Linux System Calls for HLA Programmers

1 Introduction

This document describes the interface between HLA and Linux via direct system calls. The HLA Standard Library provides a header file with a set of important constants, data types, and procedure prototypes that you can use to make Linux system calls. Unlike the "C Standard Library," the HLA-based systems calls add very little overhead to a system call (generally, they move parameters into registers and invoke Linux with very little other processing).

Note that I have copied information from the Linux *man* pages into this document. So whatever copyright (copyleft) applies to that information applies here as well. As far as I am concerned, my work on this document is public domain, so this document inherits the Linux documentation copyright (I don't know the details, but it's probably the "Free Documentation" license; whatever it is, it applies equally here).

Note that Linux *man* pages are known to contain some misinformation. That misinformation was copied straight through to this document. Of course, in a document of this size, there are probably some defects I've introduced as well, so keep this in mind when using this document.

Disclaimer: I would like to claim that every effort has been taken to ensure the accuracy of the material appearing within this document. That, however, would be a complete lie. In reality, I copied the man pages to this document and make a bunch of quick changes for HLA/assembly programmers to their descriptions. Undoubtedly, this process has introduced additional defects into the descriptions. Therefore, if something doesn't seem right or doesn't seem to work properly, it's probably due to an error in this documentation. Keep this in mind when using this document. Hopefully as time passes and this document matures, many of the defects will disappear. Also note that (as this was begin written) I have not had time to completely test every system call, constant, and wrapper function provided with HLA. If you're having a problem with a system call, be sure to check out the HLA source code for the Linux wrapper functions and whatever constants and data types you're using from the linux.hhf module.

1.1 Direct System Calls from Assembly Language

To invoke a Linux system call (i.e., a Linux API call), you load parameters into various registers, load a system call opcode into the EAX register, and execute an INT($80) instruction. Linux returns results in the EAX register and, possibly, via certain pass by reference parameters.

The HLA "linux.hhf" header file contains constant declarations for most of the Linux system call opcodes. These constants take the form "sys_*function*" where *function* represents a Linux system call name. For example, "sys_exit" is the symbolic name for the Linux "_exit" call (this constant just happens to have the value one).

If you read the on-line documentation for the Linux system calls, you'll find that the API calls are specified using a "C" language syntax. However, it's very easy to convert the C examples to assembly language. Just load the associated system call constant into EAX and then load the 80x86 registers with the following values:

- 1st parameter: EBX
- 2nd parameter: ECX
- 3rd parameter: EDX
- 4th parameter: ESI
- 5th parameter: EDI

Certain Linux 2.4 calls pass a sixth parameter in EBP. Calls compatible with earlier versions of the kernel pass six or more parameters in a parameter block and pass the address of the parameter block in EBX (this change was probably made in kernel 2.4 because someone noticed that an extra copy between kernel and user space was slowing down those functions with exactly six parameters; who knows the real reason, though).

As an example, consiider the Linux exit system call. This has a "C" prototype similar to the following:

```
void exit( int returnCode );
```

The assembly invocation of this function takes the following form:

```
        mov( sys_exit, eax );
        mov( returnCode, ebx );
        int( $80 );
```

As you can see, calls to Linux are very similar to BIOS or DOS calls on the PC (for those of you who are familiar with such system calls).

While it is certainly possible for you to load the system call parameters directly into the 80x86 registers, load a system call "opcode" into EAX, and execute an INT($80) instruction directly, this is a lot of work if your program makes several Linux system calls. To make life easier for assembly programmers, the Linux system call module provided with the HLA Standard Library provides *wrapper* functions that make Linux system calls a lot more convenient. These are functions that let you pass parameters on the stack (using the HLA high level procedure call syntax) which is much more convenient than loading the registers and executing INT($80). For example, consider the following implementation of the "linux._exit" function the Linux module provides:

```
procedure _exit( RtnCode: dword ); @nodisplay;
begin _exit;

    mov( sys_exit, eax );
    mov( RtnCode, ebx );
    int( $80 );

end _exit;
```

You can call this function using the HLA syntax:

```
    linux._exit( returnValue );
```

As you can see, this is far more convenient to use than the INT($80) sequence given earlier. Furthermore, this calling sequence is very similar to the "C" syntax, so it should be very familiar to those reading Linux documentation (which is based on "C").

Your code would probably be slightly smaller and a tiny bit faster if you directly make the INT($80) calls.. However, since the transition from user space to kernel space is very expensive, the few extra cycles needed to pass the parameters on the stack to the HLA functions is nearly meaningless. In a typical (large) program, the memory savings would probably be measured in hundreds of bytes, if not less. So you're not really going to gain much by making the INT($80) calls. Since the HLA code is much more convenient to use, you really should call the Standard Library functions. For those who are concerned about inefficiencies, here's what a typical HLA Standard Library Linux system call looks like. As you can see, there's not much to these functions. So you shouldn't worry at all about efficiency loss.

On occasion, certain Linux system calls become obsolete. Linux has maintained the calls for the older functions for those programs that require the old semantics, while adding new API calls that support additional features. A classic example is the LSEEK and LLSEEK functions. Originally, there was only LSEEK (that only supports two gigabyte file lengths). Linux added the LLSEEK function to allow access to larger files. Still, the old LSEEK function exists for code that was written prior to the development of the LLSEEK call. So if you use the INT($80) mechanism to invoke Linux, you probably don't have to worry too much about certain system calls disappearing on you.

There is, however, a *big* advantage to using the HLA wrapper functions. If you use the INT($80) calling mechanism and a system call becomes obsolete, your program will probably still work but it won't be able to take advantage of the new Linux features within your program without rewriting the affected INT($80) calls. On the other hand, if you call the HLA wrappers, this problem exists in only one place -- in the HLA Standard Library wrapper functions. This means that whenever the Linux system calls change, you need only modify the affected wrapper function (typically in one place), recompile the HLA Standard Library, recompile your applications, and you're in business. This is *much* easier than attempting to locate every INT($80) call in your code and checking to see if you need to change it. Combined with the ease of calling the HLA wrapper functions, you should serious consider whether it's worth it to call Linux via INT($80). For this reason, the remainder of this document will assume that you're using the HLA Linux module to call the Linux APIs. If you choose to use the INT($80) calling mechanism instead, conversion is fairly trivial (as noted above).

1.2 A Quick Note About Naming Conventions

Most Linux documentation was written assuming that the reader would be calling Linux from a C/C++ program. While the HLA header files (and this document) attempt to stick as closely to the original Linux names as possible, there are a few areas where HLA names deviate from the C names. This can occur for any of three reasons:

- The C name conflicts with an HLA reserved word (e.g., "exit" becomes "_exit" because "exit" is an HLA reserved word).
- C uses different namespaces for structs and other objects and some Linux identifiers are the same for both structs and variables (HLA doesn't allow this).
- HLA uses case neutral identifiers, C uses case sensitive identifiers. Therefore, if two C identifiers are the same except for alphabetic case, one of them must be changed when converting to HLA.
- Many Linux constant and macro declarations use the (stylistically dubious) convention of all uppercase characters. Since uppercase is hard to read, such identifiers have been converted to all lowercase in the HLA header files.

1.3 A Quick Note About Error Return Values

C/C++ programmers probably expect Linux system calls to return -1 if an error occurs and then they expect to find the actual error code in the errno global variable. Assembly language calls to Linux return the error status directly in the EAX register. Generally, if the return value is non-negative, this indicates success and the value is a function result. If the return value from a Linux system call is negative, this usually indicates some sort of error. Therefore, an assembly language program should test the value in EAX upon return for negative/non-negative to determine the error status.

Linux system calls return a wide range of negative values indicating different error values. The "linux.hhf" header file defines a set of symbolic constants in the errno namespace so you can use symbolic names rather than literal constants. These names and values are identical to those found in standard Linux documentation with three exceptions: (1) the HLA naming convention uses lower case for these identifiers rather than all uppercase letters; e.g., EPERM is spelled eperm. (2) Since the HLA names are all found in the errno namespace, you refer to them using an "errno." prefix, e.g., "errno.eperm" is the correct way to specify the "error, invalid permissions" error code. (3) The constants in the errno namespace are all negative. You do not have to explicitly negate them before comparing them with the Linux system call return result (as you would when using the C constant declarations).

2 Linux System Calls, by Functional Group

This section will describe the syntax and semantics of some of the more common Linux system calls, organized by the type of the call.

If you read the Linux on-line documentation concerning the system calls, keep on thing in mind. C Standard Library semantics require that a system call return "-1" for an error and the actual error value is in the errno global variable. Direct Linux system calls don't work this way. They usually return a negative value to indicate an error and a non-negative return value to indicate success. The error values that Linux functions typically return is the negated copy of the value the "C" documentation describes for errno return values.

2.1 File I/O

This section describes the Linux system calls responsible for file I/O. The principal functions are open, close, creat, read, write, and llseek. Advanced users make want to use some of the other functions as well.

2.1.1 File Descriptors

Linux and applications refer to files through the use of a *file descriptor* or *file handle*. This is a small unsigned integer value (held in a dword) that Linux uses as an index into internal file tables. When you open or create a file, Linux returns the file descriptor as the open/creat result. You pass this file handle to other functions to operate upon that file.

Under Linux, the file handle values zero, one, and two have special meaning. These correspond to the files associated with the standard input, standard output, and standard error devices. Theoretically, you should use constants for these values (e.g., stdin.handle, stdout.handle), but their values are so entrenched in modern UNIX programs that it would be very difficult for Linux kernel developers to change these values. Generally, the first file a process opens is given the value three for its file handle and successively open files are given the next available (sequential) free handle value. However, your programs certainly should not count on this behavior. Someone may decide to add another "standard" device handle to the mix, or a kernel developer may decide it's better to start the handles with a larger value for some reason. In general, other than to satisfy your curiosity, you should never examine or modify the file handle value. You should treat the value as Linux's private data.

2.2 linux.pushregs / linux.popregs

The HLA wrapper functions typically use the linux.pushregs and linux.popregs macros to preserve and restore affected register in Linux system calls. These macros push and pop EBX, ECX, EDX, ESI, and EDI. Note that the wrappers for the Linux system calls provided in the Linux module preserve all 80x86 registers except EAX (where the functions place the return result). Note, however, that the wrapper functions do not preserve the flags register (though you can generally assume that important state flags, like the direction flag, are not modified by Linux system calls).

2.3 More to come later....

I'm releasing this document without a lot of tutorial information because it will take a while to write this information and I don't want to delay the release of the reference material in the next section until the tutorials are complete. Check back later, I'll try to have more written in the near future.

3 Linux System Call Reference

The following sub-sections describe each of the Linux system calls. Note that this information was taken from the Linux MAN page system and modified slightly for assembly/HLA programmers. Therefore, any errors present in the man pages will probably appear here. Further, it's likely some new problems where introduced when reformatting the man pages and modifying them for HLA. So please keep this in mind when using this information.

3.1 linux.pushregs / linux.popregs

The HLA wrapper functions typically use the linux.pushregs and linux.popregs macros to preserve and restore affected register in Linux system calls. These macros push and pop EBX, ECX, EDX, ESI, and EDI. Note that the wrappers for the Linux system calls provided in the Linux module preserve all 80x86 registers except EAX (where the functions place the return result). Note, however, that the wrapper functions do not preserve the flags register (though you can generally assume that important state flags, like the direction flag, are not modified by Linux system calls).

The following is the source code for this macros at the time this document was written. Please see the linux.hhf file for the current definition of these macros as they are subject to change.

```
#macro pushregs;

     push( ebx );
     push( ecx );
     push( edx );
     push( esi );
     push( edi );

#endmacro;

#macro popregs;

     pop( edi );
     pop( esi );
     pop( edx );
     pop( ecx );
     pop( ebx );

#endmacro;
```

3.2 A Quick Reminder About Error Return Values and Parameters

C/C++ programmers probably expect Linux system calls to return -1 if an error occurs and then they expect to find the actual error code in the errno global variable. Assembly language calls to Linux return the error status directly in the EAX register. Generally, if the return value is non-negative, this indicates success and the value is a function result. If the return value from a Linux system call is negative, this usually indicates some sort of error. Therefore, an assembly language program should test the value in EAX upon return for negative/non-negative to determine the error status.

Linux system calls return a wide range of negative values indicating different error values. The "linux.hhf" header file defines a set of symbolic constants in the errno namespace so you can use symbolic names rather than literal constants. These names and values are identical to those found in standard Linux documentation with three exceptions: (1) the HLA naming convention uses lower case for these identifiers rather than all uppercase letters; e.g., EPERM is spelled eperm. (2) Since the HLA names are all found in the errno namespace, you refer to them using an "errno." prefix, e.g., "errno.eperm" is the correct way to specify the "error, invalid permissions" error code. (3) The constants in the errno namespace are all negative. You do not have to explicitly negate them before comparing them with the Linux system call return result (as you would when using the C constant declarations).

Note that the HLA Linux "wrapper" functions usually preserve all registers (except, obviously, EAX since Linux returns the error status in EAX). The wrapper functions use the Pascal calling sequence (rather than the C / CDECL calling sequence), so the wrapper function automatically removes all parameters from the stack upon returning to the caller. The calling code need not, and, in fact, must not, remove the parameters from the stack; remember, the wrapper functions are not C code; it is not the caller’s responsibility to remove parameters from the stack.

3.3 access - check user's permissions for a file

```
// access - Check to see if it is legal to access a file.

procedure linux.access( pathname:string; mode:int32 ); @nodisplay;
begin access;

    linux.pushregs;
    mov( linux.sys_access, eax );
    mov( pathname, ebx );
    mov( mode, ecx );
    int( $80 );
    linux.popregs;

end access;
```

DESCRIPTION

access checks whether the process would be allowed to read, write or test for existence of the file (or other file system object) whose name is pathname. If pathname is a symbolic link permissions of the file referred to by this symbolic link are tested.

mode is a mask consisting of one or more of R_OK, W_OK, X_OK and F_OK.

R_OK, W_OK and X_OK request checking whether the file exists and has read, write and execute permissions, respectively. F_OK just requests checking for the existence of the file.

The tests depend on the permissions of the directories occurring in the path to the file, as given in pathname, and on the permissions of directories and files referred to by symbolic links encountered on the way.

The check is done with the process's real uid and gid, rather than with the effective ids as is done when actually attempting an operation. This is to allow set-UID programs to easily determine the invoking user's authority.

Only access bits are checked, not the file type or contents. Therefore, if a directory is found to be "writable," it probably means that files can be created in the directory and not that the directory can be written as a file. Similarly, a DOS file may be found to be "executable," but the execve(2) call will still fail.

