II. Standard Utilities

TABLE OF CONTENTS

Conventions	using the utilities
cat	concatenate files to STDOUT
cd	change working directory
chmod	change the mode of a file
cmds	locate commands along execution path
cmp	compare one or more pairs of files
comm	find common lines in two sorted files
copy	copy one or more files
cp	copy one or more files
crypt	encrypt and decrypt files
cu	call up a computer
date	print or set system date and time
dd	emulate IBM dd card
deque	control stream output of queued files
detab	convert tab to equivalent number of spaces
df	find free space left in a filesystem
diff	find all differences between pairs of files
dn	transmit files uplink or downlink
e	text editor
echo	copy arguments to STDOUT
enque	queue up files for stream output
entab	convert spaces to tabs
error	redirect STDERR
exec	execute a command
exit	terminate a shell script
exp	evaluate binary expression
first	print first lines of text files
goto	branch to label in a shell script
grep	find occurrence of pattern in files
head	add title or footing to text files
kill	send signal to process
last	print final lines of text files
ln	link file to new name
lpr	drive line printer
ls	list status of files or directory contents
mail	send and receive messages
mc	display file in multiple columns
md	maintain a dictionary as needed for spell
mesg	turn on or off messages to current terminal
mkdir	make diectories
mv	move files
nice	execute a command with altered priority
nohup	run a command immune to termination signals
od	dump a file in desired format to STDOUT

page	display files n lines at a time
passwd	change login pass@word
pk	file packing utility
pr	print files in pages
pwd	print current directory pathname
rm	remove files
roff	format text
set	assign a value to a shell variable
setb	change or display stack/heap size
sh	execute programs
shift	reassign shell script arguments to shell variables
sleep	delay for a while
sort	order lines within files
spell	discover potential spelling errors within files
su	set userid
tar	read or write a tar format tape
tee	copy STDIN to STDOUT and other files
test	evaluate conditional expression
time	time a command
tp	read or write a tp format tape
tr	transliterate one set of characters to another
uniq	collapse duplicated lines in files
up	pull files uplink
wait	wait until all child processes complete
wc	count words in one or more files
who	indicate who is on the system
write	send a message to another user

NAME

Conventions - using the utilities

FUNCTION

Each of the utilities described in this section is a separate "program" that may be run, or "invoked", by typing a "command line", usually in response to a "prompt" such as the Idris "% ". This document provides a systematic guide to the conventions that govern how command lines are specified. It also summarizes the layout of the individual utility descriptions that follow, so that you know what information appears where, and in what format.

Command Lines

In general, a command line has three major parts: a program name, an optional series of flags, and a series of non-flag arguments, usually given in that order. Each element of a command line is usually a string separated by whitespace from all the others. Most often, of course, the program name is just the (file) name of the utility you want to run. Flags, if given, change some aspect of what a utility does, and generally precede all other arguments, because a utility must interpret them before it can safely interpret the remainder of the command line.

The meaning of the non-flag arguments strongly depends on the utility being run, but there are five general classes of command lines, presented here (where possible) with examples taken from the portable software tools, since they run much the same way on many different systems:

1) program name and flags followed by a series of filenames. These programs (called filters) process each file named in the order specified. An example is sort.

2) program name and flags followed by a series of arguments that are not filenames, but strings to which the program gives some other interpretation. echo is one such utility.

3) program name, followed by flags, a mandatory argument and other arguments. grep and tr belong to this class.

4) program name and flags followed by a series of "source" filenames, and a single "destination" filename. A filename to be paired with each source name is created by applying "pathname completion" to the destination name; for instance, the destination filename might be a directory of files whose names match the source filenames. These programs (called directed utilities) then perform some operation using each pair of files. diff is one example.

5) program name and flags followed by a command line that the utility executes after changing some condition under which it will run. These tend to be more sophisticated tools, like error, which redirects error messages under Idris.

A summary of the command line classes looks like this:

Class Example Syntax

filter sort <progname> <flags> <files>
string
arguments echo <progname> <flags> <args>
mandatory
argument grep <progname> <flags> <arg> <files>
directed diff <progname> <flags> <files> <dest>
prefix error <progname> <flags> <command>

Note that, in general, <flags> are optional in any command line.

Flags

Flags are used to select options or specify parameters when running a program. They are command line arguments recognized by their first character, which is always a '-' or a '+'. The name of the flag (usually a single letter) immediately follows the leading '-' or '+'. Some flags are simply YES/NO indicators -- either they're named, or they're not -- but others must be followed by some additional information, or "value". This value, if required, may be an integer, a string of characters, or just one character. String or integer values are specified as the remainder of the argument that names the flag; they may also be preceded by whitespace, and hence be given as the next argument.

The flags given to a utility may appear in any order, and two or more may be combined in the same argument, so long as the second flag can't be mistaken for a value that goes with the first one. Some flags have only a value, and no name. These are given as a '-' or '+' immediately followed by the value.

Thus all of the following command lines are equivalent, and would pass to uniq the flags -c and -f:

```
% uniq -c -f
% uniq -f -c
% uniq -fc
```

And each of the following command lines would pass the three flags -c3, -n, and -4 to pr:

```
% pr -c3 -4 -n file1
% pr -4 -nc 3 file1
% pr -n4 -c 3 file1
```

In short, if you specify flags so that you can understand them, a utility should have no trouble, either.

Usually, if you give the same flag more than once, only the last occurrence of the flag is remembered, and the repetition is accepted without comment. Sometimes, however, a flag is explicitly permitted to occur multiple times, and every occurrence is remembered. Such flags are said to be "stacked," and are used to specify a set of program parameters, instead of just one.

Another special flag is the string "--", which is taken as a flag terminator. Once it is encountered, no further arguments are interpreted as flags. Thus a string that would normally be read as a flag, because it begins with a '-' or a '+', may be passed to a utility as an ordinary argument by preceding it with the argument "--". The string "-" also causes flag processing to terminate wherever it is encountered, but, unlike "--", is passed to the utility instead of being "used up", for reasons explained below.

If you give an unknown flag to a utility, it will usually display a hint to remind you of what the proper flags are. This message summarizes the format of the command line the utility expects, and is explained below in the synopsis section of the psuedo-manual page. Should you forget what flags a utility accepts, you can force it to output this "usage summary" by giving it the flag "-help", which is never a valid flag argument. (If a utility expects a mandatory argument, you'll have to say "-help -help" to get past the argument.)

Finally, be warned that some combinations of flags to a given utility may be invalid. If a utility doesn't like the set you've given, it will output a message to remind you of what the legal combinations are.

Files

Any utility that accepts a series of input filenames on its command line will also read its standard input, STDIN, when no input filenames are given, or when the special filename "-" is encountered. Hence sort can be made to read STDIN just by typing:

```
% sort
```

while the following would concatenate file1, then STDIN, then file2, and write them to STDOUT:

```
% cat file1 - file2
```

Naturally, whenever STDIN is read, it is read until end-of-file, and so should not be given twice to the same program.

Manual Pages

The remainder of this document deals with the format of the manual pages describing each of the utilities. Manual pages are terse, but complete and very tightly organized. Because of their general spareness, getting information out of them hinges on knowing where to find what you're after, and what form it's likely to take when you find it. Manual pages are divided into several standard sections, each of which covers one aspect of using the documented utility. So, for clarity, the rest of this document is presented as a psuedo-manual page, with the remarks on each section of a real page appearing under the normal heading for that section.

NAME

name - the name and a short description of the utility

SYNOPSIS

This section gives a one-line synopsis of the command line that the utility expects. The synopsis is taken from the message that the flag -help will cause most utilities to output, and indicates the main components of the command line: the utility name itself, the flags the utility accepts, and any other arguments that may (or must) appear.

Flags are listed by name inside the delimiters "-[" and "]". They generally appear in alphabetic order; flags consisting only of a value (see above) are listed after all the others. If a flag includes a value, then the kind of value it includes is also indicated by one of the following codes, given immediately after the flag name:

Code	Kind of Value
*	string of characters
#	integer (word-sized)
##	integer (long)
?	single character

A '#' designates an integer representable in the word size of the host computer, which may limit it to a maximum as small as 32,767 on some machines. A "##" always designates a long (four-byte) integer, which can represent numbers over two billion. An integer is interpreted as hexadecimal if it begins with "0x" or "0X", otherwise as octal if it begins with '0', otherwise as decimal. A leading '+' or '-' sign is always permitted.

If a flag may meaningfully be given more than once (and stacks its values), then the value code is followed by a '^'.

Thus the synopsis of pr:

```
pr -[c# e# h l# m n s? t* w# +## ##] <files>
```

indicates that pr accepts eleven distinct flags, of which -c, -e, -l, and -w include word-sized integer values, -h, -m, and -n include no values at all, -t includes a string of characters, -s includes a single character, and the two flags +## and -## are nameless, consisting of a long integer alone.

Note that flags introduced by a '-' are shown without the '-'. Roughly the same notation is used in the other sections of a manual page to refer to flags. In the pr manual page, for example, -c# would refer to the flag listed above as c#, while -[c#\ e#\ w#] would refer to the flags "c#\ e#\ w#".

The position and meaning of non-flag arguments are indicated by "metanotions", that is, words enclosed by '<' and '>' (like <files> in the example above). Each metanotion represents zero or more arguments actually to be given on the command line. When entering a command line, you type

whatever the metanotion represents, and type it at that point in the line. In the example, you would enter zero or more filenames at the point indicated by <files>.

No attempt is made in this section to explain the semantics of the command line -- for example, combinations of arguments that are illegal. The next section serves that purpose.

FUNCTION

This section generally contains three parts: an overview of the operation of the utility, a description of each of its flags, then (if necessary) additional information on how various flags or other arguments interact, or on details of the utility's operation that affect how it is used.

Usually, the opening overview is brief, summarizing just what the utility does and how it uses its non-flag arguments. The flag descriptions following consist of a separate sentence or more of information on each flag, introduced by the same flag name and value code given under SYNOPSIS. Each description states the effect the flag has if given, and the use of its value, if it includes one. The parameters specified by flag values generally have default settings that are used when the flag is not given; such default values appear at the end of the description. Flags are listed in the same order as in the synopsis line.

Finally, one or more paragraphs may follow the flag descriptions, in order to explain what combinations of flags are not permitted, or how certain flags influence others. Any further information on the utility of interest to the general user is also given here.

RETURNS

When it finishes execution, every utility returns one of two values, called success or failure. This section states under what conditions a utility will return one rather than the other. A successful return generally indicates that a utility was able to perform all necessary file processing, as well as all other utility-specific operations. In any case, the returned value is guaranteed to be predictable; this section gives the specifics for each utility.

Note that the returned value is often of no interest -- it can't even be tested on some systems. But when it can be tested, it is instrumental in controlling command scripts.

EXAMPLE

Here are given one or more examples of how the utility can be used. The examples seek to show the use of the utility in realistic applications, if possible in conjunction with related programs.

Generally, each example is surrounded by explanatory text, and is set off from the text by an empty line and indentation. In each example, lines preceded by a prompt (such as "% ") represent user input; other lines are the utility's response to the input shown.

FILES

This section lists any external files that the utility requires for correct operation. Most often, these files are system-wide tables of information or common device names.

SEE ALSO

Here are listed the names of related utilities or tutorials, which should be examined if the current utility doesn't do quite what you want, or if its operation remains unclear to you. Another utility may come closer, and seeing the same issues addressed in different terms may aid your understanding of what's going on.

Other documents in the same manual section as the current page are simply referred to by title; documents in a different section of the same manual are referred to by title and section number.

BUGS

This section documents inconsistencies or shortcomings in the behavior of the utility. Most often, these consist of deviations from the conventions described in this manual page, which will always be mentioned here. Known dangers inherent in the improper use of a utility will also be pointed out here.

BUGS

There is a fine line between being terse and being cryptic.

NAME
cat - concatenate files to STDOUT

SYNOPSIS
cat -[b s] <files>

FUNCTION
cat copies the named files to STDOUT in the order specified, thus concatenating their contents. If no <files> are given, STDIN is copied. A filename of "-" also causes STDIN to be copied, at that point in the list of files.

This is the simplest way to list text files.

The flags are:

-b treat all input files as binary. The default is text. This distinction is meaningless under Idris/UNIX.

-s run silently. No messages are written to STDERR or STDOUT. Default is to complain about files that cannot be read.

RETURNS
cat returns success if all files can be read and written.

EXAMPLE
To list the contents of file1:

```
% cat file1
```

To put a wrapper around STDIN:

```
% cat header - footer
```

To concatenate two binary files on most systems:

```
% cat -b lib1.a lib2.a >newlib.a
```

SEE ALSO
first, last, pr

NAME

cd - change working directory

SYNOPSIS

cd <dname>

FUNCTION

cd changes the working directory to <dname>, where <dname> is the name of an existing directory. If <dname> is not specified, the shell variable $H is used.

RETURNS

cd returns failure if it cannot change to the new directory.

If cd returns failure within a shell script, the shell immediately seeks to the end of the script and exits.

EXAMPLE

To change directories to dir2, a "brother" of the current one:

```
% cd ../dir2
```

To then change to dir3, a subdirectory of dir2:

```
% cd dir3
```

SEE ALSO

sh

BUGS

Since it is a shell builtin command, cd cannot be used with prefix commands such as time, nor can metacharacters be used in <dname>.

NAME

chmod - change the mode of a file

SYNOPSIS

chmod -[g o +r r u +w w +x x #] <files>

FUNCTION

chmod changes the mode of each file in the list of file arguments as specified by the flags. The file mode consists of twelve bits, 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 (leftmost) group applies to the user who owns the file, the second to the group that owns the file, and the third to all others.

The flags are:

-g change access for group bits only.

-o change access for other bits only.

+r turn on read access.

-r turn off read access.

-u change access for user bits only.

+w turn on write access.

-w turn off write access.

+x turn on execute access.

-x turn off execute access.

-# change the file mode to the number #.

When -# is specified all other flags are ignored. Typically this is used to set the u, g, and t bits; be sure you know what you are doing. If no flags are specified +x is assumed. Read, write, and execute access changes are applied to all of user, group, and other unless one or more of the flags -u, -g, or -o are specified.

RETURNS

chmod returns success if all flags are coherent, all files exist, and the user issuing the command owns each file specified.

EXAMPLE

To give execute permission only to the user and his group:

```
% chmod +x -ug file1 file2
```

NAME

cmds - locate commands along execution path

SYNOPSIS

cmds -[X* a x] <commands>

FUNCTION

cmds locates all occurrences, in the user's execution path, of the named commands. If no commands are named, all commands in the path are listed. A file is considered to be a command only if it has execute permission for the user. Directories are never treated as commands, regardless of permissions or flags.

The flags are:

-X* use * as the execution path. The default is the inherited path, almost invariably the same as the $X variable in the shell.

-a list all files, even if they are not executable.

-x treat the file as a command if it has execute permission for anyone.

Note that cmds checks only the permissions of a file, not its contents. Thus, a file may be listed as executable even if it doesn't make sense to execute it on the current machine.

RETURNS

cmds returns success if all of the files exist and all the directories to be scanned are readable.

EXAMPLE

To look for executable files named test and xeq in one's path:

```
% cmds test xeq
./test /bin/test
./xeq
```

To look for text files in specific directories:

```
% cmds -aX'../|/lib/|/sys/lib/headers/' std.h sys.h res.h
/lib/std.h /sys/lib/headers/std.h
/lib/sys.h /sys/lib/headers/sys.h
../res.h /lib/res.h
```

To find the names of executable files in specific directories:

```
% cmds -xX"/crp/bin/|/usr/bin/"
bad directory: /crp/bin/
/usr/bin:
dbl
dbl.old
docbugs
exp
get
```

letter
memo
page
see
show

NAME

cmp - compare one or more pairs of files

SYNOPSIS

cmp -[a b h l o s u] <files> <dest>

FUNCTION

cmp does a bytewise comparison between each source file named in the list <files>, and a corresponding destination file whose name is derived from the filename <dest>. Any differences in successive pairs of files are reported to STDOUT. By default, cmp outputs only the first difference between each pair, naming the files being compared and the position of the difference, expressed as an offset in lines and bytes from the start of each file.

The flags are:

-a report differences as ASCII characters, with offsets in decimal.

-b treat input files as binary. The default is text, and STDIN is always treated as text. This distinction is meaningless under Idris/UNIX.

-h report differences and offsets in hexadecimal.

-l report all differences between each pair of files. The information output for each difference is the differing bytes and their offset in bytes from beginning of file.

-o report differences and offsets in octal.

-s compare files silently. Nothing is written to STDOUT; only the return value of cmp indicates whether a difference was found.

-u report differences and offsets in unsigned decimal.

At most one of -[l\ s] may be specified, and at most one of -[a\ h\ o\ u] may be given. Line offsets output are counted from one, byte offsets from zero; offsets and differing bytes are output in signed decimal unless otherwise requested.

If more than one source file is given, a heading line consisting of each filename followed by a colon and a newline is written to STDOUT before that file is compared. A source filename of "-" is taken as STDIN.

The last filename given is always taken to be <dest>, and is used to determine a destination file by the following procedure: Each filename from <files> is appended to <dest> using pathname completion. If a file exists with the resulting name, it is used as the destination file to be compared with the current source file. When no file exists with the resulting name, an error occurs if more than one source file was given, while if only one source file was given, the original, unmodified <dest> is used as the destination filename. In the latter case, a <dest> of "-" refers to STDIN. No promises are made if STDIN is specified both as a

source and a destination file.

RETURNS

cmp returns success if all pairs of files are identical and all files are readable.

