Idris Users' Manual

COMPUTER TOOLS INTERNATIONAL
649 STRANDER BLVD. SUITE E
SEATTLE, WA 98188
(206) 575-3606

Whitesmiths, Ltd.

IDRIS USERS' MANUAL

Date: April 1984

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

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

IDRIS USERS' MANUAL

SECTIONS

SCOPE

This manual introduces the Idris operating system to non-programmers. Section I provides detailed, tutorial descriptions of the major languages used to interact with the system, along with the conventions that guide their use. Section II describes, in much more succinct form, all of the standard utilities available for general use. Section III, of less interest to novice users, details the formats of the various kinds of files accepted or produced by Idris utilities, while Section IV is a guide to the utilities that Idris provides for system administration, as well as to some common maintenance procedures.

Information of interest specifically to programmers may be found in the Idris Programmers' Manual. Details on the machine dependent aspects of each implemetation may be found in the Idris Interface Manual for the appropriate target machine.

TABLE OF CONTENTS

I. Introduction to Idris

II. Standard Utilities

III. Standard File Formats

IV. System Administration Guide

I. Introduction to Idris

NAME

Manuals - a guide to Idris documentation

SYNOPSIS

Idris Users' Manual
- I. Introduction to Idris
- II. Standard Utilities
- III. File Formats
- IV. System Administration Guide

Idris Programmers' Manual
- I. Whitesmithing
- II. Idris System Interface
- III. Programming File Formats
- IV. Idris Support Library

Idris Machine Interface Manuals
- I. System Generation Guide
- II. Idris System Interface
- III. Device Handlers
- IV. Device Driver Interface

FUNCTION

Idris is a comprehensive operating system that provides a uniform operating environment across a broad spectrum of computers. Modelled closely after the highly successful UNIX operating system developed at Bell Labs, it incorporates numerous facilities of proven worth for both naive and experienced users. It is sufficiently elaborate, as a result, that it is not easily described in just a few brief essays.

Hence the description of Idris is partitioned into three separate manuals: one for everybody who uses the system, one for those who must add programs to the system, and one for those who must be aware of the particulars of a given underlying machine. A different version of the Idris Machine Interface Manual exists for each family of computers sharing a given instruction set; those who wish to maximize portability will do well to ignore all instances of this particular manual and confine their study to the first two, which describe Idris facilities common to all machines.

The Idris Users' Manual contains several essays, such as this one, covering the gross aspects of the operating system. Here one can find "manual pages", i.e. terse descriptions of the scores of standard utilities which are often sufficient in themselves to solve data processing chores. There is also a section describing any file formats presumed by one or more of the standard utilities, so that other utilities can be used to create or dissect them. Finally, there is a section summarizing the tools provided to the system administrator, with some guides to running a happy Idris environment.

The Idris Programmers' Manual focuses on the facilities typically used only in the process of developing new utilities or applications programs, a process known to some as "whitesmithing". Programs such as the linker, librarian, and C compiler driver are documented here, as is the portable interface to all system calls and file formats of interest principally to programmers. A library of functions used extensively by the Idris standard utilities is also documented in this manual, to simplify the produc-

tion of new utilities by users and to encourage more uniform practices.

Each Idris Machine Interface Manual begins by describing any peculiarities of the operating system implementation on the given machine. Particular attention is given to the "resident", which is the program that resides at all times in computer main memory to schedule resources and to carry out requests from other programs. Here is given information on how to configure the resident for a specific piece of hardware. The machine level interface to the system is described, and all device handlers supplied with the system are documented, to reveal their foibles and to serve as models for user written handlers. And finally, there is a section describing how to write new device handlers, which includes a set of manual pages describing the functions that may be called from a device handler or other user addition to the Idris resident.

BUGS

Part of the UNIX heritage is a tendency to provide documents that are exquisitely precise, but terse. The Idris project has fully embraced this approach to documentation, and has often achieved new heights of brevity, due to the constraints of an ambitious commercial enterprise. Everything you need to know about Idris is in these manuals, but it is often as hard to digest as grandma's brandied fruitcake.

Chew thoroughly.

NAME

Login - getting started with Idris

SYNOPSIS

login: <loginid>
password:

FUNCTION

Idris is an operating system that tries to be friendly to its users by maintaining a fairly uniform appearance across its many utilities and by trying to do something sensible for nearly any request made of it. As a consequence, it is not nearly as chatty as most "human engineered" systems, nor does it produce many or elaborate error messages. It does seem to strike a balance, however, that makes many people happy, be they bystanders or warriors in the data processing revolution.

It is an interactive system, which means it typically spends most of its time waiting for people to type things at it (as opposed to reading batches of card decks or monitoring the pressure gages in a refinery). It is also a multi-user system, which means it can talk to several people at once, and hence must have some way of telling them apart. So to get anything done with Idris, you have to first get its attention and then learn how to talk to it.

All this documentation presumes the existence of a "system administrator", who may be just another persona of the one and only user, but who will be treated as a separate entity, to keep discussions factored. The system administrator will be saddled with the chores that conventional interactive users should normally be relieved of (or denied). Candidates for system administrator are referred to the last section of this manual, which is devoted to that job.

At any rate, to start using Idris you have to: a) get a terminal logically connected to the computer, b) get Idris to issue a login process for that terminal, and c) get installed in the system a loginid and password for your personal use. See your system administrator for all of the above.

A "login process" is just a computer program, whose mission in life is to wait for a terminal to get logically connected and then write the invitation "login: " at the terminal and wait for input. The user wishing to login to Idris must correctly type a loginid, which is one to eight characters (usually letters, or at least printable stuff), followed by a carriage return (or linefeed on some terminals). If the loginid is known to the system, and if there is a password required for that loginid (there usually is), then the login program proceeds to write "password: " at the terminal and wait for another line of input; only this time the computer is careful not to show the characters of the password being typed at it.

If these hurdles are not both surmounted, the login program doggedly repeats its "login: " invitation; but upon success, the login program turns control over to whatever program the system administrator has assigned for use with that loginid. This is usually a very helpful program called, of all things, the "shell", or "sh" for short. The shell spends

most of its time waiting for you to type a line at it, which it then endeavors to interpret as the name of yet another program to run on your behalf, possibly with some information passed to it about what files to use or what options to exercise. Unlike login, the shell does not just pass on the torch and quit; instead, it hangs around to print out any post mortem, if the other program dies an unnatural death, and to read still another request line from you. But more on that in a later essay.

The important thing to note is that each piece of information requested of you takes the form of "a line of text". This is much like the line one might type on a conventional typewriter, except that the computer looks on each keystroke as a separate character, even if that character calls for tabbing, backspacing, or returning the carriage to the left margin. Thus a line of text consists of zero or more printable characters, or spaces, or tabs, etc. up to a line terminator such as a carriage return or line feed. Since we all make mistakes, Idris permits you to do a modicum of editing of the stuff you type; it promises not to look at the line until the line terminator is typed. Up until then, you can take back the last character typed by typing a backspace (usually), which neatly erases the character, on a video terminal at least, and leaves you in position to retype the character or to erase even more. You can also scrub the entire line and start over by typing the at-sign '@' (usually), which causes an immediate return to the left margin as a signal that the previous information has been discarded. Those parenthetic "usually" hedges are a hint that you can even change which keys cause character and line delete, but at the risk of some confusion until you get used to the changes.

There are other things you can do via the terminal keyboard. If output is coming too fast for you to digest, you can suspend printing by typing a ctl-S (hold down the "control" key and type the letter S). You can resume output by typing anything else, but a ctl-Q will do the job innocuously, since neither ctl-S nor ctl-Q are passed on to the programs reading the terminal. Neither of these keys affect any program running on your behalf, except perhaps to delay it if it is writing to your terminal.

Other keys do affect running programs, however. You can simulate an "end of file" condition, needed to bring to a tidy conclusion a program reading what you type at the keyboard, by typing a ctl-D. You can bring most programs to an abrupt and untidy conclusion with an "interrupt", obtained by typing a DEL (sometimes represented by the RUBOUT or DELETE keys) or by striking the BREAK key. And to terminate a program with extreme prejudice, complete with abrupt cessation and a "core" file for post mortem examination, type ctl-\.

All of these magic keys, and a few more, are summarized in Section III of this manual, on the essay titled tty (usually written tty(III)). This essay describes all of the peculiar properties of a terminal when treated as a file.

While any program running under Idris has the right to bypass all of this keyboard interpretation, it is a rare one that does. Thus, no matter what program you are talking to, you have a uniform set of rules for framing your requests and for throttling its responses. You can even type ahead quite a number of characters, assuming you know what input is appropriate

in the near future, and not worry about its being mislaid between programs.

Just what to say to each program is a strong function of the program, but there are a number of system-wide conventions. Among other things, there are standard conventions: for specifying the options to be exercised by a program, for selecting whole groups of files for processing by a program, and for selecting input and output files to be used by a program. The preoccupation with files stems from the fact that all long term memory in the Idris system takes the form of files of information, stored on disks or tapes as a rule. Programs are files, the data you care about are stored in files, even the information needed to track down files is stored in still other files. The stuff you type at the keyboard looks to most programs like just another file, albeit more transient than most, since files are almost always passed sequentially once through a program. Idris can best be viewed as an arena for the convenient creation and manipulation of files.

The login program gets you into that arena.

BUGS

Subtleties that are tempting to gloss over, but which may confuse the novice user are:

If the terminal displays garbage when you are expecting the "login: " invitation, there is probably a speed mismatch between terminal and computer. The login program is usually instructed to try a different speed each time an interrupt is received from the terminal; this can be made to happen even from a wrong speed terminal by striking the break key. If three or more tries fail to produce a clean invitation, see your trusty system administrator.

Typing a ctl-D to signal end of file from a terminal works by a bit of trickery that only comes off if the ctl-D is the first character on a line. In other positions, two ctl-D characters must be typed; this produces a partial last line which many programs find unpalatable anyway.

One non-printing character that frequently occurs in messages from the computer is the bell key (an ASCII BEL) or ctl-G. This causes a ding, beep, toot, or silence, depending upon the noisemaker available in the terminal. The login program rings BEL alot, as does Idris when it gets upset over something. Don't be scared.

NAME

Commands - the shell game

FUNCTION

So, you've logged in successfully to Idris. Some sort of message has probably been typed at you, and the last thing displayed is a percent-sign and space "% " on a line by itself. What now?

Now, the system is waiting for instructions, usually called commands, or command lines. A command (slightly misnamed) is simply a request for Idris to do something. Almost always, commands are funneled through a program called the shell, or sh. The percent-sign (called a prompt) is output by the shell to indicate that it is ready to accept a command, which generally names a program that you want to run. The programs discussed in this essay are called standard utilities. From now on, when we mention a "utility", we simply mean one of these programs, whose name you give to the shell as a command. Whenever a command finishes execution, the shell will output another prompt, meaning that it is once again waiting for input.

You should be familiar already with the basics of talking to Idris. If not, re-read the preceding essay; the conventions described there apply to the shell just like to any other program. Here, only one reminder: Idris will not act on the line you are currently entering until you terminate it, which means you can edit the boots off of it, but which also means that you must terminate it to get any response. Usually, you type a carriage return in order to terminate a line; depending on your terminal, however, line feed may serve the same purpose. We will use the single term newline to refer to whichever terminator you must use. A newline ends each line of input you type; likewise, a newline ends each line of text output or stored by the computer.

NAMING COMMANDS

In the following examples, lines preceded by a prompt are what you should type to try out the example; other lines are the system's response to the input given. The newline at the end of each input line can't be seen, but it must be there, nonetheless.

Just to test the water, try typing:

```
% date
```

Idris should respond with something like:

```
Mon Sep 07 12:18:49 1981 EDT
```

Like it? The system will provide other interesting information, too. As a further example, try:

```
% who
```

That should net you a listing of who is currently using the system, like this one:

```
tana     Sep 07 12:20:29 console
pjp      Sep 07 08:01:46 tty1
cnh      Sep 07 09:35:22 tty2
djb      Sep 07 09:54:53 tty4
```

The first field on each line is the name each user specified to login. Next is listed the time at which the user logged in, followed by the name of the terminal location being used.

COMMAND ARGUMENTS

The uses of date and who that we've just seen are examples of the simplest kind of command line, one giving a command name and nothing else. To be really useful, however, commands generally need to supply other information, as well. This is done with arguments, which are simply additional words on the command line, separated by spaces from the command name and from each other. (More precisely, an argument is a series, or string, of characters separated from its surroundings by whitespace.)

One simple utility that takes arguments is called echo. Try it out:

```
% echo boo!
```

The response explains its name:

```
boo!
```

More generally, echo will display all of its arguments, exactly as they appeared on its command line. That's all it does:

```
% echo what use is this?
what use is this?
```

Though perhaps unexciting right now, echo is quite useful in other situations, as you'll discover later.

If you make a mistake in entering a command name, the shell will tell you of it. For example, typing:

```
% data
```

yields the response:

```
data: not found
```

After displaying the warning message, the shell forgives and forgets; when the prompt reappears, it's ready to accept another command.

FLAGS

The utilities so far discussed are obviously all quite different from each other, and Idris provides dozens more, of even greater diversity. Often, however, it's desirable to alter the operation of a utility in some relatively small way, without the expense of creating another, separate program. Most utilities accept a special kind of arguments, called flags, that change some aspect of what the utility does, or select one option among many possible ones. Flags are almost always given before any other arguments, and invariably begin with a '-' or a '+', so that everyone involved can recognize them. They are usually optional.

Instead of the command line for echo shown earlier, try entering:

```
% echo -m what use is this?
```

The response may surprise you:

```
what
use
is
this?
```

The flag you gave accounts for it: -m causes echo to separate its arguments with newlines instead of spaces when it outputs them, thus making each word print on a separate line. (A newline, you may recall, is the character that ends a line of text.) This still may not appear very useful to you, but it is.

If you specify a flag that a utility doesn't recognize, the program will display a hint to remind you of what the legal flags are (assuming you have read the manual page for the program and merely need reminding). Entering:

```
% echo -q
```

results in the message:

```
usage: echo -[m n] <args>
```

which says that the program echo accepts the flags -m and -n (we haven't explored -n yet), followed by <args>, a series of arguments, as explained above. If you forget what flags a utility accepts, you can force it to output a usage message by typing the flag -help. Thus

```
% echo -help
```

will also result in the message shown above.

CREATING FILES

Doing anything significant with Idris pretty much depends on learning about files. A file can be viewed as a named storage place for informa-

tion. Most of the time, a file contains text, so creating a file under Idris amounts to getting some text into the computer and saving it under the name you want. An easy way to do that is with the text editor e. A later essay explains e in detail; for now, we'll go over the bare minimum you need in order to enter text.

First, start the editor running,, by typing:

```
% e -np
```

Don't forget the space between the 'e' and the '-'. (The -np tells e to output a '>' prompt when ready for a command, and a line number prompt when ready to append text.) e will respond with its command prompt, a '>'. Once the prompt has appeared, type an 'a', followed by a newline; this is the editor command to begin "appending" text. Henceforth e will prompt for text lines with a line number, and save any text you enter in an internal area called a "buffer", until you type a period on a line by itself. Then, e will prompt you for another editing command:

```
>a
1   any text you like --
2   for as many
3   lines
4   and lines
5   and lines
6   as you like.
7   .
>
```

Try entering some text (any text) as described. After you've ended it with a period, you can perform a variety of editing operations on it. The next step, though, is just to write what you've entered to a file. To write it to a file called trial, for instance, you would type:

```
>w trial
```

The editor will then reassure you that it has written the file by telling you the number of characters it wrote. (Of course, you can give any filename you like after the w editor command; a later essay will discuss file naming rules and conventions.) As a further exercise, try adding some more text to what you've already entered. Just type another append command, and end the text with a second '.'. Then write out the buffer again, this time to a different file, say new:

```
>a
7   then some more
8   lines on the end
9   .
>w new
```

About now you may be seized with an understandable urge to look at what you've typed in. To get e to oblige you, type:

```
>1,$p
```

The 'p' editor command prints some part of the editor's buffer. In this case, you asked to see all of it (from first line '1' to last line '$'), so e should have displayed the text from both a commands. Finally, you can quit the editor, and get your shell prompt "% " back, by typing a 'q' editor command:

```
>q
```

MANIPULATING FILES

You should now have created two files, the first of which contains the text you entered with the first append command, and the second of which contains the text from both append commands. First of all, to find out what files you currently have, list their names with the ls utility:

```
% ls
```

If you named your files as shown, the response will be:

```
new
trial
```

which are indeed the two files we asked to be created, listed alphabetically. Now that we know they're out there, what else can we do with them? Perhaps the most natural thing is to look at their contents again, just to make sure. The simplest program for doing so is called cat (for concatenate files, legend has it). To use it, just name the file (or files) you want to examine, and cat will output exactly what they contain, no more, no less:

```
% cat trial
any text you like --
for as many
lines
and lines
and lines
as you like.
```

Or, if you want better-looking output (like for a printer), use the utility pr, which divides its output into pages, and outputs at the top of each page a heading containing the name of the file being output, and other information. It, too, takes one or more files as arguments:

```
% pr trial new
```

(pr also can do many things that this essay won't explore, as indeed can all of the utilities we will discuss from here on. If you're feeling adventuresome, sample the second section of this manual, which contains separate manual pages for all of the standard utilities that Idris provides. First read the page at the front called Conventions, then turn to the manual page for the program you're interested in. Like escargot or

squid, the typical manual page is hard to get a taste for, but exquisite once you've got it.)

Once you can get text into files and back out to a display or printer, the next item of interest is moving or copying files internally, which you can manage with similar aplomb by using three basic utilities: mv (for move), cp (for copy), and rm (for remove). Both mv and cp take two arguments, the name of the existing file you want to manipulate, and the name of the file you want to create. mv moves the original file to the new one, which amounts to renaming it. The original name is deleted. Hence the following would move new to perm:

```
% mv new perm
```

cp creates a second copy of the original file, and leaves the original intact. To make a copy of trial called transcript, you might type:

```
% cp trial transcript
```

The ls command would then reveal three files:

```
% ls
perm
transcript
trial
```

If a file already exists with the new name you give to cp or mv, it will be overwritten with the contents of the file being moved or copied; the already existing contents will be lost forever. To delete files on purpose, use rm, which will remove the files you give it as arguments. To rid yourself of transcript, for instance, type:

```
% rm transcript
```

Use this utility with care, as once a file has been removed, there's no recovering it.

The utilities we've explored so far fall mostly into two categories. A few are informational, like ls or who. The rest of them move files around in one way or another. Each of these last utilities has a source of input and a destination for its output. Some, like cp or mv, use files for both. Others, like cat and pr, take a file as input, and output to your terminal. (We'll see shortly that this distinction really isn't very important.) All of them access files without altering their contents.

FILTERING TEXT

Clearly, though, instead of merely transferring a file from one place to another, a program could make some presumably useful change to the input read before outputting it. In fact, Idris provides a number of utilities, called "filters", that do just that. Used separately, each filter reads text from a source of input -- usually the files named on its command line -- performs some operation on the input text (like sorting it), and out-

puts the result. Used together, filters permit suprisingly powerful text processing, without any need for additional programming.

Filters come in two kinds: some examine their input character by character, and change it according to what they find. A second group look at their input line by line, and alter it in the same way. The line-oriented filters called last and grep (honest) may be used to output only part of each of their input files, suppressing the remainder. To look at just the last three lines of trial, for instance (skipping all the preceding lines), you might type:

```
% last -3 trial
```

and get

```
and lines
and lines
as you like.
```

Be sure you understand that the contents of trial are unchanged. Filters never alter their input files; rather, they read text from the source of input named, change the text internally, then pass the altered text on to their output. The -3 is a flag consisting solely of a number that tells last how many lines it should output. last always outputs some group of lines at the end of each input file.

grep is more flexible: it searches for lines containing a specific string, and outputs only the lines that do. The string to be looked for is given as the first argument; the files to be searched are then listed afterwards. The following would output all the lines in trial containing the word you:

```
% grep you trial
any text you like --
as you like.
```

The string searched for need not be a single word; it may be any string of characters. For example:

```
% grep "and lines" trial
```

searches trial for the entire string enclosed in quotes, and outputs only the two lines that contain all of it (and not the one containing "lines" alone):

```
and lines
and lines
```

The quotation marks cause the shell to treat everything inside as a single argument, so that the space shown, which would normally separate two arguments, is simply included as part of the single argument inside the quotes. And of course, since it's the shell that interprets quotes, you can use them with any command you please, whenever you want to turn off the special meaning the shell would usually give a character, and cause it

to be counted as part of an ordinary argument.

Other filters change their input in subtler ways. An especially useful one is a utility called sort, which reads its source of input line by line, sorts the lines read, and then writes them to its output. The following would sort the lines in trial and output the result to your terminal:

```
% sort trial
and lines
and lines
any text you like --
as you like.
for as many
lines
```

As you can see, by default (that is, if you don't specify any flags), sort outputs the lines it reads in ascending order. The basis for the ordering is the internal value (ASCII value) of successive characters on each line; leaving details aside, this ordering works as you would expect. Letters sort in alphabetical order: 'a' sorts before 'b', and so on. Uppercase letters sort before lowercase, and digits sort before either.

With a few flags and more effort, you could as readily sort files based on only parts of each line (instead of the whole thing), or sort them by numerical value, or sort them into a different order. To sort together the lines in trial and perm, and output them in descending order, this suffices:

```
% sort -r trial perm
```

The -r flag causes sort to reverse its default ordering.

A second filter often used in combination with sort is uniq. uniq searches its input for repetitions of the same line, and deletes the duplicates, outputting only one copy of each input line. To output trial with its duplicate line deleted:

```
% uniq trial
any text you like --
for as many
lines
and lines
as you like.
```

Nice as it is, uniq has a limitation that might seem fairly serious: it recognizes only adjacent repetitions of a line. If the second occurrence of a line doesn't immediately follow the first one, it won't be seen as a duplicate (or be deleted). In the example just given, the duplicate lines happened to be adjacent; obviously, though, they wouldn't always be. For uniq to be more generally useful, its input should be sorted beforehand. Since identical lines will sort to the same place, duplicate lines will be adjacent in the sorted output, whatever their position had been in the original, unsorted file.

But how can we get the output of sort to be the input to uniq? One way is to tell sort to put its output into a file, instead of sending it to your terminal. Then the sorted file can be input to uniq:

```
% sort -o sorted trial
% uniq sorted
```

When sort sees the -o flag, it uses its next argument as the name of the file where it will place its output. (The -o flag is therefore said to include a value: it is always followed by a string that gets interpreted as an output filename.) Many of the filters we will examine accept a -o flag (and accompanying filename), so using the flag is a fairly adequate way of making one filter talk to another.

STDIN and STDOUT

The shell, however, provides a far more powerful one. We've seen already that a filter is just a command that copies text from its source of input to an output destination, performing some operation on the text as it's transferred. The shell generalizes this notion by assigning all filters (and all programs, for that matter) a standard source of input called STDIN, and a standard destination for output called STDOUT. By default, these are assigned to your terminal: input comes from your keyboard, and output goes to the terminal screen.

Some utilities, like cp or mv, do almost nothing with STDIN and STDOUT; instead, they deal with files explicitly named on their command line. Filters, however, can do a great deal indeed. Whenever you run a filter without naming any files on its command line, the filter will read its input from STDIN. What's more, unless you specify a -o flag, a filter writes its output to STDOUT. Look back at the first examples for sort and uniq -- you'll note that both command lines shown name an input file (and so don't read from STDIN), but don't contain a -o flag (and so do write to STDOUT).

If STDIN and STDOUT always designated your terminal, they wouldn't be worth much, except for the odd occasion when you felt like entering the input to a filter directly from your keyboard. Their power lies in their ability to be redirected (redefined) to designated files. By using a special argument on a command line, you can cause the standard input for a program to come from a file; similarly, you can direct its standard output to go to a file. For instance, these lines have the same effect as the last example shown:

```
% sort trial > sorted
% uniq < sorted
```

The '>' on the first line redirects the STDOUT of sort to be a file named sorted, which is then created as sort runs. The redirection happens before sort is started; it neither knows nor cares that it is no longer sending output to your terminal. The '<' on the second line redirects the STDIN of uniq to be the file sorted that the previous line just created.

Once again, uniq reads its STDIN without knowing whether it desginates a terminal or a file; either is equally acceptable. A third special symbol, ">>", causes the STDOUT of a program to be appended to a file. The previous contents of the file are preserved. To sort the lines in perm, and add them to the end of the file sorted, you could type:

```
% sort perm >> sorted
```

PIPELINES

Nifty, huh? Well, praise in departing. The shell provides an additional facility called a "pipeline" that makes even more effective use of STDIN and STDOUT. A pipeline causes two programs to be run simultaneously, with the STDOUT of one program redirected to be the STDIN of a second one. If both programs are filters, then this means that, while the first program is writing text to its standard output, the second is reading the same text from its standard input. Intermediate files like sorted become unnecessary, and the last example can be rewritten again:

```
% sort trial | uniq
```

The '|' specifies a pipeline. Specifically, it causes the STDOUT of the program to its left (here, sort) to be redirected as the STDIN of the program to its right (which is uniq). Data flows through the pipeline, from left to right, with sort providing the source of the data, and uniq merely altering it as lines of text pass through the pipe. (Pipelines, incidentally, may be used to link together more than two programs. Each stage in a longer pipeline works exactly like the first one: the STDOUT of the lefthand program is redirected to be the STDIN of the next one.)

TOOLMAKING

Using pipelines, utilities can be combined to perform what are normally complex tasks in processing text. Here's an extended example of how such an impromptu "tool" might be constructed. This one is built around a character-oriented filter called tr, which looks for a specified set of "target" characters in its input and either deletes them or translates them to characters in a second set. Suppose, for instance, we wanted to eliminate all punctuation from the file trial we made earlier, and leave ourselves with a uniform list of the words trial contains. A first shot at doing so might be:

```
% tr !a-z trial
```

The first argument to tr specifies the target characters it is to look for in its input; here, the string "a-z" names the characters from lowercase a through lowercase z, and the preceding '!' causes tr to look for any characters except the ones named. tr will then delete all occurrences of the target characters from its input, and output what's left:

```
anytextyoulikeforasmanylinesandlinesandlinesasyoulike%
```

Not quite what was wanted. Even the shell prompt ended up in the wrong place (any idea why?). A better approach would be to translate the punctuation characters to something more innocuous, like spaces, that would then neatly separate the words in the list. You can do so by giving tr a -t flag, naming the set of output characters to which the target characters are to be translated. After the -t, tr expects the next string on the command line to name the desired output characters. Since we want the output character to be a space, we have to enclose it in quotes; otherwise, tr would never see it, because spaces normally separate arguments on a command line, and are not passed as arguments themselves. All of which adds up to the following revised command line, and a better result:

```
% tr !a-z -t " " trial
any text you like for as many lines and lines and lines as you like %
```

As you can see, every string of characters in trial not containing a letter has been translated to a space. (Remember that each line of trial was ended by a newline, each of which was translated.)

tr fits in very well with sort and uniq. A common application is using tr to translate a text file into a list of words like the one just created, then using sort and uniq to rework the list into useful form. Creating a glossary -- a list of all the words occurring in a file -- is easy. Believe it or not, the following pipeline will output a sorted list of the words used in trial:

```
% tr !a-z -t "\n" trial | sort | uniq
and
any
as
for
like
lines
many
text
you
```

Let's examine it piece by piece. The tr command is almost the same as the last one we used, except that now we want to translate every string that is not part of a word into a newline, instead of a space. (That way, each word is passed to sort on a separate line.) Newlines, like spaces, can't be directly specified as arguments, since typing one will terminate the line you're trying to enter. So we use the characters "\n" to stand for a newline; tr will recognize what we mean.