RETURN VALUE

On success (all requested permissions granted), zero is returned. On error (at least one bit in mode asked for a permission that is denied, or some other error occurred), a negative error code is returned in EAX.

ERRORS

errno.eacces	The requested access would be denied to the file or search permission is denied to one of the directories in pathname.
errno.erofs	Write permission was requested for a file on a read-only filesystem.

errno.efault pathname points outside your accessible address space.

errno.einval mode was incorrectly specified.

errno.enametoolong pathname is too long.

errno.enoent A directory component in pathname would have been accessible but does not exist or was a dangling symbolic link.

errno.enotdir A component used as a directory in pathname is not in fact, a directory.

errno.enomem Insufficient kernel memory was available.

errno.eloop Too many symbolic links were encountered in resolving pathname.

errno.eio An I/O error occurred.

RESTRICTIONS

access returns an error if any of the access types in the requested call fails, even if other types might be successful.

access may not work correctly on NFS file systems with UID mapping enabled, because UID mapping is done on the server and hidden from the client, which checks permissions.

Using access to check if a user is authorized to e.g. open a file before actually doing so using open(2) creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it.

CONFORMING TO

SVID, AT&T, POSIX, X/OPEN, BSD 4.3

SEE ALSO

stat(2), open(2), chmod(2), chown(2), setuid(2), setgid(2)

```
// access - Check to see if it is legal to access a file.

procedure linux.access( pathname:string; mode:int32 );
    nodisplay;
begin access;

    linux.pushregs;
    mov( linux.sys_access, eax );
    mov( pathname, ebx );
    mov( mode, ecx );
    int( $80 );
    linux.popregs;

end access;
```

3.4 acct - switch process accounting on or off

```
procedure acct( filename: string );
   returns( "eax" );
```

DESCRIPTION

When called with the name of an existing file as argument, accounting is turned on, records for each terminating process are appended to filename as it terminates. An argument of NULL causes accounting to be turned off.

RETURN VALUE

On success, zero is returned. On error, returns a negative value in EAX.

ERRORS

errno.enosys BSD process accounting has not been enabled when the operating system kernel was compiled. The kernel configuration parameter controlling this feature is CONFIG_BSD_PROCESS_ACCT.

errno.enomem Out of memory.

errno.eperm The calling process has no permission to enable process accounting.

errno.enoent The specified filename does not exist.

errno.eacces The argument filename is not a regular file.

errno.eio Error writing to the file filename.

errno.eusers There are no more free file structures or we run out of memory.

CONFORMING TO

SVr4 (but not POSIX). SVr4 documents EACCES, EBUSY, EFAULT, ELOOP, ENAMETOOLONG, ENOTDIR, ENOENT, EPERM and EROFS error conditions, but no ENOSYS.

NOTES

No accounting is produced for programs running when a crash occurs. In particular, nonterminating processes are never accounted for.

3.5 adjtimex - tune kernel clock

```
procedure adjtimex( var buf:timex );
    returns( "eax" );
```

DESCRIPTION

Linux uses David L. Mills' clock adjustment algorithm (see RFC 1305). The system call adjtimex reads and optionally sets adjustment parameters for this algorithm. It takes a pointer to a timex structure, updates kernel parameters from field values, and returns the same structure with current kernel values. This structure is declared as follows:

```
struct timex {
    int modes;           /* mode selector */
    long offset;         /* time offset (usec) */
    long freq;           /* frequency offset (scaled ppm) */
    long maxerror;       /* maximum error (usec) */
    long esterror;       /* estimated error (usec) */
    int status;          /* clock command/status */
    long constant;       /* pll time constant */
    long precision;      /* clock precision (usec) (read only) */
    long tolerance;      /* clock frequency tolerance (ppm)
                            (read only) */
    struct timeval time; /* current time (read only) */
    long tick;           /* usecs between clock ticks */
};
```

The modes field determines which parameters, if any, toset. It may contain a bitwise-or combination of zero ormore of the following bits:

```
linux.ADJ_OFFSET            $0001 /* time offset */
linux.ADJ_FREQUENCY         $0002 /* frequency offset */
linux.ADJ_MAXERROR          $0004 /* maximum time error */
linux.ADJ_ESTERROR          $0008 /* estimated time error */
linux.ADJ_STATUS            $0010 /* clock status */
linux.ADJ_TIMECONST         $0020 /* pll time constant */
linux.ADJ_TICK              $4000 /* tick value */
linux.ADJ_OFFSET_SINGLESHOT $8001 /* old-fashioned adjtime */
```

Ordinary users are restricted to a zero value for mode.

Only the superuser may set any parameters.

RETURN VALUE

On success, adjtimex returns the clock state:

```
linux.TIME_OK   0 /* clock synchronized */
linux.TIME_INS  1 /* insert leap second */
linux.TIME_DEL  2 /* delete leap second */
linux.TIME_OOP  3 /* leap second in progress */
linux.TIME_WAIT 4 /* leap second has occurred */
linux.TIME_BAD  5 /* clock not synchronized */
```

On failure, adjtimex returns -1 and sets errno.

ERRORS

errno.efault buf does not point to writable memory.

errno.eperm buf.mode is non-zero and the user is not super-user.

errno.einval An attempt is made to set buf.offset to a value outside the range -131071 to +131071, or to set buf.status to a value other than those listed above, or to set buf.tick to a value outside the range 900000/HZ to 1100000/HZ, where HZ is the system timer interrupt frequency.

CONFORMING TO

adjtimex is Linux specific and should not be used in programs intended to be portable. There is a similar but less general call adjtime in SVr4.

SEE ALSO

settimeofday(2)

3.6 alarm

```
// alarm- generates a signal at the specified time.

procedure linux.alarm( seconds:uns32 ); @nodisplay;
begin alarm;

    linux.pushregs;
    mov( linux.sys_alarm, eax );
    mov( seconds, ebx );
    int( $80 );
    linux.popregs;

end alarm;
```

DESCRIPTION

alarm arranges for a SIGALRM signal to be delivered to the process in seconds seconds.

If seconds is zero, no new alarm is scheduled.

In any event any previously set alarm is cancelled.

RETURN VALUE

alarm returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

NOTES

alarm and setitimer share the same timer; calls to one will interfere with use of the other.

sleep() may be implemented using SIGALRM; mixing calls to alarm() and sleep() is a bad idea.

Scheduling delays can, as ever, cause the execution of the process to be delayed by an arbitrary amount of time.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

SEE ALSO

setitimer(2), signal(2), sigaction(2), gettimeofday(2),
select(2), pause(2), sleep(3)

3.7 bdflush

```
// bdflush - Tunes the buffer dirty flush daemon.

procedure linux.bdflush( func:dword; address:dword ); @nodisplay;
begin bdflush;

    linux.pushregs;
    mov( linux.sys_bdflush, eax );
    mov( func, ebx );
    mov( address, ecx );
    int( $80 );
    linux.popregs;

end bdflush;
```

DESCRIPTION

bdflush starts, flushes, or tunes the buffer-dirty-flush daemon. Only the super-user may call bdflush.

If func is negative or 0, and no daemon has been started, then bdflush enters the daemon code and never returns.

If func is 1, some dirty buffers are written to disk.

If func is 2 or more and is even (low bit is 0), then address is the address of a long word, and the tuning parameter numbered (func-2)/2 is returned to the caller in that address.

If func is 3 or more and is odd (low bit is 1), then data is a long word, and the kernel sets tuning parameter num bered (func-3)/2 to that value.

The set of parameters, their values, and their legal ranges are defined in the kernel source file fs/buffer.c.

RETURN VALUE

If func is negative or 0 and the daemon successfully starts, bdflush never returns. Otherwise, the return value is 0 on success and -1 on failure, with errno set to indicate the error.

ERRORS

errno.eperm Caller is not super-user.

errno.efault address points outside your accessible address space.

errno.ebusy An attempt was made to enter the daemon code after another process has already entered.

errno.einval An attempt was made to read or write an invalid parameter number, or to write an invalid value to a parameter.

CONFORMING TO

bdflush is Linux specific and should not be used in programs intended to be portable.

SEE ALSO

fsync(2), sync(2), update(8), sync(8)

3.8 brk

```
// times - retrieves execution times for the current process.

procedure linux.brk( end_data_segment:dword ); @nodisplay;
begin brk;

    linux.pushregs;
    mov( linux.sys_brk, eax );
    mov( end_data_segment, ebx );
    int( $80 );
    linux.popregs;

end brk;
```

DESCRIPTION

brk sets the end of the data segment to the value specified by end_data_segment, when that value is reasonable, the system does have enough memory and the process does not exceed its max data size (see setrlimit(2)).

RETURN VALUE

On success, brk returns zero, and sbrk returns a pointer to the start of the new area. On error, -1 is returned, and errno is set to ENOMEM.

CONFORMING TO

BSD 4.3

SEE ALSO

execve(2), getrlimit(2), malloc(3)

3.9 chdir, fchdir

```
// chdir - Change the working directory.

procedure linux.chdir( filename:string ); @nodisplay;
begin chdir;

    linux.pushregs;
    mov( linux.sys_chdir, eax );
    mov( filename, ebx );
    int( $80 );
    linux.popregs;

end chdir;

procedure linux.fchdir( fd:dword ); @nodisplay;
begin fchdir;

    linux.pushregs;
    mov( linux.sys_fchdir, eax );
    mov( fd, ebx );
    int( $80 );
    linux.popregs;

end fchdir;
```

DESCRIPTION

chdir changes the current directory to that specified in path.

fchdir is identical to chdir, only that the directory is given as an open file descriptor.

RETURN VALUE

On success, zero is returned. On error, EAX contains the error number.

ERRORS

Depending on the file system, other errors can be returned. The more general errors for chdir are listed below:

errno.efault	path points outside your accessible address space.
errno.enametoolong	path is too long.
errno.enoent	The file does not exist.
errno.enomem	Insufficient kernel memory was available.
errno.enotdir	A component of path is not a directory.
errno.eacces	Search permission is denied on a component of path.
errno.eloop	Too many symbolic links were encountered in resolving path.
errno.eio	An I/O error occurred.

The general errors for fchdir are listed below:

errno.ebadf	fd is not a valid file descriptor.

errno.eacces	Search permission was denied on the directory open on fd.

CONFORMING TO

The chdir call is compatible with SVr4, SVID, POSIX, X/OPEN, 4.4BSD.SVr4 documents additional EINTR, ENOLINK, and EMULTIHOP error conditions but has no ENOMEM. POSIX.1 does not have ENOMEM or ELOOP error conditions. X/OPEN does not have EFAULT, ENOMEM or EIO error conditions.

The fchdir call is compatible with SVr4, 4.4BSD and X/OPEN. SVr4 documents additional EIO, EINTR, and ENOLINK error conditions. X/OPEN documents additional EINTR and EIO error conditions.

SEE ALSO

getcwd(3), chroot(2)

3.10 chmod, fchmod

```
procedure linux.chmod( filename:string; mode:linux.mode_t ); @nodisplay;
begin chmod;

    linux.pushregs;
    mov( linux.sys_chmod, eax );
    mov( filename, ebx );
    mov( mode, ecx );
    int( $80 );
    linux.popregs;

end chmod;

// fchmod: changes the permissions on a file.

procedure linux.fchmod( fd:dword; mode:linux.mode_t ); @nodisplay;
begin fchmod;

    linux.pushregs;
    mov( linux.sys_fchmod, eax );
    mov( fd, ebx );
    mov( mode, ecx );
    int( $80 );
    linux.popregs;

end fchmod;
```

DESCRIPTION

The mode of the file given by path or referenced by fildes is changed.

Modes are specified by or'ing the following:

linux.s_isuid	set user ID on execution
linux.s_isgid	set group ID on execution
linux.s_isvtx	sticky bit

linux.s_iread	read by owner
linux.s_iwrite	write by owner
linux.s_iexec	execute/search by owner
linux.s_irgrp	read by group
linux.s_iwgrp	write by group
linux.s_ixgrp	execute/search by group
linux.s_iroth	read by others
linux.s_iwoth	write by others
linux.s_ixoth	execute/search by others

The effective UID of the process must be zero or must match the owner of the file.

If the effective UID of the process is not zero and the group of the file does not match the effective group ID of the process or one of its supplementary group IDs, the S_ISGID bit will be turned off, but this will not cause an error to be returned.

Depending on the file system, set user ID and set group ID execution bits may be turned off if a file is written. On some file systems, only the super-user can set the sticky bit, which may have a special meaning. For the sticky bit, and for set user ID and set group ID bits on directories, see stat(2).

On NFS file systems, restricting the permissions will immediately influence already open files, because the access control is done on the server, but open files are maintained by the client. Widening the permissions may be delayed for other clients if attribute caching is enabled on them.

RETURN VALUE

On success, zero is returned. On error, this function returns the error code in EAX.

ERRORS

Depending on the file system, other errors can be returned. The more general errors for chmod are listed below:

errno.eperm The effective UID does not match the owner of the file, and is not zero.

errno.erofs The named file resides on a read-only file system.

errno.efault path points outside your accessible address space.

errno.enametoolong path is too long.

errno.enoent The file does not exist.

errno.enomem Insufficient kernel memory was available.

errno.enotdir A component of the path prefix is not a directory.

errno.eacces Search permission is denied on a component of the path prefix.

errno.eloop Too many symbolic links were encountered in resolving path.

errno.eio An I/O error occurred.

The general errors for fchmod are listed below:

errno.ebadf	The file descriptor fildes is not valid.
errno.erofs	See above.
errno.eperm	See above.
errno.eio	See above.

CONFORMING TO

The chmod call conforms to SVr4, SVID, POSIX, X/OPEN, 4.4BSD. SVr4 documents EINTR, ENOLINK and EMULTIHOP returns, but no ENOMEM. POSIX.1 does not document EFAULT, ENOMEM, ELOOP or EIO error conditions, or the macros S_IREAD, S_IWRITE and S_IEXEC.

The fchmod call conforms to 4.4BSD and SVr4. SVr4 documents additional EINTR and ENOLINK error conditions. POSIX requires the fchmod function if at least one of _POSIX_MAPPED_FILES and _POSIX_SHARED_MEMORY_OBJECTS is defined, and documents additional ENOSYS and EINVAL error conditions, but does not document EIO.

POSIX and X/OPEN do not document the sticky bit.

