Whitesmiths, Ltd.

IDRIS PROGRAMMERS' MANUAL

Date: January 1985

SN09

The C language was developed at Bell Laboratories by Dennis Ritchie; Whitesmiths, Ltd. has endeavored to remain as faithful as possible to his language specification. The external specifications of the IDRIS operating system, and of most of its utilities, are based heavily on those of UNIX, which was also developed at Bell Laboratories by Dennis Ritchie and Ken Thompson. Whitesmiths, Ltd. gratefully acknowledges the parentage of many of the concepts we have commercialized, and we thank Western Electric Co. for waiving patent licensing fees for use of the UNIX protection mechanism.

The successful implementation of Whitesmiths' compilers, operating systems, and utilities, however, is entirely the work of our programming staff and allied consultants.

For the record, UNIX is a trademark of Bell Laboratories; IAS, RSTS/E, VAX, VMS, P/OS, PDP-11, RT-11, RSX-11M, and nearly every other term with an 11 in it all are trademarks of Digital Equipment Corporation; CP/M is a trademark of Digital Research Co.; MC68000 and VERSAdos are trademarks of Motorola Inc.; ISIS and iRMX are trademarks of Intel Corporation; A-Natural, IDRIS, and ctext are trademarks of Whitesmiths, Ltd. C is not.

SECTIONS

I. Whitesmithing

II. IDRIS System Interface

III. Programming File Formats

IV. IDRIS Support Library

SCOPE

This manual is meant to familiarize the more technically sophisticated user with the IDRIS program development environment. Section I provides tutorial descriptions introducing the environment and tools used to build new programs. Section II contains descriptions of the system calls and other routines that constitute the IDRIS system interface across various machines. Section III details the formats of numerous files used by the IDRIS resident or utilities that are of particular note to programmers, while Section IV documents library routines developed for use with the IDRIS utilities.

Tutorials and detailed descriptions of standard utilities, as well as the System Administration Guide, may be found in the IDRIS Users' Manual. More succinct documentation for the programming utilities, may be found in the C Interface Manual for the appropriate target machine; and the machine dependent aspects of each IDRIS implementation are discussed in the IDRIS Interface Manual for each target machine.

TABLE OF CONTENTS

I. Whitesmithing

II. IDRIS System Interface

III. Programming File Formats

IV. IDRIS Support Library

SECTION ONE

WHITESMITHING

NAME

Process - rules for an Idris program

FUNCTION

Idris provides a relatively clean and simple execution environment. Memory layout is straightforward, input/output is made to look identical for a large variety of devices and files, and system services are packaged as functionally cohesive routines. Programs thus tend to be small, numerous, and reusable.

Such a world is sufficiently uncommon that it warrants a few introductory remarks:

MEMORY

The execution of a program under Idris is called a "process", a term so fundamental that it is used with a variety of connotations. Abstractly, a process is the elaboration, over time, of the instructions spelled out in a program file. More pragmatically, it is a piece of memory, together with some control registers and secret notes kept by the resident, that is initialized from a program file and that changes as the program executes. Specifically, the secret notes in the resident are sometimes referred to as the process; and at other times, the piece of memory plus control registers that get copied about is called the process.

Regardless, the concept of a memory image evolving over time lies at the heart of the matter. Idris confuses the issue greatly by supporting any number of simultaneously executing processes, so the resident is constantly copying images about memory and even to a "swap" area on disk. The image itself is thus the most concrete thing about a process.

The memory image that a programmer sees comes in two chunks, known as text (instructions, or I-space) and data (variables, or D-space). The text section (chunk) holds all the machine instructions for the program; if produced by a compiler such as C or Pascal, these instructions are further clumped into separate "functions". Idris cares little about this internal structure. It commences program execution by transferring control to the start of the text section, and it reserves the right to disallow changes to the text section (i.e., it may write protect it). That's about it.

The data section is more elaborate. Its lowest (addressed) portion is initialized from the data portion of the program file, parts of which may remain unchanged (constants, tables) and parts of which may be altered in time (variables). A program file may also specify that a certain number of bytes, right above the initialized data area, be set aside and filled with zeros at program startup; this "bss area" merely serves to save space in the program file. Above the bss area is a space reserved for the bss area to be expanded upward, to grow a "heap", and for a "stack" to grow downward.

The stack is a last-in/first-out queue of local storage frames to support recursive function calls. Both C and Pascal make extensive use of the

stack. A heap is a data area from which chunks of storage can be allocated and freed in arbitrary order. Idris supports heap mamagement by maintaining a "data break" address (part of the secret notes kept on behalf of a process memory image); the data break is moved up only upon request of the running program, to make room for more heap storage. Should the heap (data break) and stack (stack pointer) ever meet or cross, the process is deemed out of memory and is aborted. (It is sent the signal SIGSEG, to be precise, which usually forces the process to terminate abruptly.)

There is one further nicety about a process memory image. At program startup, up to 512 characters worth of NUL terminated strings may be written at the very top of memory (bottom of stack), with pointers to these strings pushed on the stack. Such strings are arguments passed to the new program by the program that caused it to startup. They can be ignored or modified as the new program sees fit, but usually they are used as a highly convenient channel for obtaining input parameters.

So this text plus data, together with registers such as the stack pointer and program counter, form the virtual machine upon which a process runs. Depending upon the implementation of Idris, the text portion may be loaded starting at location zero (in I-space), or both text and data may be loaded at separate location zeros (separate I-space and D-space), or neither may truly begin at zero. In the last case, the resident is usually obliged to alter the program image at startup time, to "rerelocate" it for the load address chosen. Once load addresses are determined, however, they cannot change for the life of a program, lest computed addresses be invalidated.

PROCESS

How are processes born? All of them are descendants of an initial "process 1", concocted by the resident at system startup. A descendant is spawned by a program requesting the system to "fork", or make a second process that is almost the identical twin of the requesting process. This alone is not very useful, except that the newer of the twins can request the system to "exec" a new program file, passing it designated argument strings. Here is the genesis of "program startup", mentioned several times above.

So the basic drill is: a running program (process) determines that another program should be run as well. It forks, instructing its child (descendant almost-twin) to exec the new program. The child memory image is overlaid with that of the new program, plus cleared bss area, plus space reservation for heap plus stack, plus arguments passed to it. And the torch is passed on.

There are other kinds of requests that a program can make of the system, about forty of them in fact. The requests are known as "system calls", which can be treated much like (highly sophisticated) machine instructions added to the instruction set of the target machine. Perhaps a third of these deal with process administration; the rest deal with various

aspects of input/output.

The other important system calls in the process administration area are: "exit", to bring a process to a tidy conclusion and report back its status; "wait", to synchronize with a child process performing an exit and collect its status report; "kill", to send a brief but powerful message to a related process; and "signal", to specify how such powerful messages are to be handled. This is pretty much the extent to which processes may directly communicate, except through file I/O. It is usually sufficient.

INPUT/OUTPUT

The Idris Users' Manual has a lot to say about how various forms of I/O can be made to look almost identical by reducing everything to text streams. And the C Programmers' Manual describes a standard system interface which is modelled almost directly by the Idris resident. A few points are worth reemphasizing here, however.

First is the fact that the Idris resident converts between disk blocks and streams of bytes for the programmer, and it optimizes such operations across time and across multiple processes. Per-process memory space thus need not be cluttered with multiple buffers to achieve the joint goals of simple I/O requests and efficient operation.

Then there is the hierarchical filesystem, which makes it easier to impose structure on collections of files and which greatly simplifies the rules for forming filenames. The protection machinery is largely concentrated in the file access modes, so it is easier to make a system secure.

About half the I/O system calls deal with files by name. All use exactly the same conventions for specifying filenames to the resident. The remaining I/O system calls deal with "file descriptors". Like those in the standard C system interface, Idris file descriptors are small non-negative integers which specify which previously opened file is to be used for an operation. Any information on the file status is kept in resident memory, so it is far less susceptible to corruption.

To some extent, the underlying structure of file input/output shows through: directories are conventionally read as ordinary files, in the absence of any special system calls for walking directories; the structure of a filesystem "inode" is made visible as part of the status returned for a file; and internal codes for physical devices appear in file status and in conjunction with special file inodes. Nonetheless, most of this information can be, and is, largely ignored by the bulk of programs that run under Idris.

CONVENTIONS

Some of the generality of the Idris environment is cheerfully sacrificed in the interest of standardizing the requirements for a program. Here are a few arbitrary limitations that are almost universally adopted:

Three file descriptors are assumed to correspond to opened files at program startup: STDIN (0) is the standard source of input, STDOUT (1) is the standard destination for output, and STDERR (2) is the standard destination for error messages. It should be possible to open upwards of a dozen more files, simultaneously within any program.

There is always at least one argument string at program startup. It is taken as the name by which the program was invoked.

Filenames should not be longer than 64 characters, counting the terminating NUL. Text lines should not be longer than 512 characters, counting the terminating newline. And the last character of a text file, if any, should always be a newline (i.e. there should be no partial last line).

NAME

Link - using link and related tools

FUNCTION

Several tools are available for building programs, under Idris and for the Idris environment. These are known as "compilers", for translating high level languages such as Pascal and C, "assemblers", for translating machine specific low level languages, and various tools for dealing with "relocatable object modules". All of them focus on delivering up to a program builder, called link, all the pieces needed to put together a program file that is digestible by the Idris exec system call.

The approach supported by these tools is a traditional one, centered around the concept of "separate compilation", much like the environment implied for FORTRAN or PL/I. It is by no means the only way to build programs. There are interpreters, for languages like Basic and APL, monolithic language systems, for COBOL and standard Pascal, and more ambitious schemes, for the language Ada. Separate compilation is generally more efficient than interpreters, more flexible than monolithic systems, and much simpler than the ambitious approach. Its major drawback is that it imposes a different, and usually much weaker, set of rules for combining modules into a program than for combining functions into a module.

At any rate, separate compilation permits a program to be put together, as described above, from an assortment of pieces. Some of these pieces can be the output of a C compiler, still others can be generated from hand written assembler code. Some modules can be linked unconditionally, still others can be selected only as needed from libraries of useful functions. All input to link, regardless of its source, must be reduced to standard object modules, which are then combined to produce the program file.

The same version of link is used, by the way, regardless of target machine. The standard object module header contains a format byte which specifies gross properties of the target machine, such as byte order within words and word size. link adopts the first format byte it encounters and uses it to adapt to the associated target environment.

One of the great strengths of the Idris link utility is that its output is, or can be, in the same format as its input. Thus, a program can be built up in several stages, possibly with special handling at some of the stages. Such special handling is rarely if ever needed in building a program for execution under Idris, but link is much more ambitious. It can be used, for example, to build programs that can run free standing, such as the system bootstraps and the Idris resident. It can be used to build programs for execution under other operating systems, such as CP/M. And it can be used to build shared libraries, or to make ROM images that are loaded one place in memory but are designed to execute somewhere else.

As a side effect of producing files that can be reprocessed, link retains information in the final program file that can be quite useful. The symbol table, or list of external identifiers, is handy when debugging programs; the programming utility db makes use of this, and the utility rel can be made to produce a readable list of symbols from an object file. There are also "relocation bits", information on how to alter the instruc-

tions and initialized data if they must be moved to a different place in memory. db uses the relocation bits to refine its output when disassembling machine instructions; and some versions of the Idris resident demand such information so they can have more flexibility in utilizing memory. Finally, each object module has in its header useful information such as text and data sizes.

On the other hand, link produces no memory map (the rel utility comes closest to doing that). It has no provision for overlays. And it can only deal with code contributions to two sections, text and data. link should be thought of as simple, but flexible.

IDRIS

Building a program for execution under Idris is straightforward. The text and data sections of Idris correspond exactly to those manipulated by link, and compilers and assemblers are predisposed to put machine instructions in the text section and variables in data. There are even entries in the object file header (oddly enough) for specifying bss and heap plus stack sizes.

The major concern, for programs written in C or Pascal, is that the events at program startup don't mesh well with the high level language environment. Control is simply transferred to the start of the text section, with some pointers to strings pushed on the stack. This must be translated into a call to the C function main, with information on the argument strings as arguments to main. And if main returns, its return value must be used as the status to provide on a call to exit. Such minor adaptation is performed by a small runtime startup sequence, coded in assembler, that is carefully fed first to link, so that it ends up at the start of the text section.

System calls are all packaged as separately compiled modules, each callable from C, which are collected into ordered libraries. Libraries are constructed by the programming utility lib, whose file format is known to link. Moreover, link presumes that any library file it encounters, among its file arguments, contains modules that are to be loaded only if there are pending references to undefined symbols, which references can be satisfied by the library module in question.

What this means is that a large corpus of modules can be assembled, each of which is occasionally useful but all of which, taken together, would be too much of a burden to inflict on every program. The programming utility rel can be used to produce a module-by-module list of symbols defined and symbols required. This list is fed to the programming utility lord, which determines an ordering of the modules such that later ones in the sequence don't require earlier ones. The ordering produced by lord is fed as instructions to lib to construct a library, suitable for conditional loading by link.

Got that?

So the sequence of files presented to link is: header file to adapt to C environment, program specific object files, then one or more libraries of object modules implementing system calls and other generally usable functions.

All of this machinery is well hidden, by the way, by the compiler driver scripts provided with Idris. They see to it that, given simple instructions about the files involved, various source files get compiled to object modules; then they arrange a call to link which brings all the necessary items together to build a program file.

SEE ALSO

The programming utilities are documented in Section II of the C Interface Manual for each target machine.

NAME

Compile - using the multi-pass compiler driver

FUNCTION

Compilers under Idris each consist of three or more separate programs, which must be run in sequence, and which communicate via intermediate files. The multi-pass command driver, c, standardizes the way in which the components of a given compiler are run, and eases the process of changing standard parameters for existing compilers, or extending them, for new ones.

Any driver meant to automate the invocation of a multi-pass program like a compiler should perform at least the following tasks:

1) naming the programs (such as compiler passes) to be run, and running them in the correct sequence.

2) specifying invariant parameters, such as flags, that are to be passed to a given program every time it is run in this particular application.

3) associating user-specified parameters, such as optional flags or source filenames, with at least the first program in the series to be run.

4) passing the output from one program to the next program in sequence, which must accept it as input.

5) end processing, which must include at least the naming of the final output file so that the user (or a later processor) can easily find it, and which might include the running of an optional linker or loader program over all of the files earlier processed.

c performs these tasks by reading an ordinary textfile, or "prototype script", containing a series of prototype command lines, each of which names one of the programs c is to run. Usually, a separate prototype script is provided for each kind of compilation to be done, and each script is given a mnemonic name, with the suffix ".proto". By convention, the script to perform native C compiles for the host machine is called "c.proto"; one to do Pascal cross-compilation for the 8080 might be called "pcx80.proto", and so on. The names can be anything at all; only the ".proto" suffix is normally required, and even that convention can be overridden when c is run.

Given the existence of these scripts, c is then installed by creating one link to it (i.e., an "alias") corresponding to each script, and sharing the same name. When c is run, it examines the alias under which it was invoked, and (by default) tries to find a corresponding prototype script. which it looks for using the same rules by which the shell searches for a command. In the standard case, what happens is this:

1) the user types to the shell one of the aliases under which c is installed (like c or pcx80), followed by the names of the source or object files to be compiled or linked.

2) if the alias given contained a slash (i.e., if it was a pathname), the shell tries to execute c under exactly that name; otherwise, it looks for a file with that name in each directory on the user's execution path.

3) once the specified alias for c is found, c is run, and examines its command line to discover what alias was given.

4) by default, c then appends ".proto" to the given alias, and mimics the shell: if the resulting name contains a slash, then c tries to read a script having exactly that filename; otherwise, it looks for a file with that name in each directory on the execution path, and reads the first one that it finds.

Most of the time, both the aliases for c and the corresponding scripts reside in the same system-wide binary directory (such as /etc/bin). However, c permits two other arrangements, which can be quite powerful if carefully used. Additional links to c may exist in other (perhaps private) directories, with their own scripts; or local scripts may be created with the same name as "official" ones, and privately installed. So long as these private directories are placed on the user's execution path, they will be read automatically whenever c is invoked.

Once the script has been located, c executes in sequence the programs named on successive lines, passing to the first program the name of one of the files that the user originally gave to c, then passing to each subsequent program the name of a temporary file produced by the previous program. This process is repeated for each filename the user specifies.

A line in a script has one of the following formats:

```
<prefix> : <pname> <pargs> [ : <pargs> ]
```

or

```
<prefix> :: <pname> <pargs> [ : <pargs> ]
```

In either format, <prefix> is a string of characters that is matched against the suffix of each command line filename (i.e., the characters in the name following the rightmost dot). Thus a prefix of 'c' would match filenames ending in ".c", 'o' would match names ending in ".o", and so on. Some conventional suffixes are:

Suffix	Type of File
.p	Pascal source
.c	C source
.s	assembler source
.o	object file (assembler output)

However, any non-null string may be used to suffix input files, and be given as a prefix; multi-character strings are perfectly acceptable.

