SECTION TWO

THE PROCESS INTERFACE

2. THE PROCESS INTERFACE

This section describes the interface between the IDRIS resident and processes running under its control. A "process" is the execution of a program under IDRIS. IDRIS creates processes from executable object files, which are built by the **link** utility. These object files, which must be in a format acceptable to the IDRIS resident, define one part of the process interface. Similarly, the memory image of an active process and the format of a core dump file (produced for an aborted process) are also elements of this interface definition.

While a process is executing under IDRIS, it can request a variety of services from the IDRIS resident. These services, known as system calls, deal with process administration and various aspects of input/output. System calls constitute the remainder of the process interface. The standard system library functions, described in Section II of the IDRIS Programmers' Manual, use these calls internally to perform their respective functions. These library functions are sufficient for most C language programming under IDRIS, and details presented here can be ignored in such cases. However, for assembly language programming, and to create customized system interface libraries, the programmer should be familiar with these details.

2.1. Executable Object Format

An executable binary image of a process consists of a leading header, followed by a text segment, a data segment, a symbol table (which may be stripped), and relocation information (which may be stripped also). The header consists of the byte 0231 **octal**, followed by a configuration byte, followed by seven (2-byte) unsigned words specifying: the number of symbol table bytes (ignored by the IDRIS resident), the number of bytes of code defined by the text segment (text size), the number of bytes defined by the data segment (data size), the number of bytes defined by the bss segment, the number of bytes needed for stack plus heap, the text segment offset (text bias), which must be zero, and the data segment offset (data bias).

A legal configuration byte for the 8086 family may have any value in the range [060, 067] octal.

2.2. Memory Image

A program's text segment will be shared if the following conditions hold:

the text size is a multiple of 512 bytes (and non-zero);
the text bias is a multiple of 512 (including zero);
the data bias is a multiple of 512 (including zero).

A shared program will have one copy of its text in memory or in the swap area. There will be a separate copy of the data area for each invocation. If the inode "sticky bit" is set, (via the command **chmod -01555 <program>**), the shared program's text image will remain in swap space even after all invocations of the program exit. If there is no swapping, the sticky bit has no effect.

Notice the fifth word of the 0231 type header: IDRIS takes this value, if nonzero, as the minimum number of bytes to reserve at execution time for stack and data area growth. If the value is zero, IDRIS gives the program a default stack and heap allocation defined by the sysgen parameter **stdheap**. If the value is odd, IDRIS will execute the program with a smaller heap plus stack size, making the process fit in the available memory.

A program must reserve the first 32 bytes of its text segment for a system call area. The resident checks the first 4 bytes for a certain pattern, then rewrites the 28 bytes that follow with a routine which does system calls for the program.

The program must include this header:

```
.text
top:
    jmp.s   top + 32
    .word   0x6969
. = top + 32
```

When first started, (by an **exec** system call) a process has **cs** set to the start of its text section, and **ds**, **ss**, and **es** set so that the first byte of the data section is offset by the **data bias**. **ip** is zero, so execution begins with the **jmp** instruction at the start of the text section. **sp** is set as described under the **exec** system call, and the bss section plus remaining stack and heap are set to zeros. All other registers are also set to zero.

Currently only the "small" model of computation is supported, i.e. it is assumed that only 16-bit offsets (pointers) are used, and that the segment registers are never modified by the process. IDRIS may move the process about in memory, and adjust the segment registers accordingly. Thus, all instruction addresses are assumed to be code segment offsets, and all other (data) addresses are assumed to be data segment offsets.

The program must use a **call *4** instruction to initiate a system call. The system call routine inserted by the resident doesn't preserve register **cx**.

2.3. Core Dump Format

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

The header consists of an identification byte (0233 octal), a configuration byte (064 octal), a word giving the size of the header in bytes, and six unsigned words containing:

a) the number of bytes defined by any separate text segment,

b) the number of bytes defined by the data segment,

c) the number of bytes between the end of the data segment and the top of stack at the time of the dump,

d) the number of bytes between top of the stack and the end of the address space (base of stack),

e) the offset of the text segment (always zero), and

f) the offset of the data segment.

The remainder of the header consists of a word giving the signal that was the cause of death and the machine panel containing the registers at the time of the dump. The registers available under IDRIS are: **ax**, **cx**, **bx**, **dx**, **sp**, **bp**,

si, di, es, cs, ss, ds, ip, and fl. Additionally, a word follows indicating whether the floating point processor was present, saved between process switches, and dumped. If this word is 1, the floating point processor contents follow; otherwise, sufficient padding is left in its place to fill out the panel. The 8087 registers are; as words or arrays of words: fcw, fsw, ftw, fip[2], fop[2], and fr[5][8]. Finally, the 8-byte program name follows the panel. See PAN86.H for the register layout.

The dumped process image following the header contains the entirety of user-alterable address space. The text segment of the process is dumped immediately preceding its data segment.

2.4. The IDRIS CPU Scheduler

The multi-task IDRIS environment is managed by a "hybrid" scheduler, which allows for both deterministic prioritization for real-time processing and reasonable response time for interactive timesharing users.

This scheduler is entirely event-driven. The process at the highest priority is run when an event occurs. The system timer, if enabled, provides for preemption of compute-bound (i.e. using the CPU only) tasks via a highly tuneable "time quantum" scheme. There is a built-in relationship between a process's priority and the time quantum it is allocated at that priority, but the value of a time quantum is tuneable over a wide range. When the time quantum for the current running process expires, that process is put at the end of the list of running processes at the same priority. A real-time process keeps its original priority, but a timesharing process decreases in priority over time. As this decrease occurs, any other timesharing processes on the run list increase in priority, thus building in a kind of "fairness" relative to how long a given job must wait to run.

2.4.1. IDRIS system clock

IDRIS counts time in 300 HZ increments. 300 HZ was chosen because, as the least common multiple of 50 HZ and 60 HZ, it allows hardware timer interrupts to correspond to some discrete number of IDRIS timer units on most U.S. and European hardware (i.e. on a system with a 50 HZ timer, a single timer interrupt is equivalent to exactly 6 timer units of one three-hundredth of a second each). Timer interrupts over the range 0 to 300 HZ are supported.

The system clock controls timekeeping, I/O timeout, and multi-task scheduling. Time-of-day is kept to a resolution of 1 HZ. I/O timeout is kept to a resolution of 60 HZ, and scheduling is tuneable, usually to a resolution of from 4 to 10 HZ.

