-

NAME

a.out − assembler and link editor output

DESCRIPTION

A.out is the output file of the assembler *as* and the link editor *ld.* Both programs make *a.out* executable if there were no errors and no unresolved external references.

This file has four sections: a header, the program and data text, a symbol table, and relocation bits (in that order). The last two may be empty if the program was loaded with the ‘‘−s’’ option of *ld* or if the symbols and relocation have been removed by *strip.*

The header always contains 8 words:

1 A magic number (407, 410, or 411(8))
2 The size of the program text segment
3 The size of the initialized portion of the data segment
4 The size of the uninitialized (bss) portion of the data segment
5 The size of the symbol table
6 The entry location (always 0 at present)
7 Unused
8 A flag indicating relocation bits have been suppressed

The sizes of each segment are in bytes but are even. The size of the header is not included in any of the other sizes.

When a file produced by the assembler or loader is loaded into core for execution, three logical segments are set up: the text segment, the data segment (with uninitialized data, which starts off as all 0, following initialized), and a stack. The text segment begins at 0 in the core image; the header is not loaded. If the magic number (word 0) is 407, it indicates that the text segment is not to be write-protected and shared, so the data segment is immediately contiguous with the text segment. If the magic number is 410, the data segment begins at the first 0 mod 8K byte boundary following the text segment, and the text segment is not writable by the program; if other processes are executing the same file, they will share the text segment. If the magic number is 411, the text segment is again pure, write-protected, and shared, and moreover instruction and data space are separated; the text and data segment both begin at location 0. See the 11/45 handbook for restrictions which apply to this situation.

The stack will occupy the highest possible locations in the core image: from 177776(8) and growing downwards. The stack is automatically extended as required. The data segment is only extended as requested by the *break* system call.

The start of the text segment in the file is 20(8); the start of the data segment is $20+S_t$ (the size of the text) the start of the relocation information is $20+S_t+S_d$; the start of the symbol table is $20+2(S_t+S_d)$ if the relocation information is present, $20+S_t+S_d$ if not.

The symbol table consists of 6-word entries. The first four words contain the ASCII name of the symbol, null-padded. The next word is a flag indicating the type of symbol. The following values are possible:

00 undefined symbol
01 absolute symbol
02 text segment symbol
03 data segment symbol
37 file name symbol (produced by ld)
04 bss segment symbol
40 undefined external (.globl) symbol
41 absolute external symbol
42 text segment external symbol
43 data segment external symbol
44 bss segment external symbol

Values other than those given above may occur if the user has defined some of his own instructions.

The last word of a symbol table entry contains the value of the symbol.

If the symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader *ld* as the name of a common region whose size is indicated by the value of the symbol.

The value of a word in the text or data portions which is not a reference to an undefined external symbol is exactly that value which will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation bits for that word, then the value of the word as stored in the file is an offset from the associated external symbol. When the file is processed by the link editor and the external symbol becomes defined, the value of the symbol will be added into the word in the file.

If relocation information is present, it amounts to one word per word of program text or initialized data. There is no relocation information if the ''suppress relocation'' flag in the header is on.

Bits 3-1 of a relocation word indicate the segment referred to by the text or data word associated with the relocation word:

00 indicates the reference is absolute
02 indicates the reference is to the text segment
04 indicates the reference is to initialized data
06 indicates the reference is to bss (uninitialized data)
10 indicates the reference is to an undefined external symbol.

Bit 0 of the relocation word indicates if *on* that the reference is relative to the pc (e.g. ''clr x''); if *off,* that the reference is to the actual symbol (e.g., ''clr *$x'').

The remainder of the relocation word (bits 15-4) contains a symbol number in the case of external references, and is unused otherwise. The first symbol is numbered 0, the second 1, etc.

SEE ALSO

as (I), ld (I), strip (I), nm (I)

-

NAME

ar − archive (library) file format

DESCRIPTION

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld.*

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 177555(8) (it was chosen to be unlikely to occur anywhere else). The header of each file is 16 bytes long:

0-7	file name, null padded on the right
8-11	modification time of the file
12	user ID of file owner
13	file mode
14-15	file size

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size give reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

SEE ALSO

ar (I), ld (I)

BUGS

Names are only 8 characters, not 14. More important, there isn't enough room to store the proper mode, so *ar* always extracts in mode 666.

