source address of messages, the value stored in the object pointed to by the fifth argument is unspecified. If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recvfrom shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, recvfrom shall fail and set errno to EAGAIN or EWOULDBLOCK. Upon successful completion, recvfrom shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recvfrom shall return 0. Otherwise, the function shall return -1 and set errno to indicate the error.

12.1.10 A Small Client Program.

In listing 12.1 there's a program demonstrating a client connection to a server running on the localhost:

Listing 12.1: client - client program to demonstrate sockets.

```
/* -*- mode: c-mode; -*- */

/* File client.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stddef.h>
#include <inttypes.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* client program. */
#define SERVER_PORT 10240
#define FOREVER for(;;)

/* Functions prototypes. */
long int client(struct sockaddr_in *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int res;
  long int ret;
  struct sockaddr_in servaddr;

  /* */
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(SERVER_PORT);
  res = inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
  ret = client(&servaddr);
  exit(ret);
}
```

```

/*
 * client -- the client function.
 */
long int client(struct sockaddr_in *sa)
{
  int sockfd;
  long int ret = EXIT_FAILURE;
  char *buff[ BUFSIZ ];

  /* */
  if(sa) {
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) >= 0) {
      printf("Created␣socket:␣%d\n", sockfd);
      if(connect(sockfd, (struct sockaddr *) sa, sizeof(struct
          sockaddr_in)) >= 0) {
        printf("Connected␣to␣0x%0.8x,␣port␣0x%0.4x\n", (u_int32_t
            ) sa -> sin_addr.s_addr, ntohs(sa -> sin_port));
        if(recv(sockfd, (void *) buff, BUFSIZ, MSG_WAITALL) >= 0)
            {
          printf("Received␣data␣from␣server:␣%s\n", buff);
          ret = EXIT_SUCCESS;
        } else
          perror("recv");
      } else
        perror("connect");
      close(sockfd);
    } else
      perror("socket");
  } else
    fprintf(stderr, "NULL␣address␣passed.\n");
  return ret;
}

/* End of client.c file. */
```

12.1.11 A Small Server Program.

The server program in listing 12.2 works with the previous example, the client program:

Listing 12.2: server - server program to demonstrate sockets.

```
/* -*- mode: c-mode; -*- */

/* File server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stddef.h>
#include <inttypes.h>
#include <unistd.h>
#include <time.h>
```

```c
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* server program. */
#define SERVER_PORT 10240
#define FOREVER for(;;)

/* Functions prototypes. */
long int server(struct sockaddr_in *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret;
  struct sockaddr_in servaddr;

  /* clear the address structures in memory. */
  bzero(&servaddr, sizeof(struct sockaddr_in));

  /* setup structures. */
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERVER_PORT);
  ret = server(&servaddr);
  exit(ret);
}

/*
 * client -- the client function.
 */
long int server(struct sockaddr_in *sa)
{
  char *buff;
  int listenfd, connfd;
  long int ret = EXIT_FAILURE;
  struct timeval now;
  struct sockaddr_in cliaddr;
  socklen_t cliaddrlen = sizeof(struct sockaddr_in);
  pid_t pid;

  /* */
  if(sa) {
    bzero(&cliaddr, sizeof(struct sockaddr_in));
    if((listenfd = socket(AF_INET, SOCK_STREAM, 0)) >= 0) {
      if(bind(listenfd, (struct sockaddr *) sa, sizeof(struct
          sockaddr_in)) >= 0) {
        printf("Waiting␣to␣accept␣a␣connection...\n");
        if(listen(listenfd, 0) >= 0) {
          FOREVER {
            cliaddrlen = sizeof(cliaddr);
            if((connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &cliaddrlen)) >= 0) {
              printf("Accepted connection from 0x%0.8x, port 0x%0.4x\n", cliaddr.sin_addr, ntohs(cliaddr.sin_port));
              if((pid = fork()) == 0) {
                close(listenfd);
                if(gettimeofday(&now, NULL) >= 0) {
                  buff = ctime(&now.tv_sec);
                  if(buff) {
                    if(send(connfd, (void *) buff, strnlen(buff, BUFSIZ), 0) >= 0) {
                      ret = EXIT_SUCCESS;
                      break;
                    } else {
                      perror("send");
                      break;
                    }
                  } else {
                    fprintf(stderr, "empty time string");
                    break;
                  }
                } else {
                  perror("gettimeofday");
                  break;
                }
              }
              close(connfd);
            } else {
              perror("accept");
              break;
            }
          }
        } else
          perror("listen");
      } else
        perror("bind");
    } else
      perror("socket");
  } else
    fprintf(stderr, "NULL address passed.\n");
  return ret;
}

/* End of server.c file. */
```

12.2 Message Queues.

Message queues are a cross between a *virtual circuit* and datagrams. Distinct message "packets" are exchanged between processes using a queue mechanism so that data arrives in order, but the messages can be received in more or less any order determined by the receiving process(es). A message queue is defined by a unique identifier called a *queue id*, which is usually a long integer. The queue itself is described by the following structure contained in <sys/msg.h>, <sys/types.h> must be included before too:

Listing 12.3: The msqid_ds structure.

```
struct msqid_ds {
  struct ipc_perm msg_perm;
  struct msg *msg_first;
  struct msg *msg_last;
  unsigned long msg_cbytes;
  unsigned long msg_qnum;
  unsigned long msg_qbytes;
  pid_t msg_lspid;
  pid_t msg_lrpid;
  time_t msg_stime;
  long msg_pad1;
  time_t msg_rtime;
  long msg_pad2;
  time_t msg_ctime;
  long msg_pad3;
  long msg_pad4[ 4 ];
};
```

The meanings of the structure members are:

msg_perm	msg queue permission bits;
msg_first	first message in the queue;
msg_last	last message in the queue;
msg_cbytes	number of bytes in use on the queue;
msg_qnum	number of msgs in the queue;
msg_qbytes	maximum number of bytes on the queue;
msg_lspid	pid of last msgsnd;
msg_lrpid	pid of last msgrcv;
msg_stime	time of last msgsnd;
msg_pad1	structure pad member;
msg_rtime	time of last msgrcv;
msg_pad2	structure pad member;
msg_ctime	time of last msgctl;
msg_pad3	structure pad member;

msg_pad4 structure pad member.

The ipc_perm structure defines the permissions on the message queue. It is defined in the include file <sys/ipc.h>:

Listing 12.4: The ipc_perm structure.

```
struct ipc_perm {
  uid_t cuid;
  gid_t cgid;
  uid_t uid;
  gid_t gid;
  mode_t mode;
  unsigned short seq;
  key_t key;
};
```

The members are defined as:

cuid creator user id;

cgid creator group id;

uid user id;

gid group id;

mode r/w permission this is a bit mask:

- IPC_R — read permission;
- IPC_W — write/alter permission;
- IPC_M — permission to change control info.

seq sequence number, to generate unique msg/sem/shmid;

key user specified msg/sem/shm key.

12.2.1 The msgget System Call.

The msgget function operates on *XSI message queues*[2]. The msgget function shall return the message queue identifier associated with the argument key. A message queue identifier, associated message queue and data structure[3], shall be created for the first argument if one of the following is true:

- the first agument is equal to IPC_PRIVATE;
- the first argument does not already have a message queue identifier associated with it and (msgflg & IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new message queue identifier shall be initialized as follows:

- msg_perm.cuid, msg_perm.uid, msg_perm.cgid and msg_perm.gid shall be set to the effective user id and effective group id, respectively, of the calling process;
- the low-order 9 bits of msg_perm.mode shall be set to the low-order 9 bits of msgflg;

[2] See the Base Definitions volume of POSIX.1-2017, Section 3.226, Message Queue.

[3] See <sys/msg.h>.

- msg_qnum, msg_lspid, msg_lrpid, msg_stime and msg_rtime shall be set to 0;
- msg_ctime shall be set to the current time;
- msg_qbytes shall be set to the system limit.

Upon successful completion, msgget shall return a non-negative integer, namely a message queue identifier. Otherwise, it shall return -1 and set errno to indicate the error.

12.2.2 The msgctl System Call.

The msgctl function operates on XSI message queues. This function takes three arguments and shall provide message control operations as specified by the second argument. The following values for the second argument and the message control operations they specify, are:

IPC_STAT place the current value of each member of the msqid_ds data structure associated with msqid into the structure pointed to by the third argument. The contents of this structure are defined in <sys/msg.h>;

IPC_SET set the value of the following members of the msqid_ds data structure associated with the first parameter to the corresponding value found in the structure pointed to by the third argument:

msg_perm.uid

msg_perm.gid

msg_perm.mode

msg_qbytes

Also, the msg_ctime timestamp shall be set to the current time. IPC_SET can only be executed by a process with appropriate privileges or that has an effective user id equal to the value of msg_perm.cuid or msg_perm.uid in the msqid_ds data structure associated with the first parameter. Only a process with appropriate privileges can raise the value of msg_qbytes;

IPC_RMID remove the message queue identifier specified by the first argument from the system and destroy the message queue and msqid_ds data structure associated with it. IPC_RMD can only be executed by a process with appropriate privileges or one that has an effective user id equal to the value of msg_perm.cuid or msg_perm.uid in the msqid_ds data structure associated with the value in the first argument.

Upon successful completion, msgctl shall return 0; otherwise, it shall return -1 and set errno to indicate the error.

12.2.3 The msgsnd and msgrcv System Calls.

The msgsnd function operates on XSI message queues. The function take four arguments and shall send a message to the queue associated with the message queue identifier specified by the first argument value. The application shall ensure that the second argument points to a user-defined buffer that contains first a field of type long specifying the type of the message and then a data portion that holds the data bytes of the message. The structure below is an example of what this user- defined buffer might look like:

Listing 12.5: Custom mymsg structure.

```
struct mymsg {
  long mtype;
  char mtext[ 1 ];
}
```

The structure member mtype is a non-zero positive type long that can be used by the receiving process for message selection. The structure member mtext is any text of length which is the third argument value in bytes. This argument can range from 0 to a system-mposed maximum. The fourth and last argument specifies the action to be taken if one or more of the following is true:

- the number of bytes already on the queue is equal to msg_qbytes[4];
- the total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- if (msgflg & IPC_NOWAIT) is non-zero, the message shall not be sent and the calling thread shall return immediately;
- if (msgflg & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - the condition responsible for the suspension no longer exists, in which case the message is sent;
  - the message queue identifier msqid is removed from the system; when this occurs, errno shall be set to EIDRM and -1 shall be returned;
  - the calling thread receives a signal that is to be caught; in this case the message is not sent and the calling thread resumes execution in the manner prescribed in sigaction;

Upon successful completion, the following actions are taken with respect to the data structure associated with msqid[5];

- msg_qnum shall be incremented by 1.
- msg_lspid shall be set to the process id of the calling process.
- msg_stime shall be set to the current time.

Upon successful completion, msgsnd shall return 0; otherwise, no message shall be sent, msgsnd shall return -1 and errno shall be set to indicate the error. The msgrcv function operates on XSI message queues as the msgsnd function and it takes five arguments. The function shall read a message from the queue associated with the message queue identifier specified by the first argument and place it in the user-defined buffer pointed to by the second argument pointer. The application shall ensure that the second argument points to a user-defined buffer that contains first a field of type long, specifying the type of the message and then a data portion that holds the data bytes of the message. The user defined structure is the same of the msgsnd function. The structure member mtype is the received message's type as specified by the sending process. The structure member mtext is the text of the message. The third argument specifies the size in bytes of the member mtext. The received message shall be truncated to the third argument value in bytes if it is larger and (msgflg & MSG_NOERROR) is non-zero. The truncated part of the message shall be lost and no indication of the truncation shall be given to the calling process. If the value of the thord argument is greater than SSIZE_MAX, the result is implementation-defined. The fourth argument specifies the type of message requested as follows:

- if is 0, the first message on the queue shall be received;
- if is greater than 0, the first message of type msgtyp shall be received;

[4] See <sys/msg.h>.

[5] See <sys/msg.h>.

- if is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the fourth argument shall be received.

The fifth argument specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- if (msgflg & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a return value of -1 and errno set to ENOMSG.
- if (msgflg & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - a message of the desired type is placed on the queue;
  - the message queue identifier msqid is removed from the system; when this occurs, errno shall be set to EIDRM and -1 shall be returned;
  - the calling thread receives a signal that is to be caught; in this case a message is not received and the calling thread resumes execution in the manner prescribed in sigaction.

Upon successful completion, the following actions are taken with respect to the data structure associated with msqid:

- msg_qnum shall be decremented by 1;
- msg_lrpid shall be set to the process id of the calling process;
- msg_rtime shall be set to the current time.

Upon successful completion, msgrcv shall return a value equal to the number of bytes actually placed into the buffer mtext. Otherwise, no message shall be received, msgrcv shall return -1, and errno shall be set to indicate the error. Listing 12.6 shows a server program that creates a message queue and then waits for a message to be sent to it. After it receives the message, the program will respond with a message of its own:

Listing 12.6: mq-server - server program to demonstrate message queues.

```
/* -*- mode: c-mode; -*- */

/* File mqserver.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* mqserver program. */
#define MSGSZ 128
#define FOREVER for(;;)

/* Declare the message structure. */
struct tagMessage {
  long mtype;
  char mtext[ MSGSZ ];
};

```

```
typedef struct tagMessage message_t;

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  int msqid;
  key_t key;
  message_t sbuf, rbuf;

  /* Create a message queue with "name" 1234. */
  key = 1234;

  /*
   * We want to let everyone read and
   * write on this message queue, hence
   * we use 0666 as the permissions.
   */
  if((msqid = msgget(key, IPC_CREAT | 0666)) >= 0) {
    printf("Wait␣for␣a␣client␣message.\n");

    /* Receive a message. */
    if(msgrcv(msqid, &rbuf, MSGSZ, 0, 0) >= 0) {

      /* Print the client message. */
      printf("client␣message:␣%s\n", rbuf.mtext);

      /* We send a message of type 2. */
      sbuf.mtype = 2;
      snprintf(sbuf.mtext, MSGSZ, "I␣received␣your␣message.");

      /* Send an answer. */
      if(msgsnd(msqid, &sbuf, strnlen(sbuf.mtext, MSGSZ) + 1, 0)
          >= 0) {
        if(msgctl(msqid, IPC_RMID, NULL) >= 0)
          ret = EXIT_SUCCESS;
        else
          perror("msgctl");
      } else
        perror("msgsnd");
    } else
      perror("msgrcv");
  } else
    perror("msgget");

  /* Exit. */
  exit(ret);
}

```

```
/* End of mqserver.c file. */
```

Listing 12.7 show a client process that sends a message to the server and then waits for a response and prints it on the screen. Before running the program, start up the server process in the background:

Listing 12.7: mq-client - client program to demonstrate message queues.

```
/* -*- mode: c-mode; -*- */

/* File mqclient.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* mqclient program. */
#define MSGSZ 128
#define FOREVER for(;;)

/* Declare the message structure. */
struct tagMessage {
  long mtype;
  char mtext[ MSGSZ ];
};

typedef struct tagMessage message_t;

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  int msqid;
  key_t key;
  message_t sbuf, rbuf;

  /* Create a message queue with "name" 1234. */
  key = 1234;

  /*
   * Get the message queue id for the
   * "name" 1234, which was created by
   * the server.
   */
  if((msqid = msgget(key, 0666)) >= 0) {

    /*
     * We'll send message type 1, the server
     * will send message type 2.
     */
    sbuf.mtype = 1;
    snprintf(sbuf.mtext, MSGSZ, "Did␣you␣get␣this?");

    /* Send message. */
    if(msgsnd(msqid, &sbuf, strnlen(sbuf.mtext, MSGSZ) + 1, 0) >=
        0) {

      /* Receive an answer of message type 2. */
      if(msgrcv(msqid, &rbuf, strnlen(rbuf.mtext, MSGSZ) + 1, 2,
         0) >= 0) {

        /* Print the answer. */
        printf("server␣message:␣%s\n", rbuf.mtext);
        ret = EXIT_SUCCESS;
      } else
        perror("msgrcv");
    } else
      perror("msgsnd");
  } else
      perror("msgget");

  /* Exit. */
  exit(ret);
}

/* End of mqclient.c file. */
```

12.3 Semaphores.

Semaphores are special types of flags used for signalling between two processes. They are tipically used to guard “critical sections” of code that modify shared data structures. In general, a section of code is written so that it cannot begin until a given semaphore is equal to a specific value. For example a program might wait until the semaphore is equal to zero. Then it would set the semaphore to one and perform some actions with a shared data structure and then reset the semaphore to zero. Other processes, also waiting until the semaphore is equal to zero, are effectively “locked out” from modifying the data structure while it is in use. When the semaphore becomes equal to zero again, the system will allow one of the waiting process to proceed. Semaphores are allocated in sets; each set is defined by unique *semaphore id*. The semaphores in a semaphore set are numbered consecutively starting from zero. The sets themselves are described in a structure of type `semid_ds`, declared in the include file `<sys/sem.h>`, `<sys/types.h>` must also be included:

Listing 12.8: The semid_ds structure.

```
struct semid_ds {
  struct ipc_perm sem_perm;
  struct sem *sem_base;
  unsigned short sem_nsems;
  time_t sem_otime;
  long sem_pad1;
  time_t sem_ctime;
  long sem_pad2;
  long sem_pad3[ 4 ];
};
```

The members are defined as:

sem_perm operation permission struct;

sem_base pointer to first semaphore in set;

sem_nsems number of sems in set;

sem_otime last operation time;

sem_pad1 SVABI/386 says I need this here (LOLx1);

sem_ctime last change time. Times measured in secs since 00:00:00 GMT, Jan. 1, 1970;

sem_pad2 SVABI/386 says I need this here (LOLx2);

sem_pad3 SVABI/386 says I need this here (LOLx3).

12.3.1 The semget System Call.

The semget system call takes three arguments and returns the semaphore identifier associated with the first argument which is the key. A new set containing a number of semaphores as per the second argument is created if either the first argument is equal to IPC_PRIVATE or the second argument does not have a semaphore set associated with it and the IPC_CREAT bit is set in the third argument. The access modes of the created semaphores is specified in the third argument as a bitwise OR of zero or more of the following values:

SEM_A alter permission for owner SEM_R read permission for owner;

(SEM_A>>3) alter permission for group (SEM_R >> 3) read permission for group;

(SEM_A>>6) alter permission for other (SEM_R >> 6) read permission for other;

If a new set of semaphores is created, the data structure associated with it, the semid_ds structure, is initialized as follows:

- sem_perm.cuid and sem_perm.uid are set to the effective UID of the calling process;
- sem_perm.gid and sem_perm.cgid are set to the effective GID of the calling process;
- sem_perm.mode is set to the lower 9 bits of the third argument;
- sem_nsems is set to the value of the second argument;
- sem_ctime is set to the current time;
- sem_otime is set to 0.

semget returns a non-negative semaphore identifier if successful. Otherwise, -1 is returned and errno is set to reflect the error.

12.3.2 The semctl System Call.

The semctl system call takes four arguments and provides a number of control operations on the semaphore specified by the fourth argument and the first one. The operation to be performed is specified in the third argument. The fourth argument is a union of the following fields:

```
int val; /* value for SETVAL */
struct semid_ds *buf; /* buffer for IPC_{STAT,SET} */
u_short *array; /* array for GETALL & SETALL */
```

The semid_ds structure used in the IPC_SET and IPC_STAT commands is defined as follows in <sys/sem.h>. See 12.8. The ipc_perm structure used inside the semid_ds structure is defined in <sys/ipc.h>. See 12.4. semctl provides the following operations:

GETVAL return the value of the semaphore;

SETVAL set the value of the semaphore to arg.val;

GETPID return the pid of the last process that did an operation on this semaphore;

GETNCNT return the number of processes waiting to acquire the semaphore;

GETZCNT return the number of processes waiting for the value of the semaphore to reach 0;

GETALL return the values for all the semaphores associated with semid;

SETALL set the values for all the semaphores that are associated with the semaphore identifier semid to the corresponding values in arg.array;

IPC_STAT gather statistics about a semaphore and place the information in the semid_ds structure pointed to by arg.buf;

IPC_SET set the value of the sem_perm.uid, sem_perm.gid and sem_perm.mode fields in the structure associated with the semaphore. The values are taken from the corresponding fields in the structure pointed to by arg.buf. This operation can only be executed by the super-user or a process that has an effective user id equal to either sem_perm.cuid or sem_perm.uid in the data structure associated with the message queue;

IPC_RMID remove the semaphores associated with semid from the system and destroy the data structures associated with it. Only the super-user or a process with an effective UID equal to the sem_perm.cuid or sem_perm.uid values in the data structure associated with the semaphore can do this.

The permission to read or change a message queue[6] is determined by the sem_perm.mode field in the same way as is done with files[7], but the effective UID can match either the sem_perm.cuid field or the sem_perm.uid field and the effective GID can match either sem_perm.cgid or sem_perm.gid. For the GETVAL, GETPID, GETNCNT and GETZCNT operations, semctl returns one of the values described above if successful. All other operations will make semctl return 0 if no errors occur. Otherwise -1 is returned and errno set to reflect the error.

[6] See *semop*(2).