EXAMPLE

If file1 is the line "this is a test" and file2 is the line "this is not a test", a comparison might yield:

```
% cmp -al file1 file2
    8   a   n
    9       o
    11  e
    12  s   a
    13  t
    14  \n  t
EOF reached on file1
```

And file1 and file2 could be compared against two files of the same name in the directory dir with the command:

```
% cmp file1 file2 dir
```

SEE ALSO

comm, diff

NAME

comm - find common lines in two sorted files

SYNOPSIS

comm -[d* 1 2 3] +[a b d l n r t? #.#-#.#] file1 file2

FUNCTION

comm reads input from two sorted files simultaneously, checking for lines common to both files. Unless otherwise specified, comm will output all lines in three columns, the first for all lines unique to file1, the second for all lines unique to file2, and the third for all lines common to both file1 and file2. If "-" is specified as one of the files, STDIN is assumed.

The flags are:

-d* delimit columns with string *. The default is the tab character.

-1 print lines unique to file1.

-2 print lines unique to file2.

-3 print lines common to both files.

Note that the order of the flags 1, 2, and 3 determine the order in which the output is displayed. If no flags are specified, -1 -2 -3 are assumed in that order.

In addition, up to ten ordering rules may be specified: if the first rule results in an equal comparison then the second rule is applied, and so on until lines compare unequal or the rules are exhausted.

Rules take the form:

[adln][b][r][t?][#.#-#.#]

where

a - compares character by character in ASCII collating sequence. A missing character compares lower than any ASCII code.

b - skips leading whitespace.

d - compares character by character in dictionary collating sequence, i.e., characters other than letters, digits, or spaces are omitted, and case distinctions among letters are ignored.

l - compares character by character in ASCII collating sequence, except that case distinctions among letters are ignored.

n - compares by arithmetic value, treating each field as a numeric string consisting of optional whitespace, optional minus sign, and digits with an optional decimal point.

r - reverses the sense of comparisons.

t? - uses ? as the tab character for determining offsets (described below).

#.#-#.# - describes offsets from the start of each text line for the beginning (first character used) and, after the minus '-', for the end (first character not used) of the text field to be considered by the rule. The number before each dot '.' is the number of tab characters to skip, and the number after each dot is the number of characters to skip thereafter. Thus, in the string "abcd=efgh", with '=' as the tab character, the offset "1.2" would point to 'g', and "0.0 would point to 'a'. A missing number # is taken as zero; a missing final pair "-#.#" points just past the last of the text in each of the lines to be compared. If the first offset is past the second offset, the field is considered empty.

If no tab character is specified, each contiguous string of whitespace will be taken as a tab. Thus, in the string "\ ABC\ \ DEF\ GHI", the offset "3" would point to 'G'.

Only one of a, d, l, or n may be present in a rule; if none are present, the default is a. In a given rule, letters appearing after "#.#-#.#" will be ignored.

Null fields compare lower in sort order than all non-null fields, and equal to other null fields.

RETURNS

comm returns success if the flags are valid, and all the files can be open.

EXAMPLE

To list all files in a glossary, but not in a dictionary:

```
% comm -1 glossary dictionary
```

SEE ALSO

sort, uniq

NAME

copy - copy one or more files

SYNOPSIS

copy -[b] <files> <dest>

FUNCTION

copy copies each source file named in the list <files> to a destination file whose name is derived from the filename <dest>.

The flag is:

-b open all files as binary. The default is text, and STDIN and STDOUT are always treated as text. This distinction is meaningless under Idris/UNIX.

If more than one source file is given, a heading line consisting of each filename followed by a colon and a newline is written to STDOUT before that file is copied. A source filename of "-" is taken as STDIN.

The last filename given is always taken to be <dest>, and is used to find a destination file by the following procedure. Each source filename from <files> is appended to <dest> using pathname completion. If a file can be created with the resulting name, it is used as the destination file to which the current source file will be copied. When a file with the resulting name cannot be created, an error occurs if more than one source file was given, while if only one source file was given, the original, unmodified <dest> is used as the destination filename. In the latter case, a <dest> of "-" refers to STDOUT.

Under Idris or UNIX, an attempt is made to make the mode of the destination match the mode of the source.

RETURNS

copy returns success if no error messages are written to STDERR, that is, if all opens, creates, reads, and writes succeed.

EXAMPLE

To copy the file hither to the file yon:

```
% copy hither yon
```

This will make a copy of hither named yon; any previous contents of yon will be replaced by the contents of hither.

SEE ALSO

cat

NAME

cp - copy one or more files

SYNOPSIS

cp -[c d r s] <files> <dest>

FUNCTION

cp copies each of the source files in the list <files> to <dest>. A source filename of "-" is taken as STDIN; no promises are made if STDIN is specified more than once. If only one source file is specified and the fully qualified destination file does not exist, the destination is taken to be <dest>. A <dest> of "-" is taken as STDOUT.

If more than one source file is being copied, a heading line containing each filename followed by a colon and a newline is written to STDOUT before that file is copied. The destination file in this instance must be a directory file. Pathname completion is used, to determine the destination file, i.e., for each source file a destination filename is formed by appending to <dest> a '/' followed by the longest suffix of the source filename not containing a '/'.

If the source file is a directory, -d is specified, and <dest> either does not exist or is an ordinary file, cp will make <dest> a directory. Whole trees may be copied if -r is specified.

The mode of a newly created destination file is made to match that of the source.

The flags are as follows:

-c copy owner. When <dest> is a new file created by cp, the owner is made to match the owner of the source. This flag works only for the superuser.

-d permit recursive directory creation. When source is a directory and <dest> does not exist or is an ordinary file, cp will create a directory. Meaningful only in conjunction with the -r flag.

-r when source is a directory, recursively descend subtrees below it, copying all entries.

-s run silently. No messages are written to STDERR or STDOUT.

RETURNS

cp returns success if all opens, creates, reads, and writes succeed.

EXAMPLE

To copy the file hither to file yon:

```
% cp hither yon
```

This will make a copy of hither in yon. Any previous contents of yon are replaced by the contents of hither.

To create the directory newdir and copy olddir and all of its entries into it:

```
% cp -rd olddir newdir
olddir/file1:
olddir/file2:
olddir/file3:
```

SEE ALSO

cat, mv, tee

BUGS

An attempt to copy a tree to a lower level of itself will cause mayhem.

NAME
crypt - encrypt and decrypt files

SYNOPSIS
crypt -[b d n o* p*] <files>

FUNCTION
crypt takes a password from the command line or from STDIN (prompting if necessary), and uses it to encrypt or decrypt the files in the list <files>. If no <files> are given, crypt takes input from STDIN. A filename of "-" also causes STDIN to be read, at that point in the list of files. By default, the encrypted or decrypted data is output to STDOUT.

The flags are:

-b treat all files as binary.

-d decrypt the input.

-n do not delete NULs in the output of a decryption.

-o* place output in the file *.

-p* use the string * as password.

If -d is not specified, encrypting the input is assumed. Only the first eight characters of the password input (or the characters up to the first newline, whichever are fewer) are significant. Passwords are NUL-padded to an eight-character length. Under Idris/UNIX, if an interactive STDIN is prompted for a password, echoing will be disabled beforehand, and all input up to the first newline will not appear at the terminal. Note that combining password and data input from STDIN is dangerous and not very useful.

RETURNS
crypt returns success if no error messages are written to STDERR, that is, if all opens, creates, reads, and writes succeed.

EXAMPLE
Given the file cleartext, the following would encrypt it to the file muddy, using the password "chortle":

```
% crypt -pchortle -omuddy cleartext
```

The encrypted file muddy could be decrypted with the single command:

```
% crypt -d muddy
password:
```

NAME

cu - call up a computer

SYNOPSIS

cu -[s* l* w#]

FUNCTION

cu connects your terminal to another tty file, refered to as a "link". Each character you type is written to the link, and a receive process started by cu reads each character from the link and writes it to your terminal. To another computer this link line appears to be a terminal; the link can be hardwired or connected via modem.

cu is mostly transparent, except that some escape sequences you can type cause local action. You may send or receive files over the link, which is done with or without an error checking protocol, depending upon whether you are talking to another Idris system or to alien software. You may also run local Idris commands by invoking a shell process, much the same as from the editor.

The flags are:

-s* set the speed of the link to *. The link speed must match the speed of the remote computer and your terminal. The speed can be one of the baud rates {0, 50, 75, 110, 134.5, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, exta, or extb}. The default is 1200 baud.

-l* define the name of the link file to be *. The default is /dev/lnk0, which is usually an alias for a free tty file.

-w# write # characters per second to the link when transmitting a file (\< file, described below). Default is 100, meaning that cu will write 100 characters then sleep for one second. This value depends on the link speed, and the ability of the remote computer to absorb the characters.

Escape sequences begin with a backslash '\'. Nothing is echoed when you type the backslash; cu waits for the next character typed. The following escapes are recognized by cu, but not by the receive process:

\<filename write the file filename on the local system to the link. The sending speed is governed by the -w flag desribed above. cu takes its input from this file instead of your terminal, until end of file. The characters are sent with no protocol. There need be no cooperating software in the remote computer; you appear to be a very fast typist.

\ctl-Q # - turn throttling on or off. If the digit # is 1 to 9, # is the number of seconds between the automatic transmission of ctl-S and ctl-Q. The delay between them is always one second. If # is 0, or anything else, throttling is turned off. Throttling can be a useful feature if your system is slow, or if you are reading into a file (see \> file below), provided that the remote computer honors the throttling protocol.

\u -[u 1* o* p* a*] <files> - send <files> from your system "uplink" to the remote Idris system in a packet format, with error checking. At the remote end, you must invoke the up utility first, which receives the filenames and packets and builds the files in the remote computer. The \u command actually starts the dn program to do the file transfer. dn can deal with multiple files, and can construct new pathnames based on the flags. See the manual page on dn for details.

\q - quit. The link is closed and cu exits.

\!<command> - execute a shell command. All characters you type after \! up to the next carriage return or line feed are assembled into a command line with "sh -c " prepended and passed to a shell process. cu waits for the invoked shell to terminate.

The following escape sequences are recognized by the receive process. These must be presented by the remote computer either by running a program or by causing them to be echoed:

\> [>*]file read all input from the link into the file named. A leading > means append to the end of the file, don't discard its old contents first. A leading * means don't clear the parity bit of each character received. This permits binary files to be transferred, provided the data link is transparent to eight-bit bytes.

All output from the remote computer is sent to the file named on your system; if the file cannot be created, the output comes to the terminal. Once output is redirected to a file, the only escape sequence recognized is \. which closes the file and resumes output to the terminal.

RETURNS

cu returns success if the link was established.

EXAMPLE

```
% cu -s9600
login:
```

FILES

/bin/dn for \u command, /odd/alarm for read timeout, /odd/recv for receiving the link, /odd/throttle for sending ctl-S and ctl-Q automatically.

SEE ALSO

dn, up

BUGS

Interrupting packet mode transmission can be a problem. The DEL key interrupt will stop the \u command; however you may need to type \. twice to get the remote up utility to exit. The DEL key will interrupt the remote system during \> (receive into local file); however you cannot see anything at your terminal until you get \. echoed. \q will fix everything -- it exits cu and disconnects the link.

NAME

date - print or set system date and time

SYNOPSIS

date -[c i t y# +#] <yymmddhhmm>

FUNCTION

date writes the current date and time to STDOUT. If flags other than -c or -t are specified, the current date is altered as specified before being output. If the flags -c or -t are specified, the system date is never altered, regardless of any other flags that are specified.

The flags are:

-c output date in canonical form, but don't alter the system date.

-i get <yymmddhhmm> from STDIN

-t output date, but don't alter the system date.

-y# alter the current year to #. If the number given is less than 100 then 1900 is added to it.

+# alter the system date by adding # minutes to it.

The system date may also be set by <yymmddhhmm>, a character string up to 10 characters indicating year, month, day, hour, and minute. A short string leaves unchanged the fields that are unspecified on the left.

A new date is written to /adm/date, a log entry is appended to /adm/log, and a sync is done. Errors in opening these files are ignored.

Note that +0 causes the current date to be written to /adm/date and @/adm/log, and does a sync. This is useful for time stamping the root filesystem before a system shutdown.

Only the superuser can change the system date or do a date +0.

RETURNS

date returns success if it can convert the flags and/or date string given into a reasonable date and if it can honor requests to change the system date.

EXAMPLE

```
# date
Tue Aug 04 12:20:03 1981 EDT
# date +3
Tue Aug 04 12:23:03 1981 EDT
```

Or, to set the hour to 11:00 AM:

```
# date 1100
Tue Aug 04 11:00:00 1981 EDT
```

FILES

/adm/date for the new date, /adm/log for logging date changes, /adm/zone for determining the local time zone.

NAME

dd - emulate IBM dd card

SYNOPSIS

dd -[bs# b# c## ib# ob# o* si## so## s##] <files>

FUNCTION

dd copies the input <files> given, or STDIN if none, to the output file specified, doing reads and writes of exactly the length requested on its command line. This behavior is important when reading from and/or writing to character special devices, since the characteristics of external records are often determined by the actual byte count on each read or write call.

The flags are:

-bs# set buffer size for both input and output to # bytes. Default is 512.

-b# open all files as binary, instead of text, and create the output file with a record size of # bytes. This flag is meaningless under Idris/UNIX.

-c## copy only ## input records. By default, the copy proceeds until EOF on input.

-ib# set input buffer size to # bytes. By default, the size used is that given with -bs, or 512 if no -bs flag appears.

-ob# set output buffer size to # bytes. By default, the size used is that given with -bs, or 512 if no -bs flag appears.

-o* write output to file *. By default, output is written to STDOUT.

-si## seek to byte ## on input before starting the transfer. By default, no seek is done.

-so## seek to byte ## on output before starting the transfer. By default, no seek is done.

-s## skip # input records before starting the transfer. By default, none are skipped.

If both -si and -s are specified, the seek is done before the records are skipped.

If no input files are given, STDIN is read. If "-" is encountered as an input file, then STDIN is read at that point in processing <files>. In either case, it is always treated as text. If more than one input file is given, the name of each file, followed by a colon and a newline, is output to STDERR before it is copied. If an input file ends with a partial record, dd transfers the record without modification.

dd immediately exits if unable to open an input file, or on any read or write error. Where the host operating system permits, dd will also exit on receipt of an interrupt signal. On exit, dd outputs the number of full records (and any additional bytes) input and output.

RETURNS

dd returns success if it runs uninterrupted and encounters no open, read or write errors.

EXAMPLE

To copy 512 bytes after skipping 16 in the file boot.o:

```
% dd -bs1 -s16 -c512 -o bootstrap xeq
```

SEE ALSO

cat, cp, pr

BUGS

If an input file ends with a partial record, the first read from the next file in sequence (if any) will be of as many bytes as needed to fill the input buffer (so that a full output record can be written). The two partial records read will be counted in the transfer summary as one full input record.

NAME

deque - control stream output of queued files

SYNOPSIS

deque -[a aft* bef* c* dir* l* n o* w#]

FUNCTION

deque despools files made by enque. It scans the spool directory and runs a specified program once for each enqueued file found there. The files are considered on a first-in first-out basis.

The flags are:

-a seek to the end of STDOUT before writing to it. Generally used with "-o*".

-aft* take * as a command to execute once after all despooling is complete.

-bef* take * as a command to execute once before any despooling occurs.

-c* take * as a command to execute that is sufficient to dequeue each file. Any occurrence of "\f" within the command is replaced with the name of the file currently being despooled. The file is removed after despooling. The default value for this flag is "/bin/cat \f".

-dir* take * as the name of the spool directory from which to get queued up input. Default is the current directory.

-l* take * as the name of a lock file whose existence indicates that a despooler is already running. Unless * contains a '/', the name of the spool directory is prepended to it. If "-n" is also specified, deque will wait until it can create this file itself. The default name is "deque.lock" in the spool directory.

-n Wait for any locks to clear and start a new despooler. Default is to go away quietly if the lock is present.

-o* open * as STDOUT for the current process and all invoked commands. If "-a" is given, the file is opened for appending; otherwise, it is created anew. If the file cannot be accessed, deque senses this as the interlock mechanism. Thus if "-n" is specified, deque will wait until the file can be accessed; otherwise deque silently exits.

-w# wait # seconds after despooling all files found on the current scan of the spool directory, then rescan the directory for new files. This rescanning continues indefinitely. Default is to make a single pass over the spool directory.

The command named with "-bef*" is invoked when despooling is ready to begin, i.e., when the lock file and any output file have been opened, and at least one spooled file encountered. The command named with "-aft*" is invoked when deque exits, and only if the "-bef*" command has already been run.

All commands are invoked from the directory in which deque was run. The effective userid and real userid of the invoker are passed on to the commands run. They also share the execution path of the invoker.

The three "terminal" interrupts (the "interrupt", "quit" and "broken pipe" signals) will terminate any "-c*" command currently being executed by deque. The "-aft*" and "-bef*" commands ignore these signals, as does the deque process itself while it is running any command. If deque is not running a command, these signals will cause deque to terminate.

If a file named "deque.pause" is found in the spool directory, deque suspends execution until "deque.pause" is removed. If a file named "deque.quit" is found in the spool directory, or if the lock file created by deque is removed, deque executes the command specified by the flag "-aft*", removes any "deque.quit" file and exits, leaving unprocessed spooler files alone. deque checks for these control files before each scan of the spool directory, and before each spooled file is processed.

deque was designed to be invokable by anyone as a general utility. As such, it runs with the privileges of the invoker. On a fairly lax system it is permissible for deque to be invoked with a few preset flags via a shell script. On a more secure system, deque will need to be invoked by a program with userid set to the owner of the spool directory, and with all flags pre-specified and terminated by "--" to prevent security gaffes. Better yet is to start an instance of it at system startup via the @/adm/init control file.

