SECTION THREE

PROGRAMMING FILE FORMATS

NAME

Files - special file formats

FUNCTION

The files documented in this section are those used in the process of developing programs under Idris, or those which are likely to be usable only by people with some knowledge of programming. All formats presented here are dictated, to a greater or lesser extent, by the Idris resident; most are produced or read directly by the resident.

Many other file formats of a more general nature may be found in Section III of the Idris Users' Manual.

NAME

bnames - block device names pseudo file

SYNOPSIS

/dev/bnames

FUNCTION

/dev/bnames is a character special file which reads out the names of the block special device handlers.

It consists of a concatenation of text lines, one for each entry in the table of block special devices; the first line corresponds to major device number 0.

/dev/bnames cannot be written.

SEE ALSO

cnames

NAME

cnames - character device names pseudo file

SYNOPSIS

/dev/cnames

FUNCTION

/dev/cnames is a character special file which reads out the names of the character special device handlers.

It consists of a concatenation of text lines, one for each entry in the table of character special devices; the first line corresponds to major device number 0.

/dev/cnames cannot be written.

SEE ALSO

bnames

NAME

core - core dump format

FUNCTION

core is the ancient term for main computer storage, dating back to the widespread use of magnetic cores to implement random access memory. A core dump, by extension, is an image of process memory, together with register contents and other status information, that is preserved when a process comes to an untimely end and may be in need of post mortem analysis. Idris can be coerced by a variety of means into producing a core dump, always into a file called "core" in the current directory. Such a file is best interpreted by the post mortem debugger, db, described in the C Interface Manual for the host machine.

A core dump image consists of a header followed by the user alterable portion of a process address space.

The header consists of an identification byte 0x9b, a configuration byte, a short int giving the size of the header in bytes, and six unsigned ints containing: the number of bytes defined by any separate text segment (or 0 if none), the number of bytes defined by the data segment (or combined text/data segment), the number of bytes between the end of the data segment and the top of stack at the time of the dump, the number of bytes between top of stack and the end of the address space (base of stack), the offset of any separate text segment, and the offset of the data segment (or combined text/data segment). The remainder of the header consists of an int giving the cause of death of the process, a machine-dependent panel containing the registers at the time of the dump, and sufficient filler bytes to bring the header length to the value indicated earlier.

The byte order and size of all ints in the header are determined by the configuration byte, which has the format given in the description of standard object files.

The dumped process image following the header contains the entirety of user alterable address space. If a separate "I-space" existed during the run, its contents are not dumped; otherwise, the text segment of the process is dumped immediately preceding its data segment.

SEE ALSO

object

NAME

inodes - resident inode list pseudo file

SYNOPSIS

/dev/inodes

FUNCTION

/dev/inodes is a character special file which reads out the resident list of inode. It is used by the standard utility ps to display the information in a format more or less palatable to human beings, or at least to gurus.

It consists of a concatenation of INODE entries, as documented in the resident header file /lib/fio.h.

/dev/inodes cannot be written.

SEE ALSO

mount

NAME

kmem - kernel memory pseudo file

SYNOPSIS

/dev/kmem

FUNCTION

/dev/kmem is a character special device that reads and writes kernel memory (where the Idris resident lives) as if it were a file. Nonexistent memory may be written, with no ill effect even in the presence of hardware trapping; nonexistent memory reads, if trapped, as all ones.

Its primary use is for on-the-fly inspection and patching of the resident, preferably by a guru armed with at least a symbol table.

SEE ALSO

mem

BUGS

It always accesses memory a byte at a time, which can confuse primitive memory mapped peripheral controllers wired only for word accesses.

NAME

library - standard library format

FUNCTION

Standard libraries are administered by the programming utility lib, primarily for conditional linking of object modules by the utility link. (Both utilities are documented in the C Interface Manual for any target machine.) They permit any number of files, typically object modules but possibly anything, to be administered as a single file; any file can be extracted by scanning the library for a matching name or other desirable properties. Standard library files are written in a machine independent fashion, so that they can be used unchanged across various implementations of Idris, plus other systems for which the portable C interface is supported.