[7] See *chmod*(2).

12.3.3 The semop System Call.

semop provides a number of atomic operations on a set of semaphores. It takes three arguments. The semaphore set is specified by its first argument. The second argument is an array of semaphore operations and the third is the number of operations in this array. The sembuf structures in the array contain the following members:

```
u_short sem_num; /* semaphore # */
short sem_op; /* semaphore operation */
short sem_flg; /* operation flags */
```

Each operation, specified in sem_op, is applied to semaphore number sem_num in the set of semaphores specified by the first function argument. The value of sem_op determines the action taken in the following way:

- sem_op is less than 0. The current process is blocked until the value of the semaphore is greater than or equal to the absolute value of sem_op. The absolute value of sem_op is then subtracted from the value of the semaphore and the calling process continues. Negative values of sem_op are thus used to enter critical regions;

- sem_op is greater than 0. Its value is added to the value of the specified semaphore. This is used to leave critical regions;

- sem_op is equal to 0. The calling process is blocked until the value of the specified semaphore reaches 0.

The behavior of each operation is influenced by the flags set in sem_flg in the following way:

IPC_NOWAIT in the case where the calling process would normally block, waiting for a semaphore to reach a certain value, IPC_NOWAIT makes the call return immediately, returning a value of -1 and setting errno to EAGAIN;

SEM_UNDO keep track of the changes that this call makes to the value of a semaphore, so that they can be undone when the calling process terminates. This is useful to prevent other processes waiting on a semaphore to block forever, should the process that has the semaphore locked terminate in a critical section.

Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. On OpenBSD there is also an implementation of POSIX compliant semaphores which we will not describe here. Listing 12.9 shows a program to create a group of semaphores.

Listing 12.9: semcreate - creates a semaphore group.

```
/* -*- mode: c-mode; -*- */

/* File semcreate.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/sem.h>

/* semcreate program. */
#define FOREVER for(;;)

```

```
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int c, i, oflag, semid, nsems;
  long int ret = EXIT_FAILURE;
  key_t key;

  /* */
  oflag = IPC_CREAT | 0666;
  while((c = getopt(argc, argv, "e")) != -1) {
    switch(c) {
    case 'e':
      oflag |= IPC_EXCL;
      break;

    default:
      ;
      break;
    }
  }
  if(optind == (argc - 2)) {
    nsems = atoi(argv[ optind + 1 ]);
    printf("Creating␣%d␣semaphore%s", nsems, nsems > 1 ? "s.\n" : ".\n");
    if((key = ftok(argv[ optind ], 0)) >= 0) {
      printf("creating␣key␣from␣path␣%s:␣%d\n", argv[ optind ], key);
      if((semid = semget(key, nsems, oflag)) >= 0)
        ret = EXIT_SUCCESS;
      else
        perror("semget");
    } else
      perror("ftok");
  } else
    fprintf(stderr, "usage:␣semcreate␣[␣-e␣]␣<pathname>␣<nsems>\n");
  exit(ret);
}

/* End of semcreate.c file. */
```

12.4 Shared Memory.

Shared memory provides a method for two or more programs to share a segment of virtual memory and use it as if it were actually part of each program. This is useful, possibly in conjunction with semaphores, for having multiple processes update the same data structures. A shared memory segment is described by a unique identifier called a shared memory id. The shared memory segment itself is described by a structure of type shmid_ds, declared in the include file <sys/shm.h>,

<sys/types.h> must also be included before:

Listing 12.10: The shmid_ds structure.

```
struct shmid_ds {
  struct ipc_perm shm_perm;
  int shm_segsz;
  pid_t shm_lpid;
  pid_t shm_cpid;
  short shm_nattch;
  time_t shm_atime;
  time_t shm_dtime;
  time_t shm_ctime;
  void *shm_internal;
};
```

The members are defined as:

shm_perm	operation permissions;
shm_segsz	size of segment in bytes;
shm_lpid	pid of last shm op;
shm_cpid	pid of creator;
shm_nattch	number of current attaches;
shm_atime	last shmat time;
shm_dtime	last shmdt time;
shm_ctime	last change by shmctl;
shm_internal	sysv stupidity.

12.4.1 The shmget System Call.

The shmget function operates on XSI shared memory, it shall return the shared memory identifier associated with key. It takes three argument:

- first argument is a key;
- second argument is the size of the shared memory segment in bytes;
- third argument is a mask of bits which are flags.

A *shared memory identifier*, associated data structure, and shared memory segment of at least size bytes, see <sys/shm.h>, are created for key if one of the following is true:

- the first argument is equal to IPC_PRIVATE;
- the first argument does not already have a shared memory identifier associated with it and the third argument anded with IPC_CREAT is non-zero.

Upon creation, the data structure associated with the new shared memory identifier shall be initialized as follows:

- the values of shm_perm.cuid, shm_perm.uid, shm_perm.cgid and shm_perm.gid are set to the effective user id and effective group id, respectively, of the calling process;

- the low-order nine bits of shm_perm.mode are set to the low-order nine bits of third argument;
- the value of the second argument is set to the value of size;
- the values of shm_lpid, shm_nattch, shm_atime and shm_dtime are set to 0;
- the value of shm_ctime is set to the current time.

When the shared memory segment is created, it shall be initialized with all zero values. Upon successful completion, shmget shall return a non-negative integer, namely a shared memory identifier; otherwise, it shall return -1 and set errno to indicate the error.

12.4.2 The shmctl System Call.

shmctl system call takes three arguments and performs some control operations on the shared memory area specified by the first argument. Each shared memory segment has a data structure associated with it, parts of which may be altered by shmctl and parts of which determine the actions of shmctl. This structure is defined in <sys/shm.h> and it is reported in 12.10. The ipc_perm structure used inside the shmid_ds structure is defined in <sys/ipc.h> which is first reported in 12.4. The operation to be performed by shmctl is specified in its second argument and is one of:

IPC_STAT gather information about the shared memory segment and place it in the structure pointed to by the third argument;

IPC_SET set the value of the shm_perm.uid, shm_perm.gid and shm_perm.mode fields in the structure associated with the first argument. The values are taken from the corresponding fields in the structure pointed to by the third argument. This operation can only be executed by the super-user or a process that has an effective user id equal to either shm_perm.cuid or shm_perm.uid in the data structure associated with the shared memory segment;

IPC_RMID mark the shared memory segment specified by first argument for removal when it is no longer in use by any process. When it is removed, all data associated with it will be destroyed too. Only the superuser or a process with an effective UID equal to the shm_perm.cuid or shm_perm.uid values in the data structure associated with the queue can do this.

The read and write permissions on a shared memory identifier are determined by the shm_perm.mode field in the same way as is done with files[8], but the effective UID can match either the shm_perm.cuid field or the shm_perm.uid field and the effective GID can match either shm_perm.cgid or shm_perm.gid. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

12.4.3 The shmat and shmdt System Calls.

shmat takes three arguments and it maps the shared memory segment associated with the shared memory identifier in the first argument into the address space of the calling process. The address at which the segment is mapped is determined by the second argument. If it is equal to NULL, the system will pick an address itself. Otherwise, an attempt is made to map the shared memory segment at the address specified in the second argument. If SHM_RND is set in the third argument, the system will round the address down to a multiple of SHMLBA bytes[9]. A shared memory segment can be mapped read-only by specifying the SHM_RDONLY flag in the third argument. shmdt takes one

[8] See *chmod*(2).

[9] SHMLBA is defined in <sys/shm.h>.

parameter and unmaps the shared memory segment that is currently mapped at the first argument from the calling process' address space. This argument must be a value returned by a prior shmat call. A shared memory segment will remain existent until it is removed by a call to *shmctl*(2) with the IPC_RMID command. shmat returns the address at which the shared memory segment has been mapped into the calling process' address space when successful, shmdt returns 0 on successful completion. Otherwise, a value of -1 is returned, and the global variable errno is set to indicate the error. Listing 12.11 shows a small server program that obtains a shared memory segment and puts some data into it for a client process to read. It then waits until the first element of the segment is changed by the client, indicating that the segment has bee read.

Listing 12.11: shm-server - server program to demonstrate shared memory.

```c
/* -*- mode: c-mode; -*- */

/* File shm-server.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

/* shm-server program. */
#define FOREVER for(;;)
#define SHMSZ 32

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  char c;
  char *shm, *s;
  int shmid;
  long int ret = EXIT_FAILURE;
  key_t key;

  /*
   * We'll name our shared memory segment
   * "5678".
   */
  key = 5678;

  /* Create the segment. */
  if((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) >= 0) {

    /* Now we attach the segment to our data space. */
    if((shm = shmat(shmid, NULL, 0)) >= 0) {

      /*
       * Now put some things into the memory for the
       * process to read.
       */
      s = shm;
      for(c = 'a'; c <= 'z'; c++)
        *s++ = c;
      *s = '\0';
      printf("Data␣at␣0x%0.8x:␣%s\n", (size_t) shm, (char *) shm);

      /*
       * Finally, we wait until the other process
       * changes the first character of our memory
       * to '*', indicating that it has read what
       * we put there.
       */
      printf("Waiting␣for␣client␣to␣change␣the␣shared␣memory.\n");
      while(*shm != '*')
        sleep(1);
      printf("Client␣succesfully␣modified␣shared␣data␣segment:␣%s\n", shm);
      ret = EXIT_SUCCESS;
    } else
      perror("shmat");
  } else
    perror("shmget");
  exit(ret);
}

/* End of shm-server.c file. */
```

Listing 12.12 shows the client program that reads the shared memory segment, prints it on the screen and then changes the first element of the segment so that the server can exit. Before running this program, the server process must be started in the background.

Listing 12.12: shm-client - client program to demonstrate shared memory.

```
/* -*- mode: c-mode; -*- */

/* File shm-client.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

/* shm-client program. */
#define FOREVER for(;;)
#define SHMSZ 32

/* Functions prototypes. */
int main(int, char *[]);
```

```

/* Main function. */
int main(int argc, char *argv[])
{
  char c;
  char *shm, *s;
  int shmid;
  long int ret = EXIT_FAILURE;
  key_t key;

  /*
   * We need to get the segment named
   * "5678", created by the server.
   */
  key = 5678;

  /* Locate the segment. */
  if((shmid = shmget(key, SHMSZ, 0666)) >= 0) {

    /* Now we attach the segment to our data space. */
    if((shm = shmat(shmid, NULL, 0)) >= 0) {
      printf("Server␣data␣at␣0x%0.8x:␣", (size_t) shm);

      /* Now we read what the server put in the memory. */
      for(s = shm; *s != '\0'; s++)
        putchar(*s);
      putchar('\n');

      /*
       * Finally, change the first character of the
       * segment to '*', indicating we have read
       * the segment.
       */
      *shm = '*';
      ret = EXIT_SUCCESS;
    } else
      perror("shmat");
  } else
    perror("shmget");
  exit(ret);
}

/* End of shm-client.c file. */
```

Chapter 13

Networking.

OpenBSD provides an extensive facility for interprocess communication between processes running on different machines. This is done using the *Transmission Control Protocol and Internet Protocol*, *TCP/IP*, as specified by the *Defence Advanced Research Project Agency*, *DARPA*, for use on their international network, the ARPANET. The networking facilities is based on the socket mechanism and work in much the same way as the interprocess communication facility discussed in Chapter 11, Important Points.. Rather than using the UNIX domain, however, the networking facilities operate in the Internet domain.

13.1 Addresses.

In the UNIX domain, the address of a program is specified by using a standard UNIX path name. In the *Internet domain*, however, this is not viable for two reasons:

- first, standard path names do not provide any method of specifying which computer a program is located on;
- second, not all the computers connected to a network will necessarily be running OpenBSD or another UNIX-like operating system.

The addresses used in the Internet domain consist of two numbers. The first number is a 32-bit *internetwork number* of the computer which the program to be accessed reside on. Each machine on a network, whether it be the global ARPANET or simply a *local-area network*, has a unique internetwork number. It should be noted here that although a network number functions as the name of a machine, it is not the same thing as the *hostname* of a machine. A hostname is usually a text string, such as "intrepid.ecn.purdue.edu" or "sri-nic.arpa" and is not easily used as a network address because it does not give any information about how to access the machine itself. Because the same host can reside on more than one network, it is possible for a single hostname to be associated with several network numbers. Each network number specifies to the operating system how to reach the machine by using a different network path. The second number making up an Internet domain address is a 16-bit *port number*. Each networking program on a machine uses a separate port number, the port number is somewhat similar to the path name used in the UNIX domain. For example, the ssh program uses port number 22 and the ftp file transfer server uses port number 21[1]. Thus a program wishing to connect to the file transfer server residing on the machine with network number 12345 would specify the Internet address (12345, 21). Without using port numbers, it would be difficult for any machine to run more than one network at a time.

13.2 Translating Hostnames Into Network Numbers.

As mentioned in the previous section, a hostname cannot function as a network address; it must be converted to a network number. The relationships between hostnames and network numbers are stored in the text file /etc/hosts. To translate hostnames into network numbers, the gethostbyname library routine is used. This routine takes a single argument, a character string containing the name of the host to be looked up. It returns a pointer to a structure of type hostent, as defined in the include file <netdb.h>:

Listing 13.1: The hostent structure.

```
struct  hostent {
  char *h_name;
  char **h_aliases;
  int h_addrtype;
  int h_length;
  char **h_addr_list;
};

#define h_addr h_addr_list[ 0 ]
```

The members of this structure are:

h_name official name of the host;

h_aliases a NULL-terminated array of alternate names for the host;

h_addrtype the type of address being returned;

h_length the length, in bytes, of the address;

h_addr_list a NULL-terminated array of network addresses for the host. Host addresses are returned in network byte order;

h_addr the first address in h_addr_list; this is for backward compatibility.

The h_addr_list element of this structure contains all the network numbers associated with the hostname. The h_addr "element" is for backward compatibility, but is still often used in programs that don't really care which network number they use to access a machine. If the hostname cannot be found in the database, the constant NULL is returned. Another library routine gethostbyaddr, exists to look up network numbers and obtain the hostname associated with them. It also returns a pointer to a structure of type hostent; the h_name field of this structure will contain the hostname.

13.2.1 The gethostbyname and gethostbyaddr Library Routines.

The gethostbyname function return a pointer to an object of type struct hostent describing an Internet host referenced by the first argument. gethostbyaddr takes three argument. The first argument is a string containing the Internet address of the host, with length in the second argument and address family in the third argument. The hostent structure contains either information obtained from a name server, broken-out fields from a line in */etc/hosts* or database entries supplied

[1] See */etc/services* for the list of ports and their associated service/program.

by the *yp*(8) system. *resolv.conf*(5) describes how the particular database is chosen. The function gethostbyname will search for the named host in the current domain and its parents using the search lookup semantics detailed in *resolv.conf*(5) and *hostname*(7). The gethostbyaddr function will search for the specified address of length len in the address family af. The only address family supported is AF_INET. Listing 13.2 shows a program retrieving host informations from the hostname databases.

Listing 13.2: hostent - program to demonstrate the usage of host database.

```
/* -*- mode: c-mode; -*- */

/* File hostent.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <netdb.h>

/* hostent program. */
#define FOREVER for(;;)

/* Function prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i;
  char **alias;
  long int ret = EXIT_FAILURE;
  struct hostent *host;

  /* Check the arguments. */
  if(argc == 2) {

    /* Get the specified host from the database. */
    if((host = gethostbyname(argv[ 1 ])) != NULL) {
      printf("official␣host␣name:␣%s\n", host -> h_name);
      printf("alias␣list:␣");
      alias = host -> h_aliases;
      while(*alias)
        printf("%s␣", *alias++);
      printf("\n");
      printf("address␣type:␣%d\n", host -> h_addrtype);
      printf("addresses:␣");
      for(i = 0; i < host -> h_length; i++)
        printf("0x%0.8x␣", host -> h_addr_list[ i ]);
      printf("\n");
      ret = EXIT_SUCCESS;
    } else
      fprintf(stderr, "Host␣%s␣not␣found␣in␣hosts␣database.\n",
          argv[ 1 ]);
  } else
    fprintf(stderr, "Usage␣hostent␣<hostname>\n");
  exit(ret);
}

/* End of hostent.c file. */
```

13.3 Obtaining Port Numbers.

Most network services, file transfer, secure login, etc., programs usually use standard "well-known" port numbers – that is, port numbers which are the same everywhere and are set forth in the specifications of the protocols which use them. This enables a client program on one machines to contact a server program on any other machine without having to guess at what port the server resides[2]. Port numbers for *well-known* services are listed, along with their service names, in the text file */etc/services*. The fields of one line of this file are contained in the servent structure defined in <netdb.h>:

Listing 13.3: The servent structure.

```
struct  servent {
  char *s_name;
  char **s_aliases;
  int s_port;
  char *s_proto;
};
```

The members of this structure are:

s_name the official name of the service;

s_aliases a null-terminated list of alternate names for the service;

s_port the port number at which the service resides. Port numbers are returned in network byte order;

s_proto the name of the protocol to use when contacting the service.

To get the port and service informations we use two library routine: getservbyname and getservbyport.

13.3.1 The getservbyname and getservbyport Library Calls.

The getservbyname and getservbyport functions each return a pointer to an object with the servent structure, described in 13.3, containing the broken-out fields of a line in the network services database, */etc/services*. The getservbyname and getservbyport functions sequentially search from the beginning of the file until a matching protocol name or port number, specified in network byte order, is found, or until EOF is encountered. If a non-null protocol name is also supplied, searches must also match the protocol. The structure must be zero-filled before it is used and should be considered opaque for the sake of portability. The getservbyport and getservbyname functions return a pointer to a servent structure on success or a NULL pointer if end-of-file is reached or an error occurs. getservbyname takes two argument: first argument is a string containing the service name, the second argument the protocol name. getservbyport takes two arguments too: the first argument is the port number, the second argument is the protocol name. Listing 13.4 shows the usage for service database querying.

[2] Sometimes ports are choosen randomly between client and server for security purpose.

Listing 13.4: servent - program to demonstrate the usage of serfvices database.

```
/* -*- mode: c-mode; -*- */

/* File servent.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <netdb.h>

/* servent program. */
#define FOREVER for(;;)

/* Function prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i;
  char **alias, *servicename, *protocolname;
  long int ret = EXIT_FAILURE;
  struct servent *service;

  /* Check the arguments. */
  if(argc < 2)
    fprintf(stderr, "Usage␣servent␣<service␣name>␣<protocol␣name>\n");
  else {
    if (argc == 3) {
      servicename = argv[ 1 ];
      protocolname = argv[ 2 ];
    } else if(argc == 2) {
      servicename = argv[ 1 ];
      protocolname = argv[ 2 ];
    }
    if((service = getservbyname(servicename, protocolname)) != NULL) {
      printf("official␣service␣name:␣%s\n", service -> s_name);
      printf("alias␣list:␣");
      alias = service -> s_aliases;
      while(*alias)
        printf("%s␣", *alias++);
      printf("\n");
      printf("port:␣0x%0.4x\n", htons(service -> s_port));
      printf("protocol:␣%s\n", service -> s_proto);
      ret = EXIT_SUCCESS;
    } else
      fprintf(stderr, "Service␣%s␣with␣protocol␣%s␣not␣found␣in␣services␣database.\n", argv[ 1 ], argv[ 2 ]);
  }
  exit(ret);
}

/* End of hostent.c file. */
```

13.4 Network Byte Order.

Before discussing the system calls used for networking, it is necessary to discuss the *byte order* of numbers used by the networking software. The method in which integers are stored in computers is called *endiannes* and varies from vendor to vendor. Some computers store integers with the most significant bit in the lowest address – and are called *big endian*, while others store them with the most significant bit in the highest address – and they are called *little endian*. Because great chaos would result if two machines using different byte orders were try to communicate directly, the network software requires that all data be exchanged in *network byte order*. In order to convert integers to network byte order, two library routines, `htons` and `htonl`, are provided. These convert short and long integers, respectively, from *host type order* to network byte order. Likewise, two other routines `htohs` and `ntohl`, exist to convert short and long integers from network byte order to host byte order. The `gethostbyname` and `getservbyname` routines return all data in their structures in network byte order.

13.5 Networking System Calls.

The system calls used to perform networking tasks are the same system calls used for interprocess communication, described in Chapter 11, Important Points.. There are a few differences in the parameters passed to these system calls, however:

- the first parameter to `socket` is now given as `AF_INET`, which specifies the Internet domain. The second parameter may still be either `SOCK_STREAM` or `SOCK_DGRAM`;
- the type of `sackaddr` structure used with `accept`, `bind`, `connect`, `sendto` and `recvfrom` is now of type `sockaddr_in` and is declared in the include file `<netinet/in.h>`:

  Listing 13.5: The sockaddr structure.

  ```
  struct sockaddr {
    __uint8_t sa_len;
    sa_family_t sa_family;
    char sa_data[ 14 ];
  };
  ```

  Where:

  `sa_len` total length;

  `sa_family` address family;

  `sa_data` actually longer; address value.

- the `sin_port` element of the structure `sockaddr_in` should contain the port number, in network byte order, to be connect to. The `sin_addr` element should contain the network number, in network byte order, of the machine the port resides on;
- two new system calls, `gethostname` and `sethostname`, can be used to obtain and set the name of the host the program is running on respectively.