It writes the list of words to STDOUT, which is redirected to be the STDIN of sort. sort reads the list from STDIN (because no input files are named on its command line), sorts it, and outputs it to its own STDOUT, which is redirected to be the STDIN of uniq. uniq also reads its input from STDIN, eliminates duplicate lines (i.e., multiple occurrences of the same word), and writes unique lines to STDOUT. Because the STDOUT of uniq hasn't been redirected, the final list of words is output to your terminal.

A problem with the glossary-maker as it stands is that it deals only with lowercase letters. We could make it handle uppercase, and ignore case distinctions, simply by putting another tr command at the start of the pipeline, this one to translate uppercase letters to lowercase:

```
% tr A-Z -t a-z trial | tr !a-z -t "\n" | sort | uniq
```

Note that this new tr command works slightly differently from the previous one. Because the output set of characters is now the same length as the target set, tr translates each occurrence of a target character into the corresponding character in the output set, instead of compressing strings of target characters into a single output character. And notice, too, that the second tr command in the pipe, since it no longer has any input files, reads its STDIN just as sort and uniq do. With this final change, you've produced a surprisingly useful tool, assembled on the fly from pre-existent parts. Making tools like this one, quickly and easily, is what Idris is all about.

You may have wondered by now whether programs other than the filters we've looked at can be used in a pipeline. The answer is yes -- nearly all utilities that normally output something to your terminal are really writing it to STDOUT, and most utilities that usually process text files will read STDIN if no files are given on their command lines. pr, for instance, can function as a filter, and might be used to gussy up the list of words produced in the last example. It will paginate text read from STDIN just as it would the text read from a file:

```
% tr A-Z -t a-z trial | tr !a-z -t "\n" | sort | uniq | pr
```

This pipeline is the longest you'll be seeing in this essay. But don't be overawed -- adding pr at the end of the pipe changes nothing at all in the rest of the command line.

Our introductory tour of the shell and the most basic standard utilities ends here; the capabilities of the programs you've seen, though, do not. Only a small minority of Idris utilities have been presented, and even these (as well as the shell) can do many things that this essay did not cover. Read on in this series of tutorials, and look at the detailed manual pages on the utilities to be found in Section II. You should by now be acquainted with the power and ease of use that Idris provides; getting to know it better will only increase your awareness of them.

NAME

Editing - the text editor e

SYNOPSIS

e -np <file>

FUNCTION

A text editor is a program used to create and edit a file of "text", that is, a file containing information entered by a user at a keyboard. Some common examples of text are documents, letters of correspondence, computer programs, lists of data, and so on.

e is used whenever you want to create a text file from scratch, or modify text that already exists. In either case, it will allow you to manipulate your material in a variety of ways, to transform time-consuming editing tasks into quick and efficient ones. For example, e allows you to:

1) add new lines to an old file of text,

2) delete lines you no longer want,

3) search for typing errors and correct them,

4) shift the position of various portions of text,

5) display what the file looks like at a given point in time, allowing you to verify changes,

6) make a permanent record of the file.

All of the text in this manual, and all of the programs that go with it, were typed in to this editor (or to some earlier version). It is a very powerful and important tool.

How does e handle all of these jobs? By nature, e is "interactive", meaning that it depends upon you, the user, to interact with it if anything is to be accomplished. This means that you must learn how to talk to e in order to get a reasonable response. You will get e's attention and cooperation if you use "editor commands" to tell it what you would like it to do. For each task there is a specific command entered by the user (these commands are given in detail in this essay); if the command is entered correctly, e will do its best to execute it for you. Once you know its language, you will find that dealing with e is not only practical, in terms of saving valuable work time, but also easy and enjoyable.

e is really nothing more than a "tool", so the more you use it, the more skill you acquire in handling it. This implies that the way to exploit e's potential is simply to make frequent and pragmatic use of it. This tutorial will introduce you to the commands, provide examples to illustrate their functions, and suggest exercises that will develop your skill in using them. We will begin with the basic utilities, and then move on to fancier usages of e. Feel free to experiment with the files you create in the test exercises -- since they are just practice files, you can't do any harm by playing with them. After completing the

tutorial, you will have a good understanding of how e is used; for further details and for the complete set of commands e offers, refer to the manual page e in Section II of this manual.

Since the editor is a program like any other program you might want to run, you'll first need to know how to get started on the computer and then, when you're ready to edit some text, how to enter into e. In response to the shell prompt "% ", you would type:

```
% e -np
```

followed, as always, by a newline (carriage return). The flags "-np" tell e to precede all lines of text with a line number, and to prompt for all input, much like the shell. More experienced users often prefer more silent operation, and so leave off the flags.

If you're starting from scratch, that is, if your file doesn't exist yet and you need to create it, then you may want to go ahead and give your file a name. Let's suppose you give your file the name new. Type:

```
% e -np new
```

Be sure to follow the e and the -np with spaces before typing in the name of the file you want to edit. The editor's response should be simply:

```
?
```

followed by its prompt '>' on the next line. This is the editor's way of saying that the file new doesn't yet exist and you need to add material to it. You can enter e with either of these commands; if you don't specify the filename now, you can do it at the end of the session.

Before going any further, it will be helpful to understand just how e handles text. e makes use of a temporary storage area, or workspace, known as a "buffer". The buffer is just an empty region made available when you run e. When you enter text it is written into the buffer; any further changes to the text are also made there. Then, when you're done, the final product can be copied from the buffer to a permanent place. From then on, whenever you use e to modify the file, e makes a copy of the file into its buffer. You'll see that working with a buffer is a comfortable way to experiment with your text without harming any previous work you may have done.

To get the editor's attention once you know what you want to do, you'll enter a specific command. e obeys commands that you give it if and only if you enter the command in a form it can understand. An editor command generally consists of a single character, typed in lowercase, followed (of course) by a newline. (From now on, we'll assume that you won't forget to complete every command or text line with a newline.)

APPENDING - ADDING TEXT

Your first job, then, when creating text from scratch, will be to use the a command, which allows you to "append" or add lines of text to the buffer. The procedure looks like this:

```
>a
1   (lines you want to enter)
2   .
```

Type the 'a' on a line by itself, and the editor will commence prompting by "line number". The prompt "1 " indicates that the next line you enter will be taken as text to be added as the first line of the buffer. The next line will have a "2 " prompt, and so on. When you've finished entering lines, type a '.' (period or dot) in response to the prompt. The '.' signals the editor that you have finished appending. The editor responds by reverting to the '>' command prompt; without the '.' the editor just keeps adding whatever lines you type to the buffer. If you want the file new to contain the beginning stanza of a poem, for instance, you might type:

```
>a
1   Tweedledum and Tweedledee
2   Agreed to have a battle;
3   For Tweedledum said Tweedledee
4   Had spooled his rice new rattle.
5   .
```

Now the buffer contains the four lines:

```
Tweedledum and Tweedledee
Agreed to have a battle;
For Tweedledum said Tweedledee
Had spooled his rice new rattle.
```

The line containing the 'a' and the one with '.' are not recorded, because e recognized that they were simply instructions and not part of the text. Similarly, the prompting line numbers are also not recorded with the text.

CHANGING TEXT

Notice that you made a mistake when typing line 4, and instead of "spoiled his nice" you actually typed "spooled his rice". You can correct these spelling errors with the c or "change" command. Type:

```
>4c
4   Had spoiled his nice new rattle.
5   .
```

That says: "Take line 4, and change whatever is currently written there to what I have just written here". Or, in other words,

Had spooled his rice new rattle.

is replaced by the new line you typed in with the c command. Again, the c and the '.' are not recorded. If you accidentally made a mistake in typing the command, the editor will let you know. If instead of typing "4c" you typed "c4", for example, e quickly throws a '?' at you. It's confused (can you blame it?) and is saying: "Sorry, please try entering that command again!" It will not go into long explanations of just how you botched things, but that's part of e's nature. A quick signal is all you get, but you'll probably see right away what you did wrong. Try again. Once you've changed line 4, you may be ready to add more material to your file; type another a command and e will place any new lines in the buffer.

WRITING TEXT TO A FILE

At this rate, you could keep appending and changing lines all day long, but chances are that you'll be interrupted and have to move onto some other project eventually. Or, perhaps you've added everything that you intended to add. In either case, you may need to save your text, so that it's available when you want to either look at it later on or make further changes to it. The w command "write" takes a picture of the buffer and copies its contents into the file new (the name you originally gave to e as your new filename, remember?). Type:

```
>w
```

or

```
>w new
```

to specify that you want your file to be named new. You'll need to supply the name if you didn't specify it earlier, when you first entered e. (Notice that the w command doesn't require a '.' at the end.)

The write command will destroy any previous contents of a file that you write to, so be careful in selecting filenames, and double check the filename you typed before terminating the line.

After you type your command, the editor prints out a number, such as:

```
115
```

that tells you how many characters e copied from the buffer into your file. If you were now to look at the contents of new, they would be identical to what's currently in the buffer. Since you're still in the editor, the buffer does not get erased; if you want to add more lines, keep on adding. The point to note is that the file itself changes only when you execute a w command; the buffer is simply a copy of what's in the file, and not the file itself. Whenever the buffer looks exactly like what you'd like to see in the file, finish with a w. As a matter of fact, it's a good idea to write out the contents onto a file frequently, in case you're distracted or the computer loses power or some other unforeseen

event occurs. That way, whatever work you've done up to that point will be saved (for later retrieval). As soon as you save your text with the w command, you're ready to end your editing session. You previously entered the editor with the e command, if you recall, and to leave the editor you must also issue a command -- the q command "quit". Type:

```
>q
```

and the prompt "% " appears on the screen. This is the official indication that you're back in the shell, no longer in the editor.

An interesting thing happens when you attempt to leave the editor before you've saved your text with w. Let's say you've typed in a business letter and you forgot all about w and typed q instead. Well, the editor knows that you've inserted text in the buffer and it doesn't want it to be lost unless you're sure you don't want it. So the q command, in this case, doesn't cause the editor to vanish but instead prints out:

```
are you sure?
```

meaning: "Are you sure you want to leave the editor without saving your text?" In other words, the prompt question reminds you that you haven't issued a w command. Just type anything other than 'Y' or 'y' in response then, on the next line type the w command. Now if you try to quit, the editor won't raise a ruckus but will quietly disappear.

If you respond with a 'Y' or 'y' to "are you sure? ", e assumes that you know what you are doing; the q command causes the editor to vanish in spite of the fact that your material isn't saved.

EXERCISE 1:

Try creating a file called abc and then writing it out. After entering e, add some text with a, then write it out with w. Now quit the editor. After the "% " appears, you can use the command

```
% cat abc
```

to see if the file really contains what you think it should. If it doesn't, try the sequence again.

ENTERING TEXT INTO THE BUFFER

Let's return to new, the file containing stanza 1 of the nursery rhyme. You're ready to enter the remaining stanza now, so if you're still in e, just type the e or "enter" command:

```
>e new
```

new is copied into the buffer and the system prints the number of characters it contains. This command (which is deliberately identical to the

name of the editor) clears the buffer before copying new into it, so if anything was stored there it is promptly deleted. Now you can append the last stanza.

Or, perhaps you discover that a co-worker has already typed the remainder of the poem into the file poem. Instead of retyping it yourself, you can read the file poem into the buffer, right after stanza 1, with the r or "read" command.

```
>r poem
```

A character count appears to indicate the total number of characters in this file; now the buffer contains:

```
Tweedledum and Tweedledee
Agreed to have a battle;
For Tweedledum said Tweedledee
Had spoiled his nice new rattle.

Just then flew down a monstrous crow,
As black as a tar-barrel;
Which frightened both the heroes so,
They quite forgot their quarrel!
```

To save the text, type a w and now the complete poem is contained in new. As you will see shortly, you can read in a file at any point in the buffer, not just at the end.

The e command can be used in two instances; first to enter the editor, then as a command within the editor when you want to enter another file into the buffer. For example, let's suppose that the two-stanza poem is still in the buffer. If we type:

```
>e inventory
```

then the file inventory is entered into the buffer, causing the current buffer contents to be deleted. Again, if you haven't saved the changes made to new, you would be warned at this point with "are you sure? " before the buffer is cleared.

Since you may enter or read more than one file into the buffer in a given editing session, you may need to know which file you're actually working on. You can easily find out by typing:

```
>f
```

The f command displays the currently "remembered" filename, in this case "inventory". (The "remembered" filename is the last one you entered with an e or r command.) If you want to modify inventory and then call it "inventory2", you could cause "inventory2" to become the remembered filename instead of "inventory" by typing:

```
>f inventory2
```

Now when you go to write out the file, e will remember that "inventory2" is the correct filename. Thus, the f command is used in two ways: to let you know which file is the currently remembered one, or to let you set the remembered filename to whatever name you choose.

PRINTING OUT TEXT

No doubt, you will frequently want to see what lines are contained in your text. The p command "print" will show you what you've got, by outputting the specified lines to your terminal. This is useful, especially when you're making changes frequently and can't remember exactly how a line reads. If new is the file you're editing, type p preceded by a line number to display a given line:

```
>1p
```

should display the contents of line 1. As a matter of fact, you can step through the file line by line by typing a newline; each newline will display the following line in the buffer. To display the entire poem, type:

```
>1,9p
```

This prints out the range of lines 1 through 9. Any range could be specified with the formula "m,np", where 'm' and 'n' are decimal numbers referring to specific lines in your text. The same results will be had with:

```
>1,$p
```

The '$' is a special character representing the last line in the buffer. So this command, like the previous one, displays all lines, first to last. This is the most convenient way to output the buffer's contents: by using the '$' you don't have to remember the actual number of lines present (a silly and often impossible task).

EXERCISE 2:

Create a second file following the procedure given in EXERCISE 1. Be sure to write it out. While this file is in the buffer, enter the original file created in EXERCISE 1, using the e command. Check with the f command to see which file is currently remembered. Next, use the r command to read the second file back into the buffer.

Display the entire buffer with the "1,$p" command to see if your r succeeded; you should see the contents of your original file immediately followed by the contents of the second.

ADDRESSING LINES

You've noticed by now that e keeps track of "lines" of text. When you access or display portions of the buffer, then, you are really addressing selected lines. e numbers lines, starting with line 1, as soon as you append material, and keeps adding line numbers as you enter lines of text. But these numbers are not recorded with the text; they're just a convenient way to talk about lines in the file in order to make changes.

In fact, most editor commands contain line addresses at the beginning to let e know which lines you want to alter in some way. Fortunately, the text editor is very proficient at helping you get around. First of all, it keeps track of the "current line" and "last line" of the buffer. The "current line" refers to the most recent line affected by a command that you entered; the symbol for "current line" is the character '.' (period or dot, hereinafter known as "dot"). And you'll remember that '$' represents the last line. You can type:

```
>.p
```

in order to display the current line; '.' is not always easy to predict, so you have to realize that different commands set '.' to different locations. As an example, when you enter a file, dot is set to the last line in the buffer. If line 7 needs correcting:

```
>7c
7   (line as it should read)
8   .
```

causes '.' to be set at line 7, the last line you altered in some way. Now type:

```
>.p
```

to print the current line, which indeed is the newly changed line 7. When you enter many of the commands, the editor will assume that you want the command to apply to the current line '.' if you don't specify any other line number. So simply typing:

```
>p
```

will do the same thing; it assumes you are referring to '.' since you didn't provide any other number. Whenever you lose track of '.' and want to know what line it's referring to, type:

```
>=
```

or:

```
>.=
```

and the line number will be displayed. Of course, you can also just print the current line and see what number precedes it. To output only the last line of the file, type:

```
>$p
```

'$' and '.', then, are methods of addressing specific lines which could also be addressed by specifying decimal numbers. As you get more familiar with e, however, you'll find yourself preferring them to numbers whenever possible.

There are other ways to move around, too. It has been mentioned that "m,n" will address a range of lines, as in:

```
>2,11p
```

which displays all lines beginning with line 2 and ending with line 11, inclusive. An address may also consist of more than one part, or "term". If two terms of an address are separated by a plus '+' or minus '-', then the addressed line is whatever line happens to be the sum or difference of the two terms. For instance,

```
>7+3c
```

means that line 10 is to be changed. Or:

```
>$-3p
```

displays the third line from the end of the buffer. If '+' or '-' precedes the first term, as in:

```
>+3p
```

then e assumes that '.' is meant to be another term, to which you want to add 3. This command is identical to:

```
>.+3p
```

Displayed is the third line beyond the current line. If you were to type a '+' or '-' by itself, you are actually referring to "+1" or "-1". So:

```
>++++p
```

means "move forward four lines from the current line and print it". Either '+' or '-' can also follow an address term, with the same effect:

```
>8---p
```

means "back up three lines from line 8 and print it"; you're now at line 5. All of these methods can be combined at will; choose whichever are most palatable to your editing tastes. Sequences frequently used, in addition to "1,$p" mentioned earlier, are

```
>.,$p
```

or:

```
>$-10,$p
```

The first outputs all lines from the current to the last line, while the second displays the last 11 lines of the buffer.

EXERCISE 3:

Experiment freely with line addressing, using the alternate methods described above. Note what happens if you try:

```
>5,7,12p
```

then try:

```
>7,4p
```

and see what you get. You may use 0 (zero) as a line address with some of the commands but not all -- it's a fictitious place you can put things after, but can't do things to. Typing:

```
>0r header
```

allows you to read the file called header at the start of the buffer; what was previously in the buffer now follows the text you have just read in, and is thus renumbered. Notice how the line numbers change.

Observe that

```
>0a
```

has a different effect than

```
>0c
```

e won't let you do the last one, because it calls for a change to the fictitious line 0.

INSERTING TEXT

Now that you've mastered moving around in the buffer, we can introduce more editor commands. The i command is used to insert text before a specified line. If the file contains the three lines:

```
Day 1:
Day 3:
Day 4:
```

and you wish to enter "Day 2:" into the list in its appropriate place, type:

```
>2i
2   Day 2:
3   .
```

The editor interprets the command to mean: "Go to line 2, and insert in front of line 2 the text given". Now the file contains 4 lines, with line 2 containing "Day 2". "Day 3:" has moved down to line 3. You can insert more than one line at a time, and when you've finished, you'll find that '.' is set to the last line inserted. Typing:

```
>1i
1   (lines to be inserted)
2   .
```

will insert a line at the start of the buffer. Note that i and a are very similar: each is typed as a line by itself, followed by text to be added, followed by a '.' alone on a line. There is, however, a fundamental difference -- you insert text before a given line, while you append after a given line.

DELETING TEXT

We've spoken of changing lines and inserting new ones, but what if you need to get rid of some altogether? This situation could arise when portions of your file are no longer needed and should be erased; or perhaps when you've got redundant material as a result of inserting new lines. You could use c to change the lines, then type nothing in their place (i.e., just a dot); but there is a more convenient way. The d command will "delete" or remove specified lines from the buffer. For example, if a file read:

```
Four score and seven years ago
Our fathers brought forth upon this continent
A new nation
A new nation, conceived in liberty,
```

Somehow in your typing, you duplicated the phrase "A new nation" (certainly Mr. Lincoln would not approve). You'll need to delete line 3, and

```
>3d
```

will do it. The current line is now

```
A new nation, conceived in liberty,
```

In other words, '.' refers to the line originally after the last line deleted. Line 4 moves up to become line 3. If you delete '$', however, the current line is set to the new '$':

```
>7,$d
```

will delete all lines from line 7 to the end of the buffer, and will put '.' at line 6. Be careful when using d, as deleted text is forever gone from the buffer.

You may recognize by now that the editor commands provide a range of options to the user in the creation or modification of text. For example, deleting and inserting text when done sequentially will produce nearly the same effect as changing the text with the c command. Or, to enter text from scratch you can use either the a or i command. The choice is yours, and no doubt you will develop preferences as you make use of e. But each command has its own modus operandi and you must learn its behavior. Note, for instance, what happens when you try

```
>2a
2   .
```

and do then the same procedure replacing a with i and then with c. List your file after each (with "1,$p") to see the results; they may not be what you'd expect!

EXERCISE 4:

With your practice files, insert new material and then delete portions of the text. Keep track of '.' to see how it moves around with different commands. Notice that you only use one line address with i, but with d you can specify a range.

MODIFYING TEXT - SUBSTITUTION

We've talked quite a bit about changing files by modifying whole lines at a time -- either appending, deleting, changing, inserting, or reading them. Quite often, however, you'll simply need to change one or two letters or words within a line. If you've misspelled a certain word or perhaps forgotten to insert a comma or period, you shouldn't have to retype the entire line -- correcting the single letter (or word) is much more efficient. The tool e uses for this task is the s or "substitute" command. The command works by substituting one string of characters on a line for another; its command line names both strings. For instance, if our line reads:

```
O, for a draught of vinage. . .
```

you'll need to correct "vinage" to read "vintage". The command to do so would be:

```
>s/vinage/vintage/p
```

and is interpreted like this: "Substitute for the letters "vinage" the letters "vintage". Another more general way of expressing it would be:

```
substitute for / this string / that string /
```

"that string" is also known as a "replacement string" as it replaces characters as they are on the line with the characters that should be on the line. If you don't specify line addresses, s assumes you want it to work on the current line. You may also specify a range of lines, as in:

```
>1,10s/this/that/p
```

This command examines each of the lines 1 through 10 and changes the phrase "this" to "that" whenever it occurs. (The p at the end is our familiar command which displays the last line changed -- this is one of the places when two commands can be written together.) However, the substitution only changes the first (leftmost) occurrence of "this" in a given line; if there is more than one occurrence, you need to add a g to execute the command "globally" on every occurrence of "this". Thus:

```
>1,$s/usa/USA/gp
```

will examine the entire file, changing usa from lower to upper case everywhere it finds it, and print the last line changed. (If no changes are made, you will get the old faithful "?" instead.) It's also possible to replace something with nothing:

```
>s/,//p
```

will delete the first comma from a line. The "//" in the example, a slash followed immediately by another slash, refers to a string containing nothing at all, not even a space. Spaces, in fact, are typical inhabitants of an s command line:

```
>s/thedog/the dog/p
```

Or, a quicker way to do the same thing, once you get cockier:

```
>s/ed/e d/p
```

It is a good idea to print the result of each substitution, particularly such a cocky one, since the wrong part of the line may match the pattern. Remember that it is the leftmost matching string that wins; and if there is more than one with the same starting position, then the longest string is chosen.

CONTEXT SEARCHING

Consider the following situation. You find out after typing in a rather lengthy text that you need to change the word "unicorn" to "dragon". The problem is remembering which lines contain "unicorn". No small task, in fact impossible. The solution lies in performing what's known as a "context search" on your file. This mysterious-sounding feat is a way of locating a particular line by searching for a specified string of characters appearing on that line. You execute the search by placing slashes around the string you want to nab:

```
>/unicorn/
```

Given this "line number", e will search forward in your file and arrive at the next line that contains "unicorn"; it will even print it out for you, since the default command is p. Now perform the s command:

```
>s/unicorn/dragon/p
```

A context search always starts at the line just after the current line and moves forward; if it arrives at the end of the buffer and still hasn't found the desired string, it "wraps around" to line 1 and keeps on looking. If it arrives at the line where you started and still hasn't found it, the standard complaint "?" indicates an unsuccessful search -- the string was not to be found.

Significant here is the understanding that /string/ determines a line number; it points to a particular line where that string occurs. Thus it can precede a command just like any other address:

```
>/unicorn/s//dragon/p
```

accomplishes in one command what the previous two steps did: look for the next line containing "unicorn", substitute in that line "dragon" for "unicorn", and print out the corrected line. The "//" refers to the string specified previously, so you needn't type it out again. e remembers the last string specified. It also remembers the last replacement string you entered. so if you want identical changes on more than one line, just type s to get your results (or "sgp" if you desire all occurrences changed, and the line printed out).

This remembered string comes in handy on another occasion, too. If there is more than one line containing "unicorn" but you don't want to change that word to "dragon" in every instance, you'll need to look at each line and then decide if you want to change it. Once you've specified your string and e finds the first occurrence of it, just type:

```
>//
```

to arrive at the second line containing an occurrence of unicorn and to print the line. Repeat this to locate the third line containing an occurrence, and so on. If you use them all up, by changing each "unicorn" to "dragon", for instance, the context search eventually finishes with a "?".

Finally, you may also specify a range of lines using /string/. If you want to display a paragraph whose first line begins with "START" and whose last line contains "END",

```
>/START/,/END/p
```

will work fine.

You can even search backwards, if you need to. Instead of "/string/" you'll use "?string?" or "%string%" and the search will start at the line preceding the current line, move back to the next nearest occurrence of

"string", and display the line. If no match has been found before reaching the start of the buffer, the search wraps around to the last line, $, and keeps searching. Again, if no match is found, the error message "?" appears.

Note that "??" serves the same function in a backwards search as "//" in the forward search -- it searches backwards for the last expression specified, saving you from typing it again. Search strings are highly useful items.

GLOBAL SEARCHING

Another command making use of search strings is the g command -- g searches the buffer "globally" for all lines matching the string you specify, and then executes another command for each of those lines:

```
>g/INTERIM/d
```

will search for every line containing "INTERIM" and then delete it. If you then list the buffer, all of those lines will be gone.

If you want to locate every line in your file that contains an occurrence of the word "company", and then change all occurrences of "company" to "corporation",

```
>g/company/s//corporation/gp
```

will make the change and print all the corrected lines. You can follow a g command with any number of commands, separated by semicolons ';'. (The rule about one command per line was a white lie.)

```
>g/micro/s//macro/;.+3s/bits/bytes/;.-6sg;.,$p
```

The v command acts as the converse of g; it will search for any lines not matching the specified string, and perform commands on those lines. Thus:

```
>v/./p
```

prints out every line not containing a period.

EXERCISE 5:

Practice the s and g commands using simple examples to start. Try substituting '?' for '.' in each line of your file:

```
>1,$s/./?/
```

Did you forget to get every occurrence of '.' within every line? Do the same command, only add "gp" at the end. This same task can be done with g, and it would look like this:

```
>g/./s//?/gp
```

The first g assures that the command searches every line in the buffer; the last g sees that it makes the substitution on every occurrence of the string in each line that fits the match. The difference is that the second version prints every line changed, while the first one prints only the last line changed.

REGULAR EXPRESSIONS

It's now a convenient time to introduce a new concept -- the notion of a "regular expression". Simply speaking, a regular expression is a shorthand notation for a sequence of characters contained in a line. In other words, the regular expression is a (possibly) abbreviated representation of certain characters you want to pinpoint. The characters are said to "match" the regular expression. Regular expressions are used most frequently in commands calling for searches or substitutions, because they provide a method for specifying character strings.

The most obvious regular expression is one you're already familiar with: a character string which is an identical match to a string present in a file line. Remember the context searches?

```
>/xyz/
```

implies a search for a line containing the same string "xyz". So, any ordinary character can be considered a "regular expression" which matches that very character when it occurs on a line.

There are other regular expressions which are not ordinary at all; in fact, they are somewhat magical and are regarded as "special" by e. One of them is '?'. Instead of specifying particular characters, you can point to "any" character at all with '?'. For example:

```
>s/?/M/
```

will search until it arrives at any character on the line (including a space) and will replace it with 'M'. In essence, it simply replaces the first character. Or:

```
>/t?e/
```

will match either "the", "tie", "toe", "tee", "t5e", "t!e", or any other three-character string that begins with 't' and ends with 'e'. It simply uses the first match it finds.

A '^' (caret) following a character matches zero or more occurrences of that character; that is, if your line reads:

```
baaaa!
```

then

```
>/aa^/
```

would match the entire string of 'a's. The logic goes like this: "Find an 'a', and then see if it's followed by another 'a', and another 'a', etc." So /aa^/ would have matched either one 'a' or thirty 'a's, if that many were present in the string.

