NAME

70boot − 11/70 bootstrap procedures

DESCRIPTION

To bootstrap programs from a wide range of storage media, the PDP-11/70 has a dedicated diagnostic bootstrap loader called the M9301-YC. The M9301-YC contains two 256 word ROMs (17 765 000 to 17 765 776 and 17 773 000 to 17 773 776) which contain hardware verification diagnostic routines and bootstrap loader routines.

The diagnostic portion tests the basic CPU to verify correct operation. The branches, registers, all addressing modes, and most of the instructions are checked. If requested, memory management and the UNIBUS map are turned on. Then memory is tested from virtual address 001 000 to 157 776 with the cache disabled. Next the cache is enabled and tested.

The physical memory tested is determined by the console switches. Console switches <15:12> are used to set physical address bits <19:16>. If console switches <15:12> are zero, memory management and the UNIBUS map will not be enabled, so that physical memory 0 to 157 776 will be used. If console switches <15:12> are non-zero, then memory management, the UNIBUS map, and 22-bit mapping will be enabled. Table I describes the physical address ranges for each switch setting. In all cases, virtual addresses 160 000 to 177 776 are mapped to the peripheral page, physical addresses 17 600 000 to 17 777 776. Note that physical memory above 512K words is not accessible by this program even though the physical memory maximum is 1920K words.

The bootstrap portion of the M9301-YC attempts to BOOT from the device and drive number specified in the console switches. Console switches <7:3> select the device and console switches <2:0> select the drive number. Table II describes the devices selected for each switch setting. If console switches <7:0> are zero, the program will read a set of switches on the M9301-YC, set by field service, to determine a default boot device and drive number. These switches appear at location 17 773 024, however bits <8:4> select the device and bits <3:1> select the drive number.

Having selected a boot device, the program will read a block of data into memory starting at virtual address 0, and then jump to virtual address 0. Table III describes the details of booting for each device. Note that the physical address selection is the same as described above for the diagnostic portion. Excluding the RX11/RX01 floppy disk, bootstrap programs must fit in one block of 256 words, even though this program may read in more.

To start operation of the bootstrap loader, halt the CPU by depressing the HALT switch, set the Address Display select switch to Console Physical, set the Console Switch Register to 165 000, and depress the Load Address switch. Then reset the console switches to 0 and set switches <15:12> for the desired physical memory (normally 0) and switches <7:0> for the desired device (normally 0 for the default boot). Put the HALT switch in the ENABLE position and depress the START switch. The diagnostic portion will then run followed by the boot from the selected media. This takes approximately three seconds.

Any error during the diagnostic portion will cause the CPU to halt. Table IV lists the addresses and error indications. Only cache errors are recoverable in that by pressing the CONTINUE switch the program will disable the cache by forcing misses and proceed to the bootstrap section. If there is an error in reading the boot block, the program will do a RESET instruction and jump back to the memory test section (test 24) and then attempt to boot again.

SEE ALSO

unixboot(VIII)

Table I − Physical Memory Selection

Console switches <15:12>	Physical addresses
00	00 000 000 - 00 157 776
01	00 200 000 - 00 357 776
02	00 400 000 - 00 557 776
03	00 600 000 - 00 757 776
04	01 000 000 - 01 157 776
05	01 200 000 - 01 357 776
06	01 400 000 - 01 557 776
07	01 600 000 - 01 757 776
10	02 000 000 - 02 157 776
11	02 200 000 - 02 357 776
12	02 400 000 - 02 557 776
13	02 600 000 - 02 757 776
14	03 000 000 - 03 157 776
15	03 200 000 - 03 357 776
16	03 400 000 - 03 557 776
17	03 600 000 - 03 757 776

Table II − Device selection

Console switches <7:3>	Device	
00	illegal	
01	TM11/TU10	Magnetic tape
02	TC11/TU56	DECtape
03	RK11/RK05	Disk pack
04	RP11/RP03	Disk pack
05	reserved	
06	RH70/TU16	Magnetic tape
07	RH70/RP04	Disk pack
10	RH70/RS04	Fixed head disk
11	RX11/RX01	Diskette
12-37	illegal	

Table III − Boot procedures

TU10:	Select drive, wait until online, set to 800 bpi, rewind, space forward 1 record, read 1 record (maximum of 256 words).
TU56:	Select drive, rewind, read 512 words.
RK05 or RP03:	Select drive, start at block 0, read 512 words.
TU16:	Select drive on first TM02, wait until online, set to 800 bpi, PDP format, rewind, space forward 1 record, read 1 record (maximum of 512 words).
RP04:	Select drive, read-in preset, set to 16-bits/word, ECC inhibit, start at block 0, read 512 words.
RS04:	Select drive, start at block 0, read 512 words.
RX01:	Select drive 0 or 1, start at track 1, sector 1 (IBM standard), read 64 words.

Table IV − Error halts

Address displayed	Test	Subsystem under test
17 765 004	1	Branch
17 765 020	2	Branch
17 765 036	3	Branch
17 765 052	4	Branch
17 765 066	5	Branch
17 765 076	6	Branch
17 765 134	7	Register data path
17 765 146	10	Branch
17 765 166	11	CPU instruction
17 765 204	12	CPU instruction
17 765 214	13	CPU instruction
17 765 222	14	CPU instruction
17 765 236	14	CPU instruction
17 765 260	15	CPU instruction
17 765 270	16	Branch
17 765 312	16	CPU instruction
17 765 346	17	CPU instruction
17 765 360	20	CPU instruction
17 765 374	20	CPU instruction
17 765 450	21	Kernel PAR
17 765 474	22	Kernel PDR
17 765 510	23	JSR
17 765 520	23	JSR
17 765 530	23	RTS
17 765 542	23	RTI
17 765 550	23	JMP
17 765 742	25	Main memory data compare error
17 765 760	25	Main memory data compare error
17 776 000	25	Main memory parity error; no recovery possible from this error
17 773 644	26	Cache memory data compare error
17 773 654	26	Cache memory no hit, recoverable
17 773 736	27	Cache memory data compare error
17 773 746	27	Cache memory no hit, recoverable
17 773 764	25/26	Cache memory parity error, recoverable

NAME

ac − login accounting

SYNOPSIS

/etc/ac −sys [**−w** wtmp] [**−p** [people]]

DESCRIPTION

Ac produces a printout giving the connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. *sys* specifies the name of the system (e.g., A) on which *ac* is run. This information is used for the output heading. **−w** is used to specify an alternate *wtmp* file. **−p** without *people* prints all individual totals; without this option, only system totals are printed. Any *people* specified after the **−p** option will limit the printout to only the specified login names. If no *wtmp* file is given, */usr/actg/data/wtmp* is used.

Ac also updates the file */usr/actg/data/log* with the last date on which each user-id was logged into the system, and */usr/actg/data/uacct* with any fees from the previous day, by reading those fees from */usr/actg/data/fee.*

The accounting file */usr/adm/wtmp* is maintained by *init* and *login.* Neither of these programs creates the file, so if it does not exist, no connect-time accounting is done. To start accounting, that file should be created with length 0. On the other hand, if the file is left undisturbed it will grow without bound, so the desired information should be collected periodically, and the file truncated.

Normally, */usr/adm/wtmp* should be copied into the file specified by the **−w** option (or, in the absence of that option, into */usr/actg/data/wtmp*) before *ac* is invoked; this is also the time at which */usr/adm/wtmp* should be normally truncated.

FILES

/usr/adm/wtmp
/usr/actg/data/fee
/usr/actg/data/uacct
/usr/actg/data/log

SEE ALSO

init(VIII), login(I), wtmp(V)

NAME

bcopy − disk block copy

SYNOPSIS

bcopy

DESCRIPTION

Bcopy is an interactive command that allows disk blocks (512 bytes/block) to be selectively copied from one file to another file. When initially invoked, *bcopy* requires the user to supply responses to the following descriptor fields:

to Full path name of the *destination* location, which can be either a special file or an ordinary file. If the file does not exist, *bcopy* will create it.

offset A *decimal* integer that represents the relative position (block address) in the *destination* location at which the copy is to start.

from Full path name of *source* file, which can be either a special file or an ordinary file.

offset A *decimal* integer that represents the relative position (block address) in the *source* location at which the copy is to start.

count A *decimal* integer that represents the number of disk blocks that is to be copied.

The command terminates on receiving either a new-line character in response to the **to** prompt, or a DEL.

DIAGNOSTICS

‘‘Can’t create’’ message can occur due to restricted read/write permissions, etc.
‘‘bad character in number’’ because of a non-numeric character in a numeric argument.

NAME

check − file system consistency check

SYNOPSIS

/etc/check [**−ib** [numbers]] [filesystem]

DESCRIPTION

Check examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. It also reads directories and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a check of the default file systems is performed. The normal output of *check* includes a report of

The number of blocks missing; i.e. not in any file nor in the free list,
The number of special files,
The total number of files,
The number of large files,
The number of directories,
The number of indirect blocks,
The number of blocks used in files,
The highest-numbered block appearing in a file,
The number of free blocks.

The occurrence of **i** *n* times in a flag argument **−ii...i** causes *check* to store away the next *n* arguments which are taken to be i-numbers. When any of these i-numbers is encountered in a directory a diagnostic is produced, as described below, which indicates among other things the entry name.

Likewise, *n* appearances of **b** in a flag like **−bb...b** cause the next *n* arguments to be taken as block numbers which are remembered; whenever any of the named blocks turns up in a file, a diagnostic is produced.

FILES

/etc/checklist

SEE ALSO

checklist(V), fs(V), clri(VIII), clrm(VIII), crash(VIII), fsdb(VIII), restor(VIII)
PWB/UNIX Operations Manual by M. E. Pearlman
Repairing Damaged PWB/UNIX File Systems by P. D. Wandzilak

DIAGNOSTICS

If a read error is encountered, the block number of the bad block is printed and *check* exits. ‘‘Bad freeblock’’ means that a block number outside the available space was encountered in the free list. ‘‘*n* dups in free’’ means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

An important class of diagnostics is produced by a routine which is called for each block which is encountered in an i-node corresponding to an ordinary file or directory. These have the form

b# complaint **; i=** *i#* **(***class* **)**

Here *b#* is the block number being considered; *complaint* is the diagnostic itself. It may be

blk if the block number was mentioned as an argument after **−b;**
bad if the block number has a value not inside the allocatable space on the device, as indicated by the device’s super-block;
dup if the block number has already been seen in a file;
din if the block is a member of a directory, and if an entry is found therein whose i-number is outside the range of the i-list on the device, as indicated by the i-list size specified by the super-block.

Unfortunately this diagnostic does not indicate the offending entry name, but since the i-number of the directory itself is given (see below) the problem can be tracked down.