The gethostname function returns the standard hostname for the current machine, as previously set by sethostname. The second argument specifies the size of the array pointed by the first argument. If insufficient space is provided, the returned name is truncated. The returned name is always null-terminated. If no space is provided, an error is returned. sethostname sets the name of the host machine to be the first argument, which has length specified in the second argument. This call is restricted to the super-user and is normally used only when the system is bootstrapped. If the call succeeds, a value of 0 is returned. If the call fails, a value of -1 is returned and an error code is placed in the global variable errno. Listing 13.6 and 13.7 show a small server and client program, respectively. These example programs are from the Chapter 11, Important Points. in listings 12.1 and 12.2.

Listing 13.6: inet-client - a client to demonstrate internet domain sockets.

```
/* -*- mode: c-mode; -*- */

/* File inet-client.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stddef.h>
#include <inttypes.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* inet-client program. */
#define SERVER_PORT 10240
#define FOREVER for(;;)

/* Functions prototypes. */
long int client(struct sockaddr_in *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int res;
  long int ret;
  struct sockaddr_in servaddr;

  /* */
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(SERVER_PORT);
  res = inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
  ret = client(&servaddr);
  exit(ret);
}

/*
 * client -- the client function.
 */
long int client(struct sockaddr_in *sa)
{
  int sockfd;
  long int ret = EXIT_FAILURE;
  char *buff[ BUFSIZ ];

  /* */
  if(sa) {
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) >= 0) {
      printf("Created␣socket:␣%d\n", sockfd);
      if(connect(sockfd, (struct sockaddr *) sa, sizeof(struct
          sockaddr_in)) >= 0) {
        printf("Connected␣to␣0x%0.8x,␣port␣0x%0.4x\n", sa ->
            sin_addr,         ntohs(sa -> sin_port));
        if(recv(sockfd, (void *) buff, BUFSIZ, MSG_WAITALL) >= 0)
            {
          printf("Received␣data␣from␣server:␣%s\n", buff);
          ret = EXIT_SUCCESS;
        } else
          perror("recv");
      } else
        perror("connect");
      close(sockfd);
    } else
      perror("socket");
  } else
    fprintf(stderr, "NULL␣address␣passed.\n");
  return ret;
}

/* End of inet-client.c file. */
```

Listing 13.7: inet-server - a server to demonstrate internet domain sockets.

```
/* -*- mode: c-mode; -*- */

/* File inet-server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stddef.h>
#include <inttypes.h>
#include <unistd.h>
#include <time.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

```

```
/* inet-server program. */
#define SERVER_PORT 10240
#define FOREVER for(;;)

/* Functions prototypes. */
long int server(struct sockaddr_in *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret;
  struct sockaddr_in servaddr;

  /* clear the address structures in memory. */
  bzero(&servaddr, sizeof(struct sockaddr_in));

  /* setup structures. */
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERVER_PORT);
  ret = server(&servaddr);
  exit(ret);
}

/*
 * client -- the client function.
 */
long int server(struct sockaddr_in *sa)
{
  char *buff;
  int listenfd, connfd;
  long int ret = EXIT_FAILURE;
  struct timeval now;
  struct sockaddr_in cliaddr;
  socklen_t cliaddrlen = sizeof(struct sockaddr_in);
  pid_t pid;

  /* */
  if(sa) {
    bzero(&cliaddr, sizeof(struct sockaddr_in));
    if((listenfd = socket(AF_INET, SOCK_STREAM, 0)) >= 0) {
      if(bind(listenfd,         \
        (struct sockaddr *) sa,     \
        sizeof(struct sockaddr_in)) >= 0) {
        printf("Waiting␣to␣accept␣a␣connection...\n");
        if(listen(listenfd, 0) >= 0) {
          FOREVER {
            cliaddrlen = sizeof(cliaddr);
            if((connfd = accept(listenfd,         \
              (struct sockaddr *) &cliaddr, \
              &cliaddrlen)) >= 0) {
```

```
              printf("Accepted␣connection␣from␣0x%0.8x,␣port␣0x%0.4x\n", \
                cliaddr.sin_addr,         \
                ntohs(cliaddr.sin_port));
              if((pid = fork()) == 0) {
                close(listenfd);
                if(gettimeofday(&now, NULL) >= 0) {
                  buff = ctime(&now.tv_sec);
                  if(buff) {
                    if(send(connfd,       \
                      (void *) buff,      \
                      strnlen(buff, BUFSIZ), 0) >= 0) {
                      ret = EXIT_SUCCESS;
                      break;
                    } else {
                      perror("send");
                      break;
                    }
                  } else {
                    fprintf(stderr, "empty␣time␣string");
                    break;
                  }
                } else {
                  perror("gettimeofday");
                  break;
                }
              }
              close(connfd);
            } else {
              perror("accept");
              break;
            }
          }
        } else
          perror("listen");
      } else
        perror("bind");
    } else
      perror("socket");
  } else
    fprintf(stderr, "NULL␣address␣passed.\n");
  return ret;
}

/* End of inet-server.c file. */
```

Chapter 14

The File System.

OpenBSD offers the possibility to deal with different *file system* types to ease data exchange with other operating systems. On version 7.5 we can handle:

- ext2, ext3, ext4 linux file systems;
- Microsoft MS-DOS, FAT and NTFS file systems;
- ISO9660 file system;
- NFS file system;
- UDF file system;
- UNIX Fast File System and UNIX Enhanced Fast File System which is the default choice for system disks.

A file system is a way to organize data on a storage media such like disks, tape or a DVD optical disk in a way that it is possible to manipulate easily those data and more important to store them for an undefined amount of time.

14.1 Disk Terminology.

A *disk* is a device than can store data by means of write operations and then the stored data can be retrieved by means of read operations. A disk is usually connected to the computer using electronic interfaces and it is configured and managed by a disk driver software in the operating system. To store and manage data a disk could use different technologies. The most convenient and used kind of disks are:

- mechanical;
- solid state;
- optical.

whatever technology is involved, the disk is composed of two main parts: a media for the physical storage of data and a controlling electronics which operates on the media part to perform certain operations such as write and read. The first media technology involved in the storage of data was the mechanical one which survived until now. A mechanical disk is composed roughly of a number of coaxial rigid disc plates whose surface are made with a magnetic material[1] and are flown by heads. Those disks are spinned by a motor which can reach speeds from 3000 rpm to 10000 rpm depending on the disk type. Modern disk drive unit has got one disk and two heads, one per side. The heads are connected rigidly by a rod moved by an actuator by means of an arm so they can swing spanning on the two disk surfaces and thus assume a precise position. If one *head* reach a position on the surface of the disk to a precise distance from the rotational center, as the disk rotates under the head, it describes a circle which is called a *track*. The tracks on the disk which are identified by the same position of the heads on the respective surfaces and thus are all at the same distance from the center, form a *cylinder*. Since the tracks on the surface of a disk are concentric, so are the corresponding cylinders. Unlike the vinil disks, an hard disk have several tracks per surface that can be accessed just moving the head assembly. A part of a track with a fixed length is called a *sector*. Heads, sectors, tracks and cylinders are referred as the *disk geometry*. Nowadays mechanical disks are still used for data storage but they are often replaced by static mass storage memories, the ssd. Those devices are totally static they are more reliable and faster compared to same size mechanical counterpart.

	Mechanical Disk	**Solid State Disk**	**USB pendrive Disk**
Interface	SATA 6 GB/s	4 port PCIe G4 NVMe 2.0	USB3.1
Maximum Transfer rate in MB/s (R/W)	190	7400/6400	300/200
Capacity in TB	2	2.048	2
Bytes per sectors in B	4096	-	-
Weight in g	630	51	-
MTBF in h	-	$1.8 \cdot 10^6$	-
Power in W	2.5	5.7	-

Table 14.1: Comparison between mass storage devices.

Optical disks, used for removable media devices, use a technology based on the laser. Data is encoded on a surface of a disk using non reflective or reflective spots. The reading/writing head provide a laser LED to create non reflective spots and a sensor to detect reflected laser light. Unlike the mechanical disk drive which uses the magnetization of a surface to read and write data.

14.2 The OpenBSD Enhanced Fast File System.

The *Enhanced Fast Filesystem* (FFS2) is the new file system by default on nearly all architectures, since OpenBSD 6.7. Some characteristics are:

- FFS2 is faster than its predecessor FFS when creating the file system, as well as analyzing it with *fsck*(8);
- FFS2 uses 64-bit timestamps and block numbers; so it is not subject to the Y2038 bug;
- FFS2 supports very large partitions (>= 1TB, since 4.2).

[1] On both sides.

14.2.1 The disk label.

Each *disk* or *disk pack* on a system may contain a *disk label* which provides detailed information about the geometry of the disk and the partitions into which the disk is divided. The disk label structure is defined in <sys/disklabel.h>:

```
#define NDDATA 5
#define NSPARE 4
#define MAXMAXPARTITIONS 22

struct disklabel {
  u_int32_t d_magic;
  u_int16_t d_type;
  u_int16_t d_subtype;
  char d_typename[ 16 ];
  char d_packname[ 16 ];
  u_int32_t d_secsize;
  u_int32_t d_nsectors;
  u_int32_t d_ntracks;
  u_int32_t d_ncylinders;
  u_int32_t d_secpercyl;
  u_int32_t d_secperunit;
  u_char d_uid[ 8 ];
  u_int32_t d_acylinders;
  u_int16_t d_bstarth;
  u_int16_t d_bendh;
  u_int32_t d_bstart;
  u_int32_t d_bend;
  u_int32_t d_flags;
  u_int32_t d_spare4[ NDDATA ];
  u_int16_t d_secperunith;
  u_int16_t d_version;
  u_int32_t d_spare[ NSPARE ];
  u_int32_t d_magic2;
  u_int16_t d_checksum;
  u_int16_t d_npartitions;
  u_int32_t d_spare2;
  u_int32_t d_spare3;
  struct  partition {
    u_int32_t p_size;
    u_int32_t p_offset;
    u_int16_t p_offseth;
    u_int16_t p_sizeh;
    u_int8_t p_fstype;
    u_int8_t p_fragblock;
    u_int16_t p_cpg;
  } d_partitions[ MAXPARTITIONS ];
};
```

d_magic the magic number;

d_type drive type:

- DTYPE_SMD — SMD, XSMD; VAX hp/up;

- DTYPE_MSCP — MSCP;
- DTYPE_DEC — other DEC (rk, rl);
- DTYPE_SCSI — SCSI;
- DTYPE_ESDI — ESDI interface;
- DTYPE_ST506 — ST506 etc.;
- DTYPE_HPIB — CS/80 on HP-IB;
- DTYPE_HPFL — HP Fiber-link;
- DTYPE_FLOPPY — floppy;
- DTYPE_CCD — was: concatenated disk device;
- DTYPE_VND — vnode pseudo-disk;
- DTYPE_ATAPI — ATAPI;
- DTYPE_RAID — was: RAIDframe;
- DTYPE_RDROOT — ram disk root;

d_subtype	controller/d_type specific;
d_typename	type name, e.g. "eagle";
d_packname	pack identifier;
d_secsize	number of bytes per sector;
d_nsectors	number of data sectors per track;
d_ntracks	number of tracks per cylinder;
d_ncylinders	number of data cylinders per unit;
d_secpercyl	number of data sectors per cylinder;
d_secperunit	number of data sectors (low part);
d_uid	unique label identifier;
d_acylinders	number of alt. cylinders per unit;
d_bstarth	start of useable region (high part);
d_bendh	size of useable region (high part);
d_bstart	start of useable region;
d_bend	end of useable region;
d_flags	generic flags;
d_spare4	structure pad data;
d_secperunith	number of data sectors (high part);
d_version	version number (1=48 bit addressing);
d_spare	structure pad data, reserved for future use;
d_magic2	the magic number (again);

`d_checksum` xor of data incl. partitions;

`d_npartitions` number of partitions in following;

`d_spare2` spare member;

`d_spare3` spare member;

`d_partitions` the partition table, array of a structure with the following members:

- `p_size` — number of sectors (low part);
- `p_offset` — starting sector (low part);
- `p_offseth` — starting sector (high part);
- `p_sizeh` — number of sectors (high part);
- `p_fstype` — filesystem type, see below;
- `p_fragblock` — encoded filesystem frag/block;
- `p_cpg` — UFS: FS cylinders per group.

It should be initialized when the disk is formatted, and may be changed later with the *disklabel*(8) program. This information is used by the system disk driver and by the bootstrap program to determine how to program the drive and where to find the file systems on the disk partitions. Additional information is used by the file system in order to use the disk most efficiently and to locate important information. The description of each partition contains an identifier for the partition type: standard file system, swap area, etc.. The file system updates the in-core copy of the label if it contains incomplete information about the file system itself. The *label* is located in sector number `LABELSECTOR` of the drive, usually sector 0 where it may be found without any information about the disk geometry. It is at an offset `LABELOFFSET` from the beginning of the sector, to allow room for the initial bootstrap. A copy of the in-core label for a disk can be obtained with the `DIOCGDINFO ioctl`; this works with a file descriptor for a block or character, *raw*, device for any partition of the disk. The in-core copy of the label is set by the `DIOCSDINFO ioctl`. The offset of a partition cannot generally be changed while it is open, nor can it be made smaller while it is open. One exception is that any change is allowed if no label was found on the disk and the driver was able to construct only a *skeletal label* without partition information. The `DIOCWDINFO ioctl` operation sets the *in-core label* and then updates the *on-disk label*; there must be an existing label on the disk for this operation to succeed. Thus, the initial label for a disk or disk pack must be installed by writing to the raw disk. The `DIOCGPDINFO ioctl` operation gets the default label for a disk. This simulates the case where there is no physical label on the disk itself and can be used to see the label the kernel would construct in that case. The `DIOCRLDINFO ioctl` operation causes the kernel to update its copy of the label based on the physical label on the disk. It can be used when the on-disk version of the label was changed directly or, if there is no physical label, to update the kernel's skeletal label if some variable affecting label generation has changed, e.g. the fdisk partition table. All of these operations are normally done using *disklabel*(8). Note that when a disk has no real *BSD disk label* the kernel creates a default label so that the disk can be used. This default label will include other partitions found on the disk if they are supported on your architecture. For example, on systems that support *fdisk*(8) partitions the default label will also include DOS and Linux partitions. However, these entries are not dynamic, they are fixed at the time *disklabel*(8) is run. That means that subsequent changes that affect non-OpenBSD partitions will not be present in the default label, though you may update them by hand.

Listing 14.1: disklabel - a program to retrieve disk label.

```
/* -*- mode: c-mode; -*- */

```

```c
/* File disklabel.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/dkio.h>
#include <sys/disklabel.h>

/* program disklabel. */
#define FOREVER for(;;)

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int diskfd;
  long int ret;
  struct disklabel label;

  /* Check arguments. */
  if(argc == 2) {
    if(pledge("stdio␣disklabel␣unveil␣rpath␣wpath", NULL) >= 0) {
      if(unveil(argv[ 1 ], "rw") >= 0) {
        if((diskfd = open(argv[ 1 ], O_RDWR)) >= 0) {
          if(ioctl(diskfd, DIOCGPDINFO, &label) >= 0) {
            printf("magic␣number:␣0x%0.8x\n", label.d_magic);
            printf("drive␣type:␣0x%0.4x\n", label.d_type);
            printf("drive␣subtype:␣0x%0.4x\n", label.d_subtype);
            printf("type␣name:␣%s\n", label.d_typename);
            printf("pack␣name:␣%s\n", label.d_packname);
            printf("bytes␣per␣sector:␣0x%0.8x\n", label.d_secsize);
            printf("sectors␣per␣track:␣0x%0.8x\n", label.d_nsectors);
            printf("tracks␣per␣cylinder:␣0x%0.8x\n", label.d_ntracks);
            printf("data␣cylinders␣per␣unit:␣0x%0.8x\n", label.d_ncylinders);
            printf("data␣sectors␣per␣cylinder:␣0x%0.8x\n", label.d_secpercyl);
            printf("data␣sectors␣per␣unit:␣0x%0.8x\n", label.d_secperunit);
            ret = EXIT_SUCCESS;
          } else
            perror("ioctl");
          close(diskfd);
        } else
          perror("open");
      } else
        perror("unveil");
    } else
      perror("pledge");
  } else
    fprintf(stderr, "usage:␣disklabel␣<device>\n");
  exit(ret);
}

/* End of disklabel.c file. */
```

In the listing 14.1 we used some new system calls: `unveil` and `pledge`. The latter, `pledge`, allows you to limit a program's access to system calls very easily. This is a huge improvement in security, for example: even if a binary is compromised, its chances to misbehave are greatly reduced. The usage is very simple:

```c
int main(int argc, char *argv[])
{
  ...
  if(pledge("stdio␣rpath", NULL) == -1)
    err(1, "pledge");
  ...
}
```

The first call to `unveil` that specifies a path removes visibility of the entire file system from all other file system-related system calls, such as *open*(2), *chmod*(2) and *rename*(2), except for the specified path and permissions. The `unveil` system call remains capable of traversing to any path in the file system, so additional calls can set permissions at other points in the file system hierarchy. After establishing a collection of path and permissions rules, future calls to `unveil` can be disabled by passing two NULL arguments. Alternatively, *pledge*(2) may be used to remove the `unveil` promise. In listing 14.2 we showed a program to retrieve, from disklabel, the partitions information:

Listing 14.2: disklabel2 - a program to retrieve partitions information.

```c
/* -*- mode: c-mode; -*- */

/* File disklabel2.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/dkio.h>
#include <sys/disklabel.h>

/* program disklabel2. */
#define FOREVER for(;;)

/* Functions prototypes. */
int main(int, char *[]);
```

```

/* Main function. */
int main(int argc, char *argv[])
{
  int i, diskfd;
  long int ret;
  struct disklabel label;

  /* Check arguments. */
  if(argc == 2) {
    if(pledge("stdio␣disklabel␣unveil␣rpath␣wpath", NULL) >= 0) {
      if(unveil(argv[ 1 ], "rw") >= 0) {
        if((diskfd = open(argv[ 1 ], O_RDWR)) >= 0) {
          if(ioctl(diskfd, DIOCGDINFO, &label) >= 0) {
            for(i = 0; i < label.d_npartitions; i++) {
              printf("\npartion␣#%d\n", i);
              printf("partition␣number␣of␣sectors:␣%u\n", (off_t)
                   label.d_partitions[ i ].p_size | ((off_t)
                  label.d_partitions[ i ].p_sizeh << 32));
              printf("partition␣starting␣sector:␣%u\n", (off_t)
                  label.d_partitions[ i ].p_offset | ((off_t)
                  label.d_partitions[ i ].p_offseth << 32));
              printf("partition␣filesystem␣type:␣%d\n",     label
                  .d_partitions[ i ].p_fstype);
              printf("partition␣encoded␣filesystem␣frag/block:␣%d
                  \n",      label.d_partitions[ i ].p_fragblock);
              printf("partition␣cylinders␣per␣group:␣%d\n",
                  label.d_partitions[ i ].p_cpg);
            }
            ret = EXIT_SUCCESS;
          } else
            perror("ioctl");
          close(diskfd);
        } else
          perror("open");
      } else
        perror("unveil");
    } else
      perror("pledge");
  } else
    fprintf(stderr, "usage:␣disklabel␣<device>\n");
  exit(ret);
}

/* End of disklabel2.c file. */
```

14.2.2 The file system.

The files `<ufs/ffs/fs.h>` and `<ufs/ufs/inode.h>` declare several structures and define variables and macros which are used to create and manage the underlying format of file system objects on random access devices such as disks. The *block size* and *number of blocks* are defining parameters of the file system. Sectors beginning at BBLOCK and continuing for BBSIZE are used for a *disklabel* and for some hardware primary and secondary bootstrapping programs. The actual file system begins at sector SBLOCK with the super-block that is of size SBSIZE. The following structure describes the super-block and is from the file <ufs/ffs/fs.h>:

Listing 14.3: The fs structure.

```
#define FS_MAGIC 0x011954
#define MAXMNTLEN 468
#define MAXVOLLEN 32
#define NOCSPTRS ((128 / sizeof(void *)) - 4)
#define FSMAXSNAP 20

struct fs {
  int32_t fs_firstfield;
  int32_t fs_unused_1;
  int32_t fs_sblkno;
  int32_t fs_cblkno;
  int32_t fs_iblkno;
  int32_t fs_dblkno;
  int32_t fs_cgoffset;
  int32_t fs_cgmask;
  int32_t fs_ffs1_time;
  int32_t fs_ffs1_size;
  int32_t fs_ffs1_dsize;
  int32_t fs_ncg;
  int32_t fs_bsize;
  int32_t fs_fsize;
  int32_t fs_frag;
  int32_t fs_minfree;
  int32_t fs_rotdelay;
  int32_t fs_rps;
  int32_t fs_bmask;
  int32_t fs_fmask;
  int32_t fs_bshift;
  int32_t fs_fshift;
  int32_t fs_maxcontig;
  int32_t fs_maxbpg;
  int32_t fs_fragshift;
  int32_t fs_fsbtodb;
  int32_t fs_sbsize;
  int32_t fs_csmask;
  int32_t fs_csshift;
  int32_t fs_nindir;
  int32_t fs_inopb;
  int32_t fs_nspf;
  int32_t fs_optim;
  int32_t fs_npsect;
  int32_t fs_interleave;
  int32_t fs_trackskew;
  int32_t fs_id[ 2 ];
  int32_t fs_ffs1_csaddr;
  int32_t fs_cssize;
  int32_t fs_cgsize;
```

```
int32_t fs_ntrak;
int32_t fs_nsect;
int32_t fs_spc;
int32_t fs_ncyl;
int32_t fs_cpg;
int32_t fs_ipg;
int32_t fs_fpg;
struct csum fs_ffs1_cstotal;
int8_t fs_fmod;
int8_t fs_clean;
int8_t fs_ronly;
int8_t fs_ffs1_flags;
u_char fs_fsmnt[ MAXMNTLEN ];
u_char fs_volname[ MAXVOLLEN ];
u_int64_t fs_swuid;
int32_t fs_pad;
int32_t fs_cgrotor;
void *fs_ocsp[ NOCSPTRS ];
u_int8_t *fs_contigdirs;
struct csum *fs_csp;
int32_t *fs_maxcluster;
u_char *fs_active;
int32_t fs_cpc;
int32_t fs_maxbsize;
int64_t fs_spareconf64[ 17 ];
int64_t fs_sblockloc;
struct csum_total fs_cstotal;
int64_t fs_time;
int64_t fs_size;
int64_t fs_dsize;
int64_t fs_csaddr;
int64_t fs_pendingblocks;
int32_t fs_pendinginodes;
int32_t fs_snapinum[ FSMAXSNAP ];
int32_t fs_avgfilesize;
int32_t fs_avgfpdir;
int32_t fs_sparecon[ 26 ];
u_int32_t fs_flags;
int32_t fs_fscktime;
int32_t fs_contigsumsize;
int32_t fs_maxsymlinklen;
int32_t fs_inodefmt;
u_int64_t fs_maxfilesize;
int64_t fs_qbmask;
int64_t fs_qfmask;
int32_t fs_state;
int32_t fs_postblformat;
int32_t fs_nrpos;
int32_t fs_postbloff;
int32_t fs_rotbloff;
int32_t fs_magic;
u_int8_t fs_space[ 1 ];
```

```
};
```

The members are:

`fs_firstfield`	historic file system linked list, used for incore super blocks;
`fs_unused_1`	unused member;
`fs_sblkno`	address of super-block / frags;
`fs_cblkno`	offset of cylinder-block / frags;
`fs_iblkno`	offset of inode-blocks / frags;
`fs_dblkno`	offset of first data / frags;
`fs_cgoffset`	cylinder group offset in cylinder;
`fs_cgmask`	used to calc mod fs_ntrak;
`fs_ffs1_time`	last time written;
`fs_ffs1_size`	number of blocks in fs / frags;
`fs_ffs1_dsize`	number of data blocks in fs;
`fs_ncg`	number of cylinder groups;
`fs_bsize`	size of basic blocks / bytes;
`fs_fsize`	size of frag blocks / bytes;
`fs_frag`	number of frags in a block in fs;
`fs_minfree`	minimum percentage of free blocks;
`fs_rotdelay`	number of ms for optimal next block;
`fs_rps`	disk revolutions per second;
`fs_bmask`	“blkoff” calc of blk offsets;
`fs_fmask`	“fragoff” calc of frag offsets;
`fs_bshift`	“lblkno” calc of logical blkno;
`fs_fshift`	“numfrags” calc number of frags;
`fs_maxcontig`	maximum number of contiguous blocks;
`fs_maxbpg`	maximum number of blocks per cylinder group;
`fs_fragshift`	block to frag shift;
`fs_fsbtodb`	fsbtodb and dbtofsb shift constant;
`fs_sbsize`	actual size of super block;
`fs_csmask`	csum block offset (now unused);
`fs_csshift`	csum block number (now unused);
`fs_nindir`	value of NINDIR;

fs_inopb	i-nodes per file system block;
fs_nspf	DEV_BSIZE sectors per frag;
fs_optim	optimization preference;
fs_npsect	DEV_BSIZE sectors/track + spares;
fs_interleave	DEV_BSIZE sector interleave;
fs_trackskew	sector 0 skew, per track;
fs_id	unique filesystem id;
fs_ffs1_csaddr	block address of cylinder groyup summary area;
fs_cssize	cylinder group summary area size / bytes;
fs_cgsize	cylinder group block size / bytes;
fs_ntrak	tracks per cylinder;
fs_nsect	DEV_BSIZE sectors per track;
fs_spc	DEV_BSIZE sectors per cylinder;
fs_ncyl	cylinders in file system;
fs_cpg	cylinders per group;
fs_ipg	inodes per group;
fs_fpg	blocks per group * fs_frag;
fs_ffs1_cstotal	cylinder summary information;
fs_fmod	super-block modified flag;
fs_clean	file system is clean flag;
fs_ronly	mounted read-only flag;
fs_ffs1_flags	see FS_ below;
fs_fsmnt	name mounted on;
fs_volname	volume name;
fs_swuid	system-wide uid;
fs_pad	due to alignment of fs_swuid;
fs_cgrotor	last cg searched;
fs_ocsp	padding; was list of fs_cs bufs;
fs_contigdirs	number of contiguously allocated directories;
fs_csp	cg summary info buffer for fs_cs;
fs_maxcluster	maximum cluster in each cylinder group;
fs_active	reserved for snapshots;

fs_cpc	cylinder per cycle in postbl;
fs_maxbsize	maximum blocking factor permitted;
fs_spareconf64	old rotation block list head;
fs_sblockloc	offset of standard super block;
fs_cstotal	cylinder summary information;
fs_time	time last written;
fs_size	number of blocks in fs;
fs_dsize	number of data blocks in fs;
fs_csaddr	block address of cylinder group summary area;
fs_pendingblocks	blocks in process of being freed;
fs_pendinginodes	i-nodes in process of being freed;
fs_snapinum	space reserved for snapshots;
fs_avgfilesize	expected average file size;
fs_avgfpdir	expected number of files per directory;
fs_sparecon	reserved for future constants;
fs_flags	see FS_ flags below;
fs_fscktime	last time *fsck*(8)ed;
fs_contigsumsize	size of cluster summary array;
fs_maxsymlinklen	maximum length of an internal symlink;
fs_inodefmt	format of on-disk i-nodes;
fs_maxfilesize	maximum representable file size;
fs_qbmask	~fs_bmask - for use with quad size;
fs_qfmask	~fs_fmask - for use with quad size;
fs_state	validate fs_clean field;
fs_postblformat	format of positional layout tables;
fs_nrpos	number of rotational positions;
fs_postbloff	(u_int16) rotation block list head;
fs_rotbloff	(u_int8) blocks for each rotation;
fs_magic	magic number;
fs_space	list of blocks for each rotation.

Each disk drive contains some number of file systems. A file system consists of a number of cylinder groups. Each *cylinder group* has *inodes* and data. A file system is described by its super-block, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes. Addresses stored in inodes are capable of addressing fragments of "blocks". File system blocks of at most size MAXBSIZE can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be DEV_BSIZE, or some multiple of a DEV_BSIZE unit. Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a small file is allocated only as many fragments of a large block as are necessary. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the inode, using the blksize macro. The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined. The *root inode*, as the name implies, is the root of the file system. Inode 0 can't be used for normal purposes and historically bad blocks were linked to inode 1[2] . Thus the root inode is 2. The fs_minfree element gives the minimum acceptable percentage of file system blocks that may be free. If the freelist drops below this level, only the super-user may continue to allocate blocks. The fs_minfree element may be set to 0 if no reserve of free blocks is deemed necessary, although severe performance degradations will be observed if the file system is run at greater than 95% full; thus the default value of fs_minfree is 5%. Empirically the best trade-off between block fragmentation and overall disk utilization at a loading of 95% comes with a fragmentation of 8; thus the default fragment size is an eighth of the block size. The element fs_optim specifies whether the file system should try to minimize the time spent allocating blocks (FS_OPTTIME), or if it should attempt to minimize the space fragmentation on the disk (FS_OPTSPACE). If the value of fs_minfree is less than 5%, then the file system defaults to optimizing for space to avoid running out of full sized blocks. If the value of fs_minfree is greater than or equal to 5%, fragmentation is unlikely to be problematical and the file system defaults to optimizing for time. The fs_flags element specifies how the file system was mounted:

FS_UNCLEAN the file system was mounted uncleanly.

14.2.3 Cylinder group related limits.

Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. With the default of 1 distinct rotational position, the resolution of the summary information is 16 ms for a typical 3600 RPM drive. The element fs_rotdelay was once used to tweak block layout. Each file system has a statically allocated number of inodes, determined by its size and the desired number of file data bytes per inode at the time it was created. See *newfs*(8) for details on how to set this and other file system parameters. By default, the inode allocation strategy is extremely conservative. MINBSIZE is the smallest allowable block size. With a MINBSIZE of 4096 it is possible to create files of size 2^{32} with only two levels of indirection. MINBSIZE must be big enough to hold a cylinder group block, thus changes to struct cg must keep its size within MINBSIZE. Note that super-blocks are never more than size SBSIZE. The path name on which the file system is mounted is maintained in fs_fsmnt. MAXMNTLEN defines the amount of space allocated in the super-block for this name. Per *cylinder group information* is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from fs_csaddr, of size fs_cssize, in addition to the super-block. Note that sizeof(struct csum) must be a power of two in order for the fs_cs macro to work.

[2] Inode 1 is no longer used for this purpose; however, numerous dump tapes make this assumption, so we are stuck with it.

14.2.4 Super-block for a file system.

The size of the *rotational layout tables* is limited by the fact that the super-block is of size SBSIZE. The size of these tables is inversely proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats, fs_cpc. The size of the rotational layout tables is derived from the number of bytes remaining in struct fs. The number of blocks of data per cylinder group is limited because cylinder groups are at most one block. The inode and free block tables must fit into a single block after deducting space for the cylinder group structure struct cg. In listing 14.4 we show a program that read the superblock of a file system and shows some informations.

Listing 14.4: superblock - a program to retrieve a file system superblock.

```
/* -*- mode: c-mode; -*- */

/* File superblock.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <util.h>
#include <fstab.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/dkio.h>
#include <sys/buf.h>
#include <sys/disklabel.h>
#include <ufs/ffs/fs.h>
#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>

/* program superblock. */
#define FOREVER for(;;)

/* Types. */
union tagFS {
  struct fs u_fs;
  char u_pad[ SBSIZE ];
};

typedef union tagFS fsu_t;

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  char *name, *realdev;
  int i, diskfd;
  long int ret;
  fsu_t fsun1;
  off_t sbtry[] = SBLOCKSEARCH;
  ssize_t n;
  struct fstab *fs;

  /* Check arguments. */
  if(argc == 2) {
    if(pledge("stdio␣rpath␣disklabel", NULL) >= 0) {
      if((fs = getfsfile(argv[ 1 ])) != NULL)
        name = fs -> fs_spec;
      else
        name = argv[ 1 ];
      printf("Opening:␣%s\n", name);
      if((diskfd = opendev(name, O_RDONLY, 0, NULL)) >= 0) {
        for(i = 0; sbtry[ i ] != 1; i++) {
          n = pread(diskfd, &fsun1.u_fs, SBLOCKSIZE, (off_t) sbtry[ i ]);
          if(n == SBLOCKSIZE && (fsun1.u_fs.fs_magic == FS_UFS1_MAGIC ||  \
            (fsun1.u_fs.fs_magic == FS_UFS2_MAGIC &&            \
            fsun1.u_fs.fs_sblockloc == sbtry[ i ])) &&          \
            !(fsun1.u_fs.fs_magic == FS_UFS1_MAGIC &&         \
            sbtry[ i ] == SBLOCK_UFS2) &&               \
            fsun1.u_fs.fs_bsize <= MAXBSIZE &&              \
            fsun1.u_fs.fs_bsize >= sizeof(struct fs)) {
            printf("super-block␣shift␣constant:␣%d\n", fsun1.u_fs.fs_fsbtodb);
            printf("super-block␣magic␣number:␣0x%0.8x\n", sun1.u_fs.fs_magic);
            printf("super-block␣offset:␣%d\n", fsun1.u_fs.fs_sblkno);
            ret = EXIT_SUCCESS;
            break;
          }
          if(sbtry[ i ] == -1)
            fprintf(stderr, "Could␣not␣find␣superblock␣for␣%s\n", argv[ 1 ]);
        }
        close(diskfd);
      } else
        perror("opendev");
    } else
      perror("pledge");
  } else
    fprintf(stderr, "usage:␣superblock␣<fs>\n");
  exit(ret);
}

/* End of superblock.c file. */
```

The program shows also the usage of the system call getfsfile. This function return a pointer to an object with the following structure containing the broken-out fields of a line in the file system description file <fstab.h>:

Listing 14.5: The fstab structure.

```
struct fstab {
  char *fs_spec;
  char *fs_file;
  char *fs_vfstype;
  char *fs_mntops;
  char *fs_type;
  int fs_freq;
  int fs_passno;
};
```

The members are:

fs_spec block special device name;

fs_file file system path prefix;

fs_vfstype type of file system;

fs_mntops comma separated mount options;

fs_type rw, ro, sw, or xx;

fs_freq dump frequency, in days;

fs_passno pass number on parallel fsck.

The fields have meanings described in *fstab*(5). getfsfile function searches the entire file, opening it if necessary, for a matching special file name or file system file name. All entries in the file with a type field equivalent to FSTAB_XX are ignored. Lines which are formatted incorrectly are silently ignored. The getfsfile function returns a null pointer on EOF or error. It is interesting to note that depending on the file system type, the super-block is located at different positions. To achieve a correct search, the offset in pread have to assume the values in the array sbtry which values are provided by the SBLOCKSEARCH macro from <ufs/ffs/fs.h>:

```
#define BBSIZE          8192
#define SBSIZE          8192
#define BBOFF           ((off_t)(0))
#define SBOFF           ((off_t)(BBOFF + BBSIZE))
#define BBLOCK          ((daddr_t)(0))
#define SBLOCK          ((daddr_t)(BBLOCK + BBSIZE / DEV_BSIZE))
#define SBLOCK_UFS1     8192
#define SBLOCK_UFS2     65536
#define SBLOCK_PIGGY    262144
#define SBLOCKSIZE      8192
#define SBLOCKSEARCH \
        { SBLOCK_UFS2, SBLOCK_UFS1, SBLOCK_PIGGY, -1 }
```

To computes right values for the various quantities involved in the file system structure there are a number of macros defined in <ufs/ffs/fs.h>:

fsbtodb(fs,b) turn file system block numbers into disk block addresses;

dbtofsb(fs,b) this maps file system blocks to DEV_BSIZE (a.k.a. 512-byte) size disk blocks.

The following are cylinder group macros to locate things in cylinder groups, they compute file system addresses of cylinder group data structures:

`cgbase(fs,c)`	cylinder group base;
`cgdata(fs,c)`	cylinder group data zone;
`cgmeta(fs,c)`	cylinder group meta data;
`cgdmin(fs,c)`	cylinder group 1st data;
`cgimin(fs,c)`	cylinder group inode blk;
`cgsblock(fs,c)`	cylinder group super blk;
`cgtod(fs,c)`	cylinder group block;
`cgstart(fs,c)`	start of cylinder group;

Macros for handling inode numbers:

`ino_to_cg(fs,x)`	inode number to file system block offset;
`ino_to_fsba(fs,x)`	inode number to file system block address;
`ino_to_fsbo(fs,x)`	inode number to file system block offset;
`dtog(fs,d)`	give cylinder group number for a file system block;
`dtogd(fs,d)`	give frag block number in cylinder group for a file system block;
`blkmap(fs,map,loc)`	extract the bits for a block from a map;
`cbtocylno(fs,bno)`	compute the cylinder block address;
`cbtorpos(fs,bno)`	compute the cylinder rotational position block address;

The following macros optimize certain frequently calculated quantities by using shifts and masks in place of divisions /, modulos % and multiplications *:

`blkoff(fs,loc)`	calculates `(loc % fs->fs_bsize)`;
`fragoff(fs,loc)`	calculates `(loc % fs->fs_fsize)`;
`lblktosize(fs,blk)`	calculates `((off_t) blk * fs->fs_bsize)`;
`lblkno(fs,loc)`	calculates `(loc / fs->fs_bsize)`;
`numfrags(fs,loc)`	calculates `(loc / fs->fs_fsize)`;
`blkroundup(fs,size)`	calculates `roundup(size, fs->fs_bsize)`;
`fragroundup(fs,size)`	calculates `roundup(size, fs->fs_fsize)`;
`fragstoblks(fs,frags)`	calculates `(frags / fs->fs_frag)`;
`blkstofrags(fs,blks)`	calculates `(blks * fs->fs_frag)`;
`fragnum(fs,fsb)`	calculates `(fsb % fs->fs_frag)`;
`blknum(fs,fsb)`	calculates `rounddown(fsb, fs->fs_frag)`;

`freespace(fs,p)` determine the number of available frags given a percentage to hold in reserve.

Determining the size of a file block in the file system:

`blksize(fs,ip,lbn)` dimension of a block;

`dblksize(fs,dip,lbn)` dimension of dinode block;

`sblksize(fs,size,lbn)` dimension of the super-block;

`NSPB(fs)` number of disk sectors per block; assumes `DEV_BSIZE` byte sector size;

`NSPF(fs)` number of disk sectors per fragment; assumes `DEV_BSIZE` byte sector size;

`INOPB(fs)` number of inodes per file system block (`fs->fs_bsize`);

`INOPF(fs)` number of inodes per file system fragment (`fs->fs_fsize`);

`NINDIR(fs)` number of indirects in a file system block.

`FS_KERNMAXFILESIZE(pgsiz,fs)` maximum file size the kernel allows. Even though ffs can handle files up to 16 TB, the max file is limited to 2^{31} pages to prevent overflow of a 32-bit unsigned int. The buffer cache has its own checks but a little added paranoia never hurts:

14.2.5 Inodes.

The inode is the focus of all file activity in the UNIX file system. There is a unique inode allocated for each active file, each current directory, each mounted-on file, text file and the root. An i-node is *named* by its device/i-number pair. The *on-disk inode* structure is called `dinode` and is defined in the include file `<ufs/ufs/dinode.h>`:

Listing 14.6: The dinode structures.

```
#define NXADDR 2    /* External addresses in inode */
#define NDADDR 12   /* Direct addresses in inode. */
#define NIADDR 3    /* Indirect addresses in inode. */

struct  ufs1_dinode {
  u_int16_t di_mode;        /*   0: IFMT, permissions; see below.
      */
  int16_t di_nlink;         /*   2: File link count. */
  union {
    u_int16_t oldids[2];    /*   4: Ffs: old user and group ids.
      */
    u_int32_t inumber;      /*   4: Lfs: inode number. */
  } di_u;
  u_int64_t di_size;        /*   8: File byte count. */
  int32_t di_atime;         /*  16: Last access time. */
  int32_t di_atimensec;     /*  20: Last access time. */
  int32_t di_mtime;         /*  24: Last modified time. */
  int32_t di_mtimensec;     /*  28: Last modified time. */
  int32_t di_ctime;         /*  32: Last inode change time. */
```

```
    int32_t di_ctimensec;      /*  36: Last inode change time. */
    int32_t di_db[ NDADDR ];   /*  40: Direct disk blocks. */
    int32_t di_ib[ NIADDR ];   /*  88: Indirect disk blocks. */
    u_int32_t di_flags;        /* 100: Status flags (chflags). */
    int32_t di_blocks;         /* 104: Blocks actually held. */
    u_int32_t di_gen;          /* 108: Generation number. */
    u_int32_t di_uid;          /* 112: File owner. */
    u_int32_t di_gid;          /* 116: File group. */
    int32_t di_spare[ 2 ];     /* 120: Reserved; currently unused */
  };

  struct ufs2_dinode {
    u_int16_t di_mode;         /*   0: IFMT, permissions; see below.
        */
    int16_t di_nlink;          /*   2: File link count. */
    u_int32_t di_uid;          /*   4: File owner. */
    u_int32_t di_gid;          /*   8: File group. */
    u_int32_t di_blksize;      /*  12: Inode blocksize. */
    u_int64_t di_size;         /*  16: File byte count. */
    u_int64_t di_blocks;       /*  24: Bytes actually held. */
    int64_t di_atime;          /*  32: Last access time. */
    int64_t di_mtime;          /*  40: Last modified time. */
    int64_t di_ctime;          /*  48: Last inode change time. */
    int64_t di_birthtime;      /*  56: Inode creation time. */
    int32_t di_mtimensec;      /*  64: Last modified time. */
    int32_t di_atimensec;      /*  68: Last access time. */
    int32_t di_ctimensec;      /*  72: Last inode change time. */
    int32_t di_birthnsec;      /*  76: Inode creation time. */
    int32_t di_gen;            /*  80: Generation number. */
    u_int32_t di_kernflags;    /*  84: Kernel flags. */
    u_int32_t di_flags;        /*  88: Status flags (chflags). */
    int32_t di_extsize;        /*  92: External attributes block. */
    int64_t di_extb[ NXADDR ];/*  96: External attributes block. */
    int64_t di_db[ NDADDR ];   /* 112: Direct disk blocks. */
    int64_t di_ib[ NIADDR ];   /* 208: Indirect disk blocks. */
    int64_t di_spare[ 3 ];     /* 232: Reserved; currently unused */
  };
```

As mentioned previously, one of the reasons to read the raw file system structure rather than going through the operating system is to calculate disk space usage. To retrieve informations about a directory or a file we can use fstat which reads these from the disk. fstat and related struct fstat are defined in <sys/stat.h> described in 5.1. Listing takes inode informations from a file specified in the command:

Listing 14.7: inode - a program to retrieve a file inode information.

```
/* -*- mode: c-mode; -*- */

/* File inode.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#include <util.h>
#include <fstab.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include <sys/dkio.h>
#include <sys/buf.h>
#include <sys/disklabel.h>
#include <sys/stat.h>

/* program inode. */
#define FOREVER for(;;)

/* Types. */

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i, fd;
  long int ret;
  off_t offset;
  ssize_t n;  /* sysv stupidity */
  struct stat sb;

  /* Check arguments. */
  if(argc == 2) {
    printf("Opening␣file:␣%s\n", argv[ 1 ]);
    if((fd = open(argv[ 1 ], O_RDONLY, 0, NULL)) >= 0) {
      if(fstat(fd, &sb) >= 0) {
        printf("inode's␣device:␣%lld\n", sb.st_dev);
        printf("inode's␣number:␣%lld\n", sb.st_ino);
        printf("inode␣protection␣mode:␣0x%0.6x\n", sb.st_mode);
        printf("number␣of␣hard␣links:␣%lld\n", sb.st_nlink);
        printf("user␣ID␣of␣the␣file's␣owner:␣%lld\n", sb.st_uid);
        printf("group␣ID␣of␣the␣file's␣group:␣%lld\n", sb.st_gid);
        printf("device␣type:␣%d\n", sb.st_rdev);
        printf("time␣of␣last␣access:␣%s", ctime(&sb.st_atim.tv_sec));
        printf("time␣of␣last␣data␣modification:␣%s", ctime(&sb.st_mtim.tv_sec));
        printf("time␣of␣last␣file␣status␣change:␣%s", ctime(&sb.st_ctim.tv_sec));
        printf("file␣size␣in␣bytes:␣%lld\n", sb.st_size);
      } else
        perror("stat");
      close(fd);
    } else
      perror("open");
  } else
    fprintf(stderr, "usage:␣inode␣<filename>\n");
  exit(ret);
}

/* End of inode.c file. */
```

Chapter 15

Miscellaneous Routines.