2.4.2. Time Quantums

The "time quantum" for a process is the maximum amount of time that a process will execute before it is pre-empted by the scheduler. The actual value of the time quantum for a given process is a function of its priority, in conjunction with two other "sysgen" parameters, both of which are user-settable provided the user has an IDRIS Sysgen Kit. The relationships between these parameters are defined as follows:

time slice = **schrate** / 300 HZ

minquan = X number of time slices

time quantum = | priority | + minquan

The variable **schrate** specifies the "scheduler rate". This value, divided by the number of timer units per second (always 300 under IDRIS) sets the length of the time slice. **schrate** is a value which users can set using the Sysgen Kit. The default value of **schrate** is 75, giving a default time slice of 250

milliseconds. The variable **minquan** is a given number of time slices, also user-settable via the Sysgen Kit. The default value is 3 (i.e. 3 time slices is the minimum quantum). **minquan** is, by definition, the length of one time quantum for a process at priority 0. Hence the name "minimum quantum".

A time quantum corresponds to the absolute value of a process' priority, plus the minimum quantum. More specifically, this relationship holds true in all cases where a "priority bias" for a process has not been set via the **nice** command or system call. (This concept is explained in detail in subsection 2.4.5.) In cases where **nice** is used to modify a process' time quantum, its time quantum is a function of the absolute value of the "priority bias", plus the minimum quantum. Thus, the more highly positive (low) or negative (high) a priority, the more time it will spend executing before being pre-empted. Since a process is scheduled based on its priority, the higher its priority, the less realtime wait a process will have before it is run again. Processes with priorities in the range [0, +5] for example, have short time quantums but often preempt running processes. This allows for reasonable response times for interactive processes, which fall into this range.

2.4.3. Time Quantum Allocation

Processes are initially allocated a full time quantum. A process is given an additional time quantum only after using all of the first (assuming that nothing else of high priority needs to run and that the process doesn't road-block). IDRIS decrements the time quantum with each passing time slice. The process is assumed to have run throughout the previous time slice, even if it has not.

2.4.4. Redetermining Process Priority

When the time quantum for a given process reaches 0, the priority for that process is recomputed, and a new time quantum allocated based on the result. The rules for establishing new priorities are as follows:

1.) if the process had 0 or positive priority, its new priority will be:

```
new priority = old priority + 1
```

2.) if the process had negative priority, its priority is fixed and does not change, allowing for "round-robin" scheduling of CPU time among processes of the same priority.

The scheduler's "run list" is rebuilt each time that the time quantum for the current running process expires. For processes of negative (high) priority, the scheme is straightforward. Only the superuser or the resident itself may give a process a negative priority. Once given, the priority is fixed, and scheduling is therefore "deterministic", or determined solely by the relative importance of the job. Negative priorities cause time-division multiplexing of 2 or more processes of the same (negative) priority. Processes with priorities in the range [-20, -128] are locked in memory: that is, they are never swapped out. Processes with priorities in the range [-3, -20] are grouped by priority and run in round-robin fashion, being allocated new time quantums and going to the end of the queue of processes of equal priority until they exit and are done. Priorities in the range [0, -3] are for internal

IDRIS use; for block, inode, and swap I/O.

Positive-priority processes, conversely, vary in priority while they remain on the run list. When the current process finishes its time quantum, 1 is added to its priority, as stated above. In the meantime, the priorities of all processes having priorities lower than the current running process are decremented by 1, and new time quantums are assigned to them on that basis. Thus, processes requiring more CPU time gradually get lower and lower priority, but run longer and longer.

These rules support the concepts of "fairness" to processes that have been bypassed, and allow processes that have been running to "age", or gradually decrease in priority. In no case is the priority of a process increased beyond +5. This in effect creates an "interactive" priority domain [0, +5] for "fast" terminal response and a "compute-bound" priority domain [+5, +127] for CPU-intensive processes.

2.4.5. Bias Priority Using 'nice'

The priority "bias value" for a process can be set via a call to the **nice** command or system call. (Actually, the **nice** command invokes the **nice** system call.) Users can be "nice" to the system by giving lower priority to tasks requiring a lot of CPU time. In short, users can directly determine the time quantum they want for a process through the use of **nice**.

If the bias is a negative number, however, then this bias value is used as the fixed priority of the process. Such a process must be privileged and is scheduled as a real time task. If the bias is 0 or positive, then this bias becomes the highest priority a process may achieve. Such a process loses priority as it executes and is scheduled as a timesharing task. The default priority bias is 0.

When a timesharing process is restarted upon the completion of terminal input, it is put at priority 0 plus its priority bias, and it is immediately run if it now has the highest priority of any process on the run list. As it runs it quickly loses priority and begins to compete with processes already running. The processes which were pre-empted had been gaining priority at the same time that this process was losing priority. However, a process will only increase in priority up to a point. That point is referred to as a priority "fence". This moveable fence is at 5 plus the priority bias set by the call to **nice**. Thus, the priority range [0, +5] is reserved for interactive tasks.

2.4.6. Tuning the Scheduler

If you have a Sysgen Kit, you can modify user-settable parameters within the scheduler to suit your environment, accounting for such factors as CPU speed and number of users. Tinker with these values as follows:

1.) Determine optimum scheduler rate (**schrate**). Base this on how often you want the scheduler to consider pre-empting a process. For most environments, a reasonable rate for this to happen is 4 to 10 times a second. The default **schrate** is 75, giving a **time slice** of .25 seconds. A **schrate** of 30 gives a **time slice** of .10 seconds. In general, the faster the CPU, the lower **schrate** can be. Frequent reshuffling of processes adds to

overhead, but provides faster response time for interactive processes.

A higher **schrate** value lowers system overhead, but delays response to interactive users. This may be an advantage on a slow or overworked CPU, where bigger time slices let jobs get done that might otherwise never complete.

2.) Select a value for **minquan**. **minquan** determines the length of a **time quantum** at priority 0. **minquan** in large part determines response time for interactive processes (priority == 0 to 5) by establishing the smallest time quantum for them. A **minquan** of 0 disables aging and pre-emption of all processes at (pri == 0).

Bypassed processes never "rejuvenate" (i.e. increase in priority) beyond +5. IDRIS as shipped has values of 75 for **schrate** and 3 for **minquan**, giving a default time quantum of .75 seconds. Thus, two competing processes switching off at priority 5 would have time quantums of (5 + 3), or 8, yielding time slices of 2 seconds each. [Note: If both processes don't fit in memory, they'll swap at this rate also.]

A process will never increase in priority (rejuvenate) higher than its priority bias plus 5. Since the default priority bias is 0, a priority "fence" is created at priority 5. By using the **nice** system call (see Section II of the IDRIS Programmers' Manual) or the **nice** prefix command (see Section II of the IDRIS Users' Manual), a process priority can be set to an even lower (more positive) value. The "fence" can thus be moved for a process and its decendants. (Priority bias is inherited, or passed from parent to child process, on a **fork** system call.)

