CHAPTER 8: THE UNIX SYSTEM INTERFACE

The material in this chapter is concerned with the interface between C   
programs and the UNIXt operating system. Since most C users are on UNIX   
systems, this should be helpful to a majority of readers. Even if you use C   
on a different machine, however, you should be able to glean more insight   
into C programming from studying these examples.

The chapter is divided into three major areas: input/output, file system,   
and a storage allocator. The first two parts assume a modest familiarity with   
the external characteristics of UNIX.

Chapter 7 was concerned with a system interface that is uniform across a   
variety of operating systems. On any particular system the routines of the   
standard library have to be written in terms of the I/O facilities actually   
available on the host system. In the next few sections we will describe the   
basic system entry points for I/O on the UNIX operating system, and illus­   
trate how parts of the standard library can be implemented with them.

8.1 File Descriptors

In the UNIX operating system, all input and output is done by reading or   
writing files, because all peripheral devices, even the user's terminal, are   
files in the file system. This means that a single, homogeneous interface   
handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary   
to inform the system of your intent to do so, a process called "opening" the   
file. If you are going to write on a file it may also be necessary to create it.   
The system checks your right to do so (Does the file exist? Do you have   
permission to access it?), and if all is well, returns to the program a small   
positive integer called a *file descriPtor.* Whenever I/O is to be done on the   
file, the file descriptor is used instead of the name to identify the file. (This   
is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.)   
All information about an open file is maintained by the system; the user

t UNIX is a Trademark of Bell Laboratories.

159

160 THE C PROGRAMMING LANGUAGE CHAPTER 8

program refers to the file only by the file descriptor.

Since input and output involving the user's terminal is so common, spe­   
cial arrangements exist to make this convenient. When the command inter­   
preter (the "shell") runs a program, it opens three files, with file descriptors   
0, 1, and 2, called the standard input, the standard output, and the standard   
error output. All of these are normally connected to the terminal, so if a   
program reads file descriptor 0 and writes file descriptors 1 and 2, it can do   
terminal I/O without worrying about opening the files.

The user of a program can *redirect* I/O to and from files with < and >:

prog <infile >outfile

In this case, the shell changes the default assignments for file descriptors 0   
and 1 from the terminal to the named files. Normally file descriptor 2   
remains attached to the terminal, so error messages can go there. Similar   
observations hold if the input or output is associated with a pipe. In all   
cases, it must be noted, the file assignments are changed by the shell, not by   
the program. The program does not know where its input comes from nor   
where its output goes, so long as it uses file 0 for input and 1 and 2 for out­   
put.

8.2 Low Level I/O — Read and Write

The lowest level of I/O in UNIX provides no buffering or any other ser­   
vices; it is in fact a direct entry into the operating system. All input and   
output is done by two functions called read and write. For both, the first   
argument is a file descriptor. The second argument is a buffer in your pro­   
gram where the data is to come from or go to. The third argument is the   
number of bytes to be transferred. The calls are

n\_read = read(fd, buf, n);   
n\_written = write(fd, buf, n);

Each call returns a byte count which is the number of bytes actually   
transferred. On reading, the number of bytes returned may be less than the   
number asked for. A return value of zero bytes implies end of file, and —1   
indicates an error of some sort. For writing, the returned value is the   
number of bytes actually written; it is generally an error if this isn't equal to   
the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two   
most common values are 1, which means one character at a time   
("unbuffered"), and 512, which corresponds to a physical blocksize on   
many peripheral devices. This latter size will be most efficient, but even   
character at a time I/O is not inordinately expensive.

CHAPTER 8 THE UNIX SYSTEM INTERFACE 161

Putting these facts together, we can write a simple program to copy its   
input to its output, the equivalent of the file copying program written for   
Chapter 1. In UNIX, this program will copy anything to anything, since the   
input and output can be redirected to any file or device.

#define BUFSIZE 512 /\* best size for PDP-11 UNIX \*/

main() /\* copy input to output \*/

char buf[BUFSIZE];   
jilt n;

while ((n = read(0, buf, BUFSIZE)) > 0)   
write(1, buf, n);

If the file size is not a multiple of BUFSIZE, some read will return a   
smaller number of bytes to be written by write; the next call to read after   
that will return zero.