The *i#* in the form above is the i-number in which the named block was found. The *class* is an indicator of what type of block was involved in the difficulty:

sdir indicates that the block is a data block in a small file;
ldir indicates that the block is a data block in a large file (the indirect block number is not available);
idir indicates that the block is an indirect block (pointing to data blocks) in a large file;
free indicates that the block was mentioned after **−b** and is free;
urk indicates a malfunction in *check.*

When an i-number specified after **−i** is encountered while reading a directory, a report is given in the form

i# **ino; i=** *d#* **(***class* **)** *name*

where *i#* is the requested i-number. *d#* is the i-number of the directory, *class* is the class of the directory block as discussed above (virtually always ‘‘sdir’’) and *name* is the entry name. This diagnostic gives enough information to find a full path name for an i-number. The**−i** *n* option is used to find an entry name and the i-number of the directory containing the reference to *n,* then recursively use **−i** on the i-number of the directory to find its name.

Another important class of file system diseases indicated by *check* is files for which the number of directory entries does not agree with the link-count field of the i-node. The diagnostic is hard to interpret. It has the form

i# delta

Here *i#* is the i-number affected. *Delta* is an octal number accumulated in a byte, and thus can have the value 0 through 377(8). The easiest way (short of rewriting the routine) of explaining the significance of *delta* is to describe how it is computed.

If the associated i-node is allocated (that is, has the *allocated* bit on) add 100 to *delta.* If its link-count is non-zero, add another 100 plus the link-count. Each time a directory entry specifying the associated i-number is encountered, subtract 1 from *delta.* At the end, the i-number and *delta* are printed if *delta* is neither 0 nor 200. The first case indicates that the i-node was unallocated and no entries for it appear; the second that it was allocated and that the link-count and the number of directory entries agree.

Therefore (to explain the symptoms of the most common difficulties) *delta* = 377 (−1 in 8-bit, 2’s complement octal) means that there is a directory entry for an unallocated i-node. This is somewhat serious and the entry should be be found and removed forthwith. *Delta* = 201 usually means that a normal, allocated i-node has no directory entry. This difficulty is much less serious. Whatever blocks there are in the file are unavailable, but no further damage will occur if nothing is done. A *clri* followed by a *icheck −s* will restore the lost space at leisure.

In general, values of *delta* equal to or somewhat above 0, 100, or 200 are relatively innocuous; just below these numbers there is danger of spreading infection.

BUGS

Since *check* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.
It believes even preposterous super-blocks and consequently can get core images.

NAME

clri − clear i-node

SYNOPSIS

/etc/clri i-number [filesystem]

DESCRIPTION

Clri writes zeros on the 32 bytes occupied by the i-node numbered *i-number.* If the *filesystem* argument is given, the i-node resides on the given device, otherwise on a default file system. The *filesystem* argument must be a special file name referring to a device containing a file system. After *clri*, any blocks in the affected file will show up as ''missing'' in an *check* of the *filesystem.*

Read and write permission is required on the specified *filesystem* device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

SEE ALSO

clrm(VIII), check(VIII)

BUGS

Whatever the default file system is, it is likely to be wrong. Specify the *filesystem* explicitly.

If the file is open, *clri* is likely to be ineffective.

Clri is not as handy as *clrm(VIII).*

NAME

clrm − clear mode of i-node

SYNOPSIS

/etc/clrm filesystem i-number ...

DESCRIPTION

Clrm writes a zero in the two byte mode of the associated i-node numbered *i-number.* The *filesystem* argument must be a special file name referring to a device containing a file system. More than one *i-number* may be given. After*clrm,* an y blocks in the affected file will show up as ‘‘missing’’ in a *check* of the *filesystem.*

The primary purpose of this routine is to remove a file which does not appear in a directory. Internal checks are performed on each *i-number* to protect the *filesystem* against clearing either a directory or a special file.

Since *clrm* does not destroy the i-node’s original disk block address when invoked, the i-node, if cleared by mistake, may then be salvaged more efficiently.

SEE ALSO

check(VIII), clri(VIII)

NAME

config − configure a system

SYNOPSIS

/etc/config [**−l** file] [**−c** file] [**−m** file] dfile

DESCRIPTION

Config is a program that takes a description of a PWB/UNIX system and generates two files. One file is an assembler program containing the contents of low core (octal addresses 0-0777), showing all of the interrupt and trap vectors, as well as the run-time interface to C programs. The other file is a C program defining the configuration tables for the various types of devices on the system.

The **−l** option specifies the name of the output file containing the assembler program; *low.s* is the default name.

The **−c** option specifies the name of the output file containing the C program; *conf.c* is the default name.

The **−m** option specifies the name of the file that contains all the information regarding supported devices; */etc/master* is the default name. This file is supplied with the PWB/UNIX system and should *not* be modified unless the user *fully* understands its construction.

The user must supply *dfile;* it must contain device information for the user's system. This file is divided into two parts. The first part contains physical device specifications. The second part contains system-dependent information. Any line with an asterisk (*) in column 1 is treated as a comment.

All configurations are assumed to have the following devices:

one dl 11 (for the console)
one kw11-l line clock or kw11-p programmable clock

with standard interrupt vectors and addresses. These two devices *must not* be specified in the *dfile.* Note that UNIX needs only one clock, but can handle both types.

First part of *dfile:*

Each line contains four or five fields, delimited by blanks and/or tabs in the following format:

devname vector address bus number

where *devname* is the name of the device (as it appears in the */etc/master* device table), *vector* is the interrupt vector location (octal), *address* is the device address (octal), *bus* is the bus request level (4 through 7), and *number* is the number (decimal) of devices associated with the corresponding controller; *number* is optional, and if omitted, a default value which is the maximum value for that controller is used.

Second part of *dfile:*

The second part contains three different types of lines. Note that *all* specifications of this part *are required,* although their order is arbitrary.

1. *root / dump device specification*

Two lines of three fields each:

root devnameminor
dump devnameminor

where *minor* is the minor device number (in octal).

2. *swap device specification*

One line that contains five fields as follows:

swap devname minor swplo nswap

where *swplo* is the lowest disk block (decimal) in the swap area and *nswap* is the number of disk blocks (decimal) in the swap area.

3. *parameter specification*

Ten lines of two fields each as follows (*number* is decimal):

buffers	number
inodes	number
files	number
mounts	number
coremap	number
swapmap	number
calls	number
procs	number
texts	number
clists	number

EXAMPLE

Suppose we wish to configure a system with the following devices:

one rp04 controller with 6 drives
one dh11 with 16 lines (default number)
one dm11 with 16 lines (for the dh11)
one dh11 with 8 lines
one dm11 with 8 lines (for the dh11)
one lp11
one tu16 controller with 2 drives
one dl 11

Note that PWB/UNIX only supports dh11 units that require corresponding dm11 units. It is wise to specify them in dh-dm pairs to facilitate understanding the configuration. Note also that, in the preceding case, the dl 11 that is specified is *in addition* to the dl 11 that was part of the initial system. We must also specify the following parameter information:

root device is an rp04 (drive 0, section 0)
swap device is an rp04 (drive 1, section 4),
with a swplo of 6000 and an nswap of 2000
dump device is a tu16 (drive 0)
number of buffers is 40
number of processes is 150
number of mounts is 15
number of inodes is 120
number of files is 120
number of calls is 30
number of texts is 35
number of character buffers is 150
number of coremap entries is 50
number of swapmap entries is 50

The actual system configuration would be specified as follows:

```
rp04    254     776700      5       6
dh11    320     760020      5
dm11    300     770500      4
dh11    330     760040      5       8
dm11    304     770510      4       8
lp11    200     775514      5
tu16    224     772440      5       2
dl 11   350     775610      5
root    rp04    00
swap    rp04    14          6000    2000
dump    tu16    0
*
* Comments can be inserted in this manner
*
buffers 40
procs   150
mounts  15
inodes  120
files   120
calls   30
texts   35
clists  150
coremap 50
swapmap 50
```

FILES

/etc/master	default input master device table
low.s	default output low core assembler code (in the current directory)
conf.c	default output configuration tables − C code (in the current directory)

SEE ALSO

master(V)
Setting up PWB/UNIX by R. C. Haight, W. D. Roome, and L. A. Wehr

DIAGNOSTICS

Diagnostics are routed to the standard output and are self-explanatory.

NAME

crash − what to do when the system crashes

DESCRIPTION

This section gives at least a few clues about how to proceed if the system crashes. It can't pretend to be complete.

How to bring it back up. If the reason for the crash is not e vident (see below for guidance on 'evident') you may want to try to dump the system if you feel up to debugging. At the moment a dump can be taken only on magtape. With a tape mounted and ready, stop the machine, load address 44, and start. This should write a copy of all of core on the tape with an EOF mark.

In restarting after a crash, always bring up the system single-user. This is accomplished by following the directions in Operations Manual; a single-user system is indicated by having a particular value in the switches (173030 unless you've changed *init)* as the system starts executing. When it is running, perform a*c heck(VIII)* on all file systems which could have been in use at the time of the crash. If any serious file system problems are found, they should be repaired. When you are satisfied with the health of your disks, check and set the date if necessary, then come up multi-user. This is most easily accomplished by changing the single-user value in the switches to something else, then logging out by typing an EOT.

To even boot UNIX at all, three files (and the directories leading to them) must be intact. First, the initialization program */etc/init* must be present and executable. If it is not, the CPU will loop in user mode at location 6. For *init* to work correctly, */dev/tty8* and */bin/sh* must be present. If either does not exist, the symptom is best described as thrashing. *Init* will go into a *fork/e xec* loop trying to create a Shell with proper standard input and output.

If you cannot get the system to boot, a runnable system must be obtained from a backup medium. The root file system may then be doctored as a mounted file system as described below. If there are any problems with the root file system, it is probably prudent to go to a backup system to avoid working on a mounted file system.

Repairing disks. The first rule to k eep in mind is that an addled disk should be treated gently; it shouldn't be mounted unless necessary, and if it is very valuable yet in quite bad shape, perhaps it should be dumped before trying surgery on it.

The problems reported by *check* typically fall into two kinds. There can be problems with the free list: duplicates in the free list, or free blocks also in files. These can be cured easily with an *icheck −s[drive type],* where the drive type corresponds to either the number 3 or 4 (RPO*3,* RPO*4).* If the same block appears in more than one file or if a file contains bad blocks, the files can be deleted, and the free list reconstructed. The best way to delete a file containing bad blocks is to use *clrm(VIII),* then remove its directory entries. Files that contain duplicate blocks are done differently. All the file's i-numbers that reference the duplicate block are first identified by *check −b [block #] device.* Their names can then be found by *ncheck −i* and removed after the file system is mounted.

Finally, there may be inodes reported by *check* that have 0 links and 0 entries. These occur on the root device when the system is stopped with pipes open, and on other file systems when the system stops with files that have been deleted while still open. A *clrm* will free the inode, and an *icheck −s[drive type]* will recover any missing blocks.

Why did it crash? UNIX types a message on the console terminal when it voluntarily crashes. Here is the current list of such messages, with enough information to provide a hope at least of the remedy. The message has the form 'panic: ...', possibly accompanied by other information. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

blkdev

The *getblk* routine was called with a nonexistent major device as argument. Definitely hardware or software error.

devtab
: Null device table entry for the major device used as argument to *getblk*. Definitely hardware or software error.

timeout table overflow
: Too many entries in timeout table.

iinit
: An I/O error reading the super-block for the root file system during initialization.

no fs
: A device has disappeared from the mounted-device table. Definitely hardware or software error.

no imt
: Like ‘no fs’, but produced elsewhere.

no clock
: During initialization, neither the line nor programmable clock was found to exist.