NAME

ascii − map of ASCII character set

SYNOPSIS

cat /usr/pub/ascii

DESCRIPTION

Ascii is a map of the ASCII character set, to be printed as needed. It contains:

<table>
<tr><td>000 nul</td><td>001 soh</td><td>002 stx</td><td>003 etx</td><td>004 eot</td><td>005 enq</td><td>006 ack</td><td>007 bel</td></tr>
<tr><td>010 bs</td><td>011 ht</td><td>012 nl</td><td>013 vt</td><td>014 np</td><td>015 cr</td><td>016 so</td><td>017 si</td></tr>
<tr><td>020 dle</td><td>021 dc1</td><td>022 dc2</td><td>023 dc3</td><td>024 dc4</td><td>025 nak</td><td>026 syn</td><td>027 etb</td></tr>
<tr><td>030 can</td><td>031 em</td><td>032 sub</td><td>033 esc</td><td>034 fs</td><td>035 gs</td><td>036 rs</td><td>037 us</td></tr>
<tr><td>040 sp</td><td>041 !</td><td>042 "</td><td>043 #</td><td>044 $</td><td>045 %</td><td>046 &</td><td>047 ´</td></tr>
<tr><td>050 (</td><td>051)</td><td>052 *</td><td>053 +</td><td>054 ,</td><td>055 −</td><td>056 .</td><td>057 /</td></tr>
<tr><td>060 0</td><td>061 1</td><td>062 2</td><td>063 3</td><td>064 4</td><td>065 5</td><td>066 6</td><td>067 7</td></tr>
<tr><td>070 8</td><td>071 9</td><td>072 :</td><td>073 ;</td><td>074 <</td><td>075 =</td><td>076 ></td><td>077 ?</td></tr>
<tr><td>100 @</td><td>101 A</td><td>102 B</td><td>103 C</td><td>104 D</td><td>105 E</td><td>106 F</td><td>107 G</td></tr>
<tr><td>110 H</td><td>111 I</td><td>112 J</td><td>113 K</td><td>114 L</td><td>115 M</td><td>116 N</td><td>117 O</td></tr>
<tr><td>120 P</td><td>121 Q</td><td>122 R</td><td>123 S</td><td>124 T</td><td>125 U</td><td>126 V</td><td>127 W</td></tr>
<tr><td>130 X</td><td>131 Y</td><td>132 Z</td><td>133 [</td><td>134 \</td><td>135]</td><td>136 ^</td><td>137 _</td></tr>
<tr><td>140 `</td><td>141 a</td><td>142 b</td><td>143 c</td><td>144 d</td><td>145 e</td><td>146 f</td><td>147 g</td></tr>
<tr><td>150 h</td><td>151 i</td><td>152 j</td><td>153 k</td><td>154 l</td><td>155 m</td><td>156 n</td><td>157 o</td></tr>
<tr><td>160 p</td><td>161 q</td><td>162 r</td><td>163 s</td><td>164 t</td><td>165 u</td><td>166 v</td><td>167 w</td></tr>
<tr><td>170 x</td><td>171 y</td><td>172 z</td><td>173 {</td><td>174 |</td><td>175 }</td><td>176 ~</td><td>177 del</td></tr>
</table>

FILES

found in /usr/pub

NAME

core − format of core image file

DESCRIPTION

UNIX writes out a core image of a terminated process when any of various errors occur. See *signal (II)* for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called ‘‘core’’ and is written in the process’s working directory (provided it can be; normal access controls apply).

The first 1024 bytes of the core image are a copy of the system’s per-user data for the process, including the registers as they were at the time of the fault. The remainder represents the actual contents of the user’s core area when the core image was written. If the text segment is write-protected and shared, it is not dumped; otherwise the entire address space is dumped.

The format of the information in the first 1024 bytes is described by the *user* structure of the system. The important stuff not detailed therein is the locations of the registers. Here are their offsets. The parenthesized numbers for the floating registers are used if the floating-point hardware is in single precision mode, as indicated in the status register.

fpsr	0004	
fr0	0006	(0006)
fr1	0036	(0022)
fr2	0046	(0026)
fr3	0056	(0032)
fr4	0016	(0012)
fr5	0026	(0016)
r0	1772	
r1	1766	
r2	1750	
r3	1752	
r4	1754	
r5	1756	
sp	1764	
pc	1774	
ps	1776	

In general the debuggers *db (I)* and *cdb (I)* are sufficient to deal with core images.