It is instructive to see how read and write can be used to construct   
higher level routines like getchar, putchar, etc. For example, here is a   
version of getchar which does unbuffered input.

#define CMASK 0377 /\* for making char's > 0 \*/   
getchar() /\* unbuffered single character input \*/

char c;

return((read(0, &c, 1) > 0) ? c & CMASK : Eor);

c *must* be declared char, because read accepts a character pointer. The   
character being returned must be masked with 0377 to ensure that it is   
positive; otherwise sign extension may make it negative. (The constant   
0377 is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of getchar does input in big chunks, and hands   
out the characters one at a time.

162 THE C PROGRAMMING LANGUAGE CHAPTER 8

#define CMASK 0377 /\* for making char's > 0 \*/   
#define BUFSIZE 512

getchar() /\* buffered version \*/

static char buf [BUFSIZE];

static char \*bufp = buf;

static int n = 0;

if (n == 0) ( /\* buffer is empty \*/

n = read(0, buf, BUFSIZE);

bufp = buf;

return((--n >= 0) ? \*bufp++ & CMASK : EOF);

8.3 Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must   
explicitly open files in order to read or write them. There are two system   
entry points for this, open and creat [sic].

open is rather like the fopen discussed in Chapter 7, except that   
instead of returning a file pointer, it returns a file descriptor, which is just an   
int.

int fd;

fd = open(name, rwmode);

As with fopen, the name argument is a character string corresponding to   
the external file name. The access mode argument is different, however:   
rwmode is 0 for read, 1 for write, and 2 for read and write access. open   
returns —1 if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point   
creat is provided to create new files, or to re-write old ones.

fd = creat(name, pmode);

returns a file descriptor if it was able to create the file called name, and —1   
if not. If the file already exists, creat will truncate it to zero length; it is   
not an error to creat a file that already exists.

If the file is brand new, creat creates it with the *protection mode*    
specified by the pmode argument. In the UNIX file system, there are nine   
bits of protection information associated with a file, controlling read, write   
and execute permission for the owner of the file, for the owner's group, and   
for all others. Thus a three-digit octal number is most convenient for speci­   
fying the permissions. For example, 0755 specifies read, write and execute

CHAPTER 8 THE UNIX SYSTEM INTERFACE 163

permission for the owner, and read and execute permission for the group   
and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp,* a pro­   
gram which copies one file to another. (The main simplification is that our   
version copies only one file, and does not permit the second argument to be   
a directory.)

#define NULL 0

#define BUFSIZE 512

#define PMODE 0644 /\* RW for owner, R for group, others \*/

main(argc, argv) /\* cp: copy f1 to f2 \*/

int argc;

char \*argv[];

int f1, f2, n;   
char buf [BUFSIZE];

if (argc != 3)

error("Usage: cp from to", NULL);   
if ((f1 = open(argv[1], 0)) == -1)

error("cp: can't open %s", argv[1]);   
if ((f2 = creat(argv[2], PMODE)) == -1)   
error("cp: can't create %s", argv[2]);

while ((n = read(f1, buf, BUFSIZE)) > 0)   
if (write(f2, buf, n) != n)

error("cp: write error", NULL);

exit (0)

error(s1, s2) /\* print error message and die \*/   
char \*s1, \*s2;

printf(s1, s2);   
printf("\n");   
exit(1);

There is a limit (typically 15-25) on the number of files which a program   
may have open simultaneously. Accordingly, any program which intends to   
process many files must be prepared to re-use file descriptors. The routine   
close breaks the connection between a file descriptor and an open file, and   
frees the file descriptor for use with some other file. Termination of a pro­   
gram via exit or return from the main program closes all open files.

The function unlink ( f ilename ) removes the file filename from   
the file system.