The rest of each prototype line is used to construct a command line that c will execute for each matching input file. <pname> specifies the name (generally the full pathname) of the program to be run. The first group of <pargs> is one or more strings that will be used as the opening arguments of the command line executed. The first string becomes argument zero, the program name actually passed to the program. The colon following, if present, marks the point at which each user-specified filename will be inserted in the command line, while the second group of <pargs> is zero or more strings that will follow the filename on the command line. If the colon and second <pargs> are omitted, then each input filename is appended to the command line. Here, for instance, is a typical script for native C compilation on a PDP-11:

```
c:/etc/bin/pp   pp -x -i /lib/¦../
:/odd/p1    p1
:/odd/p2.11 p2
s:/etc/bin/as.11    as.11
o::/etc/bin/link    link -et_etext -ed_edata -eb_end -lc.11 /lib/Crts.11
```

For each file specified by the user, c searches the script for a line whose prefix matches the file's suffix. It inserts the filename as instructed into the matching line, then starts executing the commands in the script from that point; any previous lines are skipped. The script shown might begin execution at any of three places, depending on the suffix of each input file. For ".c" files, it would start at the first line (which has the matching prefix 'c'), while for ".s" files it would begin at the fourth line, and for ".o" files at the last one.

Thus, if this script were installed at c.proto, and the user typed:

```
% c prog.c
```

the first line of the script would match the input file, and c would create, then execute, the following command line:

```
pp -o /tmp/xxxxxc1 -x -i/lib/¦../ prog.c
```

c adds the flag -o in order to specify the name of a file in which the program being run will place its output (which means that any program used under c must accept the -o convention). If another program is still to be run, then this filename is passed to it as input. Here, if pp returned success, p1 would be executed, with this command line:

```
p1 -o /tmp/xxxxxc2 /tmp/xxxxxc1
```

Then, presuming p1 returned success, its output file would be passed to the following program, and so on. If a program being run returns failure, then the rest of the script is skipped for the current file, and c moves on to the next file given on its command line.

Assembler source files could also be assembled with this script. For each ".s" file given, c would skip the first three prototype lines, and begin by executing the fourth one, thus running as.11, but not any of the compiler passes.

By default, c acknowledges the start of processing of each input file by outputting the filename to STDOUT, followed by a colon and a newline. When (and if) the link line program explained below is run, c outputs its name in the same manner. The flag -v causes c to output each command line generated as well, as it is executed. This verbose option can be used to double-check new scripts, or just to see exactly what command series c generates for a given input file:

```
% c -v prog.c
```

The final line in the script shown is a special one, called a "link" line, as indicated by the double colon following its prefix. The program named on a link line is not run for each command line file. Instead, the output files resulting from each command line file are stored on the link line; then the link line is executed after all the command line files have been processed, presuming that all of the processing succeeded. If any execution of a previous program failed, then the link line program is not run.

The link line accumulates three kinds of files: those output by the preceding program in the script, command line files whose suffix matches its prefix, and any files that are unmatched by the other prototype lines. Hence, the line in the script above would accumulate all the files output by as.11, and ".o" files named by the user, as well as any files with names not ending in ".c" or ".s".

A link line has one other special characteristic, namely that each file output by the program immediately preceding it is made permanent, rather than being temporary like the files output by prior programs. The permanent file output is given the same name as the input file originally specified by the user, but with its suffix changed to be the prefix of the link line.

For instance, the use of c shown above would normally result in as.11 being executed with the command line

```
as.11 -o prog.o /tmp/xxxxxc1
```

The name of the assembler's output file would be taken from the input file prog.c, and its suffix, from the prefix 'o' of the link line.

The link line thus serves as a "terminus" for all previous processing, since it causes the final output from each command line file to be separately stored in a permanent file. Because of this, and because it accumulates all files that "fall through" to it from prior lines in the script, any link line present must be the last line in a script. However, any line in a script having a non-null prefix can be treated as a terminus for a particular run, simply by giving its prefix to c. Then, each time the preceding line is executed, its output will be directed to a permanent file with a suffix given by the prefix of the terminus. c then skips any lines remaining in the script, and starts processing its next input file.

If, for example, the above script were run by typing:

```
% c +s prog.c
```

The line labelled 's' would be considered the terminus of the pass through the script, and so the preceding line would create a permanent output file named prog.s, which would contain the symbolic assembly code generated for prog.c. Neither of the last two script lines would be executed. Similarly, typing:

```
% c +o prog.c
```

would cause execution of the script to stop with the line before the link line, which would create a permanent output file named prog.o containing the object code resulting from prog.c.

Note that c expects to submit each of its input files to at least one prototype command. An input file not matching one of the commands preceding the current terminus will cause an error.

Any prefix can play the additional role of execution terminus. In fact, another type of script is one whose final line contains only a terminating prefix, and thus serves solely to provide a suffix for the output files of the preceding line. Such scripts are useful for compilations in which linking together the individual output files is not desirable. For instance, a script to cross-compile C source files under Idris for assembly and loading under VERSAdos might look like:

```
c:/etc/bin/pp   pp -x -dUTEXT -i /lib/¦../
:/odd/p1    p1 -al -n8 -u
:/odd/p2x.68k   p2
s:
```

Here, only one usage of the script is possible: each ".c" file given will be processed by the three compiler passes, and the output from each run of p2 will be directed to a corresponding ".s" (VERSAdos assembler source) file.

In addition to the compile-only and link-line scripts presented so far, scripts can be used whose last line is a full prototype line, but not a link line. When such a script is run, the output from the last program is directed to a temporary file, and so the script produces no permanent output files. For example, a script to do C syntax checking only, with no code generation, might be:

```
c:/etc/bin/pp   pp -x -i/lib/
:/odd/p1    p1
```

Since c was designed to automate compilation, it contains features to aid the orderly generation of output files. The flag -o may be used to name the output file of the link line program specified by a given script; and -p may be used to specify a pathname that will be used to prefix all the permanent output files from a given run. Thus the command

```
% c -o prog prog.c
```

would cause the executable file produced by link to be given the name prog, while

```
% c +o -p /ship/c780/obj/ *.c
```

would compile each of the ".c" files in the current directory into a corresponding ".o" file in the directory /ship/c780/obj, permitting the source directory to be treated as read-only. If both -o and -p are given, the prefix is also added to the link line output filename, if the name doesn't already contain a slash. Hence, the command

```
% c -o prog -p/ship/c780/obj/ prog.c
```

would place the executable output of link in the file /ship/c780/obj/prog (and would create /ship/c780/obj/prog.o along the way).

Finally, the normal lookup procedure for a prototype script can be overridden by naming with a -f flag exactly the script desired for a particular run. A script named in this way need not have the suffix ".proto":

```
% c -f myscript prog.c
```

If the script name given is "-", then a script will be read from STDIN.

The most effective way to learn about c is simply to use it, perhaps with -v specified so that its internal operation can be seen. With the command driver, the flexible use of existing compilers is trivial, while the use of new compilers depends solely on setting up a new prototype script. And because scripts are ordinary textfiles, existing ones can be varied at will to obey local conventions. Most important of all, numerous compilers can be run in exactly the same way, with the same options and conventions. Learning c once is learning it for all of them.

BUGS

c does not currently permit flags to be passed to the programs named in a prototype script. The only way to change the flags with which one of the named programs is run is to use a changed version of the script.

NAME

Debug - using the binary editor db

FUNCTION

db is an interactive editor for binary files that is loosely patterned after the text editor e. It is most useful in working with relocatable or executable object files in standard format, but can also be used to manipulate any direct-access file. db makes available different sets of commands and addressing facilities according to what "mode" it is used in:

1) "absolute" mode, to examine or modify an arbitrary file;

2) "standard" mode, to inspect or patch a relocatable or executable file in standard format; or

3) "debug" mode, to inspect a core image generated by Idris, in conjunction with the executable file whose contents it represents.

This essay presumes some familiarity on the reader's part with the commands and addressing conventions of e; where db provides close parallels to them, neither instance will be discussed at length here. Nor will this essay provide a complete summary of all the commands db makes available (which can be found on the db manual page). Rather, the emphasis will be on how db differs from e, and in particular, how it can be used effectively as a truly portable binary editor and debugger.

Like e, db reads a series of single-character commands from STDIN, and writes any output to STDOUT. Just as commands in e may be preceded by line addresses, db commands may be preceded by byte addresses, whose syntax is a superset of e address syntax. A byte address refers to the start of an item, which may be an integer of length one, two, or four bytes, or a disassembled machine instruction. db supports a large subset of the editing commands provided in e, except that they operate on such items, instead of on lines of text. Given this underlying difference, the two editors in fact have very similar user interfaces. In db, editing generally centers around a "current item" (addressed by a dot '.'), the counterpart of the current line in e. And in general, the addresses preceding a given command designate the range of items on which it will operate.

In absolute mode, db considers its input file to be a series of items, of arbitrary length, which may be updated at will. In standard mode, db honors standard object file format, meaning that it restricts the scope of updates to the file being edited, but makes full use of any available symbol table or relocation information in interpreting addresses. In debug mode, db operates similarly, and also provides a set of commands to access run-time information saved in the core file, such as the machine registers and user stack.

Initialization

The mode in which db operates is established at the start of each run. Normally, the file to be edited is specified on the command line, perhaps preceded by flags detailing how it is to be accessed. If the flag -a is

given, or if the command line file cannot be interpreted as an object file in standard format, db comes up in absolute mode, and outputs a heading line noting the fact.

Otherwise, if -c is given, db operates in debug mode, and expects the command line file to be a standard object file; the associated core file must reside in the current directory, under the name "core". In debug mode, db usually accesses only one of the two input files at a time; initially, db accesses the core file.

If -c is not given and the command line file is in standard format, db operates in standard mode (and so edits an object file alone). In debug mode or standard mode, db outputs a header giving the length (in decimal) of the text, data and bss segments of the object file being edited.

For instance, either of the following exchanges would open a disk image directly for editing:

```
% db /dev/rm0
/dev/rm0: absolute
```

or:

```
% db -a /dev/rm0
/dev/rm0: absolute
```

While the following would edit the executable file myprog, and the core file it generated when run:

```
% db -c myprog
myprog: 10438T + 1124D + 0B
```

External Interface

Unlike e, db does not make an internal copy of the file(s) being edited. Instead, it interacts directly with the permanent version of each file, meaning that any changes are made immediately to that version. This also means that db can edit files of arbitrarily large size. In particular, files edited in absolute mode have no pre-defined maximum size; an attempt to access a given item succeeds if the necessary I/O does.

With this significant difference, db supports a set of absolute mode commands for interacting with external files that is analogous to the set e provides. The command 'e', followed by a filename, closes the file currently open for editing and opens the file named. The new file will always be accessed in the same mode as the previous one. The command 'r', followed by a filename, overwrites some part of the current file with the contents of the file named. The command may be preceded with an address specifying the first item to be overwritten; if no address is given, the external file overwrites the current file starting at the current item.

The command 'w', followed by a filename, writes some part of the current file to the file named, which the command always newly creates. The command may be preceded by one or two addresses specifying the range of bytes

that will be output; by default, all of the current file is output. The file named in the 'r' or 'w' commands may not be the file being edited. Finally, the command 'f' may be used to display the name of the current file.

Of these commands, only 'f' may be used in standard mode or debug mode, except that db always permits one initial 'e' command to be given.

Addressing

In absolute mode, the byte addresses used to access the input file closely correspond to the line addresses used in e. Only two differences exist: bytes are numbered starting at zero, and there is no pre-defined "last byte", so that the symbol '$' has only part of its meaning in e. In general, \&'$' can be used only as the second of two addresses specifying a range of items, and can never appear alone.

Byte addresses themselves (as their name implies) designate nothing more than byte offsets within the file. Depending on the current input/output format, the item itself may be of varying length, but the address itself is always counted in bytes.

In debug mode or standard mode, addressing is more complicated. Interacting with an object or core file means examining particular locations within the text or data segment the file contains, and not merely looking at a physical location inside the file. So in these cases, an address has two components: first, an indicator of which segment (text or data) it refers to, and second, the runtime location in "memory" that a given item would start at, if the object file were actually loaded.

db obtains the biases of the text and data segments from the standard file header, whence it also gets their lengths; any bss segment is presumed to immediately follow the data segment. Two ranges of addresses are then accessible: those in the text segment, which extend from the text bias up for as many bytes as the segment is long, and those in the data/bss segment, which extend similarly upward from the data bias. An access to a given address is mapped to the physical offset in the object or core file to which the address corresponds; the segment component of an address unambiguously indicates to which range it belongs.

In an object file, reading an address in the bss region returns all zeroes, and addresses in that region cannot be written to. In a core file, the data/bss segment is interpreted differently. The data segment extends from the core file data bias up to the "system break" (i.e., the highest dynamically-allocated storage) in use at the time of the dump. The bss segment extends from the system break up to the end of the user process image (i.e., the base of the runtime stack, which extends downward from the "top" of memory).

All residents currently allocate to a process a single swatch of memory, so that its data segment immediately follows its text segment. No overlap or gap between the two ranges of addresses is possible. Object and core files under an 'R' resident will always have a text bias of zero, since exec-time relocation is unnecessary. Under a 'B' or 'S' resident,

however, core files will always have a non-zero text bias, which is the actual location at which the process ran, since the resident makes no use of memory management hardware. And object files may be linked with a non-zero text bias for reasons of efficiency.

When the current output format is machine instructions, these complications are largely hidden from the user. If a symbol table and relocation streams are available in the object file, the disassembler will always output relocatable addresses as a symbol name plus a decimal byte offset, so long as a symbol exists with a value less than or equal to the address, and the same relocation base. If no relocation information is present, any symbol table available will still be used, but constants imbedded in the instruction stream may be mistaken for addresses, since db will have no way to tell them apart.

In debug mode, db presumes that the text and static data areas of a core file are "parallel" to those in the original object file, so it is able to find relocatable addresses in the core file by using the object's relocation streams. If the core file has a different text bias than the object file, the offset between them will be applied as needed to make the object file symbol table usable for core file address references. db will not tell an out and out lie -- if an address from such a core file cannot be output symbolically, then db will output its numeric value as stored in the file. But if the address, when altered, can be output symbolically, then the symbolic form will be used.

Address Terms

Addresses in db are composed of one or more address terms, which may be combined by the same rules e uses. However, db provides a slightly expanded set of terms. First, in standard or debug mode, \&'$' always refers to the last address of the current segment, text or data/bss. Since in this context '$' has a well-defined value, it can be used by itself, though items cannot be read past the end of a segment.

Also, when a symbol table is available in the object file being edited, db permits symbols to be specified as address terms. A symbol specifies the address location indicated by its value, and has the segment component indicated by its relocation base. A text-relative symbol refers to the text segment; data or bss-relative symbols refer to the data/bss segment.

Not all address terms have explicit segment components. An offset from '.' or '$' refers to the same segment as dot currently does. A symbol that is not text, data, or bss-relative, or a number appearing alone as an address, refers to the same segment as the last previous address given for the same command, or to the segment that dot refers to, if no previous address was given.

In both standard and debug mode, db provides three constants to simplify the use of object or core file addresses. The constant 'T' has the value of the first address in the current text segment, while 'D' and 'B' have the value of the start of the data segment and bss segments, respectively. All three also have the appropriate segment component. Two main uses for these constants are to ease access to an item at a given offset from the

start of a segment, or to force the current item address (dot) into one segment or the other, so a subsequent command will work properly. Thus to look at an item 10 bytes into data segment, you might type:

```
D10 1
```

while to set dot to the start of the text segment, and then print several instructions, you could use:

```
T:+50 pm
```

Finally, a note about numbers: db outputs all numbers that are part of an address using the conventions of the language C. That is, numbers in hexadecimal are preceded by "0x", numbers in octal by "0", and numbers in decimal by nothing at all. The same conventions are used for numbers input as address terms. However, these conventions apply only to addresses; the contents of an item are output without a prefix, even when output in hexadecimal or octal. Likewise, items must be input without C-style prefixes. (See Editing, below.)

Items

Items are the units into which db divides the data in its input file, which means they're also the units on which many editing commands operate. However, an "item" is not defined statically, but in terms of the current input/output format, which can be changed at will. This format specifies the length of an item (one, two, or four bytes, or the word length of the target machine), and defines how each item is to be converted on input or output. Items can be treated as integers in the three common bases, or as strings of ASCII characters. Alternatively, items may consist of disassembled machine instructions, output in conventional symbolic form. An input/output format is named as a base code followed by a size code:

Base code	Item output as:	Size code	Item length:
a	ASCII characters	c	char (1 byte)
h	hexadecimal	s	short (2 bytes)
o	octal	i	int (2 or 4 bytes)
u	unsigned decimal	l	long (4 bytes)
m	machine instruction		

A base code may be given with no size, in which case the size defaults to int; or a size code may be given with no base, causing the base to be taken as signed decimal. The base code 'm' may not be followed by a size, since the size of each item in this format is just the length of the disassembled instruction. Thus a format of "al" defines long items output as strings of ASCII characters, while a format of 'h' calls for int items to be output as hexadecimal integers, and 's' calls for short items output in signed decimal. Note that the base code 'a' converts characters like the 'l' command in e. Characters with values in the range [0, 7] are output as "\[0-7]", those in the range [8, 13] are output as "\[btnvlr]", and any other character that is not printable ASCII is output as a '\' followed by the three-digit octal number giving its value.