This chapter describes some useful system calls and library routines whose description don't fit well into the previous chapters.

15.1 Resource Limits.

On OpenBSD each process operates with certain limits on the resources it may use. These limits prevent processes from creating files that are considered *too large*, using too much CPU time and so on.

15.1.1 The `getrlimit` and `setrlimit` System Call.

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the `getrlimit` call and set with the `setrlimit` call. The first parameter of both system calls is one of the following:

- `RLIMIT_CORE` — the largest size (in bytes) core file that may be created;
- `RLIMIT_CPU` — the maximum amount of CPU time (in seconds) to be used by each process;
- `RLIMIT_DATA` — the maximum size, in bytes, of the data segment for a process; this includes memory allocated via *malloc*(3) and all other anonymous memory mapped via *mmap*(2);
- RLIMIT_FSIZE — the largest size (in bytes) file that may be created;
- RLIMIT_MEMLOCK — the maximum size, in bytes, which a process may lock into memory using the *mlock*(2) function;
- `RLIMIT_NOFILE` — the maximum number of open files for this process;
- `RLIMIT_NPROC` — the maximum number of simultaneous processes for this user id;

- RLIMIT_RSS — the maximum size, in bytes, to which a process's resident set size may grow. This setting is no longer enforced, but retained for compatibility;
- RLIMIT_STACK — the maximum size (in bytes) of the stack segment for a process, which defines how far a process's stack segment may be extended. Stack extension is performed automatically by the system, and is only used by the main thread of a process;

A *resource limit* is specified as a *soft limit* and a *hard limit*. When a soft limit is exceeded a process may receive a signal[1], but it will be allowed to continue execution until it reaches the hard limit or modifies its resource limit. The rlimit structure is used to specify the hard and soft limits on a resource:

Listing 15.1: The rlimit structure.

```
struct rlimit {
  rlim_t rlim_cur;
  rlim_t rlim_max;
};
```

The members are:

rlim_cur current (soft) limit;

rlim_max hard limit.

Only the super-user may raise the maximum limits. Other users may only alter rlim_cur within the range from 0 to rlim_max or, irreversibly, lower rlim_max. An *infinite* value for a limit is defined as RLIM_INFINITY. A value of RLIM_SAVED_CUR or RLIM_SAVED_MAX will be stored in rlim_cur or rlim_max respectively by getrlimit if the value for the current or maximum resource limit cannot be stored in an rlim_t. The values RLIM_SAVED_CUR and RLIM_SAVED_MAX should not be used in a call to setrlimit unless they were returned by a previous call to getrlimit. Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; limit is thus a built-in command to *csh*(1) and ulimit is the *sh*(1) equivalent. The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a *brk*(2) call fails if the data space limit is reached. When the stack limit is reached, the process receives a segmentation fault (SIGSEGV); if this signal is not caught by a handler using the signal stack, this signal will kill the process. A file I/O operation that would create a file larger than the process' soft limit will cause the write to fail and a signal SIGXFSZ to be generated; this normally terminates the process, but may be caught. When the soft CPU time limit is exceeded, a signal SIGXCPU is sent to the offending process. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable errno is set to indicate the error. The usual method to change resource limits is shown in listing 15.2.

Listing 15.2: setlim - change resource limits.

```
/* -*- mode: c-mode; -*- */

/* File setlim.c. */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
```

[1] For example, if the CPU time or file size is exceeded.

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

/* setlim program. */

/* Functions protitypes. */
long int setlim(int, rlim_t);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  bool ok = false;
  char *bad;
  int limit;
  long int ret = EXIT_FAILURE;
  rlim_t value;

  /* Checks arguments. */
  if(argc == 3) {
    ok = true;
    if(strncmp(argv[ 1 ], "cpu", 3) == 0)
      limit = RLIMIT_CPU;
    else if(strncmp(argv[ 1 ], "filesize", 8) == 0)
      limit = RLIMIT_FSIZE;
    else if(strncmp(argv[ 1 ], "data", 4) == 0)
      limit = RLIMIT_DATA;
    else if(strncmp(argv[ 1 ], "stack", 5) == 0)
      limit = RLIMIT_STACK;
    else if(strncmp(argv[ 1 ], "core", 4) == 0)
      limit = RLIMIT_CORE;
    else if(strncmp(argv[ 1 ], "rss", 3) == 0)
      limit = RLIMIT_RSS;
    else if(strncmp(argv[ 1 ], "memorylock", 10) == 0)
      limit = RLIMIT_MEMLOCK;
    else if(strncmp(argv[ 1 ], "nproc", 5) == 0)
      limit = RLIMIT_NPROC;
    else if(strncmp(argv[ 1 ], "openfiles", 9) == 0)
      limit = RLIMIT_NOFILE;
    else {
      ok = false;
      perror("unknown␣limit");
    }
    if(ok == true) {
      if(strncmp(argv[ 2 ], "infinity", 8) == 0)
        value = RLIM_INFINITY;
      else {
        value = (rlim_t) strtoul(argv[ 2 ], &bad, 0);
        if(*bad != '\0') {
          ok = false;
        }
      }
      if(ok == true) {
        printf("set␣limit:␣%s(%d)\tto␣value:␣%lld\n", argv[ 1 ], limit, value);
        if(setlim(limit, value) == EXIT_SUCCESS)
          ret = EXIT_SUCCESS;
        else
          perror("error␣setting␣limit");
      } else
        perror("bad␣numerical␣value␣for␣limit");
    }
  } else
    fprintf(stderr, "usage:␣setlim␣<limit>␣<value>\n");
  exit(ret);
}

/*
 * setlim -- set the limit for the process.
 */
long int setlim(int lim, rlim_t val)
{
  long int ret = EXIT_FAILURE;
  struct rlimit rlim;

  /*
   * Get the current limits so  we
   * will know the maximum value.
   */
  bzero(&rlim, sizeof(struct rlimit));
  if(getrlimit(lim, &rlim) >= 0) {
    printf("current␣limit:␣%lld\tmaximum␣limit:␣%lld\n", rlim.rlim_cur, rlim.rlim_max);

    /* Now change the current limit. */
    rlim.rlim_cur = val;
    if(setrlimit(lim, &rlim) >= 0)
      ret = EXIT_SUCCESS;
  }
  return ret;
}

/* End of setlim.c file. */
```

15.2 Obtaining Resource Usage Information.

OpenBSD allows user to take tracks of resources usage on the system. To achieve that a data structure and a system call were provided and defined in `<sys/resource.h>`. getrusage returns resource usage information and takes two arguments. The first for argument, which can be one of the following:

- `RUSAGE_SELF` — resources used by the current process;
- `RUSAGE_CHILDREN` — resources used by all the terminated children of the current process;

- RUSAGE_THREAD — resources used by the current thread.

The buffer to which the second argument points will be filled in with the following structure:

Listing 15.3: The rusage structure.

```
struct rusage {
  struct timeval ru_utime;
  struct timeval ru_stime;
  long ru_maxrss;
  long ru_ixrss;
  long ru_idrss;
  long ru_isrss;
  long ru_minflt;
  long ru_majflt;
  long ru_nswap;
  long ru_inblock;
  long ru_oublock;
  long ru_msgsnd;
  long ru_msgrcv;
  long ru_nsignals;
  long ru_nvcsw;
  long ru_nivcsw;
};
```

The fields are interpreted as follows:

ru_utime the total amount of time spent executing in user mode;

ru_stime the total amount of time spent in the system executing on behalf of the process(es);

ru_maxrss the maximum resident set size utilized, in kilobytes;

ru_ixrss an "integral" value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * ticks-of- execution;

ru_idrss an integral value of the amount of unshared memory residing in the data segment of a process, expressed in units of kilobytes * ticks-of-execution;

ru_isrss an integral value of the amount of unshared memory residing in the stack segment of a process, expressed in units of kilobytes * ticks-of-execution;

ru_minflt the number of page faults serviced without any I/O activity; here I/O activity is avoided by *reclaiming* a page frame from the list of pages awaiting reallocation;

ru_majflt the number of page faults serviced that required I/O activity;

ru_nswap the number of times a process was *swapped* out of main memory;

ru_inblock the number of times the file system had to perform input;

ru_oublock the number of times the file system had to perform output;

ru_msgsnd the number of ipc messages sent;

ru_msgrcv the number of ipc messages received;

ru_nsignals the number of signals delivered;

`ru_nvcsw` the number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed, usually to await availability of a resource;

`ru_nivcsw` the number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

The numbers `ru_inblock` and `ru_oublock` account only for real I/O; data supplied by the caching mechanism is charged only to the first process to read or write the data. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable `errno` is set to indicate the error. Listing 15.4 shows the usage data for the rusage program itself.

Listing 15.4: rusage - get usage data for the process itself.

```c
/* -*- mode: c-mode; -*- */

/* File rusage.c. */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>

/* rusage program. */

/* Functions protitypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  struct rusage usage;

  /* */
  if(getrusage(RUSAGE_SELF, &usage) >= 0) {
    printf("user␣time␣used:␣%ld␣s\n", (time_t) usage.ru_utime.
        tv_sec);
    printf("system␣time␣used:␣%ld␣s\n", (time_t) usage.ru_stime.
        tv_sec);
    printf("maximum␣resident␣set␣size:␣%ld␣kB\n", usage.ru_maxrss
        );
    printf("integral␣shared␣text␣memory␣size:␣%ld␣kBt\n", usage.
        ru_ixrss);
    printf("integral␣unshared␣data␣size:␣%ld␣kBt\n", usage.
        ru_idrss);
    printf("integral␣unshared␣stack␣size:␣%ld␣kBt\n", usage.
        ru_isrss);
    printf("page␣reclaims:␣%ld\n", usage.ru_minflt);
    printf("page␣faults:␣%ld\n", usage.ru_majflt);
    printf("swaps:␣%ld\n", usage.ru_nswap);
    printf("block␣input␣operations:␣%ld\n", usage.ru_inblock);
    printf("block␣output␣operations:␣%ld\n", usage.ru_oublock);
    printf("messages␣sent:␣%ld\n", usage.ru_msgsnd);
    printf("messages␣received:␣%ld\n", usage.ru_msgrcv);
    printf("signals␣received:␣%ld\n", usage.ru_nsignals);
    printf("voluntary␣context␣switches:␣%ld\n", usage.ru_nvcsw);
    printf("involuntary␣context␣switches:␣%ld\n", usage.ru_nivcsw);
    ret = EXIT_SUCCESS;
  }
  exit(ret);
}

/* End of rusage.c file. */
```

15.3 Manipulating Byte Strings.

Most users know the strcmp, strcpy, strncmp and strncpy routines, they work fine for NUL terminated strings of characters. They do not fit for generic usage, where we have to deal with arrays of non-printing characters such as '\0'. In that case, we could use bcmp, bcopy, bzero, memcmp, memcpy, memmove and memset.

15.3.1 The bcmp routine.

The bcmp function takes three arguments. It compares byte pointed by the first parameter against byte string pointed by the second argument, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be of length in bytes as specified in the third argument. Zero-length strings are always identical. The strings may overlap.

15.3.2 The bcopy routine.

The bcopy routine takes three arguments. It copies a number of bytes as specified in the third argument from the buffer pointed by the first argument to the buffer pointed by the second argument. The two buffers may overlap. If the length specified in the third argument is zero, no bytes are copied.

15.3.3 The bzero routine.

The bzero routine takes two arguments. It writes a number, specified in the second argument, of zero bytes to the string pointed by the first argument. If the length in the second argument is zero, bzero does nothing. The explicit_bzero variant behaves the same, but will not be removed by a compiler's dead store optimization pass, making it useful for clearing sensitive memory such as a password.

15.3.4 The memcmp routine.

The memcmp function takes three arguments. It compares byte in the string, pointed by the first argument, against byte string, pointed by the second argument. Both strings are assumed to be of length specified in the third argument. The memcmp function returns zero if the two strings are identical otherwise returns the difference between the first two differing bytes, treated as unsigned char values, so that '\200' is greater than '\0', for example. Zero-length strings are always identical.

15.3.5 The memcpy routine.

The memcpy routine takes three arguments. The memcpy function copies a number of bytes, specified by the third argument, from buffer pointed by the second argument, to buffer pointed by the first argument. If the two buffers may overlap, *memmove*(3) must be used instead. The memcpy function returns the original value of the first argument.

15.3.6 The memmove routine.

The memmove function takes three arguments. It copies the number of bytes, specified in the third argument, bytes from the buffer pointed by the second argument to the buffer pointed by the first argument. The two buffers may overlap; the copy is always done in a non-destructive manner. The memmove function returns the original value of the first argument.

15.3.7 The memset routine.

The memset function takes three arguments. It writes a count of bytes, as specified in the third argument, of the same value of the second argument, converted to an unsigned char, to the string pointed by the first argument. The memset function returns the original value of the first argument.

15.4 Environment Variables.

To handle environment variables, OpenBSD provides a set of system routines: getenv, setenv, putenv and unsetenv. These functions set, unset, and fetch environment variables from the host environment list. The getenv function obtains the current value of the environment variable name. If the variable name is not in the current environment, a null pointer is returned. The setenv function inserts or resets the environment variable name in the current environment list. If the variable name does not exist in the list, it is inserted with the given value. If the variable does exist, the argument overwrite is tested; if overwrite is zero, the variable is not reset, otherwise it is reset to the given value. The putenv function takes an argument of the form name=value. The memory pointed to by string becomes part of the environment and must not be deallocated by the caller. If the variable already exists, it will be overwritten. A common source of bugs is to pass a string argument that is a locally scoped string buffer. This will result in corruption of the environment after leaving the scope in which the variable is defined. For this reason, the setenv function is preferred over putenv. The unsetenv function deletes all instances of the variable name pointed to by name from the list. The putenv, setenv and unsetenv functions return the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error. The getenv function returns a pointer to the requested value, or NULL if it could not be found. If getenv is successful, the string returned should be considered read-only.

15.5 The Current Working Directory.

OpenBSD provides a function to return the current working directory to the user: getcwd. The getcwd function copies the absolute pathname of the current working directory into the memory referenced by the first argument and returns a pointer to the buffer. The second argument is the size, in bytes, of the array referenced by the buffer pointed by the first argument. If this pointer is not NULL and the length of the pathname plus the terminating NUL character is greater than the second argument, a null pointer is returned and errno is set to ERANGE. As an extension to IEEE Std 1003.1-2001 ("POSIX.1"), if the first argument is NULL, space is allocated as necessary to store the pathname. In this case, it is the responsibility of the caller to *free*(3) the pointer that getcwd returns. Upon successful completion, a pointer to the pathname is returned. Otherwise a null pointer is returned and errno is set to indicate the error.

15.6 Searching for Characters in Strings.

Two function are provided by OpenBSD to achieve character search in a NUL terminated string: strchr and strrchr both defined in <string.h> They takes two arguments: strchr locates the first occurrence, strrchr the last occurrence, of the character specified by the second argument, converted to a char, in the string pointed by the first argument. The terminating NUL character is considered part of the string itself. If the second argument is '\0', strchr and strrchr locate the terminating '\0'. They returns a pointer to the located character or NULL if the character does not appear in the string.

15.7 Determining Whether a File is a Terminal.

OpenBSD provides four functions: ttyname, ttyname_r, isatty defined in <unistd.h> and ttyslot defined in <stdlib.h>. These functions operate on the system file descriptors for terminal type devices. These descriptors are not related to the standard I/O FILE typedef, but refer to the special device files found in */dev* and named */dev/ttyXX* and for which an entry exists in the initialization file */etc/ttys*, see *ttys*(5). The isatty function determines if the file descriptor in the first argument refers to a valid terminal type device. The ttyname and ttyname_r functions get the related device name of a file descriptor for which isatty is true. The ttyname_r function stores the NUL-terminated pathname of the terminal associated with the file descriptor of the first argument in the character array referenced by the second argument. The array length in bytes is specified in the third argument and should have space for the name and the terminating NUL character. The maximum length of the terminal name is TTY_NAME_MAX. The ttyslot function fetches the current process's controlling terminal number from the *ttys*(5) file entry. The ttyname function returns the NUL-terminated name if the device is found and isatty is true; otherwise a null pointer is returned and errno is set to indicate the error. The ttyname_r function returns zero if successful; otherwise an error number is returned. The isatty function returns 1 if the file descriptor in the first argument is associated with a terminal device; otherwise it returns 0 and errno is set to indicate the error. The ttyslot function returns the unit number of the device file if found; otherwise the value zero is returned.

15.8 Printing Error Messages.

OpenBSD provides perror psignal, strerror and strsignal functions to help the user to deal with error conditions. perror function is defined in <stdio.h>, psignal function in <signal.h>, strerror and strsignal functions in <string.h>.

15.8.1 The perror routine.

The perror function looks up the error message string affiliated with an error number and writes it, followed by a new-line, to the standard error stream. If the argument string is not the NULL pointer and is not zero length, it is prepended to the message string and separated from it by a colon and a space, ': '. Otherwise, only the error message string is printed. The contents of the error message string are the same as those returned by strerror with argument errno.

15.8.2 The psignal routine.

The psignal routine takes two arguments. It locates the descriptive message string for the given signal number in the first argument and writes it to the standard error. If the second argument is not NULL it is written to the standard error file descriptor prior to the message string, immediately followed by a colon and a space, ': '. If the signal number is not recognized, see *sigaction*(2) for a list, the string "Unknown signal" is produced. The message strings can be accessed directly using the external array sys_siglist, indexed by recognized signal numbers. The external array sys_signame is used similarly and contains short, upper-case abbreviations for signals which are useful for recognizing signal names in user input. The defined value NSIG contains a count of the strings in sys_siglist and sys_signame.

15.8.3 The strerror routine.

There are three versions of this function: strerror, strerror_l and strerror_r. These functions map the error number specified in the first argument to an error message string. strerror and strerror_l return a string containing a maximum of NL_TEXTMAX characters, including the trailing NUL. This string is not to be modified by the calling program. The string returned by strerror may be overwritten by subsequent calls to strerror in any thread. The string returned by strerror_l may be overwritten by subsequent calls to strerror_l in the same thread. strerror_r is a thread safe version of strerror that places the error message in the specified buffer pointed by the second argument. On OpenBSD, the global locale, the thread-specific locale, and the locale argument are ignored. strerror and strerror_l return a pointer to the error message string. If an error occurs, the error code is stored in errno. strerror_r returns zero upon successful completion. If an error occurs, the error code is stored in errno and the error code is returned. These functions are defined in <string.h>.

Listing 15.5: strerror.c - prints out errors names.