SEE ALSO

open(2), chown(2), execve(2), stat(2)

3.11 chown, fchown, lchown

```
// chown - Change the ownership of a file.

procedure linux.chown( path:string; owner:linux.uid_t; group:linux.gid_t );
    @nodisplay;
begin chown;

    linux.pushregs;
    mov( linux.sys_chown, eax );
    mov( path, ebx );
    movzx( owner, ecx );
    movzx( group, edx );
    int( $80 );
    linux.popregs;

end chown;

// fchown: changes the owner of a file.

procedure linux.fchown( fd:dword; owner:linux.uid_t; group:linux.gid_t );
    @nodisplay;
begin fchown;

    linux.pushregs;
    mov( linux.sys_fchown, eax );
    mov( fd, ebx );
    movzx( owner, ecx );
    movzx( group, edx );
    int( $80 );
    linux.popregs;

end fchown;

// chmod - Changes the file permissions.

procedure linux.lchown
(
    filename:string;
    user       :linux.uid_t;
    group      :linux.gid_t
);
    @nodisplay;
begin lchown;

    linux.pushregs;
    mov( linux.sys_lchown, eax );
    mov( filename, ebx );
    movzx( user, ecx );
    movzx( group, edx );
    int( $80 );
    linux.popregs;

end lchown;
```

DESCRIPTION

The owner of the file specified by path or by fd is changed. Only the super-user may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member. The super-user may change the group arbitrarily.

If the owner or group is specified as -1, then that ID is not changed.

When the owner or group of an executable file are changed by a non-super-user, the S_ISUID and S_ISGID mode bits are cleared. POSIX does not specify whether this also should happen when root does the chown; the Linux behaviour depends on the kernel version. In case of a non-group- executable file (with clear S_IXGRP bit) the S_ISGID bit indicates mandatory locking, and is not cleared by a chown.

RETURN VALUE

On success, zero is returned. On error, EAX contains the error code.

ERRORS

Depending on the file system, other errors can be returned. Themore general errors for chown are listed

below:

errno.eperm The effective UID does not match the owner of the file, and is not zero; or the owner or group were specified incorrectly.

errno.erofs The named file resides on a read-only file system.

errno.efault path points outside your accessible address space.

errno.enametoolong path is too long.

errno.enoent The file does not exist.

errno.enomem Insufficient kernel memory was available.

errno.enotdir A component of the path prefix is not a directory.

errno.eacces Search permission is denied on a component of the path prefix.

errno.eloop Too many symbolic links were encountered in resolving path.

The general errors for fchown are listed below:

errno.ebadf The descriptor is not valid.

errno.enoent See above.

errno.eperm See above.

errno.erofs See above.

errno.eio A low-level I/O error occurred while modifying the inode.

NOTES

In versions of Linux prior to 2.1.81 (and distinct from 2.1.46), chown did not follow symbolic links. Since Linux 2.1.81, chown does follow symbolic links, and there is a new system call lchown that does not follow symbolic links. Since Linux 2.1.86, this new call (that has the same semantics as the old chown) has got the same syscall number, and chown got the newly introduced number.

CONFORMING TO

The chown call conforms to SVr4, SVID, POSIX, X/OPEN. The 4.4BSD version can only be used by the superuser (that is, ordinary users cannot give away files). SVr4 documents EINVAL, EINTR, ENOLINK and EMULTIHOP returns, but no ENOMEM. POSIX.1 does not document ENOMEM or ELOOP error conditions.

The fchown call conforms to 4.4BSD and SVr4. SVr4 documents additional EINVAL, EIO, EINTR, and ENOLINK error conditions.

RESTRICTIONS

The chown() semantics are deliberatelyviolated on NFS file systems which have UID mapping enabled. Additionally, the semantics of all system calls which access the file contents are violated, because chown() may cause immediate access revocation on already open files. Client side caching may lead to a delay between the time where ownership have been changed to allow access for a user and the time where the file can actually be accessed by the user on other clients.

SEE ALSO

chmod(2), flock(2)

3.12 chroot

```
// chroot - changes the root directory ("/").

procedure linux.chroot( path:string ); @nodisplay;
begin chroot;

    linux.pushregs;
    mov( linux.sys_chroot, eax );
    mov( path, ebx );
    int( $80 );
    linux.popregs;

end chroot;
```

DESCRIPTION

chroot changes the root directory to that specified in path. This directory will be used for path names beginning with /. The root directory is inherited by all children of the current process.

Only the super-user may change the root directory.

Note that this call does not change the current working directory, so that `.' can be outside the tree rooted at `/'. In particular, the super-user can escape from a `chroot jail' by doing `mkdir foo; chroot foo; cd ..'.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

Depending on the file system, other errors can be returned. The more general errors are listed below:

errno.eperm The effective UID is not zero.

errno.efault path points outside your accessible address space.

errno.enametoolong path is too long.

errno.enoent The file does not exist.

errno.enomem Insufficient kernel memory was available.

errno.enotdir A component of path is not a directory.

errno.eacces Search permission is denied on a component of the path prefix.

errno.eloop Too many symbolic links were encountered in resolving path.

errno.eio An I/O error occurred.

CONFORMING TO

SVr4, SVID, 4.4BSD, X/OPEN. This function is not part of POSIX.1. SVr4 documents additional EINTR, ENOLINK and EMULTIHOP error conditions. X/OPEN does not document EIO, ENOMEM or EFAULT error conditions. This interface is marked as legacy by X/OPEN.

SEE ALSO

chdir(2)

3.13 clone

```
// clone: Starts a thread.
//
//  This one can't be a trivial wrapper function because
// creating a new thread also creates a new stack for the
// child thread.  Therefore, things like return addresses
// and parameters only belong to the parent thread.  Since
// this procedure gets called by the parent, we have to
// pull some tricks to make sure we can still return to
// the caller from both the parent and the child thread.
//
// sysclone- Simple wrapper that simulates the standard
// Linux int( $80 ) invocation.

procedure linux.sysclone( var child_stack:var; flags:dword );
    @nodisplay;
    @noframe;

begin sysclone;

    push( ebx );
    push( ecx );

    // Okay, we've got to copy the EBX, ECX, and
    // return address values to the child's stack.
    // Note that the stack looks like this:
    //
    // Parent:
    //
    //   child_stack+16
    //   flags    +12
    //   return   +8
    //   ebx          +4
    //   ecx          +0 (esp)
    //

    mov( [esp+16], eax );// Get ptr to child's stack.
    sub( 20, eax );   // Make it look like parent's.
    mov( ebx, [eax+4] );// Simulate the pushes
    mov( ecx, [eax] );
    mov( [esp+8], ebx );// Copy the return address.
    mov( ebx, [eax+8] );

    mov( eax, ecx );// sys_clone expects ESP in ECX.
    mov( [esp+12], ebx );// Get flags parameter into EBX.
    mov( linux.sys_clone, eax );
    int( $80 );

    pop( ecx );           // Cleans up the stack for
    pop( ebx );           //  both parent and child after
    ret( 8 );             //  all the work above.

end sysclone;


// linux.clone-
//
//  Fancy version of clone that lets you specify a thread starting
// address and pass a parameter to that function.
```

```
procedure linux.clone
(
         fn          :linux.clonefn_t;
    var  child_stack:var;
         flags    :dword;
         arg         :dword
);
    @nodisplay;
    @noalignstack;

begin clone;

    begin InvalidArgument;

         xor( eax, eax );
         cmp( eax, fn );
         exitif( @e ) InvalidArgument;
         cmp( eax, child_stack );
         exitif( @e ) InvalidArgument;

         // Set up the new stack:

         mov( child_stack, eax );
         sub( 16, eax );// Create room on new stack for rtnadrs & arg

         // Copy the argument to the new stack:

         push( arg );
         pop( (type dword [eax+12]) );

         // Copy the start address to the child stack as a temporary
         // measure while we switch stacks:

         push( (type dword fn ) );
         pop( (type dword [eax+8]) );

         // Save EBX and ECX on both stacks

         push( ebx );
         push( ecx );
         mov( ebx, [eax+4] );
         mov( ecx, [eax] );

         // Do the sysclone system call:

         mov( flags, ebx );
         mov( eax, ecx );
         mov( linux.sys_clone, eax );
         int( $80 );

         // Retrieve EBX and ECX from the stack (which stack depends upon
         // the particular return from sys_clone).

         pop( ecx );
         pop( ebx );

         test( eax, eax );
         exitif( @s || @nz ) clone;// Exit if parent or an error occurs.

         // Invoke the thread here
```

```
        xor( ebp, ebp );// ebp=0 marks end of stack frame.
        pop( eax );         // Get fn address (pushed on stack earlier)
        call( eax );    // Call the thread's code.
        linux._exit( eax );// Terminate thread.

    end InvalidArgument;
    mov( errno.einval, eax );

end clone;
```

DESCRIPTION

clone creates a new process, just like *fork*(2). clone is a library function layered on top of the underlying *clone* system call, hereinafter referred to as *sysclone*. A description of *sysclone* is given towards the end of this page.

Unlike *fork*(2), these calls allow the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers. (Note that on this manual page, "calling process" normally corresponds to "parent process". But see the description of linux.clone_parent below.)

The main use of *clone* is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.

When the child process is created with *clone*, it executes the function application *fn*(arg). (This differs from *fork*(2), where execution continues in the child from the point of the *fork*(2) call.) The *fn* argument is a pointer to a function that is called by the child process at the beginning of its execution. The *arg* argument is passed to the *fn* function. Note that the *fn* procedure must have the "@NODISPLAY" procedure option on the program will crash when *fn* attempts to build a display. This occurs because the clone code (see above) sets EBP to zero prior to calling *fn* and *fn*, if it builds a display, need to reference objects pointed at by EBP (hence the crash). In practice, the procedure you use as a thread should not be nested inside another procedure. In theory, it is possible to use nested procedures and even access intermediate variables in other procedures; however, keep in mind that the code cannot build a display, so if you need to access intermediate variables, you will need to pass in a pointer to the activation record yourself and build the display manually (you could, for example, use the single argument to *fn* to pass in the address of the caller’s activation record).

When the *fn(arg)* function application returns, the child process terminates. The integer returned by *fn* is the exit code for the child process. The child process may also terminate explicitely by calling *_exit*(2) or after receiving a fatal signal.

The *child_stack* argument specifies the location of the stack used by the child process. Since the child and calling process may share memory, it is not possible for the child process to execute in the same stack as the calling process. The calling process must therefore set up memory space for the child stack and pass a pointer to this space to *clone*. Stacks grow downwards on all processors that run Linux (except the HP PA processors), so *child_stack* usually points to the topmost address of the memory space set up for the child stack.

The low byte of *flags* contains the number of the signal sent to the parent when the child dies. If this signal is specified as anything other than signals.sigchld, then the parent process must specify the __WALL or __WCLONE options when waiting for the child with *wait*(2). If no signal is specified, then the parent process is not signaled when the child terminates.

flags may also be bitwise-or'ed with one or several of the following constants, in order to specify what is shared between the calling process and the child process:

linux.clone_parent

(Linux 2.4 onwards) If **clone_parent** is set, then the parent of the new child (as returned by getppid(2)) will be the same as that of the calling process.

If **clone_parent** is not set, then (as with *fork*(2)) the child's parent is the calling process. Note that it is the parent process, as returned by getppid(2), which is signaled when the child terminates, so that if **clone_parent** is set, then the parent of the calling process, rather than the calling process itself, will be signaled.

linux.clone_fs

If **clone_fs** is set, the caller and the child processes share the same file system information. This includes the root of the file system, the current working directory, and the umask. Any call to chroot(2), chdir(2), or umask(2) performed by the callng process or the child process also takes effect in the other process.

If **clone_fs** is not set, the child process works on a copy of the file system information of the calling process at the time of the clone call. Calls to chroot(2), chdir(2), umask(2) performed later by one of the processes do not affect the other process.

linux.clone_files

If **clone_files** is set, the calling process and the child processes share the same file descriptor table. File descriptors always refer to the same files in the calling process and in the child process. Any file descriptor created by the calling process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags, the other process is also affected.

If **clone_files** is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of clone. Operations on file descriptors performed later by either the calling process or the child process do not affect the other process.

linux.clone_sighand

If **clone_sighand** is set, the calling process and the child processes share the same table of signal handlers. If the calling process or child process calls sigaction(2) to change the behavior associated with a signal, the behavior is changed in the other process as well. However, the calling process and child processes still have distinct signal masks and sets of pending signals. So, one of them may block or unblock some signals using sigprocmask(2) without affecting the other process.

If **clone_sighand** is not set, the child process inherits a copy of the signal handlers of the calling process at the time clone is called. Calls to sigaction(2) performed later by one of the processes have no effect on the other process.

linux.clone_ptrace

If **clone_ptrace** is specified, and the calling process is being traced, then trace the child will also be traced (see ptrace(2)).

linux.clone_vfork

If **clone_vfork** is set, the execution of the calling process is suspended until the child releases its virtual memory resources via a call to execve(2) or _exit(2) (as with vfork(2)).

If **clone_vfork** is not set, then both the calling process and the child are schedulable after the call, and an application should not rely on execution occurring in any particular order.

linux.clone_vm

If **clone_vm** is set, the calling process and the child processes run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with mmap(2) or munmap(2) by the child or calling process also affects the other process.

If **clone_vm** is not set, the child process runs in a separate copy of the memory space of the calling process at the time of clone. Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with fork(2).

linux.clone_pid

If **clone_pid** is set, the child process is created with the same process ID as the calling process.

If **clone_pid** is not set, the child process possesses a unique process ID, distinct from that of the calling process.

This flag can only be specified by the system boot process (PID 0).

linux.clone_thread

(Linux 2.4 onwards) If **clone_thread** is set, the child is placed in the same thread group as the calling process.

If **clone_thread** is not set, then the child is placed in its own (new) thread group, whose ID is the same as the process ID.

(Thread groups are feature added in Linux 2.4 to support the POSIX threads notion of a set of threads sharing a single PID. In Linux 2.4, calls to getpid(2) return the thread group ID of the caller.)

SYSCLONE

The sysclone system call corresponds more closely to fork(2) in that execution in the child continues from the point of the call. Thus, sysclone only requires the flags and child_stack arguments, which have the same meaning as for clone. (Note that the order of these arguments differs from clone.)

Another difference for sysclone is that the child_stack argument may be zero, in which case copy-on-write semantics ensure that the child gets separate copies of stack pages when either process modifies the stack. In this case, for correct operation, the **clone_vm** option should not be specified.

RETURN VALUE

On success, the PID of the child process is returned in the caller's thread of execution. On these functions return a negative valued error code in EAX.

