IV. System Administration Guide

TABLE OF CONTENTS

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 ac@count@ing 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 "wno:" 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 ac@count@ing 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 ac@count@ing 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 ac@count@ing 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>. If <loginid> is a decimal number and is not a valid login name then it is used as a userid. In this case -u is assumed. Only the superuser succeeds with this command.

The flags are:

-g change groupid only.

-u change userid only.

The default is to change both userid and groupid.

RETURNS

chown returns success when all files specified have their ownership changed.

EXAMPLE

To change ownership to root on the files file1, file2 and file3:

```
# chown root file1 file2 file3
```

To change the userid to userid 9 on the file daemon:

```
# chown 9 daemon
```

NAME

dcheck - check links to files and directories

SYNOPSIS

/etc/dcheck -[i#^ patch] <files>

FUNCTION

dcheck checks the integrity of one or more filesystems 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.

If <files> is an ordinary file in the current directory, then <files> is assumed to contain a filesystem. If no <files> are specified, or <files> is not a filesystem, dcheck uses the device on which the current directory or the <files> resides. If any of <files> does not exist, to it is prepended "/dev/" for a second try. The output of dcheck is preceded with that filesystem name.

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 NULed links (with nonzero inode numbers) to "XXXX", and increase the size of a directory if it has links beyond the current size specified. This requires write permission on the specified filesystem.

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 w] <cmd>

FUNCTION

devs invokes <cmd> repeatedly, once for each entry in the external table of mounted filesystems. It is frequently used to initiate filesystem integrity checks after a crash. For each invocation of <cmd>, the name of a mounted on block special file is appended as an additional argument.

The flags are:

-r skip the root device.

-w use writeable devices and skip readonly devices

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

To list freespace everywhere:

```
% 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
```

To check all filesystems:

```
# devs dcheck && devs icheck
```

FILES

/adm/mount for the external 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# since* 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 dump cannot open a next-higher drive it cycles back around to the first 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.

-since# dump only files created or modified since the canonical date #. Implies -i without the requisite history file. The date is specified exactly as for the date command i.e. yymmddhhmm. Missing leading digits use the values from the current date.

-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
```

To backup / to 700 block floppy disks:

```
# dump -r700 -f/dev/floppy0 /
full dump of /dev/winchester
556 files
3104 blocks
5.0 tapes
next tape ready? y
next tape ready? y
next tape ready? y
next tape ready? y
0 phase errors
```

SEE ALSO

restor

NAME

fcheck - chase inodes down by number

SYNOPSIS

/etc/fcheck -[b i# o patch p t] <file>

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.

If <file> is an ordinary file in the current directory, then <file> is assumed to contain a filesystem. If <file> is not specified, or <file> is not a filesystem, fcheck uses the device on which the current directory or the <file> resides. If <file> does not exist, to it is prepended "/dev/" for a second try. The output of fcheck is preceded with that filesystem name.

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 the 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
```

To examine two inodes on the root device:

```
% fcheck -bot -i500 -i25 /
500:
offset of inode: 41140
-r-xr-sr-x  1 root        6518 Apr 19 10:19
type: indirect
    b0 ->   3655
    b1 ->      0
    b2 ->      0
    b3 ->      0
    b4 ->      0
    b5 ->      0
    b6 ->      0
    b7 ->      0
25:
offset of inode: 3400
crw--w--w-  1 dave      6,    0 May 03 00:43
char special device
    b0 ->   1536
    b1 ->      0
    b2 ->      0
    b3 ->      0
    b4 ->      0
    b5 ->      0
    b6 ->      0
    b7 ->      0
```

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 a 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.

Metacharacters occuring before a slash are patterns which only match directories. glob will scan all matched subdirectories, and do this recursively as long as there are metacharacters left in the pathname. Patterns of this form must match an existing file or directory, or the search is abandoned. Thus, * matches all files in the current directory, /* matches all files in the root directory, */ matches all subdirectories in the current directory, and */* matches all files in all subdirectories.

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 can run out of memory or build more arguments than the system can handle and will return an error. 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] <files>

FUNCTION

icheck verifies internal consistency of one or more filesystems, 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.

If <files> is an ordinary file in the current directory, then <files> is assumed to contain a filesystem. If no <files> are specified, or <files> is not a filesystem, icheck uses the device on which the current directory or the <files> resides. If any of the <files> does not exist, to it is prepended "/dev/" for a second try. The output of icheck is preceded with that filesystem name.

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.

