

NAME

boot procedures — UNIX/32V startup

DESCRIPTION

How to start UNIX/32V. UNIX/32V is started by placing it in memory at location zero and transferring to zero. Since the system is not reenterable, it is necessary to read it in from disk or tape. The LSI-11 console computer is used to do this.

The LSI-11 must first be in the right frame of mind. When it is, it will type three greater-than signs '>>>' as a prompt. If you do not see this prompt, type control-P and "HALT\r" to gain the attention of the LSI-11. If the last printout on the console is a six-digit octal number and an at-sign '@', then the LSI-11 has crashed and is in ODT mode. Type "173000G" to boot the LSI-11. If this fails, open the VAX-11 cabinet and flick the HALT/ENABLE switch on the LSI-11 first to HALT, then to ENABLE.

To boot from disk: type the standard DEC console command "BOOT\r" when the console is in LSI-11 mode. This causes the LSI-11 to read block 0, unit 0, MBA 0 into memory at location 0 and start the VAX-11 at location 0 when the read has finished. In a standard file system, block 0 contains the program *uboot*. *uboot* executes the file */fboot* which it finds in the file system which resides at the beginning of unit 0, MBA 0. *fboot* prints a prompt, reads a file name (usually 'unix') from the console, finds that file, reads it into memory at location 0, and transfers to it with **calls \$0,*\$0**.

If the console BOOT command does not work, here is a position-independent program which will read the boot block from the disk.

RP06 disk (MBA 0, drive 0)

```

00009fde  movl    *$0x20010000,r1      # MBA 0
512001
d0        movl    $1,4(r1)      # initialize MBA
04a101
d0        movl    $0x13,0x400(r1) # volume valid
0400c113
10008f32  cvtwtl  $0x1000,0x424(r1)    # 16-bit format
0424c1
d4        clrl    12(r1)        # map reg. 0, offset 0
0ca1
8fd0      movl    $0x80000000,0x800(r1) # valid, page 0/0
80000000
0800c1
32        cvtwtl  $-512,16(r1)   # byte count
alfe008f
10
28c1d4    clrl    0x428(r1)      # cylinder
04
14c1d4    clrl    0x414(r1)      # sector, track
04
c139d0    movl    $0x39,0x400(r1) # read 1 block
0400
00        halt

```

To boot from magnetic tape: The *tp(1)* command places the bootstrap program *mboot* on block 0 of the tape. When read into memory at location 0 and executed, *mboot* prompts with an equal sign '=', reads a file name from the console, then loads and executes that file from the *tp* tape. Unfortunately DEC does not provide a console command to read a block from tape to memory. Here is a position-independent program which does:

TM03 magtape (MBA 1, drive 0) from load point:

```

20009fde  moval  *$0x20012000,r1      # MBA 1
512001
d0         movl   $1,4(r1)          # initialize
04a101
32         cvtwl  $0x13c0,0x424(r1) # drive characteristics
c113c08f
0424
ald4       clrl   12(r1)            # map reg. 0, offset 0
0c
008fd0     movl   $0x80000000,0x800(r1) # valid, page 0/0
c1800000
0800
8f32       cvtwl  $-512,16(r1)      # byte count
10a1fe00
00c139d0   movl   $0x39,0x400(r1)    # read 1 block
04
00         halt

```

Then give the console command “START 0\r”.

mboot and *fboot* do not perform character erase and line kill editing. Instead, they start over with the prompt for file name whenever the requested file cannot be found.

Be sure that *mboot* exists whenever a *tp* tape is made. Remember to put *uboot* and *fboot* in file systems when running *mkfs*.

FILES

/unix	UNIX/32V code
/usr/mdec/uboot	disk bootstrap
/fboot	file system bootstrap
/usr/mdec/mboot	<i>tp</i> magtape bootstrap

SEE ALSO

tp(1), *init*(8)

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.