```c
/* -*- mode: c-mode; -*- */

/* File strerror.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/* strerror program. */
#define FOREVER for(;;)

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i;
  long int ret;

  /* */
  ret = EXIT_FAILURE;
  for(i = 0; i <= EPROTO; i++)
    fprintf(stderr, "Error␣%d␣=␣%s\n", i, strerror(i));
  ret = EXIT_SUCCESS;
  exit(ret);
}

```

```
/* End of strerror.c file. */
```

15.8.4 The strsignal routine.

The strsignal function returns a pointer to the string describing the signal specified in the first argument. The array pointed to is not to be modified by the program, but may be overwritten by subsequent calls to strsignal.

Listing 15.6: strsignal - program to list signals names.

```
/* -*- mode: c-mode; -*- */

/* File strsignal.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

/* strsignal program. */
#define FOREVER for(;;)

/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int i;
  long int ret;

  /* */
  ret = EXIT_FAILURE;
  for(i = SIGHUP; i <= SIGTHR; i++)
    printf("Signal␣%d␣=␣%s\n", i, strsignal(i));
  ret = EXIT_SUCCESS;
  exit(ret);
}

/* End of strsignal.c file. */
```

15.9 Sorting Arrays in Memory.

OpenBSD provides three functions for sorting arrays: qsort, heapsort and mergesort. They are defined in <stdlib.h> file and take four parameters. The qsort function is a modified partition-exchange sort, or quicksort. The heapsort function is a modified selection sort. The mergesort function is a modified merge sort with exponential search intended for sorting data with pre-existing order. The qsort and heapsort functions sort an array of a number of objects specified in the second argument, the initial member of which is pointed to by the first argument. The size of each object is specified by the third argument. mergesort behaves similarly, but requires that the third argument be greater than sizeof(void *) / 2. The contents of the array pointed by the first argument are sorted in ascending order according to a comparison function pointed to by the fourth argument, which requires two arguments pointing to the objects being compared. The comparison function must return an int less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. The functions qsort and heapsort are not stable, that is, if two members compare as equal, their order in the sorted array is undefined. The function mergesort is stable. The qsort function is an implementation of C. A. R. Hoare's *quicksort* algorithm, a variant of partition-exchange sorting; in particular, see D. E. Knuth's Algorithm Q. qsort takes $O\left(n\lg n\right)$ average time. This implementation uses median selection to avoid its $O\left(n^2\right)$ worst-case behavior and will fall back to heapsort if the recursion depth exceeds $2\lg n$. The heapsort function is an implementation of J. W. J. William's *heapsort* algorithm, a variant of selection sorting; in particular, see D. E. Knuth's Algorithm H. heapsort takes $O\left(n\lg n\right)$ worst-case time. This implementation of heapsort is implemented without recursive function calls. The function mergesort requires additional memory of second argument value * third argument value bytes; it should be used only when space is not at a premium. mergesort is optimized for data with pre-existing order; its worst case time is $O\left(n\lg n\right)$; its best case is $O\left(n\right)$. Normally, qsort is faster than mergesort, which is faster than heapsort. Memory availability and pre-existing order in the data can make this untrue. The heapsort and mergesort functions return the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error. Listing 15.7 show an application of qsort.

Listing 15.7: sort - a program to show qsort capability.

```c
/* -*- mode: c-mode; -*- */

/* File sort.c. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *array[] = { "XX", "YYY", "Z" };

#define N (sizeof(array) / sizeof(array[ 0 ]))

/* Functions prototypes. */
int cmp(const void *, const void *);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  size_t i;

  /* */
  qsort(array, N, sizeof(array[0]), cmp);
  for(i = 0; i < N; i++)
    printf("%s\n", array[i]);
  ret = EXIT_SUCCESS;
  exit(ret);
}

/*
 * cmp -- comparing elements function.
 */
```

```
int cmp(const void *a, const void *b)
{
  /*
   * a and b point to elements of the array.
   * Cast and dereference to obtain the actual elements,
   * which are also pointers in this case.
   */
  size_t lena = strlen(*(const char **) a);
  size_t lenb = strlen(*(const char **) b);

  /*
   * Do not subtract the lengths. The difference between values
   * cannot be represented by an int.
   */
  return lena < lenb ? -1 : lena > lenb;
}

/* End of sort.c file. */
```

Appendix A

FORTRAN vs C Interoperability.

The OpenBSD gcc C and g95 FORTRAN compilers were written to use the same object code format. This feature permits the programmer to call FORTRAN functions from C programs and vice-versa. FORTRAN programs can use many of the C library functions, system calls. C programs can call funtions from FORTRAN libraries. Note that the information in this appendix is based on the OpenBSD gcc C and g95 FORTRAN compilers on amd64 architecture.

A.1 Data Representation.

The following tab. A.1 is of corresponding FORTRAN and C variable declarations.

Table A.1: FORTRAN 90 vs C Declarations.

FORTRAN 90	C	Extension
`character(c_char) :: x`	`char x;`	no
`integer(c_int) :: x`	`int x;`	no
`integer(c_short) :: x`	`short int x;`	no
`integer(c_long) :: x`	`long int x;`	no
`integer(c_long_long) :: x`	`long long int x;`	no
`integer(c_signed_char) :: x`	`char x; unsigned char x;`	no
`integer(c_size_t) :: x`	`size_t x;`	no
`integer(c_int8_t) :: x`	`int8_t x;`	no
`integer(c_int16_t) :: x`	`int16_t x;`	no
`integer(c_int32_t) :: x`	`int32_t x;`	no
`integer(c_int64_t) :: x`	`int64_t x;`	no
`integer(c_int128_t) :: x`	`int128_t x;`	yes

Table A.1: FORTRAN 90 vs C Declarations.

FORTRAN 90	C	Extension
`character(c_char) :: x`	`char x;`	no
`integer(c_int_least8_t) :: x`	`int_least8_t x;`	no
`integer(c_int_least16_t) :: x`	`int_least16_t x;`	no
`integer(c_int_least32_t) :: x`	`int_least32_t x;`	no
`integer(c_int_least64_t) :: x`	`int_least64_t x;`	no
`integer(c_int_least128_t) :: x`	`int_least128_t x;`	yes
`integer(c_int_fast8_t) :: x`	`int_fast8_t x;`	no
`integer(c_int_fast16_t) :: x`	`int_fast16_t x;`	no
`integer(c_int_fast32_t) :: x`	`int_fast32_t x;`	no
`integer(c_int_fast64_t) :: x`	`int_fast64_t x;`	no
`integer(c_int_fst128_t) :: x`	`int_fast128_t x;`	yes
`integer(c_intmax_t) :: x`	`intmax_t x;`	no
`integer(c_intptr_t) :: x`	`intptr_t x;`	no
`integer(c_ptrdiff_t) :: x`	`ptrdiff_t x;`	TS 29113
`real(c_float) :: x`	`float x;`	no
`real(c_double) :: x`	`double x;`	no
`real(c_long_double) :: x`	`long double x;`	no
`real(c_float128) :: x`	`_Float128 x;`	yes
`complex(c_float_complex) :: x`	`float _Complex x;`	no
`complex(c_double_complex) :: x`	`double _Complex x;`	no
`complex(c_long_double_complex) :: x`	`long double _Complex x;`	no
`complex(c_float128_complex) :: x`	`_Float128 _Complex x;`	yes
`logical(c_bool)`	`_Bool x;`	no

It should be noted that when dealing with arrays, C arrays are starting from element indexed as 0 to n-1, while FORTRAN arrays are indexed from 1 to n by default. FORTRAN arrays may be made of index 0 by declaring them as `name(0:n-1)` instead of `name(n)`. C stores arrays in row-major order, while FORTRAN stores them in column-major order. This means that if a two-dimensional array in C is subscripted as `name[ i ][ j ]`, the same array in FORTRAN would be subscribed as `name(j, i)`. Likewise, the dimensions of the array would be exchanged when declaring it in the two languages. In the following code example a FORTRAN 90 program is using C code, the hello function:

Table A.2: FORTRAN 90 Program using C code.

FORTRAN Code	C Code

Table A.2: FORTRAN 90 Program using C code.

FORTRAN Code

```
! -*- mode: f90-mode; -*-

! hello1-for.f90 file.


program hello1

  use, intrinsic :: iso_c_binding, only: c_int
  implicit none

  interface
     subroutine hello(count) bind(C)
       use, intrinsic :: iso_c_binding, only:
            c_int
       implicit none
       integer(c_int), value :: count
     end subroutine hello
  end interface

  integer(c_int) :: x
  x = 10
  call hello(x)
  stop

end program hello1

! End of hello1-for.f90 file.
```

C Code

```
/* -*- mode: c-mode; -*- */

/* hello1-c.c file. */
#include <stdio.h>

void hello(int count)
{
  printf("Hello,␣%d␣worlds.\n",  count);
}

/* End of hello1-c.c file. */
```

In the second example a C program is using FORTRAN 90 code:

Table A.3: C program using FORTRAN 90 code.

FORTRAN Code

```
/* -*- mode: c-mode; -*- */

/* hello2-c.c file. */
#include <stdio.h>
#include <stdlib.h>

void hello_(int);
int main(int, char *argv[]);

/* Main function. */
int main(int argc, char *argv[])
{
  hello_(10);
  exit(EXIT_SUCCESS);
}

/* End of hello2-c.c file. */
```

C Code

```
! -*- mode: f90-mode; -*-

! hello2-for.f90 file.

subroutine hello(count)
  use, intrinsic :: iso_c_binding, only: c_int
  implicit none
  integer(c_int), value :: count
  print "('Hello,␣',␣i3,␣'␣worlds!')", count
end subroutine hello

! End of hello2-for.f90 file.
```

A.2 Routines Naming.

The FORTRAN compiler appends an underscore character "_" to each user-defined common procedure or function. The purpose is to avoid conflicts with C functions and variables of the same name, most of the FORTRAN libraries are written in C. Unfortunately, it means the programmer must be careful when naming his or her procedure.

A.2.1 Naming C Routines to be Called from FORTRAN

In order for function[1] written in C to be callable from a FORTRAN 90 program one can use an *interface block*. All functions the reader have seen so far are internal functions that are contained in a program or a module. Functions that are not contained in any program or modules are *external functions*. A program can use internal functions, external functions and functions in modules. Moreover, external functions can be in the same file of the program or in several files. External functions can be considered as program units that are independent of each other. Thus, the only way of communication among external functions, the main program and modules is through arguments. In other words, from outside of an external function, it is impossible to use its variables, parameters and internal functions. Any external function to be used should be listed in an interface block along with the declaration of its arguments and their types and the type of the function value. Note that an external function can be in the file containing the main program or module. As long as that function is not contained in any program, function, or module, it is external and an interface block is required in any program, function or module where this function is used such as in listing A.2.

A.2.2 Naming FORTRAN Routines to be Called from C

At the same time when a C program needs a FORTRAN 90 function or procedure we can write something like the ones in listing A.3. In this case the calling convention is different. We specified a function prototype corresponding to `hello()` procedure from FORTRAN code with a "_" appended at the name. In the C program then this function will be called `hello_()`. Listing A.3 shows the calling convention for a C program that wants to use FORTRAN 90 code.

A.3 Returning Values from Functions.

A.3.1 Return Values from C Code.

In the listings A.1 and A.2 a FORTRAN 90 program calling a C function is presented.

Listing A.1: mean - FORTRAN code

```
! -*- mode: f90-mode; -*-

! mean-for.f90 file.


program meaning

  use, intrinsic :: iso_c_binding, only: c_size_t, c_double
  implicit none

  interface

    function mean(values, count) bind(C, name = "mean")
      use, intrinsic :: iso_c_binding, only: c_size_t, c_double
      implicit none
      real(c_double) :: mean
      integer(c_size_t), value :: count
      real(c_double) :: values(1,count)
    end function mean

```

[1] A C function returning void could be regarded as a FORTRAN procedure.

```
  end interface

  integer(c_size_t), parameter :: count = 10
  real(c_double) :: values(count)
  !
  values(:) = (/1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
     10.0/)
  print "('The␣mean␣is␣',␣f16.6)", mean(values, count)
  stop

end program meaning

! End of mean-for.f90 file.
```

Listing A.2: mean - C code.

```
/* -*- mode: c-mode; -*- */

/* mean-c.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

double mean(double values[], size_t count)
{
  double ret = 0.;
  size_t i;

  /* Computing the mean for "count" values. */
  for(i = 0; i < count; i++)
    ret += values[ i ];
  return (ret / (double) count);
}

/* End of mean-c.c file. */
```

A.3.2 Returning Values from FORTRAN 90 Code.

In these following listings: A.3 and A.4, a C program calls a FORTRAN 90 function computing the two norm of a vector.

Listing A.3: norm2 - C code.

```
/* -*- mode: c-mode; -*- */

/* File norm2-c.c. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

/* Functions prototypes. */
```

```
double norm2_(double [], size_t);

/* Main function. */
int main(int argc, char *argv[])
{
  double values[] = {
    1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
  };

  /* Computes the two norm of a vector. */
  printf("2␣norm:␣%lf\n", norm2_(values, 10));
  exit(EXIT_SUCCESS);
}
```

Listing A.4: norm2 - FORTRAN Code.

```
! -*- mode: f90-mode; -*-

! norm2-for.f90 file.

function norm2(values, count)
  use, intrinsic :: iso_c_binding, only: c_size_t, c_double
  implicit none
  real(c_double) :: norm2
  real(c_double) :: r
  integer(c_size_t), value :: count
  real(c_double) :: values(1 : count)
  integer :: i
  !
  do i = 1, count
    if (i .eq. 1) then
      norm2 = values(1) ** 2.0
    else
      norm2 = norm2 + values(i) ** 2.0
    end if
  end do
  norm2 = sqrt(norm2)
end function norm2

! End of norm2-for.f90 file.
```

A.4 Passing Arguments.

A.4.1 Passing Arguments to a C Function.

Listing A.5: fft - FORTRAN code.

```
! -*- mode: f90-mode; -*-

! fft-for.f90 file.

program ffting
```

```

  use, intrinsic :: iso_c_binding
  implicit none

  interface

    function dft(x, count) bind(C, name = "dft")
      use, intrinsic :: iso_c_binding
      implicit none
      logical(c_bool) :: dft
      integer(c_size_t), value :: count
      complex(c_double_complex) :: x(1 : count)
    end function dft

  end interface

  integer(c_size_t), parameter :: count = 10
  complex(c_double_complex) :: x(1 : count)
  integer :: i
  !
  x = (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 /)
  if (dft(x, count) .eqv. .true.) then
    do i = 1, count
      print *, x(i)
    end do
  else
    print *, "Error␣computing␣the␣fft."
  end if

end program ffting

! End of fft-for.f90 file.
```

Listing A.6: fft - C code.

```
/* -*- mode: c-mode; -*- */

/* fft-c.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <inttypes.h>
#include <math.h>
#include <complex.h>
#include <string.h>

bool dft(double complex x[], size_t count)
{
  bool ret = false;
  double complex *temp, wn;
  size_t j, k;

  if(count > 0) {
    if((temp = (double complex *) calloc(count, sizeof(double complex))) != NULL) {
      wn = cexp(-2.0 * M_PI * I / (double) count);
      for(j = 0; j < count; j++) {
        for(k = 0; k < count; k++) {
          temp[ j ] += x[ k ] * cpow(wn, (double) (j * k));
        }
      }
      memcpy(x, temp, sizeof(double complex) * count);
      free(temp);
      ret = true;
    }
  }
  return ret;
}

/* End of fft-c.c file.*/
```

A.4.2 Passing Arguments to a FORTRAN 90 procedure/function.

Listing A.7: ifft - FORTRAN code.

```
! -*- mode: f90-mode; -*-

function idft(x, count)
  use, intrinsic :: iso_c_binding
  implicit none
  logical(c_bool) :: idft
  integer(c_size_t), value :: count
  complex(c_double_complex), intent(out) :: x(1 : count)
  complex(c_double_complex) :: temp(1 : count), wn
  real(c_double), parameter :: pi = 2.0 * asin(1.0)
  integer :: j, k
  !
  idft = .false.
  if(count .gt. 0) then
    wn = exp(-2.0 * (0.0, 1.0) * pi / count)
    do j = 1, count
      temp(j) = (0.0, 0.0)
      do k = 1, count
        temp(j) = temp(j) + x(k) * wn ** ((j - 1) * (k - 1))
      end do
      temp(j) = temp(j) / count
    end do
    x(1 : count) = temp(1 : count)
    idft = .true.
  end if
end function idft

! End of ifft-for.f90 file.
```

Listing A.8: ifft - C code.

```c
/* -*- mode: c-mode; -*- */

/* ifft-c.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <inttypes.h>
#include <math.h>
#include <complex.h>
#include <string.h>

/* Functions prototypes. */
bool idft_(double complex [], size_t);
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  long int ret = EXIT_FAILURE;
  size_t i;
  double complex x[] = {
    0.0 + I * 0.0,
      1.0 + I * 0.0,
      2.0 + I * 0.0,
    2.0 + I * 0.0,
    1.0 + I * 0.0,
    0.0 + I * 0.0,
    -1.0 + I * 0.0,
    -2.0 + I * 0.0,
    -2.0 + I * 0.0,
    -1.0 + I * 0.0,
    0.0 + I * 0.0
  };

  if(idft_(x, 11) == true) {
    for(i = 0; i < 11; i++) {
      printf("%lf␣", creal(x[ i ]));
      if(cimag(x[ i ]) >= 0.0)
        printf("␣+␣");
      printf("%lf␣i\n", cimag(x[ i ]));
    }
    ret = EXIT_SUCCESS;
  }
}

/* End of ifft-c.c file. */
```

Appendix B

The Workstation Console Access.

wscons stands for *workstation console access*, the driver provides support for machine-independent access to the console. It is made of a number of cooperating modules, in particular:

- hardware support for display adapters, keyboards and mice: *wsdisplay*(4), *wskbd*(4) and *wsmouse*(4);
- input event multiplexor described in *wsmux*(4);
- terminal emulation modules;
- compatibility options to support control operations and other low-level behaviour of existing terminal drivers;

B.1 Terminal Emulations.

Terminal emulations wscons does not define its own set of terminal control sequences and special keyboard codes in terms of *termcap*(5). Instead, a *terminal emulation* is assigned to each virtual screen when the screen is created. Different terminal emulations can be active at the same time on one display. The following choices are available:

dumb this minimal terminal support is always available. No control sequences are supported besides the ASCII control characters. The cursor is not addressable. Only ASCII keyboard codes will be delivered, cursor and functions keys do not work;

sun The "sun" console emulation is available by default on the sparc64 architecture, or if option WSEMUL_SUN was specified at kernel build time. It supports the control sequences of SUN machine consoles and delivers its keyboard codes for function and keypad keys, as far as present on the actually used keyboard. ANSI colors are also supported on this emulation, if the TERM environment variable is set to rcons-color; This emulation is sufficient for full-screen applications;

vt100 is available by default, but can be disabled with option `WSEMUL_NO_VT100`. It provides the most commonly used functions of DEC VT100 terminals with some extensions introduced by the DEC VT220 and DEC VT320 models. The features of the original VT100 which are not, or not completely, implemented are:

- VT52 support, 132-column-mode, smooth scroll, light background, keyboard autorepeat control, external printer support, keyboard locking, newline/linefeed switching: Escape sequences related to these features are ignored or answered with standard replies. (DECANM, DECCOLM, DECSCLM, DECSCNM, DECARM, DECPFF, DECPEX, KAM, LNM);
- function keys are not reprogrammable and fonts cannot be downloaded. DECUDK and DECDLD sequences will be ignored;
- neither C1 control set characters will be recognized nor will 8-bit keyboard codes be delivered;
- the "DEC supplemental graphic" font is approximated by the ISO-latin-1 font, though there are subtle differences;
- the actual rendering quality depends on the underlying graphics hardware driver. Characters might be missing in the available fonts and be substituted by more or less fitting replacements. Depending on the keyboard used, not all function keys might be available.

In addition to the plain VT100 functions, the following features are supported:

- ANSI colors;
- some VT220-like presentation state settings and -reports (DECRSPS), especially tabulator settings.

In most applications, wscons will work sufficiently as a VT220 emulator.

The wscons infrastructure is the subdivided in four sub modules, these take care of: the display device, the keyboard device, the mouse device and the keyboard/mouse multiplexor.

B.2 Generic Display Device Support.

The *wsdisplay* driver is an abstraction layer for display devices within the *wscons*(4) framework. It attaches to the hardware specific display device driver and makes it available as text terminal or graphics interface. Display devices have the ability to display characters on them, without help of an X server, either directly by hardware or through software drawing pixel data into the display memory. The wsdisplay driver will connect a terminal emulation module and provide a tty-like software interface. The console locator in the configuration line refers to the device's use as output part of the operating system console. A device specification containing a positive value here will only match if the device is in use as system console. The console device selection in early system startup is not influenced. This way, the console device can be connected to a known wsdisplay device instance. The mux locator in the configuration line refers to the *wsmux*(4) that will be used to get keyboard events. If this locator is -1, no mux will be used. The logical unit of an independent contents displayed on a display, sometimes referred to as "virtual terminal", is called a *screen* here. If the underlying device driver supports it, multiple screens can be used on one display. As of this writing, only the *lcd*(4) and *vga*(4) display drivers provide this ability. Screens have different minor device numbers and separate tty instances. One screen possesses the *focus*, this means it is displayed on the display and its tty device will get the keyboard input. In some cases, if no screen is set up or if a screen was just deleted, it is possible that no focus is present at all. The focus can be switched by either special keyboard input, typically CTL- ALT-Fn, or an `ioctl` command issued by a user program. Screens are set up or deleted through the /dev/ttyCcfg control device, preferably using the *wsconscfg*(8) utility. In addition and with help from backend drivers the following features are also provided:

- loading, deleting and listing the loaded fonts;
- browsing backwards in the screen output, the size of the buffer for saved text is defined by the particular hardware driver;
- blanking the screen by timing out on inactivity in the screen holding the input focus. Awakening activities consist of:
  - pressing any keys on the keyboard;
  - moving or clicking the mouse;
  - any output to the screen.

  Blanking the screen is usually done by disabling the horizontal sync signal on video output, but may also include blanking the vertical sync in which case most monitors go into power saving mode. See wsconsctl(8) for controlling variables.

Consult the back-end drivers' documentation for which features are supported for each particular hardware type.

B.2.1 The `ioctl` Interface.

The following *ioctl*(2) calls are provided by the wsdisplay driver or by devices which use it. Their definitions are found in `<dev/wscons/wsconsio.h>`:

- `WSDISPLAYIO_GTYPE u_int` — retrieve the type of the display. The list of types is in `<dev/wscons/wsconsio.h>`;
- `WSDISPLAYIO_GINFO struct wsdisplay_fbinfo` — retrieve basic information about a framebuffer display. The returned structure is as follows;

  Listing B.1: The wsdisplay_fbinfo structure.