-fsize# override the internally recorded size of the filesystem and use # instead for checking boundaries. The default is to use the size recorded by mkfs in the filesystem itself.

-patch reconstitute the free list on a filesystem. All blocks not otherwise accounted for are taken as free. If -patch is combined with -fsize#, then the filesystem is either extended or truncated to # blocks. Truncating is only safe when # is greater than the highest allocated block.

A list is written to STDOUT 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.

The next two lines indicate the block number of the highest allocated block and the inode number of the highest allocated inode. Note that since the files are not stored continguously and the freespace is maintained as a list, unallocated blocks may come both before and after the highest allocated block. Also, since many files may be created yet all but the last file deleted, free inodes may come both before and after the highest allocated inode. 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:
specl       0
files    3310
direc     201
small    3150
large     361
indir     361
bhigh   12045
ihigh    3687
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# i 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.

-i read device names from STDIN.

-m# set access mode to #. The default is 0666.

-u# use # as the minor number for the first file, or the first file of each line read from STDIN if -i used. The 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.

If -i is specified, the names are read from STDIN. The first line read defines the devices for the major number given by -b# or -c#. The major number is incremented for each line that follows. The minor names on a line are terminated by a space character. The minor number is set to -u# or 0 for each line, and is incremented for each space character in the line. An unused minor number is skipped by placing s single space character on the line.

RETURNS

mkdev returns success if all the device nodes are successfully made.

EXAMPLE

To remake the /dev directory:

```
# mkdev -c0 -i < /dev/cnames
# mkdev -b0 -i < /dev/bnames
```

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
```

#mkfs -s500 /dev/rx0

SEE ALSO

mount

NAME

mount - attach a new filesystem

SYNOPSIS

/etc/mount -[i m p* r t] <blkdev> <node> [<blkdev> <node> ...]
or
/etc/mount -[p* u] <blkdev> [<blkdev> ...]

FUNCTION

mount maintains the internal and external tables of mounted devices. It is used to attach and detach devices from the system.

In its first form, node is pathname to any type of file on any mounted disk and mount logically attaches the filesystem on the block special device blkdev there, causing all future references to node to refer to the root of the filesystem on blkdev. 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. <blkdev> may optionally be the node pathname on which the device to unmount is attached.

The flags are:

-i initialize the external mount table. mount will try to unmount all of the devices in the internal system mount table, and then make a new external mount table, with the initial entry for the root filesystem. Typically done once at system startup, mount -i will initialize the external mount table whether or not it succeeded in unmounting all devices.

-m print the internal system mount table. The contents of the internal system mount table are listed in printable form on STDOUT, one line per mounted filesystem. The display shows the device and inode number of where each device is attached and the root device.

-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 a filesystem if it is read-only.

-t List the contents of the external 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 or -m require superuser privileges. Some installations may opt to make mount a set-userid root program. Doing this will make mount available to all users, regardless of privilege. It is very difficult to enforce security if it is possible to mount arbritrary disks and diskettes, so elevating the privilege of mount in this way should only be done in reasonably friendly environments.

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 external mount table is not present, mount will still work, but it refuses to respond to a -t. An inconsistent external mount table will cause the pwd command to fail. The internal system mount table used by -m should always be available unless the /dev directory has been damaged or altered.

If the external mount table becomes too confused, use mount -m to find out which disks are really mounted. The device name and inode are sufficient input to ncheck to find the path name of where the device was originally mounted. The best course of action is to unmount all disks, then run mount -i.

When a device refuses to unmount it is because one or more of its inodes are busy. The current directory being on a device (or the current directory of a parent process) is most often the cause of "busy" messages. The ps -i command will display all inodes locked in memory for all filesystems. Again, ncheck is used to determine the path name of an inode.

RETURNS

mount returns success if no diagnostics are produced and it can modify the mounted filesystem tables as specified.

EXAMPLE

To mount the device /dev/rm3 at /x:

```
# mount rm3 /x
```

To unmount the devices rm1, rm2 and rk1:

```
# mount -u rm[12] rk1
```

FILES

/adm/mount for the external mount table. /dev/mount for the internal system mount table.

SEE ALSO

mkfs, ncheck, ps, pwd

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.

e or E exec the command. multi replaces itself with the command. This will be the last line executed.

k or K kill the process associated with this line. Acts as a place holder; no command is started for this line.

other ignore this line. Useful as a place saver. Note that tabs and spaces also serve as place holders.

If a lowercase control character is used, the program will not have STDIN, STDOUT, or STDERR files opened. This is used for programs that do not associate with the console, such as login or sync. You will not see any error messages from a program with a lowercase control character. 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:

```
W /bin/echo "check the date"
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 signaling multi must be done with great care.