What to do first. (Someday the LSI-11 will do this automatically.) Write the date and time on the console log. Use the console commands to examine the registers, program status long word, and the top several locations on the stack. A suggested command sequence is

```
E PSL<return>
E R0/NE:F<return>
E SP<return>
E/V @ /NE:20<return>
```

How to bring it back up. If the reason for the crash is not evident (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 magnetic tape. With a tape mounted and ready and the cpu halted, insure that the cpu is in kernel mode, deposit 80000200 into the pc, and continue. The sequence of console commands is

```
D PSL 0
D PC 80000200<return>
C<return>
```

This should write a copy of all of memory on the tape, followed by two EOF marks. Caution: Any error is taken to mean the end of memory has been reached. This means that you must be sure the ring is in, the tape is ready, and the tape is clean and new. If the dump fails, you can try again, but some of the registers will be lost. See below for what to do with the tape.

To restart after a crash, follow the directions in *bproc*(8). The system will come up in single-user mode. When it is running, perform a *dcheck* and *icheck*(1) 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, log out by typing an EOT (<control-D>). The command sequence in */etc/rc* will be executed and the system will be in multi-user mode.

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, then the cpu will loop at location 0x13. For *init* to work correctly, */dev/console* and */bin/sh* must be present. If either does not exist, the symptom is best described as thrashing. *Init* will go into a *fork/exec* 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 keep 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. This is an area where experience and informed courage count for much.

The problems reported by *icheck* 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*. If the same block appears in more than one file or if a file contains bad blocks, the files should be deleted, and the free list reconstructed. The best way to delete such a file is to use *clri*(1), then remove its directory entries with *rm*(1). (Do not use *mv*(1).) If any of the affected files is really precious, you can try to copy it to another device first.

Dcheck may report files which have more directory entries than links. Such situations are potentially dangerous; *clri* discusses a special case of the problem. All the directory entries for the file should be removed. If on the other hand there are more links than directory entries, there is no danger of spreading infection, but merely some disk space that is lost for use. It is sufficient to copy the file (if it has any entries and is useful) then use *clri* on its inode and remove any directory entries that do exist.

Finally, there may be inodes reported by *dcheck* 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 *clri* will free the inode, and an *ichack -s* will recover any missing blocks.

Why did it crash? UNIX types a message on the console typewriter 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: ...', 'TRAP FROM KERNEL MODE', or 'ILLEGAL I/E VECTOR' (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.

iinit

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

out of inodes

A mounted file system has no more i-nodes when creating a file. Sorry, the device isn't available; the *ichack* should tell you.

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 inodes

The in-memory inode table is full. Try increasing NINODE in param.h. Shouldn't be a panic, just a user error.

no clock

During initialization, neither the line nor programmable clock was found to exist.

swap error

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

unlink — iget

The directory containing a file being deleted can't be found. Hardware or software.

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.

out of text

A pure procedure program is being executed, and the table for such things is full. This shouldn't be a panic.

TRAP FROM KERNEL MODE

An unexpected trap has occurred within the system. The trap type can be determined by examining the top word of the stack (the trap type) with the console commands. The trap types are:

0	reserved addressing mode
1	privileged instruction
2	BPT
3	XFC
4	reserved operand
5	CHMK (system call)
6	arithmetic trap
7	reschedule trap (software level 3)
8	segmentation fault
9	protection fault
10	trace pending (TP bit)

ILLEGAL I/E VECTOR, HALTED AT xx

an illegal interrupt or exception has occurred. The possible addresses are

4	machine check (hardware error).
8	kernel stack not valid
C	power failure

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 to examine memory; 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 */usr/sys/core*; *cp(1)* 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. Use *adb(1)* to examine */usr/sys/core*. The location *dumpstack-80000000* is the bottom of a stack onto which were pushed the stack pointer *sp*, *PCBB* (containing the physical address of a *u_area*), *MAPEN*, *IPL*, and registers *r13-r0* (in that order). *r13(fp)* is the system frame pointer and the stack is used in standard *calls* format. Use *adb(1)* to get a reverse calling order. In most cases this procedure will give an idea of what is wrong. A more complete discussion of system debugging is impossible here.

SEE ALSO

clri(1), *icheck(1)*, *dcheck(1)*, *bproc(8)*

BUGS

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*(8).

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 minute.