I/O error in swap
: An unrecoverable I/O error during a swap. Really shouldn’t be a panic, but it is hard to fix.

out of swap space
: A program needs to be swapped out, and there is no more swap space. It has to be increased. This really shouldn’t be a panic, but there is no easy fix.

trap
: An unexpected trap has occurred within the system. This is accompanied by three numbers: a ‘ka6’, which is the contents of the segmentation register for the area in which the system’s stack is kept; ‘aps’, which is the location where the hardware stored the program status word during the trap; and a ‘trap type’ which encodes which trap occurred. The trap types are:

  0 bus error
  1 illegal instruction
  2 BPT/trace
  3 IOT
  4 power fail
  5 EMT
  6 recursive system call (TRAP instruction)
  7 programmed interrupt request
  8 floating point trap
  9 segmentation violation

In some of these cases it is possible for octal 20 to be added into the trap type; this indicates that the processor was in user mode when the trap occurred. If you wish to examine the stack after such a trap, either dump the system, or use the console switches to examine core; the required address mapping is described below.

Interpreting dumps. All file system problems should be taken care of before attempting to look at dumps. The dump should be read into the file */sys/sys/core; cp*(I) will do. At this point, you should execute *ps −alxk* and *who* to print the process table and the users who were on at the time of the crash. You should dump (*od*(I)) the first 30 bytes of */sys/sys/core.* Starting at location 4, the registers R0, R1, R2, R3, R4, R5, SP and KDSA6 (KISA6 for 11/40s) are stored. Next, take the value of KA6 (location 22(8) in the dump) multiplied by 100(8) and dump USIZE bytes starting from there. This is the per-process data associated with the process running at the time of the crash. Relabel the addresses 140000. R5 is C’s frame or display pointer. Stored at (R5) is the old R5 pointing to the previous stack frame. At (R5)+2 is the saved PC of the calling procedure. Trace this calling chain. Each PC should be looked up in the system’s name list using *db*(I) and its ‘:’ command, to get a reverse calling order. In most cases this procedure will give an idea of what is wrong.

SEE ALSO

check(VIII), dcheck(VIII), icheck(VIII), ncheck(VIII), clri(VIII), clrm(VIII), fsdb(VIII)
Repairing Damaged PWB/UNIX File Systems by P. D. Wandzilak
PWB/UNIX Operations Manual by M. E. Pearlman

NAME

cron − clock daemon

SYNOPSIS

/etc/cron

DESCRIPTION

Cron executes commands at specified dates and times according to the instructions in the file /usr/lib/crontab. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file /etc/rc; see *init*(VIII).

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1=monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by *cron* every hour. Thus it could take up to an hour for entries to become effective. If it receives a hangup signal, however, the table is examined immediately; so ‘kill −1 ...’ can be used.

FILES

/usr/lib/crontab
/usr/lib/cronlog log of commands executed

SEE ALSO

init(VIII), setuid(VIII), sh(I), kill(I)

DIAGNOSTICS

None − illegal lines in crontab are ignored.

BUGS

A more efficient algorithm could be used. The overhead in running *cron* is about one percent of the machine, exclusive of any commands executed.

NAME

cu − call UNIX

SYNOPSIS

cu telno [**−t**] [**−s** speed] [**−a** acu] [**−l** line]

DESCRIPTION

Cu calls up another UNIX system, a terminal, or possibly a non-UNIX system. It manages an interactive conversation with possible transfers of text files. *Telno* is the telephone number with minus signs at appropriate places for delays. The **−t** flag is used to dial out to a terminal. *Speed* gives the transmission speed (110, 134, 150, 300, 1200); 300 is the default value.

The **−a** and **−l** values may be used to specify pathnames for the ACU and communications line devices. They can be used to override the following two built-in choices:

−a /dev/cua0 −l /dev/cul0

After making the connection, *cu* runs as two processes: the *send* process reads the standard input and passes most of it to the remote system; the *receive* process reads from the remote system and passes most data to the standard output. Lines beginning with "~" have special meanings.

The *send* process interprets the following:

~**.**	terminate the conversation.
~EOT	terminate the conversation.
~<file	send the contents of *file* to the remote system, as though typed at the terminal.
~!	invoke an interactive Shell on the local system.
~!cmd ...	run the command on the local system (via **sh −c**).
~$cmd ...	run the command locally and send its output to the remote system.
~%take from [to]	copy file "from" (on the remote system) to file "to" on the local system. If "to" is omitted, the "from" name is used both places.
~%put from [to]	copy file "from" (on local system) to file "to" on remote system. If "to" is omitted, the "from" name is used both places.
~~**...**	send the line "~**...**".

The *receive* process handles output diversions of the following form:

~>[>][**:**]file
zero or more lines to be written to the file
~>

In any case, output is diverted (or appended, if ">>" is used) to the file. If "**:**" is used, the diversion is *silent*, i.e., it is written only to the file. If "**:**" is omitted, output is written both to the file and to the standard output. The trailing "~>" terminates the diversion.

FILES

/dev/cua0
/dev/cul0
/dev/null

SEE ALSO

dh(IV), dn(IV), tty(IV), ln(I), ttys(V)

EXIT CODES

zero for normal exit, nonzero (various values) otherwise.

BUGS

The use of **~%put** requires *stty* and *cat* on the remote side. It also requires that the current erase and kill characters on the remote system be identical to the current ones on the local system. Backslashes are inserted at appropriate places.

The use of **~%take** requires the existence of *echo* and *tee* on the remote system. Also, **stty tabs** mode is required on the remote system if tabs are to be copied without expansion.

In order to use the **−a** and **−l** default values, the */dev/cua0* and */dev/cul0* special files must be linked (see *ln*(I)) to their respective devices. For example, one might link */dev/cua0* and */dev/cul0* to */dev/dn0* and */dev/ttyh*, respectively. Note that, for this example, */dev/ttyh* must be marked ‘‘ignored’’ in the */etc/ttys* file.

NAME

dcat − read/write synchronous line

SYNOPSIS

dcat −t[telephone−number]
dcat −r[telephone−number]

DESCRIPTION

Dcat is designed to be used with medium-speed synchronous dial lines such as the Bell System ‘‘201C’’ 2400 baud modem, and a DEC® interface such as the DU11 (2000 baud/DP11 works too). Assuming a UNIX at each end (one with a DN11 and ACU), *dcat* can be used to transfer files in either direction.

Example 1)

UNIX 1:

```
dcat −t555−1234 <somefile
```

UNIX 2:

```
dcat −r >somefile
```

Example 2)

UNIX 1:

```
dcat −r555−4321 >somefile
```

UNIX 2:

```
dcat −t <somefile
```

FILES

/dev/dn?, /dev/d[pu]?

SEE ALSO

dn(IV), dp(IV)

NAME

dcheck − file system directory consistency check

SYNOPSIS

/etc/dcheck [**−i** numbers] [filesystem]

DESCRIPTION

Dcheck reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

The **−i** flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

DIAGNOSTICS

When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

SEE ALSO

fs(V), check(VIII), icheck(VIII), ncheck(VIII), clri(VIII), clrm(VIII), fsdb(VIII)

BUGS

Since *dcheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

NAME

devnm − device name

SYNOPSIS

/etc/devnm [argument(s)...]

DESCRIPTION

Devnm identifies the special file *special* (e.g., "/dev/rp2"), associated with the mounted file system where the argument name resides.

Devnm prints the unqualified path name of the special file followed by the argument name.

Example:

```
/etc/devnm /unix
rp2 /unix
```

FILES

/dev/rf?, /dev/rk?, /dev/rp?

NAME

diskboot − disk bootstrap programs

DESCRIPTION

There are two versions of the disk bootstrap program, *uboot.* One handles RP03 type drives and the other RP04 type drives. Otherwise, the versions are functionally equivalent.

The program must be located in block 0 of the disk pack. The space available for the program is thus only one block (256 words) which severely constrains the amount of error handling. Block 0 is unused by the UNIX file system, so this does not affect normal file system operation. To boot, the program must be read into memory starting at address 0 and started at address 0. This may be accomplished by standard DEC® ROM bootstraps, special ROM bootstraps, or manual procedures.

After initial load, the program relocates itself to high core as specified when assembled (typically 24K words, maximum of 28K). Next, memory below the program is cleared and the prompt '#' is typed on the console. A two digit field specifying the file system to use is expected. The first digit indicates which drive to use and the second digit which logical section of the pack to use. For example, 24 would correspond to /dev/rp24, or drive 2, section 4. By convention, section 0 always starts at cylinder 0 and is safest to use. No error checking is done on this field, invalid data will cause unpredictable results. Also, there is no error checking on disk reads.

After the file system select, the program prompts with a '='. The user must then enter the UNIX pathname of the desired file. The '#' character will erase the last character typed, the '@' character will kill the entire line, and 'A' through 'Z' is translated to 'a' through 'z'. Also, carriage return (CR) is mapped into line feed (LF) on input, and LF is output as CR-LF. The upper-case to lower-case conversion is used to handle upper-case-only terminals such as the TELETYPE® Model 33 or the DEC LA30. Therefore, a file name with upper case characters cannot be booted using this procedure.

After the name has been completely entered by typing CR or LF, the program searches the file system specified for the pathname. Note, the pathname may be any valid UNIX file system pathname. If the file does not exist, or if the file is a directory or special file, the bootstrap starts over and prompts '#'. Otherwise, the file is read into memory starting at address 0. If address 0 contains 000 407, a UNIX a.out program is assumed and the first 8 words are stripped off by relocating the loaded program toward address 0. Finally, a jump to address 0 is done by executing "jsr pc,*$0".

FILES

/usr/mdec/uboot - disk bootstrap
/sys/source/util/rp03boot.s - RP03 source
/sys/source/util/rp04boot.s - RP04 source

SEE ALSO

a.out(V), fs(V), unixboot(VIII)

NAME

dump − incremental file system dump

SYNOPSIS

/etc/dump [key [arguments] filesystem]

DESCRIPTION

Dump makes an incremental file system dump on magtape of all files changed after a certain date. The *key* argument specifies the date and other options about the dump. *K ey* consists of characters from the set **abcfiu0hds.**

a Normally files larger than 1000 blocks are not incrementally dump; this flag forces them to be dumped.

b The next argument is taken to be the maximum size of the dump tape in blocks (see **s**).

c If the tape overflows, increment the last character of its name and continue on that drive. (Normally it asks you to change tapes.)

f Place the dump on the next argument file instead of the tape.

i the dump date is taken from the entry in the file /etc/dtab corresponding to the last time this file system was dumped with the **−u** option.

u the date just prior to this dump is written on /etc/dtab upon successful completion of this dump. This file contains a date for every file system dumped with this option.