164 THE C PROGRAMMING LANGUAGE CHAPTER 8

Exercise 8-1. Rewrite the program cat from Chapter 7 using read,   
write, open and close instead of their standard library equivalents. Per­   
form experiments to determine the relative speeds of the two versions. El

8.4 Random Access — Seek and Lseek

File I/0 is normally sequential: each read or write takes place at a   
position in the file right after the previous one. When necessary, however,   
a file can be read or written in any arbitrary order. The system call lseek   
provides a way to move around in a file without actually reading or writing:

lseek(fd, offset, origin);

forces the current position in the file whose descriptor is fd to move to   
position offset, which is taken relative to the location specified by   
origin. Subsequent reading or writing will begin at that position.   
offset is a long; fd and origin are it's. origin can be 0, 1, or 2   
to specify that offset is to be measured from the beginning, from the   
current position, or from the end of the file respectively. For example, to   
append to a file, seek to the end before writing:

lseek(fd, OL, 2);

To get back to the beginning ("rewind"),

lseek(fd, OL, 0);

Notice the OL argument; it could also be written as (long) 0.

With lseek, it is possible to treat files more or less like large arrays, at   
the price of slower access. For example, the following simple function reads   
any number of bytes from any arbitrary place in a file.

get(fd, pos, buf, n) /\* read n bytes from position pos \*/

int fd, n;

long pos;

char \*buf;

(

lseek(fd, pos, 0); /\* get to pos \*/

return(read(fd, buf, n));

)

In pre-version 7 UNIX, the basic entry point to the I/O system is called   
seek. seek is identical to lseek, except that its offset argument is an   
int rather than a long. Accordingly, since PDP-11 integers have only 16   
bits, the offset specified for seek is limited to 65,535; for this reason,   
origin values of 3, 4, 5 cause seek to multiply the given offset by 512   
(the number of bytes in one physical block) and then interpret origin as if   
it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large   
file requires two seeks, first one which selects the block, then one which has

CHAPTER 8 THE UNIX SYSTEM INTERFACE 165

origin equal to 1 and moves to the desired byte within the block.

Exercise 8-2. Clearly, seek can be written in terms of lseek, and vice   
versa. Write each in terms of the other. 0

8.5 Example — An Implementation of Fopen and Getc

Let us illustrate how some of these pieces fit together by showing an   
implementation of the standard library routines fopen and getc.

Recall that files in the standard library are described by file pointers   
rather than file descriptors. A file pointer is a pointer to a structure that   
contains several pieces of information about the file: a pointer to a buffer, so   
the file can be read in large chunks; a count of the number of characters left   
in the buffer; a pointer to the next character position in the buffer; some   
flags describing read/write mode, etc.; and the file descriptor.

The data structure that describes a file is contained in the file stdio.h,   
which must be included (by #include) in any source file that uses rou­   
tines from the standard library. It is also included by functions in that   
library. In the following excerpt from stdio.h, names which are intended   
for use only by functions of the library begin with an underscore so they are   
less likely to collide with names in a user's program.

#define \_BUFSIZE 512

#define \_NFILE 20 /\* #files that can be handled \*/

typedef struct \_iobuf (

**char \*\_ptr; /\* next character position \*/**

**int \_cnt; /\* number of characters left \*/**

**char \*\_base; /\* location of buffer \*/**

**int \_flag; /\* mode of file access \*/**

**int \_fd; /\* file descriptor \*/**

**) FILE;**

**extern FILE \_iob[\_NFILE];**

#define stdin (&\_iob[0])   
#define stdout (&\_iob[1])   
#define stderr (&\_iob[2])

#define \_READ 01 /\* file open for reading \*/

#define \_WRITE 02 /\* file open for writing \*/

#define \_UNBUF 04 /\* file is unbuffered \*/

#define \_BIGBUF 010 /\* big buffer allocated \*/   
#define \_EOF 020 /\* EOF has occurred on this file \*/   
#define \_ERR 040 /\* error has occurred on this file \*/   
#define NULL 0

#define EOF (-1)

166 THE C PROGRAMMING LANGUAGE CHAPTER 8

#define getc(p) (--(p)->\_cnt >= 0 \

? \*(p)->\_ptr++ & 0377 : \_fillbuf(p))

#define getchar() getc(stdin)

#define putc(x,p) (--(p)->\_cnt >= 0 \

? \*(p)->\_ptr++ = (x) : \_flushbuf((x),p))

#define putchar(x) putc(x,stdout)

The getc macro normally just decrements the count, advances the   
pointer, and returns the character. (A long #define is continued with a   
backslash.) If the count goes negative, however, getc calls the function   
\_fillbuf to replenish the buffer, re-initialize the structure contents, and   
return a character. A function may present a portable interface, yet itself   
contain non-portable constructs: getc masks the character with 0377,   
which defeats the sign extension done by the PDP-11 and ensures that all   
characters will be positive.