A '*' (asterisk) is a bit more ambitious: it matches anything at all! That means, if you type:

```
>/m*n/
```

the match will be any string that starts with 'm' and ends in 'n'. Try replacing "jolly" with "jovial" in the following line:

```
Jupiter greeted the jolly, joyous Juno.

>s/j*y/jovial/p
```

will produce:

```
Jupiter greeted the jovialous Juno.
```

It didn't work as desired, because the '*' will match the longest possible string. The first 'j' it encountered was in "jolly" but the last 'y' was in "joyous", so it chose the longer match. The correct version would have read:

```
>s/j*ly/jovial/p
```

To delete everything on a line:

```
>s/*//
```

that is, substitute for everything, nothing. To replace all contents up to (and including) a ':', type:

```
>s/*:/replacement/
```

The special character '^' mentioned earlier has another important usage. When it is the leftmost character of the regular expression, it points to the beginning of the line. If you specify:

```
>/^TUT/
```

then only if "TUT" are the first characters in the line will the match occur. If our line reads:

```
KING TUT
```

then the search attempt won't find the match. Similarly, '$' signifies the end of a line. For example:

```
>/TUT$/
```

will match the string "TUT" only if it occurs at the end of the line. Thus:

```
>/^Hurrah$/
```

locates a line whose entire contents are "Hurrah".

If we use brackets '[' and ']' to enclose characters, that too has a special effect. For instance:

```
>/[abcdef]/
```

will match any one of the characters in the bracketed list, that is, an 'a' or a 'b', etc. through 'f'. Another way to express this is:

```
>/[a-f]/
```

On the line "7 days remain to teach 12 women 5 ways to dance.", the command:

```
>s/[0-9]/N/p
```

will change '7' to 'N'. '7' was the first character in the list that was found. (If you do the same substitution again, which character changes?)

A tricky inversion of this is:

```
>s/[!a-zA-Z]/?/p
```

means locate the leftmost character that is not a letter, change it to '?', then print the line.

The '!' before the list of characters means: "Match any character except one in the bracketed list." Thus the match could find a numeric, a punctuation character, or any other character not in the class [a-zA-Z].

These regular expressions are all valuable editing tools, but they can cause problems in your substitute commands. Let's suppose the line you need to change has some of these magical creatures on it:

```
ZOUNDS!*?^$*!
```

You want to replace the '*' with a ':', and your first attempt seems reasonable:

```
>s/*/:/p
```

Reasonable, yes, but magic characters don't respond to reason. As you can see, the contents of the entire line have been replaced by a ':'. e has given '*' special powers -- to match a string of any length -- and now you're asking it to regard '*' as an ordinary character. The solution is to insert another magical character, the backslash '\', in front of the

special character. This takes away the magical meaning and forces e to treat '*' like any other character. Thus:

```
>s/\*/:/p
```

will work. Since '\' is a special character too, you must apply the same rule when deleting a '\' from a line; precede the '\' with yet another '\':

```
>s/\\//p
```

The backslash can also be used to turn on magical properties of otherwise tame characters:

\b is a visible way of typing a backspace,

\t becomes a horizontal tab,

\r becomes a carriage return,

\v becomes a vertical tab,

\f becomes a formfeed, and

\n is the only way to talk about newlines inside character strings, since a literal newline is invariably taken as the end of the command line.

Finally, you can represent even more exotic unprintable characters by writing up to three digits after a backslash -- a practice you rarely need to engage in, but it's provided for. If you've typed a tab key to skip four spaces between columns, your line resembles:

```
1981    COST    ASSETS
```

To remove the spaces and replace them with '+', type

```
>s/\t/+/gp
```

and you get:

```
1981+COST+ASSETS
```

Warning: when making substitutions on lines that contain lots of special characters, the procedure can become quite complicated. If things get too messy, (if for instance you find yourself trying unsuccessfully to edit a string of backslashes!) just give up and start the command over. In extreme cases, you may want to abandon the s command and retype the line from scratch, using c instead.

FANCY TRICKS

We've now illustrated most of e's capabilities, but a few more fancy tricks remain which deserve brief mention. One is the '&' (ampersand) character. This becomes special only when it's used on the right-hand side of a substitute command, where it is transformed into the character string which the left-hand side matched. If we want to put the word "perhaps" in parentheses, for instance, we could type it out twice, surrounding it by parentheses in the replacement string:

```
>s/perhaps/(perhaps)/p
```

But the easier solution is:

```
>s/perhaps/(&)/p
```

which substitutes for the string "perhaps" the same string, surrounded by parentheses. You can even repeat the match:

```
>s/very /&&/p
```

which turns, for example, "very big" into "very very big". (Note the careful inclusion of a trailing space.)

If you have an unfinished phrase you want to complete, the ampersand again saves typing:

```
>s/grand,/& I thought to myself,/
```

which might result in:

```
She looks grand, I thought to myself, due to her recent success.
```

If you want to break lines or perhaps join them, use the regular expression "\n" mentioned above, which represents a newline. To break the line:

```
The computer will be unavailable; my apologies!
```

try:

```
>s/; /;\n/p
```

(Again note the space.) You'll now have the two lines:

```
The computer will be unavailable;
my apologies!
```

Or, to join a line with the next, just delete the newline:

```
s/\n//p
```

i.e., substitute for the newline, nothing; the second line is moved up to the current line. (Note: you will not actually see the lines joined as one, unless you print them both at once, but when you write out your file,

they will be merged.)

e provides a fancy method for keeping track of certain lines of text in the buffer; you can give some lines a "tag" (any lowercase letter) which is used to address the line. The command is called k, or "know" the addressed line to be 'x' (the lowercase letter you choose). For example, if we want the line

```
WARNING: HIGH VOLTAGE
```

to be tagged with the letter 'v', type:

```
>/WARNING/kv
```

This looks for the line we want, giving it the tag 'v'. To address the line from now on, type "'v" (no need to worry about its location in the buffer; e remembers it as "'v"). This can be useful when you're doing lots of modifying, (i.e., '.' will be changing frequently) but also want to keep a certain line readily accessible.

Another tagging mechanism can be used with regular expressions in the s command by surrounding the regular expression with "\(" and "\)". The characters matched by the expression can then be included in the replacement string under the tag name "\(#" where # is any digit from 1 to 9; the numbers correspond to the order of the expressions in the first part of the command. An example will elucidate matters. If we need to reverse the order of the words in the line "boy=girl",

```
>s/\(*\)=\(*\)/\(2=\(1/p
```

prints out:

```
girl=boy
```

It divides the line into two regular expressions, tags them with "\(" and "\)", then reverses the second one with the first one. Such "tagged" regular expressions are hard to type correctly, and seldom used, but very helpful when you need to do some messy edit repeatedly.

The m or "move" command is used to shuffle portions of text around from one location to another. Basically, you move a set of lines from where they currently reside to immediately after another line. If we want to move the last ten lines of the file to the beginning, say:

```
>$-10,$m0
```

In general terms, the command goes like this:

```
(range of lines) move (to after this line)
```

```
>10,'vm3
```

will move all lines from 10 to 'v (our WARNING: HIGH VOLTAGE line) to after line 3. '.' is set to the last line moved. This command also takes

a little practice to master, but is definitely worth the investment.

The t command is similar to m; it makes a "twin" of a set of lines and places the duplicate lines immediately after a specified line. If lines 1 to 3 contain:

```
This software is
a product of
Whitesmiths, Ltd.
```

and we want to repeat that message at the end of our document, type:

```
>1,3t$
```

Note that both t and m change the positions of lines, but t makes a copy of them, while m moves the original lines themselves.

We spoke earlier of e as a significant tool which will do editing jobs (of all sorts) quickly and thoroughly. A further extension of its capability lies in the creation of "scripts", i.e. canned sequences of editing commands stored in files. Suppose you want to perform the same editing task on a number of files. Here, for instance, are the commands needed to discard trailing spaces and to fold long lines at the end of a word:

```
>g/  ^$/s///
>g/\(????????????????????????????????????????[! ]^\)  ^/s//\(1\n/g
>w
```

The first command looks for one or more spaces at the end of a line, then deletes them. The second one looks for a long line and inserts newlines in place of spaces at convenient intervals (honest). The third command writes the file back out. Now this is a hard sequence to type correctly even once, let alone repeatedly, so the best thing to do is use the editor to prepare its own input! If you type this in as ordinary text and write it to a file called, say, fixup, you can then launder a file called, say, needsit, by running the editor:

```
% e needsit <fixup
```

and the job is done. Not only that, it can be done over and over again with a minimum of typing -- you have made a useful tool.

There are many more things that the editor can do, and much more that you can do with just the features introduced here. Read the manual page for e to see what's there, then practice what you have learned. Soon, you will develop a style of editing which best suits you; the more advanced features you can refer back to only when you need them.

Good luck!

NAME

Files - reading and writing data

FUNCTION

So far, little has been said directly about the nature of files under Idris. This was intentional, in part because it is important to understand how to interact with the shell and the editor to facilitate exploring the properties of files; but it also illustrates that you can do quite a lot of work without knowing much about the underlying machinery of reading and writing.

Most of the examples have, in fact, been in terms of keyboard input and display output to a terminal, hardly the sort of thing one associates with the term "file". And yet the shell makes it easy for a program to read a file instead of keyboard input, or to divert output to a file instead of to the display, so there must be some important way in which terminal I/O (input and output) can be made to look identical to file I/O.

The common ground is the "text stream".

If a program expects to read its input just once, sequentially from beginning to end, then it hardly matters whether the data is typed in on the fly or copied out of a file; if the program has to browse around in a data base, that's a different story. And if a program produces its output in one sequential pass, from beginning to end, then it hardly matters whether the data is directed straight to the display or copied into a file for later display; if the program must back and fill to generate the output, that too is a different story. As it turns out, a large number of useful programs do perform such stream or sequential I/O, and hence are "filters" equally usable with files or with people sitting at terminals.

Another consideration is whether a program deals with the kind of characters that can be easily typed at a keyboard, and meaningfully written to a display, or whether it produces all the possible binary codes that can be represented in a single "byte" or character position. Sometimes a program must communicate with other programs with such binary codes, that are hard for terminals to represent or for people to understand, but once again this is remarkably rare. (Perhaps it is more honest to say that the Idris system makes a point of avoiding binary I/O.) At any rate, human consumable data is called "text", and the text stream is the lingua franca of the Idris world.

To further standardize the representation of text streams, Idris always deals with the American Standard Code for Information Interchange, known as ASCII, so that a lowercase 'a' is recognized as such by any program that cares. This means that character codes may be mapped on the way in, and unmapped on the way out, for non-ASCII I/O devices, or "peripherals". Even within the ASCII community, some folks end a line with a carriage return followed by a linefeed, some with a linefeed followed by a carriage return, some with just a carriage return, and some with just a linefeed. In the last case, the linefeed is usually called by its alias "newline"; and this is the case that Idris has standardized on. All text streams, and all text files, consist of zero or more "lines", each terminated by a single newline.

There are a few additional rules about how long lines can be and what can be in them, but these are less important. See the manual pages called ASCII and text in Section III of this manual for more detailed information.

The important thing to remember is that I/O to terminals, other peripherals, and files can usually be made to look the same if it is in terms of text streams. The shell underscores the importance of this by providing each program a standard input stream STDIN, a standard output stream STDOUT, and a standard error reporting stream STDERR; and it makes STDIN and STDOUT dead easy to redirect from the terminal to files or other peripherals. Often, these streams may be arbitrary binary codes and need not be text, but text is always acceptable. This is why pipelines and file redirection are so powerful.

FILESYSTEMS

It is not enough to be able to read and write terminals, however, nor is it enough to be able to read and write all the peripherals on a given computer as if they were interchangeable data streams. The Idris system proper needs at least a hundred different files to hold all its programs and working data; assigning one file to each peripheral would be unthinkable. A disk drive can usually hold the contents of many different files, if only there be some way to administer the available storage space. The way Idris does it is to impose a structure on disk storage (or on other high speed, random access storage) called a "filesystem"; if the disk is large enough, it may even be chopped up into more than one "logical device", each of which can have a filesystem structure imposed on it.

The System Administrator is responsible for laying out filesystems. What matters for now is that an Idris system can have one or more filesystems available or "on line" at any given time, that each filesystem can support one or more "directories" or collections of files, and that each file within a filesystem is recorded with a number of useful attributes. Among the more important attributes of a file are its contents (of course), its size in bytes, and the name (or names) by which it can be accessed.

And the way to find out about the various attributes of files is with the standard listing utility ls. You have already used ls without flags to determine the names of files you have created:

```
% ls
abc
new
perm
poem
sorted
trial
```

Now try the long form listing with -l:

```
% ls -l
total    6
-rw-rw-rw-  1 pjp          36 Oct 16 16:27 abc
-rw-rw-rw-  1 pjp         250 Oct 16 16:25 new
-rw-rw-rw-  1 pjp         104 Oct 16 16:25 perm
-rw-rw-rw-  1 msk         135 Oct 15 09:27 poem
-rw-rw-rw-  1 pjp         176 Oct 16 16:27 sorted
-rw-rw-rw-  1 pjp          72 Oct 16 16:27 trial
```

What a surfeit of information! Over on the right are the filenames, as before; but now to the left is the date and time the file was last modified, and to the left of that is the number of bytes in the file. To the left of that is information concerning who owns the file, how many different names it has, and what access permissions are recorded for the file -- these attributes will be covered later. There is lots of whitespace in the middle to leave room for owner loginids of up to eight characters and file lengths of up to 16 million bytes.

DIRECTORIES

But where are all the other files? Somewhere must be stored files with names like cat, echo, ls, and sh, to hold the various standard utilities, not to mention the rest of the hundred-odd Idris files mentioned above. The answer is, they're in other directories. Unless told to do otherwise, ls confines itself to listing files in the "current directory", which is typically your personal collection of files. At login time, you were placed in this personal directory (i.e. it was made the current directory) before your shell was given control. So far, that has been the only part of the filesystem you've known how to access directly.

Often it is enough. The usual convention under Idris is to give each user a personal directory, to which other people might even be denied access if security warrants it, where each is free to think up filenames without fear of collision with others. Only when you must cooperate with others in a common directory, or when your collection of files becomes large enough to require some internal structure of its own, do you have to worry about the directory structure of a filesystem.

To see where you live in the directory structure of your system, use the pwd utility (to "print working directory"):

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

The response will be some funny sequence of slashes and terse names like the one shown. This says that the current, or working, directory is called pjp (which is often the same as your loginid, but need not be), and that pjp is a subdirectory within the parent directory called usr, and that usr is a subdirectory within the fundamental or "root" directory of the entire system. The list of slashes and names /usr/pjp is called a "pathname", because it traces a path from a known place (the root in this case) to a given filesystem entity (the working directory pjp in this case).

Thus, all files within the system are organized into a "tree", or branching hierarchy, because any one can be reached by tracing a path from the root (of the tree) to the appropriate entity; directories are nodes, or branching points along the way. So the files you have been working with have pathnames like /usr/pjp/poem and /usr/pjp/abc, even though you have known them by their family names, shorn of slashes.

By keeping track of your current working directory, Idris permits you this convenient shorthand. The rule is: a pathname that begins with a slash is called an "absolute pathname" and always traces a path from the root; whereas any other pathname traces a path from the current directory. So poem and abc are acceptable ways of naming files if you are working in the directory /usr/pjp, and the absolute pathnames /usr/pjp/poem and /usr/pjp/abc are acceptable and have the same meaning anywhere.

Any "link" in the pathname can be up to fourteen characters long (additional characters are simply ignored), and may consist of any characters except a slash (of course) or an ASCII NUL (which has the value zero and is fortunately hard to type). Convention imposes much stricter rules, however. Anything other than printable characters within pathnames can be difficult if not impossible to get past the shell to various utilities. It is rarely possible to abbreviate directory names, so long ones quickly become tedious to type. And it is foolish to use names obscure in meaning or hard to type, such as "0001I10". A later essay in this section describes some shorthand that the shell supports for filenames; this often suggests a useful discipline for naming groups of files within a directory.

You can make a "subdirectory" of your very own by using the mkdir utility:

```
% mkdir poems
```

This creates an initially empty directory called poems (or /usr/pjp/poems). To see what you now have:

```
% ls -l
total    7
-rw-rw-rw-  1 pjp          36 Oct 16 16:27 abc
-rw-rw-rw-  1 pjp         250 Oct 16 16:25 new
-rw-rw-rw-  1 pjp         104 Oct 16 16:25 perm
-rw-rw-rw-  1 msk         135 Oct 15 09:27 poem
drwxrwxrwx  2 pjp          32 Oct 16 16:30 poems
-rw-rw-rw-  1 pjp         176 Oct 16 16:27 sorted
-rw-rw-rw-  1 pjp          72 Oct 16 16:27 trial
```

Note that poems has somewhat different permissions than the other files, including a leading 'd' to indicate a directory, but it is in many ways a file just like all the others.

What can you do with the new subdirectory? Well, you can copy files into it:

```
% cp poem poems/poem
% cp new poems/new
```

and so on. Now try

```
% ls -l poems
```

and see what you get. You can also make poems the current working directory by using cd:

```
% cd poems
% pwd
/usr/pjp/poems
% ls
new
poem
```

LINKS

You can get back to your original directory one of three ways: 1) cd with no argument usually returns you to your "home" directory, the one you were put in after login; 2) cd with the absolute pathname /usr/pjp will get you there from anywhere; or 3) cd with the relative pathname .. will climb up one level of the directory hierarchy, because .. is always a local alias or "link" pointing at the parent directory. The .. was put there quietly by mkdir, when it made the directory poems, along with another link called, simply, . (dot or period) which is an alias for the directory poems itself. Neither of these names is listed, nor any other links that begin with a dot, unless you give ls the -a flag:

```
% ls -al
drwxrwxrwx  2 pjp          64 Oct 16 19:50 .

total    2
drwxrwxrwx  2 pjp          64 Oct 16 19:50 .
drwxrwxrwx  3 pjp         144 Oct 16 17:25 ..
-rw-rw-rw-  1 pjp         250 Oct 16 19:50 new
-rw-rw-rw-  1 pjp         135 Oct 16 19:50 poem
```

The links . and .. are placed in every directory as a matter of convenience and for uniformity. The alias . is useful whenever you want to talk about the current directory and don't want to have to specify its absolute path name. It is the default argument to ls, for instance, as seen more clearly in the last example. And .. is a useful back link for relative navigation, as with the cd above. You could also refer to /usr/pjp/trial, from /usr/pjp/poems, as ../trial (got that?). These are, in fact, the only loops permitted in the pure tree structure of the Idris filesystem, so that utilities which climb over directory subtrees know how to stay out of trouble.

To be more explicit about the underlying machinery: All of the information about a file is stored in a central place (called an "inode" for bizarre reasons). Each directory entry merely provides a named link to

this central place, so there can be any number of pathnames, or aliases, or links leading to the same file. The number just to the left of the owner loginid counts the number of links to a file. Note that /usr/pjp/poems has two links, the second one being /usr/pjp/poems/., of course.

You already know how to erase a link; the remove utility rm does the job. The file actually disappears only when the last link to it is erased. And there is a utility for explicitly creating aliases, called ln for link. Suppose, for instance, you didn't want two copies of poem, but merely two ways of referring to the same file:

```
% rm poem
% ln ../poem poem
```

Try this, then use ls to inspect the results. Perhaps by now you have guessed that the move utility mv merely performs a link, a la ln, to establish the new name, then removes or "unlinks" the old name.

PERMISSIONS

The only remaining unexplored field in the long format ls listing is the cryptic "-rw-rw-rw-" and its ilk. These are highly abbreviated notes on what various people are permitted to do with and to a given file. There are three kinds of access permission: 1) read permission 'r' means that the file or directory may be read, 2) write permission 'w' means that the file may be written, or that the directory may have links added or deleted within it, and 3) execute permission 'x' means that the file may be executed (as a utility or other program), or that the directory may be scanned in the process of tracing a pathname. There are also three classes of people: 1) the owner is the user whose loginid matches that displayed by ls with the -l flag, 2) the group is the user or users whose groupid matches that displayed by ls with the -lg flag, or 3) everyone else. The System Administrator can tell you what group you are in.

Any program that attempts to access a file first is put in one of the three classes above, then has its permissions checked against the appropriate rwx group displayed (reading left to right for owner, group, and everyone else). No permissions, no access. As you can see, most files are created with read/write access for all concerned, and directories are fully accessible. In a tighter environment, your home directory might deny scan permission to anyone but yourself, so that anyone outside your directory subtree can never get inside it to access your files, regardless of their individual permissions.

You can add or delete permissions, for files that you own, by using the chmod (for "change mode") utility. To deny write permission to anyone but yourself for the file poem:

```
% chmod -wgo poem
```

Try this, followed by ls to see the result once again. And read the manual page in Section II for chmod, to see what all those flags mean. It

is also possible to change the owner of a file, but this is a function reserved to the System Administrator, for various obvious and subtle reasons.

There are two or three more permissions not described here, called set userid, set groupid, and save text, which are generally of interest only to System Administrators setting up special utilities. They are mentioned in passing on the manual page for chmod in Section II; at least one of them is clever enough to have been patented; but they are beyond the scope of this essay.

There are no explicit limits on how large any one file can get, short of the 16 million byte maximum size, nor are there any per user limits imposed on how much disk space you can use. A file must, however, fit entirely within one filesystem, which could be quite small if written on a diskette, or it could be as large as 32 million bytes. What matters, when space is at a premium anyway, is that disk space within a filesystem is handed out in 512-byte chunks, called "blocks". If you write 500 one-byte files, you are going to consume 500 blocks (more actually, because you need space to represent the large directory as well).

The "total" line at the head of each long format ls listing is a guide to the space conscious user. It sums the size, in blocks, of all the files listed (overestimating if there are multiple links, by the way). To find out how much space is left in the filesystem you are using, try the "disk free" utility df:

```
% df
/dev/rm3:
free blocks: 6318 / 16320
```

This says that there 6318 blocks available, in a filesystem that occupies a total of 16320 disk blocks. It also, by the way, tells you the physical device name of the disk that the filesystem is written on -- a piece of information the innocent user seldom needs to know, nor care about. As mentioned before, there can be many filesystems made available within the same hierarchy, all with different sizes and different amounts of free space. Aside from a few firewalls (such as files and links not bridging filesystems), you are seldom reminded of these physical boundaries.

PHYSICAL DEVICES

When you do have to be aware of the underlying peripherals, however, Idris provides a nice mechanism for accessing them. Certain files are called "block special" or "character special", because they cause actual I/O devices to twitch when read or written. They bridge between the world of named files and the world of physical I/O.

A block special device is one that can be treated as an ordered sequence of 512-byte blocks, whose blocks can be read and written in any order. This is also known as "random access" or "direct access". Disks usually can be made to look like this, even if their natural block size is smaller or larger. Tapes can be also, but usually to a lesser extent; it may be

possible to read a tape as a block special device, but write it only sequentially. Filesystems must be written on block special devices before they can be used directly by Idris.

A character special device is typically a "tty" or terminal (terminal port, actually). If so, it has various magical properties such as the ability to control process execution (DEL key and ctl-\, for instance), and facilities for editing input and throttling output. Other character special devices are things like line printers, serial tape drives, or bizarre hardware too idiosyncratic to be made to look like a block special device or a tty. Often the same device will have both block special and character special interfaces, for a variety of esoteric control reasons.

There are also a handful of character special files supported directly by the Idris resident. These do things like: make available to ps, the "process status" utility, information on the programs that are running; provide a place for discarding arbitrary amounts of output (/dev/null); and permit the System Administrator to peer inside the resident while it is running, on the rash assumption that this is a good idea.

You can spot a special file in a long form ls listing, because the first character of the access permissions is neither a '-' for a plain file, nor a 'd' for a directory; instead it is a 'b' for block special, or a 'c' for character special. Only the System Administrator has the power to make special files; and generally they are confined to the directory /dev. Try

```
% ls -l /dev
```

to see what sort of devices are on your system. Compare this with the listing you get from who to see what is done with ownership and permissions of the active tty files.

The best thing about special devices, as was stated at the outset of this essay, is that you really don't have to know much about them to make meaningful use of Idris. Nor, for that matter, do you have to worry much about links, inodes, or filesystems.

You do have to mess with access permissions, from time to time; they offer considerable flexibility in setting up protected systems. And you should be familiar with directory hierarchies, pathnames, and relative navigation among nearby directories. The best way to learn is to look around, as this essay has encouraged you to do. Read the manual page for ls in Section II; it is a powerful probe.

You can look on the Idris filesystem as a sophisticated way to organize large quantities of information. Or you can take advantage of its ability to shield you from most of the complexity needed to support a multi-user system. Take your pick.

NAME

Advanced - processing and programming with the shell

FUNCTION

By now, you should know your way around the editor, and around an Idris filesystem. And you've seen enough of the shell to use it more or less like the "command interpreter" or "executive" of many an operating system. Here, you'll discover how much more (and how much less) the shell really is. It is, first of all, a mechanism for precisely manipulating instances of other programs (called processes); but given a few interface conventions and several builtin facilities of its own, the shell can also serve as a programming language of surprising power and utility.

PROCESSES

A "process" is simply a running instance of a program. Whenever you type the name of a utility to the shell, it starts a process to execute the utility. The process runs until terminated, usually by the exit of the utility. Typing a pipeline to the shell results in the startup of one process for each program in the pipe; the processes are run simultaneously, with their input and output redirected as needed.

A given process may start and control an arbitrary number of other processes, called its "children", which may in turn have children of their own, to an extent limited only by the internal table space of the operating system. Each user in an Idris environment thus has his own tree of processes, at the top of which is usually a process running the shell (though any program at all could be run there).

In all the command lines examined so far, the shell started up a process for a single utility, or a set of processes for a pipeline, waited for the processes to terminate, then prompted you for another command line. You can also make the shell execute a sequence of commands before prompting again, by separating the commands with a ';', and giving them on the same line of input. For instance, to change to a directory and immediately examine its contents, you might type:

```
% cd /final/doc; ls -l
```

In response, the shell would change to the directory, then execute the ls utility, and would prompt again only when ls terminated.

The shell can also be made to prompt immediately after starting up a process, instead of waiting around for it to terminate. (Try it with a long ls.) In this way, long or tedious tasks can be run "in the background" simultaneously with whatever you're doing at your terminal, freeing you (and the terminal) for other things. Processes running in the background still can type to your terminal, so interactive programs should usually be run this way only if their STDIN and STDOUT are redirected. (Otherwise, they will get an empty file for input, and will send their output to your terminal -- a confusing situation.) The following would run e in the background, taking commands from the file fixup, editing the file needsit, and sending its output to the file trace:

```
% e needsit <fixup >trace &
```

When you initiate a command with '&', the shell outputs a number called the processid, which uniquely identifies the process just started to execute the command. If the command involved more than one process, then a separate processid is output for each process started. Also, '&', like ';', can be used as a command separator, which causes the shell to start the left-hand command, then immediately start the right-hand command.

In order to kill a background process, rather than wait for its natural death, you can give its processid to the kill utility. The following would terminate the process with processid 317:

```
% kill 317
```

In addition, should you forget a processid, or just want to see what's going on, you can use the utility ps (for "process status") to display all of the processes associated with your terminal:

```
% ps
1577 run  ps
1576 run  sh
1544 run  pr
1520 wait List
 331 wait sh
```

In this case, five processes are present. Listed for each one are its processid, its execution status, and the name of the program being run.

Finally, the wait command may be used to wait for all background processes to terminate before continuing. wait returns only when all processing is complete:

```
% wait
```

CONNECTIVES

Both ';' and '&' cause a set of commands to be executed unconditionally. Clearly, however, it would sometimes be quite useful to execute commands based on the result of some previous command. Any program under Idris returns one of two values, called success or failure, when it terminates. You can execute a command based on the return value of the last previous command by separating the two with "&&" or "||".