Because of the way clone works, you cannot assume register preservation from the parent process to the child process (in particular, EBX and ECX will get trashed when cloning a child thread).

ERRORS

errno.eagain	Too many processes are already running.
errno.enomem	Cannot allocate sufficient memory to allocate a task structure for the child, or to copy those parts of the caller's context that need to be copied.
errno.einval	Returned by clone when a zero value is specified for child_stack.
errno.eperm	CLONE_PID was specified by a process with a non-zero PID.

BUGS

As of version 2.1.97 of the kernel, the **clone_pid** flag should not be used, since other parts of the kernel and most system software still assume that process IDs are unique.

CONFORMING TO

The clone and sys_clone calls are Linux-specific and should not be used in programs intended to be portable. For programming threaded applications (multiple threads of control in the same memory space), it is better to use a library implementing the POSIX 1003.1c thread API, such as the LinuxThreads library (included in glibc2). See pthread_create(3thr).

SEE ALSO

fork(2), wait(2), pthread_create(3thr)

3.14 close

```
// close - closes a file.

procedure linux.close( fd:dword ); @nodisplay;
begin close;

    linux.pushregs;
    mov( linux.sys_close, eax );
    mov( fd, ebx );
    int( $80 );
    linux.popregs;

end close;
```

DESCRIPTION

close closes a file descriptor, so that it no longer refers to any file and may be reused. Any locks held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If fd is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using unlink(2) the file is deleted.

RETURN VALUES

close returns zero on success, a negative value in EAX if an error occurs.

ERRORS

errno.ebad fd isn't a valid open file descriptor.

errno.eintr The close() call was interrupted by a signal.

errno.eio An I/O error occurred.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. SVr4 documents an
additional ENOLINK error condition.

NOTES

Not checking the return value of close is a common but nevertheless serious programming error. File system implementations which use techniques as ``write-behind'' to increase performance may lead to write(2) succeeding, although the data has not been written yet. The error status may be reported at a later write operation, but it is guaranteed to be reported on closing the file. Not checking the return value when closing the file may lead to silent loss of data. This can especially be observed with NFS and disk quotas.

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel defers writes. It is not common for a filesystem to flush the buffers when the stream is closed. If you need to be sure that the data is physically stored use fsync(2) or sync(2), they will get you closer to that goal (it will depend on the disk hardware at this point).

SEE ALSO

open(2), fcntl(2), shutdown(2), unlink(2), fclose(3)

3.15 creat, open

```
// creat - Create a new file.

procedure linux.creat( pathname:string; mode:linux.mode_t );
    @nodisplay;
begin creat;

    linux.pushregs;
    mov( linux.sys_creat, eax );
    mov( pathname, ebx );
    mov( mode, ecx );
    int( $80 );
    linux.popregs;

end creat;

procedure linux.open( filename:string; flags:dword; mode:linux.mode_t );
    @nodisplay;
begin open;

    linux.pushregs;
    mov( linux.sys_open, eax );
    mov( filename, ebx );
    mov( flags, ecx );
    mov( mode, edx );
    int( $80 );
    linux.popregs;

end open;
```

DESCRIPTION

The open() system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This call creates a new open file, not shared with any other process. (But shared open files may arise via the fork(2) system call.) The new file descriptor is set to remain open across exec functions (see fcntl(2)). The file offset is set to the beginning of the file.

The parameter flags is one of **linux.o_rdonly**, **linux.o_wronly** or **linux.o_rdwr** which request opening the file read-only, write-only or read/write, respectively, bitwise-or'd with zero or more of the following:

linux. o_creat If the file does not exist it will be created. The owner (user ID) of the file is set to the effective user ID of the process. The group ownership (group ID) is set either to the effective group ID of the process or to the group ID of the parent directory (depending on filesystem type and mount options, and the mode of the parent directory, see, e.g., the mount options bsdgroups and sysvgroups of the ext2 filesystem, as described in mount(8)).

linux.o_excl When used with O_CREAT, if the file already exists it is an error and the open will fail. In this context, a symbolic link exists, regardless of where its points to. O_EXCL is broken on NFS file systems, programs which rely on it for performing locking tasks will contain a race condition. The solution for performing atomic file locking using a lockfile is to create a unique file on the same fs (e.g., incorporating hostname and pid), use link(2) to make a link to the lockfile. If link() returns 0, the lock is successful. Otherwise, use stat(2) on the unique file to check if its link count has increased to 2, in which case the lock is also successful.

linux.o_noctty If pathname refers to a terminal device -- see tty(4) -- it will not become the process's controlling terminal even if the process does not have one.

linux.o_trunc If the file already exists and is a regular file and the open mode allows writing (i.e., is **linux.o_rdwr** or **linux.o_wronly**) it will be truncated to length 0. If the file is a FIFO or terminal device file, the **linux.o_trunc** flag is ignored. Otherwise the effect of **linux.o_trunc** is unspecified. (On many Linux versions it will be ignored; on other versions it will return an error.)

linux.o_append The file is opened in append mode. Before each write, the file pointer is positioned at the end of the file, as if with lseek. O_APPEND may lead to corrupted fileson NFS file systems if more than one process appends data to a file at once. This is because NFS does not support appending to a file, so the client kernel has to simulate it, which can't be done without a race condition.

linux.o_nonblock,

linux.o_ndelay When possible, the file is opened in non-blocking mode. Neither the open nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait. For the handling of FIFOs (named pipes), see also fifo(4). This mode need not have any effect on files other than FIFOs.

linux.o_sync The file is opened for synchronous I/O. Any writes on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware. See RESTRICTIONS below, though.

linux.o_nofollow If pathname is a symbolic link, then the open fails. This is a FreeBSD extension, which was added to Linux in version 2.1.126. Symbolic links in earlier components of the pathname will still be followed. The headers from glibc 2.0.100 and later include a definition of this flag; kernels before 2.1.126 will ignore it if used.

linux.o_directory If pathname is not a directory, cause the open to fail. This flag is Linux-specific, and was added in kernel version 2.1.126, to avoid denial-of-service problems if opendir(3) is called on a FIFO or tape device, but should not be used outside of the implementation of opendir.

linux.o_largefile On 32-bit systems that support the Large Files System, allow files whose sizes cannot be represented in 31 bits to be opened.

Some of these optional flags can be altered using fcntl after the file has been opened.

The argument *mode* specifies the permissions to use in case a new file is created. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask). Note that this mode only applies to future accesses of the newly created file; the open call that creates a read-only file may well return a read/write file descriptor.

The following symbolic constants are provided for mode:

linux.s_irwxu	user (file owner) has read, write and execute permission
linux.s_iread	user has read permission
linux.s_iwrite	user has write permission
linux.s_iexec	user has execute permission
linux.s_irwxg	group has read, write and execute permission
linux.s_irgrp	group has read permission
linux.s_iwgrp	group has write permission
linux.s_ixgrp	group has execute permission
linux.s_irwxo	others have read, write and execute permission
linux.s_iroth	others have read permission
linux.s_iwoth	others have write permisson
linux.s_ixoth	others have execute permission

mode should always be specified when **linux.o_creat** is in the flags, and is ignored otherwise.

creat is equivalent to *open* with flags equal to **linux.o_creat | linux.o_wronly | linux.o_trunc**.

RETURN VALUE

open and *creat* return the new file descriptor, or a negative error code if an error occurs. Note that *open* can open device special files, but *creat* cannot create them - use mknod(2) instead.

On NFS file systems with UID mapping enabled, open may return a file descriptor but e.g. read(2) requests are denied with **linux.eacces**. This is because the client performs open by checking the permissions, but UID mapping is performed by the server upon read and write requests.

If the file is newly created, its atime, ctime, mtime fields are set to the current time, and so are the ctime and mtime fields of the parent directory. Otherwise, if the file is modified because of the **linux.o_trunc** flag, its ctime and mtime fields are set to the current time.

ERRORS

errno.eexist pathname already exists and O_CREAT and O_EXCL were used.

errno.eisdir pathnamerefers to a directory and the access
requested involved writing.

errno.eacces The requested access to the file is not allowed, or
one of the directories in pathname did not allow
search (execute) permission, or the file did not
exist yet and write access to the parent directory
is not allowed.

errno.enametoolong pathname was too long.

errno.enoent A directory component in pathname does not exist or
is a dangling symbolic link.

errno.enotdir A component used as a directory in pathname is not,
in fact, a directory, or O_DIRECTORY wasspecified
and pathname was not a directory.

errno.enxio O_NONBLOCK | O_WRONLY is set, the named file is a
FIFO and no process has the file open for reading.
Or, the file is a device special file and no corre≠
sponding device exists.

errno.enodev pathname refers to a device special file and no
corresponding device exists. (This is a Linux ker≠
nel bug- in this situationENXIOmust be
returned.)

errno.erofs pathname refers to a file on a read-only filesystem
and write access was requested.

errno.etxtbsy pathname refers to an executable imagewhich is
currently beingexecuted and write access was
requested.

errno.efault pathname points outsideyour accessible address space.

errno.eloop Too many symbolic links were encountered in resolving pathname, or O_NOFOLLOW was specified but pathname was a symbolic link.

errno.enospc pathnamewas to be created but the device containing pathname has no room for the new file.

errno.enomem Insufficient kernel memory was available.

errno.emfile The process already has the maximum number of files open.

errno.enfile The limit on the total number of files open on the system has been reached.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3 The **linux.o_nofollow** and **linux.o_directory** flags are Linux-specific. One may have to define the _GNU_SOURCE macro to get their definitions.

RESTRICTIONS

There are many infelicities in the protocol underlying NFS, affecting amongst others **linux.o_sync** and **linux.o_ndelay**.

POSIX provides for three different variants of synchronised I/O, corresponding to the flags **linux.o_sync**, **linux.o_dsync** and **linux.o_rsync**. Currently (2.1.130) these are all synonymous under Linux.

SEE ALSO

read(2), write(2), fcntl(2), close(2), link(2), mknod(2), mount(2), stat(2), umask(2), unlink(2), socket(2), fopen(3), fifo(4)

3.16 create_module

```
// create_module- Registers a device driver module.

procedure linux.create_module( theName:string; size:linux.size_t );
    @nodisplay;
begin create_module;

    linux.pushregs;

    mov( linux.sys_create_module, eax );
    mov( theName, ebx );
    mov( size, ecx );
    int( $80 );
    linux.popregs;

end create_module;
```

DESCRIPTION

create_module attempts to create a loadable module entry and reserve the kernel memory that will be needed to hold the module. This system call is only open to the superuser.

RETURN VALUE

On success, returns the kernel address at which the module will reside. On error, this call returns a negative error code in EAX.

ERRORS

errno.eperm	The user is not the superuser.
errno.eexist	A module by that name already exists.
errno.einval	The requested size is too small even for the module header information.
errno.enomem	The kernel could not allocate a contiguous block of memory large enough for the module.
errno.efault	name is outside the program's accessible address space.

SEE ALSO

init_module(2), delete_module(2), query_module(2).

3.17 delete_module

```
// delete_module- Removes a device driver module.

procedure linux.delete_module( theName:string );
    @nodisplay;
begin delete_module;

    linux.pushregs;

    mov( linux.sys_delete_module, eax );
    mov( theName, ebx );
    int( $80 );
    linux.popregs;

end delete_module;
```

DESCRIPTION

delete_module attempts to remove an unused loadable module entry. If name is NULL, all unused modules marked auto- clean will be removed. This system call is only open to the superuser.

RETURN VALUE

On success, zero is returned. On EAX returns a negative error code.

ERRORS

errno.eperm The user is not the superuser.

errno.enoent No module by that name exists.

errno.einval name was the empty string.

errno.ebusy The module is in use.

errno.efault name is outside the program's accessible address space.

SEE ALSO

create_module(2), init_module(2), query_module(2).

3.18 dupfd (dup), dup2

```
// dupfd - duplicates a file descriptor.

procedure linux.dupfd( oldfd:dword );
    @nodisplay;
begin dupfd;

    linux.pushregs;
    mov( linux.sys_dup, eax );
    mov( oldfd, ebx );
    int( $80 );
    linux.popregs;

end dupfd;

procedure linux.dup2( oldfd:dword; newfd:dword );
    @nodisplay;
begin dup2;

    linux.pushregs;
    mov( linux.sys_dup2, eax );
    mov( oldfd, ebx );
    mov( newfd, ecx );
    int( $80 );
    linux.popregs;

end dup2;
```

Note: HLA uses *dupfd* rather than the standard Linux name "dup" because "dup" is an HLA reserved word.

DESCRIPTION

dupfd and *dup2* create a copy of the file descriptor *oldfd*.

After successful return of *dupfd* or *dup2*, the old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using *lseek* on one of the descriptors, the position is also changed for the other.

The two descriptors do not share the close-on-exec flag, however.

dupfd uses the lowest-numbered unused descriptor for the new descriptor.

dup2 makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary.

RETURN VALUE

dupfd and *dup2* return the new descriptor, or a negative error code in EAX.

ERRORS

errno.ebadf *oldfd* isn't an open file descriptor, or *newfd* isout of the allowed range for file descriptors.

errno.emfile The process already has the maximum number of file descriptors open and tried to open a new one.

WARNING

The error returned by *dup2* is different to that returned by *fcntl*(..., F_DUPFD, ...) when *newfd* is out of range. On some systems *dup2* also sometimes returns **errno.einval** like F_DUPFD.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. SVr4 documents additional **errno.eintr** and **errno.enolink** error conditions. POSIX.1 adds **errno.eintr**.

SEE ALSO

fcntl(2), open(2), close(2)

3.19 execve

```
// execve - Execute some process.

procedure linux.execve( filename:string; var argv:var; var envp:var );
    @nodisplay;
begin execve;

    linux.pushregs;
    mov( linux.sys_execve, eax );
    mov( filename, ebx );
    mov( argv, ecx ) ;
    mov( envp, edx );
    int( $80 );
    linux.popregs;

end execve;
```

DESCRIPTION

execve executes the program pointed to by *filename*. *filename* must be either a binary executable, or a script starting with a line of the form "#! interpreter [arg]". In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as interpreter [arg] filename.

argv is an array of argument strings passed to the new program. *envp* is an array of strings, conventionally of the form *key=value*, which are passed as environment to the new program. Both, *argv* and *envp* must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as int main(int argc, char *argv[], char *envp[]).

execve does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's PID, and any open file descriptors that are not set to close on exec. Signals pending on the calling process are cleared. Any signals set to be caught by the calling process are reset to their default behaviour. The **signals.sigchld** signal (when set to **signals.sig_ign**) may or may not be reset to **signals.sig_dfl**.