RETURNS

deque returns success unless forced to terminate early. Early termination can be caused by a terminal interrupt, by the creation of a "deque.quit" file, or by the deletion of the lock file.

EXAMPLE

A script to print every file put into the current directory by enque:

```
: turn off echoing yet take commands
: uses pr to force each file onto a new page
deque  -w5 -n \
    -bef "stty -echo" \
    -c "pr -h \f" \
    -aft "stty +echo" \
&
```

A script to control the output generated above:

```
: goes by several names in the spool directory
: q - quit printing
: p - pause printer
: g - restart printer
set 0 -s/ ./$0
test $0 == 'q' && : >deque.quit && exit
test $0 == 'p' && : >deque.pause && exit
test $0 == 'g' && rm deque.pause && exit
```

FILES

"." is used as the default spool directory, "/<spooldir>/[0-9]^:[0-9]^" as the acceptable pattern for spooled files, and "/bin/sh" as the interpreter for commands run by the flags -[aft* bef* c*].

SEE ALSO

enque

NAME
detab - convert tab to equivalent number of spaces

SYNOPSIS
detab -[e# #^] <files>

FUNCTION
detab reads the named files, or STDIN if none are given, and writes the files to STDOUT in the order specified, expanding tabs to spaces in the process.

Flags may be used as follows to set tab stops: If no tab stops are explicitly given, tabs are set every four columns. If explicit tab stops are given, tabs are set in every column after the rightmost explicitly given stop.

-e# set tab stops every # columns, beginning after the highest numbered column explicitly set by a -# flag, or after column 1 if none is given. By default, -e1 is assumed if any -# flags appear, and -e4 if none do.

-#^ set a tab stop in column #; columns are numbered from 1. Up to 32 stops may be given.

detab properly interprets spaces, backspaces, tabs, and newlines. All other non-printing characters are taken as zero length. A tab is always mapped to at least one space.

RETURNS
detab succeeds if all of its file reads and writes are successful.

EXAMPLE
To set tab stops in columns 10, 18, 26, 34, etc.:

```
% detab -10 -e8 file1
```

SEE ALSO
entab

NAME
df - find free space left in a filesystem

SYNOPSIS
df -[b i] <files>

FUNCTION
df indicates the amount of free space left in one or more filesystems. If no <files> are specified, or <files> is not a filesystem, df uses the device on which the current directory or the <files> resides. If any of <files> doesn not exist, to it is prepended "/dev/" for a second try. The output of df is preceded with that filesystem name.

The flags are:

-b output to STDOUT the number of free blocks versus the total number of blocks in each filesystem.

-i output to STDOUT the number of free inodes versus the total number of inodes in each filesystem.

If no flags are given, the default is "-b". The output for each filesystem is preceded by the filesystem name. All numbers output are in decimal.

RETURNS
df returns success if there is at least one unallocated inode on the filesystems specified and the freelist on each filesystem is uncorrupted.

EXAMPLE
To find the name of the current filesystem and print the number of free inodes left on it:

```
% df -i
/dev/ry1:
free inodes:  200 / 384
```

NAME
diff - find all differences between pairs of files

SYNOPSIS
diff -[e] <files> <dest>

FUNCTION
diff summarizes the differences between each source text file named in the list <files>, and a corresponding destination file whose name is derived from the filename <dest>. Differences are represented as the changes needed, on a line by line basis, to transform the "old" source file into the "new" destination file.

Each group of changes is summarized as a line number range in the old file, followed by an arrow, followed by the line number range in the new file into which the lines have been changed. The change involved is expressed as an operation defined by the text editor e: an 'a', 'c', 'd', or 'm' indicates that the change represents an append, change, delete, or move operation, respectively. This summary is followed by the text of the lines it refers to, the old lines preceded by the heading "old:", and the new lines by the heading "new:". If old lines are deleted, no new lines are output; if new lines are appended, no old lines are output; and if lines are moved they are printed only once, with no headings, since they are the same in both files.

The flag is:

-e produce a script for e that may be used to transform the old file into the new file.

If more than one file is given in <files>, a heading line consisting of each filename followed by a colon and a newline is output to STDOUT before that file is compared. A filename of "-" is taken as STDIN.

The last filename given is always taken to be <dest>, and is used to determine a destination file by the following procedure: Each filename from <files> is appended to <dest> using pathname completion. If a file exists with the resulting name, it is used as the destination file to be compared with the current source file. When no file exists with the resulting name, an error occurs if more than one file was given in <files>, while if only one file was given, the original, unmodified <dest> is used as the destination filename. In the latter case, a <dest> of "-" refers to STDIN. No promises are made if STDIN is specified both as a source and a destination file.

RETURNS
diff returns success if all files are readable, and if all pairs of files are identical.

EXAMPLE
An atypical (but illustrative) comparison might yield:

```
% diff otext ntext
1,3 d
```

```
old:
    These lines
    have been
    deleted from otext.
4 -> 5,6 a
new:
    These 2 lines
    have been added.
10,12 -> 20 m
    These 3
    lines have
    moved.
13,14 -> 25 c
old:
    Two old lines
    have been replaced.
new:
    by one new line.
```

The command line:

```
% diff -e otext ntext
```

produces an edit script that, when input to e, would transform otext into ntext.

To find differences between files of the same name located in different directories:

```
% diff file1 file2 file3 dir
file1:
file2:
file3:
```

This command compares files in the current directory to files of the same name in the directory "dir".

SEE ALSO

cmp, comm, e

NAME
dn - transmit files uplink or downlink

SYNOPSIS
dn -[u l* o* p* a*] <files>

FUNCTION
dn is invoked by the call up utility cu to send files uplink, and is invoked on the remote computer by the user to send files downlink. For each file given in <files>, dn creates a filename and sends it to the local or remote system, along with the file contents in packet format. This format provides for error checking and synchronization with the remote computer. Flags to dn control the way files are named in the receiving computer system.

The flags are:

-a* append * to each filename given. This flag may be combined with -p* below.

-p* prepend * to each filename given, after removing all characters in filename up to the rightmost '/'.

-o* concatenate all files to the output file * in the remote system. The -a and -p flags are ingored.

-l* use * as the link input and output file. Default is STDIN and STDOUT.

-u run in "up" mode. Used when cu calls dn to send files uplink.

dn can be made to terminate by typing \. or by hitting the DEL key.

RETURNS
dn returns success if all operations succeed.

EXAMPLE
Getting files into /sys/src:

```
% dn -p/sys/src/ *.c
```

SEE ALSO
cu, up

FILES
/odd/alarm for read timeouts.

NAME

e - text editor

SYNOPSIS

e -[blk# b e n p s v w#] <file>

FUNCTION

e is an editor for text files. It operates by copying <file> if it exists into an internal buffer and then accepting commands from STDIN and writing responses to STDOUT. If <file> does not exist, it can be created by entering text into the buffer. The buffer is maintained as a temporary file, permitting editing of files substantially larger than available memory; practical limits are typically on the order of several thousand lines and half a million characters. Commands exist for copying text files into and out of the internal buffer, for modifying the buffer, and for displaying portions of the buffer. Changes made to the buffer do not affect external files in any way until a w (write) command is given.

The flags are:

-blk# block the temporary file into lines of size # characters. Lines will be displayed with that length; the newline character is not treated as a line terminator. The range of # may be from 1 to 510. The default value 0 causes the editor to treat files as text and newline characters as terminators.

-b# treat input file as binary with a record size of #. The default is text. This distinction is meaningless under Idris/UNIX.

-e echo to STDERR a trace of each command line after it is accepted. Useful for debugging when the editor is accepting command lines from a redirected STDIN.

-n precede displayed text lines with a line number, and prompt for lines to be appended with a line number.

-p display a prompt character before accepting each editor command line.

-s suppress character counts for e, r, and w commands and suppress error messages.

-v turn on verbose prompting and display last line affected by a command.

-w# set screen width to # characters for l command (default is 72).

If <file> is given on the command line, e behaves as if the first command entered were "e <file>".

e is line oriented, but imposes no structure on lines. The number of a line, i.e. the relative position of the line from the start of the buffer, is used for display and accessing purposes. However, these numbers are not recorded with the line. Moreover, e attaches no significance to particular columns or positions in a line.

A command line to the editor consists of: optional "line addresses", a single-character command, possible additional parameters, and the usual terminating newline. The line addresses typically specify the inclusive range of lines in the buffer over which the command is to act. Adjacent line addresses are separated by a comma ´,´ or, in the special instance described below, by a colon ´:´. Every command which needs line addresses has default addresses, so line addresses can often be omitted; e complains if a line address is referred to that doesn´t exist. Each command is terminated by a newline. However, more than one command may be entered per line by separating them with semicolons and terminating the last command with a newline.

The most common way to enter text into the buffer is by means of an a, c, or i command. The line addresses and single character commands are entered on the initial line. e collects subsequent lines of text entered by the user and places them in the appropriate place in the buffer. In these collected lines, command characters, line addresses, etc. lose their special meaning and are stored as literal characters, i.e., exactly as they were entered. Lines are so collected until a line is encountered that contains only a period ´.´. This signals the end of inserted text and the line containing the period is not copied into the buffer. Subsequent lines are interpreted as commands once again. An alternate form may be used to enter exactly one line of text into the buffer -- follow the a, c, or i with a single space and any characters to be inserted, terminated with a newline. e will now regard any subsequent characters as commands.

In commands where special meanings apply to certain characters, the special meaning may usually be nullified by preceding the characters with a backslash ´\´. This usage of the backslash may itself be turned off by preceding the backslash with yet another backslash.

Throughout the editor, frequent use is made of the notion of <u>regular expression</u>. A regular expression is a shorthand notation for a sequence of target characters contained in a text file line. These characters are said to "match" the regular expression. The following regular expressions are allowed:

An ordinary character is considered a regular expression which matches that character.

The character sequences "\b", "\f", "\n", "\r", "\t", "\v", in upper or lower case, are regular expressions each representing the single character cursor movements of, respectively: backspace, formfeed, newline, carriage return, tab, vertical tab. Additionally, any single character in the character set may be represented by the form "\ddd" where ddd is the one to three digit octal representation of the character; this is the safest way to match most non-printing characters, and the only way to match ASCII NUL "\0".

A ´?´ matches any single character except a newline.

A ´^´ as the leftmost character of a series of regular expressions constrains the match to begin at the beginning of the line.

A '^' following a character matches zero or more occurrences of that character. This pattern may thus match a null string which occurs at the beginning of a line, between pairs of characters, or at the end of the line. A '^' enclosed in "\(" and "\)", or following either a '\' or an initial '^', is taken as a literal '^', however.

A '^' in any position other than the ones mentioned above is taken as a literal '^'.

A '*' matches zero or more characters, not including newline. It is conceptually identical to the sequence "?^".

A character string enclosed in square brackets "[]" matches a single character which may be any of the characters in the bracketed list but no other. However, if the first character of the string is a '!', this expression matches any character except newline and the ones in the bracketed list. A range of characters in the character collating sequence may be indicated by the sequence of: <lowest character>, '-', <highest character>. ([z-a] never matches anything.) Thus, [ej-maE] is a regular expression which will match one character that may be E, a, e, j, k, l or m. When matching a literal "-", the "-" must be the first or last character in the bracketed list, or must immediately follow a range, as in [0-9-+]; otherwise it is taken to specify a range of characters.

A regular expression enclosed between the sequences "\(" and "\)" tags this expression in a way useful for substitutions, but otherwise has no effect on the characters the expression matches. (See the s command for further explanation.)

A concatenation of regular expressions matches the concatenation of strings matched by individual regular expressions. In other words, a regular expression composed of several subexpressions will match a concatenation of the strings implied by each of the individual subexpressions.

A '$' as the rightmost character after a series of regular expressions constrains the match, if any, to end at the end of the line prior to the newline.

A null regular expression standing alone stands for the last regular expression encountered.

Note that arbitrary grouping and alternation are not fully supported by this notation, as the text patterns utilized are not the full class of regular expressions beloved by mathematicians. They are, however, remarkably comprehensive.

Lines are addressed in several ways. A line address is simply a series of zero or more terms. The editor, for instance, uses the terms "current line" and "last line" to keep track of the numbers of lines in the buffer. The "current line" is typically the most recent line affected by the previously entered command. For specific values, see the command descriptions below. The "current line" is specified by the character '.'

(period or dot); in the descriptions following "dot" will be used. The last line of the buffer is known by the editor as '$'. In addition, the user can address a particular line in the buffer by specifying a decimal number n. The whole set of acceptable terms is as follows:

The character '.' addresses the current line, dot.

The character '$' addresses the last line in the buffer.

A decimal number n addresses the nth line of the buffer relative to the beginning of the buffer.

"'x", i.e. the sequence of single quote followed by a lowercase letter, addresses a line that has been previously assigned that letter as described in the k command below.

A regular expression enclosed in '/', causes a forward context search of the buffer. The context search starts with the line after the current line and proceeds toward the end of the buffer until a line containing a matching string is found. If no match occurs before the end of the buffer, the search wraps around to the beginning and ends with the current line. The line number of the matching line is the value of the term; an error is signaled if the search fails.

A regular expression enclosed in '%' or '?' causes a backward search towards the beginning of the buffer. The context search starts with the line before the the current line. The search wraps from 1 to $ if necessary and ends with the current line. The line number of the matching line is the value of the term; an error is signaled if the search fails.

A line address is resolved by scanning terms from left to right. Two terms separated by a comma will address the inclusive range of lines specified by the two terms. Two terms separated by a plus '+' (or a minus '-'), are taken to be a term that is the sum (or difference) of the line numbers referenced by the two terms.

If an address begins with a '+' or '-', then dot is assumed to be the term at the left of the '+' or '-'. The symbol '^' is equivalent to '-' in this context. Thus, "+3" standing alone is taken to be the same as ".+3". Also, ".3" is equivalent to ".+3".

A '+' ('-' or '^') not followed by a term is taken to mean +1 (-1). Thus '-' alone stands for ".-1", "++" stands for ".+2", "6---" stands for '3' and so on.

A line address may be arbitrarily complex, so long as its value lies between 0 and $ inclusive. There can be any number of line addresses preceding a command so long as the last one or two are legal for that command. For a command requiring two addresses, it is an error for the second address to refer to a line less than the first. Many commands disallow line 0 as well.

When line addresses are separated by a ':', dot is set to the line address at the left of the ':' before interpreting the rest of the command. Thus,

```
/abc/:///p
```

displays the range of lines between the second and third occurrences after the current line of a line containing "abc".

The descriptions for each command are given below, in the following format:

(line addresses) command <parameters> [display format]

In cases where there are alternate methods for entering a command, both are given. Addresses in parentheses are the default line addresses that will be used for the command if none are entered. The parentheses are not to be entered with the command, but are present as a syntactical notation. If a command is described without parentheses, no addresses are required. For a two address command, supplying only one address will default the second to be identical with the one supplied. It is an error to supply addresses for commands that do not require any. Although multiple commands are entered separated by a semicolon, many commands may be followed directly by l or p as indicated by the notation "[lp]"; the effect is to display the current line, dot, after the command is completed using either the l or p display format. This format is remembered and applied to subsequent (implicit) displays until explicitly changed by the occurrence of another l or p.

(.)a <text> or

```
(.)a
<text>
\&.
```

Append text to the buffer. The lines to be added are placed in the buffer right after the line specified. Dot is set to the number of the last line input. 0 may be used as an address for this command, in which case text is placed at the beginning of the buffer.

(.)c <text> or

```
(.,.)c
<text>
\&.
```

Change the addressed lines. First the addressed lines are deleted from the buffer, then the accepted text is added to the buffer in their place. The accepted text may be a greater or lesser quantity of lines than existed in the deleted range, in which case the buffer is accordingly expanded or contracted. Dot is set to the last line input, if any. If no lines were input, dot is set to the line before the group deleted.

(.,.)d[lp]

Delete the addressed lines from the buffer. Dot is set to the line originally after the last deleted line. If the addressed range was at the end of the buffer, then dot is set to the new $ value.

e <file>
Enter the named file into the buffer by first deleting the current contents of the buffer and then reading the file. A count of the number of characters entered will be displayed. Dot is set to the last line of the buffer. If <file> is omitted, a fresh copy of the file being edited is brought in. If <file> does not exist, the buffer is cleared and the filename given is remembered. A '?' appears instead of a character count in this case, signalling that the editor has no text in its buffer. <file> is remembered and will be used as the default filename for any subsequent r or w command. Before the deletion step, the editor will check if changes have been made to the buffer. If the s flag is not on, and changes have occurred since the last write has been performed, the prompt "are you sure? " will be displayed. If the response begins with a 'Y' or 'y' the e command will be completed. Any other response will abort the enter command. ready to accept more commands.

f <file>
Display the currently remembered filename or optionally set it if <file> is given.

(1,$)g/regular expression/command list
Globally perform "command list" on each line in the range which contains an instance of the regular expression. First, the range of lines specified is examined and all lines which have such an instance are given a secret mark. Then, starting at the first marked line and proceeding towards the end of the file, dot is set to each marked line and the command list executed. For a multi-command list, every command but the last must be terminated by a ';'. g and v commands are not permitted in the command list. If command list is omitted, the p command is assumed.

(.)i <text> or

(.)i
<text>
\&.
Insert text immediately before the addressed line. Dot is set to the last line input; if none were entered, dot is set to the addressed line. This command is different from the a command only in the way it interprets the addressed line with regard to the starting point of text insertion -- insert in front of, append after. 0 may be used as an address for this command, in which case text is placed at the end [sic] of the buffer.