2.4.7. Example Uses of 'nice'

The following examples make use of the "priority bias" concept in several appropriate contexts. An explanation follows each sample command line.

```
# nice +5 read_proc &
```

This superuser shell command starts the **nice** prefix command in the background. The **nice** command simply does a **nice** system call, setting the priority bias to negative 5 and then executing the program **read_proc**. **read_proc** will then run at the fixed high priority of -5. **read_proc** is presumably a program which spends most of its time waiting for some input or signal, and then runs immediately when input occurs.

```
% nice make_the_world &
```

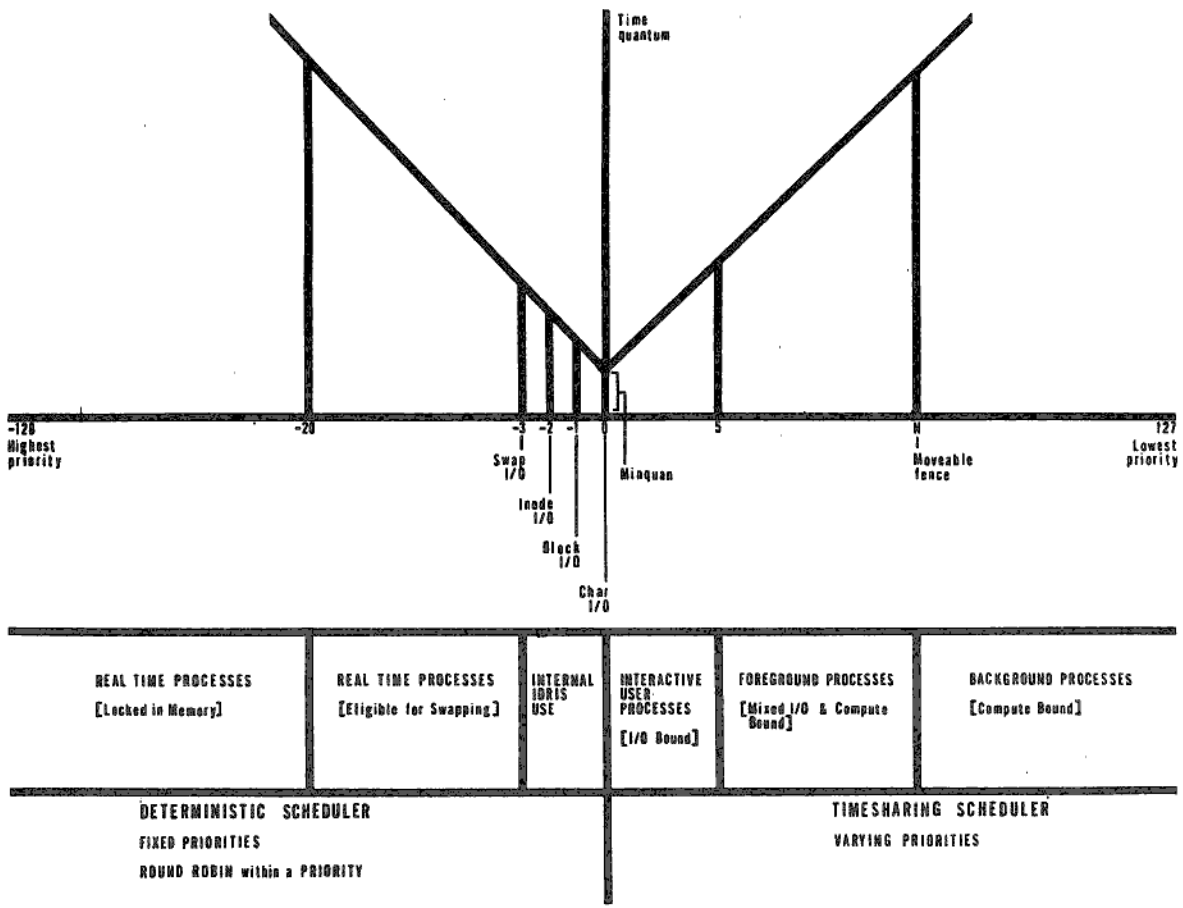
This shell command starts the **nice** prefix command in the background. **nice** sets a priority bias of +20 by default, and then executes the **make_the_world** script. Any of the commands run by this script will run at priority 25 (20 + 5) or lower, and therefore interfere very little with other interactive users.

```
% nice nohup error -s sh -e make_this > /tmp/TRACE &
```

This example illustrates a combination of the prefix commands **nice**, **nohup**, and **error**. (For details of these commands see the appropriate manual pages in Section II of the IDRIS Users' Manual.) STDOUT (standard output) is

redirected to /tmp/TRACE, and **nice** is then started. **nice** lowers the priority of all subsequent commands, and then starts **nohup**. **nohup** prevents hangup, quit, or DEL-key signals from reaching subsequent commands, and then runs **error**. **error -s** redirects STDERR (standard error) to STDOUT before starting **sh**. (Error output now goes to /tmp/TRACE as well.) **sh -e** echoes each line of the **make_this** script to the already-redirected STDOUT. All output of the programs started by **make_this** is collected in the file /tmp/TRACE. This file may be typed out at any time in order to follow the progress of the **make_it** sequences.

IDRIS SCHEDULER



2.5. IDRIS Swapping Strategy

Swapping programs between disk and memory only occurs when there are more processes created than can fit in available memory. Processes may be swapped out of memory even though they are "running" (i.e. in the "run" state). The CPU scheduler described above concerns itself with processes that are running **and** in memory. As long as a program is not in memory (by virtue of the fact that it has been swapped out) process 0, called **SWAP**, is always trying to bring it in. Other processes may be swapped out to make room for processes on disk to come into memory. The business of the swapper program **SWAP** (process 0) is to keep memory filled with running processes of the highest priority. Processes that are not running are in a "wait" state, and have associated with them a "waiting" priority. Such processes often require the receipt of some signal before they can run again. The swapper will consider "waiting" processes as a group before "running" processes as candidates to be swapped out. When the time comes for a process waiting on disk to be "awakened" (i.e. put into the "run" state) the swapper tries to bring it into memory. It can only do so if there are processes in memory of lower priority than the waiting process. Thus, all processes of lower priority that are in the "run" state, as well as all processes in the "wait" state, become candidates for swap out.

It is also possible for a running process to be swapped out because an even higher-priority process needs to be run (given that all waiting processes have already been kicked out). The swapper then immediately attempts to bring these processes right back in. Incessant back-and-forth swapping (known as "thrashing") is prevented by a combination of the priority scheme and intra-systemic hysteresis or feedback. When a process is swapped into memory a flag is set which inhibits further swapping until it runs to the end of its time quantum or until it roadblocks (i.e. goes to a "wait" state).

2.6. System Calls

The following pages describe each of the system calls supported by IDRIS. Special terminology and error codes associated with the calls are discussed first, followed by details of each call. You may want to familiarize yourself with these basic definitions, so that they will make sense in the context in which they appear later on.