Although we will not discuss any details, we have included the definition   
of putc to show that it operates in much the same way as getc, calling a   
function \_f lushbuf when its buffer is full.

The function f open can now be written. Most of f open is concerned   
with getting the file opened and positioned at the right place, and setting the   
flag bits to indicate the proper state. fopen does not allocate any buffer   
space; this is done by \_fillbuf when the file is first read.

CHAPTER 8 THE UNIX SYSTEM INTERFACE 167

#include <stdio.h>

#define PMODE 0644 /\* R/W for owner; R for others \*/

FILE \*fopen(name, mode) /\* open file, return file ptr \*/   
register char \*name, \*mode;

register int fd;   
register FILE \*fp;

if (\*mode != 'r' && \*mode != 'w' && \*mode != 'a') (   
fprintf(stderr, "illegal mode %s opening %s\n",   
mode, name);

exit (1);

for (fp = \_iob; fp < \_iob + \_NFILE; fp++)

if ((fp->\_f lag & (\_READ I \_WRITE)) == 0)

break; /\* found free slot \*/

if (fp >= \_iob + \_NFILE) /\* no free slots \*/

return (NULL)

if (\*mode == 'w') /\* access file \*/

fd = creat(name, PMODE);

else if (\*mode == 'a') (

if ((fd = open(name, 1)) == -1)

fd = creat(name, PMODE);

lseek(fd, OL, 2);

) else

fd = open (name, 0);

if (fd == -1) /\* couldn't access name \*/

return(NULL);

fp->\_fd = fd;

fp->\_cnt = 0;

fp->\_base = NULL;

fp->\_f lag &= -LREAD I \_WRITE);

fp->\_f lag 1= (\*mode == 'r') ? \_READ : \_WRITE;

return(fp);

The function fillbuf is rather more complicated. The main com­   
plexity lies in the fact that \_fillbuf attempts to permit access to the file   
even though there may not be enough memory to buffer the I/O. If space   
for a new buffer can be obtained from cal loc, all is well; if not,   
\_fillbuf does unbuffered I/O using a single character stored in a private   
array.

168 THE C PROGRAMMING LANGUAGE CHAPTER 8

#include <stdio.h>

\_fillbuf(fp) /\* allocate and fill input buffer \*/   
register FILE \*fp;

static char smallbuf[\_NFILE]; /\* for unbuffered I/O \*/   
char \*calloc();

if ((fp->\_flag&\_READ)==0 II (fp->\_flag&(\_EOFI\_ERR))1=0)

return (EOF);

while (fp->\_base == NULL) /\* find buffer space \*/

if (fp->\_flag & \_UNBUF) /\* unbuffered \*/

fp->\_base = &smallbuf[fp->\_fd];

else if ((fp->\_base=calloc(\_BUFSIZE, 1)) == NULL)

fp->\_flag I= \_UNBUF; /\* can't get big buf \*/

else

fp->\_flag I= \_BIGBUF; /\* got big one \*/

fp->\_ptr = fp->\_base;

fp->\_cnt = read(fp->\_fd, fp->\_ptr,

fp->\_flag & \_UNBUF ? 1 : \_BUFSIZE);

if (--fp->\_cnt < 0) (

if (fp->\_cnt == -1)

fp->\_flag I= \_EOF;

else

fp->\_flag I= \_ERR;

fp->\_cnt = 0;

return (EOF);

return(\*fp->\_ptr++ & 0377); /\* make char positive \*/

The first call to getc for a particular file finds a count of zero, which forces   
a call of \_fillbuf. If \_fillbuf finds that the file is not open for read­   
ing, it returns EOF immediately. Otherwise, it tries to allocate a large   
buffer, and, failing that, a single character buffer, setting the buffering infor­   
mation in \_flag appropriately.

Once the buffer is established, \_fillbuf simply calls read to fill it,   
SOS the count and pointers, and returns the character at the beginning of   
the buffer. Subsequent calls to \_fillbuf will find a buffer allocated.