If the current program is being ptraced, a **signals.sigtrap** is sent to it after a successful *execve*.

If the set-uid bit is set on the program file pointed to by filename the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, when the set-gid bit of the program file is set the effective group ID of the calling process is set to the group of the program file.

If the executable is an a.out dynamically-linked binary executable containing shared-library stubs, the Linux dynamic linker ld.so(8) is called at the start of execution to bring needed shared libraries into core and link the executable with them.

If the executable is a dynamically-linked ELF executable, the interpreter named in the PT_INTERP segment is used to load the needed shared libraries. This interpreter is typically /lib/ld-linux.so.1 for binaries linked with the Linux libc version 5, or /lib/ld-linux.so.2 for binaries linked with the GNU libc version 2.

RETURN VALUE

On success, *execve* does not return, on it returns a negative error code in EAX.

ERRORS

errno.eacces The file or a script interpreter is not a regular file.

errno.eacces Execute permission is denied for the file or a script or ELF interpreter.

errno.eacces The file system is mounted noexec.

errno.eperm The file system is mounted nosuid, the user is not the superuser, and the file has an SUID or SGID bit set.

errno.eperm The process is being traced, the user is not the superuser and the file has an SUID or SGID bit set.

errno.e2big The argument list is too big.

errno.enoexec An executable is not in a recognised format, is for the wrong architecture, or has some other format error that means it cannot be executed.

errno.efault *filename* points outside your accessible address space.

errno.enametoolong *filename* is too long.

errno.enoent The file *filename* or a script or ELF interpreter does not exist, or a shared library needed for file or interpreter cannot be found.

errno.enomem Insufficient kernel memory was available.

errno.enotdir A component of the path prefix of filename or a script or ELF interpreter is not a directory.

errno.eacces Search permission is denied on a component of the path prefix of *filename* or the name of a script interpreter.

errno.eloop Too many symbolic links were encountered in resolving *filename* or the name of a script or ELF interpreter.

errno.etxtbsy Executable was open for writing by one or more processes.

errno.eio An I/O error occurred.

errno.enfile The limit on the total number of files open on the system has been reached.

errno.emfile The process has the maximum number of files open.

errno.einval An ELF executable had more than one PT_INTERP segment (i.e., tried to name more than one interpreter).

errno.eisdir An ELF interpreter was a directory.

errno.elibbad An ELF interpreter was not in a recognized format.

CONFORMING TO

SVr4, SVID, X/OPEN, BSD 4.3. POSIX does not document the #! behavior but is otherwise compatible. SVr4 documents additional error conditions **errno.eagain, errno.eintr, errno.elibacc, errno.enolink, errno.emultihop;** POSIX does not document **errno.etxtbsy, errno.eperm, errno.efault, errno.eloop, errno.eio, errno.enfile, errno.emfile, errno.einval, errno.eisdir** or **errno.elibbad** error conditions.

NOTES

SUID and SGID processes can not be ptrace()d.

Linux ignores the SUID and SGID bits on scripts.

The result of mounting a filesystem nosuid vary between Linux kernel versions: some will refuse execution of SUID/SGID executables when this would give the user powers s/he did not have already (and return **errno.eperm**), some will just ignore the SUID/SGID bits and exec successfully.

A maximum line length of 127 characters is allowed for the first line in a #! executable shell script.

SEE ALSO

chmod(2), fork(2), execl(3), environ(5), ld.so(8)

3.20 _exit (exit)

```
// _exit - Quits the current process and returns control
// to whomever started it.  Status is the return code.
//
//  Note: actual Linux call is "exit" but this is an
// HLA reserved word, hence the use of "_exit" here.

procedure linux._exit( status:int32 ); @nodisplay; @noframe;
begin _exit;

    mov( [esp+4], ebx );         // Get the status value
    mov( linux.sys_exit, eax );  // exit opcode.
    int( $80 );                  // Does not return!

end _exit;
```

DESCRIPTION

_exit terminates the calling process immediately. Any open file descriptors belonging to the process are closed; any children of the process are inherited by process 1, init, and the process's parent is sent a **signals.sigchld** signal.

status is returned to the parent process as the process's exit status, and can be collected using one of the wait family of calls.

RETURN VALUE

_exit never returns.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

NOTES

_exit does not flush standard I/O buffers.

SEE ALSO

fork(2), execve(2), waitpid(2), wait4(2), kill(2), wait(2), exit(3)

3.21 fchdir

See chdir.

3.22 fcntl

```
// Macro that provides overloading for fcntl (two vs. three parameters):

    #macro fcntl( fd, cmd, arg[] );

         #if( @elements( arg ) = 0 )

               fcntl2( fd, cmd )

         #else

               fcntl3( fd, cmd, @text( arg[0] ))

         #endif

    #endmacro;

// Two-parameter version of fcntl:

procedure linux.fcntl2( fd:dword; cmd:dword );
    @nodisplay;
begin fcntl2;

    linux.pushregs;
    mov( linux.sys_fcntl, eax );
    mov( fd, ebx );
    mov( cmd, ecx );
    int( $80 );
    linux.popregs;

end fcntl2;

// fcntl3 - three parameter form of the fcntl function.

procedure linux.fcntl3( fd:dword; cmd:dword; arg:dword );
    @nodisplay;
begin fcntl3;

    linux.pushregs;
    mov( linux.sys_fcntl, eax );
    mov( fd, ebx );
    mov( cmd, ecx );
    mov( arg, edx );
    int( $80 );
    linux.popregs;

end fcntl3;
```

DESCRIPTION

fcntl performs one of various miscellaneous operations on *fd*. The operation in question is determined by *cmd*:

linux.f_dupfd Find the lowest numbered available file descriptor greater than or equal to *arg* and make it be a copy of *fd*. This is different form *dup2*(2) which uses exactly the descriptor specified.

The old and new descriptors may be used interchangeably. They share locks, file position pointers and flags; for example, if the file position is modified by using *lseek* on one of the descriptors, the position is also changed for the other.

The two descriptors do not share the close-on-exec flag, however.The close-on-exec flag of the copy is off, meaning that it will not be closed on exec.

On success, the new descriptor is returned.

linux.f_getfd

Read the close-on-exec flag. If the **linux.fd_cloexec** bit is 0, the file will remain open across exec, otherwise it will be closed.

linux.f_setfd

Set the close-on-exec flag to the value specified by the **linux.fd_cloexec** bit of *arg*.

linux.f_getfl

Read the descriptor's flags (all flags (as set by open(2)) are returned).

linux.f_setfl

Set the descriptor's flags to the value specified by *arg*. Only **linux.o_append**, **linux.o_nonblock** and **linux.o_async** may be set; the other flags are unaffected.

The flags are shared between copies (made with *dup*(2), *fork*(2), etc.) of the same file descriptor.

The flags and their semantics are described in *open*(2).

linux.f_getlk,

linux.f_setlk and

linux.f_setlkw

are used to manage discretionary file locks. The third argument lock is a pointer to a struct *flock_t* (that may be overwritten by this call).

linux.f_getlk

Return the *flock_t* structure that prevents us from obtaining the lock, or set the *l_type* field of the lock to **linux.f_unlck** if there is no obstruction.

linux.f_setlk

The lock is set (when l_type is **linux.f_rdlck** or **linux.f_wrlck**) or cleared (when it is **linux.f_unlck**). If the lock is held by someone else, this call returns **errno.eacces** or **errno.eagain** in EAX.

linux.f_setlkw

Like **linux.f_setlk**, but instead of returning an error we wait for the lock to be released. If a signal that is to be caught is received while *fcntl* is waiting, it is interrupted and (after the signal handler has returned) returns immediately (with return **errno.eintr**).

linux.f_getown,

linux.f_setown,

linux.f_getsig and

linux.f_setsig

are used to manage I/O availability signals:

linux.f_getown

Get the process ID or process group currently receiving **signals.sigio** and **signals.sigurg** signals for events on file descriptor *fd*. Process groups are returned as negative values.

linux.f_setown Set the process ID or process group that will receive **signals.sigio** and **signals.sigurg** signals for events on file descriptor *fd*. Process groups are specified using negative values. (**linux.f_setsig** can be used to specify a different signal instead of **signals.sigio**).

If you set the **linux.o_async** status flag on a file descriptor (either by providing this flag with the *open*(2) call, or by using the **linux.f_setfl** command of *fcntl*), a **signals.sigio** signal is sent whenever input or output becomes possible on that file descriptor.

The process or process group to receive the signal can be selected by using the **linux.f_setown** command to the *fcntl* function. If the file descriptor is a socket, this also selects the recipient of **signals.sigurg** signals that are delivered when out-of-band data arrives on that socket. (**signals.sigurg** is sent in any situation where *select*(2) would report the socket as having an "exceptional condition".) If the file descriptor corresponds to a terminal device, then **signals.sigio** signals are sent to the foreground process group of the terminal.

linux.F_GETSIG Get the signal sent when input or output becomes possible. A value of zero means **signals.sigio** is sent. Any other value (including **signals.sigio**) is the signal sent instead, and in this case additional info is available to the signal handler if installed with **signals.sa_siginfo**.

linux.F_SETSIG Sets the signal sent when input or output becomes possible. A value of zero means to send the default **signals.sigio** signal. Any other value (including **signals.sigio**) is the signal to send instead, and in this case additional info is available to the signal handler if installed with **signals.sa_siginfo**.

By using **linux.f_setsig** with a non-zero value, and setting **signals.sa_siginfo** for the signal handler (see *sigaction*(2)), extra information about I/O events is passed to the handler in a *siginfo_t* structure. If the *si_code* field indicates the source is **signals.si_sigio**, the *si_fd* field gives the file descriptor associated with the event. Otherwise, there is no indication which file descriptors are pending, and you should use the usual mechanisms (*select*(2), *poll*(2), *read*(2) with **linux.o_nonblock** set, etc.) to determine which file descriptors are available for I/O.

By selecting a POSIX.1b real time signal (value >= **signals.sigrtmin**), multiple I/O events may be queued using the same signal numbers. (Queuing is dependent on available memory). Extra information is available if **signals.sa_siginfo** is set for the signal handler, as above.

Using these mechanisms, a program can implement fully asynchronous I/O without using *select*(2) or *poll*(2) most of the time.

The use of **linux.o_async**, **linux.f_getown**, **linux.f_setown** is specific to BSD and Linux. **linux.f_getsig** and **linux.f_setsig** are Linux-specific. POSIX has asynchronous I/O and the *aio_sigevent* structure to achieve similar things; these are also available in Linux as part of the GNU C Library (Glibc).

RETURN VALUE

For a successful call, the return value depends on the operation:

linux.f_dupfd The new descriptor.

linux.f_getfd Value of flag.

linux.f_getfl Value of flags.

linux.f_getown Value of descriptor owner.

linux.f_getsig Value of signal sent when read or write becomes possible, or zero for traditional **signals.sigio** behaviour.

All other commands

Zero.

On error, EAX will contain a negative valued error code.

ERRORS

errno.eacces Operation is prohibited by locks held by other processes.

errno.eagain Operation is prohibited because the file has been memory-mapped by another process.

errno.ebadf *fd* is not an open file descriptor.

errno.edeadlk It was detected that the specified **linux.f_setlkw** command would cause a deadlock.

errno.efault *lock* is outside your accessible address space.

errno.eintr For **linux.f_setlkw**, the command was interrupted by a signal. For **linux.f_getlk** and **linux.f_setlk**, the command was interrupted by a signal before the lock was checked or acquired. Most likely when locking a remote file (e.g. locking over NFS), but can sometimes happen locally.

errno.einval For **linux.f_dupfd**, *arg* is negative or is greater than the maximum allowable value. For **linux.f_setsig**, *arg* is not an allowable signal number.

errno.emfile For **linux.f_dupfd**, the process already has the maximum number of file descriptors open.

errno.enolck Too many segment locks open, lock table is full, or a remote locking protocol failed (e.g. locking

over NFS).

errno.eperm Attempted to clear the **linux.o_append** flag on a file that has the append-only attribute set.

NOTES

The errors returned by *dup2* are different from those returned by **linux.f_dupfd**.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. Only the operations linux.f_dupfd, linux.f_getfd, linux.f_setfd, linux.f_getfl, linux.f_setfl, linux.f_getlk, linux.f_setlk and linux.f_setlkw are specified in POSIX.1. linux.f_getown and linux.f_setown are BSDisms not supported in SVr4; linux.f_getsig and linux.f_setsig are specific to linux. The flags legal for linux.f_getfl/linux.f_setfl are those supported by *open*(2) and vary between these systems; linux.o_append, linux.o_nonblock, linux.o_rdonly, and linux.o_rdwr are specified in POSIX.1. SVr4 supports several other options and flags not documented here.

SVr4 documents additional errno.eio, errno.enolink and errno.eoverflow error conditions.

SEE ALSO

dup2(2), flock(2), open(2), socket(2)

3.23 fdatasync

```
procedure linux.fdatasync( fd:dword );
    @nodisplay;
begin fdatasync;

    linux.pushregs;
    mov( linux.sys_fdatasync, eax );
    mov( fd, ebx );
    int( $80 );
    linux.popregs;

end fdatasync;
```

DESCRIPTION

fdatasync flushes all data buffers of a file to disk (beforethe system call returns). It resembles *fsync* but is not required to update the metadata such as access time.

Applications that access databases or log files often write a tiny data fragment (e.g., one line in a log file) and then call *fsync* immediately in order to ensure that the written data is physically stored on the harddisk. Unfortunately, fsync will always initiate two write operations: one for the newly written data and another one in order to update the modification time stored in the inode. If the modification time is not a part of the transaction concept *fdatasync* can be used to avoid unnecessary inode disk write operations.

RETURN VALUE

On success, zero is returned. On error, EAX will contain a negative error code.

ERRORS

errno.ebadf *fd* is not a valid file descriptor open for writing.

errno.erofs,

errno.einval *fd* is bound to a special file which does not support synchronization.

errno.eio An error occurred during synchronization.

BUGS

Currently (Linux 2.2) fdatasync is equivalent to fsync.

CONFORMING TO

POSIX1b (formerly POSIX.4)

SEE ALSO

fsync(2), B.O. Gallmeister, POSIX.4, O'Reilly, pp. 220-223 and 343.