  ```
  struct wsdisplay_fbinfo {
    u_int height;
    u_int width;
    u_int depth;
    u_int cmsize;
  };
  ```

  The height and width members are counted in pixels. The depth member indicates the number of bits per pixel, and cmsize indicates the number of color map entries accessible through `WSDISPLAYIO_GETCMAP` and `WSDISPLAYIO_PUTCMAP`. This call is likely to be unavailable on text-only displays;
- `WSDISPLAYIO_GETSCREENTYPE struct wsdisplay_screentype` — retrieve basic information about a screen. The returned structure is as follows:

  Listing B.2: The wsdisplay_screentype structure.

  ```
  #define WSSCREEN_NAME_SIZE 16

  struct wsdisplay_screentype {
    int idx;
    int nidx;
  ```

```
  char name[ WSSCREEN_NAME_SIZE ];
  int ncols;
  int nrows;
  int fontwidth;
  int fontheight;
};
```

the idx member indicates the index of the screen. The nidx member indicates the number of screens. The name member contains a human readable string used to identify the screen. The ncols and nrows members indicate the available number of columns and rows. The fontwidth and fontheight members indicate the dimensions of a character cell, in pixels;

- WSDISPLAYIO_GETCMAP struct wsdisplay_cmap — retrieve the current color map from the display. This call needs the following structure set up beforehand:

Listing B.3: The wsdisplay_cmap structure.

```
struct wsdisplay_cmap {
  u_int index;
  u_int count;
  u_char *red;
  u_char *green;
  u_char *blue;
};
```

The index and count members specify the range of color map entries to retrieve. The red, green and blue members should each point to an array of count u_chars. On return, these will be filled in with the appropriate entries from the color map. On all displays that support this call, values range from 0 for minimum intensity to 255 for maximum intensity, even if the display does not use eight bits internally to represent intensity;

- WSDISPLAYIO_PUTCMAP struct wsdisplay_cmap — change the display's color map. The argument structure is the same as for WSDISPLAYIO_GETCMAP, but red, green and blue are taken as pointers to the values to use to set the color map. This call is not available on displays with fixed color maps;

- WSDISPLAYIO_GVIDEO u_int — get the current state of the display's video output. Possible values are:

  - WSDISPLAYIO_VIDEO_OFF — the display is blanked;
  - WSDISPLAYIO_VIDEO_ON — the display is enabled;

- WSDISPLAYIO_SVIDEO u_int — set the state of the display's video output. See WSDISPLAYIO_GVIDEO above for possible values.

- WSDISPLAYIO_GCURPOS struct wsdisplay_curpos — retrieve the current position of the hardware cursor. The returned structure is as follows:

Listing B.4: The wsdisplay_curpos structure.

```
struct wsdisplay_curpos {
  u_int x;
  u_int y;
};
```

The x and y members count the number of pixels right and down, respectively, from the top-left corner of the display to the hot spot of the cursor. This call is not available on displays without a hardware cursor.

- WSDISPLAYIO_SCURPOS struct wsdisplay_curpos — set the current cursor position. The argument structure, and its semantics, are the same as for WSDISPLAYIO_GCURPOS. This call is not available on displays without a hardware cursor.

- WSDISPLAYIO_GCURMAX struct wsdisplay_curpos — retrieve the maximum size of cursor supported by the display. The x and y members of the returned structure indicate the maximum number of pixel rows and columns, respectively, in a hardware cursor on this display. This call is not available on displays without a hardware cursor.

- WSDISPLAYIO_GCURSOR struct wsdisplay_cursor — retrieve some or all of the hardware cursor's attributes. The argument structure is as follows:

  Listing B.5: The wsdisplay_cursor struct.

  ```
  struct wsdisplay_cursor {
    u_int which;
    u_int enable;
    struct wsdisplay_curpos pos;
    struct wsdisplay_curpos hot;
    struct wsdisplay_cmap cmap;
    struct wsdisplay_curpos size;
    u_char *image;
    u_char *mask;
  };
  ```

  The which member indicates which of the values the application requires to be returned. It should contain the logical OR of the following flags:

  – WSDISPLAY_CURSOR_DOCUR — get enable, which indicates whether the cursor is currently displayed (non-zero) or not (zero);

  – WSDISPLAY_CURSOR_DOPOS — get pos, which indicates the current position of the cursor on the display, as would be returned by WSDISPLAYIO_GCURPOS;

  – WSDISPLAY_CURSOR_DOHOT — get hot, which indicates the location of the "hot spot" within the cursor. This is the point on the cursor whose position on the display is treated as being the position of the cursor by other calls. Its location is counted in pixels from the top-left corner of the cursor;

  – WSDISPLAY_CURSOR_DOCMAP — get cmap, which indicates the current cursor color map. Unlike in a call to WSDISPLAYIO_GETCMAP, cmap here need not have its index and count members initialized. They will be set to 0 and 2 respectively by the call. This means that cmap.red, cmap.green, and cmap.blue must each point to at least enough space to hold two u_chars;

  – WSDISPLAY_CURSOR_DOSHAPE — get size, image, and mask. These are, respectively, the dimensions of the cursor in pixels, the bitmap of set pixels in the cursor and the bitmap of opaque pixels in the cursor. The format in which these bitmaps are returned, and hence the amount of space that must be provided by the application, are device-dependent;

  – WSDISPLAY_CURSOR_DOALL — get all of the above.

  The device may elect to return information that was not requested by the user, so those elements of struct wsdisplay_cursor which are pointers should be initialized to NULL if not otherwise used. This call is not available on displays without a hardware cursor.

- WSDISPLAYIO_SCURSOR struct wsdisplay_cursor — set some or all of the hardware cursor's attributes. The argument structure is the same as for WSDISPLAYIO_GCURSOR. The which member specifies which attributes of the cursor are to be changed. It should contain the logical OR of the following flags:
  - WSDISPLAY_CURSOR_DOCUR — if enable is zero, hide the cursor. Otherwise, display it;
  - WSDISPLAY_CURSOR_DOPOS — set the cursor's position on the display to pos, the same as WSDISPLAYIO_SCURPOS;
  - WSDISPLAY_CURSOR_DOHOT — set the *hot spot* of the cursor, as defined above, to hot;
  - WSDISPLAY_CURSOR_DOCMAP — set some or all of the cursor color map based on cmap. The index and count elements of cmap indicate which color map entries to set, and the entries themselves come from cmap.red, cmap.green, and cmap.blue;
  - WSDISPLAY_CURSOR_DOSHAPE — set the cursor shape from size, image, mask. See above for their meanings;
  - WSDISPLAY_CURSOR_DOALL — do all of the above.

  This call is not available on displays without a hardware cursor.

- WSDISPLAYIO_GMODE u_int — get the current mode of the display. Possible results include:
  - WSDISPLAYIO_MODE_EMUL — the display is in emulating text mode;
  - WSDISPLAYIO_MODE_MAPPED — the display is in mapped graphics mode;
  - WSDISPLAYIO_MODE_DUMBFB — the display is in mapped frame buffer mode.

- WSDISPLAYIO_SMODE u_int — set the current mode of the display. For possible arguments, see WSDISPLAYIO_GMODE.

- WSDISPLAYIO_LDFONT struct wsdisplay_font — loads a font specified by the wsdisplay_font structure:

  Listing B.6: The wsdisplay_font structure.

```
#define WSFONT_NAME_SIZE 32

struct wsdisplay_font {
  char name[ WSFONT_NAME_SIZE ];
  int index;
  int firstchar
  int numchars;
  int encoding;
  u_int fontwidth;
  u_int fontheight;
  u_int stride;
  int bitorder;
  int byteorder;
  void *cookie;
  void *data;
};
```

  The name member contains a human readable string used to identify the font. The index member may be used to select a driver-specific font resource, for non-raster frame buffers. A value of -1 will pick the first available slot. The firstchar member contains the index of the first character in the font, starting at zero. The numchars member contains the number of characters in the font. The encoding member describes the font character encoding, using one of the following values:

  - WSDISPLAY_FONTEC__ISO — ISO-8859-1 encoding, also known as Latin-1. This is the preferred encoding for raster frame buffers;
  - WSDISPLAY_FONTENC_IBM — IBM code page number 437. This is the preferred encoding for text-mode displays.

  The fontwidth and fontheight members specify the dimensions of a character cell. The stride member specify the number of bytes of font data per character cell line, usually fontwidth rounded up to a byte boundary. The bitorder and byteorder members specify the bit- and byte-ordering of the font data, using either one of the following values:

  - WSDISPLAY_FONTORDER_L2R — leftmost data contained in the most significant bits, left- to-right ordering. This is the most commonly encountered case;
  - WSDISPLAY_FONTORDER_R2L — leftmost data contained in the least significant bits, right- to-left ordering;

  The data field contains the font character data to be loaded. The cookie field is reserved for internal purposes.

- WSDISPLAYIO_LSFONT struct wsdisplay_font — retrieves the data for a loaded font into the wsdisplay_font structure. The index field is set to the font resource to query. For the argument structure, see WSDISPLAYIO_LDFONT;

- WSDISPLAYIO_USEFONT struct wsdisplay_font — selects the font specified in the name field. An empty name selects the next available font. For the argument structure, see WSDISPLAYIO_LDFONT;

- WSDISPLAYIO_GBURNER struct wsdisplay_burner — retrieves the state of the screen burner. The returned structure is as follows:

  Listing B.7: The wsdisplay_burner structure.

  ```
  struct wsdisplay_burner {
    u_int off;
    u_int on;
    u_int flags;
  };
  ```

  The off member contains the inactivity time before the screen is turned off, in milliseconds. The on member contains the time before the screen is turned back on, in milliseconds. The flags member contains a logical OR of the following flags:

  - WSDISPLAY_BURN_VBLANK — when turning the display off, disable the vertical synchronization signal;
  - WSDISPLAY_BURN_KBD — monitor keyboard activity;
  - WSDISPLAY_BURN_MOUSE — monitor mouse activity, this only works for mice using the *wsmouse*(4) driver;
  - WSDISPLAY_BURN_OUTPUT — monitor display output activity.

  If none of the activity source flags are set, the screen burner is disabled.

- WSDISPLAYIO_SBURNER struct wsdisplay_burner — sets the state of the screen burner. The argument structure, and its semantics, are the same as for WSDISPLAYIO_GBURNER.

- WSDISPLAYIO_ADDSCREEN struct wsdisplay_addscreendata — creates a new screen:

Listing B.8: The wsdisplay_addscreendata structure.

```
#define WSEMUL_NAME_SIZE 16

struct wsdisplay_addscreendata {
  int idx;
  char screentype[ WSSCREEN_NAME_SIZE ];
  char emul[ WSEMUL_NAME_SIZE ];
};
```

The `idx` member is the index of the screen to be configured. The `screentype` member is matched against builtin screen types, which will be driver-dependent. The `emul` member indicates the terminal emulation type. Available terminal emulations are:

- sun — Sun terminal emulation. This is the default on the sparc64 architecture;
- vt100 — Dec VT100 terminal emulation, with some VT220 features. This is the default on all other architectures;
- dumb — dumb terminal.

An empty string will select the default emulation;

- `WSDISPLAYIO_DELSCREEN struct wsdisplay_delscreendata` — deletes a screen:

Listing B.9: The wsdisplay_delscreendata structure.

```
struct wsdisplay_delscreendata {
  int idx;
  int flags;
};
```

The `idx` member indicates the index of the screen to be deleted. The `flags` member is a logical OR of zero or more of the following:

- `WSDISPLAY_DELSCR_FORCE` — force deletion of screen even if in use by a userspace program;
- `WSDISPLAY_DELSCR_QUIET` — don't report deletion to console.

- `WSDISPLAYIO_GETSCREEN struct wsdisplay_addscreendata` — returns information on the screen indicated by `idx` or the current screen if `idx` is -1. The screen and emulation types are returned in the same structure, see `WSDISPLAYIO_GETPARAM`;
- `WSDISPLAYIO_SETSCREEN u_int` — switch to the screen with the given index;
- `WSDISPLAYIO_WSMOUSED struct wscons_event` — this call is used by the *wsmoused*(8) daemon to inject mouse events gathered from serial mice, as well as various control events;
- `WSDISPLAYIO_GETPARAM struct wsdisplay_param` — retrieves the state of a display parameter. This call needs the following structure set up beforehand:

Listing B.10: The wsdisplay_param structure.

```
struct wsdisplay_param {
  int param;
  int min;
  int max;
  int curval;
  int reserved[ 4 ];
};
```

The param member should be set with the parameter to be returned. The following parameters are supported:

- WSDISPLAYIO_PARAM_BACKLIGHT — the intensity of the display backlight, usually on laptop computers;
- WSDISPLAYIO_PARAM_BRIGHTNESS — the brightness level;
- WSDISPLAYIO_PARAM_CONTRAST — the contrast level.

On return, min and max specify the allowed range for the value, while curval specifies the current setting. Not all parameters are supported by all display drivers.

- WSDISPLAYIO_SETPARAM struct wsdisplay_param — sets a display parameter. The argument structure is the same as for WSDISPLAYIO_GETPARAM, with the param and curval members filled in. Not all parameters are supported by all display drivers.
- WSDISPLAYIO_LINEBYTES u_int — get the number of bytes per row when the device is in WSDISPLAYIO_MODE_DUMBFB mode.

The following code shows the usage of some of the ioctl calls listed above:

Listing B.11: wsdisplay - program to show WSDISPLAYIO ioctl calls.

```c
/* -*- mode: c-mode; -*- */

/* wsdisplay.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <dev/wscons/wsconsio.h>

/* program wsdisplay.c */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd;
  long int ret = EXIT_FAILURE;
  u_int gtype;

  /* Check arguments count. */
  if(argc == 2) {
    fd = open(argv[ 1 ], O_RDONLY | O_EXCL, 0666);
    if(fd >= 0) {
      if(ioctl(fd, WSDISPLAYIO_GTYPE, &gtype) >= 0) {
        printf("type␣of␣display␣for␣%s:␣%d\n", argv[ 1 ], gtype);
        ret = EXIT_SUCCESS;
      } else
        perror("ioctl");
      close(fd);
    } else
      perror("open");
  } else
    fprintf(stderr, "usage:␣wsdisplay␣<device>\n");
  exit(ret);
}

/* End of wsdisplay.c file. */
```

B.3 Generic Keyboard Device Support.

The wskbd driver handles common tasks for keyboards within the *wscons*(4) framework. It is attached to the hardware specific keyboard drivers and provides their connection to *wsdisplay* devices and a character device interface. The common keyboard support consists of:

- mapping from keycodes, defined by the specific keyboard driver, to keysyms, hardware independent, defined in <dev/wscons/wsksymdef.h>;
- handling of *compose* sequences. Characters commonly not present as separate keys on keyboards can be generated after either a special *compose* key is pressed or a *dead accent* character is used;
- certain translations, like turning an *ALT* modifier into an *ESC* prefix;
- automatic key repetition, *typematic*;
- parameter handling for *keyboard bells*;
- generation of *keyboard events* for use by X servers.

The wskbd driver provides a number of `ioctl` functions to control key maps and other parameters. These functions are accessible through the associated wsdisplay device as well. A complete list is in <dev/wscons/wsconsio.h>. The console locator in the configuration line refers to the device's use as input part of the operating system console. The wskbd driver traps certain key sequences intended to perform special functions. The Ctrl+Alt+Esc sequence will initiate the *ddb*(4) kernel debugger if the *ddb.console sysctl*(8) variable is set.

B.3.1 The `ioctl` Interface.

As in the WSDISPLAY driver, wskbd driver provides a number of ioctl calls. These calls are defined in
<dev/wscons/wsconsio.h>, they are:

- WSKBDIO_BELL — plays the wscons bell;
- WSKBDIO_COMPLEXBELL struct wskbd_bell_data — it uses the struct wskbd_bell_data to play the bell. The structure is defined in <dev/wscons/wsconsio.h>:

Listing B.12: The wskbd_bell_data structure.

```
struct wskbd_bell_data {
  u_int which;
  u_int pitch;
  u_int period;
  u_int volume;
};
```

The which member could be one of:

  - WSKBD_BELL_DOPITCH — to get or set the bell pitch;
  - WSKBD_BELL_DOPERIOD — to get or set the bell period;
  - WSKBD_BELL_DOVOLUME — to get or set the bell volume;
  - WSKBD_BELL_DOALL — to set all the parameters at once.

The pitch member is the frequency in Hz of the sound to be emitted. The period member is the value in milliseconds of the durantion of the bell sound and the volume member is the percentage of the maximum value for the bell volume. In listing B.17 we manipulate the wskbd_bell_data structure to modify the bell sound;

- WSKBDIO_SETBELL struct wskbd_bell_data — this call set the parameters for the bell as stated in the WSKBDIO_COMPLEXBELL call;
- WSKBDIO_GETBELL struct wskbd_bell_data — retrieve the parameters for the bell specified in the wskbd_bell_data structure as stated in WSKBDIO_COMPLEXBELL;
- WSKBDIO_SETDEFAULTBELL struct wskbd_bell_data — as the previous ioctl call but related to the default bell;
- WSKBDIO_GETDEFAULTBELL struct wskbd_bell_data — as the previous ioctl call but related to the default bell;
- WSKBDIO_SETKEYREPEAT struct wskbd_keyrepeat_data — set keyboard autorepeat settings. The structure wskbd_keyrepeat_data is defined in <dev/wscons/wsconsio.h>:

  Listing B.13: The wskbd_keyrepeat_data.

  ```
  struct wskbd_keyrepeat_data {
    u_int which;
    u_int del1;
    u_int delN;
  };
  ```

  The which member could be one of:

  - WSKBD_KEYREPEAT_DODEL1 — get or set del1 member;
  - WSKBD_KEYREPEAT_DODELN — get or set delN member;
  - WSKBD_KEYREPEAT_DOALL — all of the above.

  To get or set the corresponding struct member. The del1 member represents the delay before the first pressure or a key in milliseconds and the last member delN sets the delay before rest in milliseconds;
- WSKBDIO_GETKEYREPEAT struct wskbd_keyrepeat_data — get key repeat data as specified in WSKBDIO_SETKEYREPEAT;

- WSKBDIO_SETDEFAULTKEYREPEAT struct wskbd_keyrepeat_data — set key repeat data as specified in WSKBDIO_SETKEYREPEAT for the default keyboard;

- WSKBDIO_GETDEFAULTKEYREPEAT struct wskbd_keyrepeat_data — get key repeat data as specified in WSKBDIO_SETKEYREPEAT for the default keyboard;

- WSKBDIO_SETLEDS int — set the status for the keyboard leds as per the following constants:

  - WSKBD_LED_CAPS;
  - WSKBD_LED_NUM;
  - WSKBD_LED_SCROLL;
  - WSKBD_LED_COMPOSE.

  Listing B.18 shows the usage of these calls.

- WSKBDIO_GETLEDS u_int — get leds status as specified in WSKBDIO_SETLEDS;

- WSKBDIO_GETMAP struct wskbd_map_data — get the keyboard mapping settings. They are stored in a structure of type wskbd_map_data:

  Listing B.14: The wskbd_map_data structure.

  ```
  struct wskbd_map_data {
    u_int maplen;
    struct wscons_keymap *map;
  };
  ```

  The maplen member is the number of entries in the map, the maximum count is defined in the WSKBDIO_MAXMAPLEN constant. The map member is a pointer to the array of struct wscons_keymap defined in <dev/wscons/wsksymvar.h>:

  Listing B.15: The wscons_keymap structure.