Editing

Items can be examined with the db commands 'l' and 'p', which are rough analogues to their namesakes in e. In db, the difference between the two is that 'p' (for "print") sets the current item address (dot) equal to the last location examined, while 'l' (for "look") leaves dot unchanged. The current input/output format may be changed by following either command letter with the appropriate base and size codes. For instance, to print three shorts in hexadecimal, starting at the current item address:

```
.,+5 phs
0x100   81cc
0x102   0002
0x104   1401
```

When, as here, items are treated as integers, db converts as necessary from the native byte order of the target machine. When items are being output in ASCII, however, each item is treated as a array of characters, which are output in order of increasing index in the array. Thus, depending on the target machine, the order of the characters output may differ from the order in which the same bytes would appear in an integer output format. Here the results of four examinations of a long written for a PDP-11 (using db11):

```
. lal
0x200   abcd

. lhl
0x200   62616463

.,+3 lhs
0x200   6261
0x202   6463

.,+3 lhc
0x200   61
0x201   62
0x202   63
0x203   64
```

Note that the '.' preceding the first two 'l' commands is unnecessary, since by default 'p' or 'l' display only the current item.

Items may be changed with the commands 's' and 'u'. The 'u' command (for "update") is the single db equivalent of the e commands 'a', 'c', and 'i'. It replaces some series of items in the file being edited with items read from STDIN using the current input/output format. Items are read as ordinary text strings, one per line, until a '.' is read on a line by itself. Each line may contain only characters legal in the current format; each is converted to exactly one item of the correct length. Again, prefixes are not recognized here, so items entered in hexadecimal should not be preceded with "0x" (nor octal ones with a "0").

db will convert integers to native byte order before writing them to the file. Integer values too large to fit in the current item length will be truncated without comment. If the input format is ASCII, however, a line specifying too many characters to fit in an item will not be accepted. Also, if input is given in this format, a line containing too few characters to fill an item will be right padded with NULs.

The 's' command closely resembles its counterpart in e. It is followed by a target string to be searched for, and a second string that will replace any occurrences of the target. As in e, the target may be a regular expression. The command then converts each item it examines to output representation (using the current input/output format), replaces any occurrences of the text of the target string with the text of the replacement string, and converts the result back to internal format. In that form, it replaces the original item, by the same rules as given above for the 'u' command. Note that in db the 's' command always outputs the revised item that contained the last instance of the target string. (That is, in terms of the 's' command in e, the command in db always acts as if it were suffixed with 'p'.)

Items cannot be updated as machine instructions, since the disassembler has no counterpart for interpreting input assembly code. Hence, neither 's' nor 'u' may be used if the current input/output format is machine instructions.

NAME

Headers - standard include files

SYNOPSIS

```
#include <std.h>
#include <sys.h>
   etc.
```

FUNCTION

Information to be shared among C source files is conventionally concentrated in one or more "header" files, to be #include'd as needed at the top of each source file. All such files provided with Idris are in the standard library /lib, and the various C compiler drivers are wired to search /lib for any include files written in angle brackets.

Of all these files, std.h is by far the most important. It is included in all source files for the compilers and Idris. It defines the notation used throughout these manuals. And it provides definitions used by all the other header files. (See Section II of the C Programmers' Manual for a description of its contents.) Consequently, it should be included in every source file written to interact with the standard C environment, and it should be named before any other files.

The file sys.h plays a similar role, for programs that must interact with the Idris resident. It defines all the constants and data structures that cross the interface between programs and the resident. And it provides definitions used by the more specialized header files. (See Section II of this manual for a description of its contents.) Thus, it should be included in every source file written to interact with the Idris environment, and it should be named right after std.h.

The complete list of header files provided with Idris is:

bio.h block I/O within the resident. Used by some device handlers.

cio.h character I/O, primarily tty style, within the resident. Used by some device handlers.

cpu.h process management, within the resident. Used to interpret files such as /dev/myps and /dev/ps.

dump.h dump file format. Used by dump and restor.

fio.h file I/O, within the resident. Used to interpret files such as /dev/inodes and /dev/mount.

ino.h filesystem format. Used by programs that must manipulate filesystems directly.

mount.h mount file format. Used by programs that must manipulate the external mount table.

pan*.h panel layout for registers within a core file, dumped by the resident. Used by the resident and db to interpret core dumps. There is

a different file for each target machine, such as pan11.h, pan80.h, etc.

pascal.h file I/O, within the Pascal runtime. Used by Pascal runtime library to administer files.

pat.h pattern matching codes. Used by programs that use the standard library functions amatch, makpat, and match to match regular expressions.

res.h resident conventions. Used in all resident source files to define system wide constants and types. Sort of an internal sys.h.

sh.h shell conventions. Used in all shell source files, and by programs that invoke other programs via the standard Idris functions xecl and xecv.

sort.h sort key conventions. Used by programs that use the standard library function getkeys to define sort keys.

std.h the universal header file, as described above. Used to implement the standard C environment.

sup.h filesystem format. Used by resident source files that must manipulate the filesystem directly. Sort of an internal ino.h.

swp.h timer and space management structures, within the resident. Used by various Idris schedulers.

sys.h the Idris interface header file, as described above. Used to implement the standard Idris environment.

time.h time and date structure. Used by the Idris support library functions atime, ltime, and vtime to manipulate time information.

usr.h per-process information swapped with the process image, within the resident. Used by resident source files that must manipulate the process image.

who.h log and who file formats. Used by programs that must manipulate the administrative files /adm/log and /adm/who.

Most of these files can be ignored most of the time. But when a program must interact with an existing corpus of code, including the relevant header files is a safe and convenient way to enforce the necessary conventions.

SECTION TWO

IDRIS SYSTEM INTERFACE

NAME

Interface - Idris system interface

FUNCTION

The functions in this section define the Idris system interface that is visible to a C program, regardless of target machine. It is actually the union of two interfaces:

1) the standard portable C system interface, as documented in Section III of the C Programmers' Manual, and

2) the system calls supported by all Idris implementations, provided as C callable functions. Each Idris Interface Manual describes the actual machine level system call formats for a given target machine.

By writing in C, and observing the conventions outlined in this section, the programmer can be assured that code written for one Idris implementation should work on all current and future implementations.

TYPES

A number of special data types are used to help document the Idris system interface. The more heavily used of these are defined in the standard system header file sys.h. The special types are:

DEV an unsigned short integer, whose more significant byte is the "major number" of a physical device and whose less significant byte is the "minor number". The major number is used to index into one of two resident device tables (for block special or character special devices) to select a device handler. The device number DEV is tucked in an odd corner of a block or character special inode, is provided as part of the file status delivered up for stat (or fstat), and is passed to the selected device handler in case it deals with multiple devices. Note that a device number is written on disk less significant byte first, regardless of target machine.

DIR a directory link entry, consisting of a two-byte inode number and a 14-byte link name. No system call delivers up a DIR, but such creatures are frequently read when scanning directories. Note that the inode number is written on disk less significant byte first, regardless of target machine. Defined in <sys.h>.

ERROR a short integer, capable of holding any error return code from the resident. Zero or a positive number usually indicates success; Idris error code is usually negated. Defined in <sys.h>.

PID a short integer, capable of holding any processid. Valid processids are always positive, nonzero.

SIG a character, capable of holding any signal.

STAT the file status returned by the fstat or stat system calls. Defined in <sys.h> and described along with fstat.

TTY the tty status returned by gtty and expected by stty. Defined in <sys.h> and described along with gtty.

UID an unsigned character, capable of holding any userid or groupid.

Extensive use is also made of two more conventional C types:

FILE a short integer, capable of holding a negated error code or any valid file descriptor returned by Idris.

TEXT * a pointer to character, usually to the first of a NUL-terminated string of characters variously known as a string, filename, or pathname. Note that this declaration is sometimes used merely to indicate a pointer with unknown storage boundary constraints, as an arbitrary user memory address.

ERROR

An important group of system parameters is the error codes. These are returned, when a system call fails, to indicate the general nature of the failure. Most system calls can return a variety of error codes, and most error codes can be returned by a variety of system calls, so no attempt is made to correlate the two groups. The error codes are:

```
E2BIG (7)  argument list too big for exec.
EACCES (13)  file access prohibited.
EAGAIN (11)  exec or fork failed, but may work if you try again.
EBADF (9)  bad file descriptor.
EBUSY (16)  can't mount or unmount busy filesystem.
ECHILD (10)  no children to wait for.
EDOM (33)  math function argument outside defined domain.
EEXIST (17)  file already exists.
EFAULT (14)  can't access argument on system call.
EFBIG (27)  file too big.
EINTR (4)  system call was interrupted.
EINVAL (22)  illegal argument value on system call.
EIO (5)  unrecoverable physical I/O error.
EISDIR (21)  file is a directory.
EMFILE (24)  too many files opened by process.
EMLINK (31)  too many links to a file.
ENFILE (23)  no system storage left to represent opened file.
ENODEV (19)  not a defined operation for the device.
ENOENT (2)  directory entry does not exist.
ENOEXEC (8)  wrong file format for exec.
ENOMEM (12)  not enough memory to exec program or to grow heap.
ENOSPC (28)  no space on filesystem.
ENOTBLK (15)  not a block special device.
ENOTDIR (20)  not a directory.
ENOTTY (25)  not a tty.
```

ENXIO (6) nonexistent I/O device.
EPERM (1) user lacks permission.
EPIPE (32) write to broken pipe.
ERANGE (34) math function result outside representable range.
EROFS (30) read-only filesystem.
ESPIPE (29) can't seek on a pipe.
ESRCH (3) can't find process to signal.
ETXTBSY (26) shared text portion of file is in use by exec.
EXDEV (18) can't link across filesystems.

SIGNALS

There are also a number of signals that can be sent to processes, for standard system action or for handling by the program itself. These are:

SIGALRM (14) alarm clock timeout.
SIGBUS (10) bus error, as for nonexistent memory.
SIGDOM (7) domain error for a math function.
SIGFPT (8) floating point arithmetic error.
SIGHUP (1) hangup, as when dataphone disconnects.
SIGILIN (4) illegal instruction.
SIGINT (2) interrupt, as when DEL is typed.
SIGKILL (9) kill request.
SIGPIPE (13) broken pipe.
SIGQUIT (3) quit, as when ctl-\ is typed.
SIGRNG (6) range error for a math function.
SIGSEG (11) segmentation, or memory protection, violation.
SIGSYS (12) bad system call.
SIGTERM (15) terminate request, a catchable variant of kill.
SIGTRC (5) instruction execution trace.

Along with the signals is defined the special user memory address: NOSIG (1) signal is to be ignored.

NOSIG is to be distinguished from the other special pointer value, NULL, which calls for standard system handling of a signal. Any other pointer value is used as the address of a function to handle the signal in question. See the manual page on signal for more information.

BUGS

All of the types used in this section should appear in <sys.h>.

NAME

Conventions - Idris system subroutines

SYNOPSIS

#include <sys.h>

FUNCTION

All standard system library functions callable from C follow a set of uniform conventions, many of which are supported at compile time by including a standard header file, <sys.h>, at the top of each program. Note that this header is used in addition to the standard header <std.h>. The system header defines various system parameters and a useful macro or two.

Herewith the principal definitions:

DIRSIZE - 14, the maximum directory name size
E2BIG - 7, the error codes returned by system calls
EACCES - 13
EAGAIN - 11
EBADF - 9
EBUSY - 16
ECHILD - 10
EDOM - 33
EEXIST - 17
EFAULT - 14
EFBIG - 27
EINTR - 4
EINVAL - 22
EIO - 5
EISDIR - 21
EMFILE - 24
EMLINK - 31
ENFILE - 23
ENODEV - 19
ENOENT - 2
ENOEXEC - 8
ENOMEM - 12
ENOSPC - 28
ENOTBLK - 15
ENOTDIR - 20
ENOTTY - 25
ENXIO - 6
EPERM - 1
EPIPE - 32
ERANGE - 34
EROFS - 30
ESPIPE - 29
ESRCH - 3
ETXTBSY - 26
EXDEV - 18
NAMSIZE - 64, the maximum filename size, counting NUL at end
NSIG - 16, the number of signals, counting signal 0
SIGALRM - 14, the signal numbers
SIGBUS - 10

```
SIGDOM - 7
SIGFPT - 8
SIGHUP - 1
SIGILIN - 4
SIGINT - 2
SIGKILL - 9
SIGPIPE - 13
SIGQUIT - 3
SIGRNG - 6
SIGSEG - 11
SIGSYS - 12
SIGTERM - 15
SIGTRC - 5
```

The macro isdir(mod) is a boolean rvalue that is true if the mode mod, obtained by a getmod call, is that of a directory. Similarly isblk(mod) tests for block special devices, and ischr(mod) tests for character special devices.

NAME

_pname - program name

SYNOPSIS

```
TEXT *pname;
```

FUNCTION

_pname is the (NUL terminated) name by which the program was invoked, as obtained from the command line argument zero. It overrides any name supplied by the program at compile time.

It is used primarily for labelling diagnostic printouts.

NAME

brk - set system break to address

SYNOPSIS

```
TEXT *brk(addr)
    TEXT *addr;
```

FUNCTION

brk sets the system break, at the top of the data area, to addr. Addresses between the system break and the current top of stack are not considered part of the valid process image; they may or may not be preserved. It is considered an error for the top of stack ever to extend below the system break.

RETURNS

If successful, brk returns the old system break; otherwise the value returned is -1.

EXAMPLE

```
if (brk(end + nsyms * sizeof (symbol)) == -1)
    {
    putstr(STDERR, "not enough room!\n", NULL);
    exit(NO);
    }
```

SEE ALSO

sbreak

NAME

chdir - change working directory

SYNOPSIS

```
ERROR chdir(fname)
    TEXT *fname;
```

FUNCTION

chdir changes the working directory to fname.

RETURNS

chdir returns zero if successful, else a negative number, which is the Idris error return code, negated.

EXAMPLE

```
chdir("/tmp");
```

NAME

chmod - change mode of file

SYNOPSIS

```
ERROR chmod(fname, mode)
    TEXT *fname;
    BITS mode;
```

FUNCTION

chmod changes the mode of the file fname to match mode. Only the low order twelve bits of mode are used, to specify ugtrwxrwxrwx, where u is the set userid bit, g is the set groupid bit, t is the save text image bit, and the rwx groups give access permissions for the file. Access permissions are r for read, w for write, and x for execute (or scan permission for a directory); the first group applies to the owner of the file, the second to the group that owns the file, and the third to the hoi polloi.

RETURNS

chmod returns zero if successful, else a negative number, which is the Idris error return code, negated.

EXAMPLE

To make a file executable:

```
chmod("xeq", 0777);
```

SEE ALSO

getmod

NAME

chown - change owner of file

SYNOPSIS

```
ERROR chown(fname, owner)
    TEXT *fname;
    UCOUNT owner;
```

FUNCTION

chown changes the owner of file fname to be the less significant byte of owner, and changes its group to be the more significant byte of owner. Only the superuser succeeds with this call.

RETURNS

chown returns zero if successful, else a negative number, which is the Idris error return code, negated.

EXAMPLE

To give ownership of a file to the person who invoked you:

```
chown(newfile, getuid() | getgid() << 8);
```

SEE ALSO

getgid, getuid

NAME

close - close a file

SYNOPSIS

```
ERROR close(fd)
    FILE fd;
```

FUNCTION

close closes the file associated with the file descriptor fd, making fd available for future open or create calls.

RETURNS

close returns zero, if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

To copy an arbitrary number of files:

```
while (0 < ac && 0 <= (fd = open(av[--ac], READ, 0)))
    {
    while (0 < (n = read(fd, buf, BUFSIZE)))
        write(STDOUT, buf, n);
    close(fd);
    }
```

SEE ALSO

create, open, remove, uname

NAME

creat - make a new file

SYNOPSIS

```
FILE creat(fname, perm)
    TEXT *fname;
    BITS perm;
```

FUNCTION

creat makes a new file with name fname, if it did not previously exist, or truncates the existing file to zero length. In the former case, the file is given access permission specified by perm; in the latter, the access permission is left unchanged. Access permissions are described under chmod. The file is opened for writing.

RETURNS

creat returns a file descriptor for the created file or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if ((fd = creat("xeq", 0777)) < 0)
    putstr(STDERR, "can't creat xeq\n", NULL);
```

SEE ALSO

chmod, close, create, open, remove, uname

NAME

create - open an empty instance of a file

SYNOPSIS

```
FILE create(fname, mode, rsize)
    TEXT *fname;
    COUNT mode;
    BYTES rsize;
```

FUNCTION

create makes a new file with name fname, if it did not previously exist, or truncates the existing file to zero length. An existing file has its permissions left alone; otherwise if the filename returned by uname is a prefix of fname, the (newly created) file is given restricted access (0600); if not, the file is given general access (0666). If (mode == 0) the file is opened for reading, else if (mode == 1) it is opened for writing, else (mode == 2) of necessity and the file is opened for updating (reading and writing).

rsize is the record size in bytes, which must be nonzero on many systems if the file is not to be interpreted as ASCII text. It is ignored by Idris, but should be present for portability.

RETURNS

create returns a file descriptor for the created file or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if ((fd = create("xeq", WRITE, 1)) < 0)
    putstr(STDERR, "can't create xeq\n", NULL);
```

SEE ALSO

close, open, remove, uname

NAME

dup - duplicate a file descriptor

SYNOPSIS

```
FILE dup(fd)
    FILE fd;
```

FUNCTION

dup allocates a file descriptor that points at the same file, and has the same current offset, as the file descriptor fd. It is promised that the smallest available file descriptor is allocated on any creat, dup, open, or pipe call, so file descriptors can be rearranged by judicious use of dup and close calls.