A "&&" causes its right-hand command to be executed only if the left-hand command returned success, while "||" executes its right-hand command only if the left-hand one returned failure. For instance, the utility cmp compares pairs of files, and returns success if each pair is identical. The following would delete a personal copy of the file manpage only if it had already been copied into a second directory:

```
% cmp manpage /final/doc/manpage && rm manpage
```

while the following would update a personal copy from the second directory if the file had changed there:

```
% cmp manpage /final/doc/manpage || cp /final/doc/manpage manpage
```

(There are shorter ways of writing these calls on cmp and cp -- see their manual pages in Section II.) These "logical connectives" don't cause any data to be passed between the commands they connect; they are used exclusively by the shell to determine whether or not to execute the right-hand command.

Any series of commands may be grouped together by enclosing them in braces "{}". The series will then be treated as if it were a single, simple command. Thus to record some lines in a file before running an editor script, you could type:

```
% {grep string file1 >before; e <script >trace file1} &
```

Note that if you entered these two commands without grouping them, then grep would be executed in the "foreground" (i.e., while you waited), then e would be run in the background, because '&', as a separator, affects only the command immediately preceding it.

This point brings up the more general question of how the shell connectives we've looked at can be used together. They group together according to the following hierarchy:

1) a simple command (i.e., a command name and a series of ordinary arguments) may contain a single STDIN redirection (signalled by '<' or "<<"), a single STDOUT redirection (signalled by '>' or ">>"), or both.

2) a simple command may also be used as part of a pipeline, separated by '|' from the other commands in the pipe; however, because a pipeline involves redirection, a pipeline "source" (the left-hand command) cannot have its STDOUT already redirected, while a pipeline "sink" (the right-hand command) cannot have its STDIN already redirected.

3) any two of the above groups (simple commands or pipelines) may be connected by "&&", which causes the right-hand group to be executed only if the left-hand one succeeds.

4) any two of the above groups (including groups connected by "&&") may be connected by "||", which causes the right-hand group to be executed only if the left-hand one fails.

5) any two of the above groups (including groups connected by "||") may be separated by ';', which causes the left-hand group, then the right-hand group, to be executed in sequence.

6) any two of the above groups (including groups separated by ';') may be separated by '&', which causes the left-hand group to be started,

then the right-hand group to be started, without waiting for the first group to terminate.

7) braces "{}" may be used to enclose any other construct, and cause it to be treated as if it were a simple command (i.e., as a single unit), thus overriding the hierarchy as parentheses would in an arithmetic expression.

All of the operators listed here obviously have special meaning to the shell, and are not normally passed to a command as arguments. If you want to strip them (or any other characters) of special meaning, you can do so in two ways. Any string of characters can be enclosed in quotes, causing each character inside to be passed literally to the command. Or, any single character (outside a quoted string) may be passed literally by preceding it with a backslash '\'. The quotes (or backslash) themselves are removed by the shell, and are not passed to the command.

PREFIX COMMANDS

Several utilities exist that take a command as an argument, and execute it after changing some condition under which it will run. One of these "prefix" utilities, called error, enables you to redirect the error output for a command, either to STDOUT or to another file. Normally, such error output is written to a standard file called STDERR (a counterpart to STDIN and STDOUT which gets nearly all error messages); STDERR, however, cannot be redirected straight from the command line. So to redirect it you would type something like:

```
% error -o trace grep string file1 file2 file3
```

which would run the grep command shown, after redirecting its STDERR to the file trace, to be created before grep is run.

Two other prefix utilities, called nice and nohup, enable you to run a command at a different priority than usual (be nice to others), or immune from certain interrupts (don't die if dataphone hangs up). A third, named time, records the amount of time consumed by its argument command, and outputs the totals to STDOUT after the command terminates. Because of the way the shell parses a compound command, a prefix utility can accept only a simple command as an argument; however, any command can be treated as a simple one just by enclosing it in braces. Thus to time the execution of a pipeline, you could type:

```
% time {sort data | uniq -c >output}
```

time, in turn, would output the amount of real time consumed by the pipeline, and the time it spent executing user code and system calls. (The last two figures together represent the "execution time".)
Note that, without the braces, time would be timing sort alone, and the entire command line would be interpreted as a pipe, with the output of time piped into uniq.

FILENAME SHORTHAND

In addition to the process control operators presented so far, the shell provides several varieties of shorthand to ease the use of complex or frequently invoked commands. First of all, the shell permits whole groups of files to be named as a single argument, by using each argument containing any of the characters "*?^[" as a regular expression to be matched against the names of the files in the current directory. (A regular expression is a shorthand for naming a set of strings; see the editor essay earlier in this Section for a full description.) Each such argument is stripped of any directory pathname (i.e., all the characters up to and including the rightmost slash), then compared against the set of filenames. If any filenames completely match the remaining pattern, they replace it in the command line. Otherwise, it is left untouched. For instance, the command

```
% echo *
```

would output the names of all the files in the current directory, and

```
% cmp file[a-g] /final/doc
```

would compare any files with the names filea, fileb, ..., fileg to a set of files in /final/doc with the same names.

Since the shell bestows this expansion on you quite unbidden, it also permits you to disable it, by quoting any argument you don't want expanded. Thus

```
% echo "*"
```

would pass to echo a single argument, consisting of a literal asterisk.

SHELL VARIABLES

A more flexible form of shorthand is provided by "shell variables", each of which can be assigned a string as its value, then used in place of the string. Shell variables have one-character names drawn from the set [0-9a-zA-Z$]. You refer to a shell variable by giving its name, preceded by a '$'. Wherever a reference, like "$a", occurs, the shell substitutes the current value of the variable. The text of the value is then scanned by the shell exactly as if it had been directly specified in the command line. This means that, if the value contains special characters (including further variable references) they will be fully interpreted.

Variables may be assigned values by using the command set, which takes two arguments: the name of the variable you're assigning to, and its new value. Suppose, for example, you were frequently accessing a directory other than one you were working in. Rather than constantly naming the second directory, you could assign its name to a shell variable, and then use the variable in place of the name:

```
% set p /final/doc
```

Once this set command had been given,

```
% cmp manpg1 $p
```

would be exactly equivalent to

```
% cmp manpg1 /final/doc
```

The new value must be quoted if it is to contain whitespace, since set takes only its second argument, not the rest of the command line, to be the value.

Four shell variables of the possible 63 are used by the shell to specify parameters, and so have special meaning. The variable P specifies the prompt that an interactive shell issues before accepting a command; H names your "home directory" (i.e., the directory changed to by a cd with no arguments); X specifies your "execution path"; and $ contains the processid of the process running the shell. To change any of these parameters, just use the set command on the appropriate variable. The following would suffice to alter your prompt:

```
% set P A:
```

An execution path names the set of directories that the shell will search for files with names matching the command names you give. Utilities like cmp or pr are merely executable files stored in a standard directory that is normally included in your execution path. By default, you may be given the path, say,

```
|/bin/|/etc/bin/
```

which causes the shell to look for commands first in the current directory, then in the directory /bin, and finally in the directory /etc/bin. Each string between '|' is a pathname prefix that the shell prepends to the command name you give. It then tries to execute a file with the resulting name. If it can't do so, it tries the next pathname; if no pathnames are left, you get the "not found" message you've probably already seen.

An execution path is most commonly modified to cause the shell to search some private command directory when it looks for commands. That way, you can create your own commands, put them in a single place, and then use them from anywhere at all. For example,

```
% set X "|/usr/myid/bin/|/bin/|/etc/bin/"
```

would add /usr/myid/bin to your execution path, where it would be searched before the system-wide binary directories. You could then name any executable file stored there, and the shell would automatically find it. The execution path string given must always be quoted, since otherwise the shell would interpret each '|' as a pipeline delimiter. A shorter way of giving the same path would be:

```
% set X '|/usr/myid/bin/|$X'
```

Inside single quotes, but not double quotes, references to shell variables are expanded, but the expanded text is inserted into the command line with no interpretation, except for the expansion of any further variable references it contains. Here, the old contents of X would be concatenated with the new pathname to make a single string, which would then become the new value of X. Except for this limited expansion of variable references, single quotes have the same effect as double: they suppress the ordinary interpretation of whatever characters they enclose.

SCRIPTS

A previous essay noted in passing that the shell is nothing more nor less than a standard utility named sh. You haven't seen it used that way, because sh is automatically run for you when you login to a session, in what is called interactive mode. In this mode, the shell prompts for commands from STDIN, and ignores interrupts (so that you can use DEL to stop shell commands, and not the shell itself). But since it is an ordinary program, sh can also be run directly. If you type:

```
% sh -i
```

you'll start up a new interactive shell underneath your current one that will act just like its parent. (So of course typing ctl-D will terminate it.)

More interestingly, the shell will also read commands from a file other than STDIN, specified on its command line. This permits you to place shell commands in an ordinary text file (called a "shell script"), then pass the filename to sh as an argument. Thus potentially complex operations that you need to perform frequently can be invoked with a single, short command line. For instance,

```
% sh cleanup
```

would start up a shell to read the commands stored in cleanup, which might contain the actions you normally take before logging off.

And scripts can be invoked directly by name, just as if they were binary programs, by giving them execute permission. Through a bit of chicanery when it tries to run a command, the shell will interpret as a script any executable file that is not a legitimate binary program, and will start up a second shell to read the script. So, if you typed:

```
% chmod cleanup
```

in order to make cleanup executable, you could then run it exactly like a binary program. And, if you stored it in some directory on your execution path, you could run it from anywhere just by typing its name:

```
% cleanup
```

Though simple scripts like this one can be a valuable form of shorthand, shell variables greatly extend their potential, by making it possible to pass arguments to a script like the ones passed to a binary program. Whenever you run a script, the variable 0 is set to the name of the script, and variables 1 through 9 are set to the values of its first 9 arguments. If fewer than nine arguments are given, the extra variables are assigned an empty string as their value; and outside a script, the whole group contains empty strings.

If we wanted to simplify the usage of the glossary-maker in the first essay, we could put this line into a shell script named gloss, and then invoke the script instead of typing the command line:

```
tr A-Z -t a-z $1 | tr !a-z -t "\n" | sort | uniq
```

When the script is run, the reference "$1" is replaced by the value of the first argument to the script. So to run gloss on a file, all you'd need to do is name the file as the first argument. Presuming gloss had been made executable, the following would produce a glossary of trial, just like the last example of the first essay:

```
% gloss trial
```

In addition to the numeric variables, two special variables, @ and *, are provided, which have as their value all of the arguments passed to a given script. The reference "$@" is replaced by the literal text of the arguments, each one treated as a separate string; "$*" is replaced by the same text, but quoted so as to be treated as a single string, consisting of the arguments separated by spaces. Thus, to make the script gloss produce a combined glossary from any number of input files, you could change it to be the following line:

```
tr A-Z -t a-z $@ | tr !a-z -t "\n" | sort | uniq
```

which would pass to tr all of the arguments given to the script. Outside a shell script, @ and * always have the empty string as their value; never can they be assigned new ones.

This ability to access command line arguments from within a script is central to still another form of shorthand, the prefix command exec. exec takes as an argument a command that the current process will start running instead of the shell. The shell you had been running is lost forever. From an interactive shell, exec is not generally useful, and is even somewhat dangerous, since the shell does not return when the specified command finishes execution. (Instead, you'll usually get logged off.)

From within a script, however, exec is a fast way to invoke a single (perhaps complicated) command line. It's fast because, since the shell running the script is destroyed when exec is run, it doesn't hang around for the command line to finish. If, for instance, you frequently used pr to generate tabular data, you could save yourself time and potential errors by putting the special pr command you need into a script, say maketab, then using the script whenever you wanted to generate data:

```
exec pr -t -m -s: $@
```

(See the manual page for pr in Section II for the meaning of all these neat flags.) So long as maketab were executable, you could form a table from three files just by typing:

```
% maketab file1 file2 file3
```

since any filenames you give to maketab are automatically passed to pr.

The special variable @ makes it very easy to pass an entire argument list to a command inside a script. But suppose we wanted to have the same external interface (i.e., the ability to accept an arbitrary number of arguments), and also wanted to process each argument separately? To do both, you might make the script contain a loop, each iteration of which would process a single argument file. For instance, the following would selectively remove each file passed to it as an argument, based on whether or not it also existed in a second directory:

```
: loop
    echo $1:
    cmp $1 /final/doc && rm $1
    shift && goto loop
```

The first line is a label that will be used by the later goto command, while the next line outputs to STDOUT (presumably a terminal) the name of each file, followed by a colon, as it is processed. The last line does two things. First, shift is used to "shift" the arguments to the script. That is, shell variable 1 is made to point to the previous value of variable 2, 2 is made to point to the value of 3, and so on. In this manner, the filenames on the command line are shifted left each time shift is run, and each one in turn is associated with variable 1. shift succeeds only if variable 1 was assigned a non-empty string, i.e., if there was another filename to be shifted into it. So long as there was, goto is run, which scans the script from the beginning for a "label command": whose first argument matches the first argument to goto; if such a label command exists, then the execution of the script resumes with the line following it. When shift returns failure, goto is not run, and the script terminates.

It should be emphasized that, once any script is made executable, it can be used in a command line exactly like a binary program. So if the last script were stored in an executable file called remove, a command line like this one could be used to remove a series of files:

```
% remove file1 file2 file3
```

Or to redirect the output of maketab to the file table, you could type:

```
% maketab column1 column2 column3 > table
```

while the following would suffice to pipe the output of gloss into pr:

```
% gloss file1 file2 | pr
```

SHELL PROGRAMMING

Scripts like remove, which execute some set of commands on each of a list of files, are examples of the programming language that the shell defines, a language of which scripts can be viewed as the procedures or even the subroutines (because, of course, scripts can be named as commands inside other scripts). The shell language, however, has further capabilities. Scripts are capable of full interaction with a terminal, and can perform conditional branching based on the responses they receive. The following lines might appear at the start of a (rather simple-minded) script managing the access to a file of data records:

```
: input
    echo -m "enter next command --" "(newline to quit)"
    set c -i
    test $c || exit
    test $c == a && goto add
    test $c == d && goto delete
    test $c == h && goto help
    echo "don't recognize response -- type h for help"
    goto input
```

Each time it was executed, this excerpt would prompt STDIN for a single line of input, which it would try to interpret as a single-letter command. The opening echo command -- banal enough if typed in from your keyboard -- here serves to output the two strings shown to the terminal. The set command specifies a flag, -i, that causes the value for the variable c to be read as a single line of STDIN, instead of being taken from the set command line.

What follows is a series of checks of the value that c has just been assigned. The utility test compares two strings for equality (s1 == s2), or one string for being non-null, and succeeds only if the relationship tested is true. The first test command checks whether c has a non-null value. If not (i.e., if test fails), then the user typed an empty line, and exit is run -- which, as its name implies, causes the script to exit. The subsequent test commands all check whether or not c is equal to a specific command letter. (The "==" is what checks for equality; "!=" could be used instead to check for inequality.) If one of the command letters does match, then test succeeds, and the corresponding goto is executed. Otherwise, the script falls through to the following echo, which outputs an error message and goes back to get another command line.

The exit command can also be used to control the return value of a script, or more precisely, of the shell reading the script (which returns success or failure just like any other program). If an exit is executed with no arguments, the shell returns success; if the exit command contains an argument, the shell returns failure. Or, if the shell terminates by reading off the end of a script (and not with an explicit exit), it returns the return value of the last command it executed. Hence, this line could be inserted at the start of the script remove, to force it to fail when not

given at least one argument:

```
test $1 || echo "usage: $0 <files>" && exit NO
```

The argument "NO" to exit causes the shell to exit reporting failure (any non-null argument would do). Note the use of $0 to pick up the name by which the shell script was invoked, for printing out the error message.

Another versatile command is set -i, which, since it reads one line of STDIN, can easily be used as a sink in a pipeline. A frequent usage is:

```
pwd | set d -i
```

which permits a script to record what directory it's running in, so that the directory can be restored later.

One additional feature of scripts is that they may contain embedded instances of interactive commands. If a command contains the redirection symbol "<<", then the current source of shell command input is read, up to a line containing only the string following the "<<". All of the lines read are put into a temporary file that is then made to be the STDIN for the command. For instance, if shell variable w already contained the name of the user running a script, then the run could be recorded at the start of a log file with the commands:

```
date | set t -i
cat > /tmp/logent << END

run by $w at $t:
END
cat /final/doc/log >> /tmp/logent
mv /tmp/logent /final/doc/log
```

The first line, of course, gets the output of date into the variable t. The cat command following reads its STDIN (because it has no input files), and writes the lines read to the file /tmp/logent. Because of the "<<", the STDIN for cat is created from the next lines in the script, which is read until END is encountered on a line by itself. (The terminating string can be anything at all, so long as it matches the one given after the "<<".) The final two lines add the new entry to the start of the log file, by appending the old log to the new entry, then naming the result so as to replace the old log.

As shown above, shell variable references are expanded in the STDIN created by "<<", though by simpler rules than in a shell command line. In a "<<" STDIN, any sequence "$x", where x is a valid variable name, will be replaced with the contents of x, as for an unquoted command line reference. However, if x is not a legal name, then the two characters are left untouched. Any '$' may be treated as literal by preceding it with a backslash -- an important thing to remember when creating command input for e, to which '$' has special meaning.

Since all of these last features are part of the shell, they can also be used from your terminal in interactive mode, though with varying utility.

A few, like labels and the goto command, make sense only in scripts. Others, like "<<", can be quite useful when used interactively. Consider the following line, which would enable you to enter a script of editing commands from the keyboard, then execute the editor in the background, working on file1:

```
% e << END >trace file1 &
```

When you entered this command, the shell would read lines from STDIN until you typed END on a line by itself; only then would it start up a background process to run e, which would read the commands you just entered.

All of which provides a final example of a more general fact: the shell is designed to encourage experimentation, and the innovative use of its facilities. It provides a versatile set of parts out of which "programs" (perhaps as simple as a pipeline) may be quickly and easily built to solve a broad range of data processing problems. The goal of this essay has been to introduce those parts, and give some hints of how they can be used. But this has been only an introduction -- the power of the shell and of the utilities accompanying it lies in the many, flexible, and often surprising ways in which they can be combined, which no essay could ever adequately cover. So experiment.

NAME

Glossary - a lexicon of Idris terms

FUNCTION

The terms defined here are either jargon specific to the world of data processing under Idris, or terms with a slightly more specialized meaning in this context than may at first be apparent:

argument - a whitespace delimited string on a command line, frequently a filename, conveyed to a program to modulate its behavior.

ASCII - acronym for American Standards Code for Information Interchange, the method of representing characters as small numbers, in the range [0, 128), used by Idris.

backspace - ASCII BS code, which causes the cursor (or print head) to move backward one character position.

binary - a positional numbering system with two digits [0-1], used to perform all arithmetic, at the lowest level, in a computer. Also, containing byte values other than the ASCII character codes, hence in opposition to text. Also, used to describe programs directly executable by the computer, as opposed to shell scripts.

bit - a binary digit, hence a number with just two values (0/1, false/true, off/on). Usually grouped in multiples of eight to represent positional binary numbers. All computer storage and computation is, at the lowest level, in terms of bits.

block - a contiguous group of 512 bytes, the basic unit of space allocation in filesystems.

bootstrap - the process or program that gets a computer going from a standing start.

buffer - an area of temporary storage, used to stage data before it is copied to its final destination.

byte - a group of eight binary digits, hence a number with 256 possible values. The basic unit of data storage and transmission, since it can comfortably represent all the ASCII character codes, one per byte.

character - a single keystroke, usually stored as ASCII in a single byte, typically representing a letter, digit, punctuation, or other printable text.

child - the younger member in a hierarchical relationship, as among directories or processes.

close - to break the connection between an opened file and a program.

command - a line of text interpreted by an interactive program, such as the shell or the editor, that tells it what to do next.

completion - (pathname), a method for determining the name of a file, given its parent directory and a source file name.

copy - to duplicate, as when copying a file or other data stream to a new place.

core - common name for a file used to record the running status and memory image of a program that was forced to terminate abruptly.

create - to make a new instance of, as when creating a file.

ctl- - ASCII FS code, typed at a terminal keyboard on a controlling tty to send a quit signal, which causes most programs to terminate abruptly, with a core dump.

ctl-D - ASCII EOT code, typed at a terminal keyboard to simulate end-of-file condition.

ctl-Q - ASCII DC1 code, typed at a terminal keyboard to force printout to be resumed.

ctl-S - ASCII DC3 code, typed at a terminal keyboard to force printout to be suspended.

decimal - a positional numbering system with ten digits [0-9], used to count all the fingers on your hands.

decrypt - to translate encrypted data back to a recognizable form, by using a secret password.

default - a value assumed in the absence of an explicit input value.

destination - the place where data is to end up.

device - a piece of machinery or electronics connected to a computer for the purpose of entering or extracting data.

diagnostic - an error message output by a program to inform the user of some problem.

directory - a file whose contents are the names of other files.

editor - the standard utility providing commands to create and modify text files.

encrypt - to translate data to a form that is neither recognizable nor easily translated back to recognizable form in the absence of a secret password.

executable - having access permission, and the proper format, to be run as a program.

exit - to terminate program execution.

failure - status returned on program exit, indicating something went wrong. Testable by the shell.

file - a contiguous collection of bytes stored within a filesystem, known by one or more names or links.

filename - the name by which a file is referenced, synonymous with pathname under Idris.

filesystem - a logical data structure imposed on a contiguous collection of disk blocks, used to represent files and directories under Idris.

filter - a program that consumes an input stream from STDIN and produces and output stream to STDOUT. Often the streams are text.

flag - a shell command argument beginning with a '-' or '+', used to pass parameters to programs.

footer - a line of text appearing at the bottom of a formatted page.

free - available for reuse, as a block of a filesystem.

groupid - a number in the range [0, 256) used to encode group membership for access permission checking.

hexadecimal - a positional numbering system with sixteen digits [0-9a-f], used as a shorthand for groups of four bits.

I/O - input/output, generic term for data transmission between a program and its environment.

Idris - a Persian god, the patron deity of craftsmen, credited with the invention of any number of tools and crafts, including the art of sewing things together.

inode - a filesystem entity that contains all of the attributes of a file, except its contents and name(s).

interactive - a mode of program operation in which a person can type a command, then see the result before deciding on the next command.

interrupt - a signal, generated by striking the DEL key on a controlling tty, that causes most programs to terminate abruptly.

invoke - to cause a program to begin execution.

kill - a signal, generated by the kill utility, that causes any program to terminate abruptly, with a core dump.

link - a directory entry that points to a file, one component of a pathname.

loginid - a string of one to eight characters used to identify a user to the system at login time, and to associate a password, userid, groupid, and other operating information with a given user.

match - to meet the requirements of a regular expression, as for a substring within a text line or for a filename in a directory.

mode - of a file, the access permissions associated with the file.

mount - to make a filesystem available through the directory hierarchy.

move - to change the location of, as when renaming a file.

newline - ASCII linefeed (LF) code, used in all Idris text streams and text files to terminate each line.

octal - a positional numbering system with eight digits [0-7], used as a shorthand for groups of three bits.

open - to make a file available to a program for input and/or output.

parameter - a value used to modulate the behavior of a program, often passed as an argument.

parent - the older member in a hierarchical relationship, as among directories or processes.

password - a short text string used to encrypt and decrypt confidential data, also used to authenticate a loginid by encrypting a known constant. Preferably kept imaginative and secret.

pathname - a sequence of slashes and links that specifies the name of a file, relative to either the current or root directory.

peripheral - synonym for device, often used redundantly as in "peripheral devices".

permission - for a file, the license to access it for reading, writing, or executing. For a directory, the license to access it for reading, altering, or scanning.

pipeline - a data stream connection between two simultaneously executing programs that permits one to read what the other is writing.

prefix - (command), a command which runs other commands, possibly first modifying the execution environment.

process - the execution of a program, which in Idris is controlled by a resident data structure often also called a process.

program - a file that can be directly executed by the computer, with the assistance of the Idris resident.

prompt - a string of text written to a terminal to signal that an interactive program is ready for a command line to be input.

recursive - a self-referential rule for performing an operation, as when climbing over a directory tree to visit all subdirectories.

redirection - altering the source for input to a program, or the destination for its output, to a file specified on a command line to the shell.

regular - (expression), a notation for representing text patterns, used for matching.

resident - that portion of the Idris system which resides in main computer memory at all times, to assist executing programs and to administer peripheral devices and other shared resources.

return - (carriage), to bring the cursor (or print head) back to the left margin, as with an ASCII carriage return (CR) code. For a program, to deliver up a success or failure indication upon completion.

root - the fundamental directory of any filesystem, in particular, the fundamental directory of the filesystem specified at system startup. Also the conventional loginid of the superuser.

script - a file containing a sequence of commands for a program that is normally interactive, such as the shell or the editor.

shell - the standard utility used to interpret commands to run other programs.

source - the place from which data is to originate.

space - the ASCII character generated by pressing the space bar on a keyboard, which causes a one-character wide space to be left when displayed.

special - (file), a file that causes a peripheral device to twitch when accessed. There are character special files, such as ttys, and block special files, such as disks.

startup - (system), the time right after bootstrap and before normal operation, when the Idris system initializes everything in sight.

STDERR - standard error text stream, made available to each utility run by the shell.

STDIN - standard input text stream, made available to each utility run by the shell.

STDOUT - standard output text stream, made available to each utility run by the shell.

stream - an ordered sequence of data that passes through a program once, in sequence.

success - status returned on program exit, indicating nothing went wrong. Testable by the shell.

superuser - userid number zero, usually a persona of the System Administrator, having the power to circumvent most access restrictions and to perform restricted operations.

tab - ASCII HT or VT code, which advances the display to the next tab stop. Horizontal tabs (HT) are assumed every four columns under Idris, but can be passed on to the terminal to expand. Vertical tabs (VT) are always passed on.

text - a stream of printable characters, usually intelligible to people.

title - a line of text appearing at the top of a formatted page.

tty - a character special file that supports an interactive terminal.

unmount - to disconnect a mounted filesystem from the directory hierarchy.

userid - a number in the range [0, 256) used to encode the identity of a user for access permission checking.

utility - a program of general usefulness provided by the Idris system.

whitespace - a generic term for spaces, tabs, newlines, and other ASCII characters that cause printer motion but produce no graphics.

word - in text processing, a contiguous string of letters and punctuation, delimited by whitespace. In some circles, a group of bits that is a natural size, in some sense, for a given computer.

II. Standard Utilities

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 and a mandatory argument, which precedes the rest of the command line, followed by flags 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> <arg> <flags> <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. 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 usually are graded, going from quite simple to more elaborate, and 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 a blank 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 complains and returns failure if it cannot change to the new directory.

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

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 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, for each tab to be skipped a string of spaces, followed by non-spaces other than newlines, is skipped instead. Thus, in the string " ABC DEF GHI", the offset "3" would point to the space just after 'I'.

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

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 -[d e 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:

-d decrypt the input.

-e encrypt 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 neither or both of -[d e] are given, -e 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 -p chortle -o muddy 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 -[t y# +#] <date>

FUNCTION

date writes the current date and time to STDOUT. If <date> is present, the current date is altered as specified before being output.

The flags are:

-t translate <date> into output format, 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 <date>, a character string up to 8 characters indicating month, day, hour, and minute in the form MMDDHHMM. A short string leaves unchanged the fields that are unspecified on the left.

A new date is also written to /adm/date, and a log entry is appended to /adm/log if it exists. Errors in opening these files are ignored. Note that +0 causes the current date to be altered, but left unchanged; this is useful for time stamping the root filesystem before a system shutdown.

The system date can be changed only by the superuser.