3.24 flock

```
// flock - file locking.

procedure linux.flock( fd:dword; operation:int32 );
    @nodisplay;
begin flock;

    linux.pushregs;
    mov( linux.sys_flock, eax );
    mov( fd, ebx );
    mov( operation, ecx );
    int( $80 );
    linux.popregs;

end flock;
```

DESCRIPTION

Apply or remove an advisory lock on an open file. The file is specified by *fd*. Valid operations are given below:

linux.lock_sh	Shared lock. More than one process may hold a shared lock for a given file at a given time.
linux.lock_ex	Exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.
linux.lock_un	Unlock.
linux.lock_nb	Don't block when locking. May be specified (by or'ing) along with one of the other operations.

A single file may not simultaneously have both shared and exclusive locks.

A file is locked (i.e., the inode), not the file descriptor. So, *dup*(2) and *fork*(2) do not create multiple instances of a lock.

RETURN VALUE

On success, zero is returned. On error, EAX will contain a negative error code.

ERRORS

errno.ewouldblock	The file is locked and the **linux.lock_nb** flag was selected.

CONFORMING TO

4.4BSD (the flock(2) call first appeared in 4.2BSD).

NOTES

flock(2) does not lock files over NFS. Use *fcntl*(2) instead: that does work over NFS, given a sufficiently recent version of Linux and a server which supports locking.

flock(2) and *fcntl*(2) locks have different semantics with respect to forked processes and *dup*(2).

SEE ALSO

open(2), *close*(2), dup(2), *execve*(2), *fcntl*(2), *fork*(2), *lockf*(3)

There are also locks.txt and mandatory.txt in /usr/src/linux/Documentation.

3.25 fork

```
// fork - starts a new process.
//
//  To the parent- returns child PID in EAX.
//  To the child- returns zero in EAX.

procedure linux.fork; nodisplay; noframe;
begin fork;

    linux.pushregs;
    mov( linux.sys_fork, eax );
    int( $80 );
    linux.popregs;
    ret();

end fork;
```

DESCRIPTION

fork creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

Under Linux, fork is implemented using copy-on-write pages, so the only penalty incurred by fork is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

RETURN VALUE

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a negative error code will be returned in EAX in the parent's context, and no child process will be created.

ERRORS

errno.eagain *fork* cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

errno.enomem *fork* failed to allocate the necessary kernel structures because memory is tight.

CONFORMING TO

The fork call conforms to SVr4, SVID, POSIX, X/OPEN, BSD 4.3.

SEE ALSO

clone(2), execve(2), vfork(2), wait(2)

3.26 fstat, lstat, stat

```
// lstat: Retrieve file info.

procedure linux.fstat( fd:dword; var buf:linux.stat_t );
    @nodisplay;
begin fstat;

    linux.pushregs;
    mov( linux.sys_fstat, eax );
    mov( fd, ebx );
    mov( buf, ecx );
    int( $80 );
    linux.popregs;

end fstat;

// lstat: Retrieve file info.

procedure linux.lstat( filename:string; var buf:linux.stat_t );
    @nodisplay;
begin lstat;

    linux.pushregs;
    mov( linux.sys_lstat, eax );
    mov( filename, ebx );
    mov( buf, ecx );
    int( $80 );
    linux.popregs;

end lstat;

// stat: Retrieve file info.

procedure linux.stat( filename:string; var buf:linux.stat_t );
    @nodisplay;
begin stat;

    linux.pushregs;
    mov( linux.sys_stat, eax );
    mov( filename, ebx );
    mov( buf, ecx );
    int( $80 );
    linux.popregs;

end stat;
```

DESCRIPTION

These functions return information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file.

stat stats the file pointed to by *file_name* and fills in *buf*.

lstat is identical to *stat*, except in the case of a symbolic link, where the link itself is stated, not the file that it refers to.

fstat is identical to *stat*, only the open file pointed to by *filedes* (as returned by open(2)) is stated in place of *file_name*.

They all return a *stat_t* structure, which contains the following fields:

```
stat_t:record
    st_dev     :word;
    __pad1     :word;
    st_ino     :dword;
    st_mode    :word;
    st_nlink   :word;
    st_uid     :word;
    st_gid     :word;
    st_rdev    :word;
    __pad2     :word;
    st_size    :dword;
    st_blksze  :dword;
    st_blocks  :dword;
    st_atime   :dword;
    __unused1  :dword;
    st_mtime   :dword;
    __unused2  :dword;
    st_ctime   :dword;
    __unused3  :dword;
    __unused4  :dword;
    __unused5  :dword;
endrecord;
```

The value *st_size* gives the size of the file (if it is a regular file or a symlink) in bytes. The size of a symlink is the length of the pathname it contains, without trailing NUL.

The value *st_blocks* gives the size of the file in 512-byte blocks.(This may be smaller than *st_size*/512 e.g. when the file has holes.) The value *st_blksize* gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read- modify-rewrite.)

Not all of the Linux filesystems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See `noatime' in mount(8).)

The field *st_atime* is changed by file accesses, e.g. by *exec*(2), *mknod*(2), *pipe*(2), *utime*(2) and *read*(2) (of more than zero bytes). Other routines, like *mmap*(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, e.g. by *mknod*(2), *truncate*(2), *utime*(2) and *write*(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is not changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The following flags are defined for the st_mode field:

linux.s_ifmt	bitmask for the file type bitfields
linux.s_ifsock	socket

linux.s_iflnk	symbolic link
linux.s_ifreg	regular file
linux.s_ifblk	block device
linux.s_ifdir	directory
linux.s_ifchr	character device
linux.s_ififo	fifo
linux.s_isuid	set UID bit
linux.s_isgid	set GID bit (see below)
linux.s_isvtx	sticky bit (see below)
linux.s_irwxu	mask for file owner permissions
linux.s_iread	owner has read permission
linux.s_iwrite	owner has write permission
linux.s_iexec	owner has execute permission
linux.s_irwxg	mask for group permissions
linux.s_irgrp	group has read permission
linux.s_iwgrp	group has write permission
linux.s_ixgrp	group has execute permission
linux.s_irwxo	mask for permissions for others (not in group)
linux.s_iroth	others have read permission
linux.s_iwoth	others have write permisson
linux.s_ixoth	others have execute permission

The set GID bit (**linux.s_isgid**) has several special uses: For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective gid of the creating process, and directories created there will also get the **linux.s_isgid** bit set. For a file that does not have the group execution bit (**linux.s_ixgrp**) set, it indicates mandatory file/record locking.

The `sticky' bit (**linux.s_isvtx**) on a directory means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, and by root.

RETURN VALUE

On success, zero is returned. On error, EAX will contain a negative error code.

ERRORS

errno.ebadf	*filedes* is bad.
errno.enoent	A component of the path file_name does not exist, or the path is an empty string.
errno.enotdir	A component of the path is not a directory.
errno.eloop	Too many symbolic links encountered while traversing the path.
errno.efault	Bad address.
errno.eacces	Permission denied.
errno.enomem	Out of memory (i.e. kernel memory).
errno.enametoolong	File name too long.

CONFORMING TO

The *stat* and *fstat* calls conform to SVr4, SVID, POSIX, X/OPEN, BSD 4.3. The *lstat* call conforms to 4.3BSD and SVr4. SVr4 documents additional *fstat* error conditions linux.eintr, linux.enolink, and linux.eoverflow. SVr4 documents additional stat and lstat error conditions linux.eacces, linux.eintr, linux.emultihop, linux.enolink, and linux.eoverflow. Use of the *st_blocks* and *st_blksize* fields may be less portable. (They were introduced in BSD. Are not specified by POSIX. The interpretation differs between systems, and possibly on a single system when NFS mounts are involved.)

POSIX does not describe the linux.s_ifmt, linux.s_ifsock, linux.s_iflnk, linux.s_ifreg, linux.s_ifblk, linux.s_ifdir, linux.s_ifchr, linux.s_ififo, linux.s_isvtx bits, but instead demands the use of the macros linux.s_isdir(), etc. The linux.s_islnk and linux.s_issock macros are not in POSIX.1-1996, but both will be in the next POSIX standard; the former is from SVID 4v2, the latter from SUSv2.

Unix V7 (and later systems) had linux.s_iread, linux.s_iwrite, linux.s_iexec, where POSIX prescribes the synonyms linux.s_irusr, linux.s_iwusr, linux.s_ixusr.

SEE ALSO

chmod(2), chown(2), readlink(2), utime(2)

3.27 fstatfs, statfs

```
// fstatfs: returns information about a mounted file system.

procedure linux.fstatfs( fd:dword; var buf:linux.statfs_t );
    @nodisplay;
begin fstatfs;

    linux.pushregs;
    mov( linux.sys_fstatfs, eax );
    mov( fd, ebx );
    mov( buf, ecx );
    int( $80 );
    linux.popregs;

end fstatfs;

// statfs: returns information about a mounted file system.

procedure linux.statfs( path:string; var buf:linux.statfs_t );
    @nodisplay;
begin statfs;

    linux.pushregs;
    mov( linux.sys_statfs, eax );
    mov( path, ebx );
    mov( buf, ecx );
    int( $80 );
    linux.popregs;

end statfs;
```

DESCRIPTION

statfs returns information about a mounted file system. path is the path name of any file within the mounted filesystem. buf is a pointer to a *statfs* structure defined as follows:

```
statfs_t:record
     f_type    :dword;
     f_bsize   :dword;
     f_blocks  :dword;
     f_bfree   :dword;
     f_bavail  :dword;
     f_files   :dword;
     f_ffree   :dword;
     f_fsid    :@global:kernel.__kernel_fsid_t;
     f_namelen :dword;
     f_spare   :dword[6];
endrecord;
```

File system types:

linux/affs_fs.h:	AFFS_SUPER_MAGIC	$ADFF
linux/efs_fs.h:	EFS_SUPER_MAGIC	$00414A53
linux/ext_fs.h:	EXT_SUPER_MAGIC	$137D
linux/ext2_fs.h:	EXT2_OLD_SUPER_MAGIC	$EF51
	EXT2_SUPER_MAGIC	$EF53
linux/hpfs_fs.h:	HPFS_SUPER_MAGIC	$F995E849
linux/iso_fs.h:	ISOFS_SUPER_MAGIC	$9660
linux/minix_fs.h:	MINIX_SUPER_MAGIC	$137F /* orig. minix */
	MINIX_SUPER_MAGIC2	$138F /* 30 char minix */
	MINIX2_SUPER_MAGIC	$2468 /* minix V2 */
	MINIX2_SUPER_MAGIC2	$2478 /* minix V2, 30 char names */
linux/msdos_fs.h:	MSDOS_SUPER_MAGIC	$4d44
linux/ncp_fs.h:	NCP_SUPER_MAGIC	$564c
linux/nfs_fs.h:	NFS_SUPER_MAGIC	$6969
linux/proc_fs.h:	PROC_SUPER_MAGIC	$9fa0
linux/smb_fs.h:	SMB_SUPER_MAGIC	$517B
linux/sysv_fs.h:	XENIX_SUPER_MAGIC	$012FF7B4
	SYSV4_SUPER_MAGIC	$012FF7B5
	SYSV2_SUPER_MAGIC	$012FF7B6
	COH_SUPER_MAGIC	$012FF7B7
linux/ufs_fs.h:	UFS_MAGIC	$00011954
linux/xfs_fs.h:	XFS_SUPER_MAGIC	$58465342
linux/xia_fs.h:	_XIAFS_SUPER_MAGIC	$012FD16D

Fields that are undefined for a particular file system are set to 0. *fstatfs* returns the same information about an open file referenced by descriptor *fd.*

RETURN VALUE

On success, zero is returned. On error, EAX is returned with a negative error code.

ERRORS

For *statfs*:

errno.enotdir A component of the path prefix of path is not a directory.

errno.enametoolong path is too long.

errno.enoent The file referred to by path does not exist.

errno.eacces Search permission is denied for a component of the path prefix of path.

errno.eloop Too many symbolic links were encountered in translating path.

errno.efault Buf or path points to an invalid address.

errno.eio An I/O error occurred while reading from or writing to the file system.

errno.enomem Insufficient kernel memory was available.

errno.enosys The filesystem path is on does not support *statfs*.

For *fstatfs*:

errno.ebadf *fd* is not a valid open file descriptor.

errno.efault *buf* points to an invalid address.

errno.eio An I/O error occurred while reading from or writing to the file system.

errno.enosys The filesystem *fd* is open on does not support *fstatfs*.

CONFORMING TO

The Linux *statfs* was inspired by the 4.4BSD one (but they do not use the same structure).

SEE ALSO

stat(2)

3.28 fsync

```
// fsync: writes volatile data to disk.

procedure fsync( fd:dword );
    @nodisplay;
begin fsync;

    linux.pushregs;

    mov( linux.sys_fsync, eax );
    mov( fd, ebx );
    int( $80 );
    linux.popregs;

end fsync;
```

DESCRIPTION

fsync copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage. It also updates metadata stat information. It does not necessarily ensure that the entry in the directory containing the file has also reached disk. For that an explicit *fsync* on the file descriptor of the directory is also needed.

fdatasync does the same as *fsync* but only flushes user data, not the meta data like the mtime or atime.

RETURN VALUE

On success, zero is returned. On error, EAX returns with a negative error code.

ERRORS

errno.ebadf	*fd* is not a valid file descriptor open for writing.
errno.erofs, **errno.einval**	*fd* is bound to a special file which does not support synchronization.
errno.EIO	An error occurred during synchronization.

NOTES

In case the hard disk has write cache enabled, the data may not really be on permanent storage when *fsync/fdatasync* return.

When an ext2 file system is mounted with the sync option, directory entries are also implicitely synced by *fsync*.

On kernels before 2.4, *fsync* on big files can be inefficient. An alternative might be to use the linux.o_sync flag to *open*(2).

CONFORMING TO

POSIX.1b (formerly POSIX.4)

SEE ALSO

bdflush(2), open(2), sync(2), mount(8), update(8), sync(8)

3.29 ftruncate, truncate

```
// ftruncate: shortens a file to the specified length

procedure linux.ftruncate( fd:dword; length:linux.off_t );
    @nodisplay;
begin ftruncate;

    linux.pushregs;
    mov( linux.sys_ftruncate, eax );
    mov( fd, ebx );
    mov( length, ecx );
    int( $80 );
    linux.popregs;

end ftruncate;

// truncate: shortens a file to the specified length

procedure linux.truncate( path:string; length:linux.off_t );
    @nodisplay;
begin truncate;

    linux.pushregs;
    mov( linux.sys_truncate, eax );
    mov( path, ebx );
    mov( length, ecx );
    int( $80 );
    linux.popregs;

end truncate;
```

DESCRIPTION

Truncate causes the file named by path or referenced by fd to be truncated to at most length bytes in size. If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is unspecified whether the file is left unchanged or is extended. In the latter case the extended part reads as zero bytes. With *ftruncate*, the file must be open for writing.