2.6.1. Conventions

A number of special terms are used to help document the IDRIS system calls. They are:

dev - an unsigned short integer, the more significant byte of which is the "major number" of a physical device, of which the less significant byte is the "minor number" of that device. The major number is used as an index into one of two resident device tables (for block special or character special devices) to select a device handler. The device number **dev** is stored in place of the first disk block pointer in a block or character special inode and is provided as part of the file status **fstat**. The full device code is also passed in the device handler. Note that the device number is written on disk, less significant byte first, regardless of target machine.

fd - a short integer, capable of holding a valid file descriptor returned by IDRIS. Its range is currently [0, 19].

fname - a pointer to a character, usually the first of a NUL-terminated string of characters variously known as a string, filename, or pathname.

long - a 32-bit signed integer. On the 8086, bytes are stored in ascending order of significance; when in **ax/dx**, **ax** holds the less significant half.

native mode - the representation strategy used by the CPU usually refers to the ordering of significant bytes in an integer compared to increasing byte addresses in memory.

orphan - a process, the parent process of which has exited. The exit status of orphaned processes is not saved.

pid - a short integer capable of holding any processid. Valid processids are always positive and nonzero. Some **pids** are special. **pid** 0 is the swapper, which is always resident and ignores all signals. **pid** 1 is the initialization process **init**, which is restarted by the resident whenever it exits.

pointer - the address of an item.

short - a 16-bit signed integer. On the 8086, the less significant byte is stored at the lower address.

zombie - the exit status of a process that has terminated, but is not yet laid to rest. This status information is read and deleted when the parent process does a **wait** system call.

2.6.2. Error Codes

Error codes are returned to a process when a system call fails. The codes are used to indicate the general nature of the failure. Most system calls can return a variety of error codes, and most error codes can be returned by a variety of system calls.

The error codes are:

Value	Mnemonic	Meaning
1	EPERM	user lacks permission
2	ENOENT	directory entry does not exist
3	ESRCH	can't find process to signal
4	EINTR	system call was interrupted
5	EIO	unrecoverable physical I/O error
6	ENXIO	nonexistent I/O device
7	E2BIG	argument list too big for exec
8	ENOEXEC	wrong file format for exec
9	EBADF	bad file descriptor
10	ECHILD	no children to wait for
11	EAGAIN	failed because memory is temporarially unavailable
12	ENOMEM	program is too big for physical memory
13	EACCES	file access prohibited
15	ENOTBLK	not a block special device
16	EBUSY	can't mount or unmount busy filesystem
17	EEXIST	file already exists
18	EXDEV	can't link across filesystems
19	ENODEV	not a defined operation for the device
20	ENOTDIR	not a directory
21	EISDIR	file is a directory
22	EINVAL	illegal argument value on system call
23	ENFILE	too many files opened system wide
24	EMFILE	too many files opened by process
25	ENOTTY	not a tty
26	ETXTBSY	shared text portion of file is in use by exec
27	EFBIG	file too big
28	ENOSPC	no space on filesystem

29	ESPIPE	can't seek on a pipe
30	EROFS	read-only filesystem
31	EMLINK	too many links to a file
32	EPIPE	write to broken pipe
33	EDOM	math function argument outside defined domain
34	ERANGE	math function result outside representable range
100	ENOSYS	undefined system call
106	EFAULT	can't access argument on system call

The error code is always returned as a native mode integer, negated (i.e. made negative).

2.6.3. Descriptions of Each Call

Detailed descriptions of each IDRIS system call are presented in the following manual pages.

The assembly language code fragment needed to generate each call is described in the **SYNOPSIS** section of each manual page. Note that the IDRIS resident does not pop the arguments off the stack, so the sequences shown must be followed by code to balance the stack. In fact, the system calling conventions are optimized for writing functions called from C, with the arguments placed in the stack in the proper order by the C calling sequence. **<dummy-ret>** is thus a symbolic placeholder for the return link pushed on a C function call, which is otherwise unused during the system call.

Every call returns a value in **ax**, and sets or resets the carry flag. If an error has occurred, the value in **ax** is the negated IDRIS error code, and the carry flag is set. If the expected returned value is more then one word, **dx** is used.

To check the outcome of a system call, it is sufficient to test the returned value in **ax**; the carry flag can be ignored in all cases. Therefore, a C callable function can simply return, since **ax** (or **dx/ax**) holds the conventional function return value.

NAME

brk - set system break to address

SYNOPSIS

```
push <addr>
push <dummy-ret>
mov  ax, 17
call *4
```

FUNCTION

brk sets the system break , at the top of the data area, to **addr.** Addresses between the system break and the current top-of-stack are not considered part of the valid process image; they may or may not be preserved. It is considered an error for the top-of-stack ever to extend below the system break.

If the system break is moved to a higher address, all the bytes between the old address and the new one are made zero.

If **addr** is -1, the current setting is returned, and the system break is not changed.

RETURNS

brk returns the old system break in **ax.** If an error occurs, **ax** is set to -1.

NOTE: **brk** is often called via the function **sbreak,** which moves the break by a specified increment instead of setting it to a given number.

NAME

chdir - change working directory

SYNOPSIS

```
push <fname>
push <dummy-ret>
mov  ax, 12
call #4
```

FUNCTION

chdir changes the working directory to **fname.**

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is returned in **ax.** Otherwise, the system returns with a zero in **ax.**

NAME

chmod - change mode of file

SYNOPSIS

```
push <mode>
push <fname>
push <dummy-ret>
mov  ax, 15
call *4
```

FUNCTION

chmod changes the mode of the file **fname** to match **mode**. Only the low-order 12 bits of **mode** are used, to specify **ugtrwxrwxrwx**, where **u** is the set userid bit, **g** is the set groupid bit, **t** is the save text image bit (the "sticky bit"), and the **rwx** groups give access permissions for the file. Access permissions are **r** for **read**, **w** for write, and **x** for **execute** (or scan permission for a directory); the first group applies to the owner of the file, the second to the group that owns the file, and the third to all other system users.

Only the superuser can alter the save text image bit **t**. Only the superuser or owner of the file may alter the other eleven bits.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

NAME

chown - change owner of file

SYNOPSIS

```
push <owner>
push <fname>
push <dummy-ret>
mov  ax, 16
call *4
```

FUNCTION

chown changes the owner of the file **fname** to be the less significant byte of **owner**, and changes its group to be the more significant byte of **owner**. Only the superuser or the owner succeeds with this call. The owner and group are the same as the **uid** and **gid** fields from a **stat** or **fstat** inode structure.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

getgid, getuid

NAME
close - close a file

SYNOPSIS
```
push <fd>
push <dummy-ret>
mov  ax, 6
call *4
```

FUNCTION
close closes the file associated with the file descriptor **fd**, making **fd** available for future **open** or **creat** calls.