RETURNS

dup returns the newly allocated file descriptor or a negative number, which is the Idris error return code, negated.

EXAMPLE

To redirect STDIN from fd:

```
close(STDIN);
dup(fd);
close(fd);
```

SEE ALSO

close, creat, create, open, pipe

NAME

execl - execute a file with argument list

SYNOPSIS

```
ERROR execl(fname, s0, s1, ..., NULL)
    TEXT *fname, *s0, *s1, ...
```

FUNCTION

execl invokes the executable program file fname and passes it the NULL terminated list of string arguments specified in the argument list s0, s1, etc. The invoked file overlays the current program, inheriting all its open files and ignored signals; the current program is forever gone. Signals that were caught by the current program revert to system handling.

If the set userid bit in the mode for fname is set, the effective userid of the invoked file becomes that of the owner of the file; the effective groupid may be changed in a similar manner by the set groupid bit. (The save text bit is currently ignored.)

The invoked program begins execution at the start of the text section, with the string arguments at the top of user memory, just above the stack, i.e., on the stack is initially pushed a NULL fence, followed by pointers to the stacked strings in reverse order, followed by a count of the number of strings passed as arguments. Thus, for a machine with two-byte integers:

```
0(sp)   is the count of arguments, typically > 0.
2(sp)   points to the zeroeth argument string
4(sp)   points to the first argument string, etc.
```

Note that the stack is not well conditioned for direct entry into a C function; the C runtime startup header is usually linked at the start of the text section. It enforces conventions such as setting _pname, isolating the execution search path, calling the C main function, and calling exit. Note also that the fence, placed at the end of the stacked argument strings, is -1 under UNIX/V6, and not NULL.

By convention, the zeroeth argument is always present and is taken as the name _pname by which the file is invoked; if it contains a vertical bar '|', the string before the first vertical bar is taken as argument zero and the string after that bar is taken as a search path, or concatenation of filename prefixes separated by vertical bars, used for locating executable files. Additional arguments are typically optional; their interpretation is left purely to the whim of the invoked program.

RETURNS

execl will return only if the file cannot be invoked, in which case the value returned is the Idris error return code, negated. Specifically, E2BIG means that too many argument characters are being sent, ENOMEM means that the program is too large for available memory, and ENOEXEC means that the program has execute permission but is not a proper binary object module.

EXAMPLE

```
execl("/bin/mv", file, direc, NULL);
putstr(STDERR, "can't exec mv\n", NULL);
exit(NO);
```

SEE ALSO

_pname, execv, exit, fork

BUGS

Only 512 characters of argument strings may be sent, counting terminating NULs.

NAME

execv - execute a file with argument vector

SYNOPSIS

```
COUNT execv(fname, args)
    TEXT *fname, **args;
```

FUNCTION

execv invokes the executable program file fname and passes it the NULL terminated list of string arguments specified in the vector args. Its behavior is otherwise identical to execl.

EXAMPLE

```
avec[0] = "mv";
avec[1] = file;
avec[2] = direc;
avec[3] = NULL;
execv("/bin/mv", avec);
putstr(STDERR, "can't exec mv\n", NULL);
exit(NO);
```

SEE ALSO

execl

NAME

exit - terminate program execution

SYNOPSIS

```
VOID exit(success)
    BOOL success;
```

FUNCTION

exit calls all functions registered with onexit, then terminates program execution. If success is non-zero (YES), a zero byte is returned to the invoker, which is the normal Idris convention for successful termination. If success is zero (NO), a one is returned to the invoker.

RETURNS

exit will never return to the caller.

EXAMPLE

```
if ((fd = open("file", READ)) < 0)
    {
    putstr(STDERR, "can't open file\n", NULL);
    exit(NO);
    }
```

SEE ALSO

onexit

NAME

fork - create a new process

SYNOPSIS

PID fork()

FUNCTION

fork creates a new process which is identical to the initial process, except for the value returned. All open files and all signal settings are the same in both processes. It is customary for the child process to invoke a program file via execl or execv, shortly after birth, while the parent either waits to learn the termination status of the child or proceeds on to other matters.

RETURNS

In the child process, fork returns a zero. In the parent process, fork returns the processid of the child if successful, or a negative number, which is the Idris error return code, negated. Failure occurs only if the system is out of resident heap space or (possibly) out of swap space.

EXAMPLE

```
if ((pid = fork()) < 0)
    putstr(STDERR, "try again\n", NULL);
else if (pid)
    while (wait(&status) != pid)
        ;
else
    {
    execl("prog", "prog", NULL);
    putstr(STDERR, "can't exec prog\n", NULL);
    exit(NO);
    }
```

SEE ALSO

execl, execv, wait

BUGS

If the parent never waits, the dead child will remain a zombie until the parent dies. A prolific parent can thus overpopulate the system.

NAME

fstat - get status of open file

SYNOPSIS

```
ERROR fstat(fd, buf)
    FILE fd;
    struct {
        UCOUNT dev; ino;
        BITS mode;
        UTINY nlinks;
        UID  uid, gid;
        UTINY  msize;
        UCOUNT lsize, addr[8];
        ULONG actime, modtime;
        } *buf;
```

FUNCTION

fstat obtains the status of the opened file fd in the structure pointed to by buf. The structure is essentially that of a filesystem inode, preceded by the device dev and the inode number ino on that device. The less significant byte of dev is the device minor number, the more significant byte is its major number.

mode takes the form 'a*zzlugtrwxrwxrwx', where a is set to indicate that the file is allocated. zz is 00 for a plain file, 01 for character special, 10 for a directory, and 11 for block special. l is set for a large (4096 <= size) file. The remaining bits give access permissions as described under chmod.

nlinks counts the number of directory entries that point at this inode. uid is the userid of the owner, and gid is the groupid of the owning group. The size of the file in bytes is ((LONG)msize << 16) + lsize.

For character and block special files, addr[0] contains the device major and minor numbers, the latter in the less significant byte. The eight block addresses are otherwise magic from the standpoint of most users.

The last accessed time, actime, and last modified time, modtime, are both in seconds from the 1 Jan 1970 epoch.

RETURNS

fstat returns zero if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if (fstat(STDIN, &sbuf) < 0 || buf.dev != dev)
    putstr(STDERR, "wrong filesystem\n", NULL);
```

SEE ALSO

chmod, getmod, stat

NAME
getcsw - get console switches

SYNOPSIS
BYTES getcsw()

FUNCTION
getcsw reads the console switches.

RETURNS
getcsw always succeeds in reading the console switches, even if they don't exist.

EXAMPLE
```
if (getcsw() == 0173030)
    sync();
```

BUGS
Many systems have no console switches in real life.

NAME
getegid - get effective groupid

SYNOPSIS
```
UID getegid()
```

FUNCTION
getegid obtains the current effective groupid.

RETURNS
getegid always returns the effective groupid.

EXAMPLE
To forget who invoked you:

```
setgid(getegid());
```

SEE ALSO
getgid, setgid

NAME
geteuid - get effective userid

SYNOPSIS
```
UID geteuid()
```

FUNCTION
geteuid obtains the current effective userid.

RETURNS
geteuid always returns the effective userid.

EXAMPLE
To forget who invoked you:

```
setuid(geteuid());
```

SEE ALSO
getuid, setuid

NAME
getgid - get real groupid

SYNOPSIS
```
UID getgid()
```

FUNCTION
getgid obtains the current real groupid.

RETURNS
getgid always returns the real groupid.

EXAMPLE
To revert ownership to whoever invoked you:

```
setgid(getgid());
```

SEE ALSO
getegid, setgid

NAME

getmod - get mode of file

SYNOPSIS

```
BITS getmod(fname)
    TEXT *fname;
```

FUNCTION

getmod obtains the mode of the file fname. The low order twelve bits of mode are used to specify access permissions as described for chmod.

RETURNS

getmod returns the (always non zero) mode of the file if successful, else zero.

EXAMPLE

To copy the mode of a file:

```
chmod(newfile, getmod(oldfile));
```

SEE ALSO

chmod

NAME

getpid - get processid

SYNOPSIS

```
PID getpid()
```

FUNCTION

getpid obtains the processid of the currently running process, which is not very meaningful but has the virtue of being unique among all living processes. Hence it serves as a useful seed for temporary filenames.

RETURNS

getpid returns the (always positive) processid.

EXAMPLE

```
name[itob(name, getpid(), 10)] = '\0';
fd = create(name, WRITE);
```

SEE ALSO

uname

NAME
getuid - get real userid

SYNOPSIS
```
UID getuid()
```

FUNCTION
getuid obtains the current real userid.

RETURNS
getuid always returns the real userid.

EXAMPLE
To revert ownership to whoever invoked you:

```
setuid(getuid());
```

SEE ALSO
geteuid, setuid

NAME

gtty - get tty status

SYNOPSIS

```
ERROR gtty(fd, buf)
    FILE fd;
    struct {
        BITS t_speeds;
        TEXT t_erase, t_kill;
        BITS t_mode;
        } *buf;
```

FUNCTION

gtty obtains the status of a tty or other character special device, under control of fd, that responds to stty system calls. For a tty, six bytes of status are returned for the character special file fd, the information being written in the structure pointed at by buf. Other character special devices may refuse to honor a gtty request, or they return other than six characters, depending strongly upon the device. If the device is a tty, the information can be interpreted as follows:

mask	value	meaning of speeds field
S_ISPEED	0x000f	input speed
S_IBREAK	0x0010	break received
S_ILOST	0x0020	input lost (overrun)
S_IMASK4	0x0040	reserved
S_IREADY	0x0080	input ready to be read
S_OSPEED	0x0f00	output speed
S_OBREAK	0x1000	send break char
S_ONXON	0x2000	don't output X-ON/X-OFF codes
S_OMASK4	0x4000	reserved
S_OREADY	0x8000	output is finished

The input speed and output speed are codes for baudrates of the set: {0, 50, 75, 110, 134.5, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400}. A baud rate of 0 calls for the tty to hangup. By no means do all devices support all speeds.

break received and input lost flags are reset after the gtty is done. input ready, if set, assures that a read of the terminal will not roadblock. output ready assures that the output queues are empty, and that the transmitter is ready for another character. output break is reset when the break character is sent. Since all devices cannot generate break characters, this bit may be ignored (and left set).

erase is the character which, if typed in other than raw mode, calls for the preceding character on the current line (if any) to be deleted. kill is the character which calls for the entire current line to be deleted. Defaults are '\b' (backspace) and '\25' (ctl-u). If the sign bit of either erase or kill is set, the sequence backspace-space-backspace used to erase characters on a CRT screen will be inhibited.

mask	value	meaning of mode field
M_RARE	0x0001	rare mode
M_XTABS	0x0002	expand tabs
M_LCASE	0x0004	map uppercase to lowercase
M_ECHO	0x0008	echo input
M_CRMOD	0x0010	map carriage return to linefeed
M_RAW	0x0020	raw mode
M_ODDP	0x0040	generate odd parity
M_EVENP	0x0080	generate even parity
M_NL3	0x0300	newline delay
M_HT3	0x0c00	horizontal tab delay
M_CR3	0x3000	carriage return dealy
M_FF1	0x4000	form feed delay
M_BS1	0x8000	backspace delay

rare (the least significant bit) puts the handler in a semi-transparent mode. DEL and FS characters cause interrupt signals, X-ON and X-OFF cause output to start and stop, and proper parity is generated on output. Parity bits are not removed on input.

expand tabs calls for each tab to be expanded to spaces. lcase maps all uppercase characters typed in to lowercase, and all lowercase characters typed out to uppercase. echo steers all characters typed in back out for full duplex operation. crmod accepts carriage returns CR as linefeeds LF, and expands all typed out LFs to CR-LF sequences.

raw instructs the handler to ignore interpretation of input characters, including the processing of erase and kill characters, the recognition of interrupt codes DEL and FS, and the treatment of EOT as end of file. Characters are read and written transparently as eight-bit bytes, with no parity checking or mapping, and with no typeout delays.

evenp and oddp control parity generation on output. If even and odd are off, the parity is zero. If even and odd are on, the parity is one. Otherwise, even selects even parity and odd selects odd parity.

Various typeout delays may be requested, for newlines with nl3, horizontal tabs with ht3, carriage returns with cr3, formfeeds and vertical tabs with ff1, and for backspaces with bs1. Actual delays are in multiples of 1/60 second ticks. The ranges are (0, 4, 8, 12) ticks for horizontal tabs, newlines, and carrage returns; (0, 64) for formfeeds and vertical tabs; and (0, 16) for backspaces.

RETURNS

gtty returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE

To put a tty in raw mode, with minimum perturbation:

```
gtty(fd, &stat);
stat.t_mode =| M_RAW;
stty(fd, &stat);
```

SEE ALSO
stty

NAME
kill - send signal to a process

SYNOPSIS
```
ERROR kill(pid, sig)
    PID pid;
    SIG signo;
```

FUNCTION
kill sends the signal signo to the process identified by processid pid. The sender must either have the same effective userid as the receiver or be the superuser; if pid is zero, the signal is sent to all processes under control of the same tty as the sender. A process cannot, however, kill itself.

The signals that may be sent are:

NAME	VALUE	MEANING
SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3	quit (core dump)
SIGILIN	4	illegal instruction (core dump)
SIGTRC	5	trace trap (core dump)
SIGRNG	6	range error (core dump)
SIGDOM	7	domain error (core dump)
SIGFPT	8	floating point exception (core dump)
SIGKILL	9	kill
SIGBUS	10	bus error (core dump)
SIGSEG	11	segmentation violation (core dump)
SIGSYS	12	bad system call (core dump)
SIGPIPE	13	broken pipe
SIGALRM	14	alarm clock
SIGTERM	15	terminate

A core dump may not occur on those signals that have been caught or ignored by the receiving process. Note that SIGALRM and SIGTERM are not defined in UNIX/V6, and that SIGRNG and SIGDOM have somewhat less general meaning on that system.

RETURNS
kill returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE
To hangup a long-idle terminal (as superuser):

```
kill(pid, 1);
```

SEE ALSO
signal

BUGS
kill is a misnomer, as it can be used to perform many other functions.

NAME
link - create link to file

SYNOPSIS
```
ERROR link(old, new)
    TEXT *old, *new;
```

FUNCTION
link creates a new directory entry for a file with existing name old; the added name is new. No checks are made for whether this is a good idea.

RETURNS
link returns zero if successful, else a negative number, which is the Idris error return code, negated.

EXAMPLE
```
    if (link(new, old) < 0 && unlink(old) < 0)
        putstr(STDERR, "can't move file\n", NULL);
```

SEE ALSO
unlink

BUGS
A program executing as superuser can scramble a directory structure by injudicious calls on link.

NAME

lseek - set file read/write pointer

SYNOPSIS

```
COUNT lseek(fd, offset, sense)
    FILE fd;
    LONG offset;
    COUNT sense;
```

FUNCTION

lseek uses the long offset provided to modify the read/write pointer for the file fd, under control of sense. If (sense == 0) the pointer is set to offset, which should be positive; if (sense == 1) the offset is algebraically added to the current pointer; otherwise (sense == 2) of necessity and the offset is algebraically added to the length of the file in bytes to obtain the new pointer. Idris uses only the low order 24 bits of the offset; the rest are ignored.

The call lseek(fd, 0L, 1) is guaranteed to leave the file pointer unmodified and, more important, to succeed only if lseek calls are both acceptable and meaningful for the fd specified. Other lseek calls may appear to succeed, but without effect, as when rewinding a terminal.

RETURNS

lseek returns the file descriptor if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

To read a 512-byte block:

```
BOOL getblock(buf, blkno)
    TEXT *buf;
    BYTES blkno;
    {
    lseek(STDIN, (LONG) blkno << 9, 0);
    return (read(STDIN, buf, 512) != 512);
    }
```

NAME

mkexec - make file executable

SYNOPSIS

```
BOOL mkexec(fname)
    TEXT *fname;
```

FUNCTION

mkexec adds "execute" permissions to the file fname. "Read" and "write" permissions are left unchanged.

RETURNS

mkexec returns YES if the file fname exists and permits its mode to be changed. Otherwise it returns NO.

EXAMPLE

```
if (load1() && load2())
    return (mkexec(xfile));
```

NAME

mknod - make a special inode

SYNOPSIS