The standard library format consists of a two-byte header having the value 0177565, written less significant byte first, followed by zero or more entries. Each entry consists of a fourteen-byte name, left justified and NUL padded, followed by a two-byte unsigned file length, also stored less significant byte first, followed by the file contents proper. If a name begins with a NUL byte, it is taken as the end of the library file.

Note that this differs in several small ways from UNIX/V6 archive file format, which has a header of 0177555, an eight-byte name, six bytes of miscellaneous UNIX-specific file attributes, and a two-byte file length. Moreover, a file whose length is odd is followed by a NUL padding byte in the UNIX format, while no padding is used in standard library format.

UNIX/V7 format is characterized by a header of 0177545, a fourteen-byte name, eight bytes of UNIX-specific file attributes, and a four-byte length. Odd length files are also padded to even.

The utility lib is capable of administering any of these formats.

BUGS

There should be a NUL at the end of all libraries, so that they are properly terminated even when written on a diskette.

NAME

mem - user memory pseudo file

SYNOPSIS

/dev/mem

FUNCTION

/dev/mem is a character special device that reads and writes per process (user) memory as if it were a file. This is seldom wise but occasionally useful.

Its behaviour in the presence of illegal access requests is the same as kmem.

SEE ALSO

kmem

NAME

mount - resident mount list pseudo file

SYNOPSIS

/dev/mount

FUNCTION

/dev/mount is a character special file which reads out the resident list of mounted filesystems. It is used by the standard utility ps to determine the number of filesystems actually mounted; it is capable of delivering even more information.

It consists of a concatenation of MOUNT entries, as documented in the resident header file /lib/fio.h.

/dev/mount cannot be written.

SEE ALSO

inodes

NAME
myps - current user process status psuedo file

SYNOPSIS
/dev/myps

FUNCTION
/dev/myps is a character special file which reads out the resident process list. It is used by the standard utility ps to determine the status of processes started at a given terminal.

Its behaviour is identical to /dev/ps, except that only processes having the same controlling teletype as the reader are read out.

SEE ALSO
ps

NAME

object - relocatable object file format

FUNCTION

For a file to be executable under Idris, it must look like an "object file", a file produced by or for the program builder utility called link, which is described in the C Interface Manual for the target machine. This is a (possibly) special case of a "relocatable" object file, one that contains sufficient information to be combined (by link) with other object files and/or altered to execute properly in different parts of memory. Thus, object files are widespread, and form an important part of the Idris environment.

Note that a file may have execute permission and still not be an object file -- it may be a directory with scan permission or it may be a command script to be interpreted by the shell. Nor is an object file necessarily executable -- it may be a bootstrap file (image of the resident), or a file executable on another machine, or a file that must be combined by link with other object files to make a complete program (its name probably ends in ".o" in this case).

A relocatable object image consists of a header, followed by a text segment, a data segment, the symbol table, and relocation information.

The header consists of an identification byte 0x99, a configuration byte, a short int containing the number of symbol table bytes, and six unsigned ints giving: the number of bytes of object code defined by the text segment, the number of bytes of object code defined by the data segment, the number of bytes needed by the bss segment, the number of bytes needed for stack plus heap, the text segment offset, and the data segment offset. Byte order and size of all ints in the header are determined by the configuration byte.

The configuration byte contains all information needed to fully represent the header and remaining information in the file. Its value val defines the following fields: ((val & 07) << 1) + 1 is the number of characters in the symbol table name field, so that values [0, 7] provide for odd lengths in the range [1, 15]. If (val & 010) then ints are four bytes; otherwise they are two bytes. If (val & 020) then ints are represented least significant byte first, otherwise, most significant byte first; byte order is assumed to be purely ascending or purely descending. (val & 0140) >> 5 is the strongest enforced storage bound restriction; values of 0, 1, 2, 3 provide for bounds that are multiples of 0, 2, 4, 8 bytes, respectively. If (val & 0200) no relocation information is present in this file.

The text segment is relocated relative to the text segment offset given in the header (usually zero), while the data segment is relocated relative to the data segment offset (usually the end of the text segment). In all cases the bss segment is relocated relative to the end of the data segment.