0 the dump date is taken as the epoch (beginning of time). Thus this option causes an entire file system dump to be taken.

h the dump date is some number of hours before the current date. The number of hours is taken from the next argument in *arguments.*

d the dump date is some number of days before the current date. The number of days is taken from the next argument in *arguments.*

s the size of the dump tape is specified in feet. The number of feet is taken from the next argument in *arguments.* It is assumed that there are 9 standard UNIX records per foot. When the specified size is reached, the dump will wait for reels to be changed. The default size is 2200 feet.

If no arguments are given, the *key* is assumed to be **i** and the file system is assumed to be /dev/rp0.

Full dumps should be taken on quiet file systems as follows:

```
dump 0u /dev/rp0
ncheck /dev/rp0
```

The *ncheck* will come in handy in case it is necessary to restore individual files from this dump. Incremental dumps should then be taken when desired by:

```
dump
```

When the incremental dumps get cumbersome, a new complete dump should be taken. In this way, a restore requires loading of the complete dump tape and only the latest incremental tape.

DIAGNOSTICS

If the dump requires more than one tape, it will ask you to change tapes. Reply with a new-line when this has been done. If the first block on the new tape is not writable, e.g., because you forgot the write ring, you get a chance to fix it. Generally, however, read or write failures are fatal.

FILES

/dev/mt0 magtape
/dev/rp0 default file system
/etc/dtab

SEE ALSO

dump(V), restor(VIII), ncheck(VIII)

NAME

fsdb − file system debugger

SYNOPSIS

etc/fsdb special [−]

DESCRIPTION

Fsdb can be used to patch up a damaged file system after a crash. It has conversions to translate block and i-numbers into their corresponding disk addresses. Also included are mnemonic offsets to access different parts of an i-node. These greatly simplify the process of correcting control block entries or descending the file system tree.

Fsdb contains several error checking routines to verify i-node and block addresses. These can be disabled if necessary by invoking *fsdb* with the optional ‘‘−’’ argument or by the use of the ‘‘O’’ symbol. (*Fsdb* reads the i-size and f-size entries from the superblock of the file system as the basis for these checks.)

Numbers are considered decimal by default. Octal numbers must be prefixed with a zero. During any assignment operation, numbers are checked for a possible truncation error due to a size mismatch between source and destination.

Fsdb reads a block at a time and will therefore work with raw as well as block I/O. A buffer management routine is used to retain commonly used blocks of data in order to reduce the number of read system calls. All assignment operations result in an immediate write-through of the corresponding block.

The symbols recognized by *fsdb* are:

#	absolute address
i	convert from i-number to i-node address
b	convert to block address
d	directory slot offset
+,−	address arithmetic
q	quit
>,<	save, restore an address
=	numerical assignment
=+	incremental assignment
=−	decremental assignment
="	character string assignment
O	error checking flip flop
p	general print facilities
f	file print facility
B	byte mode
W	word mode
D	double word mode
!	escape to shell

The print facilities generate a formatted output in various styles. The current address is normalized to an appropriate boundary before printing begins. It advances with the printing and is left at the address of the last item printed. The output can be terminated at any time by typing the delete character. If a number follows the ‘‘p’’ symbol, that many entries are printed. A check is made to detect block boundary overflows since logically sequential blocks are generally not physically sequential. If a count of zero is used, all entries to the end of the current block are printed. The print options available are:

i	print as i-nodes
d	print as directories
o	print as octal words
e	print as decimal words
c	print as characters
b	print as octal bytes

The ‘‘f’’ symbol is used to print data blocks associated with the current i-node. If followed by a number, that block of the file is printed. (Blocks are numbered from zero.) The desired print option letter follows the block number, if present, or the ‘‘f’’ symbol. This print facility works for small as well as large files. It checks for special devices and that the block pointers used to find the data are not zero.

Dots, tabs and spaces may be used as function delimiters but are not necessary. A line with just a newline character will increment the current address by the size of the data type last printed. That is, the address is set to the next byte, word, double word, directory entry or i-node, allowing the user to step through a region of a file system. Information is printed in a format appropriate to the data type. Bytes, words and double words are displayed with the octal address followed by the value in octal and decimal. A ‘‘.B’’ or ‘‘.D’’ is appended to the address for byte and double word values, respectively. Directories are printed as a directory slot offset followed by the decimal i-number and the character representation of the entry name. Inodes are printed with labelled fields describing each element.

The following mnemonics are used for i-node examination and refer to the current working i-node:

md	mode
ln	link count
uid	user id number
gid	group id number
s0	high byte of file size
s1	low word of file size
a#	data block numbers (0 − 7)
at	access time
mt	modification time
maj	major device number
min	minor device number

EXAMPLES

386i	prints i-number 386 in an i-node format. This now becomes the current working i-node.
ln=4	changes the link count for the working i-node to 4.
ln=+1	increments the link count by 1.
fc	prints, in ascii, block zero of the file associated with the working i-node.
a0b.p16o	would print the first 16 block numbers for the file if it were large. (If it were small, this would print the first 16 words of the file.)
1i.fd	prints the first 32 directory entries for the root i-node of this file system.
d5i.fc	changes the current i-node to that associated with the 5th directory entry (numbered from zero) found from the above command. The first 512 bytes of the file are then printed in ascii.

1b.p0o	prints the superblock of this file system in octal.
1i.a0b.d7=3	changes the i-number for the seventh directory slot in the root directory to 3. This example also shows how several operations can be combined on one command line.
d7.nm="name"	changes the name field in the directory slot to the given string. Quotes are optional when used with ‘‘nm’’ if the first character is alphabetic.

SEE ALSO

directory(V), fs(V)

NAME

getty − set terminal mode

SYNOPSIS

/etc/getty [char]

DESCRIPTION

Getty is invoked by *init*(VIII) immediately after a terminal is opened following a dial-up. It reads the user's name and invokes the *login*(I) command with the name as argument. While reading the name,*g etty* attempts to adapt the system to the speed and type of terminal being used.

Init calls *getty* with an argument specified by the *ttys* file entry for the terminal line. Arguments other than '0' can be used to make *getty* treat the line specially. Normally, it sets the speed of the interface to 300 baud, specifies that raw mode is to be used (break on every character), that echo is to be suppressed, and either parity allowed. It types the ''login:'' message, which includes the characters which put the Terminet 300 terminal into full-duplex and return the GSI terminal to non-graphic mode. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the ''break'' (''interrupt'') key. The speed is then changed to 150 baud and the ''login:'' is typed again, this time including the character sequence that puts a TELETYPE® Model 37 into full-duplex. If a subsequent null character is received, the speed is changed back to 300 baud.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *stty*(II)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, login is called with the user's name as argument.

SEE ALSO

init(VIII), login(I), stty(II), ttys(V)

NAME

glob − generate command arguments

SYNOPSIS

/etc/glob command [arguments]

DESCRIPTION

Glob is used to expand arguments containing ‘‘*’’, ‘‘[’’, or ‘‘?’’, in the same fashion as the Shell does. *Glob* is passed the argument list containing the metacharacters; *glob* expands the list and calls the indicated command. The actions of *glob* are detailed in the Shell writeup.

If the command name is a simple name containing no ‘/’, *glob* expects argument 0 to be a sequence of directories separated by ‘:’, i.e., a string of the same form as the Shell’s $p variable. It uses this argument to perform directory searching in the same way as the shell.

SEE ALSO

sh(I), pexec(III)

DIAGNOSTICS

‘‘No match’’ if none of the arguments produces a successful filename match.
‘‘Arg list too long’’ if too many (more than 5120) characters of arguments are generated.
Other messages are listed in *pexec(III)*.

NAME

hasp − PWB/UNIX IBM Remote Job Entry

SYNOPSIS

/usr/hasp/haspinit

/usr/hasp/hasphalt

DESCRIPTION

Hasp is the communal name for a collection of programs and a file organization that allow PWB/UNIX, equipped with an appropriate driver for the DQS11-B, to communicate with IBM's Job Entry Subsystems by mimicking an IBM 2770 remote station.

Hasp is initiated by the command *haspinit* and is terminated gracefully by the command *hasphalt.* While active, *hasp* runs in background and requires no human supervision. It quietly transmits to the IBM system jobs that have been queued by the command *send*(I) and messages that have been entered by the command *rjestat*(I). It receives from the IBM system print and punch data sets and message output. It enters the data sets into the proper PWB/UNIX directory and notifies the appropriate user of their arrival. It scans the message output to maintain a record on each of its jobs. It also makes these messages available for public inspection, so that *rjestat*(I), in particular, may extract responses.

Unless otherwise specified, all files and commands described below live in directory */usr/hasp* (first exceptions: *send* and *rjestat*).

There are two sources of data that is to be transmitted by *hasp* from PWB/UNIX to an IBM System/370. In both cases, the data is organized as files in *ebcdic*(V) format. The first is a single file *haspmesg* that is reserved for message input. It is written by the enquiry command *rjestat*(I) and is assigned a priority for transmission. The second source, containing the bulk of the data, consists of jobs that have been entered into the *xmit∗* queue by the program *haspqer.* On completion of processing, *send* invokes *haspqer.* As each file is queued, a subordinate *info/logx∗* file is created to save the name, numeric id, login directory, and tty letter of the user who is doing the queueing. Upon successful transmission of the data to the IBM system, *haspdisp* will move this information into the *jobsout* file and delete the *info/logx∗* file.

Each time *haspinit* is invoked, the *xmit∗* queue is compacted, along with the associated *info/logx∗* files, and its beginning and end are calculated. A three-digit sequence number specifying the first free slot at the end of the queue is written to file *haspstat.* This number is subsequently updated by *haspqer* each time that a new job is entered into the queue. A pointer to the beginning of the queue is maintained by *haspmain.* It is periodically compared to the current end of the queue to determine whether any jobs are waiting to be transmitted. A null lock-file *hasplock* is created with mode zero to prevent simultaneous updating of *haspstat.*

In anticipation of receiving output, *hasp* always maintains a vacant file *tmp∗* in its own directory. Output from the IBM system is initially written into this file and is classified as either a print data set, a punch data set, or message output. Print output is converted to an ASCII text file, with standard tabs. Form feeds are suppressed, but the last line of each page is distinguished by the presence of an extraneous trailing space. Punch output is converted to EBCDIC format. This classification and both conversions occur as the output is received; *tmp∗* files are moved or copied into the appropriate user's directory and assigned the name *prnt∗* or *pnch∗*, respectively, or placed into user directories under user-specified names, or used as input to programs to be automatically executed, as specified by the user. This process is driven by the ''usr=...'' specification. *Hasp* retains ownership of these files and permits read-only access to them. Files of message output are digested by *hasp* immediately and are not retained.

A record is maintained for each job that passes through *hasp.* Identifying information is extracted contextually from files transmitted to and received from the IBM system. From each file transmitted, *hasp* extracts the job name, the programmer's name, the user name, the destination directory name, and the message level. This information is temporarily stored, in the order of submission of jobs, in file*jobsout.* It is retrie ved, by job name and programmer's name, when the IBM system acknowledges the job and assigns a number to it.