(.)kx[lp]
Know the addressed line by the symbol x, which must be a lowercase letter. The editor remembers this x as a uniquely assigned tag which travels with the line irrespective of the line's future movement around the buffer, even after it has been modified by the s command. The term "'x" will address the line. Only the last tag assigned to a given line is remembered. Dot is set to the addressed line.

(.,.)l[nu]
List the addressed lines, providing visible representation of otherwise invisible characters like tabs, backspaces, and non-graphics. Non-

graphic characters are represented in o, if necessary, and long lines are broken and displayed in pieces. If an n (or u) follows the l, subsequently displayed lines will be numbered (or unnumbered); an l may also be appended to many other commands to display the last line affected.

(.,.)m<line>[lp]

Move the addressed lines from where they currently reside to immediately after the line addressed by <line>. Dot is set to the last of the moved lines. 0 is an acceptable value for <line>, in which case the lines are moved to the beginning of the buffer.

(.)n#[lp][nu]

Display the next page of text starting at the addressed line. A page is a chunk of text of size # lines. After # lines have been displayed, the display halts. If a newline is entered, a further chunk of # lines is displayed. Paging will continue in this manner until $ is reached or a new command is entered. If an n (or u) follows the #, subsequently displayed lines will be numbered (or unnumbered). Dot is set to the last line displayed. If not specified, # defaults to the value specified in the last n command in which n was explicitly stated. At editor invocation # is set to 16.

(.,.)p[nu]

Print the addressed lines to the standard output. If an n (or u) follows the p, subsequently displayed lines will be numbered (or unnumbered). Dot is set to the last line displayed. A p may also be appended to many other commands to display the last line affected.

q

Quit the editor. The buffer is not automatically written. However, if the s flag is not specified and changes to the buffer have been made since the last write of the buffer has occurred, the prompt "are you sure? " will be displayed. Only if the response begins with a 'Y' or 'y' will the quit be effective. Any other response will abort the quit.

(.)r <file>

Read <file> into the buffer after the specified line address. 0 is a valid address for this command, in which case text is read into the beginning of the buffer. If <file> is not specified, the currently remembered file is used, as set by the last e or f command. If <file> is specified, it does not reset the currently remembered file unless no filename is currently remembered. If the read was successful, a count of the number of characters copied will be displayed. Dot is set to the last line copied into the buffer.

(.,.)s/regular expression/replacement/[glp]

Substitute characters within the addressed lines. Each line in the address range is examined and the leftmost occurrence of the regular expression is changed to the replacement; the longest possible match is made for a given starting character. It is considered an error if regular expression has no match in any of the addressed lines, unless s is under control of a g or v command. If the g modifier is at the end of the command, the change is made globally within the line, that is, to all non-overlapping occurrences of regular expression, left to right. An l

or p will display the last line affected. The character that separates the pieces of the command need not be '/'; it can be any character other than newline. Dot is set to the last line modified. The regular expression and the replacement are remembered for future use.

An ampersand '&' in replacement has special meaning. Each instance of '&' in the replacement will be transformed into the character string which regular expression matched. However, the sequence "\&" will cause a literal ampersand to be put in the replacement string.

More generally, any piece of regular expression may be tagged by bounding it with "\(" and "\)". The characters which the piece matches may be made a part of the replacement by including in the replacement the tag name "\(n" where n is a digit in the range of 1 to 9. The value for n is determined by counting from left to right within regular expression the occurrences of "\(". Parenthesized pieces may be nested. As an example of tagging pieces, the line "abc+def" could be changed to "def+abc" with the command

```
s/\(*\)+\(def\)/\(2+\(1/
```

Lines may be split by including in replacement the characters "\n", which is the notation for newline earlier described for regular expressions. Other movement control character notation and octal character notation earlier described is also acceptable in replacement. Joining of lines is done by first deleting newlines and then writing out the file.

A bare s, i.e. one not followed by a regular expression and replacement, is taken to be an s followed by the last regular expression used and last replacement used. A bare sg has the expected behavior; it will execute the s command globally within the line. All forms of the s command may be followed by a p or l to display the last line affected by the command.

(.,.)t<line>[lp]

Copy a twin of the addressed lines immediately after the line addressed by <line>. 0 is an acceptable value for <line> for this command in which case the copy is made at the beginning of the buffer. Dot is set to the last line of the new twin.

(1,$)v/regular expression/command list

Verify which lines do not match an instance of regular expression and perform command list as described in g command. This command is like the g command except matching lines are bypassed and non matching lines are singled out for attention instead.

(1,$)w <file>

Write the addressed lines to the given file. If the file exists, its contents prior to the write will be discarded. If it does not exist, it is created. If <file> is not specified, the remembered file (see e, f, r commands) is used. If file is specified, the remembered filename is not changed unless no filename is currently remembered. If the command completes, the number of characters written is displayed. Dot is not altered by this command.

(.)=[1p]
Print the number equal to the addressed line. Dot is set to the addressed line.

(.+1)
An empty line (newline entered alone) is equivalent to entering ".+1p". This form can be used to view one line at a time. If a line address precedes the newline, the addressed line will be displayed. Dot will be set to the displayed line.

The following commands apply only to Idris/UNIX:

(1,$)> <file>
This command operates in the same manner as the w command except that the addressed lines are written at the end of the named file, extending the named file by the amount of text written. The number of characters written is displayed. If the file does not exist, it is created with general read/write permission. ">>" is also an acceptable form of this command.

!<command>
All characters on the line to the right of the "!" are sent to a shell to be interpreted as a shell command, unless the line reads "!cd <directory>", in which case an attempt is made to change the editor's current directory to <directory>. Occurrences of the sequence "\f" will be replaced by the currently remembered file, if any, before the transfer to system level is made. Dot is not changed by this command.

On any system that can generate keyboard interrupts, an interrupt causes the editor to abort its current command, if any, and display '?'. It then will be ready to accept editor commands again. Dot is generally ill defined at this point.

EXAMPLE

```
% e
```

NAME

echo - copy arguments to STDOUT

SYNOPSIS

echo -[m n] <args>

FUNCTION

echo copies its arguments to STDOUT. They are not interpreted in any way, and may be arbitrary strings. By default, the arguments are output separated by a single space; the last argument is terminated by a newline character. If there are no arguments, nothing is output.

The flags are:

-m output each argument on a separate line.

-n suppress the newline following the last argument.

RETURNS

echo returns success if there are no arguments or if all characters are successfully written.

EXAMPLE

To make a one-line message file:

```
% echo happy new year! >motd
```

NAME

enque - queue up files for stream output

SYNOPSIS

enque -[dir* s] <files>

FUNCTION

enque stores up input for later processing by deque in a first-in first-out manner. The contents of <files> are concatenated and enqueued into a single file in the spool directory. If no <files> are given, STDIN is read. A filename of "-" also causes STDIN to be read, at that point in the list of files.

The flags are:

-dir* take * as the name of the spool directory where queued up input is stored. Default is the current directory.

-s do not complain if an input file is unreadable. Default is to write an error message for each unreadable file.

enque was designed to be invokable by anyone as a general utility. As such, it runs with the privileges of the invoker. On a fairly lax system, it is permissible for enque to be invoked with preset flags via a shell script. On a more secure system, enque will need to be invoked by a program with userid set to the owner of the spool directory, and with all flags pre-specified and terminated by "--" to prevent security gaffes.

RETURNS

enque returns success if all of the input files are readable.

EXAMPLE

To enque a few files for printing:

```
% pr prog.c | enque -dir /adm/spool/lpr
% roff prog.mp | enque -dir /adm/spool/lpr
```

To print them all back on the terminal:

```
% deque -dir /adm/spool/lpr
```

FILES

"." is used as the default spool directory, and /<spooldir>/[0-9]^:[0-9]^ as the name of each file generated by enque. (The first digit string is the internal representation of the current time; the second string is the userid of the invoker of enque.)

SEE ALSO

deque

NAME

entab - convert spaces to tabs

SYNOPSIS

entab -[e# #^] <files>

FUNCTION

entab reads the named files, or STDIN if none are given, and writes them to STDOUT in the order specified, compressing spaces into tabs in the process. A filename of "-" causes STDIN to be read at that point in the list of files.

Flags may be used as follows to set tab stops: If no flags are explicitly given, tabs are set every four columns.

-e# set tab stops every # columns, beginning after the highest numbered stop explicitly set by a -# flag, or after column 1 if none is given. By default, no additional stops are set if a -# flag appears, while -e4 is assumed otherwise.

-#^ set a tab stop in each of the columns #; columns are numbered from 1. Up to 32 stops may be given.

entab properly interprets spaces, backspaces, tabs and newlines. All other non-printing characters are taken as zero length.

RETURNS

entab succeeds if all of its file reads and writes are successful.

EXAMPLE

To set tab stops in columns 10, 18, 26, 34, etc.:

```
% entab -10 -e8 file1
```

SEE ALSO

detab

NAME

error - redirect STDERR

SYNOPSIS

error -[a* o* s] <command>

FUNCTION

error redirects STDERR in various useful ways, then executes <command>.

The flags are:

-a* append STDERR to file.

-o* write STDERR to file.

-s redirect STDERR to STDOUT.

At most one of -a, -o, or -s may be specified. Default is to redirect STDOUT to STDERR.

All error messages related to the file redirection that error performs are written to the STDERR in effect at the invocation of error; all messages related to the execution of <command> are written to the new STDERR.

RETURNS

error will complain and return failure if the flags are incorrect or the file to which STDERR is redirected cannot be written. Otherwise, the status returned is that of command.

EXAMPLE

To record timing information in a file:

```
% error -otimes time {sort file | uniq}
```

NAME

exec - execute a command

SYNOPSIS

exec <command>

FUNCTION

exec invokes <command>, which overlays the current shell. The shell image is forever gone.

exec is intended primarily for use in shell scripts. If exec is used in a shell script and <command> cannot be run, the shell immediately seeks to the end of the script and exits.

If exec is used in a pipeline, it should be used only once on the right-most side of a pipe.

RETURNS

The value returned is that of <command>.

EXAMPLE

To switch to a private shell:

```
% exec myshell -i
```

SEE ALSO

sh

BUGS

Since it is a shell builtin command, exec cannot be used with prefix commands such as time.

NAME

exit - terminate a shell script

SYNOPSIS

exit [retval]

FUNCTION

exit is used to terminate a shell script, usually because an error in processing has occurred. exit is intended for use exclusively in shell scripts.

RETURNS

exit returns failure if [retval] is specified, otherwise it returns success.

EXAMPLE

To terminate a shell script because an error has occurred:

```
test $1 || exit NO
```

SEE ALSO

sh

BUGS

Since it is a shell builtin command, exit cannot be used with prefix commands such as time.

NAME

exp - evaluate binary expression

SYNOPSIS

exp <first_operand> <second_operand> < * | x | X | / | % | + | - >

FUNCTION

exp evaluates the expression with the two operands considered as longs, and writes the result as a long decimal number to STDOUT. The line must have three arguments, as above. The 'x' and 'X' operators are a synonym for the '*' operator.

RETURNS

exp fails if the operator is not one of '*', 'x', 'X', '/', '%', '+', or '-'; if modulo zero, or division by zero is attempted; or if more than three strings appear on the command line.

EXAMPLE

```
exp 0x10 99 \* | set q -- -i

exp $a $b + | set c -- -i

exp 10 20 -
```

SEE ALSO

cp, pr

BUGS

The * operator must be enquoted or escaped with a backslash for glob to ignore it.

NAME

first - print first lines of text files

SYNOPSIS

first -[c## s##] <files>

FUNCTION

first outputs to STDOUT the opening lines of each file specified in <files>, or of STDIN if none are given. A filename of "-" also causes STDIN to be read, at that point in the list of files.

The flags are:

-c## print ## lines from each file. The default is 10.

-s## start output at line ## of each file. The default is 1.

RETURNS

first returns success if something is written from each of its input files.

EXAMPLE

To read 15 lines starting at the third line of two files:

```
% first -c15 -s3 file1 file2
```

SEE ALSO

grep, last

NAME
goto - branch to label in a shell script

SYNOPSIS
goto <label>

FUNCTION
goto is used to transfer execution to a new point in a shell script. The shell will search for <label> starting from the beginning of the shell script to the end of the file. <label> is defined as a line consisting of a colon followed by a string delimited by whitespace.

If <label> is found, processing will continue from the line following the label; otherwise an error message is printed and the shell script exits. If <label> is the target of a goto, any information following <label> is ignored.

goto is intended for use exclusively in shell scripts.

RETURNS
goto returns success if <label> is found, otherwise it returns failure.

EXAMPLE
To start processing in a shell script only if a mandatory first argument is given:

```
test $1 && goto process
error echo '$0: must specify filename' && exit NO
: process
```

SEE ALSO
sh

BUGS
Since it is a shell builtin command, goto cannot be used with prefix commands such as time.

NAME

grep - find occurrence of pattern in files

SYNOPSIS

grep -[b c f l# n o*^ p r# u* v] <pattern> <files>

FUNCTION

grep writes to STDOUT those lines in each input file containing occurrences of the regular expression <pattern>. If no files are given, STDIN is read. A filename of "-" also causes STDIN to be read, at that point in the list of files. The rules for the regular expression matching are exactly the same as those for the editor e. In brief:

Any character in the pattern matches itself.

A character string enclosed in square brackets "[]" matches any of the characters inside the brackets. If the first character inside is '!', then it matches any character but one inside the brackets. A range of characters is indicated by <low>-<high>; for instance, [a-zA-Z] matches any alphabetic.

The character '?' matches any character in the file.

A '^' following a character matches zero or more occurrences of that character.

A '*' matches zero or more characters. It is the same as "?^".

A '^' as the leftmost character of the pattern constrains the match to begin at the start of a line.

A '$' as the rightmost character of the pattern constrains the match to end at the end of a line.

The flags are:

-b write the block number and byte number of the line where the match occurred.

-c count the number of lines matched only. Don't display the lines.

-f do not write the filename where the match occurred.

-l# take page length as # lines. Implies -p.

-n write the line number, in the current file, of the line where the match occurred.

-o* take the string * as an alternate pattern; up to ten of these may be given. An input line then matches if it contains any of the patterns specified.

-p write the page number and line number in the current input file of each pattern match. A page is 56 lines in length.

-r# output # more lines after each match.

-u* match lines in the current input file until string * occurs.

-v output those lines in which a match does not occur.

At most one of -b, -c, -n, or -p may be specified. The -r flag may not be specified with the -b or -c flags.

Both -u and -r may be specified. However, output for the current input file terminates if the string supplied for -u is matched, regardless of the value specified for the -r flag.

If the -u and -c flags are both specified, the count of matched lines includes the line, if any containing a match of the string supplied for the -u flag.

RETURNS

grep returns success if all flags are coherent, all files can be opened, and any match has occurred.

EXAMPLE

The command:

```
% grep -n EOL file.c editor.c
```

might produce the result:

```
file.c:22:  #define EOL 12
file.c:47:      if (c == EOL)
editor.c:32:    if (c != EOL && c != EOF)
editor.c:78:        lex = EOL;
```

while the command:

```
grep "^{" -o"^}" *.c
```

will write all lines either beginning with "{" or beginning with "}" in all c programs.

SEE ALSO

e

NAME

head - add title or footing to text files

SYNOPSIS

head -[b* l# n# o# r t* w# +# #] <files>

FUNCTION

head adds title or footing lines, or both, to each page of the text files specified on its command line, or to STDIN, if none are given. A filename of "-" also causes STDIN to be read, at that point in the list of files.

Titles (or footings) contain three parts, which are output left-justified, centered, and right-justified, respectively, over the width of the input pages. The placement of titles and footings is user-specifiable, as are the length and width of pages. Input pages are counted, and the current page number may be output anywhere in a title by an escape sequence.

The flags are:

-b* obtain the bottom (footing) strings from *, which has the format:

/<left>/ /<right>/

where '/' may be any printable character except '%', and the strings between successive '/' are the left, center, and right components of the footing. Delimiters following the last non-null component need not be given. An occurrence in any component of the sequence "%<n>" causes the current page number to be interpolated into the component at that point on output; <n> may be a 'd' to cause the number to be output in decimal, \&'r' for lowercase Roman, or 'R' for uppercase Roman. A '%' preceding any other character simply causes that character to be output, so that a '%' may be safely output with "%%".

-l# interpret input as pages of # lines apiece. The default page length is 66 lines.

-n# make # the initial page number. The default is 1.

-o# overstrike the title and footing # times when outputting them. By default, no overstriking is done.

-r reverse the left and right components of the title and footing after each page (to ease double-sided printing). The first page is output with the components originally given.

-t* obtain the top (title) strings from *, which has the same format as the argument to -b.

-w# set titles and footings into a line # columns wide. The default page width is 78 columns.

+# put the title line # lines down from the top of the page; +0 puts it on the first line. The default is +3.

-# put the footing line # lines up from the bottom of the page; -0 puts it on the last line. The default is -2.

Title and footing lines are output in place of original lines of input text; the lines they overwrite simply disappear. Naturally, if no -b flag is given, no footing is output; no -t likewise suppresses titles. Title and footing lines are composed dynamically, by positioning their left, right, and center components (in that order) in the output line; overlap between components is ignored. All components are truncated on input, if necessary, to the line width specified by -w.

RETURNS

head returns success if all of its input files are readable.

EXAMPLE

To prettify a table of contents:

```
% head -t"//Table of Contents//" -b";V3.0;%r;8/15/81;" toc
```

would specify a centered title of "Table\ of\ Contents", and a footing consisting of the left-justified string "V3.0", the current page number (centered in lowercase Roman), and the right-justified string "8/15/81".