Relocation information consists of two sucessive byte streams, one for the text segment and one for the data segment, each terminated by a zero con-

trol byte. Control bytes in the range [1, 31] cause that many bytes in the corresponding segment to be skipped; bytes in the range [32, 63] skip 32 bytes, plus 256 times the control byte minus 32, plus the number of bytes specified by the relocation byte following.

All other control bytes control relocation of the next short or long int in the corresponding segment. If the 1-weighted bit is set in such a control byte, then a change in load bias must be subtracted from the int. The 2-weighted bit is set if a long int is being relocated instead of a short int. The value of the control byte right-shifted two places, minus 16, constitutes a "symbol code".

A symbol code of 47 is replaced by a code obtained from the byte or bytes following in the relocation stream. If the next byte is less than 128, then the symbol code is its value plus 47. Otherwise, the code is that byte minus 128 plus 175, the sum times 256, plus the value of the next relocation byte after that one.

A symbol code of zero calls for no further relocation; 1 means that a change in text bias must be added to the item (short or long int); 2 means that a change in data bias must be added; 3 means that a change in bss bias must be added. Other symbol codes call for the value of the symbol table entry indexed by the symbol code minus 4 to be added to the item.

Each symbol table entry consists of a value int, a flag byte, and a name padded with trailing NULs. Meaningful flag values are 0 for undefined, 4 for defined absolute, 5 for defined text relative, 6 for defined data relative, and 7 for defined bss relative. To this is added 010 if the symbol is to be globally known. If an undefined symbol has a non-zero value, it is taken as a request to reserve at least that many bytes for the symbol in the bss area, starting at the strongest required storage boundary.

SEE ALSO

core

NAME

profile - profile dump format

FUNCTION

A profile of a program is a histogram over time, showing how often each portion of the program was caught in execution, plus a list of entry counts for each of the instrumented functions that comprise the program. It is used by programmers, in conjunction with the prof post processor described in the C Interface Manual for the target machine, to debug and tune programs in an execution environment.

A profile dump file is produced at the end of an instrumented program execution. Its format is uniform across implementations of Idris, consisting of a header, followed by the profiling buffer and an array of function entry counts.

The header consists of an identification byte 0x9a, a configuration byte, a short int giving the number of bytes in the entry count array, and six unsigned ints giving: the (now meaningless) run-time address of the profiling buffer, the size of the buffer in bytes, the run-time offset subtracted from the pc to index the buffer, the scaling factor applied to the modified pc (given as a binary fraction in the interval [0, 1)), the text segment offset, and the offset from each function entry point of the corresponding address in the entry count array. Note that the first four ints are just the arguments to the prof system call, which is described in Section II of this manual.

The byte order and size of all ints in the header (and the byte order of ints elsewhere) are determined by the configuration byte, which has the format given in the description of standard object files. The header is immediately followed by a buffer of the size indicated, consisting of an array of short ints each of which contains a count of the clock ticks at which the pc was observed in the range of addresses corresponding to that element of the array.

Following the buffer is an array indicating the number of entries made to profiled functions during the run. Each element of the array is a structure with a pointer, called addr, and a long integer, called count. addr indicates the function to which this descriptor applies, and count contains the number of calls made to it. Specifically, addr contains the return address visible to the counting routine called at the start of each function, and so points some small (currently fixed) number of bytes above the actual entry point of the function. Its offset from the entry point is indicated by the final int in the dump file header. Note that the size of the pointer, and the byte ordering, in this record are determined by the configuration byte.

SEE ALSO

core, object

NAME

ps - process status psuedo file

SYNOPSIS

/dev/ps

FUNCTION

/dev/ps is a character special file which reads out the resident process list. It is used by the standard utility ps to determine the status of all processes in the system.

It consists of a concatenation of PROC plus ZLIST entries, one set for each of the processes administered by the resident. The structures delivered up are exact images, in native byte order, of the structures as documented in the resident header file /lib/cpu.h.

A PROC structure is followed by a list of zombies, or ZLIST structures, only if the p_zlist pointer is not NULL; the last ZLIST structure for a given process has a next field of NULL. Similarly, the PROC list ends with a structure having a next field of NULL.

/dev/ps cannot be written.

SEE ALSO

myps