FILES

`/usr/lib/crontab`

NAME

fget.demon, *fget.odemon* — fget daemons

SYNOPSIS

/etc/fget.demon time

/etc/fget.odemon time

DESCRIPTION

Fget.demon and *fget.odemon* are daemons for the 210 dataphone or spider. They are designed to retrieve files that have been requested by *fget*(1) from the Honeywell 6070 computer. The argument *time* is the number of seconds for *fget.demon* to wait for files to appear from GRTS. The default is 6 minutes. *Fget.demon* is automatically initiated by *fget*, and by *cron* (1).

On systems with both a spider and a dataphone connection to the Honeywell 6070 computer, *fget.demon* uses spider, and *fget.odemon* uses the dataphone, and is called automatically as a backup when the spider connection is down. On other systems, there is only one fget daemon, *fget.demon*, which use the dataphone.

The daemons use the spooling directory */usr/dpd*. The file *glock* in that directory is used to prevent two daemons from becoming active. After the program has successfully set the lock, it forks and the main path exits, thus spawning the daemon. GRTS is interrogated for any output for the daemon's station-id. If none, *fget.demon* will wait up to *time* seconds, interrogating GRTS every minute or so to see if any output has arrived. All problems and successful transactions are recorded in the *errors* file in the spooling directory.

To restart *fget.demon* (in the case of hardware or software malfunction), it is necessary to first kill the old *fget.demon* (if still alive), and remove the lock file before initiating *fget.demon*. This is done automatically when the system is brought up, by *latchrc* (see *init*(1)) in case there are any jobs waiting to come over.

FILES

*/usr/dpd/** spool area

*/dev/du** dataphone device

*/dev/dn** acu device

/dev/tiu/d2 spider device

SEE ALSO

fget(1), *dpd*(1)

NAME

`getty` — set terminal mode

SYNOPSIS

`/etc/getty [ char ]`

DESCRIPTION

Getty is invoked by *init*(8) immediately after a terminal is opened, following the making of a connection. While reading the name *getty* 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 1200 baud and the 'login:' is typed again; a second 'break' changes the speed to 150 baud and the 'login:' is typed again. Successive 'break' characters cycle through the speeds 300, 1200, and 150 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*(2)).

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(8), *login*(1), *stty*(2), *ttys*(5)

BUGS

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 port on which a user may log in.

init first opens the console terminal */dev/console* for reading and writing, and then invokes a Shell. This feature brings up the system in single-user mode. The *getty* and *login* routines mentioned below and described elsewhere are not used yet.

When the single-user shell terminates (usually caused by typing an EOT on the console), *init* invokes another Shell, with input taken from the file */etc/rc*. This command file performs house-keeping operations such as 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 is dialed up and carrier established on the channel. Then */etc/getty* is called with argument as specified by the second character of the *ttys* file line. *Getty* reads the user's name and invokes *login* 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 SIGHUP) and interprets it to mean that the file */etc/ttys* should be 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/console, */dev/tty?*, */etc/utmp*, */usr/adm/wtmp*, */etc/ttys*, */etc/rc*

SEE ALSO

login(1), *kill*(1), *sh*(1), *ttys*(5), *getty*(8)

NAME

lpd — line printer daemon

SYNOPSIS

/etc/lpd

DESCRIPTION

Lpd is the daemon for the line printer. *Lpd* uses the directory */usr/lpd*. The file *lock* in that directory is used to prevent two daemons from becoming active. After the program has successfully set the lock, it forks and the main path exits, thus spawning the daemon. The directory is scanned for files beginning with **df**. Each such file is submitted as a job. Each line of a job file must begin with a key character to specify what to do with the remainder of the line.

L specifies that the remainder of the line is to be sent as a literal.

B specifies that the rest of the line is a file name.

F is the same as **B** except a form feed is prepended to the file.

U specifies that the rest of the line is a file name. After the job has been transmitted, the file is unlinked.

M is followed by a user ID; after the job is sent, a message is mailed to the user via the *mail(1)* command to verify the sending of the job.

Any error encountered will cause the daemon to wait 1 minute and start over. This means that an improperly constructed *df* file may cause the same job to be submitted every 20 minutes.

Lpd is automatically initiated by the line printer command, *lpr*.