SEE ALSO

cdb (I), db (I), signal (II)

-

NAME

dir – format of directories

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry. Directory entries are 16 bytes long. The first word is the i-number of the file represented by the entry, if non-zero; if zero, the entry is empty.

Bytes 2-15 represent the (14-character) file name, null padded on the right. These bytes are not cleared for empty slots.

By convention, the first two entries in each directory are for "**.**" and "**..**". The first is an entry for the directory itself. The second is for the parent directory. The meaning of "**..**" is modified for the root directory of the master file system and for the root directories of removable file systems. In the first case, there is no parent, and in the second, the system does not permit off-device references. Therefore in both cases "**..**" has the same meaning as "**.**".

SEE ALSO

file system (V)

NAME

dump − incremental dump tape format

DESCRIPTION

The *dump* and *restor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of blocks of 512-bytes each. The first block has the following structure.

```
struct {
        int     isize;
        int     fsize;
        int     date[2];
        int     ddate[2];
        int     tsize;
};
```

Isize, and *fsize* are the corresponding values from the super block of the dumped file system. (See file system (V).) *Date* is the date of the dump. *Ddate* is the incremental dump date. The incremental dump contains all files modified between *ddate* and *date.* *Tsize* is the number of blocks per reel. This block checksums to the octal value 031415.

Next there are enough whole tape blocks to contain one word per file of the dumped file system. This is *isize* divided by 16 rounded to the next higher integer. The first word corresponds to i-node 1, the second to i-node 2, and so forth. If a word is zero, then the corresponding file exists, but was not dumped. (Was not modified after *ddate)* If the word is −1, the file does not exist. Other values for the word indicate that the file was dumped and the value is one more than the number of blocks it contains.

The rest of the tape contains for each dumped file a header block and the data blocks from the file. The header contains an exact copy of the i-node (see file system (V)) and also checksums to 031415. The next-to-last word of the block contains the tape block number, to aid in (unimplemented) recovery after tape errors. The number of data blocks per file is directly specified by the control word for the file and indirectly specified by the size in the i-node. If these numbers differ, the file was dumped with a ‘phase error’.

SEE ALSO

dump (VIII), restor (VIII), file system(V)

-

NAME

fs − format of file system volume

DESCRIPTION

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the *super block.* Starting from its first word, the format of a super-block is

```
struct {
        int     isize;
        int     fsize;
        int     nfree;
        int     free[100];
        int     ninode;
        int     inode[100];
        char    flock;
        char    ilock;
        char    fmod;
        int     time[2];
};
```

Isize is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. *Fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an ‘‘impossible’’ block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *free* array contains, in *free[1], ... , free[nfree−1],* up to 99 numbers of free blocks. *Free[0]* is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain block is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 blocks is the link to the next member of the chain. To allocate a block: decrement *nfree,* and the new block is *free[nfree].* If the new block number is 0, there are no blocks left, so give an error. If *nfree* became 0, read in the block named by the new block number, replace *nfree* by its first word, and copy the block numbers in the next 100 words into the *free* array. To free a block, check if *nfree* is 100; if so, copy *nfree* and the *free* array into it, write it out, and set *nfree* to 0. In any event set *free[nfree]* to the freed block’s number and increment *nfree.*

Ninode is the number of free i-numbers in the *inode* array. To allocate an i-node: if *ninode* is greater than 0, decrement it and return *inode[ninode].* If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *inode* array, then try again. To free an i-node, provided *ninode* is less than 100, place its number into *inode[ninode]* and increment *ninode.* If *ninode* is already 100, don’t bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