The only remaining loose end is how everything gets started. The array   
\_iob must be defined and initialized for stdin, stdout and stderr:

FILE \_iob[\_NFILE] =(

( NULL, 0, NULL, \_READ, 0 ), /\* stdin \*/

( NULL, 0, NULL, \_WRITE, 1 ), /\* stdout \*/

( NULL, 0, NULL, \_WRITE I \_UNBUF, 2 ) /\* stderr \*/   
;

CHAPTER 8 THE UNIX SYSTEM INTERFACE 169

The initialization of the \_flag part of the structure shows that stdin is to   
be read, stdout is to be written, and stderr is to be written unbuffered.

Exercise 8-3. Rewrite f open and \_fillbuf with fields instead of explicit   
bit operations. 0

Exercise 8-4. Design and write the routines \_flushbuf and fclose. 0   
Exercise 8-5. The standard library provides a function

fseek(fp, offset, 'origin)

which is identical to lseek except that fp is a file pointer instead of a file   
descriptor. Write f seek. Make sure that your f seek coordinates properly   
with the buffering done for the other functions of the library. CI

8.6 Example — Listing Directories

A different kind of file system interaction is sometimes called for —   
determining information *about* a file, not what it contains. The UNIX com­   
mand *Is* ("list directory") is an example — it prints the names of files in a   
directory, and optionally, other information, such as sizes, permissions, and   
so on.

Since on UNIX at least a directory is just a file, there is nothing special   
about a command like */s;* it reads a file and picks out the relevant parts of   
the information it finds there. Nonetheless, the format of that information   
is determined by the system, not by a user program, so */s* needs to know   
how the system represents things.

We will illustrate some of this by writing a program called *fsize. fsize* is   
a special form of */s* which prints the sizes of all files named in its argument   
list. If one of the files is a directory, *fsize* applies itself recursively to that   
directory. If there are no arguments at all, it processes the current directory.

To begin, a short review of file system structure. A directory is a file   
that contains a list of file names and some indication of where they are   
located. The "location" is actually an index into another table called the   
"mode table." The mode for a file is where all information about a file   
except its name is kept. A directory entry consists of only two items, an   
mode number and the file name. The precise specification comes by includ­   
ing the file sys/dir .h, which contains

170 THE C PROGRAMMING LANGUAGE CHAPTER8

#define DIRSIZ 14 /\* max length of file name \*/

struct direct /\* structure of directory entry \*/   
(

ino\_t d\_ino; /\* mode number \*/

char d\_name[DIRSIZ]; /\* file name \*/   
) ;

The "type" ino\_t is a typedef describing the index into the mode   
table. It happens to be unsigned on PDP-11, UNIX, but this is not the sort   
of information to embed in a program: it might be different on a different   
system. Hence the typedef. A complete set of "system" types is found   
in sys/types .h.

The function stat takes a file name and returns all of the information   
in the mode for that file (or —1 if there is an error). That is,

struct stat stbuf;   
char \*name;

stat(name, &stbuf);

fills the structure stbuf with the mode information for the file name. The   
structure describing the value returned by stat is in sys/stat .h, and   
looks like this:

|  |  |  |  |
| --- | --- | --- | --- |
| struct stat  (  dev\_t  ino\_t | /\* structure returned by stat \*/  st\_dev; /\* device of mode \*/  st\_ino; /\* mode number \*/ | | |
| short | st\_mode; | /\* | mode bits \*/ |
| short | st\_nlink; | /\* | number of links to file \*/ |
| short | st\_uid; | /\* | owner's userid \*/ |
| short | st\_gid; | /\* | owner's group id \*/ |
| dev\_t | st\_rdev; | /\* | for special files \*/ |
| off\_t | st\_size; | /\* | file size in characters \*/ |
| time\_t | st\_atime; | /\* | time last accessed \*/ |
| time\_t | st\_mtime; | /\* | time last modified \*/ |
| time\_t | st\_ctime; | /\* | time originally created \*/ |

);

Most of these are explained by the comment fields. The st\_mode entry   
contains a set of flags describing the file; for convenience, the flag   
definitions are also part of the file sys/stat . h.