SEE ALSO

pr

NAME

kill - send signal to process

SYNOPSIS

kill -[hup int quit ins trace range dom float kill bus
seg sys pipe alarm term #] <pids>

FUNCTION

kill sends a signal to the processes whose processids are listed in <pids>. The sender must have the same effective userid as the receiver, or be the superuser. A processid of zero causes the signal to be sent to all processes attached to the sender's terminal.

The signal to be sent is specified as a flag, and is either one of the numbers or one of the mnemonics given below, preceded by a '-'. If no signal is specified, the signal kill (9) is sent by default.

The signals that may be sent are:

Number	Mnemonic	Meaning
1	hup	hangup
2	int	interrupt
3	quit	quit*
4	ins	illegal instruction*
5	trace	trace trap*
6	range	range error*
7	dom	domain error*
8	float	floating point exception*
9	kill	kill
10	bus	bus error*
11	seg	segmentation violation*
12	sys	bad system call*
13	pipe	broken pipe
14	alarm	alarm
15	term	process termination

Those signals marked with an asterisk '*' cause a core dump, if not caught or ignored by the process.

RETURNS

kill returns success if the signal given was successfully sent to all processes specified.

EXAMPLE

To kill an arbitrary background process without logging off:

```
% kill -quit 0
```

NAME

last - print final lines of text files

SYNOPSIS

last -[c##] <files>

FUNCTION

last outputs to STDOUT the closing lines of each file specified in <files>, or of STDIN if none are given. A filename of "-" also causes STDIN to be read, at that point in the list of files.

The flag is:

-c## print the last ## lines in each file. The default is 10.

RETURNS

last returns success if something is written from all of its input files.

EXAMPLE

To see the last 20 lines of two files:

```
% last -c20 file1 file2
```

SEE ALSO

first, grep

BUGS

No error message is generated if the number of lines requested exceeds the numbers of lines contained within the file; the entire contents of the file are printed.

NAME

ln - link file to new name

SYNOPSIS

ln -[patch] <oldname> <newnames>

FUNCTION

ln creates new directory entries for the file with existing name <oldname>; the names are specified by the list <newnames>. If no <newnames> are specified, "." is assumed.

The flag is:

-patch link even to a directory, which is normally disallowed. This flag is used as a companion to "rm -patch" in repairing directory linkages, and will work only for the superuser.

Pathname completion is applied to each newname, i.e. the full newname is formed by appending to it a '/', followed by the longest suffix of <oldname> that does not contain a '/'. If a link cannot be created with the resulting newname, ln attempts to link <oldname> to the original newname given.

RETURNS

ln returns success if all attempts to link succeed.

EXAMPLE

To create the aliases dir/old and newfile for the file old:

```
% ln old dir newfile
```

NAME
lpr - drive line printer

SYNOPSIS
lpr

FUNCTION
lpr camps on the line printer file until it can be opened for writing, then copies STDIN to the line printer with horizontal tabs expanded and backspaces replaced, as needed, with overstruck lines.

It is assumed that the line printer will tolerate only one writer at a time, that tab stops are set every four columns, and that a line will be overstruck if it terminates with a carriage return instead of a newline.

Since lpr does not accept any command line arguments, input is assumed to come from a pipe or a redirected STDIN.

Under some implementations, lpr is a shell script.

RETURNS
lpr returns success if it can open and write to /dev/lp.

EXAMPLE

```
% pr *.mp | lpr
```

FILES
/dev/lp for line printer.

SEE ALSO
deque, enque

BUGS
A more sophisticated spooler might be in order for larger systems.

NAME

ls - list status of files or directory contents

SYNOPSIS

ls -[a b +d d +f f g i l r s t u] <files>

FUNCTION

ls prints information about each of its argument <files>, possibly detailing the contents of any directories along the way. When no <files> are specified, the current directory "." is assumed. For each file, ls outputs its name preceded by any additonal data specified by flags. After displaying the information about each file, for those files which are directories to be expanded, the directory name is repeated followed by each entry and its associated information. The directory entries are sorted in lexical order by name, or by some other attribute selected by the flags. Any sizes given in blocks include the pointer blocks used by its inode.

The flags are:

-a list all entries, even names that begin with '.', which are usually omitted.

-b sort by size of file, largest first.

+d recursively descend all directory files, listing their entries and all optional data requested.

-d treat a directory as an ordinary file; do not list its entries.

+f pretend all files are directories and list their entries. used primarily with restored directories to trace pathnames. -l cannot be specified with this option.

-f recursively descend each directory file, listing it and its subdirectories preceded by the size in blocks of the directory subtree rooted at each. The size of "." and ".." are excluded from each total. If -f is given, all other flags are ignored.

-g list owner name corresponding to groupid, instead of userid.

-i list inode entry of each file.

-l list the mode, number of links, owner, size in bytes, and date of last modification. If the file is a special file then the size field is replaced by the major and minor device numbers.

-r reverse the order of all sorts.

-s list size in blocks for each file.

-t sort by time of last modification.

-u sort by time of last access instead of time of last modification.

Only one flag may be given from each of the sets -[+d\ d], -[+f\ f\ l], -[d\ f], -[b\ t\ u].

The full output line contains, in order: the inode number, size in blocks, mode, number of links, owner, size in bytes, date of last modification, and the filename.

If no flags or <files> are specified, the default is -d.

RETURNS

ls returns success if all the files exist and the user has permission to obtain the desired information about them.

EXAMPLE

To list the current directory, most recently altered files first:

```
% ls -lt
```

To list everything in the known universe:

```
% ls +d -isla /
```

To find the number of blocks occupied by the directory subtree dir and its subdirectories:

```
% ls -f dir
```

BUGS

The -f flag, though extremely handy, makes no attempt to compensate for files other than "." and ".." to which more than one link exists.

NAME

mail - send and receive messages

SYNOPSIS

mail -[a b* d g*^ i q r y] <loginids>

FUNCTION

mail sends and receives messages between system users, each user designated by his loginid or by the loginid of the group to which he belongs.

SENDING MAIL

If you give one or more <loginids>, or give one or more group loginids via -g flags, mail sends a message to the users named. mail collects the message by reading STDIN up to end of file, or up to a line containing only '.'. mail then prepends a header to this message indicating your loginid and the date sent. A line in the input beginning with a '!' is not included in the message, but causes the rest of the line to be immediately executed as a shell command.

Delivery is accomplished by executing a file in the receiver's home directory called ".mail", as if the receiver had run ".mail" himself. The arguments passed to ".mail" are the arguments that you gave to mail. The STDIN of the ".mail" process is the message, including the header. Its STDOUT is the STDOUT of the sender.

If the .mail file cannot be executed, the mail is spooled into the receiver's private mailbox.

The flags for sending mail are:

-a send message to all system users.

-b* interpret the input mail as binary. The header is suffixed by the string * and STDIN is read up to an end of file. Any occurrence of '.' alone on a line is ignored.

-g*^ send message to all members of the groups *. Up to ten such groups may be designated.

-r send back a receipt when the receiver reads the message. The returned receipt header is suffixed with "Receipt" and the content of the receipt is the header from the original message.

RECEIVING MAIL

If no <loginids> are specified, mail displays any pending messages for the current user, or writes "no mail" to STDOUT if none exist.

The flags for receiving mail are:

-d discard combined mail after displaying; don't prompt to save in "mbox".

-i receive mail interactively.

-q quit immediately, returning success if there is mail to be received. One of the messages "you have mail" or "no mail" is written to STDOUT as well.

-r receive mail in reverse of normal order, i.e. oldest first instead of newest first.

-y save combined mail in home directory "mbox" without prompting.

When deilvering mail, in the absence of any flags, mail combines all messages in reverse order of date sent and writes them to STDOUT. The prompt "delete? " is then displayed on STDERR; anything but a 'y' or 'Y' reply causes the mail just displayed to be prepended to the file mbox in the receiver's initial login, or home, directory.

Messages may also be received interactively, in which case each message header is displayed as a prompt. The following responses are recognized:

d discard the message.

p print the message to STDOUT.

q quit receiving mail. Any unreceived mail is returned to the private mailbox.

r return message to private mailbox for later delivery (default).

+ go on to next message.

- go back one message.

> file write the message to file.

>> file append the message to file. The default file is mbox in the receiver's home directory.

^ file prepend message to file. The default file is mbox in the receiver's home directory.

number jump to message numbered number. Characters following the number are ignored.

A blank line is taken to mean "+". An invalid response causes a summary of these commands to be displayed. In all cases, prepending a '-' causes any output to occur without the usual message header plus blank line. Writing a message to a file in any form changes the default disposition from 'r' to 'd'. A response line beginning with a '!' causes the rest of the line to be immediately executed as a shell command.

After all messages have been processed mail prompts with "start over?", anything but a 'y' or 'Y' reply causes mail to exit.

RETURNS

mail returns success if all mail is delivered successfully, or if all received mail is filed successfully, or if -q is specified and there is mail to be delivered.

EXAMPLE

The sequence:

```
% mail -a
The grand opening of the new factory outlet
is on saturday. You're all invited.
.
```

will send a message to everybody. A .mail file for bob in his login directory might contain:

```
who | grep bob >>/dev/null || goto not_logged_on
write bob <<EOF
you have mail. (mail $@)
EOF
: not_logged_on
echo -- $@ | grep bob >>/dev/null && echo Thank you.
exit NO
```

This will tell bob if any new mail arrives while he is logged in. A thank you is sent back to the sender if bob was directly specified at invocation. The script exits with NO to force the missive to be saved until bob explicitly asks for it.

FILES

/adm/mail/* for per user private mailbox directories; /adm/passwd to map userids and groupids to loginids and to determine home directories; /bin/mkdir to add mailbox directories.

SEE ALSO

write

NAME
mc - display file in multiple columns

SYNOPSIS
mc -[c# i# s# w#] <files>

FUNCTION
mc extracts one line at a time from the <file>, and formats these lines so that multiple column output is written to STDOUT. Successive lines occupy columns from left to right. The width and separation of the columns are calculated such that they will not overlap. As many columns as possible are output based on the display width and the longest line in the input. A complete pass is made of the input to determine the longest line, unless the number of columns is specified on the command line.

If no <files> are given, mc takes input from STDIN. A filename of "-" also causes STDIN to be read, at that point in the list of files. If more than one file is specified, they are treated independently of each other.

The flags are:

-c# produce output in # columns, evenly spaced in the width of the page. The default is to calculate the number of columns based on the other values.

-i# set indentation from left-hand margin. The default is 0.

-s# set separation between columns on output. The default is 2.

-w# set the display width. The default is 80.

Specification of the "-c#" flag causes the column separation to be ignored. In this case, all columns will be truncated if necessary.

RETURNS
mc returns success if all files supplied are opened and all flags are valid.

EXAMPLE
Typical use:

```
% ls | mc
address   ask       create    decr      descrip   despool   encr      exists
fcp       fe        mc.mp     md.mp     name      p         panel     pk
rtget     rtput     runoff    sort      spell.mp  spool
```

The command:

```
% ls | mc -s5 -w30
address      ask
create       decr
descrip      despool
encr         exists
fcp          fe
```

```
mc.mp          md.mp
name           p
panel          pk
rtget          rtput
runoff         sort
spell.mp       spool
```

SEE ALSO

pr

NAME

md - maintain a dictionary as needed for spell

SYNOPSIS

md -[a* d* i* o* p] <dict>

FUNCTION

md processes words from a dictionary, adding words from an addition list, deleting those from a deletion list and creating a new dictionary and a special index into this dictionary. Alternatively, md can print the contents of a dictionary to STDOUT.

The flags are:

-a* the file named should contain a sorted (by ASCII collating sequence) list of lower-case words, one per line. These words are merged into the dictionary.

-d* the list of words to be deleted from the dictionary. The named file should be sorted by ASCII collating sequence, one word per line.

-i* the named file is created, and will contain the new dictionary index.

-o* the named file is created, and will contain the new dictionary.

-p the contents of the dictionary are printed to STDOUT.

The "-p" option excludes all other specifications.

<dict> is the name of an existing dictionary. It can be omitted to create a new dictionary from the list of words to be added.

The dictionary is encoded to save space and searching time. Because of this encoding, an index to the dictionary must also be maintained. The default name for the dictionary (only used for "-p") is "/usr/lib/dict", while the index file default is the name of the output dictionary with ".ix" appended.

The dictionary must be kept ordered for spell to work correctly, and for the merge to be efficient. md will check correct ordering of all the input files, and discard any words (with error message) found after a word higher in ASCII sequence. It is not considered an error to add a pre-existing word or delete a non-existent word, so it is possible to maintain files which will turn either a virgin dictionary or a recent one into the current version.

md is unable to create an unsorted dictionary.

RETURNS

md returns success if all files supplied are opened, and all flags are valid.

EXAMPLE

The command:

```
% md -o/lib/dict -awordlist
```

creates a dictionary and corresponding index from the file wordlist, while

```
% mv /lib/dict /lib/olddict
% md -derrors -acorrections -o/lib/dict /lib/olddict
```

can fix subsequent errors.

```
% md -p >dictionary
% grep "a??dv???" dictionary
```

can be useful for solving crosswords.

SEE ALSO

spell, sort

NAME

mesg - turn on or off messages to current terminal

SYNOPSIS

mesg -[n y]

FUNCTION

mesg enables or disables the receipt of messages from other users.

The flags are:

-n disallow messages to the user.

-y allow messages to the user.

If no flags are given, mesg reports the current state of message receipt without changing it.

RETURNS

mesg always returns success.

EXAMPLE

To turn off messages while listing a file:

```
% mesg -n
% cat file
% mesg -y
```

SEE ALSO

write

NAME

mkdir - make diectories

SYNOPSIS

mkdir -[r s] <files>

FUNCTION

mkdir makes one or more directories with names specified by the list <files>. The directory entry ".", which links the directory to itself, and "..", which links the directory to its parent, are made automatically. The directory will only be made if the user has write permission in the parent directory and if no file of that name currently exists.

The flags are:

-r recursively create directories if they do not exist.

-s operate silently. Don't write error message to STDERR if a directory cannot be made.

RETURNS

mkdir returns success if the directory, and its "." and ".." entries are created.

EXAMPLE

To make the directories dir, dir/src, and dir/doc:

```
% mkdir dir dir/src dir/doc
```

To make the directories new, new/src, and new/src/headers:

```
% mkdir -r new/src/headers
```

SEE ALSO

rm

NAME
mv - move files

SYNOPSIS
mv -[s] <files> <dest>

FUNCTION
mv moves one or more source <files> to <dest>. The move is a simple renaming where possible; a move between filesystems causes a copy and a deletion of the original file. In all cases, an existing destination file is first removed (providing it is not a directory), even if it has no write permission.

Pathname completion is used to determine the destination file, i.e., for each source file a destination name is formed by appending to dest a '/' followed by the longest suffix of the source filename that does not contain a '/'. If there is only one source file, and if the completed destination pathname is unacceptable, then mv performs a simple move of source to dest.

If more than one source file is specified, a header line giving the filename, followed by a colon, is written to STDOUT before each move.

The flag is:

-s suppress writing header line.

Directories may be moved within a filesystem, provided the invoking user has write and scan permission for the destination directory. The link ".." is altered to point at its new parent.

RETURNS
mv returns success if all links, unlinks, opens, creates, reads, and writes succeed.

EXAMPLE
To move all files whose names end in ".c" to directory dir:

```
% mv *.c dir
```

NAME

nice - execute a command with altered priority

SYNOPSIS

nice -[+# #] <command>

FUNCTION

nice is a prefix command that changes the priority bias of the current process (and consequently all of its children), then executes <command>.

The flags are:

+# alter bias (to +#) to increase the priority given to the current process. Only the superuser may request a positive value.

-# alter bias (to -#) to reduce the priority given to the current process.

The superuser may specify any value in the range (-128, +128] [sic]; all other users are restricted to values in the range (-128, 0]. If no priority is specified, a value of -10 is used. To specify a priority of zero, use either +0 or -0.

Note that the priority as referred to here is the negation of the internal bias vaule used by the Idris resident, and that the bias value is only loosely related to the actual priority of the process. The process priority has three distinct classes, as follows: A value in the range @[-128, -20] (such as nice +20) locks the process in core (on implementations of Idris where such an act is safe), and blocks all signals that are sent to the process. A value in the range (-20, 0) allows the process to be swapped, but still blocks all signals. And a value in the range [0, 128) (such as nice +10) allows signals to be received, and permits the process to be swapped. To allow the greatest flexibility, normal processes should be run with a bias of 5 (nice -5) or more.

RETURNS

nice returns failure if <command> cannot be executed, otherwise it returns the status of <command>.

EXAMPLE

To use up idle computer time with minimum disruption of normal service:

```
% nice -20 makework > output&
```

To do something important in a hurry:

```
# nice +20 urgentcmd
```

SEE ALSO

nohup

NAME
nohup - run a command immune to termination signals

SYNOPSIS
nohup -[hup int quit] <command>

FUNCTION
nohup turns off processing of hangup, interrupt, and quit signals, then executes <command>. If any flags are specified, then only the coresponding signals are trapped. Note that if <command> elects to handle any of these signals, that overrides the conditions set up by nohup. nohup is normally invoked for background processing that is to continue beyond the end of a terminal session.

The interrupt and quit signals can be generated by the DEL and ctl-\ keys respectively on the controlling terminal. Hangup is generated by turning off the terminal, or hanging up a dial-up line. All three signals, as well as several others, may be generated by the kill command.

The flags are:

-hup ignore hangups (signal 1).

-int ignore interrupts (signal 2).

-quit ignore quits (signal 3).

RETURNS
nohup returns failure if the command cannot be executed; or the exit status of <command>.