Flock and *ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

Time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 0000 Jan. 1 1970 (GMT). During a reboot, the *time* of the super-block for the root file system is used to set the system’s idea of the time.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 32 bytes long, so 16 of them fit into a block. Therefore, i-node i is located in block $(i + 31) / 16$, and begins $32 \cdot ((i + 31) \pmod{16})$ bytes from its start. I-node 1 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node is as follows.

```
struct {
        int     flags;          /* +0: see below */
        char    nlinks;         /* +2: number of links to file */
        char    uid;            /* +3: user ID of owner */
        char    gid;            /* +4: group ID of owner */
        char    size0;          /* +5: high byte of 24-bit size */
        int     size1;          /* +6: low word of 24-bit size */
        int     addr[8];        /* +8: block numbers or device number */
        int     actime[2];      /* +24: time of last access */
        int     modtime[2];     /* +28: time of last modification */
};
```

The flags are as follows:

```
 100000   i-node is allocated
 060000   2-bit file type:
        000000   plain file
        040000   directory
        020000   character-type special file
        060000   block-type special file.
 010000   large file
 004000   set user-ID on execution
 002000   set group-ID on execution
 000400   read (owner)
 000200   write (owner)
 000100   execute (owner)
 000070   read, write, execute (group)
 000007   read, write, execute (others)
```

Special files are recognized by their flags and not by i-number. A block-type special file is basically one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files the high byte of the first address word specifies the type of device; the low byte specifies one of several devices of that type. The device type numbers of block and character special files overlap.

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large). Byte number *n* of a file is accessed as follows. *N* is divided by 512 to find its logical block number (say *b*) in the file. If the file is small (flag 010000 is 0), then *b* must be less than 8, and the physical block number is *addr[b].*

If the file is large, *b* is divided by 256 to yield *i.* If *i* is less than 7, then *addr[i]* is the physical block number of the indirect block. The remainder from the division yields the word in the indirect block which contains the number of the block for the sought-for byte.

If *i* is equal to 7, then the file has become extra-large (huge), and *addr[7]* is the address of a first indirect block. Each word in this block is the number of a second-level indirect block; each word in the second-level indirect blocks points to a data block. Notice that extra-large files are not marked by any mode bit, but only by having *addr[7]* non-zero; and that although this scheme allows for more than 256×256×512 = 33,554,432 bytes per file, the length of files is stored in 24 bits so in practice a file can be at most 16,777,216 bytes long.

For block *b* in a file to exist, it is not necessary that all blocks less than *b* exist. A zero block number either in the address words of the i-node or in an indirect block indicates that the corresponding block has never been allocated. Such a missing block reads as if it contained all zero words.

SEE ALSO

icheck, dcheck (VIII)

NAME

greek − graphics for extended TTY-37 type-box

SYNOPSIS

cat /usr/pub/greek

DESCRIPTION

Greek gives the mapping from ascii to the ‘‘shift out’’ graphics in effect between SO and SI on model 37 Teletypes with a 128-character type-box. It contains:

alpha	α	A	beta	β	B	gamma	γ	\
GAMMA	Γ	G	delta	δ	D	DELTA	Δ	W
epsilon	ε	S	zeta	ζ	Q	eta	η	N
THETA	Θ	T	theta	θ	O	lambda	λ	L
LAMBDA	Λ	E	mu	μ	M	nu	ν	@
xi	ξ	X	pi	π	J	PI	Π	P
rho	ρ	K	sigma	σ	Y	SIGMA	Σ	R
tau	τ	I	phi	ϕ	U	PHI	Φ	F
psi	ψ	V	PSI	Ψ	H	omega	ω	C
OMEGA	Ω	Z	nabla	∇	[	not	$\neg$	_
partial	∂	]	integral	$\int$	^			

SEE ALSO

ascii (VII)

-

NAME

group − group file

DESCRIPTION

Group contains for each group the following information:

group name
encrypted password
numerical group ID
a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; Each group is separated from the next by a newline. If the password field is null, no password is demanded.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

FILES

/etc/group

SEE ALSO

newgrp (I), login (I), crypt (III), passwd (I)

NAME

mtab – mounted file system table

DESCRIPTION

Mtab resides in directory */etc* and contains a table of devices mounted by the *mount* command. *Umount* removes entries.

Each entry is 64 bytes long; the first 32 are the null-padded name of the place where the special file is mounted; the second 32 are the null-padded name of the special file. The special file has all its directories stripped away; that is, everything through the last ‘‘/’’ is thrown away.

This table is present only so people can look at it. It does not matter to *mount* if there are duplicated entries nor to *umount* if a name cannot be found.