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

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] <filesys>

FUNCTION

df indicates the amount of free space left in one or more filesystems. If no filesystems are specified by <filesys>, or if "-" is encountered on the command line, df looks up the filesystem containing the current directory in /dev and the output is preceded with that filesystem name. If any of the names in <filesys> is not a valid block special file, it is assumed to refer to an ordinary disk file, and statistics are output for the device on which the file resides.

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 ntext.
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 -[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:

-b# 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.

-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 regular expression. 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; 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 nThe 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> 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.

(.)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.

(.)=[lp]
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

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.

RETURNS

The value returned is that of <command>.

EXAMPLE

To switch to a private shell:

```
% exec myshell
```

SEE ALSO

sh

BUGS

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

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 all of its input files are readable.

EXAMPLE

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

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

SEE ALSO

grep, last

NAME

grep - find occurrence of pattern in files

SYNOPSIS

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

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

At most one of -b, -c, -n, or -p may be specified.

RETURNS

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

EXAMPLE

The command:

```
% grep EOL -n 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 footer to text files

SYNOPSIS

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

FUNCTION

head adds title or footer 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 footers) 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 footers 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 (footer) 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 footer. 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.

-r reverse the left and right components of the header and footer 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 footers 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 footer line # lines up from the bottom of the page; -0 puts it on the last line. The default is -2.

Title and footer 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 footer is output; no -t likewise suppresses titles. Title and footer 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 footer 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 -[#] <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:

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

RETURNS

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

EXAMPLE

To see the last 20 lines of two files:

```
% last -20 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 <oldname> -[patch] <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.

RETURNS

lpr always fails, for no good reason.

EXAMPLE

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

FILES

/dev/lp for line printer.

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 as follows:

-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 list 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.

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 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. If one or more <loginids> are specified, or one or more group loginids via -g flags, mail reads STDIN up to end of file, or up to a line containing only '.', and delivers the lines read to the private mailboxes of all receivers, together with a postmark giving the sender and date. If any loginids are invalid, the message is written in the file "dead.letter" in the current directory.

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

The <flags> are:

-a send message to all system users.

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

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

-i receive mail interactively.

-q quit immediately, returning success if there is mail to be received. Either "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.

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

p print the message to STDOUT.

+ ask about next message, quit if no more.

- ask about previous 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.

An empty prompt line 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'.

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

```
% mail -a
Subj: Spring picnic
There will be a FREE picnic, Feb 30 1980.
All company employees and their families
are invited to attend.
                    sincerly,
                    E. Scrooge
.
% mail -i
From scrooge  16:55 Feb 29 1980? d
```

FILES

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

SEE ALSO

write

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 -[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 flag is:

-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

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

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 permission for the directory to be moved. 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
```

BUGS

Making a directory a subdirectory of itself causes the entire directory subtree to be orphaned, a disaster which requires major filesystem surgery to undo.

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 -[h i q] <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:

-h ignore hangups (signal 1).

-i ignore interrupts (signal 2).

-q 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

BUGS

Flags should be reconciled with 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. Default is 0.

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
00000000012    2 \n \n  l  i  n  e     4 \n  l  i
00000000024    n  e     5 \n  l  i  n  e     6 \n
00000000036    l  i  n  e     7 \n  l  i  n  e
00000000048    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

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

pr - print files in pages

SYNOPSIS

pr -[c# e# h l# m n 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.

-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
```

NAME

ps - print process status

SYNOPSIS

ps -[a i l m p r s#]

FUNCTION

ps prints information about processes in the system. It reads /dev/ps, to find out about all the processes in the system, or /dev/myps, to find out about the processes associated with the invoking user's terminal. ps will also print out information of interest to gurus, such as the list of active files (the inode list from /dev/inode) and the number of mounted devices (from /dev/mount).

The flags are:

-a print out all processes.

-i print out the inode list.

-l give the long form printout.

-m print out the mount list.

-p print out your processes only.

-r print out running processes only. This acts as a qualifier to -a and -p.

-s# sleep # seconds between print outs. ps runs continuously but can be stopped with the DEL key or kill command.

If no flags are given, -p is assumed.

ps prints out the processid (in decimal, the unique number of that process), the process state (running, waiting, dozing, or idling), and the first four characters of the process name (usually the name of the file executed). If -l is specified, additional information is given on: the process priority (a signed integer), the processid of the parent process, the size in blocks of the process image, and the associated tty device for the process.

An Idris process may be in one of four states: running (needs the computer, but may or may not be in memory), waiting (needs terminal input, or terminal output is backed up), dozing (needs a disk block, or access to a file), or idling (a newly created child process). Priorities of Idris processes range from -128 to +127 (negative numbers and zero are higher priority than positive numbers). Every process is started by a parent process (ppid); process 1 is the gread grandaddy of them all. Every process started by you is associated with your tty device (dev).

RETURNS

ps returns success if all of its reads and writes succeed.

EXAMPLE

```
% ps -l
 PID STAT NAME      PRI PPID SIZE  DEV
1684 run  ps          5 1683   23  dz7
1683 run  sh          6 1672   54  dz7
1672 run  e           2 1598   46  dz7
1598 wait sh          1 1591   54  dz7
1591 wait sh          1    1   54  dz7
```

FILES

/dev/cnames for tty device name, /dev/inode for the inode list, /dev/mount for the mount list, /dev/myps for your processes, /dev/ps for all processes.

BUGS

The flag -m should display more information.

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

```
% 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 leterhead	yes	
.LL	line length	no	78
.LP	block paragraph	yes	
.LS	line spacing	no	1
.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	right margin	no	78
.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.

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
.BP 5
centered text
Page five:
.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 five: 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.

-- 5 --

BUGS

Much is left to the imagination, and experimentation.

NAME

set - assign a value to a shell variable

SYNOPSIS

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

EXAMPLE

To change your shell prompt:

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

SEE ALSO

sh

NAME

sh - execute programs

SYNOPSIS

sh -[H* P* 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.

-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 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.
X an execution path as supplied by -X.
$ the current processid as a 5 digit octal number.

All single-letter variables other then H, P 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, 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.

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 in executable format, 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 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. Such a line has no other effect.

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.

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 uninterrupted.

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, for each tab to be skipped a string of spaces, followed by non-spaces other than newlines, is skipped instead. Thus, in the string " ABC DEF GHI", the offset "3" would point to the space just after 'I'.

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 -l -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 -l -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 l.

SEE ALSO

comm, uniq

NAME

stty - set terminal attributes

SYNOPSIS

stty -[bs# cr# c +echo echo ek erase* +even even ff# ht#
+hup hup i# kill* +mapcr mapcr +mapuc mapuc nl# +odd odd o#
+raw raw s* type* t +xtabs xtabs] <dev>

FUNCTION

stty displays or changes the characteristics of a terminal. If <dev> is specified ("/dev/" should be part of the name) that device is used, otherwise the device associated with STDERR is assumed. Note that only a terminal or tty file may be operated upon by stty. If no flags are given, stty will print the current status as with the "-t" flag, but omitting any switchable characteristics that are the same as the system default.

The flags are:

-bs# set typeout delay for backspaces to #. A value of 0 specifies no delay, 1 specifies 16/60 sec.

-cr# set typeout delay for carriage returns to #. Legal values for # are in the range [0, 4), giving the delay in multiples of 4/60 sec.

-c print the current status in command format. This output may be redirected to a file, and later executed to restore the terminal status. If any other flags are given, they affect the terminal, but not the output.

+echo steer all characters typed in back out for full duplex operation.

-echo turn off echoing of characters typed at the terminal.

-ek set the erase and kill characters to the system default ('\b' and '@' respectively).

-erase* make the first character of * the erase character, i.e. the character which, if typed in other than raw mode, calls for the preceding character on the current line (if any) to be deleted.

+even accept even parity characters on input. Generate even parity on output.

-even ignore even parity characters on input.

-ff# set typeout delay for formfeeds and vertical tabs to #. A value of 0 specifies no delay, 1 delays for 64/60 sec.

-ht# set typeout delay for horizontal tabs to #. Values are the same as for -cr#.

+hup hangup a connected dataphone, if possible, when you logout.

-hup don't hangup dataphone.

-i# set the input baud rate to the speed referenced by the code #. Legal values are in the range [0, 16), corresponding to the baud rates {0, 50, 75, 110, 134.5, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, exta, extb}. A baud rate of 0 calls for the tty to hangup; exta and extb are speeds strongly dependent on the particular device. By no means do all devices support all speeds.

-kill* make the first character of * the kill character, i.e. the character which, if typed in other than raw mode, calls for the entire current line to be deleted.

+mapcr accept a carriage returrn CR as a linefeed LF (or newline), and expand a newline on output to the sequence carriage return, linefeed.

-mapcr don't map carriage return to newline.

+mapuc map all uppercase characters typed in to lowercase, and all lowercase characters typed out to uppercase.

-mapuc turn off mapping of uppercase.

-nl# set typeout delay for newlines to #. Values are the same as for -cr#.

+odd accept odd parity characters on input. Generate odd parity on output.

-odd ignore odd parity characters on input.

-o# set the output baud rate to the speed referenced by the code #. Values are the same as for "-i#".

+raw ignore interpretation of input characters, including the processing of erase and kill characters, the recognition of interrupt codes DEL and FS (ctl-\), start and stop codes DC1 (ctl-Q) and DC3 (ctl-S), and the treatment of EOT (ctl-D) as end of file.

-raw interpret input characters normally, including line at a time editing with erase and kill characters, recognition of interrupt codes, start and stop codes, and the treatment of EOT as end of file.

-s* change the baud rate to * for both input and output. * must be in the set of baud rates listed above for "-i#".

-type* set the terminal type to that specified in the file /adm/stty for *. Note that if the line in the stty file calls for a flag in conflict with the one on the command line, an error will result.

-t print the status on multiple lines. If any other flags are given they affect the output, but not the terminal.

+xtabs translate horizontal tabs on output to the appropriate number of spaces to simulate tab stops every four columns.

-xtabs disable translation of horizontal tabs on output.

If neither even nor odd parity is specified, then either parity is accepted on input and a zero parity bit is provided on output. If both even and odd parity are specified, then either parity is accepted on input and a one parity bit is provided on output. In raw mode, the parity flags are ignored; the parity bit is not stripped off on input nor altered on output. Thus, raw mode is an eight-bit transparent channel.

Note that any change in the characteristics of a terminal causes all input to be discarded, thus defeating any type ahead.

RETURNS

stty returns success if no diagnostics are produced, if <dev> is a meaningful representation for a terminal, and if the invoker has read permission for it.

EXAMPLE

```
% stty
speed = 1200 baud
erase = \b kill = @
```

FILES

/adm/stty for predesignated terminal types.

SEE ALSO

stty(III), tty(III)

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

sync - synchronize disk I/O

SYNOPSIS

sync -[#]

FUNCTION

sync ensures that all disk images are synchronized with information retained within the resident. It updates all inodes and superblocks that have been modified and not yet written back, and writes all data blocks not correctly represented on block special devices.

The flag is:

-# repeat every # seconds forever. Only the superuser may invoke this option; it should only be used at system startup to create a synchronizing daemon.

sync should be used religiously before taking the system down, even if the sync daemon is running.

RETURNS

sync always returns success unless a non-superuser tries to specify the flag.

EXAMPLE

```
% sync
```

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 f* 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 zeor 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.

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

-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, followed by the name of the file. When -v is specified with -t, all information about each file to be tabulated is printed: The first displayed field is the file mode, 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 <src> -[b o* t*] <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 *. 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 !a-zA-Z -t"\n" <paper
```

NAME

uniq - collapse duplicated lines in files

SYNOPSIS

uniq -[c f g o* u] +[a b d l n 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, for each tab to be skipped a string of spaces, followed by non-spaces other than newlines, is skipped instead. Thus, in the string " ABC DEF GHI", the offset "3" would point to the space just after 'I'.

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 +l <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 w] <files>

FUNCTION

wc counts the number of 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.

-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
   12     24     84  file1
  165    730  10624  file2
   43    172    948  file3
  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

III. Standard File Formats

NAME

Files - special file formats

FUNCTION

Files under Idris can generally be thought of simply as ordered sequences of eight-bit bytes (characters), written on disk or tape. Since most programs access files just once, front to back, most data streams, such as are produced by terminals or interprogram pipelines, can be processed like ordinary files. There are, however, some special cases.

The manual pages in this section describe files whose formats are more restricted, typically because two or more programs must communicate via a common file and hence must impose additional structure on the information it contains. There are also a few "pseudo files", implemented by character special device handlers in the resident, that permit access to internal system information or provide useful services.

Additional file formats, of interest primarily to programmers, may be found in Section III of the Idris Programmers' Manual.

NAME

ASCII - standard character codes

FUNCTION

ASCII stands for American Standard Code for Information Interchange. It defines 128 character codes for printing graphics, controlling terminals, and formatting messages between text processing machines. All Idris utilities and the resident presume that text is encoded in ASCII, a safe bet because it is used universally -- except by IBM, the world's largest manufacturer of computing equipment, and a few other companies.

At any rate, the octal codes and their meanings are:

CODE	MEANS	CODE	MEANS	CODE	MEANS	CODE	MEANS
0000	NUL	0040	space	0100	@	0140	`
0001	SOH	0041	!	0101	A	0141	a
0002	STX	0042	"	0102	B	0142	b
0003	ETX	0043	#	0103	C	0143	c
0004	EOT	0044	$	0104	D	0144	d
0005	ENQ	0045	%	0105	E	0145	e
0006	ACK	0046	&	0106	F	0146	f
0007	BEL	0047	'	0107	G	0147	g
0010	BS	0050	(	0110	H	0150	h
0011	HT	0051	)	0111	I	0151	i
0012	NL	0052	*	0112	J	0152	j
0013	VT	0053	+	0113	K	0153	k
0014	NP	0054	,	0114	L	0154	l
0015	CR	0055	-	0115	M	0155	m
0016	SO	0056	.	0116	N	0156	n
0017	SI	0057	/	0117	O	0157	o
0020	DLE	0060	0	0120	P	0160	p
0021	DC1	0061	1	0121	Q	0161	q
0022	DC2	0062	2	0122	R	0162	r
0023	DC3	0063	3	0123	S	0163	s
0024	DC4	0064	4	0124	T	0164	t
0025	NAK	0065	5	0125	U	0165	u
0026	SYN	0066	6	0126	V	0166	v
0027	ETB	0067	7	0127	W	0167	w
0030	CAN	0070	8	0130	X	0170	x
0031	EM	0071	9	0131	Y	0171	y
0032	SUB	0072	:	0132	Z	0172	z
0033	ESC	0073	;	0133	[	0173	{
0034	FS	0074	<	0134	\	0174	|
0035	GS	0075	=	0135	]	0175	}
0036	RS	0076	>	0136	^	0176	~
0037	US	0077	?	0137	_	0177	DEL

Multiletter symbols are the accepted abbreviations for nonprinting characters -- their meanings (and usages) are often shrouded in antiquity.

SEE ALSO

print, text

NAME

block - block special files

FUNCTION

A block special file is a direct connection to a peripheral device capable of supporting direct (random access) reads and writes of 512-byte blocks. As many as 65,536 blocks can be accessed via a block special file, but any given device may support fewer, so long as those blocks actually present form a contiguous sequence starting with block zero. Devices with more than 65,536 blocks, and even many smaller ones, are conventionally partitioned into a number of smaller logical devices, so that all areas can be reached and each subarea can be administered separately.

Idris requires all filesystems to be written on block special devices, one to a logical device, if they are to be mounted. Conventional tape can often be treated as a block special device, so long as writes occur only sequentially and reads are never attempted past the highest numbered block written. Thus, a tape filesystem may be mounted read-only.

The Idris resident automatically provides for buffering of accesses to all block special files. This includes "write behind", or deferring writes as long as possible to combine multiple partial block writes, and "read ahead", or anticipating reads on the basis of a recent history of sequential block reads. Performance is enhanced, often dramatically, by this service, but at the risk of greater disk inconsistency on an unexpected shutdown. Thus, mechanisms are available, to the System Administrator, to cause an automatic flushing of the buffer cache at regular intervals, and to the System Generator, to defeat read ahead and/or write behind.

When read as a conventional file, a block special file will always deliver up a multiple of 512 bytes, and may report a read error at the first block beyond its defined areas, rather than a tidier end-of-file.

SEE ALSO

character

NAME

character - character special files

FUNCTION

A character special file is a direct connection to a peripheral device, about which little else can be said. One special category of character special files, the tty files, have a number of useful properties; all other files of this ilk are about as unstructured as anything supported by Idris. Typical character special files are:

raw tape - It is often desirable to write blocks larger than 512 bytes for the sake of storage economy, and it is often necessary to write blocks even smaller for the sake of portability to other systems. A raw tape interface will write records whose size is determined by the individual write requests, and will read records no larger than individual read requests. The standard utility dd can produce tailored requests, for use with raw tape devices, so that alien tapes can be administered by Idris.

raw disk - Raw disk typically permits the transfer of multiple 512-byte blocks with one read or write request, and hence can speed bulk copying. On devices with multiple sectors per block, such as diskettes, raw disk may also permit accesses to the individual sector level.

funny stuff - Printers, paper tape readers, realtime interfaces, etc. often have peculiar control requirements that don't fit completely within the confines of conventional Idris input/output. For these, the stty machinery can be used in nonstandard ways to provide the extra control needed, at the cost of requiring a special access program for each such device.

Block special devices typically provide an additional character special interface, if only to support multiple block reads and writes. Care should be taken when accessing the same physical disk area with two different logical devices, however, since the resident block buffering may be outsmarted.

Character special files may impose arbitrary constraints on read or write requests, such as: records must be an even number of bytes, or records must begin on a special storage boundary within the process, or offsets (seek pointers) within the file must be some multiple of bytes, or offsets may even be ignored. Thus, these devices are a major exception to the general rule that files are highly interchangeable among Idris utilities.

Avoid them.

SEE ALSO

block, dd(II), stty(II), tty

NAME

directory - directory files

FUNCTION

A directory is a file of named pointers to other files. The names are called "links" and the pointers are "inode numbers". Since directory entries can point at other directories, and since the "root" (inode 1) of nearly every filesystem is a directory, all the files in a filesystem are organized into a tree of arbitrary depth. To reach a given file from the root, it is necessary to traverse a path of links; hence a filename is sometimes called a "pathname". A directory consists of 16-byte records in the format:

bytes 0-1: the inode number, an unsigned short integer written less significant byte first. Zero indicates an erased link.

bytes 2-15: the link name, up to 14 characters left justified and NUL padded. A link may contain any character except a NUL or '/', since the former is taken as the end of a pathname and the latter as a separator between links in a pathname.

Although not required by the Idris resident, a number of constraints are imposed on the construction of links, to simplify the operation of numerous programs. First, each directory is created with two links, one called "." which points at the directory itself, and one called ".." which points at its parent. A parent directory is the one next closer to the root, except for the root itself, which is its own parent. Second, multiple parents are disallowed for a directory, to avoid loops in the tree structure and to make unambiguous what is meant by "next closer to the root". And third, a directory is never removed (the link from its parent erased) unless the directory contains only the links "." and "..", thus preventing the formation of orphan files.

Writes to a directory are disallowed, even to privileged programs, so that the integrity of the filesystem can be preserved; write permission for a directory merely grants the right to alter links. Similarly, execute permission means that the directory may be scanned while tracing a pathname, never that it can be executed. Reads work just as for plain files.

Note that directories are also like plain files in being unable to shrink. An erased link is left in place, for possible later reuse, but no attempt is made to shorten a directory by squeezing out erased links, or even by lopping off erased links at the end of the file. This can lead to noticeable performance degradation, as when hundreds of files are created in a directory, then later removed, leaving many erased links to skip over when scanning pathnames. The only fix is to create a new directory with the same parent and with new links to the remaining children, to remove the old, and to rename the new as the old.

It is also worth recording, but unlikely to matter, that the resident will not scan beyond the first 4,096 records in a directory.

SEE ALSO

filesystem, plain

NAME

dlog - incremental dump history

SYNOPSIS

/adm/dump

FUNCTION

/adm/dump gives a history of all incremental dumps performed. It consists of 24-byte records, whose format is:

bytes 0-19: the name of the filesystem device, left justified and NUL terminated.

bytes 20-23: the date of the dump.

A record is written to this file for each incremental dump performed.

SEE ALSO

dump

BUGS

The filesystem device should be a) a 64-byte pathname, or b) a 14-byte link in /dev, or c) the device major/minor number. Only the latest entry for a given device should be saved.

NAME

dump - dump file format

Dump files are produced by the dump command, for later use by the restor command, as a means of backing up the logical contents of a filesystem. (Both commands are described in Section IV of this manual.) They can be used to record the entire contents of a filesystem, or only those portions that have changed since an earlier dump date, in a space that is often much less than the filesystem proper. Since dump files are compatible across all implementations of Idris, and with UNIX/V6 dump tapes, they can be used as a means of communicating large numbers of files between UNIX and Idris systems.

A dump file consists of a dump header, a file map, and file header/data pairs. In the following description, unless otherwise stated, two byte quantities are unsigned short integers written less significant byte first; four-byte quantities are filesystem dates. The dump header consists of:

bytes 0-1: the number of (512-byte) blocks of inodes, 16 inodes to the block. Taken from the superblock of the dumped filesystem.

bytes 2-3: the number of blocks reserved for the filesystem, also from the superblock.

bytes 4-7: the date of the dump.

bytes 8-11: the date of last full dump. Incremental dumps consist of all files modified between this date and the date of the dump. Both dates are the same for a full dump.

bytes 12-13: the size in blocks of the dump file.

The file map consists of enough whole blocks to contain one unsigned short per inode in the dumped filesystem. The first entry corresponds to inode 1, etc. The value is interpreted as follows:

-1: file does not exist.

0: file exists, but not dumped. This is used for incremental dumps when the file is older than the date of the last dump.

otherwise: file exists, was dumped, and was one block shorter than the number given.

The file header/data blocks consist of a one block header containing a duplicate of the inode, followed by as many full blocks as necessary to store the actual file contents. Note that if the size field in the inode copy is different from the size as specified by the file map, the file was dumped with a "phase error". This may confuse a restor under UNIX, but is ignored by the Idris restor utility.

The dump header block and the file header blocks have the dump file block address in bytes 508 and 509, and a checksum in bytes 510 and 511. The

block checksums to the octal value 031415, when summed as an array of shorts, each written less significant byte first.

SEE ALSO

dlog, filesystem, tp

NAME

filesystem - Idris filesystem structure

FUNCTION

A filesystem is a data structure which, when stored on a block special device, can be administered by the Idris resident as a collection of directories and plain files. Idris requires at least one filesystem, called the root, to be available at system startup and ever after; additional filesystems may be "mounted", or spliced into the existing directory structure, and "unmounted" at will by the system administrator. Since filesystems are compatible across all implementations of Idris and with UNIX/V6 filesystems, they can be used as a means of sharing large numbers of files between UNIX and Idris systems.

A filesystem is organized as a series of 512-byte blocks. Block zero is unused and may serve as a bootstrap, for some machines, that can load stand alone programs from the filesystem. Block one is the "superblock", immediately followed by a variable number of blocks of "inodes", which is followed in turn by a variable number of data blocks. The superblock specifies these variable numbers (which cannot change during the life of a filesystem) and provides handy lists of available inodes and data blocks. An inode holds all the attributes of a file except its name(s); and a data block holds actual file contents, or an array of pointers used to locate other data blocks.

In the descriptions that follow, all one-byte numbers are unsigned char numbers, i.e., in the range [0, 256). All two-byte numbers are unsigned short integers written less significant byte first; such numbers are in the range [0, 65,536). All dates are four-byte unsigned long integers written as two such shorts, more significant short first [honest]; a date is the time in seconds since midnight 1 January 1970 GMT, which will suffice through the 21st Century.

The format of the superblock is:

bytes 0-1: the number of blocks reserved for inodes.

bytes 2-3: the total number of contiguous blocks used by the filesystem.

bytes 4-5: the number of free data/pointer blocks represented in the superblock, in the range [0, 100]. Zero means entire free list is exhausted, one means no more free blocks in the superblock array, two means one free block is present at the beginning of the superblock array, etc.

bytes 6-7: the block number of the first entry in the remaining free list, if this and the preceding number are nonzero.

bytes 8-205: the superblock array of free data/pointer blocks, as left justified short integers.

bytes 206-207: the number of free inodes represented in the superblock, in the range [0, 100].

bytes 208-407: the superblock array of free inodes, as left justified short integers. Free inodes not represented in this array must be located by scanning the actual inode array.

bytes 408-411: unused by Idris, reserved for UNIX administration.

bytes 412-415: date of last superblock modification. Used primarily to guess the date on system startup.

bytes 416-511: unused.

The remaining free list consists of zero or more entries, each written in a free data block. A free list block has the format:

bytes 0-1: the number of free data/pointer blocks represented in this entry, in the range [0,100]. Same convention as in superblock.

bytes 2-3: the block number of the next entry in the free list, if this and the preceding number are nonzero.

bytes 8-205: the array of free data/pointer blocks, as left justified short integers.

Inodes are 32-byte records, hence packed sixteen to the block, commencing with block 2. There must be at least one inode block; more than 4095 are unusable. Inodes are thus numbered from 1 to 65,520. Inode 1 is called the "root" of the filesystem and must be a directory, if the filesystem is to contain more than one file.

The format of an inode is:

bytes 0-1: the attribute or mode flags. If the octal 0100000 (most significant) bit is set, the inode is allocated; otherwise its remaining contents should be ignored. If the next two bits, masked by octal 060000, are: 000000, the entry is a plain file; 020000, the entry is a character special file; 040000, the entry is a directory; 060000, the entry is a block special file. If the octal 010000 bit is set, the file is "large", as described below; otherwise it is "small". The low twelve bits, masked by octal 07777, specify the access permissions for the file, which are described at length in conjunction with the chmod command in Section II of this manual.

byte 2: the number of "links", or directory entries within the filesystem, that should be pointing at this inode.

byte 3: the userid of the owner of the file.

byte 4: the groupid of the owner of the file.

byte 5: the size of the file in multiples of 65,536 bytes.

bytes 6-7: the size of the file in bytes, to be added to the preceding field to obtain the total size.

bytes 8-23: an array of eight short integers, usually used to hold block numbers as described below. For a character special or block special file, however, the first of these is taken as the major device number (times 256) plus the minor device number of the physical device to access.

bytes 24-27: date of last access to this inode. Not updated when accessing a directory in the process of scanning a pathname.

bytes 28-31: date of last modification to this inode.

For a small file, which is not a character special or block special file, the actual file contents are in the blocks pointed at by the eight block numbers. A block number of zero means no data block is present; it is permissible for a file to have unallocated holes, but there should be no blocks specified for bytes beyond the end of the file. Bytes 0-511 of the file are in block zero, bytes 512-1023 in block one, etc. A file over 4,096 bytes long must be large.

For a large file, each of the first seven block numbers points at a block of 256 block pointers, which point in turn to data blocks. The last block number, if nonzero, points at a block of pointers to blocks of pointers to data blocks.

Thus, a filesystem may consist of up to 65,536 blocks, containing up to 65,520 files. Each file may be up to 16,777,215 bytes long. These limits can usually be obscured, however, by mounting multiple filesystems as one directory tree and by accessing files through appropriate utilities.

SEE ALSO

block, character, directory, dump, plain

NAME

log - login history file

SYNOPSIS

/adm/log

FUNCTION

/adm/log is written by the /odd/log program to inform the who command about who has logged in and out, and about major accounting milestones such as date changes and system startups. It contains one 26-byte record for each accounting event, written in order of increasing time.