RETURNS
If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO
creat, open

NAME

creat - make a new file

SYNOPSIS

```
push <perm>
push <fname>
push <dummy-ret>
mov  ax, 8
call #4
```

FUNCTION

creat makes a new file with name **fname** if it did not previously exist, or truncates the length of any existing file with that name to zero. In the former case, the file is given access permission specified by **perm**; in the latter, the access permission is left unchanged. Access permissions are described under **chmod**. The file is then opened for writing. The lowest numbered file descriptor is returned.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, **creat** returns a file descriptor for the created file in **ax**.

SEE ALSO

chmod, close, open

BUGS

The file should be opened for updating. Current behavior is a holdover from UNIX.

NAME

csw - get console switches

SYNOPSIS

```
mov  ax, 38
call *4
```

FUNCTION

csw reads the console switches.

RETURNS

csw always succeeds in reading the console switches, even if they don't exist. The value to be returned is determined once at system startup and is returned in **ax**.

BUGS

Most systems have no console switches in real life. In this case a constant is returned. In CO-IDRIS on DOS systems, this call causes a return to DOS.

NAME

dup - duplicate a file descriptor

SYNOPSIS

```
push <fd>
push <dummy-ret>
mov  ax, 41
call *4
```

FUNCTION

dup allocated a file descriptor that points at the same file, and has the same current offset, as the file descriptor **fd**. It is promised that the lowest-numbered available file descriptor is allocated on any **creat**, **dup**, **open**, or **pipe** call, so file descriptors can be rearranged by judicious use of **dup** and **close** calls.

RETURNS

dup returns the newly allocated file descriptor in r0. If an error occurs, the carry flag is set and the IDRIS error code is in **ax**.

SEE ALSO

close, creat, open, pipe

NAME

exec - execute a file with arguments

SYNOPSIS

```
push <NULL>
push <last _arg>
    ...
push <first _arg>
push <fname>
push <dummy-ret>
mov  ax,11
call *4
```

FUNCTION

exec invokes the executable binary program file **fname** and passes it the NUL-terminated vector of arguments at **args.** The invoked file overlays the current program, inheriting all its open files and ignored signals; the current program is forever gone. Signals that were caught by the current program revert to system handling.

If the "set userid" bit in the mode for **fname** is set, the effective userid of the invoked file becomes that of the owner of the file; the effective groupid may be changed in a similar manner by the "set groupid" bit.

The "save text" bit on a shareable program causes the program's text image to forever remain in the swap area. This image is maintained as a pseudo process and may be removed with a **SIGKIL** signal.

The invoked program begins execution at the start of the text section, with the string arguments at the top of user memory, just above the stack, (i.e., a NULL fence is initially pushed on the stack, followed by pointers to the stacked strings in reverse order, followed by a count of the number of strings passed as arguments). Thus, for a machine with 2-byte integers:

```
[0][sp]   is the count of arguments, typically > 0.
[2][sp]   points to the zeroeth argument string
[4][sp]   points to the first argument string, etc.
```

Note that the stack is not well conditioned for direct entry into a C function; the C runtime startup header is usually linked at the start of the text section. It enforces conventions such as setting **_pname**, isolating the execution search path, calling the C main function, and doing an **exit** system call.

By convention, the zeroeth argument is always present and is taken as the name **_pname** by which the file is invoked; if it contains a vertical bar |, the string before the first vertical bar is taken as argument zero and the string after that bar is taken as a search path, or concatenation of filename prefixes separated by vertical bars, used for locating executable files. Additional arguments are typically optional; their interpretation is left purely to the whim of the invoked program.

RETURNS

A successful **exec** never returns to the caller. If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Specifically, **E2BIG** means that too many argument characters are being sent, **ENOMEM** means that the program is too large for available memory, and **ENOEXEC** means that the program has execute permission but is not a proper binary object module. **EAGAIN** means that no swap space is available.

Up to 4096 bytes of argument characters (including the terminating NULs) may be passed.

SEE ALSO

exit, fork

NAME

exit - terminate program execution

SYNOPSIS

```
push <status>
push <dummy-ret>
mov  ax, 1
call *4
```

FUNCTION

exit terminates program execution. Typically, a zero byte is returned to the parent, indicating successful termination. Otherwise, abnormal termination is indicated by a 1 byte returned to the parent.

RETURNS

exit will never return to the caller.

NAME

fork - create a new process

SYNOPSIS

```
mov  ax, 2
call *4
```

FUNCTION

fork creates a new process that is identical to the initial process from which it is generated. All open files and all signal settings are the same in both processes. It is customary for the child process to invoke a program file via **exec**, shortly after birth, while the parent either waits to learn the termination status of the child or proceeds to other matters.

RETURNS

In the parent process, **fork** returns the processid of the child in **ax** if successful. In the child process, **ax** contains 0. If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Failure occurs only if the system is out of resident heap space or out of swap space.

SEE ALSO

exec, wait

BUGS

If the parent process doesn't wait, the child process will remain a zombie until the parent dies. A prolific parent can thus overpopulate the system.

NAME

fstat - get status of open file

SYNOPSIS

```
push <buf>
push <dummy-ret>
mov  ax, 28
call #4
   ...
buf:
dev:     0
ino:     0
mode:    0
nlinks: .byte 0
uid:    .byte 0
gid:    .byte 0
msize:  .byte 0
lsize:   0
addr:    0; 0; 0; 0; 0; 0; 0; 0;
maketime:    0; 0;
dumptime:    0; 0;
```

FUNCTION

fstat obtains the status of the opened file **fd** in the 36-byte structure pointed to by **buf**. The structure is essentially that of a filesystem inode, preceded by the device **dev** and the inode number **ino** on that device. The less significant byte of **dev** is the device minor number; the more significant byte is its major number.

mode takes the form **azzlugtrwxrwxrwx**, where **a** is set to indicate that the file is allocated. **zz** is 00 for a plain file, 01 for a character special file, 10 for a directory, and 11 for a block special file. **l** is set for a large (4096 <= size) file. The remaining bits give access permissions in the normal manner, as described under **chmod** in this section of the manual.

nlinks counts the number of directory entries that point at the inode in question (whose number is **ino**). **uid** is the userid of the owner, and **gid** is the groupid of whatever group owns the file. The size of the file in bytes is formed by constructing a 24-bit integer from **lsize** and **msize**, with **lsize** making up the low-order 16 bits.

For character and block special files, the first word of **addr** contains the device major and minor numbers, the latter in the less significant byte. The 8 block addresses are otherwise magic (i.e. uninterpretable) from the standpoint of most users.

The last time the file contents were changed (**maketime**), and the last time the inode or file contents were changed (**dumptime**) are both in seconds from the 1 Jan 1970 epoch and are expressed in unsigned long (32-bit) integers.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in ax. Otherwise, the system returns with a zero in ax.