The IBM system automatically returns an acknowledgement message for each job it receives. Other status messages are returned in response to enquiries entered by users and in response to enquiries that *hasp* itself generates every ten minutes. All messages received by *hasp* are appended to the *resp* file. The *resp* file is automatically truncated when it reaches 32,000 bytes. Each sequence of enquiries written to the message file *haspmesg* should be preceded by an identification card image of the form ''/*$UX<process id>''. The IBM system will echo back the first portion of this card image, as this is an illegal command. The appearance of process ids in the response stream permits responses to be passed on to the proper users. *Hasp* enters process id zero on all enquiries it generates on its own behalf.

While it is active, *hasp* occupies at least the two process slots that are appropriated by *haspinit.* These slots are used to run *haspmain,* that supervises data transfers, as well as *haspdisp,* that performs dispatching functions; these two processes are connected by a pipe. The function of *haspmain* is to cycle repetitively, looking for data to transfer either to or from the IBM system. When it finds some, it spawns a child process, either *haspxmit* or *hasprecv,* to effect the transfer. It waits for its child to complete its task and then passes an event notice to *haspdisp. Haspmain* exits normally as soon as it detects the file *haspstop* (created by *hasphalt*), and exits reluctantly whenever it encounters a run of errors. An attempt is made to manage the null file *haspdead* so that it exists precisely when *haspmain* is not executing. *Haspinit* has the capability of *dialing* any remote IBM system with the proper hardware and software configuration. A file *haspsoff* is created by *hasphalt* to signal that the phone should be hung up by *haspmain.*

Ordinarily, *haspdisp* waits for event completion notices from *haspmain. Haspdisp* follows up the events described by directing output files, updating records, and notifying users. It may spawn the program *haspcopy* to copy output across file systems. *Haspdisp* references the system files */etc/passwd* and */etc/utmp* to correlate user names, numeric ids, and terminals. Normal termination of *haspmain* causes *haspdisp* to exit also. In the case of error termination, *haspdisp* delays about one minute and then reboots RJE by executing *haspinit* again.

Event notices begin with a one-digit code. The code ''0'' alone signals normal termination. Other event notices consist of a code in the range ''1'' to ''6,'' followed by the name of a file in the */usr/hasp* directory. Notices are issued as each file in the *xmit∗* queue is transmitted and as each *tmp∗* file is filled with output. These files are moved to new temporary names before the event notice is composed. Transmitted files (code 1) are renamed *zmit∗* and output files (codes 3-5) are renamed *prt∗, pch∗,* or *msg∗,* depending on their type. When *haspdisp* gets around to following up on the events described, the files will either be deleted or moved to a permanent destination.

Event notices are written to the *log* file at the time they are received by *haspdisp.* A typical section of the log looks as follows:

```
1zmit283
5msg61
1zmit284
5msg62
3prt63
```

Additional lines are written to the log by *haspinit.* Each reboot of *haspinit* is marked by a time stamp. If the previous execution of *haspmain* ended in error, an exception notice precedes the time stamp. Exception notices are formatted by *haspmain* and consist of a sequence of capital letters. The most common is ''AAAAA'', that indicates five successive failures to acquire the line for a transmission to the host. A sequence of time stamps alternating with ''AAAAA'' indicates that the host is not responding to RJE. Each time the RJE facility is booted via the *haspinit* program, the *log* file is cleaned out. A copy of its last contents is placed in a file named *slog.*

Several RJE programs, including the *send*(I) command, use the *ustat*(II) system call to determine the remaining capacity of the file systems they use, in an attempt to keep roughly 1,000 blocks and 50 i-nodes free. *Haspinit* issues a warning when fewer than 2,000 blocks or 100 i-nodes are available. *Send* shuts down when only 1,500 blocks remain, and *haspmain* stops accepting output from the IBM system when the capacity falls to 1,200 blocks. In addition, output files are limited in size at all times to a maximum of 512K bytes. Of this, only 64K bytes are guaranteed. How much more output will be accepted depends on the current capacity

of the *hasp* file system. Excess data is simply discarded, with no provision for retrieving it.

Most *hasp* files and directories are protected from unauthorized tampering. The exception is the *pool* directory, that is provided so that *send*(I) can create temporary files in the correct file system. *Haspqer* and *rjestat*(I), the user's interfaces to *hasp,* operate in *setuid* mode to contribute the necessary permission modes. *Rjestat*(I), incidentally, extends to anyone who can login as *rje* complete freedom to enter console commands. When invoked with a + argument, it suppresses the **d** that begins a display command and allows one to cancel or re-route jobs.

Some minimal oversight of each *hasp* subsystem is required. The *hasp* mailbox should be inspected and cleaned out periodically. The*job* directory should also be check ed. The only files placed there are output files whose destination file systems are out of space. Users should be given a short period of time (say, a day or two), and then these files should be removed. In the source code for the program,*haspdisp* is a *define* statement for the character string **SFILE**. If it is set to 1, all returned jobs for which PWB/UNIX RJE can not find the ''usr=**...**'' specification are placed into the *job* directory. This feature is primarily intended for those PWB/UNIX systems that are connected to the IBM systems whose output format is not known to *haspdisp.*

Usage statistics are recorded in the directory */usr/hasp/usg,* if it exists. Six files will be created and updated. Each will contain data on a per-user-id basis. File *hasp.in.sum* accumulates the number of blocks transmitted by *hasp;* file *hasp.in.cnt* records the number of transmissions; file *hasp.in.max* records the size, in blocks, of the largest job sent. Files *hasp.out.sum, hasp.out.cnt* and *hasp.out.max* contain the same statistics for output received by *hasp.* The program *usage* may be used to print these statistics; ''usage file [user-id1 ...]'' will print out the statistics gathered in *file*. If the optional user-id list is present, only the statistics for these user-ids will be printed.

The configuration table */usr/rje/lines* is accessed by all components of RJE. Its six columns may be labeled ''host'', ''system'', ''directory'', ''prefix'', ''device'', and ''types''. Each line of the table (maximum of six) defines an RJE connection. ''Host'' is the name of a remote computer: **A, B,** or **1110**. ''System'' is a string of capital letters identifying PWB/UNIX systems. The first specifies where the RJE connection is normally terminated; the remainder specify where it may be backed-up to if the primary RJE system goes down. ''Directory'' is the directory name of the servicing RJE subsystem. ''Prefix'' is the string prepended (redundantly) to several crucial files and programs in the directory: *hasp, hasp2, uvac.* ''Device'' is the name of the controlling DQS-11B, with ''*/dev/*'' excised. ''Types'' contains information on the type of connection to make. It contains the logon id and phone number to use when RJE is to automatically*dial* a remote IBM system. An**n** in this field indicates that this entry is not for dial-up. When a dial-up entry is initiated, the phone number found here is automatically dialed, PWB/UNIX RJE logs on, and normal RJE processing occurs. When*hasphalt* is e xecuted, RJE signs off and hangs up the phone automatically. If the first character in the ''types'' field is an **i**, all console status facilities are inhibited (e.g., *rjestat*(I) will not behave like a status terminal, and the ten-minutes automatic status inquiry is inhibited).

The file */usr/rje/sys* contains the single-letter name of the current PWB/UNIX system. An RJE connection will be considered available if this is its primary system or if this is one of its backup systems and the associated directory is mounted. *Send*(I) and *rjestat*(I) select an a vailable connection by indexing on the ''host'' field of the configuration table. *Hasp* programs inde x on the ''prefix'' field. A subordinate directory, *sque,* exists in */usr/rje* for use by *haspdisp* and *shqer* programs. This directory holds those output files that have been designated as standard input to some executable file. This designation is done via the ''usr=...'' specification. *Haspdisp* places the output files here and updates the file *log* to specify the order of execution, arguments to be passed, etc. *Shqer* e xecutes the appropriate files. A program called *compact* compacts the *log* file. It should be executed before *shqer* and RJE have been started.

All HASP programs are reentrant; therefore, if more than one HASP is to be run on a given PWB/UNIX

system, simply link, via *ln*(I), HASP2 program names to HASP names in */usr.*

FILES

configuration-dependent and general-purpose RJE files:

/dev/rjei	DQS11-B
/dev/tty?	terminals
/etc/utmp	list of active users
/etc/passwd	user population
/usr/rje/sys	PWB/UNIX system name, e.g., ‘‘A’’
/usr/rje/lines	PWB/UNIX RJE lines configuration table
/usr/rje/sque/log	log information for *shqer*

user files

∗/.mail	a user’s mailbox
∗/prnt∗	a user’s print data set
∗/pnch∗	a user’s punch data set

hasp files (relative to the *directory* entry in the RJE configuration table):

hasp∗	mostly programs
haspdead	inactive flag
haspsoff	dial-up hang-up signal
haspstop	halt signal
haspmesg	message slot
haspstat	queue end record
hasplock	lockout file
xmit∗	jobs queued
info/logx∗	haspqer loginfo
job/∗	output from jobs whose file systems are out of space
jobsout	fifo job store
tmp∗	output files
log	event log
resp	concatenated responses from the IBM system
status	RJE message of the day
pool/stm∗	*send*(I) temporaries
usg/∗	usage statistics

SEE ALSO

send(I), rjestat(I), rje(IV), ebcdic(V), regen(VIII)
Guide to IBM Remote Job Entry for PWB/UNIX Users by A. L. Sabsevitz.
System Components: IBM 2770 Data Communication System, IBM SRL GA27-3013.
OS/VS2 HASP II Version 4 System Programmer’s Guide IBM SRL GC27-6992.

DIAGNOSTICS

Haspinit provides brief error messages describing obstacles to bringing up *hasp.* They can best be understood in the context of the RJE source code. The most frequently occurring one is ‘‘cannot open /dev/rjei’’. This may occur if the DQS-11B status register shows something other than READY (octal 200). It will also occur if another process already has the DQS-11B open, or if the exclusive use flag (_dqsx+3, _dqsx+73, etc.) has remained set after a close of the DQS-11B.

Once *hasp* has been started, users should assist in monitoring its performance, and should notify operations personnel of any perceived need for remedial action. *Rjestat*(I) will aid in diagnosing the current state of RJE. It can detect, with some reliability, when the far end of the communications line has gone dead, and will report in this case that the host computer is not responding to RJE. It will also attempt to reboot *hasp* if it detects a

prolonged period of inactivity on the DQS-11B.

BUGS

The name *hasp* is an anachronism. It is used only as a collective name and could represent *hasp*, *jes2*, *asp*, etc.

NAME

icheck − file system storage consistency check

SYNOPSIS

/etc/icheck [−**s**dev-code] [−**b** numbers] [filesystem]

DESCRIPTION

Icheck examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. If the file system is not specified, a set of default file systems is checked. The normal output of *icheck* includes a report of

The number of blocks missing; i.e. not in any file nor in the free list,
The number of special files,
The total number of files,
The number of large and huge files,
The number of directories,
The number of indirect blocks, and the number of double-indirect blocks in huge files,
The number of blocks used in files,
The number of free blocks.

The **−s** option causes *icheck* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The **−s** option causes the normal output reports to be suppressed.