The format of each record is:

bytes 0-13: the pathname, in the /dev directory, of the tty file on which the login occurred. Left-justified and NUL padded.

bytes 14-21: the loginid of the active user, or "old:time" for the time just before a date change, or "new:time" for the time just after a date change, or "reboot:" for an initializing call to login (presumably at startup time), or "" for an inactive port. Left justified and NUL padded.

bytes 22-25: the time of the entry, as a filesystem date. Binary long integer in native byte order.

Note that this is not a text file and hence is not easily displayed except by the who command.

SEE ALSO

filesystem, tty, who

BUGS

Date should be in filesystem byte order, for portability.

NAME

mount - mounted filesystems

SYNOPSIS

/adm/mount

FUNCTION

/adm/mount is a file, administered by the /etc/mount command described in Section IV of this manual, used to record the filesystems currently mounted. Other programs use this file to trace pathnames or to inspect all mounted filesystems.

The file consists of 66-byte records, one for each filesystem, with the format:

bytes 0-1: the device minor and major numbers, of the mounted filesystem, in native byte order. The major number is multiplied by 256 and added to the minor number.

byte 2: the status flag. A null '\0' indicates no longer mounted, 'm' means mounted read/write, and 'r' means mounted read only.

bytes 3-65: the pathname of the node mounted upon, left justified and NUL padded.

Since /adm/mtab is not a text file, it is best inspected with the help of the /etc/mount command.

NAME

null - bottomless pit

SYNOPSIS

/dev/null

FUNCTION

/dev/null is a character special file that reports end of file on any read and that trashes anything written to it, without complaint. It is used by the shell as a safe place to redirect standard input (STDIN) for a background process that has not otherwise redirected its input. It is also a convenient place to send large quantities of output when all that is desired is some side effect of its production.

NAME

passwd - the system password file

SYNOPSIS

/adm/passwd

FUNCTION

/adm/passwd contains login information about each possible user; hence it is used extensively for checking and for mapping userid and groupid numbers to human readable loginids. It is a text file, with one line per user, where a line consists of colon separated fields with the following meaning:

FIELD	CONTENTS
0	loginid
1	encrypted password
2	userid
3	groupid
4	long form user name
5	login directory
6	login command

loginid is the name that the user types in response to the prompt from the login program. It should be one to eight characters, preferably printable, and distinct from all other loginids in /adm/passwd.

The encrypted password is an empty string, if the user has no password, or a twelve-character sequence generated by the passwd command. Placing anything else in this field is a sure way to prohibit logins with that userid -- a practice often followed with group name entries. Note that there is (intentionally) no obvious relation between the encrypted password and the secret password typed by a user, which is one to eight ASCII characters. Thus /adm/passwd may be safely read by all.

userid is a number, between 0 and 255, assigned by the System Administrator and stored as part of the attributes of a file, used to encode ownership and identity in just one byte of information. userid zero is reserved for the "superuser", a mythical being who is frequently granted special dispensation by the resident, including the power to cloud men's minds. Each userid should be unique within /adm/passwd, so that the ownership of files can be correctly reported and enforced. It is never necessary to know a userid outside this context, since all utilities refer to /adm/passwd to map userids to loginids.

groupid is a number just like userid, except that it designates the group membership corresponding to this loginid. The space of groupids is distinct from userids, so there can be up to 256 working groups who can share access to common files.

The long form user name is essentially unused, except to the extent that it serves to identify a loginid more fully.

The login directory is made the user's current directory on a successful login, just before the login command is executed.

The login command is the end result of the login process. For normal users it is some form of command interpreter (shell), usually "/bin/sh". The command may consist of quoted strings (single or double quotes), redirection of STDIN and STDOUT with '<' and '>' respectively, and other whitespace separated arguments. After processing redirection and eliminating the quotes, the login program executes the file named by the first argument, with all arguments passed to it as a parsed command line.

Setting up a password file is discussed, with examples, in Section IV of this manual.

NAME

pipe - pipeline pseudo files

FUNCTION

A pipe is a data connection that can be written as a sequential file on one end and read as a sequential file on the other; typically pipes are setup between cooperating programs by the shell. It is implemented as an unnamed plain file in some filesystem (determined by the Idris resident), with synchronization to keep the writer from getting too far ahead of the reader and to alert a waiting reader when data is written to the pipe.

Arbitrary eight-bit bytes can be passed through a pipe, and arbitrary numbers of bytes may be written or read at a time. There will never be more than 4,096 bytes stored in the pipeline, however, and the reader is often given just what is available, if data is present but less than what is requested. If all writers close their end of the pipe, then an end of file will be delivered to all readers once the pipe drains. If all readers close their end of the pipe, then the "broken pipe" signal is sent to all writers.

Any attempt to perform direct access (seeking) on a pipe will report failure, since the positioning information (seek offset) is commandeered for internal use by the pipe administrator.

If the system is brought down unexpectedly, as on loss of power, while a pipe is active, it may appear in its host filesystem as an allocated inode with no links to it. Its resources may be reclaimed by clearing the inode and reclaiming the lost free blocks (see fcheck and icheck in Section IV of this manual).

SEE ALSO

filesystem, plain

NAME

plain - plain files

FUNCTION

A plain file is the simplest type of file, one residing in a filesystem and not being used as a directory. It is implemented as an "inode", or central repository of status information, with zero or more blocks of data (and blocks of pointers to data blocks) defining its contents. It appears to the world as just an ordered sequence of eight-bit bytes, whose contents are arbitrary and whose length is determined by the highest numbered byte ever written; unwritten intermediate bytes read as zeros. It is reachable by one or more "pathnames" using the filesystem directory structure.

Plain files can be accessed directly (random reads or writes), an arbitrary number of contiguous bytes at a time. There are, of course, performance benefits in accessing plain files as 512-byte blocks, and many programs take advantage of this benefit. Plain files are truncated to zero length when "created" (first made to exist or whenever completely rewritten), then are extended simply by writing to ever higher byte positions. A plain file can be as long as 16,777,215 bytes, assuming there is sufficient space on its host filesystem.

SEE ALSO

directory, filesystem

NAME

print - printable file restrictions

FUNCTION

A print file is a text file, with the added restrictions that it contain no funny characters. It also should contain no lines too long for the printing device, but that is a restriction that varies widely among devices and hence is not enforced by most text processing programs; each device is at liberty to truncate or fold long lines as it sees fit.

A funny character is, basically, one that the typical printer can't deal with. In the ASCII character set, the octal codes [0040, 0177) cause certain graphics to print, so they're all right. There are also defined codes for "whitespace", or standard actions:

CODE	SYMBOL	KEY	MEANING
0007	'\7'	ctl-G	bell
0010	'\b'	ctl-H	backspace
0011	'\t'	ctl-I	horizontal tab
0012	'\n'	ctl-J	line feed (newline)
0013	'\v'	ctl-K	vertical tab
0014	'\f'	ctl-L	form feed (newpage)
0015	'\r'	ctl-M	carriage return

Of these, only the newline is reliably nonfunny, others being provided at the whim of the hardware designer, or emulated at the discretion of the software. (Most terminals provide at least some of these codes as special keys, so it is not always necessary to type the control sequence shown to input the character. On the other hand, because most terminals provide a big fat carriage return key and a skinny little line feed key, the Idris resident is easily persuaded to map carriage returns to newlines on input.)

SEE ALSO

ASCII, text

NAME

salt - encryption salt file

SYNOPSIS

/adm/salt

FUNCTION

/adm/salt is used, by programs that must encrypt a password, as a starting value for the encryption process. Up to eight characters are read from /adm/salt, if possible, to overwrite the builtin string "password", which is then exclusive-ored, on a character by character basis, with the user's loginid. The resulting mish mash is then encrypted. Including the loginid with the salt means that two users on the same system may elect the same password and never know it. And by varying the contents of /adm/salt among different systems, users may indulge in the same password multiple times and not have their bad habits revealed. Moreover, if the salt has no read permission, the work needed to crack an encrypted password increases dramatically.

Since all programs that must encrypt a password tend to be privileged, it is thus permissible as well as advisable to deny read permission to all but the superuser for /adm/salt.

NAME

stty - predefined terminal attributes

SYNOPSIS

/adm/stty

FUNCTION

/adm/stty is the text file used by the standard utility stty, in conjunction with the -type* flag, to set terminal attributes to a predefined set. Each line consists of the terminal type, used to match *, followed by attributes written just as for the stty command.

SEE ALSO

stty(II)

NAME

text - text file restrictions

FUNCTION

The commonest restriction imposed on a file or data stream is that it be a "text file". Colloquially, this means that it is human readable -- it can be copied to a terminal or printer without doing violence to either.

More precisely, a text file is presumed to consist of zero or more "lines", where a line consists of zero or more characters followed by a newline (usually written "\n" when it must be made visible, and usually causing the same effect as a typewriter carriage return when displayed). This implies, of course, that a non-empty text file must end in a newline; and indeed, any text processing program reserves the right to deal with an "incomplete last line" (one not ending in a newline) by: a) discarding it, b) leaving it incomplete, or c) appending a newline to it.

No program that processes a text file is obliged to deal with a line longer than 512 characters, counting the terminating newline; on the other hand, all programs are obliged to deal with lines at least this long. If a line is longer than 512 characters, a text processing program reserves the right to: a) truncate it and leave it incomplete, b) truncate it and append a newline, c) fold it by inserting newlines, or d) process it completely.

There are no restrictions on the character set that may appear in a text file, although clearly the newline has a special meaning. Nonprinting characters of all sorts, including NULs, should be properly processed by all programs designed to manipulate text. If the file is indeed destined to be printed out, however, additional restrictions apply, as discussed under print files.

SEE ALSO

ASCII, print

NAME

tp - tape archive format

FUNCTION

The standard utility tp administers a file, typically written to tape, that records the attributes and contents of a number of files. Since tp files are compatible across all implementations of Idris, and with UNIX/V6 tp tapes, they can be used as a means of communicating small numbers of files between UNIX and Idris systems.

A tp file is organized as a series of 512-byte blocks. Block zero is unused and may serve as a bootstrap, for some machines, that can load stand alone programs from the tp file. Blocks 1-62 contain 496 directory records, each containing 64 bytes in the format:

bytes 0-31: the NUL-terminated pathname of the archived file. If the first two bytes are NULs, the entry is empty.

bytes 32-33: the attributes or mode of the file, taken from its inode, written less significant byte first.

byte 34: the userid of the file owner.

byte 35: the groupid of the file owner.

byte 36: unused.

bytes 37-39: the size of the file in bytes, written most significant byte, then least significant byte, then middle byte [sic].

bytes 40-43: the date of last modification, written as a filesystem date.

bytes 44-45: the tp file relative block number of the start of the file contents.

bytes 46-61: unused.

bytes 62-63: the checksum, such that the record, taken as 32 short integers written less significant byte first, sums to zero.

Blocks 63 on up hold file contents, an integral number of blocks for each file, taken in order. If a directory record is empty, its contents occupy zero bytes.

SEE ALSO

filesystem

NAME

tty - terminal files

FUNCTION

A tty file is a direct connection to a peripheral device, which is a character special file employing a standard protocol for supporting "teletype" (terminal) input/output. Reading a tty file waits for keystrokes from a terminal keyboard; writing it sends characters to the terminal display; positioning requests (seeks) are simply ignored.

A tty is capable of editing its input on a line by line basis, of echoing its input to its output for full duplex terminal operation, of throttling its output under keyboard control, of inserting tailored delays in its output for sluggish displays, and of turning certain keystrokes into signals for controlling active processes. The stty command can be used to display or alter the operating mode of any tty file in a standard fashion.

An important special feature of tty files is their ability to send signals to processes associated with them. When a tty file is "opened", i.e. requested for use by a login process (typically), it makes the process wait until a connection is actually established (someone dials up, or hooks up a terminal, or turns it on) -- assuming the hardware is capable of detecting such a change. It then appoints itself "controlling tty" for the process, unless some other tty has already claimed the honor. As a controlling tty, it interprets certain keystrokes as immediate requests to broadcast associated signals to that process and to any descendants it may have; thus, a misbehaving process can be terminated, or a long editor display can be cut short.

Using stty, a tty can have its baud rates changed for input and/or output, assuming the physical device can respond to such requests; it can also be placed in "raw" mode. A raw tty outputs eight-bit bytes transparently (unchanged), with no delays, and accepts all keystrokes without interpretation as eight bit bytes. The only control left is to break the connection (hang up, or unplug the terminal, or turn it off), which causes a hangup signal to be sent to all processes under control of the tty -- assuming the physical device can detect a broken connection! Thus, raw mode should be left to programs that know what they are doing.

When not in raw mode, a tty cares about a number of other options:

parity - The state of the "eighth bit" accompanying seven-bit ASCII codes can be checked on input (bad bytes discarded) and determined on output (to please parity checkers in connected devices). Four combinations are provided: 1) accept any parity, generate zero parity bit; 2) accept even parity (sum of all bits in the byte is even, counting parity bit), generate even parity; 3) accept odd parity, generate odd parity; and 4) accept any parity, generate one parity bit. In all cases, bytes input have their parity bits set to zero.

map CR - A carriage return can be mapped on input to a newline, and a newline can be mapped on output to a carriage return followed by a line feed (newline). The former service is needed for terminals that provide a large return key, rather than a large newline key; the

latter is needed for the many terminals that require both codes to advance to the next line.

expand HT - Each horizontal tab output can be expanded to the appropriate number of spaces to simulate tab stops every four columns. Useful on terminals that can't interpret tabs.

map uppercase - Each uppercase letter can be mapped to lowercase on input, unless preceded by the escape character '\'. Also, certain poorly supported characters can be generated on input by two character escape sequences: "\'" becomes '`', "\(" becomes '{', "\!" becomes '|', "\)" becomes '}', and "\^" becomes '~'. With this option, each lowercase letter output is mapped to uppercase, and each of the above mappings is reversed on output. Needed on primitive terminals that support only one case of letters.

echo - Each input character can be written out immediately. Needed to support full duplex operation with decent human feedback.

hangup on close - The hardware device can be encouraged to hangup a phone line, assuming it has the ability to do so, when "closed", i.e. when a user logs out (typically). Used to minimize the chance of security breaches.

delay - Certain characters can be followed on output by delays, to give mechanical devices time to move, typically. The supported delays, in units of 1/60 second, are: horizontal tab [0, 4, 8, 12], carriage return [0, 4, 8, 12], newline [0, 4, 8, 12], backspace [0, 8], form-feed or vertical tab [0, 128].

erase - The character typed to erase the last character from the input, normally a backspace, can be changed to any other character, or disabled.

kill - The character typed to kill an entire line of input, normally '@', can be changed to any other character, or disabled.

Note that any pending input is discarded when a tty is first opened or when its mode is altered by stty; the former is convenient for eliminating trash at startup time, but the latter can come as a surprise to those who have grown accustomed to the convenience of type ahead. Programs that seek keyboard reassurance, with messages such as "are you sure? ", also perform an internal stty to be sure of getting only timely answers.

A tty file tends to deliver up a line at a time when read, with three exceptions. If the file is in raw mode, as many characters as possible are delivered. If the number of characters requested is less than the total line, a partial line is delivered. And if an EOT (ctl-D) is typed, that character is discarded and any partial line is delivered, even if it has no characters. Thus, typing an EOT at the beginning of a line serves to report an end of file to the reader, a process that can be repeated any number of times during a terminal session. Reporting end of file (by typing EOT) to the shell is the conventional way of logging out.

The complete set of keystrokes with special meaning is:

EOT - (ctl-D) Discard EOT and deliver partial line. Used to simulate end of file.

BS - (ctl-H or backspace) Discard last character on current line, if any, along with BS; output as BS, space, BS. Possibly delay. Default erase character, can be altered.

HT - (ctl-I or horizontal tab) Possibly expand to 1-4 spaces on output. Possibly delay.

NL - (ctl-J or newline) Deliver current line, terminated by newline. Possibly map to carriage return, line feed on output. Possibly delay.

VT - (ctl-K or vertical tab) Possibly delay.

NP - (ctl-L or newpage) Possibly delay.

CR - (ctl-M or carriage return) Possibly map to newline on input. Possibly delay.

DC1 - (ctl-Q) Discard DC1 and resume suspended output.

DC3 - (ctl-S) Discard DC3 and suspend output.

FS - (ctl-\) Discard FS and send quit signal.

'@' - Discard current line, if any, along with '@'; output as '@', NL. Default kill character, can be altered.

'\' - If next character is erase or kill, discard '\' and take next literally; if next character is uppercase and mapping is in effect, discard '\' and don't map; otherwise take '\' literally.

DEL - (or rubout) Discard DEL and send interrupt signal.

BREAK - same as DEL.

If a tty file is connected to a source of high speed input, such as another computer, it runs the risk of being overwhelmed with input. Thus, if the reader gets behind by approximately 128 characters, a ctl-S is automatically output; the balancing ctl-Q is output only when the reader has caught up more. If this fails to stop the source of input, a more drastic action is taken -- if the reader gets behind by 256 characters, all input is discarded without remark. Care must thus be taken when connecting Idris directly to inconsiderate machines.

SEE ALSO

character, stty(II)

NAME

where - system identification psuedo file

SYNOPSIS

/dev/where

FUNCTION

/dev/where is a character special device which reads out a text string from the resident, typically to advertise the presumed location and nature of the resident CPU. /dev/where can also be rewritten.

The text strings /dev/where deals with are at most 40 bytes in length and terminate, when read, with the first newline.

BUGS

Length should be 64 characters.

NAME

who - login active file

SYNOPSIS

/adm/who

FUNCTION

/adm/who is written by the login program to inform the who command about who is currently active on the system. It contains one 26-byte record for each login port, as determined at system initialization time (described in Section IV of this manual).

The format of each record is the same as for the history file /adm/log, except that no entries are made for time changes or system startups.

SEE ALSO

log

NAME

zone - time zone information

SYNOPSIS

/adm/zone

FUNCTION

/adm/zone is used by all programs that need to display system time in terms of local usage. It is a text file whose first ten characters are interpreted as:

bytes 0-3: four decimal digits for timezone, expressed in hours and minutes west of Greenwich Meridian (Boston MA, New York NY, and Secaucus NJ are all in zone "0500").

bytes 4-6: three ASCII characters for name of timezone during standard time (e.g. "EST").

bytes 7-9: three ASCII characters for name of timezone during daylight savings time (e.g. "EDT"). If this field is absent, or begins with a newline, it is assumed that the U.S. daylight savings laws do not apply, so standard time is always used.

If the file cannot be read and understood, builtin defaults are used. As shipped, these correspond to the parenthetical examples given, i.e. the file contents "0500ESTEDT".

IV. System Administration Guide

NAME

Scope - the job of System Administrator

FUNCTION

The information in this section is primarily for the use of a System Administrator -- that person who is responsible for the day to day care and feeding of an Idris system. No knowledge of programming is required for the job; all the work is done using existing utilities and predetermined protocols.

It is assumed that still another person, known as the System Generator, has already installed Idris. Setting up bootstrap procedures, using the bootstrap components shipped with Idris, is the job of the System Generator. Configuring the resident for the local set of peripherals, using the device handlers shipped with Idris, is also the job of the System Generator. All these functions are described in the Idris Interface Manual for the appropriate host machine.

Once a system is generated, however, there is much that can and should be done. Idris is a multi-user system; someone must issue loginids and allocate space in the directory hierarchy. Idris supports multiple demountable filesystems; someone must make, maintain, and backup those filesystems. And Idris is highly configurable; someone must decide what adaptations are most appropriate for local usage. Thus, the System Administrator.

This section contains several essays on the topics in the preceding paragraph, to give a more detailed overview of the job. It also contains manual pages for utilities not generally useful to (or safely used by) more innocent customers. Some of the utilities documented here reside in the standard directory /odd; they are not designed to be invoked directly by people. The remaining utilities documented here reside in the standard directory /etc; the wise System Administrator will include this directory in the standard search path for the superuser loginid, conventionally called root.

The humble System Administrator will use these utilities, and superuser powers, very carefully.

NAME

Filesystem - a guide to the standard shipped filesystem

FUNCTION

The Idris system is organized around a root filesystem containing a dozen or so directories. Each customer is at liberty to add arbitrary amounts of structure to the shipped system, or even to alter the basic directory structure in a number of ways. It is strongly recommended, however, that this basic structure be left intact: new releases will be much more easily incorporated, and the shipped structure is reasonably flexible as it stands. This essay describes the major directories of the shipped filesystem, with some hints as to their underlying rationale.

All files are owned by the root, or superuser, to begin with, so that they may be better protected from inadvertent modification by innocent users. The shipped system denies permission to remove standard directories and utilities, to modify standard files, or even to add files to standard directories. Access is further restricted in two critical cases: the per-system encryption salt file /adm/salt is read protected, to raise the cost of guessing passwords; and the mailbox directory /adm/mail denies scan permission, to keep inter-user mail confidential.

Otherwise, the general philosophy is to permit as much access as possible, short of endangering system integrity. Most utilities can be read, for instance, and not just executed. The password file may be read, because the passwords themselves are encrypted. A considerably tighter system can be had just by restricting access permissions in a number of critical places; but a certain looseness seems to be more productive, in all but the strictest of secure environments.

Standard files are dealt out among about a dozen directories:

/adm the administrative directory. Here may be found files, frequently read and written, that record the activity of the system. Items such as the dump, mount, and login history are recorded in /adm, as well as information on how the system should configured at startup and who may login.

/adm/mail the inter-user mailboxes. Each loginid that has ever received mail gets a subdirectory of the same name within /adm/mail. As mentioned, all but the superuser are denied the ability to browse among the undelivered letters.

/bin the set of standard utilities. All the utilities likely to be of use to every user are collected here. They are documented in Section II of this manual.

/dev the character and block special files. Although a special file can be placed anywhere (by mkdev), by convention all are concentrated in the /dev (for "devices") directory.

/etc the set of system administration utilities. All the utilities likely to be of use only to the System Administrator are collected here. They are documented later in this section.

/etc/bin the set of programming utilities. All the utilities likely to be of use only to people writing new programs are collected here. They are documented in Section II of the C Interface Manual for each machine.

/lib the programming libraries. All the libraries, object headers, and source header files, likely to be of use only to people writing new programs, are collected here. They are documented in the Pascal and C Manuals, as well as the various Idris Interface Manuals.

/odd the odd utilities. All the utilities likely to be invoked only by other programs, and never directly by people, are collected here. They are documented in this section, or in the various C Interface Manuals.

/stage the object file staging area. All the binary object files needed to rebuild the system from scratch are collected here. The Install guide shipped with Idris, plus Install scripts sprinkled throughout the subdirectories of /stage, document how all the utilities, bootstraps, and the resident are put together. If space is at a premium, this subdirectory may be removed from the on line filesystem (but keep a backup copy, by all means), since it is used only at installation time and for system generation.

/tmp the temporary file directory. All utilities that require temporary files are encouraged to place them in /tmp, and to remove them when done. Thus, /tmp is the only standard directory where users are permitted to create and remove files. The tidy System Administrator will purge /tmp from time to time, since not all utilities are perfect at cleaning up after themselves. It is even a good idea occasionally to remove /tmp itself and recreate it, since it may have grown large and unwieldy during earlier heavy usage.

/usr the set of personal directories. By convention, each user is given a subdirectory under /usr whose name matches the user's loginid. /usr, as shipped, is empty.

/x, /y, /z assorted mounting posts. A filesystem must be mounted on an existing inode, so these are provided as convenient places to hang filesystems, in the absence of a more compelling name to suit the contents.

You can get a map of every file on the system, from time to time, by typing:

```
# ls +dal /
```

which recursively descends the entire directory tree, printing all entries. Do this early on, to a hard copy device if possible, so that later you can repeat the operation and highlight accumulated trash.

BUGS

A hierarchical filesystem is a great way to organize information. It is also a great place to lose data.

NAME

Login - adding new users to the system

FUNCTION

To add a new user to the system, all you really have to do is add another line to the file /adm/passwd. Pick it up with the editor e, modify the file, then put it back; instantly the user is "installed". There are usually a few other items that go along with this, however.

The "password" file /adm/passwd contains one line of text for each potential user. Separated by colons, its entries left to right are:

loginid the name by which the user is to be known, within the system. One to eight characters, preferably descriptive and easily typed. Some conventions followed are: initials (pjp), surnames (jones), first names (bob), or functional classifications (admin). As shipped, the only entry is root, for the superuser; this convention borders on being universal.

password the encrypted password supplied to the passwd utility. Twelve characters, from a 64-character alphabet. An empty field means no password; anything else not filled in by passwd is de facto impossible to reproduce, and hence means no login permissible. Unless no logins are to be permitted, this field is best left empty, to be set by the user invoking passwd. The password supplied to passwd, or to a subsequent login, is one to eight characters, preferably of mixed case and/or punctuation, on the long side, and imaginative (don't use names of kids, pets, or cars). Passwords should be used on any system that has public access, particularly for the superuser.

userid a decimal number in the range [0, 256), zero being the superuser. By convention userids 1-9 are reserved for pseudo users (also known as gremlins and daemons), so numbers should be issued from 10 on up. A large community must perforce double up on userids, sadly.

groupid a decimal number in the range [0, 256). By convention, groupid zero is a privileged group occupied only by the superuser, to avoid inadvertent security leaks. The first line in /adm/passwd with a given groupid is taken as the name of the group, often a userid written all in uppercase letters. Many installations place everyone but root in group 1, effectively discarding group access as a separate class of protection.

long name an unused field, to tell the truth. By convention, however, this is the place to enter a more descriptive handle for the user; and it may be used as such in future.

home directory the directory /odd/log makes current before starting your shell. As mentioned, this is conventionally a directory under /usr, whose name matches your loginid.

shell the command line that /odd/log invokes as the last step in the login process. Typically it is some invocation of the standard shell, such as

```
/bin/sh -i /adm/.login .login -
```

This invokes an interactive shell, which first runs the standard startup script /adm/.login, then your personal startup script .login (in your home directory), then prompts for input from the keyboard. The editor e is also a satisfactory shell, particularly for beginners; and there are a number of one-shot services that some systems like to support:

```
sync::0:0::/:/bin/sync
who::0:0::/:/bin/who
```

You can also install restricted shells, such as demonstration programs.

If a conventional entry is made in /adm/passwd for a new user, then the user must be given a home directory under /usr, with the right to change its access permissions:

```
# mkdir /usr/loginid
# echo "mail -q" > /usr/loginid/.login
# chown loginid /usr/loginid /usr/loginid/.login
```

The personal startup script .login shown here merely reports "you have mail" or "no mail"; many other things can be provided at the whim of the user. The /adm/.login file shipped displays the host machine (/adm/where), and any message of the day (/adm/motd), then sets the shell variables X (search paths for execution) and H (home directory). It too is easily changed to support different system wide functions.

If the system has a steady turnover of users, it is best to recycle userids as seldom as possible. Better a file should show an owner of "20" than be falsely ascribed to some unwitting newcomer.

NAME

Startup - system startup

SYNOPSIS

/odd/init

FUNCTION

At system startup time Idris creates a process 0, which becomes the memory manager or "swapper", and a process 1, which initiates everything else. As shipped, Idris always has process 1 invoke the file /odd/init, with argument zero the string "init", and with no other arguments; all files are closed and the userid is that of the superuser. If process 1 exits, it is restarted the same way, as superuser with all files closed. Note that this startup environment differs from the standard one provided by the shell, which supplies the opened files STDIN, STDOUT, and STDERR.

Four utilities are provided which work properly as process 1:

sh the standard command language interpreter, or shell, which reads multiple commands from a tty, in this case the file /dev/console.

hsh an administrative utility which executes only one command, again from /dev/console, then expires. Appropriately enough, this is called the "half shell"; it relies heavily upon the process 1 loop for continuity.

log an administrative utility that allows restricted access to the system by authorizing preidentified users (listed in /adm/passwd); it too uses /dev/console. In addition, it performs various system accounting functions.

multi an administrative utility which can be instructed to perform initialization functions and to set up a multi-user environment (by obeying the commands in /adm/init).