SEE ALSO

chmod, stat

NAME
getgid - get real and effective groupid

SYNOPSIS
```
mov  ax, 47
call *4
```

FUNCTION
getgid obtains the current real and effective groupids.

RETURNS
getgid always returns the real groupid in **al**. **ah** contains the effective groupid.

SEE ALSO
exec, setuid

NAME

getpid - get processid

SYNOPSIS

```
mov  ax, 20
call *4
```

FUNCTION

getpid obtains the processid of the currently running process, which is not very meaningful, but has the virtue of being unique among all executing and zombie processes. Hence it serves as a useful seed for temporary filenames.

RETURNS

getpid returns the (always positive) processid in ax.

SEE ALSO

fork

NAME
getuid - get real and effective userid

SYNOPSIS
```
mov  ax, 24
call *4
```

FUNCTION
getuid obtains the current real and effective userids.

RETURNS
getuid always returns the real userid in **al**. **ah** contains the effective userid.

SEE ALSO
exec, setuid

NAME

gtty - get tty status

SYNOPSIS

```
        push <buf>
        push <fd>
        push <dummy-ret>
        mov  ax, 32
        call #4
           ...
        buf:
        speeds: 0
        erase:  .byte 0
        kill:   .byte 0
        bits:   0

        /   masks for bits field
            rare    = 01
            xtabs   = 02
            mapuc   = 04
            echo    = 010
            crlf    = 020
            raw     = 040
            odd     = 0100
            even    = 0200
            dnl     = 01400
            dht     = 06000
            dcr     = 030000
            dff     = 040000
            dbs     = 0100000

        /   masks for speeds field
            ispeed = 0x000f
            ibreak = 0x0x0010
            ilost  = 0x0020
            iready = 0x0080
            ospeed = 0x0f00
            obreak = 0x1000
            oready = 0x8000
```

FUNCTION

gtty obtains the status of a tty or other character special device, under control of **fd**, that responds to **stty** system calls. For a tty, six bytes of status are returned for the character special file **fd**, the information being written in the structure pointed at by **buf**. Other character special devices may refuse to honor a **gtty** request, or they return other than six characters, depending mainly upon the device driver. See Section III for details on the device driver interface. If the device is a tty, the information can be interpreted as follows:

speeds contains the input channel baud rate and input flags in its less significant byte, and the output channel baud rate and output flags in its more significant byte. Baud rates should have values in the range [0,16), corresponding to the baud rates {0, 50, 75, 110,

134.5, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400}. A baud rate of 0 calls for the tty to hang up.

ibreak means that the driver has received a break. **ilost** means that the driver has detected a data overrun error, indicating that input data has been lost. **iready** means that a read on this tty will return (i.e. input is ready). **ibreak** and **ilost** are reset by **gtty**, and set by the driver. **iready** is computed for each **gtty** call.

obreak is a request to the driver to send a break. **obreak** is reset by the driver when the break is sent. **oready** is set by the **gtty** call when there are no characters in the output queue. **obreak** will be zero when the last character is dequeued (transmission may be in progress when this occurs).

erase is the character that, if typed in other than raw mode, calls for the preceding character on the current line (if any) to be deleted. **kill** is the character that calls for the entire current line to be deleted. Defaults are '\b' (backspace) and '\021' (ctl-u). If the sign bit of either **erase** or **kill** is set, the <backspace> <space> <backspace> sequence (used for non-CRT terminals) will not be emitted.

rare (the least significant bit) calls for the tty to to enter a form of raw mode in which the interrupt characters **DEL** and **FS** are recognized, and **X-ON** and **X-OFF** processing is in effect. Input is 8-bit transparent and output is 7-bit transparent with parity generation.

xtabs calls for each tab to be expanded to spaces.

mapuc maps all uppercase characters typed in to lowercase, and all lowercase characters typed out to uppercase.

echo steers all characters typed in back out for full duplex operation.

crlf accepts carriage returns (**CR**) as linefeeds (**LF**), and expands all typed-out **LF**s to **CR-LF** sequences.

raw instructs the handler to ignore interpretation of input characters, including the processing of **erase** and **kill** characters, the recognition of interrupt codes **DEL** and **FS**, and the treatment of **EOT** as end of file. Characters are read and written transparently as 8-bit bytes, with no parity generation and with no typeout delays.

odd and **even** select the parity generation to be performed on output. If neither is set, a zero is generated. If only **odd** is set, only odd parity is generated. If only **even** is set, only even parity is generated. If both are set, a 1 is generated.

Various typeout delays may be requested for newlines with **dnl**, horizontal tabs with **dht**, carriage returns with **dcr**, formfeeds and vertical tabs with **dff**, and for backspaces with **dbs**. Actual delays are in multiples of 1/60 second ticks. The ranges are (0, 4, 8, 12) ticks for horizontal tabs, newlines, and carriage returns; (0, 64) for formfeeds and vertical tabs; and (0, 16) for backspaces.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

stty

NAME

kill - send signal to a process

SYNOPSIS

```
push <sig>
push <pid>
push <dummy-ret>
mov  ax, 37
call *4
```

FUNCTION

kill sends the signal **sig** to the process identified by processid **pid.** The sender must either have the same effective userid as the receiver or be the superuser; if **pid** is zero, the signal is sent to all processes under control of the same tty as the sender. A process cannot, however, kill itself.

The signals that may be sent are:

NAME	VALUE	MEANING
SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3	quit (core dump)
SIGILIN	4	illegal instruction (core dump)
SIGTRC	5	trace trap (core dump)
SIGRNG	6	range error (core dump)
SIGDOM	7	domain error (core dump)
SIGFPT	8	floating point exception (core dump)
SIGKILL	9	kill
SIGBUS	10	bus error (core dump)
SIGSEG	11	segmentation violation (core dump)
SIGSYS	12	bad system call (core dump)
SIGPIPE	13	broken pipe
SIGALRM	14	alarm clock
SIGTERM	15	terminate
SIGUC	16	intercept system calls
SIGUSR	17	user defined

A core dump may not occur on those signals that have been caught or ignored by the receiving process. Note that **SIGALRM** and **SIGTERM** are not defined in all versions of UNIX, and that **SIGRNG** and **SIGDOM** have somewhat less general meanings on UNIX systems.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax.** Otherwise, the system returns with a zero in **ax.**

SEE ALSO

signal

BUGS

kill is a misnomer, since it can be used to perform many other functions.

NAME

link - create link to file

SYNOPSIS

```
push <newfname>
push <oldfname>
push <dummy-ret>
mov  ax, 9
call *4
```

FUNCTION

link creates a new directory entry for a file with existing name **oldfname** the added name is **newfname** . No checks are made to see whether or not this is a good idea.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

unlink

BUGS

A program executing as superuser can scramble a directory structure by injudicious calls on link.