**\*/**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CHAPTER 8  #define | S\_IFMT 0160000 | |  | THE UNIX SYSTEM INTERFACE 171  /\* type of file \*/ |
| #define | S\_IFDIR | 0040000 |  | /\* directory \*/ |
| #define | S\_IFCHR | 0020000 |  | /\* character special \*/ |
| #define | S\_IFBLK | 0060000 |  | /\* block special \*/ |
| #define | S\_IFREG | 0100000 |  | /\* regular \*/ |
| #define | S\_ISUID | 04000 | /\* | set user id on execution \*/ |
| #define | S\_ISGID | 02000 | /\* | set group id on execution \*/ |
| #define | S\_ISVTX | 01000 | /\* | save swapped text after use |
| #define | S\_IREAD | 0400 | /\* | read permission \*/ |
| #define | S\_IWRITE | 0200 | /\* | write permission \*/ |
| #define | S\_IEXEC | 0100 | /\* | execute permission \*/ |

Now we are able to write the program *fsize.* If the mode obtained from   
stat indicates that a file is not a directory, then the size is at hand and can   
be printed directly. If it is a directory, however, then we have to process   
that directory one file at a time; it in turn may contain sub-directories, so   
the process is recursive.

The main routine as usual deals primarily with command-line argu­   
ments; it hands each argument to the function fsize in a big buffer.

|  |  |
| --- | --- |
| #include <stdio.h>  #include <sys/types.h>  #include <sys/dir.h>  #include <sys/stat.h>  #define BUFSIZE 256 | /\* typedefs \*/  /\* directory entry structure \*/  /\* structure returned by stat \*/ |

**main(argc, argv) /\* fsize: print file sizes \*/**

**char \*argy[];**

**char buf[BUFSIZE];**

if (argc == 1) ( /\* default: current directory \*/

strcpy(buf, ".");   
fsize(buf):

) else

while (--argc > 0) (

strcpy(buf, \*++argv);

fsize(buf);

The function fsize prints the size of the file. If the file is a directory,   
however, fsize first calls directory to handle all the files in it. Note   
the use of the flag names S\_IFMT and S\_IFDIR from stat .h.

172 THE C PROGRAMMING LANGUAGE CHAPTER 8

fsize (name) /\* print size for name \*/

char \*name;

(

struct stat stbuf;

if (stat(name, &stbuf) == -1) (

fprintf(stderr, "fsize: can't find %s\n", name);

return;

)

if ((stbuf.st\_mode & S\_IFMT) == S\_IFDIR)

directory (name);

printf("%81d %s\n", stbuf.st\_size, name);

)

The function directory is the most complicated. Much of it is con­   
cerned, however, with creating the full pathname of the file being dealt   
with.

directory (name) /\* fsize for all files in name \*/

char \*name;

(

struct direct dirbuf;

char \*nbp, \*nep;

int i, fd;

nbp = name + strlen(name);

\*nbp++ = '/'; /\* add slash to directory name \*/

if (nbp+DIRSIZ+2 >= name+BUFSIZE) /\* name too long \*/

return;

if ((fd = open(name, 0)) == -1)

return;

while (read(fd, (char \*)&dirbuf, sizeof(dirbuf))>0) (

if (dirbuf.d\_ino == 0) /\* slot not in use \*/

continue;

if (strcmp(dirbuf.d\_name, ".") == 0

II strcmp(dirbuf.d\_name, "..") == 0)

continue; /\* skip self and parent \*/

for (i=0, nep=nbp; i < DIRSIZ; i++)

\*nep++ = dirbuf.d\_name[i];

\*nep++ = '\0';

fsize (name);

)

close(fd);

\*--nbp = '\0'; /\* restore name \*/

)

If a directory slot is not currently in use (because a file has been   
removed), the mode entry is zero, and this position is skipped. Each direc­   
tory also contains entries for itself, called " . ", and its parent, " . . "; clearly

CHAPTER 8 THE UNIX SYSTEM INTERFACE 173

these must also be skipped, or the program will run for quite a while.

