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 signicant byte of dev is the device minor number, the more significant byte is its major number.

mode takes the form 'azzlugtrwxrwxrwx', 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