EXAMPLE
To generate a long listing:

```
% nohup error -o errors nroff *.mp | lpr &
```

SEE ALSO
kill

NAME

od - dump a file in desired format to STDOUT

SYNOPSIS

od -[a +b# b# c h i l o s t# u v +## ##] <files>

FUNCTION

od dumps one or more files to STDOUT. Each file is opened for binary input and is read as a sequence of characters, shorts, integers, or longs. Output is interpreted as either ASCII text, hexadecimal, octal, unsigned, or signed decimal. Each output line contains several interpreted elements, preceded by the address of the first element.

In terse mode, for two or more identical lines, only the first line is displayed followed by the address of the last matching line.

"-" in the list of files is taken as STDIN. If there are no file arguments, input is taken from STDIN. For multiple files, the output for each file is preceeded by a line giving the filename, followed by a colon.

The flags are:

-a output as ASCII characters. Bytes with internal values in the range [0, 7] are represented as "\0" - "\7"; backspace is represented as "\b"; horizontal tab as "\t"; newline as "\n"; vertical tab as "\v"; formfeed as "\f"; carriage return as "\r"; all other unprintable characters are represented as "\?".

+b# compute starting offset using # as block number. Default is 0.

-b# compute terminating offset using # as block number. Default is 0.

-c treat input as 1 byte data.

-h output as hexadecimal integers.

-i treat input as 2 or 4 byte data, depending upon host machine word size.

-l treat input as 4 byte data.

-o output as octal integers.

-s treat input as 2 byte data.

-t# output # elements per line. -t0 is a special case which prints one element per line with no offsets; -v is assumed. The default for -t# varies with the output format selected.

-u output as unsigned integers.

-v output all lines (verbose). The default is terse.

+## compute starting offset using ## as byte number. Default is 0.

-## compute terminating offset using ## as byte number.

At most one of -c, -s, -i, and -l may be specified; the default is -i. At most one of -a, -h, -o, and -u may be specified; the default is signed decimal. Integers are interpreted in native byte order for the host machine. Offsets are represented in hexadecimal when -h is specified, in octal when -o is specified. Otherwise, the offset is unsigned decimal.

Output starts at the beginning of the file and stops at the end of the file unless one or more of the flags: +b#, +##, -b#, -## are specified. Since blocks contain 512 bytes each, the offsets are computed as follows:

starting offset = 512 * starting block # + starting byte ##

terminating offset = 512 * terminating block # + terminating byte ##

RETURNS

od returns success if all flags are coherent and all files can be opened.

EXAMPLE

```
% od -ac file
00000000000    l  i  n  e     1 \n  l  i  n  e     2 \n \n  l  i  n  e
00000000020    4 \n  l  i  n  e     5 \n  l  i  n  e     6 \n  l  i  n  e
00000000040       7 \n  l  i  n  e     8 \n  l  i  n  e     9 \n


% od -hl file
00000000   696c656e 31206c0a 6e692065 0a326c0a
00000010   6e692065 0a34696c 656e3520 6c0a6e69
00000020   20650a36 696c656e 37206c0a 6e692065
00000030   0a38696c 656e3920 000a0000
```

NAME

page - display files n lines at a time

SYNOPSIS

page -[c# p*] <files>

FUNCTION

page takes input from named files and outputs them to STDOUT a specified number of lines at a time. Each set of lines is followed by a prompt (whose value can be set by the user at invocation). page pauses after the prompt for a line of input. If the input is the letter 'y' or 'Y' followed by a carriage return, page will proceed to print out the next set of lines. If the line begins with anything else, page goes on to the next file on the argument list. If the line begins with a '!', the remainder of the line is interpreted as a shell command, just as in the editor, e. This allows someone using a high-speed terminal to page through a file.

The flags are:

-c# sets # as the number of lines displayed before prompting. The default is 23.

-p* use * as the prompt string. The default is "more? ".

RETURNS

page returns success if all opens were successful, and responses to the prompt can be obtained.

NAME

passwd - change login password

SYNOPSIS

passwd -[o* n*] <loginid>

FUNCTION

passwd permits the login password corresponding to <loginid> to be changed. If <loginid> is absent, that of the current user is assumed. In the absence of flags, passwd prompts for the old password, and if it is entered correctly, prompts for the new password twice (to ensure that it is not mistyped). If the new password is typed the same way both times, passwd changes the appropriate entry in /adm/passwd to the new password. Echoing is turned off for STDIN, if possible, before asking for any password.

The flags are:

-o* use * instead of prompting for old password.

-n* use * instead of prompting for new password.

The superuser may change any password, and need know the old password only when changing the superuser password.

RETURNS

passwd returns success if all questions are answered correctly and the password is changed.

EXAMPLE

```
% passwd
old password:
new password:
retype new password:
```

FILES

/adm/passwd for the password file.

NAME

pk - file packing utility

SYNOPSIS

pk -[f* o r s t u v +v] <files>

FUNCTION

This program compresses or restores files, most often to save disk space or reduce transmission time, although compression is an effective way to remove redundancy as a preliminary to encryption. The algorithm used is Huffman Coding optionally combined with the merging of sequences of repeated characters.

The usual mode of operation is to pack or unpack the input files named in <files>, each to a separate output file. If no <files> are given, STDIN is read. A filename of "-" also causes STDIN to be read, at that point in the list of files. By default, input files are deleted after processing. When packing files, each output filename is generated by suffixing an input filename with ".k". When unpacking files, the output filename is generated by dropping a trailing ".k", if any, from the input filename. If the input filename doesn't end in ".k", its last character is dropped to derive an output name, with a warning message to that effect.

Flags are:

-f* read only a single input <file> or STDIN, and write output to file *.

-o read only a single input <file> or STDIN, and write output to STDOUT. This is the default if no <files> are named.

-r suppress the removal of the original file. Removal is always suppressed when -f or -o is specified.

-s allow the squashing of sequences of ascii characters. This flag has meaning only when packing. When unpacking, each packed file specifies whether or not the original input file was "squashed".

This is done by making any sequence of three or more identical characters into the repeated character followed by a one-byte repeat count with the parity bit set. These count bytes are repeated as needed.

In unpacking a file originally packed with -s, any unpacked byte with its parity bit set will be assumed to be a repeat count, so a binary or EBCDIC file will be corrupted.

-u unpack the file(s) instead of packing. This is the same as invoking the program with a name starting with 'u', e.g., unpk.

-v be verbose. The file names will be mentioned in passing, and some statistics on the packing process will come out. If -o or -f is specified, the statistics are written to STDERR, otherwise to STDOUT.

+v be very verbose. As well as the verbose statistics, generate tables of the individual bit patterns, which are of interest to huffmanomaniacs only, and a few more size statistics. This output is directed like that from -v.

There are interrelations between flags: +v implies -v, -u supresses the setting of -s, which is taken from the files themselves, and both -o and -f imply that at most one file may be specified, as well as implying the -r flag.

The format of a packed file is: a short integer identifying the file as packed; a long integer containing the number of data bits in the packed file; a char-sized integer whose value is non-zero if -s was used when packing the file; an encoded table of the bit strings (10n+9 bits, where n is the number of distinct bytes in the input file); the encoded data bits (see verbose output); and enough bits of packing to make a whole byte.

In case you haven't noticed, pk is usable as a filter when no files are specified.

It is worth noting that Huffman Coding is really quite effective, and the -s flag is most useful in extreme cases (not normally useful for C programs or text). For ordinary text, the compressed file is typically well over 40 percent smaller than the original; the -s flag improves this by only a few percent.

RETURNS

pk will return success unless a fatal error (e.g., I/O error or failure to create an output file) occurs. A missing input file is not considered fatal, nor is an attempt to unpack a file which wasn't packed, in which case the original isn't removed.

EXAMPLE

To save a lot of space in a huge dictionary:

```
% pk *
```

To then recover it:

```
% upk *.k
```

To generate huge amounts of output for later processing:

```
% generate lotsofoutput | pk -f lots.k
```

To subsequently process it:

```
% upk -o lots.k | processit
```

BUGS

The '-s' flag assumes (without checking) that the file contains only valid ascii characters, and if pointed at a non-ascii file will probably corrupt it. Packed files are not portable across machines with different byte

sizes.

There may be a fictitious character entered into the table to force non-trivial bit strings.

NAME

pr - print files in pages

SYNOPSIS

pr -[c# e# h l# m n p s? t* w# +## ##] <files>

FUNCTION

pr prints to STDOUT the files in the list <files>, adding a title and empty lines for page breaks, and padding to an integral number of pages. If no <files> are given, pr takes input from STDIN. A filename of "-" also causes STDIN to be read, at that point in the list of files. Each page output has a 5 line heading containing a title line between pairs of empty lines, and a 5 line footing.

The standard title consists of the last modification date of the file being output, its name, and the current page number. When a user-specified title is given, or when STDIN is the file being output, the current date and time are used instead of a modification date. Since pr is used most often for hardcopy, output is entabbed for speed.

The flags are:

-c# print each file in # columns. The default is 1.

-e# tabs are spaced at intervals of #. The default is 4. -e1 suppresses tabbing.

-h suppress headings and footings on output.

-l# output pages # lines in length. The default is 66.

-m print all files simultaneously, each in a separate column.

-n number the output lines.

-p print one page of output and pause, waiting for input from STDIN.

-s? use a single ? to separate multiple columns of output, instead of whitespace. When this option is specifed, entabbing is suppressed.

-t* use * as the filename field of the heading title.

-w# specifies a page width of # positions. The default is 72.

+## start output of each file with page ##. The default is 1.

-## stop output of each file after page ##. The default is huge.

At most one of -c# and -m may be specified.

RETURNS

pr returns success if no error messages are written to STDERR, i.e., if all flags are coherent and all files can be opened.

EXAMPLE

```
% pr -m -l21 file1 file2 file3

Mon Mar 31 14:49:36 1980  Page 1

file 1 line 1          file 2 line 1          file 3 line 1
file 1 line 2          file 2 line 2          file 3 line 2
file 1 line 3          file 2 line 3          file 3 line 3
file 1 line 4          file 2 line 4          file 3 line 4
file 1 last            file 2 line 5          file 3 line 5
                       file 2 line 6          file 3 line 6
                       file 2 line 7          file 3 line 7
                       file 2 line 8          file 3 line 8
                       file 2 line 9          file 3 line 9
                       file 2 last            file 3 line10
                                              file 3 last
```

```
% pr -c2 -h -l4 file3
file 3 line 1          file 3 line 5
file 3 line 2          file 3 line 6
file 3 line 3          file 3 line 7
file 3 line 4          file 3 line 8
file 3 line 9
file 3 line10
file 3 last
```

BUGS

When printing multicolumn output, the input must first be piped through detab to maintain proper columns on output.

NAME

pwd - print current directory pathname

SYNOPSIS

pwd

FUNCTION

pwd finds the absolute pathname of the current working directory and prints it on STDOUT, followed by a newline.

RETURNS

pwd returns success if the pathname is found and is not too long.

EXAMPLE

```
% pwd
/usr/pjp/poem
```

FILES

/adm/mount for the mounted filesystems.

NAME

rm - remove files

SYNOPSIS

rm -[d i patch r s] <files>

FUNCTION

rm removes one or more files. A file can only be removed if the user has write permission in the parent directory. If the user does not have write permission for the file itself, rm prompts the user with the file's attributes, in ls form, and attempts to remove the file only if the user response begins with 'y' or 'Y'.

Directories may be removed if -d or -patch is specified.

The flags are:

-d empty directories may be removed.

-i interactive. Before an attempt is made to remove a file, the user is prompted with the file's attributes, in ls form. If the user response begins with anything other than a 'y' or 'Y', rm makes no attempt to remove the file.

-patch remove even a non-empty directory. Used to repair damage to directory linkages. This flag will only work for the superuser and should be used with extreme caution, since it can damage a filesystem.

-r recursively descend each directory, removing each entry. The emptied directory will also be removed if -d is specified.

-s run silently. No error messages are written to STDERR or STDOUT.

RETURNS

rm returns success if all operations succeed.

EXAMPLE

To remove files interactively from dir:

```
% rm -ri dir
drwxrwxrwx  2 scp          80 May 10 10:31 dir y
-rw-rw-rw-  1 scp           2 May 14 11:38 dir/entry1 y
-rw-rw-rw-  1 scp           2 May 14 11:38 dir/entry2 y
-rw-rw-rw-  1 scp           2 May 14 11:38 dir/entry3 y
```

SEE ALSO

ls, mv

NAME

roff - format text

SYNOPSIS

roff -[l# w# +# #] <files>

FUNCTION

roff reads the named files, or STDIN if no files are given, and writes formatted text to STDOUT with pagination, filling, and justification of the right margin. It is a simple formatter that is very much tailored to supporting documents such as business letters, manuscripts, and manual pages such as this one.

Interspersed with the text to be formatted, roff accepts commands that influence the way the formatting is done. All lines that start with a dot '.' are treated as commands; unknown commands are ignored. Only the first two letters (after the dot) of a command are significant, and, with the exception of ".ds" and ".DS", case is ignored. Certain of the commands cause a line break, that is, no more filling occurs for the current line, and subsequent text is started on a new line.

Any command that doesn't take an argument will ignore any arguments that are given. Also, commands that take a numeric argument will accept an optional sign that causes the value in question to be incremented or decremented by the given amount. The commands are:

```
NAME MEANING BREAK DEFAULT
.1F page 1 footer no none
.1H page 1 header no none
.BP begin page yes 1
.BR break yes
.BU BUGS entry yes
.CE center lines no 1
.CO COURSE OUTLINE entry yes
.DE display end yes
.DS display start yes
.EF even footer no none
.EG EXAMPLE entry yes
.EH even header no none
.EN equation end yes
.EQ equation start yes
.EX extract start yes
.FI fill yes
.FL FILES entry yes
.FU FUNCTION entry yes
.HD heading yes none
.IN indent yes 0
.IP indented paragraph yes 5
.IT underline lines no 1
.LH skip letterhead yes
.LL line length no 78
.LP block paragraph yes
.LS line spacing no 1
.M1 margin 1 no 2
```

```
.M2 margin 2 no 2
.M3 margin 3 no 2
.M4 margin 4 no 2
.NA no margin alignment yes
.NE need n lines maybe 0
.NF no fill yes
.NM NAME entry yes
.OF odd footer no none
.OH odd header no none
.PL page length no 66
.PP paragraph yes
.PR PREREQUESITES entry yes
.$$ PRICE entry yes
.RM same as .LL
.RS reset yes
.RT RETURNS entry yes
.SA SEE ALSO entry yes
.SO insert source no none
.SP space n lines yes 1
.SY SYNOPSIS entry yes
.TE table end yes
.TI temporary indent yes +5
.TS table start yes
.UL underline no 1
.XE extract end yes
```

Note that the title commands (.1F, .EF, .OF, .1H, .EH, .OH) use the second argument as the title string, with the format "/left/center/right/". This is to maintain compatibility with existing documents, and future formatters. Also note that to ease conversion to the new formatter when it is available, it is recomended that only uppercase commands be used.

M1 controls the number of lines before the header, and includes the header. M2 controls the number of lines after the header. M3 controls the number of lines before the footer. M4 controls the number of lines after the footer, and includes the footer.

The flags are:

-l# set initial page length to #.

-w# set initial page width to #.

+# start output with page #.

-# stop output after page #.

If roff is given more than one file to format, it will treat each file as a continuation of the last, with the provision that each file starts on a new page. If the margins leave too little space for at least one word, then one word will be put out anyway. Leading whitespace causes a break, and the left margin is moved in accordingly. If the whitespace is a tab, the margin is moved to the next tab stop; tabs are every four columns, with column zero at the normal left margin.

RETURNS

roff returns success if it can read all of its files.

EXAMPLE

The following roff source file:

```
.PL 15
.IN 10
.RM 55
.1H "" "this is a header"
.1F "" "-- % --"
.CE
centered text
Page one:
.UL
This text will be underlined
and this will not.
.TI
This is a temporary indent.
This
.BR
is how a break looks.
.NF
No-fill text will not be truncated if the line is too long.
.FI
The footer will have the page number in it.
```

will produce the following output:

this is a header

centered text

Page one: This text will be underlined and this will not.

This is a temporary indent. This is how a break looks.

No-fill text will not be truncated if the line is too long. The footer will have the page number in it.

-- 1 --

BUGS

Much is left to the imagination, and experimentation. M[1234] must be set before PL.

NAME

set - assign a value to a shell variable

SYNOPSIS

set -[i p* s*] <v> <string>

FUNCTION

set assigns the value <string> to the shell variable <v>, which is a character from the set [0-9a-zA-Z$]. If no arguments are specified, set displays the values of all non-null shell variables. If <string> is not specified, <v> becomes null.

The flags are:

-i read one line from STDIN to obtain <string>.

-p* replace <string> by a prefix of itself extending up to but not including the rightmost occurrence of the substring *. If <string> contains no occurrence of * it is left unchanged.

-s* replace <string> by the longest suffix of itself not containing the substring *. If <string> contains no occurrence of * it becomes null.

Both -p and -s may be specified, in which case the prefix is taken first, then the suffix of the prefix.

RETURNS

set returns success if its flags are valid, and if <v> is a valid shell variable.

If set returns failure within a shell script, the shell immediately seeks to the end of the script and exits.

EXAMPLE

To change your shell prompt:

```
% set P "yeah? "
yeah?
```

SEE ALSO

sh

BUGS

Since it is a shell builtin command, set cannot be used with prefix commands such as time.

NAME
setb - change or display stack/heap size

SYNOPSIS
setb -[b## h o u] <files>

FUNCTION
setb changes or displays the stack/heap size to be allocated at exec time for the files in the list <files>, each of which must be an object file in standard format or in UNIX/V6 (a.out) format. If no files are specified, or if the filename "-" is used, the file xeq is assumed. Each file is updated in place.