RETURN VALUE

On success, zero is returned. On error, the function returns a negative error code in EAX.

ERRORS

For *truncate*:

errno.ENOTDIR A component of the path prefix is not a directory.

errno.ENAMETOOLONG A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.

errno.ENOENT The named file does not exist.

errno.EACCES Search permission is denied for a component of the path prefix.

errno.EACCES The named file is not writable by the user.

errno.ELOOP Too many symbolic links were encountered in translating the pathname.

errno.EISDIR	The named file is a directory.
errno.EROFS	The named file resides on a read-only file system.
errno.ETXTBSY	The file is a pure procedure (shared text) file that is being executed.
errno.EIO	An I/O error occurred updating the inode.
errno.EFAULT	Path points outside the process's allocated address space.

For *ftruncate*:

errno.EBADF	The *fd* is not a valid descriptor.
errno.EINVAL	The *fd* references a socket, not a file.
errno.EINVAL	The *fd* is not open for writing.

CONFORMING TO

4.4BSD, SVr4 (these function calls first appeared in BSD 4.2). SVr4 documents additional truncate error conditions errno.eintr, errno.emfile, errno.emultihp, errno.enametoolong, errno.enfile, errno.enolink, errno.enotdir. SVr4 documents for *ftruncate* additional errno.eagain and errno.eintr error conditions. POSIX has *ftruncate* but not *truncate*.

The POSIX standard does not define what happens if the file has fewer bytes than *length*.

BUGS

These calls should be generalized to allow ranges of bytes in a file to be discarded.

SEE ALSO

open(2)

3.30 getcwd

```
// getcwd - Retrieve the path of the current working directory.

procedure linux.getcwd( var buf:var; maxlen:linux.size_t );
    @nodisplay;
begin getcwd;

    linux.pushregs;
    mov( linux.sys_getcwd, eax );
    mov( buf, ebx );
    mov( maxlen, ecx );
    int( $80 );
    linux.popregs;

end getcwd;
```

DESCRIPTION

The *getcwd* function copies an absolute pathname of the current working directory to the array pointed to by *buf*, which is of length *maxlen*.

If the current absolute path name would require a buffer longer than size elements, NULL is returned, and EAX is set to errno.erange; an application should check for this error, and allocate a larger buffer if necessary.

If buf is NULL, the behaviour of *getcwd* is undefined.

RETURN VALUE

NULL on failure (for example, if the current directory is not readable), with EAX set accordingly, and *buf* on success. The contents of the array pointed to by *buf* is undefined on error.

CONFORMING TO

POSIX.1

SEE ALSO

chdir(2), free(3), malloc(3)

3.31 getdents

```
// getdents - iterates over a set of directory entries.

procedure linux.getdents( fd:dword; var dirp:linux.dirent; count:int32 );
    @nodisplay;
begin getdents;

    linux.pushregs;
    mov( linux.sys_getdents, eax );
    mov( fd, ebx );
    mov( dirp, ecx );
    mov( count, edx );
    int( $80 );
    linux.popregs;

end getdents;
```

DESCRIPTION

getdents reads several dirent structures from the directory pointed at by fd into the memory area pointed to by *dirp*. The parameter count is the size of the memory area.

The *dirent* structure is declared as follows:

```
    dirent: record
         d_ino    :dword;
         d_off    :off_t;
         d_reclen :uns16;
         d_name   :char[256];
    endrecord;
```

d_ino is an inode number. *d_off* is the distance from the start of the directory to the start of the next *dirent*. *d_reclen* is the size of this entire *dirent*. *d_name* is a null-terminated file name.

This call supersedes *readdir*(2).

RETURN VALUE

On success, the number of bytes read is returned. On end of directory, 0 is returned. On error, EAX will contain a negative error code.

ERRORS

errno.ebadf Invalid file descriptor fd.

errno.efault Argument points outside the calling process's address space.

errno.einval Result buffer is too small.

errno.enoent No such directory.

errno.enotdir File descriptor does not refer to a directory.

CONFORMING TO

SVr4, SVID. SVr4 documents additional errno.enolink, errno.eio error conditions.

SEE ALSO

readdir(2), readdir(3)

3.32 getegid, getgid

```
// getegid - gets the real group id for this process.

procedure linux.getegid;
    @nodisplay;
begin getegid;

    linux.pushregs;
    mov( linux.sys_getegid, eax );
    int( $80 );
    linux.popregs;

end getegid;

// getgid - gets the effective group ID for the current process.

procedure linux.getgid;
    @nodisplay;
begin getgid;

    linux.pushregs;
    mov( linux.sys_getgid, eax );
    int( $80 );
    linux.popregs;

end getgid;
```

DESCRIPTION

getgid returns the real group ID of the current process.

getegid returns the effective group ID of the current process.

The real ID corresponds to the ID of the calling process. The effective ID corresponds to the set ID bit on the file being executed.

ERRORS

These functions are always successful.

CONFORMING TO

POSIX, BSD 4.3

SEE ALSO

setregid(2), setgid(2)

3.33 geteuid, getuid

```
// geteuid - gets the real user id for this process.

procedure linux.geteuid;
    @nodisplay;
begin geteuid;

    linux.pushregs;
    mov( linux.sys_geteuid, eax );
    int( $80 );
    linux.popregs;

end geteuid;

// getuid - Retrieves the userID for a given process.

procedure linux.getuid;
    @nodisplay;
begin getuid;

    linux.pushregs;
    mov( linux.sys_getuid, eax );
    int( $80 );
    linux.popregs;

end getuid;
```

DESCRIPTION

getuid returns the real user ID of the current process.

geteuidreturns the effective user ID of the current pro cess.

The real ID corresponds to the ID of the calling process. The effective ID corresponds to the set ID bit on the file being executed.

ERRORS

These functions are always successful.

CONFORMING TO

POSIX, BSD 4.3.

SEE ALSO

setreuid(2), setuid(2)

3.34 getgid

See getegid.

3.35 getgroups, setgroups

```
// getgroups - Fetches a set of suplementary groups.

procedure linux.getgroups( size:dword; var list:var );
    @nodisplay;
begin getgroups;

    linux.pushregs;
    mov( linux.sys_getgroups, eax );
    mov( size, ebx );
    mov( list, ecx );
    int( $80 );
    linux.popregs;

end getgroups;

// setgroups:

procedure linux.setgroups( size:linux.size_t; var list:var );
    @nodisplay;
begin setgroups;

    linux.pushregs;
    mov( linux.sys_setgroups, eax );
    mov( size, ebx );
    mov( list, ecx );
    int( $80 );
    linux.popregs;

end setgroups;
```

DESCRIPTION

getgroups Up to size supplementary group IDs are returned in list. It is unspecified whether the effective group ID of the calling process is included in the returned list. (Thus, an application should also call *getegid*(2) and add or remove the resulting value.) If size is zero, list is not modified, but the total number of supplementary group IDs for the process is returned.

setgroups Sets the supplementary group IDs for the process. Only the super-user may use this function.

RETURN VALUE

getgroups On success, the number of supplementary group IDs is returned. On error, EAX will contain a negative error code.

setgroups	On success, zero is returned. On error, EAX contains the negative error code.

ERRORS

errno.efault list has an invalid address.

errno.eperm For setgroups, the user is not the super-user.

errno.einval For setgroups, size is greater than NGROUPS (32 for Linux 2.0.32). For *getgroups*, size is less than the number of supplementary group IDs, but is not zero.

NOTES

A process can have up to at least NGROUPS_MAX supplementary group IDs in addition to the effective group ID. The set of supplementary group IDs is inherited from the par ent process and may be changed using setgroups.The maximum number of supplementary group IDs can be found using *sysconf*(3):

```
long ngroups_max;
ngroups_max = sysconf(_SC_NGROUPS_MAX);
```

The maximal return value of *getgroups* cannot be larger than one more than the value obtained this way.

CONFORMING TO

SVr4, SVID (issue 4 only; these calls were not present in SVr3), X/OPEN, 4.3BSD. The *getgroups* function is in POSIX.1. Since *setgroups* requires privilege, it is not covered by POSIX.1.

SEE ALSO

initgroups(3), getgid(2), setgid(2)

3.36 getitimer, setitimer

```
// getitimer: Retrieve interval timer info.

procedure linux.getitimer( which:dword; var theValue:linux.itimerval );
    @nodisplay;
begin getitimer;

    linux.pushregs;
    mov( linux.sys_getitimer, eax );
    mov( which, ebx );
    mov( theValue, ecx );
    int( $80 );
    linux.popregs;

end getitimer;

// setitimer: Sets up an interval timer.

procedure linux.setitimer
    (
            which:dword;
        var   ivalue:linux.itimerval;
        var ovalue:linux.itimerval
    );
    @nodisplay;
begin setitimer;

    linux.pushregs;
    mov( linux.sys_setitimer, eax );
    mov( which, ebx );
    mov( ivalue, ecx );
    mov( ovalue, edx );
    int( $80 );
    linux.popregs;

end setitimer;
```

DESCRIPTION

The system provides each process with three interval timers, each decrementing in a distinct time domain. When any timer expires, a signal is sent to the process, and the timer (potentially) restarts.

The "which" parameter selects the particular time via one of the following three constants:

linux.itimer_real decrements in real time, and delivers signals.sigalrm upon expiration.

linux.itimer_virtual decrements only when the process is executing, and delivers signal.sigvtalrm upon expiration.

linux.itimer_prof decrements both when the process executes and when the system is executing on behalf of the process. Coupled with linux.itimer_virtual, this timer is usually used to profile the time spent by the application in user and kernel space. signals.sigprof is delivered upon expiration.

Timer values are defined by the following structures:

```
type
    itimerval: record
         it_interval: timeval;          /* next value */
         it_value : timeval;            /* current value */
    endrecord;

    imeval: record
         tv_sec :dword;                 /* seconds */
         tv_usec :dword;                /* microseconds */
    endrecord;
```

Getitimer(2) fills the structure indicated by value with the current setting for the timer indicated by which (one of linux.itimer_real, linux.itimer_virtual, or linux.itimer_prof). The element it_value is set to the amount of time remaining on the timer, or zero if the timer is disabled. Similarly, it_interval is set to the reset value. Setitimer(2) sets the indicated timer to the value in value. If ovalue is nonzero, the old value of the timer is stored there.

Timers decrement from it_value to zero, generate a signal, and reset to it_interval. A timer which is set to zero (it_value is zero or the timer expires and it_interval is zero) stops.

Both tv_sec and tv_usec are significant in determining the duration of a timer.

Timers will never expire before the requested time, instead expiring some short, constant time afterwards, dependent on the system timer resolution (currently 10ms). Upon expiration, a signal will be generated and the timer reset. If the timer expires while the process is active (always true for ITIMER_VIRT) the signal will be delivered immediately when generated. Otherwise the delivery will be offset by a small time dependent on the system loading.

RETURN VALUE

On success, zero is returned. On error, these calls return a negative error code in EAX.

ERRORS

errno.efault value or ovalue are not valid pointers.

errno.einval which is not one of linux.itimer_real, linux.itimer_virt, or linux.itimer_prof.

CONFORMING TO

SVr4, 4.4BSD (This call first appeared in 4.2BSD).

SEE ALSO

gettimeofday(2), sigaction(2), signal(2)

BUGS

Under Linux, the generation and delivery of a signal are distinct, and there each signal is permitted only one outstanding event. It's therefore conceivable that under pathologically heavy loading, linux.itimer_real will expire before the signal from a previous expiration has been delivered. The second signal in such an event will be lost.

3.37 getpgid, setpgid, getpgrp, setpgrp

```
// getpgid - Returns a process group ID for the specified process.

procedure linux.getpgid( pid:linux.pid_t );
    @nodisplay;
begin getpgid;

    linux.pushregs;
    mov( linux.sys_getpgid, eax );
    mov( pid, ebx );
    int( $80 );
    linux.popregs;

end getpgid;

// getpgrp - return's this process' parent's group ID.

procedure linux.getpgrp;
    @nodisplay;
begin getpgrp;

    linux.pushregs;
    mov( linux.sys_getpgrp, eax );
    int( $80 );
    linux.popregs;

end getpgrp;

// setpgid - changes a process group ID.

procedure linux.setpgid( pid:linux.pid_t; pgid:linux.pid_t );
    @nodisplay;
begin setpgid;

    linux.pushregs;
    mov( linux.sys_setpgid, eax );
    mov( pid, ebx );
    mov( pgid, ecx );
    int( $80 );
    linux.popregs;

end setpgid;

// setpgrp - changes a process’ parent group ID.

procedure linux.setpgrp( pid:linux.pid_t; pgid:linux.pid_t );
    @nodisplay;
begin setpgrp;

    linux.pushregs;
    mov( linux.sys_setpgrp, eax );
    mov( pid, ebx );
    mov( pgid, ecx );
    int( $80 );
    linux.popregs;

end setpgrp;
```

DESCRIPTION

linux.setpgid sets the process group ID of the process specified by pid to pgid. If pid is zero, the process ID of the current process is used. If pgid is zero, the process ID of the process specified by pid is used. If setpgid is used to move a process from one process group to another (as is done by some shells when creating pipelines), both process groups must be part of the same session. In this case, the pgid specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

linux.getpgid returns the process group ID of the process specified by pid. If pid is zero, the process ID ofthe current process is used.

In theLinux DLL 4.4.1 library, linux.setpgrp simply calls setpgid(0,0).

linux.getpgrp is equivalent to getpgid(0). Each process group is a member of a session and each process is a member of the session of which its process group is a member.

Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input: Processes that have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read. These calls are thus used by programs such as csh(1) to create process groups in implementing job control. The TIOCGPGRP and TIOCSPGRP calls described in termios(4) are used to get/set the pro cess group of the control terminal.

If a session has a controlling terminal, CLOCAL is not set and a hangup occurs, then the session leader is sent a signals.sighup. If the session leader exits, the signals.sighup signal will be sent to each process in the foreground process group of the controlling terminal.

If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a signals.sighup signal followed by a signals.sigcont signal will be sent to each process in the newly-orphaned process group.

RETURN VALUE

On success, setpgid and setpgrp return zero. On error, Linux returns a negative error code in EAX.

linux.getpgid	returns a process group on success. On error, EAX contains a negative error code.
linux.getpgrp	always returns the current process group.