NAME

mknod - make a special inode

SYNOPSIS

```
push <dev>
push <mod>
push <fname>
push <dummy-ret>
mov  ax, 14
call *4
```

FUNCTION

mknod creates an empty instance of a file with pathname **fname**, setting its mode bits to **mode** and its first address entry to **dev**. If (mode == 0140777), for instance, the new file will be a directory with general permissions; **dev** had better be zero in this case. If (mode == 0160644), the new file will be a block special device that can be read by all, but written only by the owner; **dev** then specifies the major/minor device numbers, the minor number being in the less significant byte.

Only the superuser may perform this call successfully.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

NAME

mount - mount a filesystem

SYNOPSIS

```
push <ronly>
push <fname>
push <spec>
push <dummy-ret>
mov  ax, 21
call *4
```

FUNCTION

mount associates the root of the filesystem written on the block special device pathname **spec** with the pathname **fname**; henceforth the mounted filesystem is reachable via pathnames through **fname**. If **ronly** is nonzero, the filesystem is mounted read-only (i.e. all files are write protected and access times are not updated).

fname must already exist; its contents are rendered inaccessible by the mount operation.

Only the superuser succeeds with this call.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

umount

NAME

nice - set priority

SYNOPSIS

```
push <pri>
push <dummy-ret>
mov  ax, 34
call #4
```

FUNCTION

nice modifies the scheduling priority of a process according to **pri**, which must be in the range [0, 20] for all but the superuser, who succeeds with values in the range [-128, 127]. The higher the number, the lower the priority [sic]; the default is zero.

A **pri** of -20. will lock a process into memory, so that it is never swapped out. Once locked in, it remains in memory until executed or until "unlocked" by being given a **pri** greater than 0. A program may perform two calls to **nice** (**pri** == -20 and **pri** == 0 for example), which would lock it in memory with normal scheduling.

A **pri** of less than 0 permanently sets the priority of a process, and that priority never increases or decreases. This is referred to as "deterministic priority" (see the writeup on the IDRIS scheduler in Section II of this manual).

A **pri** of 0 allows for normal scheduling. Programs lose priority as they run, and interactive programs get their priority rejuvenated with each interactive event. Preemption may occur at the end of a time slice.

A **pri** of greater than 0 creates a priority bias that effectively lowers the priority of the running program below the default priority assigned by the scheduler. A positive **pri** value also selects a system-dependent time slice, to be used regardless of the old or default priority of the process. Normally, the current priority selects a time slice, with smaller time slices being assigned to processes having higher priorities.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

BUGS

nice is a misnomer, since it can be used to do things that aren't nice.

NAME

open - open a file

SYNOPSIS

```
push <mode>
push <fname>
push <dummy-ret>
mov  ax, 5
call *4
```

FUNCTION

open opens an existing file with name **fname** and assigns a file descriptor to it. If (mode == 0), the file is opened for reading. If (mode == 1), it is opened for writing; otherwise, (mode == 2) of necessity, and the file is opened for updating (reading and writing). The smallest available file descriptor is allocated.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, open returns a file descriptor for the file in **ax**.

SEE ALSO

close, creat

NAME

pipe - set up a data pipe

SYNOPSIS

```
mov  ax, 42
call *4
```

FUNCTION

pipe sets up a pipeline , i.e., a data transfer mechanism that can be read by one file descriptor and written by another. The lowest numbered file descriptors are returned.

Since child processes created by a call to **fork** in the parent process inherit all open files, it is possible to use pipe to set up a communication link between processes having a common parent. The pipe mechanism synchronizes reading and writing, allowing the process invoking **pipe** to get ahead up to 4096 bytes before being made to wait.

Reading an empty pipe with no writeable files left results in an end-of-file return; a pipe with no readable files causes a broken pipe signal and, if the signal is ignored, causes an error return on subsequent writes.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the read file descriptor is in **ax** and the write file descriptor is in **dx**.

SEE ALSO

close, creat, dup, exec, fork, open, read, write

NAME

profil - set profiler parameters

SYNOPSIS

```
push <scale>
push <offset>
push <size>
push <buf>
push <dummy-ret>
mov  ax, 44
call *4
```

FUNCTION

profil sets the parameters used by the system for execution time profiling of the user-mode program counter. On each clock tick (60 times per second), **offset** is subtracted from the user-program counter and the result is multiplied by **scale**, which is taken as an unsigned binary fraction in the interval [0, 1]. If the result is in the interval [0, size], it is used as an index to select the element of **buf** to increment.

Unlike UNIX, IDRIS assumes that the binary fraction is always one less (when taken as an integer) than a power of two. That is, the resident considers only the highest order bit set in **scale**, and assumes that all bits to the right of it are ones. This difference should be transparent to all known uses of **profil**.

If **scale** is 0, profiling is turned off.

RETURNS

The system returns a zero in ax.

NAME

read - read from a file

SYNOPSIS

```
push <size>
push <buf>
push <fd>
push <dummy-ret>
mov  ax, 3
call *4
```

FUNCTION

read reads up to **size** characters from the file specified by **fd** into the buffer starting at **buf.**

RETURNS

If end of file is encountered, **read** returns zero in **ax;** otherwise, the value returned in **ax** is between 1 and **size,** inclusive. When reading from a disk file, **size** bytes are read whenever possible. If an error occurs, the carry flag is set and the IDRIS error code is in **ax.**

SEE ALSO

write

NAME

seek - set file read/write pointer

SYNOPSIS

```
push <sense>
push <offset_hi>
push <offset_lo>
push <fd>
push <dummy-ret>
mov  ax, 19
call *4
```

FUNCTION

seek uses the offset provided to modify the read/write pointer for the file **fd**, under control of **sense**. If (**sense** == 0), the pointer is set to **offset**, which is treated as an unsigned integer; if (**sense** == 1), the offset is algebraically added to the current pointer; if (**sense** == 2), the offset is algebraically added to the length of the file in bytes to obtain the new pointer.

If **sense** is between 3 and 5 inclusive, the offset is multiplied by 512 and the resultant long offset is used with (**sense** - 3). IDRIS currently uses only the low-order 24 bits of the offset; the rest are ignored. Block offsets are of use primarily in machines with short pointers.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

NAME

setgid - set groupid

SYNOPSIS

```
push <gids>
push <dummy-ret>
mov  ax, 46
call *4
```

FUNCTION

setgid sets the groupid, both real and effective, of the current process to **gid**. The real groupid is in the less significant byte, and the effective groupid is in the more significant byte. Only the superuser may change the **gid**.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

getgid

NAME

setuid - set userid

SYNOPSIS

```
push <uid>
push <dummy-ret>
mov  ax, 23
call #4
```

FUNCTION

setuid sets the userid of the current process, both real and effective, to **uid**. The real groupid is in the less significant byte, and the effective groupid is in the more significant byte. Only the superuser may change the uid.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in ax.