The flags are:

-b## set the stack plus heap size for each file to ## bytes.

-h display stack/heap size in hexadecimal.

-o display stack/heap size in octal.

-u display stack/heap size in unsigned decimal.

At most one of -[h o u] may be specified, default is unsigned decimal. If no -b flag is given, setb will write to STDOUT the current stack/heap size for each of its argument files, in unsigned decimal, unless otherwise specified. If more than one file was specified, the name of each file, followed by a colon and a tab, will precede the associated size.

setb will complain to STDERR if unable to process an argument file, but will still try to process the remainder of its command line.

RETURNS
setb returns success if it can interpret all of its argument files as standard object files, and if all necessary processing succeeds.

EXAMPLE
To set the stack size of xeq to hex 0x4000:

```
setb -b0x4000
```

NAME

sh - execute programs

SYNOPSIS

sh -[H* P* S* X* c e i] <args>

FUNCTION

sh reads lines of input and interprets them as commands to be executed, with arguments passed to each command. Its command language is useful either for direct input to the shell, as from a terminal, or for creating files of commands, called shell scripts, for later execution. To permit varied usages, sh has three operating modes:

interactive: the shell reads single lines, either from the files given as command line <args> or from a terminal, and will not exit when sent interrupt or quit signals. These signals typically will cause a command being executed by the shell to terminate.

script: the shell reads a file and executes the commands specified, and will exit if sent an interrupt or quit signal.

command argument: the shell takes its command input directly from <args>.

The flags are:

-H* make directory * the default, or home, directory name when the builtin command cd is issued with no arguments. An absolute pathname should be given. The home directory can be changed by using the builtin command set to modify shell variable H.

-P* set the prompt to * instead of the default, which is "# " for the superuser and "% " for all others. The prompt string can be changed by using the builtin command set to modify shell variable P.

-S* use * as the name of the subshell to invoke for all children. The default is "/bin/sh".

-X* set the "execution path" of the new process to include the directories in *. If a command name typed to the shell contains no '/', it is prefixed with each directory path given, in the order given, before the shell tries to execute it. Directories in * are separated by a vertical bar ('|'); a null name specifies no prefix, hence the current working directory. The default execution path is "|/bin/", which can be changed by using the builtin command set to modify shell variable X.

-c take command input from <args>. Each argument is interpreted as a separate command line.

-e echo each command line to STDERR before execution. This is mainly intended for debugging shell scripts.

-i read commands interactively. Any <args> given are interpreted as command input files. If a "-" is encountered as a filename, or if no

files are specified, the shell reads from STDIN and prompts to STDOUT. Input files other than STDIN are treated as shell scripts. In all cases, quit and interrupt signals are disabled within the shell, but commands executed by the shell are affected by those signals.

The flags -c and -i are mutually exclusive. If no flags are specified, and the shell is invoked by the name {, then -c is assumed. (This is how bracketed command groups get executed.) When the shell is invoked as sh and no flags are given, script mode is assumed. In script mode, the first argument is taken as the filename containing the list of commands; this and subsequent arguments are used to initialize shell variables, which are described later. If no arguments are given in script mode, script input is read from STDIN, and the shell variables mentioned are initialized to null.

Syntax. The shell reads input one line at a time, replacing shell variables by their current values before parsing the line into operand strings plus operators and separators. The simplest command line is a set of strings, not including operators, separated by whitespace. The first string on the line is taken as a command name; subsequent strings are used as arguments to the command.

Commands may be combined into arbitrarily complex expressions with the following operators, listed in order of decreasing binding strength:

| pipeline. A pipeline is a data transfer mechanism under which two commands run concurrently, with the STDOUT of the command on the left side of the pipe becoming the STDIN of the command on the right side of the pipe.

&& logical and (then if). The command immediately to the right of "&&" will execute if and only if the command to the left has executed and returned success. The value of {cmd1 && cmd2} is the value of the last command executed.

|| logical or (else if). The command immediately to the right of "||" will execute if and only if the left command fails. The value of {cmd1 || cmd2} is the value of the last command executed.

& and ; separators. Both "&" and ";" are used to separate one command from another. "&" specifies that the left command be executed in the background (the shell doesn't wait for it to complete), unaffected by interrupts and quits. The processid of the left command is printed to STDOUT so that a kill can be issued to terminate it, if necessary. The right command is executed as soon as the left side has started. ";" causes sequential processing of two commands. The right command waits until the left command has terminated before executing.

Command grouping. Commands may be grouped together within balanced pairs of braces, "{}"; a subshell is spawned to execute the enclosed commands. Thus, a pipeline could be timed as follows:

```
time {sort file?? | uniq > list}
```

What actually happens is that the expression in braces becomes two arguments, a left brace '{' alone, followed by the rest of the expression. Since { is an alias for sh, and because sh always operates in command mode when invoked by this alias, the "parenthesization" implied works as one would expect. However, because a new shell is always created to execute the enclosed commands, any commands that affect the current shell (such as exit or exec) operate on the subshell, not on the topmost one.

Metacharacters. An argument string containing one or more of the metacharacters "^*?[" is treated as a pathname pattern, with the characters to the right of the rightmost slash (if any) taken as a pattern to be matched against a set of files, and the balance of the string taken as the directory in which the pattern match is to be done. If the argument contains no slashes, then the pattern is matched against the files in the current directory. If the pattern completely matches one or more filenames, it is replaced by the sorted sequence of matching filenames. Note that the pattern matching mechanism is the same one used by grep and the editor e.

Redirection of STDIN and STDOUT. The shell reads from STDIN, file descriptor 0, and prompts to STDOUT, file descriptor 1. A command executed by the shell uses the same STDIN and STDOUT as the shell itself unless otherwise indicated; both of these files are normally connected to the terminal, but may be "redirected" to refer to other devices. Each command is also given an error output STDERR which the shell will never redirect; typically STDERR writes to the terminal (but see the error command).

STDIN may be redirected in one of two ways:

< file - input by the command from STDIN will be read from file.

<< string - the current source of shell command input (i.e., the current command line <arg> or the terminal in interactive mode, or the current script in shell script mode) will be read line by line until string is encountered on a line by itself. All lines read up to the terminating line will be written to a temporary file that will be taken as STDIN for the current command. Shell variables occurring in the lines read are replaced by their current values. Any sequence "$x" in which x is not a valid shell variable is left untouched. The sequence "\$" causes a '$' to be taken literally, while '\' itself has no special meaning before any other character.

If a command is executed in the background (as the left operand of "&") and its STDIN is not explicitly redirected by "<" or "<<", then its STDIN is implicitly redirected to /dev/null.

STDOUT may be redirected in one of two ways:

> file - output by the command to STDOUT will be written to file. If file exists, it will be truncated to zero length; if it does not exist, it will be created.

>> file - output by the command to STDOUT will be appended to the end of file. If file does not exist, it will be created.

The redirection symbols ("<", "<<", ">", ">>") may appear before or after a command, or mixed in with its arguments. The string immediately following the symbol will be taken as its associated file or terminator string.

It is often useful to submit a single command to background with an "&", (e.g., "c prog.c &"); in which case the right side of the binary operator is taken as a null command. A null command may be used with any operator, and always returns success.

Shell variables. The shell has 63 variables, which may be referenced and set by the user. The identifiers are one character in length and come from the set [0-9a-zA-Z$]. A variable is referenced by prefixing its identifier with '$'. Thus when "$a" is typed on a command line it is replaced with the contents of the variable a. The effect is as if the contents had been directly read from the current source of command input, so that if the value of a shell variable contains special characters (including references to other variables), it will be fully interpreted when inserted into the command line. (Single-quoting the reference suppresses this interpretation.)

When the shell is invoked in interactive or command argument mode, the variables are initialized as follows:

0-9	null
H	a home directory as supplied by -H.
P	a prompt string as supplied by -P.
S	a subshell string as supplied by -S.
X	an execution path as supplied by -X.
$	the current processid as a 5 digit decimal number.

All single-letter variables other then H, P, S and X are initialized to null.

When the shell is invoked in script mode, 0 is initialized to the name of the script, and 1 to 9 are initialized to the first 9 arguments of the script. All other variables are initialized as previously described. 0 to 9 may be reset to subsequent arguments, if any, by using the builtin command shift.

In script mode, two special variables @ and * are also defined. The value of @ is the series of arguments with which the script was invoked, from the current $1 onward, each argument passed as a separate string. The value of * is also this series of arguments, but quoted so as to be passed as a single string (consisting of the argument strings separated by spaces). Note that in both cases each argument is passed literally; the argument strings themselves are not expanded as normal command input is. Outside script mode, both @ and * have the null string as value. In the lines comprising input for "<<", the variable * is not recognized; the string "$@" must be used for the desired effect.

The variables @ and * cannot be assigned new values. All other variables, including H, P, S, and X, can be given new values with the builtin command set.

Quoting and escape sequences. Metacharacters, operators, all special symbols, and all whitespace characters lose their special meaning when enclosed in single or double quotes. In addition, shell variables (occurrences of the two-character sequence "$x") are not expanded within a double-quoted string, but are expanded when inside single quotes. Outside quoted strings, the character '\' followed by any other character causes the latter to be taken literally, and stripped of any special meaning. An exception to this is '\' followed by a newline; this sequence causes line continuation, as the newline is simply discarded.

Note that if the shell is invoked by the name {, the command line will be scanned twice. This means that quoted strings, escape sequences and shell variables will be parsed twice.

Shell scripts. In addition to operating in shell script mode as described above, sh will also do so implicitly: if a file exists with the same name as the command being invoked, has execute permission, but is not an executable binary file, it is assumed to be a shell script. To execute it, a shell is invoked with the command name as its first argument, followed by the arguments to the command. The use of scripts can therefore be made identical to the invocation of binary programs. Further, since in this case STDIN is not the source of command input, a script can be used as a filter just as an ordinary program would.

Builtin commands. For a variety of reasons, a few commands are builtin, that is, executed by the shell itself without the creation of a new process. These builtin commands (each described on a separate manual page) are:

cd - change current directory.

exec - execute a command and terminate.

exit - terminate a shell script.

goto - branch to label in shell script.

set - change value of a shell variable, or show all variables.

shift - percolate the shell variables @, *, and 0 through 9 to the left.

wait - wait for all children to complete.

A line consisting of a colon followed by a string delimited by whitespace, defines the string as a label that may serve as the target of a goto in a shell script. If this line is the target of a goto, any information to the right of the label is ignored, otherwise the information to the right of the label is interpreted and executed by the shell as described above.

RETURNS

sh returns failure immediately if it is given invalid flags, or if, in shell script mode, it cannot open its first <arg> as a shell script. sh also returns failure if the last command it executed returned failure. In interactive mode, sh complains but does not fail if unable to open one or more of its <args> as shell scripts.

EXAMPLE

Don't be silly.

FILES

/bin/{ for bracketed groups, /bin/sh for shell scripts, /dev/null for background STDIN, /odd/glob for filename pattern matching.

SEE ALSO

cd, e, error, exec, exit, glob(IV), goto, null(III), set, shift, test, wait

NAME

shift - reassign shell script arguments to shell variables

SYNOPSIS

shift <n>

FUNCTION

shift assigns the values of the numeric shell variables <n>+2 through 9 to variables <n>+1 through 8. The variable 9 receives the leftmost argument to the shell script that has not previously been assigned to a shell variable. If there are no arguments left, the variable 9 becomes null. <n> can be any of the decimal digits 0 through 9; 0 is the default. shift operates on the shell variables @ and * in the same way as described above. shift is line oriented, and the shift does not take affect until the next line of a shell script.

RETURNS

shift returns success if the new value of the argument <n>+1 is not null.

EXAMPLE

To keep the first two arguments, and shift the rest:

```
shift 2
```

SEE ALSO

sh

BUGS

Since it is a builtin shell command, shift cannot be used with prefix commands such as time.

NAME

sleep - delay for a while

SYNOPSIS

sleep -[h# m# s#] <secs>

FUNCTION

sleep suspends execution for the amount of time specified by the command line arguments and flags. Time delays may be entered by specifying a number of seconds <secs>, as a decimal number, or by use of one or more flags.

The flags are:

-h# interpret # as hours.

-m# interpret # as minutes.

-s# interpret # as seconds. If <secs> is present, its value overrides #.

If no arguments are specified, sleep defaults to 10 seconds.

RETURNS

sleep returns success if the sleep system call returns success.

EXAMPLE

```
% {sleep -m40; echo "END OF MEETING"}&
```

NAME

sort - order lines within files

SYNOPSIS

sort -o* +[a b d l n r t? #.#-#.#] <files>

FUNCTION

sort merges and reorders text lines among all the <files>, moving lines as specified by the ordering rules on its command line, and produces a single output file. If no <files> are given, sort takes input from STDIN. A filename of "-" also causes STDIN to be read, at that point in the list of files.

The flags are:

-o* direct output to named file. This can be used to sort a file in place.

In addition, up to ten ordering rules may be specified: if the first rule results in an equal comparison then the second rule is applied, and so on until lines compare unequal or the rules are exhausted.

Rules take the form:

[adln][b][r][t?][#.#-#.#]

where

a - compares character by character in ASCII collating sequence. A missing character compares lower than any ASCII code.

b - skips leading whitespace.

d - compares character by character in dictionary collating sequence, i.e., characters other than letters, digits, or spaces are omitted, and case distinctions among letters are ignored.

l - compares character by character in ASCII collating sequence, except that case distinctions among letters are ignored.

n - compares by arithmetic value, treating each field as a numeric string consisting of optional whitespace, optional minus sign, and digits with an optional decimal point.

r - reverses the sense of comparisons.

t? - uses ? as the tab character for determining offsets (described below).

#.#-#.# - describes offsets from the start of each text line for the beginning (first character used) and, after the minus '-', for the end (first character not used) of the text field to be considered by the rule. The number before each dot '.' is the number of tab characters to skip, and the number after each dot is the number of

characters to skip thereafter. Thus, in the string "abcd=efgh", with ´=´ as the tab character, the offset "1.2" would point to ´g´, and "0.0 would point to ´a´. A missing number # is taken as zero; a missing final pair "-#.#" points just past the last of the text in each of the lines to be compared. If the first offset is past the second offset, the field is considered empty.

If no tab character is specified, each contiguous string of whitespace will be taken as a tab. Thus, in the string "\ ABC\ \ DEF\ GHI", the offset "3" would point to ´G´.

Only one of a, d, l, or n may be present in a rule; if none are present, the default is a. In a given rule, letters appearing after "#.#-#.#" will be ignored.

Null fields compare lower in sort order than all non-null fields, and equal to other null fields.

RETURNS

sort returns success if all files supplied are opened, all flags and sort rules are valid, and a sorted file is produced.

EXAMPLE

The command:

```
% sort winter
```

takes as input the file winter and sorts the lines therein by ASCII values. The output is written to STDOUT.

To sort the file chaos into the output file order, without differentiating between upper and lowercase letters:

```
% sort +1 -o order chaos
```

To sort the file dictionary, examining specified fields within the lines:

```
% sort +at:5.0-6.0 +at:2.0-3.0 +1 -o useful disarray
```

The file disarray is first sorted by the field between the fifth and sixth tab character (´:´ in this case) then lines that compare equal are sorted by the field between the second and third colons. Equal lines are further sorted using the rule 1.

SEE ALSO

comm, uniq

NAME

spell - discover potential spelling errors within files

SYNOPSIS

spell -[d* i*] <files>

FUNCTION

spell examines words one at a time from the <files>, checking them against a special dictionary. If no <files> are given, spell takes input from STDIN. A filename of "-" also causes STDIN to be read, at that point in the list of files.

The flags are:

-d* obtain the dictionary from the file named.

-i* obtain the dictionary index from the named file.

The dictionary is encoded to save space and searching time. Because of this encoding, an index to the dictionary must also be maintained. The default name for the dictionary is "/usr/lib/dict", while the index file default is the name of the dictionary with ".ix" appended.

A word is considered to be a sequence of alphabetic characters with at most one apostrophe. A word is deemed misspelled if it is not in the dictionary, and it contains no apostrophe or the root of the word (up to but not including the first apostrophe) is also not in the dictionary. Thus, "fred's" is correctly spelled (fred is in the dictionary), "ain't" is correctly spelled (sic), but "ain" is not correct. Before checking, the words are mapped to lower case, so proper names without capitals may be deemed correct.

RETURNS

spell returns success if all files supplied are opened, all flags are valid, and there are no putative spelling errors.

EXAMPLE

The command:

```
% spell winter
```

takes as input the file winter and spells the words therein by comparing against the dictionary. The output is written to STDOUT.

SEE ALSO

md, sort

BUGS

Because of the rather naive rules for apostrophes, words like "aardvark'edxyz" are assumed correct.

NAME
su - set userid

SYNOPSIS
su -[c* g p* u] <loginid>

FUNCTION
su executes a shell, after changing the userid and/or groupid to that associated with <loginid>. If <loginid> has a password associated with it, su will validate it.

The flags are:

-c* execute the command * and quit, rather than start an interactive subshell.

-g change the groupid.

-p* take the string * as the password, instead of prompting for it.

-u change the userid.

If neither -u or -g are specified, both are assumed. If no loginid is specified, "root" is assumed. Thus the default case is "su -ug root".

RETURNS
su returns the exit status of the command executed if the password is correct.

EXAMPLE

```
% su -c "/etc/chown myid hisfile"
password:
```

FILES
/adm/passwd for the password file.