The *−s* option allows for creating optimal free-list organization and posts (in the super-block) the current free counts of i-nodes and blocks. The following forms of *dev-code* are supported:

−sk (RK disk)
−s3 (RP03)
−s4 (RP04)
−sBlocks-per-cylinder:Blocks-to-skip (for anything else)

Important note: some PWB/UNIX commands will not work unless the file system has been prepared with the −s option. See *ustat(II).*

Following the **−b** flag is a list of block numbers; whenever any of the named blocks turns up in a file, a diagnostic is produced.

Icheck is faster if the raw version of the special file is used, since it reads the i-list many blocks at a time.

FILES

Currently, /dev/rrp0 is the default file system.

SEE ALSO

ustat(II), fs(V), check(VIII), dcheck(VIII), ncheck(VIII), fsdb(VIII), clri(VIII), clrm(VIII), restor(VIII)

DIAGNOSTICS

For duplicate blocks and bad blocks (which lie outside the file system) *icheck* announces the difficulty, the i-number, and the kind of block involved. If a read error is encountered, the block number of the bad block is printed and *icheck* considers it to contain 0. ''Bad freeblock'' means that a block number outside the available space was encountered in the free list. ''*n* dups in free'' means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

BUGS

Since *icheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

It believes even preposterous super-blocks and consequently can get core images.

NAME

init − process control initialization

SYNOPSIS

/etc/init

DESCRIPTION

Init is invoked inside UNIX as the last step in the boot procedure. Generally its role is to create a process for each terminal on which a user may log in.

First, *init* checks to see if the console switches contain 173030. (This number is likely to vary between systems.) If so, the console terminal**/de v/tty8** is opened for reading and writing and the Shell is invoked immediately. This feature is used to bring up a single-user system. When the system is brought up in this way, the *getty* and *login* routines mentioned below and described elsewhere are not used. If the Shell terminates, *init* starts over looking for the console switch setting.

Otherwise, *init* invokes a Shell, with input taken from the file */etc/rc*. This command file performs housekeeping like removing temporary files, mounting file systems, and starting daemons.

Then *init* reads the file */etc/ttys* and forks several times to create a process for each terminal specified in the file. Each of these processes opens the appropriate terminal for reading and writing. These channels thus receive file descriptors 0 and 1, the standard input and output. Opening the terminal will usually involve a delay, since the *open* is not completed until someone has dialed up and established the carrier on the channel. Then */etc/getty* is called with argument as specified by the last character of the *ttys* file line. *Getty* reads the user's name and invokes *login* (q.v.) to log in the user and execute the Shell.

Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of *init*, which has been waiting for such an event, wakes up and removes the appropriate entry from the file *utmp*, which records current users, and makes an entry in */usr/adm/wtmp*, which maintains a history of logins and logouts. Then the appropriate terminal is reopened and *getty* is reinvoked.

Init catches the *hangup* signal (signal #1) and interprets it to mean that the switches should be examined as in a reboot: if they indicate a multi-user system, the */etc/ttys* file is read again. The Shell process on each line which used to be active in *ttys* but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add phone lines without rebooting the system by changing the *ttys* file and sending a *hangup* signal to the *init* process: use ''kill −1 1.''

FILES

/dev/tty?, /etc/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

SEE ALSO

login(I), kill(I), sh(I), ttys(V), getty(VIII)

NAME

lastcom − search shell accounting records

SYNOPSIS

/etc/lastcom +

/etc/lastcom [+] ttylist

/etc/lastcom [+] −logname

DESCRIPTION

Starting with the most recent entry, *lastcom* searches back through the shell accounting records for usage matching its arguments. Only the most recent 1000 entries are scanned.

The ‘‘+’’ option alone causes all 1000 entries to be printed.

The *ttylist* causes *lastcom* to print the latest entry for each tty (letter or number) specified. With the ‘‘+’’ option, all entries for each tty are listed.

The −*logname* option causes the latest command executed by the user to be printed. Again, the ‘‘+’’ means *all* commands. The *logname* may be entered as an expression as specified in *glob(VIII).*

Examples:

/etc/lastcom 012345678
lists the last command completed by each of the ttys 0−8.

/etc/lastcom + "−???"
lists all commands executed by users with 3 character login names.

FILES

/etc/sha

SEE ALSO

sh(I), sha(V), glob(VIII)

NAME

mkfs − construct a file system

SYNOPSIS

/etc/mkfs special proto

DESCRIPTION

Mkfs constructs a file system by writing on the special file *special* according to the directions found in the prototype file *proto*. The prototype file contains tokens separated by spaces or new lines. The first token is the name of a file to be copied onto block zero as the bootstrap program. See *diskboot(VIII)*. The second token is a number specifying the size of the created file system. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the i-list size in blocks (remember there are 16 i-nodes per block). The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters **−bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or − to specify set-user-id mode or not. The third is **g** or − for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions (see *chmod*(I)).

Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

If the file is a regular file, the next token is a pathname whence the contents and size are copied.

If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

If the file is a directory, *mkfs* makes the entries **.** and **..** and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token **$**.

If the prototype file cannot be opened and its name consists of a string of digits, *mkfs* builds a file system with a single empty directory on it. The size of the file system is the value of *proto* interpreted as a decimal number. The i-list size is the file system size divided by 43 plus the size divided by 1000. (This corresponds to an average size of three blocks per file for a 4000 block file system and six blocks per file at 40,000.) The boot program is left uninitialized.

A sample prototype specification follows:

```
/usr/mdec/uboot
4872 55
d−−777 3 1
usr     d−−777 3 1
        sh      −−−755 3 1 /bin/sh
        ken     d−−755 6 1
                $
        b0      b−−644 3 1 0 0
        c0      c−−644 3 1 0 0
        $
$
```

SEE ALSO

fs(V), directory(V), diskboot(VIII), icheck(VIII)

BUGS

It is not possible to initialize a file larger than 64K bytes.
The size of the file system is restricted to 64K blocks.
There should be some way to specify links.
After each *mkfs*, the super-block free counts must be initialized by *icheck(VIII)*.

NAME

mknod − build special file

SYNOPSIS

/etc/mknod name [**c**] [**b**] major minor

DESCRIPTION

Mknod makes a directory entry and corresponding i-node for a special file. The first argument is the *name* of the entry. The second is**b** if the special file is block-type (disks, tape) or **c** if it is character -type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number). These numbers are assumed to be decimal by default, octal if they begin with a zero.

The assignment of major device numbers is specific to each system. They have to be dug out of the system source file *conf.c*.

SEE ALSO

mknod(II)

NAME

mount − mount file system

SYNOPSIS

/etc/mount [special file [**−r**]]

DESCRIPTION

Mount announces to the system that a removable file system is present on the device corresponding to the special file *special* (which must refer to a disk or possibly DECtape). The *file* must exist already; its syntax must contain a leading "/", e.g., "/u8", or the mount request is disallowed and an appropriate error message is reported; otherwise, *file* becomes the name of the root of the newly mounted file system. A caution message is reported when mounting a file system with name *file*, which is different than the current root name of the device corresponding to the special file *special.*

Mount maintains a table of mounted devices (*mnttab*(V)); if invoked without an argument *mount* prints the entire table.

The optional last argument [**−r**] indicates that the *file* is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, regardless if an explicit write is attempted.

FILES

/etc/mnttab

SEE ALSO

fs(V), mnttab(V), umount(VIII)

NAME

ncheck − generate names from i-numbers

SYNOPSIS

/etc/ncheck [**−i** numbers] [**−a**] [filesystem]

DESCRIPTION

Ncheck with no argument generates a pathname vs. i-number list of all files on a set of default file systems. The **−i** flag reduces the report to only those files whose i-numbers follow. The **−a** flag allo ws printing of the names '.' and '..', which are ordinarily suppressed. A file system may be specified.

The full report is in no useful order, and probably should be sorted.

SEE ALSO

check(VIII), dcheck(VIII), icheck(VIII), sort(I)

NAME

patchup − patch up a damaged file system

SYNOPSIS

/etc/patchup

DESCRIPTION

Patchup is an interactive program that can repair certain forms of file system inconsistencies. These include a subset of the errors reported by *check*(VIII) that commonly occur after a crash. *P atchup* will correct deltas of 100, 177, 201, and 377, as follows:

100	the inode is de-allocated and any used blocks are added to the *free list.*
177	the link count for the file is incremented.
201	a pathname for the orphan file is created in a directory called ‘‘salvage’’ in the affected file system.
377	the directory entry for the inode is removed.

Patchup prompts the user for all the information that it requires.

FILES

/dev/rp??	file system to be patched
salvage	directory for orphan files
201dir	temporary directory for saving orphan files

SEE ALSO

check(VIII), fsdb(VIII), clrm(VIII), icheck(VIII), ncheck(VIII)
Repairing Damaged PWB/UNIX File Systems by P. D. Wandzilak.

DIAGNOSTICS

When it gets confused, *patchup* will print a message notifying the operator to obtain additional assistance from qualified personnel.

NAME

regen − regenerate system directories

DESCRIPTION

There are several system directories that contain various commands and archive libraries. These include */bin*, */usr/bin*, */lib*, */usr/lib*, */etc*, */usr/hasp*, and */usr/fort*. To facilitate the regeneration of these directories, the *make*(I) command is used. Each directory contains the *make* description files used to build, update, or regenerate the directory. If one description file suffices, it is named ***.****makefile;* otherwise the files are named ***.****makefile1*, ***.****makefile2*, ...

For best results, you should be logged in as ‘‘root’’ (or as super-user).

For example, the following will recompile any */bin* commands that are out-of-date with respect to their corresponding source files:

```
chdir /bin
make −f .makefile1
make −f .makefile2
make −f .makefile3
```

As another example, the following forces the recompilation of any */bin* commands that call *ctime*(III), in addition to recompiling anything that is out-of-date in */bin:*

```
chdir /bin
make −f .makefile1 CTIME=RC
make −f .makefile2 CTIME=RC
make −f .makefile3 CTIME=RC
```

The section on ‘‘Time Zones’’ below explains why this is useful. Similar arguments force recompilation of anything that uses the C compiler (CCDEP=RC), the assembler (ASDEP=RC), the *yacc*(I) processor (YACCDEP=RC), the ‘‘−lS’’ library (LSDEP=RC), the ‘‘−lPW’’ library (LPWDEP=RC), the ‘‘−lpw’’ library (LpwDEP=RC), and the UNIX operating system ‘‘include’’ files (SYSDEP=RC).

Time Zones:

Commands such as *ls*(I) and *date*(I) give Eastern times (EST or EDT). This has been known to annoy non-East Coast (USA) and non-USA users. To get your local time, you must change *ctime*(III), which converts the GMT *time*(II) time-stamps to local time, and then regenerate all commands that use *ctime*. To be precise, you must:

— Change *ctime*(III). The source currently lives in */sys/source/s4/ctime.c*, and the object resides in the standard C library, */lib/libc.a*.

— Regenerate the standard C library, as follows:

```
chdir /lib
make −f .makefile1
make −f .makefile2
make −f .makefile3
make −f .makefile4
```

This works because, as a result of the change to the source for *ctime*(III), */lib/libc.a* is now out-of-date with respect to its source, namely the *ctime* part.