SEE ALSO

getuid

NAME

signal - capture signals

SYNOPSIS

```
push <pfunc>
push <signo>
push <dummy-ret>
mov  ax, 48
call *4
```

FUNCTION

signal changes the handling of the signal **signo** according to **pfunc**. Legal values of **signo** are described under the **kill** system call. If **pfunc** is **NULL**, normal system handling occurs, i.e., the process is terminated when the signal occurs, possibly with a core dump. If **pfunc** is **NOSIG** (i.e. is equal to 1) the signal is ignored. Otherwise, **pfunc** is taken as a pointer to an instruction sequence to be entered in the user program when the signal occurs.

Note that the code sequence may not, in general, be a C function, since registers may not be properly saved and the stack may not be prepared for an orderly return. To return properly to the interrupted code, a machine-dependent code sequence must often be performed. If a system call was interrupted and the signal handler returns properly to the interrupted code, the system call reports an abnormal termination with the **EINTR** error code.

Except for "illegal instruction" and "trace trap", all signals revert to system handling after each occurrence. All signals revert to system handling on an **exec**, but not on a **fork**.

RETURNS

signal returns the old **pfunc** if successful. If an error occurs, the carry flag is set and the IDRIS error code is in **ax**.

SEE ALSO

exec, fork, kill

NAME

sleep - delay for awhile

SYNOPSIS

```
push <secs>
push <dummy-ret>
mov  ax, 35
call #4
```

FUNCTION

sleep suspends execution of the current process for **secs** seconds.

RETURNS

The system returns a zero in **ax** after the delay.

NAME
stat - get status of named file

SYNOPSIS

```
push <buf>
push <fname>
push <dummy-ret>
mov  ax, 18
call *4
```

FUNCTION
stat obtains the status of the file **fname** in the structure pointed to by **buf**. The structure is essentially that of a filesystem inode, preceded by the device **dev** and the inode number **ino** on that device; it is described under **fstat**.

RETURNS
If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO
chmod, fstat

NAME

stime - set system time

SYNOPSIS

```
push <time_hi>
push <time_lo>
push <dummy-ret>
mov  ax, 25
call *4
```

FUNCTION

stime sets the system time to **time**, an unsigned long which is the number of seconds since the 1 Jan 1970 epoch. Only the superuser succeeds with this call.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

time

NAME

stty - set tty status

SYNOPSIS

```
push <buf>
push <fd>
push <dummy-ret>
mov  ax, 31
call *4
```

FUNCTION

stty sets the status of a tty or other character special device, under control of the file descriptor **fd**, to the values in the structure pointed at by **buf**. The structure is the same as described under **gtty**.

Any type-ahead is discarded on transitions between raw, rare, and normal modes.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

gtty

NAME

sync - synchronize disks with memory

SYNOPSIS

```
mov  ax, 36
call #4
```

FUNCTION

sync ensures that all delayed writes are performed by the system, so that disk integrity is assured before taking the system down. It updates all inodes that have been modified since the last **sync**, and writes all data blocks not correctly represented on open or mounted block special devices.

If a filesystem is to be accessed in some way other than through the block special file on which it is mounted, **sync** should be performed first to ensure that the disk image is current.

RETURNS

The system returns a zero in ax.

NAME
time - get system time

SYNOPSIS
```
mov  ax, 13
call *4
```

FUNCTION
time gets the system time, which is the number of seconds since the 1 Jan 1970 epoch.

RETURNS
time returns the system time as an unsigned long integer in **ax** and **dx**. **ax** holds the low order 16 bits.

SEE ALSO
stime

NAME

times - get process times

SYNOPSIS

```
push <buf>
push <dummy-ret>
mov  ax, 43
call *4
   ...
buf:
putime: 0
pstime: 0
cutime: 0; 0
cstime: 0; 0
```

FUNCTION

times returns the cumulative times consumed by the current process and all its child processes in the structure pointed at by **buf**. **putime** is the user-mode time consumed by the process proper; **pstime** is its system-mode time. **cutime** and **cstime** are the cumulative user and system times consumed by all the child processes that have had their execution suspended by **wait** system calls, including the times of all their children (i.e. the children of the children of the parent process also subjected to suspended execution via calls to **wait**).

All times are in 1/60 second ticks. **putime** and **pstime** are unsigned integers. **cutime** and **cstime** are unsigned long (32-bit) integers.

RETURNS

times writes the process times in the structure pointed at by **buf**. The system returns a zero in **ax**.

SEE ALSO

time

NAME

umount - unmount a filesystem

SYNOPSIS

```
push <spec>
push <dummy-ret>
mov  ax, 22
call *4
```

FUNCTION

umount disassociates the root of the filesystem written on the block special device pathname **spec** with whatever node it was mounted on; henceforth the filesystem is no longer reachable via the directory tree.

Only the superuser succeeds with this call.

RETURNS

If an error occurs, the carry flag is set and the IDRIS error code is in **ax**. Otherwise, the system returns with a zero in **ax**.

SEE ALSO

mount

NAME

unlink - erase link to file

SYNOPSIS

```
push <fname>
push <dummy-ret>
mov  ax, 10
call #4
```

FUNCTION

unlink removes the link specified by the file **fname**. No checks are made to see whether or not this is a good idea. Only the superuser may unlink a directory.

RETURNS

unlink returns zero in **ax** if successful. If an error occurs, the carry flag is set and the IDRIS error code is in **ax**.

SEE ALSO

link

BUGS

A program executing as superuser can scramble a directory structure by injudicious calls to **unlink**.

NAME

wait - wait for child to terminate

SYNOPSIS

```
mov ax, 7
call *4
```

FUNCTION

wait suspends execution of the calling program until a child process terminates, so that it can return the child's termination status in **dx** and its processid in **ax**. Child processes retain zombie status until laid to rest by the parent process via a **wait** system call.

The status returned contains the number of the terminating signal, if any, in its less significant byte, and the status reported back by the child's **exit** call in its more significant byte. By convention, a status word of all zeros means normal termination. If **ax** has the 200 bit turned on, a core dump has been made.

RETURNS

wait returns the processid of the child in **ax**. The status of said child process is is **dx**. If an error occurs, the carry flag is set and the IDRIS error code is in **ax**.

SEE ALSO

fork

NAME

write - write to a file

SYNOPSIS

```
        push <size>
        push <buf>
        push <fd>
        push <dummy-ret>
        mov  ax, 4
        call #4
```

FUNCTION

write writes **size** characters starting at **buf** to the file specified by **fd**.

RETURNS

If an error occurs, the carry flag is set and the the IDRIS error code is in **ax**. Otherwise, the value returned is between 1 and **size**, inclusive.

SEE ALSO

read