```
ERROR mknod(fname, mode, dev)
    TEXT *fname;
    BITS mode;
    DEV dev;
```

FUNCTION

mknod creates an empty instance of a file with pathname fname, setting its mode bits to mode and its first address entry to dev. If (mode == 0140777) for instance, the new file will be a directory with general permissions; dev had better be zero in this case. If (mode == 0160644), the new file will be a block special device that can be read by all, but written only by the superuser; dev then specifies the major/minor device numbers, the minor number in the less significant byte.

Only the superuser may perform this call successfully.

RETURNS

mknod returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE

```
mknod("/dev/tty9", 0120622, major << 8 | minor);
```

NAME

mount - mount a file system

SYNOPSIS

```
ERROR mount(spec, fname, ronly)
    TEXT *spec, *fname;
    BOOL ronly;
```

FUNCTION

mount associates the root of the filesystem written on the block special device spec with the pathname fname; henceforth the mounted filesystem is reachable via pathnames through fname. If ronly is nonzero, the filesystem is mounted read only, i.e. all files are write protected and access times are not updated.

name must already exist; its contents are rendered inaccessible by the mount operation.

Only the superuser succeeds with this call.

RETURNS

mount returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE

```
mount("/dev/rk1", "/usr", 0);
```

SEE ALSO

umount

NAME

nice - set priority

SYNOPSIS

```
ERROR nice(pri)
    TINY pri;
```

FUNCTION

nice sets the scheduling priority of the process to pri, which must be in the range [0, 20] for all but the superuser. The higher the number, the lower the priority [sic]; default is zero.

On some implementations, a sufficiently negative pri will lock a process into memory, so that it is never swapped out.

RETURNS

nice returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE

To be polite when running a long program:

```
nice(4);
```

BUGS

nice is a misnomer, since it can be used to do unnice things.

NAME

onexit - call function on program exit

SYNOPSIS

```
VOID (*onexit())(pfn)
    VOID (*(*pfn)())();
```

FUNCTION

onexit registers the function pointed at by pfn, to be called on program exit. The function at pfn is obliged to return the pointer returned by the onexit call, so that any previously registered functions can also be called.

RETURNS

onexit returns a pointer to another function; it is guaranteed not to be NULL.

EXAMPLE

```
IMPORT VOID (*(*nextguy)())(), (*thisguy())();

if (!nextguy)
    nextguy = onexit(&thisguy);
```

SEE ALSO

exit, onintr

BUGS

The type declarations defy description, and are still wrong.

NAME

onintr - capture interrupts

SYNOPSIS

```
VOID onintr(pfn)
    VOID (*pfn)();
```

FUNCTION

onintr ensures that the function at pfn is called on a broken pipe, or on the occurrence of an interrupt (DEL key) or hangup generated from the keyboard of a controlling terminal. Any earlier call to onintr is overridden.

The function is called with one integer argument, whose value is always zero, and must not return; if it does, a message is output to STDERR and an immediate error exit is taken.

If (pfn == NULL) then these interrupts are disabled (turned off). Any disabled interrupts are not, however, turned on by a subsequent call with pfn not NULL.

RETURNS

Nothing.

EXAMPLE

A common use of onintr is to ensure a graceful exit on early termination:

```
    onexit(&rmtemp);
    onintr(&exit);
    ...
VOID rmtemp()
    {
    remove(uname());
    }
```

Still another use is to provide a way of terminating long printouts, as in an interactive editor:

```
    while (!enter(docmd, NULL))
        putstr(STDOUT, "?\n", NULL);
    ...
VOID docmd()
    {
    onintr(&leave);
```

SEE ALSO

onexit

NAME

open - open a file

SYNOPSIS

```
FILE open(fname, mode, rsize)
    TEXT *fname;
    COUNT mode;
    BYTES rsize;
```

FUNCTION

open opens a file with name fname and assigns a file descriptor to it. If (mode == 0) the file is opened for reading, else if (mode == 1) it is opened for writing, else (mode == 2) of necessity and the file is opened for updating (reading and writing).

rsize is the record size in bytes, which must be nonzero on many systems if the file is not to be treated as ASCII text. It is ignored by Idris, but should be present for portability.

RETURNS

open returns a file descriptor for the opened file or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if ((fd = open("xeq", WRITE, 1)) < 0)
    putstr(STDERR, "can't open xeq\n", NULL);
```

SEE ALSO

close, create

NAME

pipe - setup a data pipe

SYNOPSIS

```
FILE pipe(fds)
    FILE fds[2];
```

FUNCTION

pipe sets up a pipeline, i.e. a data transfer mechanism that can be read by one file descriptor and written by another. fds[0] is the read file descriptor, fds[1] is the write file descriptor.

Since children spawned by fork inherit all open files, it is possible with pipe to set up a communication link between processes having a common parent. The pipe mechanism synchronizes reading and writing, allowing the producer to get ahead up to 4096 bytes before being made to wait.

Reading an empty pipe with no writers left results in an end of file return; a pipe with no readers causes a broken pipe signal and, if the signal is ignored, causes an error return on subsequent writes.

RETURNS

pipe writes the read file descriptor in fds[0] and the write file descriptor in fds[1] if successful. The value of the function is fds[0] if successful or a negative number, which is the Idris error return code, negated.

EXAMPLE

To hook a child to STDOUT:

```
pipe(fds);
if (pid = fork())
    {
    close(STDOUT);
    dup(fds[1]);
    close(fds[0]);
    close(fds[1]);
    }
else
    {
    close(STDIN);
    dup(fds[0]);
    close(fds[0]);
    close(fds[1]);
    execl("child", "child", NULL);
    exit(NO);
    }
```

SEE ALSO

close, creat, create, dup, execl, execv, open, read, write

NAME

profil - set profiler parameters

SYNOPSIS

```
VOID profil(buf, size, offset, scale)
    COUNT *buf;
    BYTES size, offset, scale;
```

FUNCTION

profil sets the parameters used by the system for execution time profiling of the user mode program counter. On each clock tick (60 times per second), offset is subtracted from the user program counter and the result multiplied by scale, which is taken as an unsigned binary fraction in the interval [0, 1). If the result is in the interval [0, size), it is used as an index to select the element of buf to increment.

Unlike UNIX, Idris assumes that the binary fraction is always one less (when taken as an integer) than a power of two. That is, the resident considers only the highest order bit set in scale, and assumes that all bits to the right of it are ones. This difference should be transparent to all known uses of profil.

If scale is 0, profiling is turned off.

RETURNS

Nothing.

EXAMPLE

To profile to a resolution of four code bytes per counter:

```
profil(buf, size, 0, ((BYTES)-1 >> 2) + 1); /* [sic] */
```

NAME

read - read from a file

SYNOPSIS

```
COUNT read(fd, buf, size)
    FILE fd;
    TEXT *buf;
    BYTES size;
```

FUNCTION

read reads up to size characters from the file specified by fd into the buffer starting at buf.

RETURNS

If an error occurs, read returns a negative number which is the Idris error code, negated; if end of file is encountered, read returns zero; otherwise the value returned is between 1 and size, inclusive. When reading from a disk file, size bytes are read whenever possible.

EXAMPLE

To copy a file:

```
while (0 < (n = read(STDIN, buf, BUFSIZE)))
    write(STDOUT, buf, n);
```

SEE ALSO

write

NAME
remove - remove a file

SYNOPSIS
```
FILE remove(fname)
    TEXT *fname;
```

FUNCTION
remove deletes the file fname from the Idris directory structure. If no other names link to the file, the file is destroyed. If the file is opened for any reason, however, destruction will be postponed until the last close on the file.

If the file is a directory, remove will not attempt to remove it.

RETURNS
remove returns zero, if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE
```
if (remove("temp.c") < 0)
    putstr(STDERR, "can't remove temp file\n", NULL);
```

SEE ALSO
create

NAME

sbreak - set system break

SYNOPSIS

```
TEXT *sbreak(size)
    BYTES size;
```

FUNCTION

sbreak moves the system break, at the top of the data area, algebraically up by size bytes, rounded up as necessary to placate memory management hardware.

RETURNS

If successful, sbreak returns a pointer to the start of the added data area; otherwise the value returned is NULL.

EXAMPLE

```
if (!(p = sbreak(nsyms * sizeof (symbol))))
    {
    putstr(STDERR, "not enough room!\n", NULL);
    exit(NO);
    }
```

NAME

seek - set file read/write pointer

SYNOPSIS

```
ERROR seek(fd, offset, sense)
    FILE fd;
    COUNT offset, sense;
```

FUNCTION

seek uses the offset provided to modify the read/write pointer for the file fd, under control of sense. If (sense == 0) the pointer is set to offset, which is treated as an unsigned integer; if (sense == 1) the offset is algebraically added to the current pointer; if (sense == 2) the offset is algebraically added to the length of the file in bytes to obtain the new pointer.

If (sense is between 3 and 5 inclusive), the offset is multiplied by 512L and the resultant long offset is used with (sense - 3). Idris uses only the low order 24 bits of the offset; the rest are ignored. Block offsets are primarily of use in machines with short pointers.

RETURNS

seek returns zero if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

To read a 512-byte block:

```
BOOL getblock(buf, blkno)
    TEXT *buf;
    BYTES blkno;
    {
    seek(STDIN, blkno, 3);
    return (fread(STDIN, buf, 512) != 512);
    }
```

SEE ALSO

lseek

NAME
setgid - set groupid

SYNOPSIS
```
ERROR setgid(gid)
    UID gid;
```

FUNCTION
setgid sets the groupid, both real and effective, of the current process to gid. Only the superuser may change the gid.

RETURNS
setgid returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE
To revert effective groupid back to real:

```
setgid(getgid());
```

SEE ALSO
getgid

NAME
setuid - set userid

SYNOPSIS
```
ERROR setuid(uid)
    UID uid;
```

FUNCTION
setuid sets the userid, both real and effective, of the current process to uid. Only the superuser may change the uid.

RETURNS
setuid returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE
To revert effective userid back to real:

```
setuid(getuid());
```

SEE ALSO
getuid

NAME

signal - capture signals

SYNOPSIS

```
VOID (*signal(sig, pfunc))()
    SIG sig;
    VOID (*pfunc)();
```

FUNCTION

signal changes the handling of the signal sig according to pfunc. Legal values of sig are described under the kill system call. If pfunc is NULL, normal system handling occurs, i.e. the process is terminated when the signal occurs, possibly with a core dump; if pfunc is NOSIG (ie. 1), the signal is ignored; otherwise pfunc is taken as a pointer to a code sequence to be entered in the user program when the signal occurs.

Note that the code sequence may not, in general, be a C function, since registers may not be properly saved and the stack may not be prepared for an orderly return; to return properly to the interrupted code, a machine dependent code sequence must often be performed. If a system call was interrupted and the signal handler returns properly to the interrupted code, the system call reports an abnormal termination with the EINTR error code.

Except for illegal instruction and trace trap, all signals revert to system handling after each occurrence; all signals revert to system handling on an execl or execv, but not on a fork.

RETURNS

signal returns the old pfunc if successful, else -1.

EXAMPLE

To prevent hangups:

```
signal(1, NOSIG);
```

SEE ALSO

execl, execv, fork, kill

NAME

sleep - delay for awhile

SYNOPSIS

```
VOID sleep(secs)
    UCOUNT secs;
```

FUNCTION

sleep suspends execution of the current process for secs seconds.

RETURNS

sleep returns zero if successful (clock is enabled), or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
while ((fd = creat("lock", 0444)) < 0)
    sleep(5);
```

NAME

stat - get status of named file

SYNOPSIS

```
ERROR stat(fname, buf)
    TEXT *fname;
    struct {
        UCOUNT dev; ino;
        BITS mode;
        UTINY nlinks;
        UID uid, gid;
        UTINY msize;
        UCOUNT lsize, addr[8];
        ULONG actime, modtime;
        } *buf;
```

FUNCTION

stat obtains the status of the file fname in the structure pointed to by buf. The structure is essentially that of a filesystem inode, preceded by the device dev and the inode number ino on that device; it is described under fstat.

RETURNS

stat returns zero if successful, or a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if (stat(av[1], &sbuf) < 0 || !isdir(sbuf.mode))
    putstr(STDERR, av[1], " is not a directory\n", NULL);
```

SEE ALSO

chmod, fstat, getmod

NAME
stime - set system time

SYNOPSIS
```
ERROR stime(time)
    ULONG time;
```

FUNCTION
stime sets the system time to time, which is the number of seconds since the 1 Jan 1970 epoch. Only the superuser succeeds with this call.

RETURNS
stime returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE
To backup an hour:

```
stime(time() - 60 * 60);
```

SEE ALSO
time

NAME

stty - set tty status

SYNOPSIS

```
ERROR stty(fd, buf)
    FILE fd;
    struct {
        BITS t_speeds;
        TEXT t_erase, t_kill;
        BITS t_mode;
        } *buf;
```

FUNCTION

stty sets the status of a tty or other character special device, under control of the file descriptor fd, to the values in the structure pointed at by buf. The structure is the same as described under gtty.

Any type ahead is discarded if a transition is made from rare or raw mode to normal mode, and vice-versa. Transistions between rare and raw mode do not cause a buffer flush.

RETURNS

stty returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE

To change a tty speed, with minumum perturbation:

```
gtty(fd, &stat);
stat.t_speeds =& ~(T_OSPEED|T_ISPEED);
stat.t_speeds =| ospeed << 8 | ispeed;
stty(fd, &stat);
```

SEE ALSO

gtty

NAME

sync - synchronize disks with memory

SYNOPSIS

```
VOID sync()
```

FUNCTION

sync ensures that all delayed writes are performed by the system, so that disk integrity is assured before taking the system down. It updates all inodes that have been modified since the last sync, and writes all data blocks not correctly represented on open or mounted block special devices.

If a filesystem is to be accessed other than through the block special file on which it is mounted, sync should first be performed to ensure that the disk image is current.

RETURNS

Nothing.

EXAMPLE

A simple "sync daemon" is:

```
FOREVER
    {
    sync();
    sleep(30);
    }
```

NAME
time - get system time

SYNOPSIS
ULONG time()

FUNCTION
time gets the system time, which is the number of seconds since the 1 Jan 1970 epoch.

RETURNS
time returns the system time as a long integer.

EXAMPLE
To backup an hour:

```
stime(time() - 60 * 60);
```

SEE ALSO
stime

NAME

times - get process times

SYNOPSIS

```
ERROR times(buf)
    struct {
        UCOUNT putime, pstime;
        ULONG cutime, cstime;
        } *buf;
```

FUNCTION

times returns the cumulative times consumed by the current process and all its dead children in the structure pointed at by buf. putime is the user mode time consumed by the process proper; pstime is its system mode time. cutime and cstime are the cumulative user and system times consumed by all the children that have been laid to rest by wait system calls, including the times of all their children thus interred.

All times are in 1/60 second ticks.

RETURNS

times writes the process times in the structure pointed at by buf. times returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE

```
times(&vec);
putfmt("System: %.1f User: %.1f\n", vec.cstime/60.0, vec.cutime/60.0);
```

SEE ALSO

time

NAME

umount - unmount a filesystem

SYNOPSIS

```
ERROR umount(spec)
    TEXT *spec;
```

FUNCTION

umount disassociates the root of the filesystem written on the block special device spec with whatever node it was mounted on; henceforth the filesystem is no longer reachable via the directory tree.

Only the superuser succeeds with this call.

RETURNS

umount returns zero if successful, else a negative number which is the Idris error return code, negated.

EXAMPLE

```
umount("/dev/rk1");
```

SEE ALSO

mount

NAME

uname - create a unique file name

SYNOPSIS

```
TEXT *uname()
```

FUNCTION

uname returns a pointer to the start of a NUL terminated name which is guaranteed not to conflict with normal user filenames. The name is, in fact, unique to each Idris process, and may be modified by a suffix, so that a family of process-unique files may be dealt with. The name may be used as the first argument to a create, or subsequent open, call, so long as any such files created are removed before program termination. It is considered bad manners to leave scratch files lying about.

RETURNS

uname returns the same pointer on every call during a given program invocation. It takes the form "/tmp/t#####" where ##### is the processid in octal. The pointer will never be NULL.

EXAMPLE

```
if ((fd = create(uname(), WRITE, 1)) < 0)
    putstr(STDERR, "can't create sort temp\n", NULL);
```

SEE ALSO

close, create, open, remove

BUGS

A program invoked by the exec system call, without a fork, inherits the Idris processid used to generate unique names. Collisions can occur if files so named are not meticulously removed.

NAME

unlink - erase link to file

SYNOPSIS

```
ERROR unlink(fname)
    TEXT *fname;
```

FUNCTION

unlink removes the link specified by the file fname. No checks are made for whether this is a good idea. Only the superuser may unlink a directory.