SEE ALSO
sh

NAME

tar - read or write a tar format tape

SYNOPSIS

tar -[bs# c +d f# l p t v x] <files>

FUNCTION

tar administers a tar format "tape", i.e., a single physical file, written or read in some multiple of 512-byte blocks, that can represent numerous files transparently, including some of their attributes such as ownership and date of last modification. When written to a tape, diskette, or other interchange media, a tar file facilitates transfer of multiple files between Idris and UNIX systems.

When creating a tape, tar recursively descends all directory files, copying all files in each directory to the tape. If no <files> are specified, the current directory is assumed, but the directory file itself is not dumped. If a file is a link to another file already added to the tape, tar will recognize the fact and dump only the header information needed to recreate the link when extracted.

When extracting, printing, or tabulating a tape, only the files named in <files> participate. If a name is given that is a directory, all files in that directory on the tape participate. If no <files> are given, all files on the tape are used.

If an extract is performed by the superuser, the userid and groupid from the tape are retained; otherwise the new files are owned by the user doing the extraction. In either case, all of the mode bits from the tape are retained.

The flags are:

-bs# read or write # blocks at a time, for more efficient use of tape. Default is to write one block records.

-c create a tape. All named <files> are composed into a tar tape in the order specified.

+d recursively create directories when used with -x.

-f# use the string # as the name of the tape. The default is /dev/mt0, unless a blocking factor other than 1 is specified, in which case the default becomes /dev/rmt0. A single minus sign as the file name implies STDIN for tape reading operations, and STDOUT for tape writing operations.

-l warn about any files which have multiple links, where not all of the links have been dumped to tape. The default is to keep quiet about them.

-p print files from the tape to STDOUT.

-t tabulate the tape.

-v be verbose.

-x extract files from the tape into disk files with the same name.

At most one of -c, -p, -t, or -x may be specified; the default is -t. When -v is specified with -c or -x, the name of each file processed is written to STDOUT. When -v is specified with -p, a formfeed character is written before each filename output. When -v is specified with -t, all information about each file to be tabulated is printed: the file mode is displayed first, followed by userid, groupid, length of the file in bytes date and time of last file modification, and name of file. If the file is a link to another file already passed on the tape, the name is followed by " linked to <otherfile>".

RETURNS

tar returns success if all files specified can be processed.

EXAMPLE

```
% tar -vt
-rw-rw-rw-  27   2      12482 Oct 17 10:52 tar.c
-rw-rw-rw-  26   0       3139 Oct 19 18:51 tar.mp
drwxrwxrwx   0   0          0 Oct 19 18:57 tc/
-rwxrwxrwx  26   2         73 Oct 17 10:52 tc/mk
-rw-r-----  26   2        367 Oct 17 10:52 tc/tc1.c
-rw-r-----  26   2        836 Oct 17 10:52 tc/tc2.c
-rw-r-----  26   2       1367 Oct 19 18:55 tc/tc3.c
-rw-r-----  26   2       1367 Oct 19 18:55 tc/tc4.c linked to tc/tc3.c
-rw-r-----  26   2       4099 Oct 17 10:52 tc/termcap.c
-rw-r-----  26   2       2629 Oct 17 10:52 tc/tgoto.c
-rw-r-----  26   2       1854 Oct 17 10:52 tc/tputs.c
11 files tabulated
```

BUGS

Changing ownership as superuser should be optional. If tar is installed with superuser powers, its checking of permission is somewhat lax. When using -v, -c and -f- together, the file name output will conflict with the structure of the created file on standard output; tar shouldn't allow this combination but does anyway.

NAME

tee - copy STDIN to STDOUT and other files

SYNOPSIS

tee -[a] <files>

FUNCTION

tee copies its standard input to its standard output, and to each of the files in <files>.

The flag is:

-a append the contents of STDIN to each argument file. Default behavior is to create a new instance of each file before starting.

The -a flag may not work correctly on systems other than Idris/UNIX, since appending to text files is not always well supported.

RETURNS

tee returns success if it could open or create all of its argument files, and if no errors occur in writing them.

EXAMPLE

To append a note to three different files:

```
% tee -a file1 file2 file3 <note
```

NAME
test - evaluate conditional expression

SYNOPSIS
test <args>

FUNCTION
test evaluates the conditional expression formed by <args> and returns success if the expression is true, otherwise failure.

test takes zero to three <args>, which are interpreted either as strings, as the unary operator not "!", or as the binary operators equal "==" or not equal "!=". The operators are position dependent.

test thus accepts the following expressions:

string	true if string is not null.
! string	true if string is null.
string1 == string2	true if strings are identical.
string1 != string2	true if strings differ.

test can be used to advantage in conjunction with the shell operators "&&" and "||". The statement "if expr then cmd" can be expressed as

```
test expr && cmd
```

The statement "if not expr then cmd" can be expressed as

```
test expr || cmd
```

RETURNS
test returns success when the expression formed by its arguments is true; otherwise it returns failure. If the syntax of the expression formed by <args> is invalid, test complains and returns failure.

EXAMPLE
To exit a shell script if the value of $1 is null:

```
% test $1 || exit
```

SEE ALSO
sh

NAME

time - time a command

SYNOPSIS

time <command>

FUNCTION

time executes <command>, then reports to STDERR the real, user, and system time consumed. Real time represents wall clock time, user time represents user program time, and system time represents the time used in system calls. If no <command> is specified, the default command executed is date.

Each time is reported as hh.mm.ss.xx, where hh is the number of hours, mm the number of minutes, ss the number of seconds, and xx the number of 1/60 seconds.

RETURNS

time returns the completion status of <command>.

EXAMPLE

To time a ls command without printing its output:

```
% time ls -l > /dev/null


real        3.00
user        0.43
sys         2.17
```

SEE ALSO

date

NAME

tp - read or write a tp format tape

SYNOPSIS

tp -[b* c +d f* i p s* t v x] <files>

FUNCTION

tp administers a tp format "tape", i.e. a single physical file, written or read as 512-byte blocks, that can represent numerous files transparently, including some of their attributes such as ownership and date of last modification. When written to a tape, diskette, or other interchange media, a tp file facilitates transfer of multiple files between Idris and UNIX systems.

When creating a tape, tp recursively descends all directory files, copying all the non-directory files to the tape. If no <files> are specified, the current directory is assumed.

When extracting, printing, or tabulating a tape, only the named files participate. If a name is given that is a directory, all files in that directory on the tape participate. If no <files> are given, all files on the tape are used.

If an extract is performed by the superuser, the userid and groupid from the tape are retained; otherwise the new files are owned by the user doing the extraction. In either case, all of the mode bits from the tape are retained.

The flags are:

-b* write the string * as a bootstrap program to block zero when creating a tape. Default is to write a block of NULs. Useful only with -c.

-c create a tape. All named <files> are composed into a tp tape in the order specified.

+d recursively create directories when used with -x.

-f* use the string * as the name of the tape. The default is /dev/mt0.

-i take input from STDIN. Each filename must be followed by a newline.

-p print files from the tape to STDOUT.

-s* strip off prefix * from pathname before creating extracted files. If * does not end in "/", tp appends one to * before stripping prefix. -s is only meaningful with -x.

-t tabulate the tape.

-v be verbose.

-x extract files from the tape into disk files with the same name.

At most one of -c, -p, -t, or -x may be specified; the default is -t. When -v is specified with -c or -x, the name of each file processed is written to STDOUT. When -v is specified with -p, a formfeed character is written, followed by the name of the file. When -v is specified with -t, all information about each file to be tabulated is printed: the file mode is displayed first, followed by userid, groupid, beginning tape block number of file, size in bytes, date and time of last file modification, and name of file.

RETURNS

tp returns success if all files specified can be processed.

EXAMPLE

```
% tp -vt
-rwxrwxrwx   0   1    63      38 Nov 13 10:26 .profile
-rwsrwxrwx  20   1    64   24828 Jan 06 15:22 afile
-rw-rw-rw-  20   1   113      12 Jan 06 14:56 errors
-rw-rw-rw-   2   1   114    5546 Dec 29 12:49 mbox
-r--r--r--  20   1   125    3347 Dec 05 17:25 old/cmp.c
-rw-rw-rw-  20   1   132    7347 Dec 11 14:29 old/shexec.c
-rw-rw-rw-  20   1   147    5879 Dec 11 14:29 old/shmain.c
-rw-rw-rw-  20   1   159    5676 Dec 11 14:29 old/shparse.c
-rw-rw-rw-  20   1   171   10727 Jan 06 14:22 otp.c
-rw-rw-rw-  20   1   192    2554 Jan 06 14:26 temp
-rw-rw-rw-  20   1   197   11324 Jan 06 15:22 tp.c
-rw-rw-rw-  20   1   220   10800 Jan 06 14:56 tp.o
-rw-rw-rw-  20   1   242     503 Jan 06 15:22 tpls
-rw-rw-rw-   0   0   243    1504 Dec 18 13:35 tpt
```

SEE ALSO

tp(III)

BUGS

Changing ownership as superuser should be optional.

NAME
tr - transliterate one set of characters to another

SYNOPSIS
tr -[b o* t*] <src> <files>

FUNCTION
tr scans one or more input files for characters specified in <src> and writes a transliterated version to the output file. Characters not found in <src> are written to output unchanged. If no destination string <dest> is specified (with -t*), then all characters in <src> are deleted. Otherwise, each <src> character is transliterated to its corresponding <dest> character. A character found in <src> for which there is no corresponding character in <dest>, (i.e., <dest> is shorter than <src>), is mapped into the last character of <dest>; subsequent repetitions of the character are deleted on output.

\&'!' as the first character of <src> specifies that the <src> string should be expanded to include all of the 256 characters except those specified following the '!'. In this case, all <src> characters are mapped and collapsed into the last character in <dest>.

Both <src> and <dest> can contain ranges of characters (e.g. a-z indicates all lowercase letters). Escape sequences are also valid, as in the editor e (e.g. \n indicates newline and \01-\014 indicates the sequence octal 1 through octal 14). If either <src> or <dest> expand to more than 256 characters, only the first 256 are used. When a range of characters is given such that the last character is less than the first (e.g. z-a), the range is ignored. If the same character if specified more than once in the <src> string, all occurrences of the character in input are mapped into the character in <dest> corresponding to the first occurrence in <src>.

The flags are:

-b open input for binary read.

-o* write output to *. The default is STDOUT.

-t* transliterate <src> to *, which is the destination sequence <dest>. Default behaviour is to delete all characters in <src>.

"-" in the list of file arguments specifies STDIN. If no file arguments are given, input is taken from STDIN. Output is written to STDOUT unless -o is specified.

RETURNS
tr returns success if all flags are coherent and files can be opened.

EXAMPLE
To isolate one word per line from the file paper:

```
% tr -t"\n" !a-zA-Z <paper
```

NAME

uniq - collapse duplicated lines in files

SYNOPSIS

uniq -[c f g o* u] +[a b d l n r t? #.#-#.#] <files>

FUNCTION

uniq reads lines of input from the files in the list <files> and determines which lines are unique by comparing each one to the immediately adjacent lines. If no <files> are given, uniq takes input from STDIN. A filename of "-" also causes STDIN to be read, at that point in the list of files.

Comparisons may be made based on arbitrary parts of each line by using the sort keys described below. uniq can then be made to output only the unique lines, only the first of each group of duplicate lines, only the remainder of each such group, or some combination of all three. Input files must be in sort, by the same comparison rules, for duplicate lines to be reliably recognized as such.

The flags are:

-c preface each line output with a count of how often it occurs. Implies -f and -u.

-f output only the first line in each group of duplicate lines.

-g output all lines but the first, in each group of duplicate lines.

-o* direct output to file *. The default is STDOUT.

-u output only lines that are unique.

If none of -f, -g, or -u are specified, -f and -u are assumed. None of -c, -f, or -u may be given if -g is.

In addition, up to ten ordering rules may be specified: if the first rule results in an equal comparison then the second rule is applied, and so on until lines compare unequal or the rules are exhausted.

Rules take the form:

[adln][b][r][t?][#.#-#.#]

where

a - compares character by character in ASCII collating sequence. A missing character compares lower than any ASCII code.

b - skips leading whitespace.

d - compares character by character in dictionary collating sequence, i.e., characters other than letters, digits, or spaces are omitted, and case distinctions among letters are ignored.

l - compares character by character in ASCII collating sequence, except that case distinctions among letters are ignored.

n - compares by arithmetic value, treating each field as a numeric string consisting of optional whitespace, optional minus sign, and digits with an optional decimal point.

r - reverses the sense of comparisons.

t? - uses ? as the tab character for determining offsets (described below).

#.#-#.# - describes offsets from the start of each text line for the beginning (first character used) and, after the minus '-', for the end (first character not used) of the text field to be considered by the rule. The number before each dot '.' is the number of tab characters to skip, and the number after each dot is the number of characters to skip thereafter. Thus, in the string "abcd=efgh", with '=' as the tab character, the offset "1.2" would point to 'g', and "0.0 would point to 'a'. A missing number # is taken as zero; a missing final pair "-#.#" points just past the last of the text in each of the lines to be compared. If the first offset is past the second offset, the field is considered empty.

If no tab character is specified, each contiguous string of whitespace will be taken as a tab. Thus, in the string "\ ABC\ \ DEF\ GHI", the offset "3" would point to 'G'.

Only one of a, d, l, or n may be present in a rule; if none are present, the default is a. In a given rule, letters appearing after "#.#-#.#" will be ignored.

Null fields compare lower in sort order than all non-null fields, and equal to other null fields.

RETURNS

uniq returns success if all of its input files are readable.

EXAMPLE

To count duplicates, ignoring case distinctions:

```
% uniq -c +1 <sorted
```

SEE ALSO

comm, sort

NAME

up - pull files uplink

SYNOPSIS

up

FUNCTION

up is invoked on the remote computer, in conjunction with the call up utility cu, to cooperate with the \u command. up creates the files on the remote computer, using the names supplied by the \u command, and builds the files from packets received. up assumes it is talking to dn, which is invoked locally by cu.

RETURNS

up returns success if all operations succeed.

EXAMPLE

```
% up
\u -p/tmp/ *.c *.h *.mp
```

SEE ALSO

cu, dn

BUGS

A better implementation would merge the up utility and the \u command.

NAME
wait - wait until all child processes complete

SYNOPSIS
wait

FUNCTION
wait suspends execution of the current process until all child processes have finished executing. wait is useful for re-synchronizing after starting one or more processes asynchronously.

RETURNS
wait returns the status of the last child waited on.

EXAMPLE

```
% wait
```

SEE ALSO
sh

NAME

wc - count words in one or more files

SYNOPSIS

wc -[b c l n## p w] <files>

FUNCTION

wc counts the number of pages, lines, words, and characters in each of one or more files. Totals are given if more than one file is specified. A word is taken as a contiguous string of characters delimited by whitespace.

The flags are:

-b treat all input files as binary.

-c count number of characters.

-l count number of lines.

-n## change default number of lines per page to ##

-p count number of pages. The default number of lines per page is 56.

-w count number of words.

If any flags are specified, only the counts indicated are given. The default is all of the counts.

"-" in the list of arguments is taken as STDIN. If no files are specified, input is taken from STDIN.

RETURNS

wc returns success if all flags are coherent and all files can be opened.

EXAMPLE

```
% wc file1 file2 file3
    1      12     24     84  file1
    3     165    730  10624  file2
    1      43    172    948  file3
    4     220    926  11656  TOTAL
```

NAME

who - indicate who is on the system

SYNOPSIS

who -[h f*] <am i>

FUNCTION

who reports the loginid, login time, and controlling ttyname, for each current user, sorted by ttyname. If who is followed by any non-flag arguments (traditionally "am i"), information is given only for the invoking terminal. who can also be used to report the recent history of system usage, printing login and logout entries; entries are also reported for each system startup and date change.

The flags are:

-f* interpret * as the history file instead of /adm/log.

-h report history instead of current users.

History information is sorted in reverse order of time (latest first).

RETURNS

who returns success if all reads and writes succeed.

EXAMPLE

```
% who
msk       Jun 09 12:35:51 tty9
bob       Jun 09 10:40:29 ttya
tdp       Jun 09 13:21:36 ttyb
```

FILES

/adm/log for history, /adm/who for current users.

NAME

write - send a message to another user

SYNOPSIS

write -[a break d* t*] <loginid>

FUNCTION

write either sets up a conversation between two users, or sends a broadcast message to all users currently using the system. While in conversation mode, messages are sent a line at a time as they are entered. In broadcast mode, the message is saved up until it is complete, then sent to all logged in users. The conversation/broadcast is terminated with a '.' on a line by itself, or end of file (as by typing a ctl-D).

Messages sent via write are prefixed by:

Message from <your loginid>

for conversations, or

Broadcast from <your loginid>

for broadcasts.

where <your loginid> is the loginid of the user invoking write.

Commands may be executed while in write by prefixing the command line with a '!'.

The following conventions are suggested for conversations:

One user issues a write command to another and waits for the second user to respond. If the other user doesn't respond within a reasonable amount of time, the originating user should terminate the message, and try again later. If the second user does respond, the session continues with each user typing a message (as many lines as necessary) followed by "(o)" for "over". To terminate the conversation, send "(oo)" for "over & out", then wait for acknowledgement before terminating the message.

The flags are:

-a send broadcast to all logged in users.

-break break through any protection that users may have invoked. This flag will only work for the superuser.

-d* send message to /dev/*.

-t* send message to /dev/tty*.

The flags -d* and -t* are useful for disambiguating among multiple ttys logged in with the same loginid. Either the loginid or exactly one of the flags -[a d* t*] may be given.

EXAMPLE

```
# write -a
The system will be taken down in 15
minutes for maintenence.
.
Broadcast from root: a
The system will be taken down in 15
minutes for maintenance.
EOT
```

SEE ALSO

mesg, mail