FILES

/etc/mtab

SEE ALSO

mount (VIII), umount (VIII)

BUGS

-

NAME

passwd − password file

DESCRIPTION

Passwd contains for each user the following information:

name (login name, contains no upper case)
encrypted password
numerical user ID
numerical group ID (for now, always 1)
GCOS job number, box number, optional GCOS user-id
initial working directory
program to use as Shell

This is an ASCII file. Each field within each user's entry is separated from the next by a colon. The GCOS field is used only when communicating with that system, and in other installations can contain any desired information. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, the Shell itself is used.

This file resides in directory /etc. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical user ID's to names.

FILES

/etc/passwd

SEE ALSO

login (I), crypt (III), passwd (I), group (V)

NAME

tabs − set tab stops

SYNOPSIS

cat /usr/pub/tabs

DESCRIPTION

Printing this file on a suitable terminal sets tab stops every 8 columns. Suitable terminals include the Teletype model 37 and the GE TermiNet 300.

These tab stop settings are desirable because UNIX assumes them in calculating delays.

-

NAME

tp – DEC/mag tape formats

DESCRIPTION

The command *tp* dumps files to and extracts files from DECtape and magtape. The formats of these tapes are the same except that magtapes have larger directories.

Block zero contains a copy of a stand-alone bootstrap program. See boot procedures (VIII).

Blocks 1 through 24 for DECtape (1 through 62 for magtape) contain a directory of the tape. There are 192 (resp. 496) entries in the directory; 8 entries per block; 64 bytes per entry. Each entry has the following format:

path name	32 bytes
mode	2 bytes
uid	1 byte
gid	1 byte
unused	1 byte
size	3 bytes
time modified	4 bytes
tape address	2 bytes
unused	16 bytes
check sum	2 bytes

The path name entry is the path name of the file when put on the tape. If the pathname starts with a zero word, the entry is empty. It is at most 32 bytes long and ends in a null byte. Mode, uid, gid, size and time modified are the same as described under i-nodes (file system (V)). The tape address is the tape block number of the start of the contents of the file. Every file starts on a block boundary. The file occupies (size+511)/512 blocks of continuous tape. The checksum entry has a value such that the sum of the 32 words of the directory entry is zero.

Blocks 25 (resp. 63) on are available for file storage.

A fake entry (see tp (I)) has a size of zero.

SEE ALSO

file system (V), tp (I)

NAME

ttys − typewriter initialization data

DESCRIPTION

The *ttys* file is read by the *init* program and specifies which typewriter special files are to have a process created for them which will allow people to log in. It consists of lines of 3 characters each.

The first character is either '0' or '1'; the former causes the line to be ignored, the latter causes it to be effective. The second character is the last character in the name of a typewriter; e.g. *x* refers to the file '/dev/tty*x*'. The third character is used as an argument to the *getty* program, which performs such tasks as baud-rate recognition, reading the login name, and calling *login.* For normal lines, the character is '0'; other characters can be used, for example, with hard-wired terminals where speed recognition is unnecessary or which have special characteristics. (Getty will have to be fixed in such cases.)

FILES

/etc/ttys

SEE ALSO

init (VIII), getty (VIII), login (I)

-

NAME

utmp – user information

DESCRIPTION

This file allows one to discover information about who is currently using UNIX. The file is binary; each entry is 16(10) bytes long. The first eight bytes contain a user's login name or are null if the table slot is unused. The low order byte of the next word contains the last character of a typewriter name. The next two words contain the user's login time. The last word is unused.

FILES

/etc/utmp

SEE ALSO

init (VIII) and login (I), which maintain the file; who (I), which interprets it.

NAME

wtmp – user login history

DESCRIPTION

This file records all logins and logouts. Its format is exactly like utmp (V) except that a null user name indicates a logout on the associated typewriter. Furthermore, the typewriter name '~' indicates that the system was rebooted at the indicated time; the adjacent pair of entries with typewriter names '|' and '}' indicate the system-maintained time just before and just after a *date* command has changed the system's idea of the time.

Wtmp is maintained by login (I) and init (VIII). Neither of these programs creates the file, so if it is removed record-keeping is turned off. It is summarized by ac (VIII).

FILES

/usr/adm/wtmp

SEE ALSO

utmp (V), login (I), init (VIII), ac (VIII), who (I)