Using the shell, sh, as /odd/init provides a single user system with the full power of its command language, including pipelines and background processing. The half shell, hsh, provides only a subset of the shell facilities; it doesn't permit pipelines or background processes. When the machine resources available are constrained, the half shell is convenient because it makes less demands and is smaller in size. In particular, a system can be configured with a read-only root filesystem and no swap device if the half shell is run as process 1.

log also provides just a single user system, but permits restricted access to different portions of the system by authorized users. Each user can be assigned a different startup utility, not necessarily the shell; a "demo" login, for example, may run a canned script, with only simple interaction accepted.

The most frequently used startup utility, multi, can be tailored to perform all sorts of initializations at startup, then turn on an arbitrary number of ports for multi-user operation. It can, for example, check system integrity automatically, mount standard filesystems, and/or ensure

that the date is properly set. multi is often configured to start the system in one-user mode for controlled access, allowing the system administrator to check the system before allowing others to login. There is even provision for multi to tear down an active system and restart, thus allowing graceful shutdown even in the presence of uncooperative users.

INTRODUCTION TO MULTI

When multi is invoked by the resident, it reads the text file /adm/init and memorizes it. This file is a script for multi to follow while establishing the multi-user environment requested, and while maintaining that environment ever after. It is composed of lines of text, each called a <command line>, which are read and executed sequentially. The basic structure of a <command line> is:

<control character> <pathname> <arguments>

The <control character> tells multi what to do with the line, and <pathname> is the absolute pathname of the utility to run with whatever <arguments> (which may include flags and filenames) are appropriate for that utility.

An uppercase <control character> signifies that the called utility is to be invoked with STDIN, STDOUT, and STDERR opened to /dev/console, so that interactive input can be obtained and/or error messages may see the light of day. A lowercase <control character> signifies that the called utility is executed with no files opened, and hence with no controlling tty. (The first tty file a process opens becomes the controlling tty for it and all of its descendants.) Thus, the utility (and all of its descendants) can be controlled by some other tty, or even left as a free agent.

The possible control charaters for multi are:

```
w or W  = run once, wait for completion
s or S  = run once, don't wait for completion
m or M  = restart after each exit, don't wait
k or K  = kill any existing process for this command line
other   = ignore this command line
```

If the command line begins with a 'w' or 'W', the specified utility is executed once and, upon its completion, multi proceeds to the next command line. With the 's' or 'S' control character, multi will begin execution of the current command line and then move on to other things before its completion. For any utility that needs indefinite re-execution, such as log, 'm' or 'M' should begin its command line. 'k' or 'K' is used to kill a troublesome process, such as a log connected to a port that has gone crazy, started earlier by multi. The System Administrator simply edits /adm/init to change the previous <control character> for that line to 'k' or 'K', then types the (privileged) shell command:

```
# kill -1 1
```

This sends a "hangup" signal to process 1 (multi), who takes the signal as a request to reread the control file, comparing its previously remembered contents with the current /adm/init. Any lines that change cause the associated process, started by multi, to be killed and the new action initiated. In the case of 'k' or 'K' no new process is started.

Within the /adm/init file, a command line should never be deleted, since it really matters to multi that the lines present correspond to the original /adm/init file. Thus 'k' or 'K' serves as a place holder until such time as the line can be reinstated. Needless to say, such modifications to /adm/init should be performed with caution.

Any control character other than those mentioned above does nothing, except to serve as a place holder at system startup.

After the <control character> on each line, multi expects a space followed by the absolute pathname (beginning with a slash '/') of the utility to be executed. No attempt is made to search alternate directories, as the shell does, to locate a utility. For example, to invoke the standard utility sh and wait for it to complete, you would write the command:

```
W /bin/sh -i
```

This accepts commands interactively from /dev/console until a ctl-D is typed.

A WALK WITH MULTI

Here is a line by line walk through a typical /adm/init file, one used to set up a multi-user Idris system, to help you better understand its possibilities. Line numbers have been prepended to each of the command lines for ease of reference; they do not appear in the actual file. The file is:

```
 1 W /bin/echo "set the date"
 2 W /bin/sh -i -X"|/bin/|/etc/|/etc/bin/"
 3 W /bin/echo "mount:"
 4 W /etc/mount -i /dev/rm2 /sys /dev/rm6 /tmp
 5 W /bin/rm -rs /tmp
 6 W /bin/echo "log:"
 7 W /odd/log -boot
 8 W /bin/echo "sync:"
 9 m /bin/sync -30
10 W /bin/echo "logins:"
11 m /odd/log tty0 -s1200 -s300 -unum0  -- dz0 --
12 m /odd/log tty1 -s1200 -s300 -unum1  -- dz1 --
13 o /odd/log tty2 -s1200 -s300 -unum2  -- dz2 --
14 m /odd/log tty3 -s300 -s1200 -unum3  -- phone1 --
15 k /odd/log tty4 -s300 -s1200 -unum4  -- phone2 --
16 m /odd/log console -s1200    -unum5  -- console --
```

Because the date is only approximately known at startup time (from the last time the root filesystem superblock was modified), it is usually necessary to

set the date before permitting others to login; hence the first command line is a reminder to the system administrator to set the date. With the 'W' control character, multi opens STDIN, STDOUT and STDERR in preparation for the output of the utility echo. The string of characters between the double quotes will be output to STDOUT as:

set the date

and then multi moves to the next command line.

The Startup Shell

With line 2, the sh utility is started on /dev/console. This gives the System Administrator a chance to set the date, as requested, and possibly to check the integrity of the various filesystems before other users have system access. All the output is directed to STDOUT or STDERR unless otherwise directed by the shell command being executed. The control character W informs multi to remain at this command line until this initial sh is exited (by typing ctl-D).

The absolute pathname invokes sh in an interactive mode as specified by the -i flag. The -H/ flag sets the root directory as the "home directory", used by the cd command as a default argument. With -X the "execution path" is defined, i.e. the sequence of directories in which the shell will search for utilities. The search will proceed through the directories listed inside the double quotes, as follows:

1) The first vertical bar with nothing before it indicates that the current directory should be explored first.

2) If an executable file of that name is not found there, the search continues by prefixing "/bin/" to the utility name.

3) Each of the remaining prefixes, between vertical bars, is then tried in turn.

4) If none of these directories contains the command, only then will the shell type its "not found" message and give up.

There is much much more the sh utility can do. In Section I of this manual, there are two separate tutorials about the shell, and in Section II there is a detailed manual page, all of which you should consult for more information on this versatile utility.

Other Initializations

Once the initial shell is exited, multi proceeds to execute line 3 of the control file, which is a reassuring message to the System Administrator that multi has progressed far enough to begin mounting standard filesystems. echo simply types the line:

```
mount:
```

then exits, permitting multi to proceed. With line 4, multi once again waits until the mount utility is exited, as indicated by the 'W'. mount initializes the file /adm/mount, courtesy of the -i flag, causing a new mount table to be created with an initial entry for the root filesystem, then goes on to mount two other filesystems. The /adm/mount table is used by mount and other utilities to trace pathnames or to inspect all mounted filesystems.

The block special device /dev/rm2 presumably contains a filesystem which is always to be reachable through pathnames starting with "/sys", and device /dev/rm6 presumably belongs at "/tmp". Since /tmp is heavily used by the editor, compilers, and other popular utilities, it may improve performance to devote a separate device just to hold temporary files, as is done here. Line 5 clears out any detritus left in /tmp, perhaps by a utility caught unawares by early termination or system shutdown.

The line "who:" is output to the console with command line 6 to announce the process that is to be carried out by line 7. This command line initializes the file /adm/who, because of the -boot flag, to say that nobody is currently logged onto the system, then appends a "reboot:" entry to /adm/log, if it exists. An accounting of major milestones such as user login activity, date changes, and system reboots (startups) is written, by the /odd/log utility, to /adm/log. If the System Administrator wishes such logging activity to take place, the file must be created by a shell command such as:

```
echo > /adm/log
```

since /odd/log will only write to the file if it already exists. Equally, logging can be disabled by removing the file.

Each of these files contain one 26-byte record for each accounting event; the file formats are similar except that no entries are made in /adm/who for date changes or system reboots. See Section III of this manual for more details.

Note again that the uppercase control character opens STDIN, STDOUT, and STDERR so any problems can be reported to the console.

Line 8 proclaims "sync:", again for reassurance. The sync utility ensures synchronization between main memory and all disk images by forcing the resident to update all inodes and disk buffers that have been modified and not yet written back to disk. This utility is an important safety measure against loss of data, or loss of filesystem integrity, should the system be halted unexpectedly. The flag -30 requests continuous operation of sync, with a thirty second interlude between updates.

The sync utility de-optimizes the system by forcing it to perform "unnecessary" updates. By modifying the length of time between automatic updates, the System Administrator can balance the need for safety of data with the need for speed of processing. Another safety feature provided by command line 9 is the multi control character 'm', which ensures that if sync should be killed inadvertently, it will be restarted.

Starting Multiple Logins

The final phase to a multi-user startup is announced with command line 10 echoing:

```
logins:
```

Command lines 11 through 16 start multiple instances of the log utility, on various tty devices, to allow multi-user access and to keep track of user activity on the system. Each command line begins with the 'm' control character, to start the log utility without waiting for completion, and to restart log when that process exits (usually by the descendant shell exiting at logout time).

At the system level, log is responsible for changing ownership of the tty file (to restrict read access by other users and to give the user control over message delivery), and for writing accounting information into the files /adm/who and /adm/log (if it exists). The -unum# flag specifies which slot in /adm/who should record the login status for that tty. The file /adm/log, on the other hand, is more like a telephone log -- for each login or logout, a record is appended giving the tty, loginid, and the time. Thus, the who utility can display the current users of the system by reading /adm/who or, when the -h flag is specified, it can display the recent history of system activity by reading /adm/log back to front.

log is also responsible for setting the initial tty status, much like stty, and possibly adapting to different speeds required by various terminals that might be connected to that tty port. Each of the speeds specified by a -s* flag is tried, in circular order, until a login is successful. For the range of possible speeds and other settable terminal attributes, see the log and stty(II) manual pages.

The comments at the end of each log command line are permitted because log doesn't look past its flags, and "--" terminates any flag list. They conventionally serve as reminders as to what type of equipment is associated with each of the logical ports (tty files). Note that, in this example, line 15 is currently disabled, with the 'k' control character, and that line 16 is your old friend /dev/console, now reduced to just another login port.

OTHER POSSIBILITIES

If you set up your system much like the example above, it will work quite well. It is designed to provide a general computing environment for one or more simultaneous users. But there are many other possibilities for tailoring to specific needs. Here are just a few suggestions:

Have the console run as a shell:

```
m /bin/sh -i -H/ -P"## " -X"/|/bin/|/etc/|/etc/bin/"
```

The console is thus always "logged in", without appearing in the list displayed by the who utility. Most useful for a console in a secure computer room.

Verify filesystem integrity automatically at system startup, as with the script:

```
/etc/icheck > /dev/null /dev/rm0[2356] \
|| echo "Filesystem damage, call trouble number"
```

You can even have the system automatically deal with the commonest blemishes, but this is somewhat more adventuresome.

Make a /tmp filesystem from scratch on each startup:

```
W /etc/mkfs -s4400 /dev/rk1
W /etc/mount -i /dev/rk1 /tmp
```

This provides a nice clean scratch area on each system startup, at the cost of making /tmp highly volatile.

Bring the system straight up in a dedicated application. For a turnkey system with automatic hardware bootstrap on power up, it is quite possible to configure Idris to initiate system startup literally at the turn of the key. With automatic self test and damage repair, as described above, and with the ability to start anything with multi, and with the ability to give different loginids different capabilities with log, a very robust dedicated application can be supported.

BUGS

Care should be exercised in modifying /odd/init, or /adm/init if multi is used, because the system may not survive startup if these are sufficiently corrupted.

NAME

Shutdown - taking the system down

FUNCTION

An operating system is an egomaniac -- it assumes that it owns the whole world and that it will live forever. Since neither of these assumptions is ever valid, some means must be devised for coaxing a system into relinquishing its reign in a more or less graceful fashion.

Idris is actually more humble than many systems. Its biggest problem is that it tries very hard to optimize disk reads and writes; it will defer as long as possible actually writing a block back to disk, in the off chance that some more modifications might be made before the write must occur. Thus, if the system is taken down abruptly, there is a real danger that the disk is not in a consistent state. That's what the sync utility is for:

> ALWAYS MAKE SURE THAT sync HAS BEEN INVOKED BEFORE SYSTEM SHUTDOWN, OR DISK IMAGES MAY WELL BE CORRUPTED.

Common practice is to start a perpetual sync running at system startup time, dispatched by multi. If the system has been idle for longer than the repeat time specified to the perpetual sync, then you can be sure that the disks are stable. If you are running "single-user", however, with just a shell or half shell active, then it is necessary to invoke sync explicitly before halting the machine. Thus, the best habit is to invoke sync on any shutdown.

The critical thing is to get the system quiescent, so that nothing gets interrupted part way through. Many computers have the bad taste to write zeros for the remainder of a disk block write, if the machine is halted part way through the operation. Should a block of sixteen inodes be on the way out at this point, merely because the last accessed time of one of them was updated by a reference, it is quite possible that up to sixteen unrelated files might be lost, even files that have not themselves been touched for months. Little can be done to overcome such hardware limitations.

A related social problem occurs on a multi-user system -- getting everyone to quit at the same time. If the terminals are not within shouting distance of each other, the System Administrator can broadcast a shutdown request using the write utility. Even better practice is to write the shutdown message in /adm/motd, normally printed out at login time, then broadcast from there; that way, people who login during the shutdown warning period are also warned.

In extremis, the System Administrator can fall back on:

```
# kill -2 1
```

which instructs multi to kill all processes it knows about and restart. Assuming that the System Administrator has access to /dev/console, and that startup first runs a single user shell on that tty, then this mechanism can be used to kick everyone off the system. It may still be

necessary to stop any background processes, using ps to find them and kill to do them in, and one should certainly invoke sync at the very end, but this control is absolute.

Keep in mind that many programs, the editor e in particular, use temporary files in /tmp. So if the system is shutdown while an instance of e is running, even if it is quietly waiting for keyboard input, then some garbage will be left in /tmp, which must eventually be cleaned up.

The best way to bring down a conventional system, therefore, is to get everybody to logout, then wait a minute.

NAME

Mkfs - making filesystems

FUNCTION

Disk storage under Idris is conventionally organized into one or more filesystems, which are logical data structures capable of administering a large number of files. Starting with a fundamental, or root, filesystem, the system permits additional filesystems to be mounted, or attached as subtrees in the existing directory hierarchy. Once mounted, a filesystem's boundaries become nearly invisible, so that arbitrary amounts of data may be treated as a single hierarchical entity.

Multiple filesystems are desirable for other reasons as well. Given removable media, as with a diskette or cartridge drive, it is possible to store data "off line", away from the computer. This permits a large community of files even on a system with limited on line disk capacity; and it eases the problem of backing up files, an important safety measure.

Some disks are too large to be administered as a single filesystem, so the System Generator usually divides a large disk into a number of smaller logical sub-disks, each of which can support a single filesystem. This is often done even for disks less than the 32 million byte maximum for a single filesystem, because smaller filesystems are easier to inspect and back up. Thus, even a single 20 Mbyte (million byte capacity) disk may well be divided up into two to four smaller filesystems.

Since Idris runs on such a large variety of hardware, each System Administrator must confer with the System Generator to determine what device names go with what physical devices, on a given system, and how many blocks may safely be used with each device name.

Yet another important consideration, that is beyond the scope of this essay, is how the media is prepared for writing (formatted) and placed on line. Again, it is up to the System Generator to determine procedures for readying disks. The System Administrator can verify that a disk is usable by trying first to read it:

```
# od -hcvt16 /dev/disk
```

where /dev/disk is the presumed filename for the block special device in question. This operation should result in a seemingly endless printout of hexadecimal numbers, sixteen to the line, if the disk is readable; otherwise the console device will probably record a series of complaints from the resident handler for that device. You can write to a disk just as easily:

```
# echo "are you there?" > /dev/disk
# od -hcvt16 /dev/disk
00000000    61 72 65 20 79 6f 75 20 74 68 65 72 65 3f 0a 6a
    etc.
```

If you think a disk is writable, and you know how many blocks are safely usable, then you are ready to build a filesystem on it. A 3740-compatible diskette, for instance, has 128 bytes per sector, 26 sectors per track,

and 77 tracks per diskette. This multiplies out to 500 1/2 blocks of 512 bytes, the standard unit of Idris disk activity. So the largest filesystem you can safely make is:

```
# mkfs -s500 /dev/dk0
/dev/dk0:
368 inodes
474 free blocks
```

assuming that /dev/dk0 is the name of the diskette block special file. (The extra half block is not usable, as part of a filesystem.) The printout is reassurance that the filesystem was successfully constructed, and informs you that mkfs elected to allocate 368 "inodes", or file entries, on the diskette. This means that no more than 368 files can be supported on that filesystem, even if there is space (free blocks) to hold additional file contents. mkfs applies a default formula, determined from years of experience, to select an appropriate number of inodes for a given size filesystem; it can be overriden if you have your own ideas.

Initially, only one of these inodes is consumed. Inode 1 is the root inode for any filesystem, and mkfs makes it a directory with just the conventional links . and .., much as mkdir would do. Note, by the way, that .. for a root inode points back at itself, just like ., since there is no "parent" for it to point back at. This is one of those rare places where you can see the boundaries between separate filesystems -- "cd .." will not climb beyond the root of a mounted filesystem.

To get files onto the newly made filesystem, you can either 1) mount it and copy files to it, or 2) use restor to restore files from a previously dumped filesystem onto it. In the latter case, the filesystem had better be big enough, with enough inodes, to accommodate the image being restored -- restor can be made to tell you how big is big enough.

To mount a filesystem, you need an existing inode in the directory hierarchy. Any old inode will do, but by convention, it is usually one in the root directory. Say you want to keep user private directories on a separate filesystem, which is to be built up on /dev/dk0, then

```
# mount dk0 /usr
```

will mount the newly initialized diskette overtop the directory /usr. If /usr has had any additions on the original filesystem, they are now unreachable; subsequent references to /usr lead onto the diskette until it is unmounted. You can now make subdirectories on the diskette, copy over files, or do anything else that you've been doing on the root filesystem:

```
# cd /usr
# mkdir joan tom
# echo "mail -q" > joan/.login
# cp joan/.login tom
```

and so forth.

When you are done, and ready to replace /usr with a previously prepared diskette filesystem:

```
# cd /
# mount -u dk0
```

The cd is necessary because mount will refuse to unmount a "busy" filesystem, and having a current directory within the filesystem is considered busy enough. If the above unmount does not complain, you can safely remove the media and replace it with the next filesystem you wish to work on:

```
# mount dk0 /x
# ls -l /x
```

and so forth. Note that this time the diskette was mounted at /x, for whatever reason. Idris doesn't care.

The important thing to remember is that a filesystem is a logical data structure that must be explicitly written to a disk before it can be safely mounted. As diskettes come out of the box, they are not ready to mount; that's what mkfs is for. Of course, you can also make a valid filesystem by copying an existing one:

```
# dd -c500 -o /dev/dk1 /dev/dk0
```

This replicates the filesystem presumably written on dk0 onto the new diskette loaded on dk1.

If you mount a disk that is not a valid filesystem, Idris will get upset. It will probably not crash, since it is alert for such mistakes, but it won't like it. To make sure there really is a filesystem out there, try:

```
# icheck /dev/dk0
specl       0
files       0
direc       1
small       1
large       0
indir       0
total free blocks   474 / 500
total free inodes   367 / 368
```

If icheck has no complaints, Idris probably won't either.

Once your basic filesystems are set up, then it is most convenient to have the standard ones mounted automatically at system startup. This may not be advisable for easily demounted media, such as diskettes, but for a Winchester disk with multiple partitions, for instance, it makes it easy to forget about any artificial boundaries. If your system is heavily loaded, then performance may well be affected by how filesystems are laid out, but that is once again more of concern to the System Generator.

NAME

Dump - backup conventions

FUNCTION

If the Idris operating system becomes damaged, it can always be restored by repeating the installation process. Recovering your precious files is not so easy, however. Any well run system must have some provision for orderly and regular backup of data, or it will come to grief sooner or later.

Files are most easily backed up by copying them to demountable media, such as tape or diskettes. The tp utility serves this purpose fairly well, for small quantities of data. Diskette filesystems are also convenient, since the cp utility can copy entire directory subtrees at one go. And you can often make image copies of filesystems, if demountable media exists with capacity comparable to the filesystems to be saved.

The most flexible form of backup, however, is provided by the utilities dump and restor. With them, you can dump an arbitrarily large filesystem to arbitrarily small media, using as many tapes or diskettes as necessary. You can dump an entire filesystem or only those files that have changed since the last dump. You can preserve attributes such as time of last modification or time of last access, which are often useful to know for tracking changes. And finally, you only have to save the active portion of a filesystem; control information and free space don't waste space on backup media.

Given dump images, on tape or diskettes, you can fully restore a filesystem, or you can extract selected files and move them into place as needed. You can use the "incremental" dumps mentioned above to restore a filesystem to a number of checkpoints in time, without having to save redundant information. And you can restore the logical contents onto a filesystem that is much larger or (possibly) smaller than the original.

Moreover, all forms of backup -- dumps, tp "tapes", and diskette filesystems -- may be exchanged with other Idris (and UNIX/V6) sites.

So the important thing is to pick the most convenient method of backup, then use it regularly. For an active system, daily backup of active files is not out of the question; weekly full dumps should be commonplace for all but the largest and smallest systems. It is best to set aside three or four complete sets of backup media and institute a "dump cycle".

If the dump media are labelled A, B, C, and D, say, then the easiest cycle to follow is a simple ABCDABCD... A better scheme is the so called "Towers of Hanoi" sequence, ABACABAD..., which repeats only half as often. In either case, it is best to preserve all D sets, given the money and the storage space, and just cycle through A, B, and C.

A few cautions, however:

1) tapes and diskettes eventually wear out, so plan to retire the A cycle before too many repetitions.

2) tapes and diskettes can have a very short shelf life, particularly if exposed to dust, magnetic fields, heat, light, and/or moisture. Consider a bank vault for other reasons besides protection from fire and theft.

3) backups are only useful if they work. Check your dumps from time to time to see if they can be read.

If you set up these procedures and adhere to them, they are almost certain to pay off several times per year. This should not, by the way, be taken as a comment on the reliability of the Idris system -- the vast majority of all restore operations are to correct human error.

NAME

Patch - maintaining Idris filesystems

FUNCTION

Errors happen. The Idris filesystem contains sufficient redundancy so that most errors show up as inconsistencies in the internal control structure; hence they can be corrected by determining which of conflicting data contain the errors, then making that agree with the rest. Several utilities are provided to help in this process.

If you think a filesystem is damaged, then it is best to work on it while it is unmounted, because Idris retains certain information in memory that can get out of phase with the disk image. That may not be possible for the root filesystem, on a machine that supports no alternative root -- some hints will be given as to how to survive this delicate situation. At the very least, dump a damaged filesystem if you can before commencing surgery, particularly if the damage is extensive and the information important.

Three administrative utilities are used to check filesystem consistency:

icheck ensures that every block of the filesystem occurs exactly once as part of a file or as part of the free list. It also complains if an inode is not allocated, but has nonzero mode bits, a suspicious circumstance. Given a list of block numbers, it tells what inode (if any) each belongs to.

dcheck ensures that each inode has as many directory links pointing to it as its internal link count would indicate. It also complains if a directory has garbage in it, such as links "off the end" or ill-formed links. Given a list of inodes, it tells what directory entries (if any) point to it.

ncheck ensures that every allocated inode has at least one path leading from the root. Given a list of inodes, it lists all pathnames (if any) leading to them. And it can be asked to list all possible pathnames in a filesystem.

These utilities should be run in the order given, to check out a filesystem:

```
# icheck /dev/dk0
specl       0
files       0
direc       1
small       1
large       0
indir       0
total free blocks   474 / 500
total free inodes   367 / 368
# dcheck /dev/dk0
# ncheck -i0 /dev/dk0
```

This is what a healthy filesystem looks like when checked. (The -i0 flag to ncheck suppresses the complete listing of pathnames.)

If damage is reported, however, there are several tools to further analyze the problem and then make repairs:

fcheck can be used to display inode status (much like ls), to print the file contents, to list the block numbers of data blocks, to compute the octal offset of the inode entry within the filesystem, or to clear the inode (with -patch) if it is too badly damaged to remove safely.

rm can be used (with -patch) to remove undesirable links, even to directories.

ln can be used (with -patch) to add missing links, even to directories.

dcheck can be asked (with -patch) to reconstitute the inode link counts and cleanup garbage in directories.

icheck can be asked (with -patch) to reconstitute the filesystem free list, out of all blocks not part of any file.

In all of the above cases, superuser permission is generally required to perform any operation called out by -patch. Needless to say, this flag should also remind you that extra caution is advised because normal restrictions are being bypassed.

Just what tools to apply, and in what order, depend strongly upon the nature and extent of the damage. Here are some of the common cases:

icheck reports lost, bad, or duplicate blocks in free list: Easy, just reconstitute the free list with "icheck -patch" and keep going.

icheck reports bad mode: Use "fcheck -t" and "fcheck -p" to inspect the file. If its contents are valuable, copy them elsewhere with a redirected "fcheck -p". Then clear the inode with "fcheck -patch".

icheck reports duplicate block in file: Again rescue vital file contents if at all possible, using "fcheck -p". Then mount the filesystem long enough to remove the file, unmount the filesystem, and reconstitute the freelist.

icheck reports bad block in file: If there are just one or two bad blocks reported, proceed as above for duplicate blocks. If there are hundreds of these, however, then in all probability a block full of pointers to other blocks has been damaged. The only safe course is to clear the inode with "fcheck -patch", then proceed as for duplicate block in file, above.

dcheck reports nonzero nlinks, but no directory entries: Rescue contents as needed, clear the inode, and reconstitute the free list.

dcheck reports mismatch between nlinks and directory entries: Use "dcheck -patch" to correct nlinks. If problem is on the root filesystem, make

sure the current directory of the shell is / before invoking dcheck. And if inode 1 is being corrected, shutdown the system and reboot before proceeding.

dcheck reports garbage in links, or data off end of directory: Use "dcheck -patch", as above.

ncheck reports unreachable inodes: Rescue contents as needed, clear the inode, and reconstitute the free list. Again, if these occur in large quantities, then an entire subtree has probably been orphaned. There is no easy way to deal with this situation, because 1) it is hard to locate the root of the subtree, and 2) there is no machinery at present for creating a link to an arbitrary inode. All you can do is rescue the files one at a time, then clean up.

Loop occurs in directory structure: Use "rm -patch" to break the back link.

Entries . or .. disappear from directory: (Often this manifests itself as a failure of pwd or ls.) Use "ln -patch" to replace missing links.

As a final recourse, you can always use the debugger db (documented in the various C Interface Manuals) to inspect and modify control information. fcheck can provide octal offsets for inodes, but from then on you're on your own. Read carefully the description of filesystem data structures in Section III of this manual, then get a friend to check you.

And remember that some of these operations interact. "dcheck -patch" will adjust all link counts, perhaps deallocating an inode with no directory entries; so be sure to rescue such orphans before a general cleanup. The best rule is to save for last the two operations "dcheck -patch" and "icheck -patch", then perform them in that order.

BUGS

As of this writing, there is inadequate machinery for dealing with two problems: It is hard to remove a link to a cleared inode, because the resident is too persnickety about how it traces pathnames. And it is hard to rescue an orphaned subtree created by moving a directory to a subdirectory of itself (permitted by a bug in rm and exacerbated by inadequate patching machinery). Both problems can be handled by db, but require more sophistication than they should.