To restart *lpd* (in the case of hardware or software malfunction), it is necessary to first kill the old daemon (if still alive), and remove the lock file before initiating the new daemon. This is done automatically when the system is brought up, by */etc/rc*, in case there were any jobs left in the spooling directory when the system last went down.

FILES

*/usr/lpd/** spool area for line printer daemon

/etc/passwd to get the user's name

/dev/lp line printer device *lpr(1)*

NAME

makekey — generate encryption key

SYNOPSIS

/usr/lib/makekey

DESCRIPTION

Makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (i.e. to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, upper- and lower-case letters, and '.' and '/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

Makekey is intended for programs that perform encryption (e.g. *ed* and *crypt(1)*). Usually its input and output will be pipes.

SEE ALSO

crypt(1), *ed(1)*

NAME

sticky — executable files with persistent text

DESCRIPTION

While the 'sticky bit', mode 01000 (see *chmod(2)*), is set on a sharable executable file, the text of that file will not be removed from the system swap area. Thus the file does not have to be fetched from the file system upon each execution. As long as a copy remains in the swap area, the original text cannot be overwritten in the file system, nor can the file be deleted. (Directory entries can be removed so long as one link remains.)

Sharable files are made by the **-n** and **-i** options of *ld(1)*.

To replace a sticky file that has been used do: (1) Clear the sticky bit with *chmod(1)*. (2) Execute the old program to flush the swapped copy. This can be done safely even if others are using it. (3) Overwrite the sticky file. If the file is being executed by any process, writing will be prevented; see the next paragraph for a dodge. (4) Set the sticky bit again. Only the super-user can set the sticky bit.

To replace a sharable executable file while any user is executing it do: (1) Rename the directory entry for that file with *mv(1)*. (2) Install the new program with *cp*, *mv(1)*, or otherwise. (3) With *chown* and *chmod(2)*, set the mode of the new file to agree with that of the original as necessary. (4) Delete the old file when convenient.

BUGS

Are self-evident.

NAME

sysgen — UNIX/32V system generation from the distribution tape

DESCRIPTION

This section explains how to generate an initial UNIX/32V system from the distribution tape. It assumes that you have some familiarity with the hardware: how to mount and unmount disk packs and magnetic tapes, and how to use the console.

PREREQUISITES

- 1 VAX-11/780 system with at least 256K bytes of memory
- 1 RP06 disk spindle (unit 0) with controller on MBA 0
- 1 TE16 magnetic tape drive (unit 0) with TM03 formatter on MBA 1
- 1 error-free disk pack for use on the RP06 spindle
- 1 UNIX/32V distribution tape
- 90 minutes of time

OVERVIEW

The distribution tape is recorded at 800 bpi and contains three files. The first file is in *tp*(5) format and contains stand-alone utility programs for handling disks, tapes, and the console. The first file usually contains about 250 blocks of 512 characters each. The second file on the distribution tape is a *dd*(1) copy of an initial root file system (see *filsys*(5)). On the tape the file system is 250 blocks of 10240 characters. The third and last file on the tape contains source code, and also object code and miscellaneous data, from the *usr* file system. The last file is in *tar*(1) format, blocked 10240 characters per tape record, and contains several thousand records. The system generation process has three phases, corresponding to the three files on the distribution tape. The first phase formats and verifies the disk pack. The second phase creates the initial root file system on the disk. The third phase establishes the *usr* directory.

PROCEDURE

Read and understand this entire description before attempting anything else. Quotation marks “” surround literal character strings which you should type on the console. The notation “\r” stands for the RETURN key.

Turn on the VAX-11/780 to LOCAL. The POWER and ATTN lights should go on; if not, fix your hardware before proceeding.

Mount an error-free disk pack on the RP06 spindle, unit 0, MBA 0. Start the drive and set read-write (not write protect). The START, CONTROL A, and DOOR LOCKED lights should be on. If the READY light on the drive does not come on within 30 seconds, then fix your hardware before proceeding.

Mount the UNIX/32V distribution tape on the tape drive, unit 0, MBA 1. All status lights (PWR, BOT, WRL, SELECT, LOAD, ONLINE) should be on. If the WRL light is not on, then remove the write ring from the reel and start again. If the other lights are not on, then fix your hardware before proceeding.