RETURNS

unlink returns zero if successful, else a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if (!isdir(getmod(file)))
    unlink(file);
```

SEE ALSO

link, remove

BUGS

A program executing as superuser can scramble a directory structure by injudicious calls on unlink.

NAME

wait - wait for child to terminate

SYNOPSIS

```
PID wait(pstat)
    COUNT *pstat;
```

FUNCTION

wait suspends execution of the calling program until a child process terminates, so that it can return the child's termination status at pstat and its processid as the value of the function. Children remain in limbo (zombie status, actually) until laid to rest by a waiting parent.

The status returned contains the number of the terminating signal, if any, in its less significant byte, and the status reported back by the child's exit call in its more significant byte. By convention, a status word of all zeros means normal termination; if (*pstat & 0200) then a core dump has been made.

RETURNS

wait returns the processid of the child whose status is written at pstat, if any, else -1 if the caller has no children.

EXAMPLE

```
if (0 < (pid = fork()))
    while (wait(&status) != pid)
        ;
```

SEE ALSO

fork

NAME
write - write to a file

SYNOPSIS
```
COUNT write(fd, buf, size)
    FILE fd;
    TEXT *buf;
    BYTES size;
```

FUNCTION
write writes size characters starting at buf to the file specified by fd.

RETURNS
If an error occurs, write returns a negative number which is the Idris error code, negated; otherwise the value returned should be size.

EXAMPLE
To copy a file:

```
while (0 < (n = read(STDIN, buf, size)))
    write(STDOUT, buf, n);
```

SEE ALSO
read

NAME

xecl - execute a file with argument list

SYNOPSIS

```
COUNT xecl(fname, sin, sout, flags, s0, s1, ..., NULL)
    TEXT *fname;
    FILE sin, sout;
    COUNT flags;
    TEXT *s0, *s1, ...
```

FUNCTION

xecl invokes the program file fname, connecting its STDIN to sin and STDOUT to sout and passing it the string arguments s0, s1, ... If (!(flags & 3)) fname is invoked as a new process; xecl will wait until the command has completed and will return its status to the calling program. If (flags & 1) fname is invoked as a new process and xecl will not wait, but will return the processid of the child. If (flags & 2) fname is invoked in place of the current process, whose image is forever gone. In this case, xecl will never return to the caller.

To the value of flags may be added a 4 if the processing of interrupt and quit signals for fname is to revert to system handling. The value of flags may also be incremented by 8 if the effective userid is to be made the real userid before fname is executed. If sin is not equal to STDIN, or if sout is not equal to STDOUT, the file (sin or sout) is closed before xecl returns.

If fname does not contain a '/', then xecl will search an arbitrary series of directories for the file specified, by prepending to fname each path specified by the global variable _paths before trying to execute it. _paths is of type pointer to TEXT, and points to a NUL terminated series of directory paths separated by '¦'s.

If the file eventually found has execute permission, but is not in executable format, /bin/sh is invoked with the current prefixed version of fname as its first argument and, following fname, an argument vector composed of s0, s1, ...

RETURNS

If fname cannot be invoked, xecl will fail. If (!(flags & 3)) xecl returns YES if the command executed successfully, otherwise NO; if (flags & 1) xecl returns the id of the child process, if one exists, otherwise zero; if (flags & 2) xecl will never return to the caller.

In all cases, if fname cannot be executed, an appropriate error message is written to STDERR.

EXAMPLE

```
if (!xecl(pgm, STDIN, create(file, WRITE), 0, f1, f2, NULL))
    putstr(STDERR, pgm, " failed\n", NULL);
```

SEE ALSO

xecv

NAME

xecv - execute a file with argument vector

SYNOPSIS

```
COUNT xecv(fname, sin, sout, flags, av)
    TEXT *fname;
    FILE sin, sout;
    COUNT flags;
    TEXT **av;
```

FUNCTION

xecv invokes the program file fname, connecting its STDIN to sin and STDOUT to sout and passing it the string arguments specified in the NULL terminated vector av. Its behavior is otherwise identical to xecl.

SEE ALSO

xecl

SECTION THREE

PROGRAMMING FILE FORMATS

NAME

Files - special file formats

FUNCTION

The files documented in this section are those used in the process of developing programs under Idris, or those which are likely to be usable only by people with some knowledge of programming. All formats presented here are dictated, to a greater or lesser extent, by the Idris resident; most are produced or read directly by the resident.

Many other file formats of a more general nature may be found in Section III of the Idris Users' Manual.

NAME

bnames - block device names pseudo file

SYNOPSIS

/dev/bnames

FUNCTION

/dev/bnames is a character special file which reads out the names of the block special device handlers.

It consists of a concatenation of text lines, one for each entry in the table of block special devices; the first line corresponds to major device number 0.

/dev/bnames cannot be written.

SEE ALSO

cnames

NAME

cnames - character device names pseudo file

SYNOPSIS

/dev/cnames

FUNCTION

/dev/cnames is a character special file which reads out the names of the character special device handlers.

It consists of a concatenation of text lines, one for each entry in the table of character special devices; the first line corresponds to major device number 0.

/dev/cnames cannot be written.

SEE ALSO

bnames

NAME

core - core dump format

FUNCTION

core is the ancient term for main computer storage, dating back to the widespread use of magnetic cores to implement random access memory. A core dump, by extension, is an image of process memory, together with register contents and other status information, that is preserved when a process comes to an untimely end and may be in need of post mortem analysis. Idris can be coerced by a variety of means into producing a core dump, always into a file called "core" in the current directory. Such a file is best interpreted by the post mortem debugger, db, described in the C Interface Manual for the host machine.

A core dump image consists of a header followed by the user alterable portion of a process address space.

The header consists of an identification byte 0x9b, a configuration byte, a short int giving the size of the header in bytes, and six unsigned ints containing: the number of bytes defined by any separate text segment (or 0 if none), the number of bytes defined by the data segment (or combined text/data segment), the number of bytes between the end of the data segment and the top of stack at the time of the dump, the number of bytes between top of stack and the end of the address space (base of stack), the offset of any separate text segment, and the offset of the data segment (or combined text/data segment). The remainder of the header consists of an int giving the cause of death of the process, a machine-dependent panel containing the registers at the time of the dump, and sufficient filler bytes to bring the header length to the value indicated earlier.

The byte order and size of all ints in the header are determined by the configuration byte, which has the format given in the description of standard object files.

The dumped process image following the header contains the entirety of user alterable address space. If a separate "I-space" existed during the run, its contents are not dumped; otherwise, the text segment of the process is dumped immediately preceding its data segment.

SEE ALSO

object

NAME

inodes - resident inode list pseudo file

SYNOPSIS

/dev/inodes

FUNCTION

/dev/inodes is a character special file which reads out the resident list of inode. It is used by the standard utility ps to display the information in a format more or less palatable to human beings, or at least to gurus.

It consists of a concatenation of INODE entries, as documented in the resident header file /lib/fio.h.

/dev/inodes cannot be written.

SEE ALSO

mount

NAME

kmem - kernel memory pseudo file

SYNOPSIS

/dev/kmem

FUNCTION

/dev/kmem is a character special device that reads and writes kernel memory (where the Idris resident lives) as if it were a file. Nonexistent memory may be written, with no ill effect even in the presence of hardware trapping; nonexistent memory reads, if trapped, as all ones.

Its primary use is for on-the-fly inspection and patching of the resident, preferably by a guru armed with at least a symbol table.

SEE ALSO

mem

BUGS

It always accesses memory a byte at a time, which can confuse primitive memory mapped peripheral controllers wired only for word accesses.

NAME

library - standard library format

FUNCTION

Standard libraries are administered by the programming utility lib, primarily for conditional linking of object modules by the utility link. (Both utilities are documented in the C Interface Manual for any target machine.) They permit any number of files, typically object modules but possibly anything, to be administered as a single file; any file can be extracted by scanning the library for a matching name or other desirable properties. Standard library files are written in a machine independent fashion, so that they can be used unchanged across various implementations of Idris, plus other systems for which the portable C interface is supported.

The standard library format consists of a two-byte header having the value 0177565, written less significant byte first, followed by zero or more entries. Each entry consists of a fourteen-byte name, left justified and NUL padded, followed by a two-byte unsigned file length, also stored less significant byte first, followed by the file contents proper. If a name begins with a NUL byte, it is taken as the end of the library file.

Note that this differs in several small ways from UNIX/V6 archive file format, which has a header of 0177555, an eight-byte name, six bytes of miscellaneous UNIX-specific file attributes, and a two-byte file length. Moreover, a file whose length is odd is followed by a NUL padding byte in the UNIX format, while no padding is used in standard library format.

UNIX/V7 format is characterized by a header of 0177545, a fourteen-byte name, eight bytes of UNIX-specific file attributes, and a four-byte length. Odd length files are also padded to even.

The utility lib is capable of administering any of these formats.

BUGS

There should be a NUL at the end of all libraries, so that they are properly terminated even when written on a diskette.

NAME

mem - user memory pseudo file

SYNOPSIS

/dev/mem

FUNCTION

/dev/mem is a character special device that reads and writes per process (user) memory as if it were a file. This is seldom wise but occasionally useful.

Its behaviour in the presence of illegal access requests is the same as kmem.

SEE ALSO

kmem

NAME

mount - resident mount list pseudo file

SYNOPSIS

/dev/mount

FUNCTION

/dev/mount is a character special file which reads out the resident list of mounted filesystems. It is used by the standard utility ps to determine the number of filesystems actually mounted; it is capable of delivering even more information.

It consists of a concatenation of MOUNT entries, as documented in the resident header file /lib/fio.h.

/dev/mount cannot be written.

SEE ALSO

inodes

NAME

myps - current user process status psuedo file

SYNOPSIS

/dev/myps

FUNCTION

/dev/myps is a character special file which reads out the resident process list. It is used by the standard utility ps to determine the status of processes started at a given terminal.

Its behaviour is identical to /dev/ps, except that only processes having the same controlling teletype as the reader are read out.

SEE ALSO

ps

NAME

object - relocatable object file format

FUNCTION

For a file to be executable under Idris, it must look like an "object file", a file produced by or for the program builder utility called link, which is described in the C Interface Manual for the target machine. This is a (possibly) special case of a "relocatable" object file, one that contains sufficient information to be combined (by link) with other object files and/or altered to execute properly in different parts of memory. Thus, object files are widespread, and form an important part of the Idris environment.

Note that a file may have execute permission and still not be an object file -- it may be a directory with scan permission or it may be a command script to be interpreted by the shell. Nor is an object file necessarily executable -- it may be a bootstrap file (image of the resident), or a file executable on another machine, or a file that must be combined by link with other object files to make a complete program (its name probably ends in ".o" in this case).

A relocatable object image consists of a header, followed by a text segment, a data segment, the symbol table, and relocation information.

The header consists of an identification byte 0x99, a configuration byte, a short int containing the number of symbol table bytes, and six unsigned ints giving: the number of bytes of object code defined by the text segment, the number of bytes of object code defined by the data segment, the number of bytes needed by the bss segment, the number of bytes needed for stack plus heap, the text segment offset, and the data segment offset. Byte order and size of all ints in the header are determined by the configuration byte.

The configuration byte contains all information needed to fully represent the header and remaining information in the file. Its value val defines the following fields: ((val & 07) << 1) + 1 is the number of characters in the symbol table name field, so that values [0, 7] provide for odd lengths in the range [1, 15]. If (val & 010) then ints are four bytes; otherwise they are two bytes. If (val & 020) then ints are represented least significant byte first, otherwise, most significant byte first; byte order is assumed to be purely ascending or purely descending. (val & 0140) >> 5 is the strongest enforced storage bound restriction; values of 0, 1, 2, 3 provide for bounds that are multiples of 0, 2, 4, 8 bytes, respectively. If (val & 0200) no relocation information is present in this file.

The text segment is relocated relative to the text segment offset given in the header (usually zero), while the data segment is relocated relative to the data segment offset (usually the end of the text segment). In all cases the bss segment is relocated relative to the end of the data segment.

Relocation information consists of two sucessive byte streams, one for the text segment and one for the data segment, each terminated by a zero con-

trol byte. Control bytes in the range [1, 31] cause that many bytes in the corresponding segment to be skipped; bytes in the range [32, 63] skip 32 bytes, plus 256 times the control byte minus 32, plus the number of bytes specified by the relocation byte following.

All other control bytes control relocation of the next short or long int in the corresponding segment. If the 1-weighted bit is set in such a control byte, then a change in load bias must be subtracted from the int. The 2-weighted bit is set if a long int is being relocated instead of a short int. The value of the control byte right-shifted two places, minus 16, constitutes a "symbol code".

A symbol code of 47 is replaced by a code obtained from the byte or bytes following in the relocation stream. If the next byte is less than 128, then the symbol code is its value plus 47. Otherwise, the code is that byte minus 128 plus 175, the sum times 256, plus the value of the next relocation byte after that one.

A symbol code of zero calls for no further relocation; 1 means that a change in text bias must be added to the item (short or long int); 2 means that a change in data bias must be added; 3 means that a change in bss bias must be added. Other symbol codes call for the value of the symbol table entry indexed by the symbol code minus 4 to be added to the item.

Each symbol table entry consists of a value int, a flag byte, and a name padded with trailing NULs. Meaningful flag values are 0 for undefined, 4 for defined absolute, 5 for defined text relative, 6 for defined data relative, and 7 for defined bss relative. To this is added 010 if the symbol is to be globally known. If an undefined symbol has a non-zero value, it is taken as a request to reserve at least that many bytes for the symbol in the bss area, starting at the strongest required storage boundary.

SEE ALSO
core

NAME

profile - profile dump format

FUNCTION

A profile of a program is a histogram over time, showing how often each portion of the program was caught in execution, plus a list of entry counts for each of the instrumented functions that comprise the program. It is used by programmers, in conjunction with the prof post processor described in the C Interface Manual for the target machine, to debug and tune programs in an execution environment.

A profile dump file is produced at the end of an instrumented program execution. Its format is uniform across implementations of Idris, consisting of a header, followed by the profiling buffer and an array of function entry counts.

The header consists of an identification byte 0x9a, a configuration byte, a short int giving the number of bytes in the entry count array, and six unsigned ints giving: the (now meaningless) run-time address of the profiling buffer, the size of the buffer in bytes, the run-time offset subtracted from the pc to index the buffer, the scaling factor applied to the modified pc (given as a binary fraction in the interval [0, 1)), the text segment offset, and the offset from each function entry point of the corresponding address in the entry count array. Note that the first four ints are just the arguments to the prof system call, which is described in Section II of this manual.

The byte order and size of all ints in the header (and the byte order of ints elsewhere) are determined by the configuration byte, which has the format given in the description of standard object files. The header is immediately followed by a buffer of the size indicated, consisting of an array of short ints each of which contains a count of the clock ticks at which the pc was observed in the range of addresses corresponding to that element of the array.

Following the buffer is an array indicating the number of entries made to profiled functions during the run. Each element of the array is a structure with a pointer, called addr, and a long integer, called count. addr indicates the function to which this descriptor applies, and count contains the number of calls made to it. Specifically, addr contains the return address visible to the counting routine called at the start of each function, and so points some small (currently fixed) number of bytes above the actual entry point of the function. Its offset from the entry point is indicated by the final int in the dump file header. Note that the size of the pointer, and the byte ordering, in this record are determined by the configuration byte.

SEE ALSO

core, object

NAME

ps - process status psuedo file

SYNOPSIS

/dev/ps

FUNCTION

/dev/ps is a character special file which reads out the resident process list. It is used by the standard utility ps to determine the status of all processes in the system.

It consists of a concatenation of PROC plus ZLIST entries, one set for each of the processes administered by the resident. The structures delivered up are exact images, in native byte order, of the structures as documented in the resident header file /lib/cpu.h.

A PROC structure is followed by a list of zombies, or ZLIST structures, only if the p_zlist pointer is not NULL; the last ZLIST structure for a given process has a next field of NULL. Similarly, the PROC list ends with a structure having a next field of NULL.

/dev/ps cannot be written.

SEE ALSO

myps

SECTION FOUR

IDRIS SUPPORT LIBRARY

NAME

Conventions - the Idris support library

SYNOPSIS

/lib/libi*

FUNCTION

The functions documented in this section are kept in a library, separate from the standard C library, with a name whose prefix is /lib/libi (and whose suffix is machine dependent). They are used extensively in the construction of standard Idris utilities, to perform common functions uniformly and to enforce communication protocols among utilities. They are thus quite useful to anyone wishing to add utilities that are to cooperate with the existing community.

Most of the notation used here is the same as in earlier sections of this manual, but there are a few additional types that creep in from various header files:

BLOCK an unsigned short, capable of holding any filesystem block number. Block 0 is often taken as the absence of a block number. Defined in /lib/ino.h.

FINODE a filesystem inode, possibly as represented on the disk or possibly in native byte order. Defined in /lib/ino.h.

INUM an unsigned short, capable of holding any filesystem inode number. Inode 0 does not exist. Defined in /lib/ino.h.

TVEC a time vector, used to communicate parsed dates among library functions. Defined in /lib/time.h as:

```
typedef struct {
    BYTES secs;     /* seconds      [0, 60) */
    BYTES mins;     /* minutes      [0, 60) */
    BYTES hrs;      /* hours        [0, 24) */
    BYTES dmth;     /* day of month         [1, 31] */
    BYTES mth;      /* month of year        [0, 12) */
    BYTES yr;       /* years since 1900     [70, 131) */
    BYTES dwk;      /* day of week, sunday = 0  [0, 7) */
    BYTES dyr;      /* day of year              [0, 365] */
    BOOL dstf;      /* non-zero if daylight savings time */
    } TVEC;
```

Also included in this library is a version of putf (the internal routine _putf to be exact) that cannot deal with the floating point conversions %d and %f. This is provided so that utilities can perform formatted output -- using decode, errfmt, putf, and putfmt -- without dragging in (possibly extensive) floating point runtime support code that will never be needed. Be warned, however, that automatically including the Idris support library with the standard compile and link scripts will lead to puzzling behavior in a program that expects to perform floating output.

NAME

_penable - control function entry counts in profiling

SYNOPSIS