NAME
alarm - send alarm signal periodically

SYNOPSIS
/odd/alarm -[p# s#]

FUNCTION
alarm is used by the call up utility to make sure that it is interrupted periodically while waiting for input characters on the communication link. This permits missed packets to time out and be recovered.

The flags are:

-p# send alarm signal to processid #.

-s# sleep for # seconds between sending signals.

If the signal cannot be sent, alarm exits.

RETURNS
alarm loops forever, barring any errors, at least until it is killed.

SEE ALSO
recv, throttle

NAME

chown - change ownership of a file

SYNOPSIS

/etc/chown -[g u] <loginid> <files>

FUNCTION

chown changes the owner of each file in the list of file arguments <files> userid and groupid associated with <loginid>. Only the superuser succeeds with this command.

The flags are:

-g change groupid only.

-u change userid only.

Default is to change both userid and groupid.

RETURNS

chown returns success when all files specified have their ownership changed.

EXAMPLE

```
# chown root file1 file2 file3
```

NAME

dcheck - check links to files and directories

SYNOPSIS

/etc/dcheck -[i#^ patch] <filesys>

FUNCTION

dcheck checks the integrity of one or more filesystems in the list <filesys> by reading all its directories: first it counts the number of directory links to each inode, then it compares that count with the link count in each inode. It can be used 1) to check link counts, 2) to insure that all entries in a directory are reachable by the system, 3) to find hidden entries in the last block of a directory, and 4) to find all the directory inodes that contain links to a set of named inodes.

dcheck should only be run on an idle or unmounted filesystem.

The flags are:

-i#^ list each link to inode #, giving inode number of directory and link. Up to ten inodes may be specified.

-patch rewrite inode link counts as necessary to reflect the directory link count, erase characters after the first NUL in a link, change completely NULL links (with nonzero inode numbers) to "XXXX", and increase the size of a directory if it has links beyond the current size specified.

If no flags are given, the default is to perform checks 1), 2), and 3) listed above. Default is to be silent if all is well. If a damaged entry is found, the entry name and link pointer are printed along with the inode number of the directory.

RETURNS

dcheck returns success if all reads and writes are successful and all link counts are balanced.

EXAMPLE

```
# dcheck -i35 -i24 -i36 /dev/rm5
/dev/rm5:
i#     24: link       24, entry .
i#     30: link       24, entry resumes
i#    829: link       36, entry Print
i#    829: link       35, entry e6.control
```

SEE ALSO

fcheck, icheck, ncheck

NAME

devs - execute a command for each mounted device

SYNOPSIS

/etc/devs -[r] <cmd>

FUNCTION

devs invokes <cmd> repeatedly, once for each entry in the table of mounted filesystems. For each invocation of <cmd>, the name of the mounted on block special file is appended as an additional argument.

The flag is:

-r skip the root device.

If no <cmd> is specified, devs echoes the list of mounted filesystem devices.

RETURNS

devs returns success if <cmd> succeeds and if the mounted filesystem table is readable.

EXAMPLE

```
% devs df
/dev/rm03:
free blocks:   7892 / 16320
/dev/rm04:
free blocks:   8502 / 16320
/dev/rk0:
free blocks:   1331 / 4872
/dev/ry0:
free blocks:    102 / 1001
/dev/ry1:
free blocks:    256 / 1001
```

FILES

/adm/mount for the mount table, /bin/echo for the default <cmd>.

SEE ALSO

dcheck, df(II), fcheck, icheck, ncheck

NAME

dump - backup a filesystem

SYNOPSIS

/etc/dump -[b# c h i o* r# s# t] <filesys>

FUNCTION

dump copies the filesystem <filesys>, presumably onto tape, in a format suitable for partial or full restoration using restor.

dump can make either a copy of the complete filesystem or an "incremental" copy. In the latter case, dump will copy only those files which have been created or modified since the last complete dump. dump keeps a log, the dump history file, of the last complete dump of each filesystem.

When dumping an active filesystem, if a file is deleted while the dump is in progress, a zero length file is written to the tape in its stead. If a file to be included in the dump is modified while the dump is in progress, the updated version will be written to tape. Either occurrence causes a "phase error", and a warning message is displayed. If, after a dump has begun, a file is created or updated but was not originally to be included in the dump, the file is not dumped. The total number of phase errors is reported at the end of the dump.

A tape with phase errors can generally be restored properly under Idris, except for the files that changed under foot, but the UNIX/V6 restor may gag on the tape.

The flags are:

-b# write output in multiples of # 512-byte blocks. The default is 1. A blocking factor other than 1, with no -o* or -s# flags specified, cause dump to write to a character special device (raw tape), as described under -o*.

-c continue with next higher numbered drive when tape is full. A new output filename is derived by incrementing the last character of the previous one. If this flag is not specified and more than one tape is needed, dump will pause and request that another tape be mounted on the same device.

-h print to STDOUT the dump history file, containing the date of the latest full dump for each filesystem ever dumped.

-i do an incremental dump. Only files that have changed since the last complete dump of the specified filesystem (as indicated in the dump history file) are included in the current dump. If there is no entry in the dump history file for the specified filesystem then a full dump will be performed. If a full dump is performed, and no phase errors are encountered, an entry is made in the history file.

-o* output the dump to file *. The default output device is /dev/mt0. If -s# is given, the default device becomes /dev/rmt8. If -s# is not given, but -b# is present, the default becomes /dev/rmt0.

-r# assume each output tape has a capacity of # records, where a record is the multiple of 512 bytes given by the -b flag (or exactly 512 bytes if no -b flag appears). If no -r flag is given, a default file capacity is computed based upon 800 bpi, IBM-compatible tapes. The default unblocked capacity is 20000 blocks of 512 bytes, and is automatically increased to reflect blocking, if any. As a rule of thumb, an unblocked tape in the default format holds 9 blocks per foot; thus a 1200 foot tape would accomodate up to 10800 blocks.

-s# skip # complete files on the output device before beginning the dump. This flag should be used with care; dump always presumes it starts at the beginning of the tape when computing its length.

-t write to STDOUT the size of the dump, in files, tape blocks, and tenths of a tape, to STDOUT. No dump is actually performed.

If no flags are given, -t is assumed.

RETURNS

dump returns success if it was able to back up the filesystem completely without any read, write or phase errors.

EXAMPLE

To backup an rk05 disk:

```
# dump /dev/rk0
full dump of /dev/rk0
217 files
3018 blocks
0.2 tapes
0 phase errors
```

SEE ALSO

restor

NAME

fcheck - chase inodes down by number

SYNOPSIS

/etc/fcheck -[b i# o patch p t] <filesys>

FUNCTION

fcheck permits inspection of filesystem inodes. It can be used simply to find the octal offset of a specified inode, to clear a badly damaged inode for future consumption, or to print the contents of a file that may be otherwise unreachable.

The flags are:

-b give the block numbers addressed by the inode

-i# specify inode. Up to ten inodes may be processed.

-o give the offset into the filesystem in octal bytes, of the inode.

-patch clear the inodes mentioned. This requires write permission on the specified filesystem. DO NOT DO THIS LIGHTLY.

-p write to STDOUT the file contents pointed at by each inode.

-t tabulate the inode status, in the same format as ls, but without the name.

At least one "-i#" must appear. If -patch is specified, none of the flags [b o p t] may be used. If no flags are given, the default is -t, i.e. print the permissions, the number of links, the owner, the size, and the time of last update.

If multiple options are given and the -patch flag is not specified, the order of output is 1) the octal offset specified by -o, 2) the inode status line called for by -t, 3) the list of blocks directly addressed by the inode, specified by -b, and 4) the file contents called for by -p.

If multiple inodes are specified, each is preceded by a line giving 7 inode number, followed by a colon.

RETURNS

fcheck returns success if no diagnostics are produced, i.e. if all reads and writes are successful on the filesystem.

EXAMPLE

To retrieve unreachable inode number 81:

```
# fcheck -p -i81 /dev/rk0 > lazarus
```

SEE ALSO

dcheck, icheck, ncheck

NAME

glob - expand argument list and invoke a command

SYNOPSIS

/odd/glob arglist

FUNCTION

glob is the program used by the shell to expand arguments with pattern metacharacters into lists of filenames. Unlike all other programs, the first argument passed to glob is not the name with which it was invoked, but rather the name of a program to be invoked. glob creates a new argument list with the same first argument. The second through the last arguments to glob are examined for metacharacters, and if present, each is replaced by a list of filenames that match the argument taken as a pattern. If there are no metacharacters or no matches, a new argument is created identical to the old one. The program is then invoked; searching alternate directories and executing shell scripts is performed just as done directly by the shell.

Each argument examined is assumed to be a pathname. If its rightmost suffix which does not contain slash contains a metacharacter, then this suffix is used as a pattern and an attempt is made to find a match in the directory specified by the balance of the argument. If the argument did not contain a slash, the current directory is searched. The metacharacters are: "?*[". Each pattern is anchored between a '^' and a '$' before attempting a match, i.e. the full pathname must match the pattern. Patterns will not match filenames beginning with dot unless dot is specifed as the first character of the pattern.

A metacharacter may be smuggled past the matching logic by setting its parity bit. The shell does this secretly for each quoted or escaped command line character; either the shell or glob clears the parity bit on all unexpanded argument characters before invoking the program.

RETURNS

glob will complain and return failure if there are too many arguments or if the argument list is too long. Otherwise, the value returned is that of the command.

SEE ALSO

e(II), sh(II)

NAME

hsh - execute simple commands

SYNOPSIS

/odd/hsh -[f]

FUNCTION

hsh reads lines of input and interprets them as commands to execute, with arguments to be passed to the invoked program. If the name by which hsh is invoked (argument zero) is "init", then it operates interactively, writing a "# " prompt to STDOUT and reading a line of text from STDIN for each command; otherwise it reads commands without prompting and executes them. A command line is taken as a set of strings separated by whitespace. The first string on the line is taken as the name of an executable file, the command name; subsequent strings are used to create arguments to the command.

hsh is a (much) simpler version of the standard shell sh. It is more suitable than sh for operation on very small systems, particularly bootstrap systems that operate without a swap device. For this application there is a flag:

-f don't fork before executing the command.

Note that, with the -f flag, hsh exits every time it executes other than a builtin command. Moreover, in interactive mode, it closes all files and connects all standard files to "/dev/console".

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, the pattern 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.

Escape sequences. Metacharacters lose their special meaning when enclosed in single or double quotes. Embedded whitespace is included in an argument when the argument is in quotes. Outside double quotes, the character '\' followed by a single character causes the single character to be taken literally. An exception to this is '\' followed by a newline, which indicates line continuation. The newline is quietly discarded.

Redirection of STDIN and STDOUT. hsh reads from STDIN and prompts to STDOUT; both of these files are normally connected to the terminal. A command executed by the shell uses the same STDIN and STDOUT unless otherwise indicated.

STDIN may be redirected as follows:

< file - STDIN for the command will be file.

STDOUT may be redirected in one of two ways:

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

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

Locating Commands. hsh begins its search for a command by looking for a file with exactly the name specified by the first string on the command line. If no such file exists, and if the string contains no '/', then to it is prepended a succession of directory paths; hsh looks for the command under each resulting name. If a file is found with the desired name, has execute permission, but is not in executable format, it is taken as a shell script and an instance of /bin/sh is invoked to execute it, with the original command line as arguments. The directories currently searched for commands are /bin, /etc, and /etc/bin.

Builtin Commands. The command cd is builtin, i.e. it is executed by the shell itself.

RETURNS

hsh returns success if the last command line read was parsed without complaint and executed successfully.

FILES

/bin/sh for scripts, /odd/glob for metacharacters.

SEE ALSO

cd(II), sh(II)

NAME

icheck - scrutinize filesystem inodes

SYNOPSIS

/etc/icheck -[b#^ patch] <filesys>

FUNCTION

icheck verifies internal consistency of one or more filesystems in the list <filesys>, on an inode vs. block basis, by placing all of its inodes and blocks into various categories. It can be used simply to perform general checks on a filesystem, to glean information on a specific set of blocks, or to rebuild the filesystem free list.

The flags are:

-b#^ report on block whose number is #. icheck reports whether the block is pointed to by a specific inode or whether it is part of the free list. Up to ten blocks may be specified.

-patch reconstitute the free list on a filesystem. All blocks not otherwise accounted for are taken as free.

Written to STDOUT is a list, for each filesystem, consisting of: the number of blocks which are unaccounted for (lost), the number of block special or character special files, the number of plain files, and the number of directories. This is followed by the number of small files, the number of large files, and the number of blocks devoted to indirect pointers to large files.

If there are any huge files, two more lines follow indicating the number of huge files and the number of second level indirect blocks. This is followed by a ratio giving the number of free blocks left versus the total number available, and another ratio giving the number of free inodes left versus the total number available on the filesystem.

The order of output is 1) warnings of inconsistencies between inodes and blocks, giving the block number and guilty inode, 2) warnings of inconsistencies in the list of free blocks. 3) the usage list described above.

If icheck encounters difficulty in reading or writing a block, it quits with an error message indicating the offending block.

RETURNS

icheck returns success if no diagnostics are produced, i.e. if all reads and writes are successful on the desired filesystem and if no damage is found.

EXAMPLE

```
# icheck /dev/rm3
/dev/rm3:
specl       0
files    3310
direc     201
small    3150
```

```
        large    361
        indir    361
        total free blocks  2133 / 16320
        total free inodes   905 / 4416
```

SEE ALSO

dcheck, fcheck, ncheck

NAME

log - sign on to the system

SYNOPSIS

/odd/log <tty> -[bs# cr# +echo echo erase* +even even ff#
ht# +hup hup kill* +mapcr mapcr +mapuc mapuc nl# +odd odd
prompt* +raw raw s*^ unum# +xtabs xtabs]

FUNCTION

log is the program used by multi to log users onto the system. log opens the terminal <tty>, prompts for a loginid, then optionally prompts for a password after turning off echoing. Note that <tty> is assumed to be in /dev, and must not have the "/dev/" prefix. If the loginid is in the password file, and the password encrypts to match the entry in the password file, then the user is logged into the system. If the loginid has no password associated with it, the user is logged in without being prompted for a password.

At the system level, log is responsible for setting the userid and groupid, changing the ownership of <tty>, and writing accounting information into the who and history files if they exist.

At the user level, log is responsible for opening the file descriptors STDIN, STDOUT, and STDERR to <tty>, changing to the home directory, and executing the command specified in the password file.

In addition, log sets the initial stty values, and possibily adapts to different speeds, according to the flags specified. The default terminal configuration is: +echo +even +odd -hup +mapcr -mapuc -raw +xtabs -erase "\b" -ff0 -bs0 -kill "@" -nl0 -ht0 -cr0 -s300 -prompt "\7login: ", i.e. a 300 baud, full duplex terminal, with parity ignored, horizontal tabs translated into spaces, no character delays, and all typed carriage returns and line feeds treated as newlines.

The flags are much the same as for stty(II), except for:

prompt* use the string * as the login prompt.

-s*^ change the baud rate to * for both input and output. The value for * must be in the set {0, 50, 75, 110, 134.5, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, exta, extb}. Up to 14 speeds may be specified; log makes a circular list of the given speeds, and changes to the next on receipt of an interrupt (DEL key at the proper speed, framing error or BREAK key at any speed). A speed of 0 means hangup the line.

-unum# record logins and logouts in record # in the who file.

log prints the message "login incorrect " for bad loginid or password, "bad directory: dir " if it can't change to the initial directory, and "can't execute: cmd" if it can't execute the command. The last two messages indicate an error with either the password file or the system directories.

RETURNS

log returns the exit status of the command.

EXAMPLE

To start the log process on /dev/dz1, the /adm/init file would contain:

```
m /odd/log dz1 -unum3
```

FILES

/adm/who for current usage accounting, /adm/log for usage history accounting, /adm/passwd for login information.

SEE ALSO

multi, passwd(II), stty(II), who

NAME

mkdev - make a special device file

SYNOPSIS

/etc/mkdev -[b# c# m# u#] <files>

FUNCTION

mkdev defines one or more <files> as either block special or character special devices. The command may only be used by the superuser.

The flags are:

-b# use # as the major number for a block special file.

-c# use # as the major number for a character special file.

-m# set access mode to #. Default is 0666.

-u# use # as the minor number for the first file. Default is 0.

Either -c or -b must be specified, but not both. If more than one file is specified, the minor device number is incremented by one for each succeeding file. None of the files should already exist.

RETURNS

mkdev returns success if all the device nodes are successfully made.

EXAMPLE

```
# mkdev -b1 -u4 -m0644 /dev/ry0 /dev/ry1
# ls -l /dev/ry[01]
brw-r--r-- 1 root     1,  4 May 27 13:58 /dev/ry0
brw-r--r-- 1 root     1,  5 May 27 13:58 /dev/ry1
```

NAME

mkfs - make a new filesystem on a device

SYNOPSIS

/etc/mkfs -[b* i# s#] <device>

FUNCTION

mkfs writes an empty filesystem on the <device> specified, which should be a block special file if the filesystem is to be subsequently mounted. The filesystem has only its root inode allocated, which is made a directory owned by the superuser with general read, write, and scan permissions, and with the usual self-referencing links . and .., both pointing at the root directory. This is a very necessary step in preparing a block special device for mounting as an Idris filesystem.

Any existing files are sent to perdition; hence, mkfs should never be run on a mounted or otherwise valued filesystem.

The flags are:

-b* rewrite the bootstrap block from the file *. Up to 512 bytes are copied; no attempt is made to strip off any header information.

-i# set the number of inodes in the filesystem to #. The value specified is rounded up to the nearest multiple of 16. Default is one inode for every two filesystem blocks, up to 1024 blocks, one inode for every four filesystem blocks thereafter.

-s# make filesystem with # blocks total. Default is to leave the current filesystem unchanged, as when rewriting the bootstrap block.

At least one of -b and -s is required; if -i is specified, -s is required.

mkfs writes to STDOUT the number of inodes allocated and the number of free blocks, whenever it creates a new filesystem.

RETURNS

mkfs returns success if no diagnostics are produced. i.e. if all reads and writes are successful.

EXAMPLE

```
# mkfs -s1001 /dev/ry1
/dev/ry1:
496 inodes
967 free blocks
```

SEE ALSO

mount

NAME

mount - attach a new filesystem

SYNOPSIS

/etc/mount -[i p* r t] blkdev node [blkdev node ...]
or
/etc/mount -[p* u] blkdev [blkdev ...]

FUNCTION

mount logically attaches the filesystem on the block special device blkdev to the file node, causing all future references to node to refer to the root of the filesystem. The filesystem effectively replaces node for the duration of the mount. If node has any active processes or if blkdev has already been mounted, mount says "busy" and refuses to mount the filesystem. If any other errors are found, mount quits with the appropriate error message.

In its second form, mount detaches the filesystem on each of the mounted filesystems blkdev. If any files in the mounted filesystem are in use by any process, mount says "busy" and refuses to detach the filesystem.

The flags are:

-i initialize the mounted filesystem table. mount will try to unmount all of the devices in the mount table, and then make a new mount table, with the initial entry for the root filesystem. Typically done once at system startup.

-p* change default prefix for blkdev names to *. Default is the string "/dev/", which is prepended to any blkdev which does not begin with a slash.

-r mount the filesystem read-only. This prevents the resident from making any attempt to update inode access times or contents. Not even the superuser may alter the filesystem.

-t List the contents of the mounted filesystem table in a printable form on STDOUT, one line per mounted filesystem. If a filesystem is mounted read-only the string "[r]" appears following the pathname at which the device is mounted. This is the action when no arguments are present. If arguments are present, the table shown is a result of the processing of the other arguments.

Flags other than -t require superuser privileges.

At least one read permission, for user, group, or other, must be granted for read-only mounting. At least one each of read and write permissions must be granted for read-write mounting. There is no write-only mounting, and the execute permission is ignored. Thus, a mode of zero effectively protects an off-line device from being mounted.

If the mount table is not present, mount will still work, but it refuses to respond to a -t.

RETURNS

mount returns success if no diagnostics are produced and it can modify the mounted filesystem table as specified.

EXAMPLE

To mount the device /dev/rm3 at /x:

```
# mount /dev/rm3 /x
```

To unmount the devices rm1, rm2, and rk1:

```
# mount -u /dev/rm[12] /dev/rk1
```

FILES

/adm/mount for the mount table.

SEE ALSO

mkfs

BUGS

Sometimes it is not obvious why a filesystem is busy, and there is no way to display current directories of processes.

NAME

multi - start multi-user system

SYNOPSIS

/odd/multi

FUNCTION

multi is normally invoked under the alias /odd/init by process 1, to set up and maintain a multi-user environment. When multi is invoked, it reads the file /adm/init and memorizes it. Command lines in this file perform system initialization functions, such as mounting filesystems and starting the sync utility; command lines also specify login ports and baud rates to try for each port.

Commands in /adm/init are lines of text each beginning with a single character command, followed by a space, followed by the absolute path name of the program to be started, followed by the arguments to the program, and terminated by a newline. Argument 0 of the called program is the absolute path name with the leading slashes and directory names stripped off. This is the name that ps displays. If the command character is uppercase, the program is started with STDIN, STDOUT, and STDERR opened to /dev/console. The commands are:

w or W execute the command once, waiting for it to complete before going on to the next.

s or S execute the command once, but don't wait for it.

m or M execute the command, and re-execute it every time it exits.

k or K kill the process associated with this line.

other ignore this line. Useful as a place saver.

If a lowercase control character is used, the program will have no files opened. You will not see any error messages from it. Any write to STDOUT or STDERR this program may attempt will fail, possibly causing it to malfunction.

multi begins by reading /adm/init into its memory, then processes each line according to its control character. Any time after multi has started it is permissible to edit /adm/init to reconfigure the system. The command

```
# kill -1 1
```

sends a hangup signal (kill -1) to process 1, which is multi. This causes multi to reread the control file, comparing the file line for line with the previous contents. If the line is unchanged, on a character for character basis, no action is taken. Otherwise the process associated with that line is killed (sent a kill signal, which cannot be ignored) and the command line is reexecuted. New command lines should therefore be added to the end of the file; adding or deleting lines at the wrong place can be catastrophic.

The second way to rearrange the system is to send init an interrupt signal with the command:

```
# kill -2 1
```

This causes init to kill all the processes that it knows about, and then exit, which is the recommended procedure for deleting lines from the control file, or for dismantling an active system before shutdown. When multi exits as process 1, the resident automatically restarts it. Thus, an /adm/init file that starts out with a single-user shell, then goes on to multi-user, can be brought back to single-user status with this signal.

RETURNS

Nothing.

EXAMPLE

A sample single user control file is:

```
m /bin/sync -30
M /bin/sh -i -H/ -X"|/bin/|/usr/bin/"
```

FILES

/adm/init for the control file.

SEE ALSO

hsh, kill(II), log, mount, sh(II), sync(II), who(II)

BUGS

Small changes to /adm/init can raise hell with the entire system. Editing this file and signalling multi must be done with great care.

NAME

ncheck - find inode aliases

SYNOPSIS

/etc/ncheck -[a i#^] <filesys>

FUNCTION

ncheck hunts down all the possible combinations of pathnames for a given set of inode numbers. It can be used simply to find all the possible names for an inode, to find all the possible pathnames in a filesystem, or to check pathname versus allocated inode integrity.

The flags are:

-a check all allocated inodes including those starting with ".".

-i#^ print information on specified inode #. Up to ten inodes may be specified.

If no flags are given, the default is to print the inode number and unique pathnames for all files.

The order of output is 1) the inode names specified by -i#, or all the pathnames on the filesystem, 2) the list of inodes that can't be found to have a pathname.

RETURNS

ncheck returns success if no diagnostics are produced, i.e. if all reads are successful on the desired filesystem.

EXAMPLE

To find all the possible names for inode number 69:

```
# ncheck -i69 /dev/rm3
   69 /c/cpp/doc/p0.ic
```

SEE ALSO

dcheck, fcheck, icheck

NAME

recv - receive data downlink

SYNOPSIS

/odd/recv <tty>

FUNCTION

recv is invoked by the call up utility cu to camp on the link file <tty>, copy characters from the link to the controlling tty, and perform functions signalled by escape sequences sent downlink.

Its functions are described in conjunction with cu(II).

SEE ALSO

alarm, cu(II), throttle

NAME

restor - extract files from a backup tape

SYNOPSIS

/etc/restor -[a b# c f* i*^ s# t] <filesys>

FUNCTION

restor is used to recreate files and filesystems saved with the dump command. It can be used to restore complete filesystems, possibly to a smaller device than the original, or to restore individual files by inode number. The flags are:

-a recreate the entire filesystem. For a full dump, this should only be done to an empty filesystem of the appropriate size (as produced by mkfs). For an incremental dump, it should only be applied to a filesystem restored to the state at the time of the dump.

-b# presume a blocking factor of # when reading each input file. If no "-f" flag is given, restor will automatically use a character special device to access the tape drive when reading a blocked tape.

-c continue with next higher numbered drive on end of file. A new input filename is derived by incrementing the last character of the previous one.

-f* restore from file *. The default input file is /dev/rmt8 when a -s flag is given, /dev/rmt0 when a -b flag but no -s flag is given, and /dev/mt0 otherwise.

-i*^ restore the inode numbered * and put it in the current directory, with the name *. Up to 10 inodes may be given.

-s# skip # complete files on the input device before beginning to read it.

-t tabulate the contents of the input file only; no restoration actually occurs.

The flags -i, -a, and -t are mutually exclusive; if none of the three are specified, -t is assumed.

restor checks the files given in the dump map preceding the dumped filesystem against the files actually recorded in the inodes dumped. Any difference is flagged as a phase error. If the -i option is selected, restor will ignore phase errors on inodes that are not requested, but will print an error message and continue if one occurs on an inode selected for restoration. If the -a option is selected and restor encounters a phase error, the associated inode will be truncated and a message to that effect will be output. Filesystem damage may result from such an error.

RETURNS

restor returns success if no diagnostics are generated.

EXAMPLE

To restore onto an rk05:

```
# restor -rc /dev/rk0
full restor from Wed Dec 17 11:57:32 1980
dump size: 4872 blocks, 1872 inodes
minimum: 1376 blocks, 652 inodes
restor onto /dev/rk0?  yes
```

Note that the minimum size for the filesystem is 1376 blocks; this corresponds to the -s flag in mkfs. Also notice that if restor encounters an end of file on the default /dev/mt0 before it is finished, it will go on to /dev/mt1, and so on until it has finished.

SEE ALSO

dump, mkfs

NAME

throttle - send start/stop codes uplink

SYNOPSIS

/odd/throttle <#>

FUNCTION

throttle is used by the call up utility cu to send ctl-S and ctl-Q characters uplink at regular intervals, in the hopes of throttling output from the remote system. The delay after sending the stop signal ctl-S is given by the first character of <#>, in seconds, which should be a decimal digit. The delay after sending the start signal ctl-Q is always one second.

RETURNS

throttle runs forever, at least until killed.

SEE ALSO

alarm, recv

INDEX

IDRIS USERS' MANUAL

This index was generated from the name of each manual page in this manual. To use it, look up the word or phrase of interest by reading down the column at the center of the page. Once you have located a line containing the word, just read it, left to right. The name of the manual page it refers to will be in boldface at the start of the line, and the number of the page will follow in italics.

Thus the line:

e (II-27) text editor

refers to the manual page for the text editor e, which can be found under the name "e" starting on page 27 of Section II of this manual. (And you will find other index entries for this page under "text" and "editor". A line exists in the index for each important word from each manual page name.)

Whitesmiths, Ltd.
97 Lowell Road, Concord, Massachusetts 01742