Although the *fsize* program is rather specialized, it does indicate a couple   
of important ideas. First, many programs are not "system programs"; they   
merely use information whose form or content is maintained by the operat­   
ing system. Second, for such programs, it is crucial that the representation   
of the information appear only in standard "header files" like stat.h and   
dir.h, and that programs include those files instead of embedding the   
actual declarations in themselves.

8.7 Example — A Storage Allocator

In Chapter 5, we presented a simple-minded version of alloc. The   
version which we will now write is unrestricted: calls to alloc and free   
may be intermixed in any order; alloc calls upon the operating system to   
obtain more memory as necessary. Besides being useful in their own right,   
these routines illustrate some of the considerations involved in writing   
machine-dependent code in a relatively machine-independent way, and also   
show a real-life application of structures, unions and typedef.

Rather than allocating from a compiled-in fixed-sized array, alloc will   
request space from the operating system as needed. Since other activities in   
the program may also request space asynchronously, the space alloc   
manages may not be contiguous. Thus its free storage is kept as a chain of   
free blocks. Each block contains a size, a pointer to the next block, and the   
space itself. The blocks are kept in order of increasing storage address, and   
the last block (highest address) points to the first, so the chain is actually a   
ring.

When a request is made, the free list is scanned until a big enough   
block is found. If the block is exactly the size requested it is unlinked from   
the list and returned to the user. If the block is too big, it is split, and the   
proper amount is returned to the user while the residue is put back on the   
free list. If no big enough block is found, another block is obtained from   
the operating system and linked into the free list; searching then resumes.

Freeing also causes a search of the free list, to find the proper place to   
insert the block being freed. If the block being freed is adjacent to a free   
list block on either side, it is coalesced with it into a single bigger block, so   
storage does not become too fragmented. Determining adjacency is easy   
because the free list is maintained in storage order.

One problem, which we alluded to in Chapter 5, is to ensure that the   
storage returned by alloc is aligned properly for the objects that will be   
stored in it. Although machines vary, for each machine there is a most res­   
trictive type: if the most restricted type can be stored at a particular address,   
all other types may be also. For example, on the IBM 360/370, the   
Honeywell 6000, and many other machines, any object may be stored on a   
boundary appropriate for a double; on the PDP-11, int suffices.

174 THE C PROGRAMMING LANGUAGE CHAPTER 8

A free block contains a pointer to the next block in the chain, a record   
of the size of the block, and then the free space itself; the control informa­   
tion at the beginning is called the "header." To simplify alignment, all   
blocks are multiples of the header size, and the header is aligned properly.   
This is achieved by a union that contains the desired header structure and an   
instance of the most restrictive alignment type:

typedef int ALIGN; /\* forces alignment on PDP-11 \*/

union header ( /\* free block header \*/

struct

union header \*ptr; /\* next free block \*/

unsigned size; /\* size of this free block \*/

) s;

ALIGN x; /\* force alignment of blocks \*/

) ;

typedef union header HEADER;

In alloc, the requested size in characters is rounded up to the proper   
number of header-sized units; the actual block that will be allocated contains   
one more unit, for the header itself, and this is the value recorded in the   
size field of the header. The pointer returned by alloc points at the free   
space, not at the header itself.

CHAPTER 8 THE UNIX SYSTEM INTERFACE 175

static HEADER base; /\* empty list to get started \*/   
static HEADER \*allocp = NULL; /\* last allocated block \*/

char \*alloc(nbytes) /\* general-purpose storage allocator \*/   
unsigned nbytes;

HEADER \*morecore();   
register HEADER \*p, \*q;   
register int nunits;

nunits = 1+(nbytes+sizeof(HEADER)-1)/sizeof(HEADER);

if ((ci = allocp) == NULL) ( /\* no free list yet \*/

base.s.ptr = allocp = q = &base;

base.s.size = 0;

for (p=q->s.ptr; ; q=p, p=p->s.ptr) (

if (p->s.size >= nunits) ( /\* big enough \*/   
if (p->s.size == nunits) /\* exactly \*/   
q->s.ptr = p->s.ptr;

else ( /\* allocate tail end \*/

p->s.size -= nunits;

p += p->s.size;

p->s.size = nunits;

allocp = q;

return((char \*)(p+1));

if (p == allocp) /\* wrapped around free list \*/   
if ((p = morecore(nunits)) == NULL)   
return(NULL); /\* none left \*/