— Regenerate all directories, using the CTIME=RC option on the *make* command line:

```
chdir /bin
make −f .makefile1 CTIME=RC
make −f .makefile2 CTIME=RC
make −f .makefile3 CTIME=RC
```

```
chdir /usr/bin
make −f .makefile1 CTIME=RC
make −f .makefile2 CTIME=RC
make −f .makefile3 CTIME=RC
make −f .makefile4 CTIME=RC
chdir /lib
make −f .makefile1 CTIME=RC
make −f .makefile2 CTIME=RC
make −f .makefile3 CTIME=RC
make −f .makefile4 CTIME=RC
chdir /usr/lib
make −f .makefile1 CTIME=RC
make −f .makefile2 CTIME=RC
chdir /etc
make −f .makefile CTIME=RC
chdir /usr/hasp
make −f .makefile CTIME=RC
chdir /usr/fort
make −f .makefile CTIME=RC
```

The full directory list (as of this moment!) is given above for completeness. Some directories might not have anything that uses *ctime.*

Of course, you should test your new *ctime* before installing it in the standard C library. See also the ‘‘BUGS’’ section below.

/bin:

Regeneration of */bin* is straightforward, except for *as*(I), *ld*(I), and *cc*(I). The*mak e* description files for */bin* do *not* automatically regenerate *cc;* instead, a separate procedure is used (see ‘‘cc’’ below).

As and *ld* are necessary to regenerate themselves. That is, one cannot re-assemble*as* unless a w orking *as* already exists. Thus, if something goes wrong and a bad*as* or *ld* is generated, you’ ll have to retrieve a working version from a backup tape.

The moral: before regenerating */bin,* be sure that you either have a good backup tape, or else have emergency copies of *as* and *ld* somewhere.

/usr/bin:

.makefile1 and **.**makefile2 regenerate the basic */usr/bin* commands; **.**makefile3 regenerates the SCCS commands; **.**makefile4 regenerates *eqn*(I) and *troff*(I). If you do not have the PWB/UNIX *troff*(I) and *eqn*(I) package, **.**makefile4 will fail.

/lib:

As with *as* and *ld* in */bin*, *as2* in */lib* (‘‘pass 2’’ of *as*) can’t be regenerated if it doesn’t already exist. Also,*c0, c1, c2,* and *cpp* in */lib* are parts of the C compiler, and the *make* description files for */lib* do not automatically regenerate them (see ‘‘cc’’ below).

/usr/lib:

.makefile1 regenerates the basic */usr/lib* modules; **.**makefile2 regenerates tables used by *eqn*(I) and *troff*(I). If you do not have the PWB/UNIX *troff*(I) and *eqn*(I) package, **.**makefile2 will fail.

cc:

/bin/cc is just the tip of the C compiler iceberg. The real work is done by *c0, c1, c2,* and *cpp* in */lib.* Since these programs work together, they must be updated carefully. Furthermore, since the C compiler is written in C, if you install a bad C compiler you’ll have to retrieve a working version from a backup tape.

Therefore, the *make* procedures for */bin* and */lib* do not attempt to regenerate the C compiler. Instead, you must do it explicitly, by:

```
chdir /sys/c/c
make −f makefile install clean
```

That creates *cc, c0, c1, c2,* and *cpp* in */sys/c/c,* and, if no errors occur, moves them to */bin* and */lib.* ''Clean'' just removes any object files (**.***o*) left around. Note that the *make* description file for the C compiler is *makefile* (no leading '**.**'). Only the*mak e* description files that live in object directories (such as */bin*) start with '**.**'.

Often it is better to install a new version of the C compiler as *ncc,* so that it can be tested without destroying the old (working) version. The following creates and installs a complete new version of the C compiler as *ncc:*

```
chdir /sys/c/c
make −f makefile PREF=n install clean
```

As before, this creates *cc, c0, c1, c2,* and *cpp* in */sys/c/c.* But they are installed as *ncc, nc0, nc1, nc2,* and *ncpp* in */bin* and */lib.* This works because the *cc* driver looks at the name under which it was invoked. If the first character of the name is 'c', it calls *c0,* etc., in */lib.* But if the first character isn't 'c', it prepends that character to the names of the load modules that it calls. Thus *ncc* calls *nc0, nc1, nc2,* and *ncpp* in */lib.*

/usr/hasp:

The **.***makefile* for */usr/hasp* specifies whether your RJE system supports one or two lines. The following generates a two-line system:

```
chdir /usr/hasp
make −f .makefile NHASP=2
```

while:

```
chdir /usr/hasp
make −f .makefile NHASP=
```

generates a one-line system (the default is one-line). To change the default, change the ''NHASP='' line near the beginning of the **.***makefile.*

SEE ALSO

make(I)
Make − A Program for Maintaining Computer Programs by S. I. Feldman

DIAGNOSTICS

''Don´t know how to make xxx'': The source file xxx doesn't exist.

FILES

.makefile*

BUGS

Not every directory has a **.**makefile.

Nroff(I) and *troff*(I) are fixed to Eastern Standard Time; they use their own conversion routines, instead of calling *ctime*(III). Fortunately, they use just the date, not the time-of-day.

NAME

restor − incremental file system restore

SYNOPSIS

/etc/restor key [arguments]

DESCRIPTION

Restor is used to read magtapes dumped with the *dump* command. The *key* argument specifies what is to be done. *Key* is a character from the set **trxw.**

t The date that the tape was made and the date that was specified in the *dump* command are printed. A list of all of the i-numbers on the tape is also given.

r The tape is read and loaded into the file system specified in *arguments.* This should not be done lightly (see below).

x Each file on the tape is individually extracted into a file whose name is the file's i-number. If there are *arguments,* they are interpreted as i-numbers and only they are extracted.

c If the tape overflows, increment the last character of its name and continue on that drive. (Normally it asks you to change tapes.)

f Read the dump from the next argument file instead of the tape.

i All read and checksum errors are reported, but will not cause termination.

w In conjunction with the **x** option, before each file is extracted, its i-number is typed out. To extract this file, you must respond with **y.**

The **x** option is used to retrieve individual files. If the i-number of the desired file is not known, it can be discovered by following the file system directory search algorithm. First retrieve the *root* directory whose i-number is 1. List this file with *ls −fi 1*. This will give names and i-numbers of sub-directories. Iterating, any file may be retrieved.

The **r** option should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape onto this. Thus

```
/etc/mkfs /dev/rp0 40600
restor r /dev/rp0
```

is a typical sequence to restore a complete dump. Another *restor* can be done to get an incremental dump in on top of this.

A *dump* followed by a *mkfs* and a *restor* is used to change the size of a file system.

FILES

/dev/mt0

SEE ALSO

ls(I), dump(VIII), mkfs(VIII)

DIAGNOSTICS

There are various diagnostics involved with reading the tape and writing the disk. There are also diagnostics if the i-list or the free list of the file system is not large enough to hold the dump.

If the dump extends over more than one tape, it may ask you to change tapes. Reply with a new-line when the next tape has been mounted.

BUGS

There is redundant information on the tape that could be used in case of tape reading problems. Unfortunately, *restor's* approach is to exit if anything is wrong.

NAME

rmall − remove all

SYNOPSIS

/etc/rmall [**−f**] [**−r**] [**−d**] name **...**

DESCRIPTION

Rmall is similar to *rm(I)* except that it never finds the argument list too long. It searches directories without using *glob(VIII)* and removes even files whose names begin with ‘‘.’’, unlike *rm.*

The **−f** (override protection mode) and **−r** (recurse to sub-directories) are as in *rm(I).* The **−d** option causes directories (that are empty) to be removed as well.

Rmall is handy, but so devastating that it is restricted to the super-user.

SEE ALSO

rm(I)

NAME

romboot − special ROM bootstrap loaders

DESCRIPTION

To bootstrap programs from various storage media, standard DEC® ROM bootstrap loaders are often used. However, such standard loaders may not be compatible with UNIX bootstrap programs or may not exist on a particular system. Thus, special bootstrap loaders were designed which may be cut into a programmable ROM (M792 read-only-memory) or manually toggled into memory.

Each program is position-independent, that is, it may be located anywhere in memory. Normally, it is loaded into high core to avoid being overwritten. Each reads one block from drive 0 into memory starting at address 0 and then jumps to address 0. To minimize the size, each assumes that a system INIT was generated prior to execution. Also, the address of one of the device registers is used to set the byte count register or word count register. In each case, this will read in at least 256 words, which is the maximum size of bootstrap programs.

On disk devices, block 0 is read, on tape devices, one block from the current position. Thus, the tape should be positioned at the load point (endzone if DECtape) prior to booting. Also, the standard DEC bootstrap loader for magnetic tape may be emulated by positioning the tape at the load point and executing the bootstrap loader twice.

By convention, on PWB 11/45 systems, address 773 000 is the start of a tape bootstrap loader, and 773 020 the start of a disk bootstrap loader. The actual loaders used depend on the particular hardware configuration.

SEE ALSO

70boot(VIII), unixboot(VIII)

TC11 − DECtape

```
012700          mov     $tcba,r0
177346
010040          mov     r0,−(r0)        /use tc addr for wc
012740          mov     $3,−(r0)        /read bn forward
000003
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
112710          movb    $5,(r0)         /read forward
000005
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

TU10 − Magnetic Tape

```
012700          mov     $mtcma,r0
172526
010040          mov     r0,−(r0)        /use mt addr for bc
012740          mov     $60003,−(r0)    /read, 800 bpi, 9 track
060030
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

TU16 − Magnetic Tape

```
012700          mov     $mtwc,r0
172442
012760          mov     $1300,30(r0)    /set 800 bpi, PDP format
001300
000030
010010          mov     r0,(r0)         /use mt addr for wc
012740          mov     $71,−(r0)       /read
000071
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

RK05 − DEC pack

```
012700          mov     $rkda,r0
177412
005040          clr     −(r0)
010040          mov     r0,−(r0)        /use rk addr for wc
012740          mov     $5,−(r0)        /read
000005
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

RP03 − Disk Pack

```
012700          mov     $rpmr,r0
176726
005040          clr     −(r0)
005040          clr     −(r0)
005040          clr     −(r0)
010040          mov     r0,−(r0)        /use rp addr for wc
012740          mov     $5,−(r0)        /read
000005
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

RP04 − Disk Pack