```
TBOOL _penable;
```

FUNCTION

_penable is used by the function entry counting routine of the profiling package, to control whether or not calls to the routine actually will record function entries. If _penable is non-zero, function entries will be counted; otherwise, the counting routine returns without doing anything.

_profil() sets _penable to 1 just before returning. _proend() clears _penable just after being called.

SEE ALSO

_proend, _profil

NAME

_proend - end profiling

SYNOPSIS

VOID (*proend())()

FUNCTION

_proend terminates profiling, by clearing the one-byte flag _penable to disable function entry counting, calling profil() to end time profiling, and writing out the time profiling and entry count buffers to the file named in the last preceding call to _profil().

RETURNS

Since _proend() is linked into the chain of programs to be called on program exit or interrupt, it returns the address of the next one to call.

SEE ALSO

_penable, _profil

NAME

_profil - start profiling

SYNOPSIS

```
TEXT *_profil(p)
    struct {
        UCOUNT config, esize;
        BYTES pbuf, psize, offset, scale, tbias, ebias;
        TEXT *fname;
        } *p;
```

FUNCTION

_profil performs a number of useful housekeeping functions preparatory to commencing program profiling. Its single argument, p, is a pointer to a standard profile file header, which is immediately followed by the name of the file to be generated by _proend().

_profil computes file header parameters psize and scale, allocates space for the time profiling and entry count buffers at pbuf, calls profil() to initiate time profiling, and ensures that _proend() is called on program interrupt or exit. Finally, it enables function entry counting by setting the one-byte flag _penable to 1.

RETURNS

_profil returns the address of the first byte past the end of the buffer space it allocated at pbuf.

SEE ALSO

_penable, _proend

NAME

askpw - ask for a password

SYNOPSIS

```
TEXT *askpw(kbuf, key)
    TINY kbuf[8];
    TEXT *key;
```

FUNCTION

askpw fills kbuf with the NUL terminated password at key, or if (key == NULL) reads a line from STDIN to fill kbuf. Before reading a line from a terminal as STDIN, askpw will drain outstanding input by performing an stty, prompt with the string "\7password:\ ", and accept the subsequent line, echoing only the trailing newline.

In any case, the first eight characters of the password are used; a short string is padded on the right with NULs.

Note that a password may thus be obtained from any of three sources: from an argument to the function, from a terminal with prompting and with printing suppressed, or from a noninteractive STDIN with no prompting.

RETURNS

askpw returns the address of kbuf, which contains the NUL padded password.

EXAMPLE

```
bldks(ks, askpw(kbuf, NULL));
```

SEE ALSO

codepw

NAME

asure - get user response to question

SYNOPSIS

```
BOOL asure(p)
    TEXT *p;
```

FUNCTION

If STDIN looks like a terminal, asure drains its input by performing an stty, writes the NUL terminated string at *p to STDERR, followed by a space, then reads up to 32 characters from STDIN.

This meticulous sequence of events is useful when a program needs to be assured that a conscious human accomplice is present.

RETURNS

If the line read begins with a 'y' or 'Y', or if STDIN is not a terminal, asure returns YES; otherwise it returns NO.

EXAMPLE

```
if (asure("are you sure?"))
    scrog();
```

NAME

atime - convert time vector to ASCII string

SYNOPSIS

```
TEXT *atime(vt, s)
    TVEC *vt;
    TEXT *s;
```

FUNCTION

atime converts the time vector at vt to a 24 character string at s, having the form:

```
        Thu Aug 12 09:53:12 1980
```

There is no terminating NUL or newline.

RETURNS

atime writes the date in ASCII at s[0] through s[23], and returns s.

EXAMPLE

To print the date:

```
    IMPORT LONG time();
    IMPORT TEXT *_est, *_edt;
    INTERN TEXT buf[] {"012345678901234567890123 "};
    TVEC tvec;

    ltime(&tvec, time(NULL));
    putstr(STDOUT, atime(&tvec, buf), tvec.dstf ? _edt : _est,
        "\n", NULL);
```

SEE ALSO

ltime, vtime

NAME

baudcode - return code given speed text

SYNOPSIS

```
UCOUNT baudcode(s)
    TEXT *s;
```

FUNCTION

baudcode compares the text string s with the table baudlist and returns the index of the matching string.

RETURNS

The speed code in the range [0, 15], or 16, if the lookup fails.

EXAMPLE

```
#include <sys.h>

BITS s;
SGTTY tbuf;
TEXT *speed = NULL;

getflags(&ac, &av, "s*:F", &speed);
gtty(fd, &tbuf);
if (speed && (s = baudcode(speed)) < 16)
    {
    tbuf.t_speeds =& ~(T_OSPEED|T_ISPEED);
    tbuf.t_speeds =| s << 8 | s;
    stty(fd, &tbuf);
    }
else
    remark(speed, ": unavailable baudrate");
```

SEE ALSO

baudlist, baudtext

NAME

baudlist - list of speeds supported by Idris drivers

SYNOPSIS

```
TEXT baudlist[NBAUD] {
    "0", "50", "75", "110"
    "134.5", "150", "200", "300",
    "600", "1200", "1800", "2400",
    "4800", "9600", "19200", "38400"};
```

FUNCTION

baudlist is a table of text speeds supported by Idris drivers. The table is indexed by a code in the range [0, 15].

SEE ALSO

baudcode, baudtext

NAME

baudtext - return text speed given speed code

SYNOPSIS

```
TEXT *baudtext(c)
    UCOUNT c;
```

FUNCTION

baudtext returns a pointer to the speed text string corresponding to the baudrate code c. The actual strings live in the table baudlist[].

RETURNS

A pointer to the speed text string, or NULL if code was not in the range [0, 15].

SEE ALSO

baudcode, baudlist

NAME

clrbuf - clear a standard sized buffer

SYNOPSIS

```
VOID clrbuf(buf)
    TEXT buf[BUFSIZE];
```

FUNCTION

clrbuf writes zeros throughout a 512-byte buffer beginning at buf.

RETURNS

Nothing, except a clear buffer.

EXAMPLE

```
clrbuf(p->_buf);
```

NAME

codepw - encode a password

SYNOPSIS

```
TEXT *codepw(loginid, kbuf)
    TEXT *loginid, kbuf[8];
```

FUNCTION

codepw encrypts the password in kbuf, then translates it to a printable form, suitable for storing in a textfile. The encrypted form is obtained by using the DES algorithm to encrypt the first eight characters of the file /adm/salt, exclusive-ored character by character with the NUL terminated string at loginid, using the password as a key. If the salt file is not readable, the string "password" is used in its stead.

The resulting eight-character encrypted string is repacked as twelve six-bit characters, using the alphabet [0-9a-zA-Z/.]. For convenience, a NUL is placed at the end of this string.

RETURNS

codepw returns a pointer to the NUL terminated, twelve-character encoded password which is stored in an internal buffer.

EXAMPLE

```
if (!cmpstr(codepw("root", askpw(kbuf, NULL)), getpw("root", 0, 1)))
    error("sorry", NULL);
```

FILES

/adm/salt for the string to be encrypted.

SEE ALSO

askpw, getpw

NAME

cpyi - copy an inode converting between native and filesystem

SYNOPSIS

```
FINODE *cpyi(dest, src)
    FINODE *dest, *src;
```

FUNCTION

cpyi copies the entire FINODE structure pointed to by src into the structure at dest, ensuring that all fields are converted if native byte order differs from filesystem byte order. For portability, its use is encouraged even on machines that need no conversion.

It is permissible for src and dest to be the same.

RETURNS

cpyi returns dest.

EXAMPLE

To get a pointer to an inode in native order:

```
FINODE *getino(fd, ino)
    FILE fd;
    INUM ino;
    {
    INTERN FINODE buf[BUFSIZE / sizeof (FINODE)], ibuf;

    if (!getblk(fd, buf, inblk(ino)))
        return (NULL);
    else
        return (cpyi(&ibuf, ioff(buf, ino)));
    }
```

NAME

cwd - get current working directory

SYNOPSIS

```
COUNT cwd(tbuf)
    TEXT tbuf[NAMSIZE];
```

FUNCTION

cwd determines the absolute pathname of the current working directory by tracing .. entries from . back to the root. The result is placed in tbuf as a NUL terminated string. If the current directory is on a mounted filesystem the mount history file /adm/mtab is used to determine the prefix of the absoute pathname.

If the path to the root of the filesystem becomes longer than 64 characters, including the terminating NUL, cwd returns the system error code -E2BIG. If cwd cannot find an entry in a directory that it needs it returns the system error code -EMLINK. If an error occurs while trying to open any directories for reading, the partially formed path is left in tbuf and cwd returns the appropriate system return code.

RETURNS

cwd writes a NUL terminated string at tbuf. If successful, the return value is zero; otherwise, the return value is a negative number indicating cwd's displeasure (as one of the Idris error return codes, negated).

EXAMPLE

To map a pathname to absolute form:

```
if (name[0] == '/')
    cpystr(abuf, name, NULL);
else if (cwd(build) < 0)
    error("broken directory tree", NULL);
else
    cpystr(abuf, build, "/", name, NULL);
```

FILES

/adm/mtab for mounted filesystems, . for current directory, \&..[/..]^ for parents.

NAME

devname - get device name

SYNOPSIS

```
BOOL devname(s, mdev, cspec)
    TEXT *s;
    UCOUNT mdev;
    BOOL cspec;
```

FUNCTION

devname fills the buffer pointed to by s with the NUL terminated device name in the /dev directory matching the major/minor device code contained in mdev. If cspec is nonzero, the device must be a character special device; otherwise, the device must be a block special device.

RETURNS

devname returns YES if it could find the appropriate device entry. The buffer at s is filled in with the device name, written as a 14-character link right filled with NULs.

EXAMPLE

The terminal message control function is:

```
BOOL mesg(new)
    BOOL new;
    {
    FAST BOOL old;
    STAT node;
    TEXT buf[20];

    if (fstat(STDERR, &node) < 0)
        return (NO);
    old = (node.s_mode & 022) ? YES : NO;
    cpybuf(buf, "/dev/", 5);
    buf[19] = '\0';
    if (devname(buf + 5, node.s_addr[0], YES))
        chmod(buf, new ? 0622 : 0600);
    return (old);
    }
```

NAME

ename - get pathname of an entry in a directory

SYNOPSIS

```
TEXT *ename(pname, dname, pdir)
    TEXT *pname, *dname;
    DIR *pdir;
```

FUNCTION

ename creates a fully qualified pathname consisting of the directory named dname to which is appended a '/' and the entry name pointed at by pdir. This NUL terminated pathname is returned at pname.

pname should be large enough to hold lenstr(dname) + 16 characters, counting the terminating NUL.

RETURNS

ename returns a pointer to the entry name.

EXAMPLE

```
if(!cmpstr(".", pdir->d_name) && !cmpstr("..", pdir->d_name))
    remove(ename(entry, dir, pdir));
```

NAME

flushi - flush out any pending inode writes

SYNOPSIS

```
VOID flushi(fd)
    FILE fd;
```

FUNCTION

flushi writes the in-core buffer used by geti and puti to the filesystem controlled by fd. If there have been no changes to the buffer, no output is performed. In any case, the buffer is disqualified, and the next geti or puti is guarenteed to read a fresh block from the filesystem.

RETURNS

If there is pending output, and it cannot be written, the writerr condition is raised.

EXAMPLE

To process the entire inode list of a filesystem:

```
for (i = isize << 4; 1 <= i; --i)
    if (process(pi = geti(fd, &ibuf, i)))
        puti(fd, pi, i);
flushi(fd);
```

SEE ALSO

geti, puti

NAME

ftime - find modified or accessed time of a file

SYNOPSIS

```
LONG ftime(fd, modflag)
    FILE fd;
    BOOL modflag;
```

FUNCTION

ftime finds the time of last access, or the time of last modification if modflag is true, to the file under control of fd.

RETURNS

ftime returns the time specified by modflag in seconds since 1 Jan 1970, or zero if the file status is unobtainable.

EXAMPLE

To print the date on which file "xeq" was last modified:

```
INTERN TEXT buf[] {"01234567890123456789012345678901234567890123"};
FILE fd = open("xeq", READ, 0);
TVEC tvec;

putstr(STDOUT, atime(ltime(&tvec, ftime(fd, YES)), buf),
    "\n", NULL);
```

SEE ALSO

atime, ltime

NAME

getblk - get filesystem block

SYNOPSIS

```
BOOL getblk(fd, buf, bno)
    FILE fd;
    TEXT *buf;
    BLOCK bno;
```

FUNCTION

getblk reads the 512-byte block whose number is bno into buf from the file under control of fd.

RETURNS

getblk returns YES only if exactly 512 characters were read.

EXAMPLE

```
if (!getblk(fd, superbuf, 1))
    error("can't read filsys", NULL);
```

SEE ALSO

mapblk, putblk

NAME

getdn - get device name

SYNOPSIS

```
FILE getdn(s, fname, mode)
    TEXT *s, *fname;
    BOOL mode;
```

FUNCTION

getdn opens a block or character special device using mode as the mode argument to the open call. If the file fname exists and is a block or character special device, it is opened. If the file fname does not exist, to it is prepended "/dev/" for a second try. If fname does exist but is not block or character special, then the block or character special device on which fname exists is opened instead, if that can be determined by calls to devname.

RETURNS

getdn returns a file descriptor for the opened file, or a negative number which is the Idris return code, negated. The name of the opened file, or the best guess at one on failure, is copied to s.

EXAMPLE

```
if ((fd = getdn(buf, fname, pflag ? UPDATE : READ)) < 0)
    error("invalid pathname: ", buf);
```

SEE ALSO

devname, stat

NAME

geti - get inode from filesystem

SYNOPSIS

```
FINODE *geti(fd, buf, ino)
    FILE fd;
    FINODE *buf;
    INUM bno;
```

FUNCTION

geti reads the inode whose number is ino from the filesystem under control of fd. It then copies the inode into buf, converting from filesystem format to in-core format in the process.

geti shares an in-core buffer of 16 inodes (1 block) with puti, and will use the contents of the buffer when appropriate. Thus, the buffer should be disqualified by calling flushi before switching filesystems.

RETURNS

If geti cannot read the necessary inode it will raise the readerr condition. If there is pending inode output from puti and the inode cannot be written, the writerr condition will be raised. Otherwise, geti will return buf, the pointer to the in-core inode.

EXAMPLE

To process the entire inode list of a filesystem:

```
for (i = isize << 4; 1 <= i; --i)
    if (process(pi = geti(fd, &ibuf, i)))
        puti(fd, pi, i);
flushi(fd);
```

SEE ALSO

flushi, puti

NAME

getlinks - read and sort a directory

SYNOPSIS

```
DIR *getlinks(dirname, nentries, size)
    TEXT *dirname;
    BYTES *nentries;
    LONG size;
```

FUNCTION

getlinks allocates an array of size bytes, reads a directory into it, then sorts the entries in lexical order on link names, but with zero inodes at the end. The number of non-null entries in the directory, including . and .. is written at nentries. size is the size in bytes of the directory, dirname.

RETURNS

getlinks returns a pointer to the first directory entry (usually .), or NULL if the directory cannot be read. The pointer is suitable for later use on a free call.

EXAMPLE

To print all the entries in a directory, and then free the allocated space:

```
IMPORT LONG lsize();
BYTES nentries;
DIR *pdir, *p;
STAT dstat;
TEXT *dirname;

if (stat(dirname, &dstat) < 0)
    putstr(STDERR, dirname, " does not exist\n", NULL);
else
    {
    pdir = getlinks(dirname, &nentries, lsize(&dstat->s_mode));
    for (p = pdir; nentries--; ++p)
        putstr(STDOUT, p->d_name, "\n", NULL);
    free(pdir, NULL);
    }
```

NAME

getpw - retrieve a field from the password file

SYNOPSIS

```
TEXT *getpw(matchstr, matchfld, wantfld)
    BYTES matchfld, wantfld;
    TEXT *matchstr;
```

FUNCTION

getpw scans the system password file for a line with a field, specified by matchfld, that matches the NUL terminated string pointed at by matchstr.

Fields in the password file are separated by colons on a text line, as follows:

FIELD	CONTENTS
0	loginid
1	encrypted password
2	user number
3	group number
4	long name
5	home directory
6	shell

Lines are assumed to be no longer than 128 bytes.

getpw remembers the last password line obtained in an internal buffer; consequently multiple calls leading to the same line are reasonably efficient. The line should not be corrupted by the calling program, however, nor should its contents be trusted if other getpw calls intervene.

RETURNS

getpw returns a pointer to the field specified by wantfld, in the matched field. If no match is found, NULL is returned. If the field is found, but the desired field does not exist, the returned pointer will be pointing at newline.

EXAMPLE

To find the loginid of user number 5:

```
TEXT *p, id[8];

if ((p = getpw("5", 2, 0))
    {
    len = instr(p, ":\n");
    cpybuf(id, p, min(len, sizeof (id));
    }
```

FILES

/adm/passwd for the password file.

BUGS

Lines should be up to 512 bytes in the password file.

NAME
inblk - find home block of an inode

SYNOPSIS
```
BLOCK inblk(ino)
    INUM ino;
```

FUNCTION
inblk locates the block number containing ino for any desired inode.

RETURNS
inblk returns the correct block number.

EXAMPLE
To get an inode:

```
FINODE *getino(fd, ino)
    FILE fd;
    INUM ino;
    {
    INTERN FINODE buf[BUFSIZE / sizeof (INO)];

    if (!getblk(fd, buf, inblk(ino)))
        return (NULL);
    else
        return (ioff(buf, ino);
    }
```

SEE ALSO
ioff

NAME

ioff - get inode offset within block

SYNOPSIS

```
TEXT *ioff(s, ino)
    TEXT *s;
    INUM ino;
```

FUNCTION

ioff locates inode number ino within the 512-byte block at s, assuming the correct block has already been read into s.

RETURNS

ioff returns a pointer to the first byte of the inode.

EXAMPLE

To get an inode:

```
FINODE *getino(fd, ino)
    FILE fd;
    INUM ino;
    {
    INTERN FINODE buf[BUFSIZE / sizeof (INO)];

    if (!getblk(fd, buf, inblk(ino)))
        return (NULL);
    else
        return (ioff(buf, ino);
    }
```

SEE ALSO

inblk

NAME

lsize - get size of a file

SYNOPSIS

```
LONG lsize(pi)
    FINODE *pi;
```

FUNCTION

lsize obtains the size of a file in bytes from the size0 and size1 fields in the inode at pi. The inode is assumed to be in native byte order, such as is returned as part of a stat (or fstat) system call.

RETURNS

lsize returns the size as a long integer.

EXAMPLE

```
IMPORT LONG lsize();

nblocks = lsize(&ps->s_mode) >> 9;
```

NAME

lslin - convert inode information to readable form

SYNOPSIS

```
TEXT *lslin(buf, pnode, grp, atim)
    TEXT *buf;
    FINODE *pnode;
    BOOL grp, atim;
```

FUNCTION

lslin fills a 43 character buffer pointed at by buf with a printable representation of information from the file system inode pointed at by pnode. The inode is assumed to be in native byte order, such as is returned as part of a stat (or fstat) system call.

The first character in the returned buffer is the inode type;

'-' for a plain file
'c' for a character special device
'd' for a directory
'b' for a block special device

The next nine characters of this first field specify the read 'r', write 'w' and execute 'x' permissions for the owner, the owner's group and all others, in that order. A '-' in any position indicates that the associated permission is denied. An 's' in place of an 'x' means: set userid if in the owner field, set groupid if in the group field, save text if in the others field. An 'e' in place of an 'x' means much the same, except that the corresponding execute permission is not present.

The next 3 character field specifies the number of links to the inode, right justified.

A space follows.

The loginid occupies the next 8 character field, left justified. If (grp) then the loginid corresponds to the first password file entry whose groupid matches that of the file; otherwise the loginid is for the first entry whose userid matches. If the loginid is not obtainable from the password file, the above key is printed as a decimal number, left justified in the field.

If the inode is not a special device, the next eight character field specifies the size, in bytes, of the file. Otherwise the device's major and minor inode numbers are printed, right justified and separated by a comma.

In either case, a space follows.

The last field is 12 characters long and contains the month, day and time of the date last accessed, if atim is nonzero; otherwise the date of last update. If this time is more than half a year ago, the time subfield is replaced with the year of last update.

RETURNS

lslin returns buf, whose contents are replaced.

EXAMPLE

```
stat(*av, &fi);
lslin(buf, &p->s_mode, YES);
putfmt("%.43p %p\n", buf, *av);
```

which might print:

```
-r-sr-xr-x  1 root       18678 May 06  1980 /bin/rm
```

SEE ALSO

atime

NAME

ltime - convert system time to local time

SYNOPSIS

```
TVEC *ltime(pv, lt)
    TVEC *pv;
    LONG lt;
```

FUNCTION

ltime converts lt, the number of seconds elapsed since 00:00 Jan 1 1970, to a structure at pv giving the time components in local time.

The timezone is determined from the timezone file, if it can be read; it will be read at most once. Otherwise, the default internal values are used, which are correct for Secaucus NJ. Timezone information is available in the external variables:

```
BOOL  _dst {YES};       /* non-zero enables daylight savings */
BYTES _timezone {5};    /* hours west of Greenwich */
BYTES _tzmins {0};      /* minutes west of standard */
TEXT *_edt {"EDT"};
TEXT *_est {"EST"};
```

The initial values shown are the defaults. Note that fractional timezones, such as for Surinam, are supported.

If (12 <= _timezone) ltime subtracts 24 for use in its computation. Thus to get the correct time east of Greenwich, the timezone should be set to 24 minus the number of hours east of Greenwich. The timezone for Japan is 15 (24-9).

RETURNS

ltime returns the pointer pv and fills in the structure at pv.

EXAMPLE

To print the date:

```
IMPORT LONG time();
IMPORT TEXT *_est, *_edt;
TVEC tvec;
INTERN TEXT buf[] {"012345678901234567890123 "};

ltime(&tvec, time(NULL));
putstr(STDOUT, atime(&tvec, buf), tvec.dstf ? _edt : _est,
    "\n", NULL);
```

FILES

/adm/zone for the timezone.

SEE ALSO

atime, vtime

BUGS

This will fail after Feb 28, 2100.

NAME

mapblk - map logical block to physical

SYNOPSIS

```
BLOCK mapblk(fd, pi, lbn)
    FILE fd;
    FINODE *pi;
    BLOCK lbn;
```

FUNCTION

mapblk determines the physical block number corresponding to the logical block number lbn in the file whose inode is at pi. It reads indirect blocks as necessary from the filesystem under control of fd.

RETURNS

mapblk returns the physical block number, if present. Zero is returned if lbn is off the end or inside a hole in the file.

EXAMPLE

```
n = lsize(&ino) + 0777 >> 9;
for (lbn = 0; lbn < n; ++lbn)
    {
    if (pbn = mapblk(fd, &ino, lbn)
        getblk(fd, buf, pbn);
    else
        clrbuf(buf);
    write(STDOUT, buf, BUFSIZE);
    }
```

SEE ALSO

getblk, putblk

NAME

mesg - turn on or off messages to current terminal

SYNOPSIS

```
BOOL mesg(new)
    BOOL new;
```

FUNCTION

mesg allows other users to write to the current terminal when new is non-zero; otherwise it prevents other users from writing to this terminal.

RETURNS

mesg returns YES if the terminal had write permission when the function was called.

EXAMPLE

```
old = mesg(NO);
print();
mesg(old);
```

SEE ALSO

devname

NAME

mkdir - make a directory

SYNOPSIS

```
ERROR mkdir(dname)
    TEXT *dname;
```

FUNCTION

mkdir makes the directory dname, with two entries, ., linking dname to itself, and .., linking dname to its parent. mkdir can only be used by the superuser.

RETURNS

mkdir returns 0 if successful, else a negative number which is the Idris error code negated.

EXAMPLE

```
if (mkdir(name) < 0)
    putstr(STDERR, "can't make directory: ", name, "\n", NULL);
```

NAME

mv - move a file

SYNOPSIS

```
ERROR mv(old, new)
    TEXT *old, *new;
```

FUNCTION

mv creates the link new to a file and removes the link old. If a file named new already exists, it is removed. If old and new are on different filesystems and old is a plain file mv will copy old to new; otherwise, if old is a character or block special device, mv will make the appropriate device; in either case, old is removed. A directory cannot be moved to another filesystem, or to a subtree of itself.

RETURNS

mv returns zero if successful, else a negative number, which is the Idris error return code, negated.

EXAMPLE

```
if (mv("a.out", "xeq") < 0)
    putstr(STDERR, "can't move a.out\n", NULL);
```

NAME

parent - get parent name of a file

SYNOPSIS

```
TEXT *parent(buf, path)
    TEXT *buf, *path;
```

FUNCTION

parent creates a NUL terminated string at buf which is the name of the parent of the file named path. The parent is created by a) stripping off trailing '/' characters; b) stripping off trailing "/." strings; c) stripping off and remembering trailing "/.." strings; d) repeating a-c until there are no more to be stripped; e) partitioning the remainder into LEFT '/' RIGHT, where LEFT is NULL if there is not at least one slash, and RIGHT contains no slash. f) determining the "base" parent by the following table:

LEFT	SLASH	RIGHT	PARENT
none	no	none	.. (or / if path started with a /)
none	no	.	..
none	no	..	../..
none	no	other	.
none	yes	any	/
any	yes	any	<LEFT>

g) adding back the trailing "/.." strings that were stripped in c. h) stripping off leading "./" strings;

For example:

FILENAME	PARENT
abc	.
x/y	x
/a	/
a/b/..	a/..
../c	..
.	..

Note that parent is sufficiently cautious about writing into buf, that parent(name, name) is meaningful, and will work as expected.

RETURNS

parent returns buf.

EXAMPLE

```
stat(parent(buf, path), &pstat);
if (!perm(&pstat, 02))
    putstr(STDERR, "can't remove ", path, "\n", NULL);
```

NAME

perm - test permissions of a file

SYNOPSIS

```
BOOL perm(pst, mask)
    STAT *pst;
    COUNT mask;
```

FUNCTION

perm determines if the current process has all of the permissions requested by mask, for the file whose status, as returned by stat, is pointed at by pst. mask is the inclusive or of 04, to check read permission, 02, to check write permission, and 01, to check execute permission. The permissions are checked according to the real userid and groupid of the process. This means that perm may be called by set-userid programs to test whether the invoker of the program has the required permissions. If the userid of the file is the same as the real userid then the "user access" bits are checked; otherwise, if the groupid of the file matches the real groupid, the "group access" bits are checked; otherwise, the "other access" bits are checked.

RETURNS

perm returns YES if the file has all of the requested permissions. If the user is the superuser, perm returns YES, unless mask requests execute permission and none of the execute permissions are turned on for the file.

EXAMPLE

```
if (!perm(&filestat, 02))
    putstr(STDERR, "can't write\n", NULL);
```

NAME

putblk - put filesystem block

SYNOPSIS

```
BOOL putblk(fd, buf, bno)
    FILE fd;
    TEXT *buf;
    BLOCK bno;
```

FUNCTION

putblk writes the 512-byte block whose number is bno from buf to the file under control of fd. If the write fails, the function returns NO.

RETURNS

putblk returns YES if it can write the whole block else NO.

EXAMPLE

```
if (!putblk(fd, superbuf, 1))
    error("can't write superblock", NULL);
```

SEE ALSO

getblk, mapblk

NAME

puti - put inode to filesystem

SYNOPSIS

```
VOID puti(fd, pi, ino)
    FILE fd;
    FINODE *pi;
    INUM ino;
```

FUNCTION

puti writes the in-core inode pointed at by pi to the filesystem controlled by fd, after converting the inode to filesystem format. puti shares an in-core buffer of 16 inodes (1 block) with geti, and pi is actually just copied and converted into this buffer. Thus, flushi must be called before closing fd, to insure that any pending output is completed.

RETURNS

If there is pending output for another inode block, and it cannot be written, the writerr condition is raised. If the appropriate inode block is not in the buffer shared between geti and puti and it cannot be read, the readerr condition is raised. In any case, there is no useful return value from puti.

EXAMPLE

To process the entire inode list of a filesystem:

```
for (i = isize << 4; 1 <= i; --i)
    if (process(pi = geti(fd, &ibuf, i)))
        puti(fd, pi, i);
flushi(fd);
```

SEE ALSO

flushi, geti

NAME

rdir - read directory on unmounted filesystem

SYNOPSIS

```
DIR *rdir(fd, pi, lno)
    FILE fd;
    FINODE *pi;
    BYTES lno;
```

FUNCTION

rdir obtains the directory entry whose ordinal position is lno within the directory, counting from zero, by reading the unmounted filesystem at fd. It is assumed that pi points at the directory's inode, in native byte order. If lno is zero, or if the block differs from that remembered on the last call to rdir, a new block is obtained by reading from the file under control of fd.

Note well that this function is designed for sequential processing of one directory at a time.

RETURNS

rdir returns a pointer to the link within its current block in memory. If the list is off the end of the directory, rdir returns NULL.

EXAMPLE

```
for (i = 0; pd = rdir(fd, &ino, i); ++i)
    process(pd);
```

SEE ALSO

wdir

NAME

rmdir - remove a directory

SYNOPSIS

```
COUNT rmdir(dname)
    TEXT *dname;
```

FUNCTION

rmdir removes an empty directory, (i.e., one with at most one . entry and at most one .. entry). Directories can only be removed by the superuser.

RETURNS

rmdir returns zero if successful; otherwise: -ENOENT if dname does not exist, -ENOTDIR if dname is not a directory, -EISDIR if directory is not empty, or -EPERM if user does not have write permission for the parent directory.

EXAMPLE

```
if (rmdir(fname) < 0)
    putstr(STDERR, "can't remove ", fname, "\n", NULL);
```

NAME

shell - execute a shell command escape

SYNOPSIS

```
COUNT shell(cmd, fname, flags)
    TEXT *cmd, *fname;
    COUNT flags;
```

FUNCTION

shell invokes the shell command interpreter to parse and execute cmd. It is most often used within programs offering a "! cmd" shell escape or within programs offering a "-c cmd" flag.

Any trailing '\n' before the trailing NUL on cmd is ignored. If fname is not NULL, any occurrence of the sequence "\f" within cmd will be replaced by the string fname.

If the first 3 characters of cmd are "cd\ " then a chdir system call is done with the rest of cmd as an argument. This will affect the caller.

If the first 3 characters of cmd are not "cd\ " then the shell command interpreter is invoked as "sh -c cmd" under the current execution path. If (!(flags & 3)) cmd is invoked as a new process; shell will wait until the command has completed and will return its status to the calling program. If (flags & 1) cmd is invoked as a new process and shell will not wait, but will return the processid of the child. If (flags & 2) cmd is invoked in place of the current process, whose image is forever gone. In this case, shell will never return to the caller.

To the value of flags may be added a 4 if the processing of interrupt and quit signals for cmd is to revert to system handling. They are normally turned off in both the invoker and any new process for the duration of cmd. The value of flags may also be incremented by 8 if the effective userid is to be made the real userid before cmd is executed.

RETURNS

If a chdir is requested, shell will return what chdir does. If cmd cannot be invoked, shell will return failure; If (!(flags & 3)) shell returns YES if the command executed successfully, otherwise NO; if (flags & 1) shell returns the id of the child process, if one exists, otherwise zero; if (flags & 2) shell will never return to the caller.

In all cases, if cmd cannot be executed, an appropriate error message is written to STDERR.

SEE ALSO

xecl, xecv

NAME

vtime - convert system time to Greenwich Mean Time

SYNOPSIS

```
TVEC *vtime(pv, lt)
    TVEC *pv;
    LONG lt;
```

FUNCTION

vtime converts lt, the number of seconds elapsed since 00:00 Jan 1 1970, to a structure at pv giving the time components in GMT.

RETURNS

vtime returns the pointer pv and fills in the structure at pv.

EXAMPLE

To print the date:

```
IMPORT LONG time();
TVEC tvec;
INTERN TEXT buf[] {"012345678901234567890123 GMT\n"};

putstr(STDOUT, atime(vtime(&tvec, time(NULL)), buf), NULL);
```

SEE ALSO

atime, ltime

NAME

wdir - write directory to unmounted filesystem

SYNOPSIS

```
VOID wdir(fd, pi)
    FILE fd;
    FINODE *pi;
```

FUNCTION

wdir writes the last block obtained by rdir to the unmounted filesystem under control of fd. It is presumed that pi points at the inode that rdir has been reading.

Note well that this function is designed for sequential processing of one directory at a time, in conjunction with rdir.

RETURNS

Nothing.

EXAMPLE

```
for (i = 0; pd = rdir(fd, &ino, i); ++i)
    if (badlink(pd))
        {
        fixlink(pd);
        wdir(fd, &ino);
        }
```

SEE ALSO

rdir

NAME

who - read and sort who file

SYNOPSIS

```
WHO *who(n, fname)
    COUNT *n;
    TEXT *fname;
```

FUNCTION

who allocates a buffer large enough to hold the specified file, reads the file contents into it, sorts the entries by tty name, then sets the integer at pn to the number of non-null entries found. The file is presumed to have records like the standard who and log files.

RETURNS

who returns a pointer to the allocated buffer if successful, otherwise NULL.

EXAMPLE

To print the names of the current system users:

```
IMPORT WHO *who();
WHO *pwho, *p;
COUNT n;

pwho = who(&n, WHOFILE);
for (p = pwho; 0 <= --n; ++pwho)
    putfmt("%b\n", p->w_uname, sizeof (p->w_uname));
free(pwho, NULL);
```