  ```
  typedef u_int16_t keysym_t;

  struct wscons_keymap {
    keysym_t command;
    keysym_t group1[ 2 ];
    keysym_t group2[ 2 ];
  };
  ```

- WSKBDIO_SETMAP struct wskbd_map_data — set a new keymap for the keyboard, the data is provided by wskbd_map_data structure described in WSKBDIO_GETMAP;

- WSKBDIO_GETENCODING kbd_t — encodings are defined in <dev/wscons/wksymdef.h> file. See listing B.19 for an example of reading and setting encoding;

- WSKBDIO_SETENCODING kbd_t — set the encoding for the keyboard. See WSKBDIO_GETENCODING;

- WSKBDIO_GETBACKLIGHT struct wskbd_backlight — get configurations for the backlight on keyboards that support that features. The structure wskbd_backlight is defined in <dev/wscons/wsconsio.h>:

Listing B.16: The wskbd_backlight structure.

```
struct wskbd_backlight {
  unsigned int min;
  unsigned int max;
  unsigned int curval;
};
```

The min and max members set the minimum and maximum intensity for the backlight. The curval specifies the current value;

- WSKBDIO_SETBACKLIGHT struct wskbd_backlight — set the backlight configuration for keyboards that support this feature. See WSKBDIO_GETBACKLIGHT;
- WSKBDIO_SETMODE u_int — sets the mode for the keyboard. Possible values are:
  - WSKBD_TRANSLATED — keys are translated throught the keyboard map. See WSKBDIO_SETMAP;
  - WSKBD_RAW — keys are not filtered throught the keymap.
- WSKBDIO_GETMODE u_int — get the current translating mode for the keyboard. See WSKBDIO_SETMODE

Listing B.17: wskbd - program to shows the usage of the wskbd driver.

```
/* -*- mode: c-mode; -*- */

/* wskbd.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <dev/wscons/wsconsio.h>

#define FOREVER for(;;)

/* program wskbd.c */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd;
  long int ret = EXIT_FAILURE;
  struct wskbd_bell_data o_wsbelldata, n_wsbelldata;

  /* Check arguments count. */
  if(argc == 2) {
    fd = open(argv[ 1 ], O_RDWR | O_EXCL);
    if(fd >= 0) {
      printf("opened␣%s.\n", argv[ 1 ]);
      if(ioctl(fd, WSKBDIO_BELL) >= 0) {
        if(ioctl(fd, WSKBDIO_GETBELL, &o_wsbelldata) >= 0) {
          printf("old␣GETBELL:␣%d␣%d␣%d␣%d\t",  \
            o_wsbelldata.which, \
            o_wsbelldata.pitch, \
            o_wsbelldata.period,  \
            o_wsbelldata.volume);
          bzero(&n_wsbelldata, sizeof(struct wskbd_bell_data));
          n_wsbelldata.which = WSKBD_BELL_DOPITCH;
          n_wsbelldata.pitch = o_wsbelldata.pitch * 2;
          if(ioctl(fd, WSKBDIO_SETBELL, &n_wsbelldata) >= 0) {
            printf("new␣GETBELL:␣%d␣%d␣%d␣%d\t",  \
              n_wsbelldata.which, \
              n_wsbelldata.pitch, \
              n_wsbelldata.period,  \
              n_wsbelldata.volume);
            if(ioctl(fd, WSKBDIO_BELL) >= 0) {
              sleep(5);
              if(ioctl(fd, WSKBDIO_SETBELL, &o_wsbelldata) >= 0) {
                printf("restore␣old␣GETBELL:␣%d␣%d␣%d␣%d\n", \
                  o_wsbelldata.which, \
                  o_wsbelldata.pitch, \
                  o_wsbelldata.period,  \
                  o_wsbelldata.volume);
                if(ioctl(fd, WSKBDIO_BELL) >= 0) {
                  sleep(5);
                  ret = EXIT_SUCCESS;
                } else
                  perror("error␣playing␣bell");
              } else
                perror("error␣setting␣bell␣data");
            } else
              perror("error␣playing␣bell");
          } else
            perror("error␣setting␣bell␣data");
        } else
          perror("error␣retrieving␣bell␣data");
      } else
        perror("error␣playing␣bell");
      close(fd);
    } else
      perror("open");
  } else
    fprintf(stderr, "usage:␣wskbd␣<device>\n");
  exit(ret);
```

```
}

/* End of wskbd.c file. */
```

Listing B.18: wskbd-leds - program to shows the usage of the ioctl calls for leds.

```
/* -*- mode: c-mode; -*- */

/* wskbd-leds.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <dev/wscons/wsconsio.h>

#define FOREVER for(;;)

/* program wskbd-leds.c */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd;
  long int ret = EXIT_FAILURE;
  u_int i, o_gleds;

  /* Check arguments count. */
  if(argc == 2) {
    fd = open(argv[ 1 ], O_RDWR | O_EXCL);
    if(fd >= 0) {
      printf("opened␣%s.\n", argv[ 1 ]);
      if(ioctl(fd, WSKBDIO_GETLEDS, &o_gleds) >= 0) {
        for(i = 1; i <= WSKBD_LED_COMPOSE; i = (i << 1)) {
          printf("led␣=␣%d\n", i);
          ioctl(fd, WSKBDIO_SETLEDS, &i);
          sleep(2);
        }
        ioctl(fd, WSKBDIO_SETLEDS, &o_gleds);
      } else
        perror("error␣getting␣leds");
      close(fd);
    } else
      perror("open");
  } else
```

```
    fprintf(stderr, "usage:␣wskbd-leds␣<device>\n");
  exit(ret);
}

/* End of wskbd-leds.c file. */
```

Listing B.19: wskbd-enc - program to shows the configured encoding for the keyboard.

```
/* -*- mode: c-mode; -*- */

/* wskbd-enc.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <dev/wscons/wsconsio.h>
#include <dev/wscons/wsksymdef.h>
#include <dev/wscons/wsksymvar.h>

#define FOREVER for(;;)

/* program wskbd-enc.c */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd;
  long int ret = EXIT_FAILURE;
  kbd_t o_genc;

  /* Check arguments count. */
  if(argc == 2) {
    fd = open(argv[ 1 ], O_RDWR | O_EXCL);
    if(fd >= 0) {
      printf("opened␣%s.\n", argv[ 1 ]);
      if(ioctl(fd, WSKBDIO_GETENCODING, &o_genc) >= 0) {
        printf("GETENCODING:␣0x%0.8x\n", KB_ENCODING(o_genc));
      } else
        perror("error␣getting␣encoding");
      close(fd);
    } else
      perror("open");
  } else
```

```
    fprintf(stderr, "usage: wskbd-enc <device>\n");
  exit(ret);
}

/* End of wskbd-enc.c file. */
```

B.4 Generic Mouse Support.

The wsmouse driver is an abstraction layer for mice within the *wscons*(4) framework. It is attached to the hardware specific mouse drivers and provides a character device interface which returns struct wscons_event via *read*(2). For use with X servers, “mouse events” can be generated.

B.4.1 The `ioctl` interface.

The following *ioctl*(2) calls are provided by the wsmouse driver or by devices which use it. Their definitions are found in `<dev/wscons/wsconsio.h>`:

- `WSMOUSEIO_GTYPE u_int` — get the mouse type. Mice types are the following:
  - `WSMOUSE_TYPE_VSXXX` — DEC serial;
  - `WSMOUSE_TYPE_PS2 2` — PS/2-compatible;
  - `WSMOUSE_TYPE_USB` — USB mouse;
  - `WSMOUSE_TYPE_LMS` — Logitech busmouse;
  - `WSMOUSE_TYPE_MMS` — Microsoft InPort mouse;
  - `WSMOUSE_TYPE_TPANEL` — Generic Touch Panel;
  - `WSMOUSE_TYPE_NEXT` — NeXT mouse;
  - `WSMOUSE_TYPE_ARCHIMEDES` — Archimedes mouse;
  - `WSMOUSE_TYPE_ADB` — ADB;
  - `WSMOUSE_TYPE_HIL` — HP HIL;
  - `WSMOUSE_TYPE_LUNA` — OMRON Luna;
  - `WSMOUSE_TYPE_DOMAIN` — Apollo Domain;
  - `WSMOUSE_TYPE_BLUETOOTH` — Bluetooth mouse;
  - `WSMOUSE_TYPE_SUN` — SUN serial mouse;
  - `WSMOUSE_TYPE_SYNAPTICS` — Synaptics touchpad;
  - `WSMOUSE_TYPE_ALPS` — ALPS touchpad;
  - `WSMOUSE_TYPE_SGI` — SGI serial mouse;
  - `WSMOUSE_TYPE_ELANTECH` — Elantech touchpad;
  - `WSMOUSE_TYPE_SYNAP_SBTN` — Synaptics soft buttons;
  - `WSMOUSE_TYPE_TOUCHPAD` — Generic touchpad.
- `WSMOUSEIO_SRES u_int` — set the resolution for the mouse;
- `WSMOUSEIO_SCALIBCOORDS struct wsmouse_calibcoords` —
- `WSMOUSEIO_GCALIBCOORDS struct wsmouse_calibcoords` — get calibration coordinates constants from mouse device. The `struct wsmouse_calibcoords` is defined in `<dev/wscons/wsconsio.h>`:

Listing B.20: The wsmouse_calibcoords structure.

```
#define WSMOUSE_CALIBCOORDS_MAX 16

struct wsmouse_calibcoords {
  int minx;
  int miny;
  int maxx;
  int maxy;
  int swapxy;
  int resx;
  int resy;
  int samplelen;
  struct wsmouse_calibcoord {
    int rawx;
    int rawy;
    int x;
    int y;
  } samples[ WSMOUSE_CALIBCOORDS_MAX ];
};
```

The minx and miny members are the minimum values for x and y axes respectively. maxx and maxy members are the maximum values for x and y axes respectively. swapxy member is a flag which indicates the swap of x axis with the y one. resx and resy members are the x and y axes resolution respectively. samplelen member is the number of samples available or WSMOUSE_CALIBCOORDS_RESET for raw mode. The samples member is an array of WSMOUSE_CALIBCOORDS_MAX elements for the rawx and rawy members raw coordinates, instead, x and y members are the translated coordinates;

- WSMOUSEIO_SETMODE int — set mode for the mouse, allowed values are:
  - WSMOUSE_COMPAT — compatibility mode;
  - WSMOUSE_NATIVE — native mode.
- WSMOUSEIO_GETPARAMS struct wsmouse_parameters — get mouse parameters. The struct wsmouse_parameters is defined in <dev/wscons/wsconsio.h>:

Listing B.21: The wsmouse_parameters structure.

```
struct wsmouse_parameters {
  struct wsmouse_param *params;
  u_int nparams;
};
```

The params member is a pointer to the array of type struct wsmouse_param:

Listing B.22: The wsmouse_param structure.

```
struct wsmouse_param {
  enum wsmousecfg key;
  int value;
};
```

In this structure, the key member is a constant which can assume the following values:

- WSMOUSECFG_DX_SCALE — x-scale factor in [*.12] fixed-point format;

- WSMOUSECFG_DY_SCALE — y-scale factor in [*.12] fixed-point format;
- WSMOUSECFG_PRESSURE_LO — pressure limits defining start of touch;
- WSMOUSECFG_PRESSURE_HI — pressure limits defining end of touch;
- WSMOUSECFG_TRKMAXDIST — max distance to pair points for MT contact;
- WSMOUSECFG_SWAPXY — swap x- and y-axis;
- WSMOUSECFG_X_INV — map absolute coordinate x to (inv - x);
- WSMOUSECFG_Y_INV — map absolute coordinate y to (inv - y);
- WSMOUSECFG_REVERSE_SCROLLING — reverse scroll directions;
- WSMOUSECFG__FILTERS — coordinate handling, applying only in WSMOUSE_COMPAT mode;
- WSMOUSECFG_DX_MAX — ignore x deltas greater than this limit
- WSMOUSECFG_DY_MAX — ignore y deltas greater than this limit;
- WSMOUSECFG_X_HYSTERESIS — retard value for x coordinates;
- WSMOUSECFG_Y_HYSTERESIS — retard value for y coordinates;
- WSMOUSECFG_DECELERATION — threshold, distance, for deceleration;
- WSMOUSECFG_STRONG_HYSTERESIS — false and read-only, the feature is not supported anymore;
- WSMOUSECFG_SMOOTHING — smoothing factor (0-7);
- WSMOUSECFG__TPFILTERS — touchpad features;
- WSMOUSECFG_SOFTBUTTONS — 2 soft-buttons at the bottom edge;
- WSMOUSECFG_SOFTMBTN — add a middle-button area;
- WSMOUSECFG_TOPBUTTONS — 3 soft-buttons at the top edge;
- WSMOUSECFG_TWOFINGERSCROLL — enable two-finger scrolling;
- WSMOUSECFG_EDGESCROLL — enable edge scrolling;
- WSMOUSECFG_HORIZSCROLL — enable horizontal edge scrolling;
- WSMOUSECFG_SWAPSIDES — invert soft-button/scroll areas;
- WSMOUSECFG_DISABLE — disable all output except for clicks in the top-button area;
- WSMOUSECFG_MTBUTTONS — multi-touch buttons;
- WSMOUSECFG__TPFEATURES —
- WSMOUSECFG_LEFT_EDGE — ratio: left edge / total width;
- WSMOUSECFG_RIGHT_EDGE — ratio: right edge / total width;
- WSMOUSECFG_TOP_EDGE — ratio: top edge / total height;
- WSMOUSECFG_BOTTOM_EDGE — ratio: bottom edge / total height;
- WSMOUSECFG_CENTERWIDTH — ratio: center width / total width;
- WSMOUSECFG_HORIZSCROLLDIST — distance mapped to a scroll event;
- WSMOUSECFG_VERTSCROLLDIST — distance mapped to a scroll event;
- WSMOUSECFG_F2WIDTH — width limit for single touches;
- WSMOUSECFG_F2PRESSURE — pressure limit for single touches;
- WSMOUSECFG_TAP_MAXTIME — max. duration of tap contacts in milliseconds;
- WSMOUSECFG_TAP_CLICKTIME — time between the end of a tap and the button-up-event in milliseconds;

  - WSMOUSECFG_TAP_LOCKTIME — time between a tap-and-drag action and the button-up-event in milliseconds;
  - WSMOUSECFG_TAP_ONE_BTNMAP — one-finger tap button mapping;
  - WSMOUSECFG_TAP_TWO_BTNMAP — two-finger tap button mapping;
  - WSMOUSECFG_TAP_THREE_BTNMAP — three-finger tap button mapping;
  - WSMOUSECFG_MTBTN_MAXDIST — MTBUTTONS: distance limit for two-finger clicks;
  - WSMOUSECFG__TPSETUP — enable/disable debug output;
  - WSMOUSECFG_LOG_INPUT —
  - WSMOUSECFG_LOG_EVENTS —
  - WSMOUSECFG__DEBUG —

  The second member, value, is the value for the corresponding parameter.

  For the wsmouse_parameters structure, the nparams member is the count of the elements in the array pointed by params;

- WSMOUSEIO_SETPARAMS struct wsmouse_parameters — obtains and sets various mouse parameters as a key/value set. Currently these primarily relate to touchpads. The structure struct wsmouse_parameters is defined as in WSMOUSEIO_GETPARAMS. The number of parameters to read or write must be specified in nparams. For each parameter, when WSMOUSEIO_GETPARAMS is used, a key must be specified. When WSMOUSEIO_SETPARAMS is used, a key and a value must be specified. A single ioctl may retrieve up to WSMOUSECFG_MAX nparams.

B.5 The Console Keyboard/Mouse Multiplexor.

The wsmux is a pseudo-device driver that allows several *wscons*(4) input devices to have their events multiplexed into one stream. The typical usage for this device is to have two multiplexors, one for mouse events and one for keyboard and bell events. All *wsmouse*(4) devices should direct their events to the mouse mux, normally 0 and all keyboard devices, except the console, should direct their events to the keyboard mux, normally 1. A device will send its events to the mux indicated by the mux locator. If none is given the device will not use a multiplexor. The keyboard multiplexor should be connected to the display, using the *wsconscfg*(8) command. It will then receive all keystrokes from all keyboards and, furthermore, keyboards can be dynamically attached and detached without further user interaction. In a similar way, the window system will open the mouse multiplexor and receive all mouse events; mice can also be dynamically attached and detached. If a wskbd(4) or wsmouse(4) device is opened despite having a mux it will be detached from the mux. It is also possible to inject events into a multiplexor from a user program.

B.5.1 The ioctl interface.

The following ioctl(2) calls are available for th wsmux device defined in <dev/wscons/wsconsio.h>:

- WSMUXIO_INJECTEVENT struct wscons_event — injects a wscons_event in the queue. The structure is defined in <dev/wscons/wsconsio.h>:

Listing B.23: The wscons_event structure.

```
struct wscons_event {
  u_int type;
  int value;
  struct timespec time;
};
```

The type member could be one of:

- WSCONS_EVENT_KEY_UP — key code;
- WSCONS_EVENT_KEY_DOWN — key code;
- WSCONS_EVENT_ALL_KEYS_UP — void;
- WSCONS_EVENT_MOUSE_UP — button # leftmost = 0;
- WSCONS_EVENT_MOUSE_DOWN — button # leftmost = 0;
- WSCONS_EVENT_MOUSE_DELTA_X — x delta amount;
- WSCONS_EVENT_MOUSE_DELTA_Y — y delta amount;
- WSCONS_EVENT_MOUSE_ABSOLUTE_X — x location;
- WSCONS_EVENT_MOUSE_ABSOLUTE_Y — y location;
- WSCONS_EVENT_MOUSE_DELTA_Z — z delta amount;
- WSCONS_EVENT_MOUSE_ABSOLUTE_Z — legacy;
- WSCONS_EVENT_MOUSE_DELTA_W — w delta amount;
- WSCONS_EVENT_MOUSE_ABSOLUTE_W — legacy;
- WSCONS_EVENT_SYNC — synchronization signal generated;

Following events are not real wscons_event but are used as parameters of the WSDISPLAYIO_WSMOUSED ioctl:

- WSCONS_EVENT_WSMOUSED_ON — *wsmoused*(8) active;
- WSCONS_EVENT_WSMOUSED_OFF — *wsmoused*(8) inactive.

- WSMUXIO_ADD_DEVICE struct wsmux_device — adds a new multiplexor to the devices list. The wsmux_device structure is:

  Listing B.24: The wsmux_device structure.

  ```
  struct wsmux_device {
    int type;
    int idx;
  };
  ```

  The type member is one of:

  - WSMUX_MOUSE;
  - WSMUX_KBD;
  - WSMUX_MUX;

  The idx member is the index inside the wmux multiplexors list.

- WSMUXIO_REMOVE_DEVICE struct wsmux_device — removes the corresponding multiplexor from wmux list. See WSMUXIO_ADD_DEVICE;

- WSMUXIO_LIST_DEVICES struct wsmux_device_list — retrieves the multiplexors list of the devices in wmux. The wsmux_device_list is:

Listing B.25: The wsmux_device_list structure.

```
#define WSMUX_MAXDEV 32

struct wsmux_device_list {
  int ndevices;
  struct wsmux_device devices[ WSMUX_MAXDEV ];
};
```

The ndevices member could span from 0 to WSMUX_MAXDEV values. The devices member is an array of WSMUX_MAXDEV wsmux_device elements present in the wmux.

Listing B.26: wsmux - program to shows devices in wsmux.

```
/* -*- mode: c-mode; -*- */

/* wsmux.c file. */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <dev/wscons/wsconsio.h>

#define FOREVER for(;;)

/* program wsmux.c */
/* Functions prototypes. */
int main(int, char *[]);

/* Main function. */
int main(int argc, char *argv[])
{
  int fd;
  int i;
  long int ret = EXIT_FAILURE;
  struct wsmux_device_list mdevices;

  /* Check arguments count. */
  if(argc == 2) {
    fd = open(argv[ 1 ], O_RDWR | O_EXCL);
    if(fd >= 0) {
      if(ioctl(fd, WSMUXIO_LIST_DEVICES, &mdevices) >= 0) {
        printf("opened␣%s.\n", argv[ 1 ]);
        for(i = 0; i < mdevices.ndevices; i++) {
          printf("LIST_DEVICES:␣%d␣%d\n",    \
            mdevices.devices[ i ].type,    \
            mdevices.devices[ i ].idx);
        }
      } else
        perror("error␣retrieving␣devices␣list");
      close(fd);
    } else
      fprintf(stderr, "error␣open␣device␣%s.\n", argv[1]);
  } else
    fprintf(stderr, "usage:␣wsmux␣<device>\n");
  exit(ret);
}

/* End of wsmux.c file. */
```

Index

Bibliography

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. PTR Prentice Hall, Englewood Cliffs, New Jersey 07632, second edition, 1988.

[2] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S/ Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 6th edition, June 1999.

[3] W. Richars Stevens. *UNIX Network Programming - Interprocess Communications*, volume 2. Pearson Education International, second edition, March 2007.

[4] W. Richars Stevens. *UNIX Network Programming - The Sockets Networking API*, volume 1. Pearson Education International, third edition, August 2013.