```
012700          mov     $rpcs1,r0
176700
012720          mov     $21,(r0)+       /read−in preset
000021
012760          mov     $10000,30(r0)   /set to 16−bits/word
010000
000030
010010          mov     r0,(r0)         /use rp addr for wc
012740          mov     $71,−(r0)       /read
000071
105710  1:      tstb    (r0)            /wait for ready
002376          bge     1b
005007          clr     pc              /transfer to zero
```

NAME

sa − Shell accounting

SYNOPSIS

/etc/sa [**−abcjlnrstuv**] [file]

DESCRIPTION

When a user logs in, if the Shell is able to open the file */etc/sha*, then as each command completes the Shell writes at the end of this file the name of the command, the user, system and real time consumed, and the user ID. *Sa* reports on, cleans up, and generally maintains this and other accounting files. To turn accounting on and off, the accounting file must be created or destroyed externally.

Sa is able to condense the information in */etc/sha* into a summary file */usr/adm/sht* which contains a count of the number of times each command was called and the time resources consumed. This condensation is desirable because on a large system *sha* can grow by 100 blocks per day. The summary file is read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; *sha* is the default. There are zillions of options:

a Place all command names containing unprintable characters and those used only once under the name ‘‘***other.’’

b Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.

c Besides total user, system, and real time for each command print percentage of total time over all commands.

j Instead of total minutes time for each category, give seconds per call.

l Separate system and user time; normally they are combined.

n Sort by number of calls.

r Reverse order of sort.

s Merge accounting file into summary file */usr/adm/sht* when done.

t For each command, report ratio of real time to the sum of user and system times.

u Superseding all other flags, print for each command in the accounting file the day of the year, time, day of the week, user ID and command name.

v If the next character is a digit *n*, then type the name of each command used *n* times or fewer. Await a reply from the terminal; if it begins with ‘‘y’’, add the command to the category ‘‘**junk**.’’ This is used to strip out garbage.

FILES

/etc/sha accounting
/usr/adm/sht summary

SEE ALSO

sha(V), ac(VIII)

NAME

setmnt − establish mnttab table

SYNOPSIS

/etc/setmnt

DESCRIPTION

Setmnt creates the **/etc/mnttab** table (see *mnttab*(V)), which is needed for both the *mount*(VIII) and *umount*(VIII) commands. *Setmnt* reads standard input, and creates a *mnttab* entry for each line. Input lines have the format:

filesys node

where *filesys* is the name of the file system's *special file* (e.g., ''rp??''), and *node* is the root name of that file system. Thus*filesys* and *node* become the first tw o strings in the *mnttab*(V) entry.

FILES

/etc/mnttab

SEE ALSO

mnttab(V)

BUGS

Evil things will happen if *filesys* or *node* are longer than 10 characters.

Setmnt silently enforces an upper limit on the maximum number of *mnttab* entries.

NAME

setuid − set user id of command

SYNOPSIS

/etc/setuid username [login-directory]
command . . .

DESCRIPTION

Setuid sets the real and effective user ID of the command that follows to it's argument. It also changes the login *name* and, optionally, the login *directory.* The line following the setuid command is executed by the shell. The user ID then reverts to 'root' (the only one who can use this command, after all).

NAME

shutdown − terminate all processing

SYNOPSIS

/etc/shutdown

DESCRIPTION

Shutdown is part of the PWB/UNIX operation procedures. Its primary function is to terminate all currently running processes in an orderly and cautious manner. The procedure is designed to interact with the user (i.e., the person who invoked *shutdown*). *Shutdown* may instruct the user to perform some specific tasks, or to supply certain responses before execution can resume. *Shutdown* goes through the following steps:

- All users logged on the system are notified to log off the system by a broadcasted message. The user may display his/her own message at this time. Otherwise, the standard file save message is displayed, which resides in */etc/getoff*.
- If the user wishes to run the file-save procedure, *shutdown* unmounts all file systems.
- All file systems' super blocks are updated before the system is to be stopped (see *sync*(I)). This must be done before re-booting the system, to insure file system integrity.

DIAGNOSTICS

The most common error diagnostic that will occur is *device busy.* This diagnostic happens when a particular file system could not be unmounted. See *umount*(VIII).

FILES

/etc/getoff

SEE ALSO

umount(VIII)

NAME

tapeboot − magnetic tape bootstrap programs

DESCRIPTION

There are two magnetic tape bootstrap programs to handle the problem of booting a PDP-11/45 or PDP-11/70 from a TU10 or TU16 tape transport. In both programs, the tape density used is 800 bpi.

The first bootstrap program, *mboot*, is designed to be used in conjunction with the *tp* command. The *tp* command first copies a bootstrap program /usr/mdec/mboot, if magnetic tape, or /usr/mdec/tboot, if DECtape, to the first record (block 0, 256 words) of the tape. Note, the bootstrap mboot or tboot, as appropriate, must exist whenever a tape is made using *tp*. When using DECtape, booting is relatively simple since the standard DEC® ROM bootstrap loader reads block 0. However, when using magnetic tape, a special ROM or some manual procedure must be used to read in block 0. The standard DEC loaders for magnetic tape read the second record (block 1) of the tape, since the first record (block 0) contains header information on DEC operating systems.

To boot from magnetic tape using mboot, first create a tape using *tp*. For example, to make a boot tape containing the UNIX operating system, execute

tp mr unix

Next, read the first record of the tape into memory starting at address 0 and then jump to address 0, using a special ROM or some manual procedure (toggle in the program). The bootstrap program, mboot, relocates itself to high core as specified when assembled (typically 24K words, maximum of 28K). It then determines whether to use the TU10 code or the TU16 code. The TU10 is used if the TM11 command register (772 552) exists and the function (bits <3:1>) is non-zero, otherwise the TU16 is used. Next, the program rewinds the tape and prompts the user by typing '=' on the console. The user must then enter the name of the desired program exactly as contained in the table of contents on the tp tape. The '#' character will erase the last character typed, the '@' character will kill the entire line, and 'A' through 'Z' is translated to 'a' through 'z'. Also, carriage return (CR) is mapped into line feed (LF) on input, and LF is output as CR-LF. The upper-case to lower-case conversion is used to handle upper-case-only terminals such as the TELETYPE® Model 33 or the DEC LA30. Therefore, a file name with upper case characters can not be booted using this procedure.

After the name has been completely entered by typing CR or LF, the program reads the table of contents into memory starting at address 0. This table of contents is 24 blocks long for DECtape and 62 blocks for magnetic tape. If the name matches an entry in the table, memory below the bootstrap program is cleared, the tape is positioned to the address found in the table entry, and read into memory starting at address 0. Note, the amount of data read is only the low order 16 bits of the 24 bit size field in the table of contents entry. Next, the bootstrap program rewinds the tape. If address 0 contains 000 407, a UNIX a.out program is assumed and the first 8 words are stripped off by relocating the loaded program toward address 0. Finally, a jump to address 0 is done by executing "jsr pc,*$0".

If the name entered does not match or if the loaded program returns, the bootstrap starts over and prompts '='. Also, if there is an error while reading the tape, the program will backspace one record and attempt to read the record again. This may lead to a looping condition if there is a hard tape error.

The other bootstrap program, *tapeboot*, does not involve the *tp* command. The complete program fits in one 256 word block, but is duplicated so that one copy resides in block 0 and another in block 1. Thus, both the standard DEC ROM bootstrap loaders and the special ROM loaders will work. For example, to create a boot tape, execute

cat /usr/mdec/tapeboot program-to-boot > /dev/mt0

After the program is loaded and started, it relocates itself to high core and determines whether to use the TU10 code or the TU16 code as described above. It then types on the console "PWB tape boot loader", rewinds the tape, reads two blocks to skip past itself on the tape, clears memory, and reads the rest of the tape, to the tape mark, into memory starting at address 0. The loaded program is adjusted if address 0 contains 000 407 as above. Finally, a jump to address 0 is done by executing "jsr pc, *$0".

If there is an error while reading the tape, the bootstrap program will type "tape error" and attempt to read the record again.

FILES

/usr/mdec/mboot - *tp* magtape bootstrap
/usr/mdec/tboot - *tp* DECtape bootstrap
/usr/mdec/tapeboot - TU10/TU16 magtape bootstrap
/sys/source/util - source directory

SEE ALSO

tp(I), unixboot(VIII)

NAME

umount − dismount file system

SYNOPSIS

/etc/umount special

DESCRIPTION

Umount announces to the system that the removable file system previously mounted on special file *special* is to be removed.

SEE ALSO

mount(VIII), mnttab(V)

FILES

/etc/mnttab mounted device table

DIAGNOSTICS

It complains if the special file is not mounted or if it is busy. The file system is busy if there is an open file on it or if someone has his/her current directory there.

NAME

unixboot − UNIX startup and boot procedures

DESCRIPTION

How to start UNIX. UNIX is started by placing it in core at location zero and transferring to zero. Since the system is not reenterable, it is necessary to read it in from disk or tape. See *diskboot(VIII)* or *tapeboot(VIII).*

The switches. The console switches play an important role in the use and especially the booting of UNIX. During operation, the console switches are examined 60 times per second, and the contents of the address specified by the switches are displayed in the display register. If the switch address is even, the address is interpreted in kernel (system) space; if odd, the rounded-down address is interpreted in the current user space.

If any diagnostics are produced by the system, they are printed on the console only if the switches are non-zero. Thus it is wise to have a non-zero value in the switches at all times.

During the startup of the system, *init(VIII)* reads the switches and will come up single-user if the switches are set to 173030.

FILES

/unix − UNIX code

SEE ALSO

tp(I), init(VIII), 70boot(VIII), diskboot(VIII), romboot(VIII), tapeboot(VIII)

NAME

volcopy, labelit − copy filesystems with label checking

SYNOPSIS

/etc/vc10 fsname special1 volname1 special2 volname2
/etc/vc50 fsname special1 volname1 special2 volname2
/etc/vc88 fsname special1 volname1 special2 volname2
/etc/labelit special [fsname volume [−n]]

DESCRIPTION

Volcopy makes a literal copy of the filesystem using a blocksize matched to the device (10 blocks for tape *[vc10]*, 50 blocks for RP03 *[vc50]*, or 88 blocks for RP04,5,6 *[vc88])*. Using *vc10*, a 2400 foot/800 bpi tape will hold 40K blocks; 65K blocks fit at 1600 bpi.

The *fsname* argument represents the mounted name (e.g.: ‘root’, ‘u1’, etc.) of the filsystem being copied.

The *special* should be the physical disk or tape (e.g.: ‘/dev/rrp15’, ‘/dev/rmt0’, etc.).

The *volname* is the physical volume name (e.g.: ‘pk3’, ‘t0122’, etc.) and should match the external label sticker. Such label names are limited to five or fewer characters.

Special1 and *volname1* are the device and volume from which the copy of the filesystem is being extracted. *Special2* and *volname2* are the target device and volume.

Fsname and *volname* are recorded in the last 12 characters of the superblock (char fsname[6], volname[6];).

Labelit can be used to provide initial labels for unmounted disk or tape filesystems. With the optional arguments omitted, *labelit* prints current label values. The **−n** option pro vides for initial labelling of new tapes or disks (this destroys previous contents).

FILES

/etc/log/filesavelog: a record of filesystems/volumes copied

SEE ALSO

fs(V)

NAME

wall − write to all users

SYNOPSIS

/etc/wall

DESCRIPTION

Wall reads its standard input until an end-of-file. It then sends this message to all currently logged in users preceded by ‘‘Broadcast Message ...’’. It is used to warn all users, typically prior to shutting down the system.

The super-user can send to all people logged in, regardless of protections; if used immoderately, this is a quick way to become unpopular.

FILES

/dev/tty?

SEE ALSO

mesg(I), write(I)

DIAGNOSTICS

‘‘Cannot send to ...’’ when the open on a user’s tty file fails.