Enter the commands HALT, UNJAM, INITIALIZE on the console. These may usually be abbreviated to “H\r”, “U\r”, and “I\r”, respectively.

Deposit in memory at a high address the absolute tape boot program found in *bproc*(8). Execute the code you deposited, using the “START” console command. One 512-byte block (the *tp* boot program) will be read from the tape into VAX-11 memory beginning at location 0.

Give the console command “START 0\r”. This starts the *tp* boot program, which relocates itself high in memory, rewinds the tape, and then types an equal sign ‘=’ on the console as a prompt.

Type `"rp6fmt\r"`. The *tp* boot will load the RP06 disk formatting program into memory and transfer control to it.

Enter the MBA number and unit number (in this case, both `"0\r"`) in response to the prompts. The *rp6fmt* program will then format the disk; this should take approximately 20 minutes. Error messages may appear on the console if bad spots are found on the disk. There should not be any (you mounted an error-free pack, remember). If there are any error messages, sit tight and hope for the best.

When the *rp6fmt* program finishes, it will ask to format another pack. Type `"-1\r"` to exit. The equal sign prompt `"="` of the *tp* boot should appear again. Type `"rpread\r"`. This program verifies that the entire disk can be read. Enter the unit number and starting block number (both `"0\r"`) in response to the prompts. Enter `"\r"` when asked for the number of blocks; this will read the whole pack. Reading the disk takes 10 minutes. Messages indicating bad spots may appear on the console. The bad spots should match those found by the *rp6fmt* program. All ECC errors should be correctible. If there are any errors which are not correctible, you should probably start the whole boot procedure over with a better pack.

After the *rpread* program finishes, it will prompt for another read. Type `"-1\r"` to exit; another `"="` prompt from the *tp* boot will appear. Type `"tdcopy\r"`. In general, *tdcopy* copies tape-to-disk. It is used here to copy an initial root file system from the tape onto the disk beginning at cylinder 0, track 0, sector 0. Respond to the prompts from *tdcopy* with `"1\r"` (tape MBA #), `"0\r"` (tape unit #), `"1\r"` (tape file offset), `"0\r"` (tape block offset), `"0\r"` (disk MBA #), `"0\r"` (disk unit), `"0\r"` (disk block offset), and `"250\r"` (number of input blocks). The copy takes 5 minutes.

You are now ready to boot UNIX (yea!). The `"="` prompt from *tp* boot should appear again. Type `"rpboot\r"`. The *rpboot* program should then prompt with `"file:"`. Type `"unix\r"`. This loads the operating system from disk and executes it. The system will report the physical memory in your configuration and the memory not used by the operating system. Then it will type `"#"`. This is the super-user prompt for single-user operation of the UNIX system.

Set the date and time using the *date(1)* command.

Check the integrity of the root file system with

```
icheck /dev/rp0a
dcheck /dev/rp0a
```

Create the */usr* file system with

```
/etc/mkfs /dev/rp0g 300000
/etc/mount /dev/rp0g /usr
```

This takes 5 minutes.

Extract the remaining files from the distribution tape with

```
cd /usr
cp /dev/rmt4 /dev/null           ;; bypasses tp file
cp /dev/rmt4 /dev/null           ;; bypasses root
tar xbf 20 /dev/rmt0
```

This takes 20 minutes.

Check the integrity of the */usr* file system with

```
cd /
/etc/umount /dev/rp0g
icheck /dev/rp0g
```

Establish the desired entries in the password file */etc/passwd*, using the text editor *ed(1)*.

Set the terminal lines configuration in */etc/tty*s using the text editor *ed*(1).

Edit the reconfiguration file */etc/rc* as desired.

Type EOT (control-D) to start multi-user operation.

NAME

update — periodically update the super block

SYNOPSIS

/etc/update

DESCRIPTION

Update is a program that executes the *sync*(2) primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file.

SEE ALSO

sync(2), *sync*(1), *init*(8)

BUGS

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. A fix would be to have *sync*(1) temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.