NAME

ncheck - find inode aliases

SYNOPSIS

/etc/ncheck -[a i#^] <files>

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.

If <files> is an ordinary file in the current directory, then <files> is assumed to contain a filesystem. If no <files> are specified, or <files> is not a filesystem, ncheck uses the device on which the current directory or the <files> resides. If any of the <files> does not exist, to it is prepended "/dev/" for a second try. The output of ncheck is preceded with that filesystem name.

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.

By special dispensation, the flag -i0 is taken to mean search the entire filesystem for an inode that can never exist. This has the effect of checking every inode to make sure there is at least one pathname from the root to that inode.

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
```

To do a connectivity check:

```
# ncheck -i0 /dev/rm3
```

SEE ALSO

dcheck, fcheck, icheck

NAME

ps - print process status

SYNOPSIS

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

FUNCTION

ps prints information about processes and files in the system. It reads /dev/ps, to find out about all the processes in the system, /dev/myps, to find out about the processes associated with the invoking user's terminal, or /dev/inode, to find out about the list of active files (inodes) in the system.

The flags are:

-a print out information on all processes in the system.

-i print out information on the inode list.

-l print out information on processes giving the long form of printout.

-p print out information on your processes only. The output from -p follows any output generated by -a or -t.

-r limit print out to 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 an interrupt or kill signal.

-t print out psuedo-processes implementing shared text segments. This flag implies -a, as no person owns shared texts.

If none of -a, -t, or -i flags are given, -p is assumed.

ps prints out the processid (in decimal, the unique number of that process), the process state and the first part of the process name (usually the name of the file executed).

An Idris process may be in one of six states:

running needs the computer, but may or may not be in memory
waiting for children to finish, for input, or output is backed up
dozing needs a disk block, or access to a file (uninterruptable)
execing a newly created child process
sharing a psuedo-process implementing a shared text
exiting hard to catch

Further, a process may be totally dead, but not yet laid to rest (colloquially a "zombie") by a waiting parent; such processes are shown as a special line immediately after their own parent. Their state is "done" and the exit status is listed in parenthesis.

If -l is specified, additional information is given under:

PRI process priority (a signed integer, lower is better)
PPID the processid of the parent process
BASE the base in blocks of the process image
SIZE the size in blocks of the process image
FLAGS the internal flags for the process (detailed below)
CHANNEL the internal event being waited for
DEV the associated tty device for the process

Priorities of Idris processes range from -128 to +20 (negative numbers and zero are higher priority than positive numbers). Every process is started by a parent process (PPID); process 1 is the great grandaddy of them all. Events are listed in hex as a data cell address in the resident. Every process started by you is associated with your tty device (DEV). Shared text psuedo-processes do not have parents, priorities or associated controlling tty devices.

There are 9 internal flags that can be printed. Most have meaning only to a system with swapping enabled.

More than one of these flags will commonly be set. A flag that is not set produces a space in the corresponding position.

FLAGS MNEMONIC
W work - cannot be swapped out until work has been done
I inmotion - the text segment is being filled or emptied
O textout - the text segment is on disk already
Z proc0 - this is the swapper itself
L lock - not eligible to be swapped out
M inmem - resident in memory
F fork - creating another process
S sticky - this shared text will be retained until killed
X exec - beginning execution of a different program

Note that 'F' and 'S' overlap in the same column. They are mutually exclusive. Shared text processes don't fork.

If the inode list is requested a table is given listing: the internal flags (FLG), the number of references (REF), the last block accessed (LAST); or -1 for end-of-file, the mounted device it came from (DEV), the inode number on that device (INUM), the mode bits (described under the fstat() system call) expressed in octal (MODE), the number of links to the file (LNK), the file size (SIZE), and the first few block addresses pointed at by the inode proper (ADDRS).

Note that if the mode bits indicate a device (i.e. leading 012 or 016 for character or block respectively) then the number in the address field is the major-minor device code.

RETURNS

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

EXAMPLE

To determine what is happening:

```
% ps -alt
  PID STAT NAME      PRI   PPID   BASE   SIZE   FLAGS     CHANNEL DEV
 5760 shr  tee                    167      9       M
 5759 wait tee         9   5579    176     35       M        a1ce dz1
 5758 run  ps         12   5579    140     27       M           0 dz1
 5678 wait log         8      1    106     17       M        a1aa dz0
 5594 wait log        10      1    135     17   0           a222 dz2
 5579 wait nsh        12      1    268     67   0   M       a1c8 dz1
 5761 done echo     exit(0)
 5502 wait log         8      1    106     17   0           a240 dz3
 5467 wait log        10      1    174     17   0           a1e6 modem1
 3063 wait log         5      1    231     17   0   M       9ed4 net1
 2194 shr  log                    248     20   0   M
   29 shr  sh                     180     26   0
   21 wait sh            5      1    191     39   0           a108 console
   20 shr  sync                   220      5   0   M
   19 wait sync          5      1    214      6   0   M       9ed4 none
    1 wait init          7      0    123     17   0   M       adfa none
    0 doze swap         -3      0      0      2     Z M       9e88 none
```

To show the same information periodically:

```
% ps -alts5
```

FILES

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

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. If a drive cannot be opened restor cycles back to the first one indicated.

-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 -ac /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

stty - set terminal attributes

SYNOPSIS

stty -[bs# cr# c +echo echo ek erase* +even even ff# ht#
+rare rare 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.

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' (ctl-h) '\25' (ctl-u) 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 generate even parity on output. Generate 1 parity if +odd is set.

-even generate 0 parity if -odd is set.

-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#.

-i# set the input baud rate to the speed referenced by the code #. Speed code values are in the range [0, 15], corresponding to the baud rates below:

code	speed
0	0 (hangup)
1	50
2	75
3	110
4	134.5
5	150
6	200
7	300
8	600
9	1200
10	1800
11	2400
12	4800
13	9600
14	19200
15	38400

By no means do all devices support all speeds. Values in the range [16, 65534] have very special meaning to some device handlers and should be used with care.

-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 generate odd parity on output.

-odd generate 0 parity if -even set.

-o# set the output baud rate to the speed referenced by the code #. Values are the same as for "-i#". Only the low 8 bits of # are used. Values in the range [15, 255] have very special meaning to device drivers.

+rare switch to rare mode. Similar to +raw except that DEL, FS, DC1, and DC3 are recognized. Line-editing is not performed. +raw overrides +rare.

-rare reset rare mode.

+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 or X-ON) and DC3 (ctl-S or X-OFF) or 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#". -s0 causes the line to turn off modem signals. (DTR and RTS low). -s1200 is equivalent to -i9 -o9.

-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 no flags are given, stty prints the current status as with the "-t" flag, omitting any switchable characteristics that are the same as the system default.

The sequence "backspace-space-backspace" is sent to the terminal which erases the character on the screen of most CRT terminals. Setting the character to a value greater than 127 inhibits this erase sequence. (I.E. Use "-erase\210" "-kill\225" to inhibit the normal erase sequence.)

If -even and -odd are selected together, the output parity bit is set to 0. If +even and +odd are selected together, the output parity bit is set to 1. Otherwise, +even selects even parity on output +odd selects odd parity on output. Input parity is normally stripped off.

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.

In rare mode, proper parity is generated on output, parity is not stripped off on input, ctl-s (X-OFF) causes output to suspend, ctl-q (X-ON) resumes suspended output, DEL causes an interrupt signal, and ctl-\ causes an quit signal. As in normal mode (-raw and -rare), a ctl-s character is output when a (system dependent) input threshold is achieved, and a ctl-q character is generated when enough of the input is consumed. Thus, rare mode is a hybrid of raw and normal modes.

Note that a change from raw or rare mode to normal mode, or vice-versa, causes all output and input to be discarded, thus defeating any type

ahead. Transitions between rare and raw modes do not have this effect.

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

To display the values set for a terminal:

```
% stty
speed = 1200 baud
erase = \b kill = \25
```

FILES

/adm/stty for predesignated terminal settings.

SEE ALSO

stty(III), tty(III)

NAME

sync - synchronize disk I/O

SYNOPSIS

sync -[#] <files>

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. This should only be done at system startup time.

If <files> are present, sync will open as many of them as it can, to force the system to keep the associated inode in memory. If no -# is specified, # is set to 30 when files are present.

sync should be used religiously before taking the system down, even if the sync daemon is running. An alternate form (date +0) is also safe before shutdown.

RETURNS

sync always returns success.

EXAMPLE

To make sure that disk i/o has been completed:

```
% sync
```

Or, to start a sync daemon from the /adm/init file:

```
m /bin/sync -60 /adm /bin /dev /etc /etc/bin /tmp /usr
```

This should be done after any mounts that are appropriate.

SEE ALSO

date