The variable base is used to get started; if allocp is NULL, as it is at   
the first call of alloc, then a degenerate free list is created: it contains one   
block of size zero, and points to itself. In any case, the free list is then   
searched. The search for a free block of adequate size begins at the point   
(allocp) where the last block was found; this strategy helps keep the list   
homogeneous. If a too-big block is found, the tail end is returned to the   
user; in this way the header of the original needs only to have its size   
adjusted. In all cases, the pointer returned to the user is to the actual free   
area, which is one unit beyond the header. Notice that p is converted to a   
character pointer before being returned by alloc.

The function morecore obtains storage from the operating system.   
The details of how this is done of course vary from system to system. In   
UNIX, the system entry sbrk(n) returns a pointer to n more bytes of   
storage. (The pointer satisfies all alignment restrictions.) Since asking the

176 THE C PROGRAMMING LANGUAGE CHAPTER 8

system for memory is a comparatively expensive operation, we don't want to   
do that on every call to al loc, so morecore rounds up the number of   
units requested of it to a larger value; this larger block will be chopped up as   
needed. The amount of scaling is a parameter that can be tuned as needed.

#define NALLOC 128 /\* #units to allocate at once \*/

static HEADER \*morecore(nu) /\* ask system for memory \*/

unsigned nu;

char \*sbrk();

register char \*cp;   
register HEADER \*up;   
register int mu;

mu = NALLOC \* ((nu+NALLOC-1) / NALLOC);

cp = sbrk (mu \* sizeof(HEADER));

if ((int)cp == -1) /\* no space at all \*/

return(NULL);

up = (HEADER \*)cp;

up->s.size = mu;

free ((char \*)(up+1));

return(allocp);

sbrk returns —1 if there was no space, even though NULL would have   
been a better choice. The —1 must be converted to an int so it can be   
safely compared. Again, casts are heavily used so the function is relatively   
immune to the details of pointer representation on different machines.

free itself is the last thing. It simply scans the free list, starting at   
allocp, looking for the place to insert the free block. This is either   
between two existing blocks or at one end of the list. In any case, if the   
block being freed is adjacent to either neighbor, the adjacent blocks are com­   
bined. The only troubles are keeping the pointers pointing to the right   
things and the sizes correct.

CHAPTER 8 THE UNIX SYSTEM INTERFACE 177

free(ap) /\* put block ap in free list \*/   
char \*ap;

register HEADER \*p, \*q;

p = (HEADER \*)ap - 1; /\* point to header \*/

for (q=allocp; > q && p < q->s.ptr); q=q->s.ptr)   
if (q >= q->s.ptr && (p > q II p < q->s.ptr))

break; /\* at one end or other \*/

if (p+p->s.size == q->s.ptr) ( /\* join to upper nbr \*/

p->s.size += q->s.ptr->s.size;

p->s.ptr = q->s.ptr->s.ptr;

) else

p->s.ptr = q->s.ptr;

if (q+q->s.size == p) ( /\* join to lower nbr \*/

q->s.size += p->s.size;

q->s.ptr = p->s.ptr;

) else

q->s.ptr = p;

allocp = q;

Although storage allocation is intrinsically machine dependent, the code   
shown above illustrates how the machine dependencies can be controlled   
and confined to a very small part of the program. The use of typedef and   
union handles alignment (given that sbrk supplies an appropriate pointer).   
Casts arrange that pointer conversions are made explicit, and even cope with   
a badly-designed system interface. Even though the details here are related   
to storage allocation, the general approach is applicable to other situations as   
well.

Exercise 8-6. The standard library function ca (n, size) returns a

pointer to n objects of size size, with the storage initialized to zero. Write   
ca 3.1oc, using alloc either as a model or as a function to be called. 0

Exercise 8-7. alloc accepts a size request without checking its plausibility;   
free believes that the block it is asked to free contains a valid size field.   
Improve these routines to take more pains with error checking. 0

Exercise 8-8. Write a routine bfree (p, n) which will free an arbitrary   
block p of n characters into the free list maintained by alloc and free.   
By using bfree, a user can add a static or external array to the free list at   
any time. 0