ERRORS

errrno.einval	pgid is less than 0.
errrno.eperm	Various permission violations.
errrno.esrch	pid does not match any process.

CONFORMING TO

The functions setpgid and getpgrp conform to POSIX.1. The function linux.setpgrp is from BSD 4.2. The function linux.getpgid conforms to SVr4.

NOTES

POSIX took setpgid from the BSD function setpgrp. Also SysV has a function with the same name, but it is identical to setsid(2).

SEE ALSO

getuid(2), setsid(2), tcsetpgrp(3), termios(4)

3.38 getpid, getppid

```
// getpid- Returns a process' ID.

procedure linux.getpid;
    @nodisplay;
begin getpid;

    linux.pushregs;
    mov( linux.sys_getpid, eax );
    int( $80 );
    linux.popregs;

end getpid;

// getppid - return's this process' parent's process ID.

procedure linux.getppid;
    @nodisplay;
begin getppid;

    linux.pushregs;
    mov( linux.sys_getppid, eax );
    int( $80 );
    linux.popregs;

end getppid;
```

DESCRIPTION

linux.getpid returns the process ID of the current process in EAX. (This is often used by routines that generate unique temporary file names.)

linux.getppid returns the process ID of the parent of the current process (in EAX)

CONFORMING TO

POSIX, BSD 4.3, SVID

SEE ALSO

exec(3), fork(2), kill(2), mkstemp(3),tmpnam(3), tempnam(3), tmpfile(3)

3.39 getpriority, setpriority

```
// getpriority: get the scheduling priority for a process.

procedure linux.getpriority( which:dword; who:dword );
    @nodisplay;
begin getpriority;

    linux.pushregs;
    mov( linux.sys_getpriority, eax );
    mov( which, ebx );
    mov( who, ecx );
    int( $80 );
    linux.popregs;

end getpriority;

// setpriority: sets the scheduling priority for a process.

procedure linux.setpriority( which:dword; who:dword );
    @nodisplay;
begin setpriority;

    linux.pushregs;
    mov( linux.sys_setpriority, eax );
    mov( which, ebx );
    mov( who, ecx );
    int( $80 );
    linux.popregs;

end setpriority;
```

DESCRIPTION

The scheduling priority of the process, process group, or user, as indicated by which and who is obtained with the getpriority call and set with the setpriority call. Which is one of linux.prio_process, linux.prio_pgrp, or linux.prio_user, and who is interpreted relative to which (a process identifier for linux.prio_process, process group identifier for linux.prio_pgrp, and a user ID for linux.prio_user). A zero value of who denotes the current process, process group, or user. Prio is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favorable scheduling.

The linux.getpriority call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The setpriority call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

RETURN VALUE

Since getpriority can legitimately return the value -1, it is necessary to verify that EAX is outside the range -20..+20 when testing for an error return value. The setpriority call returns 0 if there is no error, or a negative value if there is an error.

ERRORS

errno.esrch	No process was located using the which and who values specified.
errno.einval	Which was not one of linux.prio_process, linux.prio_pgrp, or linux.prio_user.

In addition to the errors indicated above, setpriority will fail if:

errno.eperm A process was located, but neither its effective nor real user ID matched the effective user ID of the caller.

errno.eacces A non super-user attempted to lower a process priority.

CONFORMING TO

SVr4, 4.4BSD (these function calls first appeared in 4.2BSD).

SEE ALSO

nice(1), fork(2), renice(8)

3.40 getresgid, getresuid

```
// getresgid - Retrieves the group IDs for a process.

procedure linux.getresgid
(
    var  rgid  :linux.gid_t;
    var  egid  :linux.gid_t;
    var  sgid  :linux.gid_t
);
    @nodisplay;
begin getresgid;

    linux.pushregs;
    mov( linux.sys_getresgid, eax );
    mov( rgid, ebx );
    mov( egid, ecx );
    mov( sgid, edx );
    int( $80 );
    linux.popregs;

end getresgid;

// getresuid - Retrieves the various user IDs for a process.

procedure linux.getresuid
(
    var  ruid  :linux.uid_t;
    var  euid  :linux.uid_t;
    var  suid  :linux.uid_t
);
    @nodisplay;
begin getresuid;

    linux.pushregs;
    mov( linux.sys_getresuid, eax );
    mov( ruid, ebx );
    mov( euid, ecx );
    mov( suid, edx );
    int( $80 );
    linux.popregs;

end getresuid;
```

DESCRIPTION

getresuid and getresgid (both introduced in Linux 2.1.44) get the real, effective and saved user ID's (resp. group ID's) of the current process.

RETURN VALUE

On success, zero is returned. On error, EAX will contain a negative error code.

ERRORS

errno.efault One of the arguments specified an address outside the calling program's address space.

CONFORMING TO

This call is Linux-specific.

SEE ALSO

getuid(2), setuid(2), getreuid(2), setreuid(2), setresuid(2)

3.41 getrlimit, getrusage, setrlimit

```
// getrlimit - Retrieves resource limitations.

procedure linux.getrlimit( resource:dword; var rlim:linux.rlimit );
    @nodisplay;
begin getrlimit;

    linux.pushregs;
    mov( linux.sys_getrlimit, eax );
    mov( resource, ebx );
    mov( rlim, ecx );
    int( $80 );
    linux.popregs;

end getrlimit;

// getrlimit - Retrieves resource limitations.

procedure linux.getrlimit( resource:dword; var rlim:linux.rlimit );
    @nodisplay;
begin getrlimit;

    linux.pushregs;
    mov( linux.sys_setrlimit, eax );
    mov( resource, ebx );
    mov( rlim, ecx );
    int( $80 );
    linux.popregs;

end getrlimit;

// setrlimit - Sets resource limitations.

procedure linux.setrlimit( resource:dword; var rlim:linux.rlimit );
    @nodisplay;
begin setrlimit;

    linux.pushregs;
    mov( linux.sys_setrlimit, eax );
    mov( resource, ebx );
    mov( rlim, ecx );
    int( $80 );
    linux.popregs;

end setrlimit;
```

DESCRIPTION

linux.getrlimit and linux.setrlimit get and set resource limits respectively. resource should be one of:

linux.rlimit_cpu	CPU time in seconds

linux.rlimit_fsize	Maximum filesize
linux.rlimit_data	max data size
linux.rlimit_stack	max stack size
linux.rlimit_core	max core file size
linux.rlimit_rss	max resident set size
linux.rlimit_nproc	max number of processes
linux.rlimit_nofile	max number of open files
linux.rlimit_memlock	max locked-in-memory address space
linux.rlimit_as	ddress space (virtual memory) limit

A resource may unlimited if you set the limit to linux.rlim_infinity. linux.rlimit_ofile is the BSD name for linux.rlimit_nofile.

The rlimit structure is defined as follows :

```
type
   rlimit: record
        rlim_cur :rlim_t;
        rlim_max :rlim_t;
   endrecord;
```

linux.getrusage returns the current resource usages, for a who of either linux.rusage_self or linux.rusage_children.

```
type
   rusage :record
        ru_utime     :timeval;     /* user time used */
        ru_stime     :timeval;     /* system time used */
        ru_maxrss    :dword;       /* maximum resident set size */
        ru_ixrss     :dword;       /* integral shared memory size */
        ru_idrss     :dword;       /* integral unshared data size */
        ru_isrss     :dword        /* integral unshared stack size */
        ru_minflt    :dword;       /* page reclaims */
        ru_majflt    :dword;       /* page faults */
        ru_nswap     :dword;       /* swaps */
        ru_inblock   :dword;       /* block input operations */
        ru_oublock   :dword;       /* block output operations */
        ru_msgsnd    :dword;       /* messages sent */
        ru_msgrcv    :dword;       /* messages received */
        ru_nsignals  :dword;       /* signals received */
        ru_nvcsw     :dword;       /* voluntary context switches */
        ru_nivcsw    :dword;       /* involuntary context switches */
   endrecord;
```

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.efault	rlim or usage points outside the accessible address space.

errno.einval	linux.getrlimit or setrlimit is called with a bad resource, or getrusage is called with a bad who.
errno.eperm	A non-superuser tries to use setrlimit() to increase the soft or hard limit above the current hard limit, or a superuser tries to increase linux.rlimit_nofile above the current kernel maximum.

CONFORMING TO

SVr4, BSD 4.3

The above structure was taken from BSD 4.3 Reno. Not all fields are meaningful under Linux. Right now (Linux 2.4) only the fields linux.ru_utime, linux.ru_stime, linux.ru_minflt, linux.ru_majflt, and linux.ru_nswap are maintained.

SEE ALSO

quotactl(2), ulimit(3)

3.42 getsid

```
// getsid - Returns the session ID of the calling process.

procedure linux.getsid( pid:linux.pid_t );
    @nodisplay;
begin getsid;

    linux.pushregs;
    mov( linux.sys_getsid, eax );
    mov( pid, ebx );
    int( $80 );
    linux.popregs;

end getsid;
```

DESCRIPTION

linux.getsid(0) returns the session ID of the calling process. linux.getsid(p) returns the session ID of the process with process ID p.

ERRORS

On error, errno.esrch will be returned. The only error which can happen is errno.esrch, when no process with process ID p was found.

CONFORMING TO

SVr4, which documents an additional EPERM error condition.

SEE ALSO

setsid(2)

3.43 gettimeofday, settimeofday

```
// gettimeofday - Retrieves the current time.

procedure linux.gettimeofday( var tv:linux.timeval; var tz:linux.timezone );
    @nodisplay;
begin gettimeofday;

    linux.pushregs;
    mov( linux.sys_gettimeofday, eax );
    mov( tv, ebx );
    mov( tz, ecx );
    int( $80 );
    linux.popregs;

end gettimeofday;

// settimeofday - Sets the current time.

procedure linux.settimeofday( var tv:linux.timeval; var tz:linux.timezone );
    @nodisplay;
begin settimeofday;

    linux.pushregs;
    mov( linux.sys_settimeofday, eax );
    mov( tv, ebx );
    mov( tz, ecx );
    int( $80 );
    linux.popregs;

end settimeofday;
```

DESCRIPTION

linux.gettimeofday and linux.settimeofday can set the time as well as a timezone. tv is a timeval struct, as specified in /usr/hla/include/linux.hhf:

```
type
    timeval :record
        tv_sec    :dword;    /* seconds */
        tv_usec   :dword;    /* microseconds */
    endrecord;
```

and tz is a timezone :

```
type
    timezone :record;
        tz_minuteswest   :int32;    /* minutes W of Greenwich */
        tz_dsttime       :int32;    /* type of dst correction */
    endrecord;
```

The use of the timezone struct is obsolete; the tz_dsttime field has never been used under Linux - it has not been and will not be supported by libc or glibc. Each and every occurrence of this field in the kernel source (other than the declaration) is a bug.

Under Linux there is some peculiar `warp clock' semantics associated to the settimeofday system call if on the very first call (after booting) that has a non-NULL tz argument, the tv argument is NULL and the tz_minuteswest field is nonzero. In such a case it is assumed that the CMOS clock is on local time, and that it has to be incremented by this amount to get UTC system time. No doubt it is a bad idea to use this feature.

Only the super user may use settimeofday.

RETURN VALUE

linux.gettimeofday and linux.settimeofday return 0 for success, or a negative error code in EAX.

ERRORS

errno.eperm	settimeofday is called by someone other than the superuser.
errno.einval	Timezone (or something else) is invalid.
errno.efault	One of tv or tz pointed outside your accessible address space.

CONFORMING TO

SVr4, BSD 4.3

SEE ALSO

date(1), adjtimex(2), time(2), ctime(3), ftime(3)

3.44 getuid

See “geteuid, getuid” on page 57.

3.45 get_kernel_syms

```
// get_kernel_syms- Fetch the kernel symbol table.

procedure linux.get_kernel_syms( var table:linux.kernel_sym );
    @nodisplay;
begin get_kernel_syms;

    linux.pushregs;

    mov( linux.sys_get_kernel_syms, eax );
    mov( table, ebx );
    int( $80 );
    linux.popregs;

end get_kernel_syms;
```

DESCRIPTION

If table is NULL, get_kernel_syms returns the number of symbols available for query. Otherwise it fills in a table of structures:

```
type
    kernel_sym : record;
          value     :dword;;
          name      :char[60];
    endrecord;
```

The symbols are interspersed with magic symbols of the form #module-name with the kernel having an empty name.

The value associated with a symbol of this form is the address at which the module is loaded.

The symbols exported from each module follow their magic module tag and the modules are returned in the reverse order they were loaded.

RETURN VALUE

Returns the number of symbols returned. There is no possible error return.

SEE ALSO

create_module(2), init_module(2), delete_module(2), query_module(2).

BUGS

There is no way to indicate the size of the buffer allo cated for table. If symbols have been added to the kernel since the program queried for the symbol table size, memory will be corrupted.

The length of exported symbol names is limited to 59 (ASCIIZ string).

Because of these limitations, this system call is deprecated in favor of query_module.

3.46 init_module

```
// init_module- Initializes a device driver module.

procedure linux.init_module( theName:string; var image:linux.module_t );
    @nodisplay;
begin init_module;

    linux.pushregs;

    mov( linux.sys_init_module, eax );
    mov( theName, ebx );
    mov( image, ecx );
    int( $80 );
    linux.popregs;

end init_module;
```

DESCRIPTION

init_module loads the relocated module image into kernel space and runs the module's init function.

The module image begins with a module structure and is followed by code and data as appropriate. The module structure is defined as follows:

```
type
    module_t:
         record
              size_of_struct     :uns32;
              next               :dword; // pointer to module_t
              theName            :pointer to char;
              size               :uns32;
              uc:
                  union
                      usecount   :@global:atomic.atomic_t;
                      pad        :dword;
                  endunion;

              flags              :dword;
              nsyms              :uns32;
              ndeps              :uns32;

              syms               :pointer to module_symbol;
              deps               :pointer to module_ref;
              refs               :pointer to module_ref;

              init               :procedure; returns( "eax" );
              cleanup            :procedure;

              ex_table_start     :pointer to exception_table_entry;
              ex_table_end       :pointer to exception_table_entry;

              persist_start      :pointer to module_persist;
              persist_end        :pointer to module_persist;

              can_unload         :procedure; returns( "eax" );
         endrecord;
```

All of the pointer fields, with the exception of next and refs, are expected to point within the module body and be initialized as appropriate for kernel space, i.e. relocated with the rest of the module.

This system call is only open to the superuser and is of use to device driver writers.

RETURN VALUE

On success, zero is returned. On error, EAX will contain an appropriate negative valued error code.

ERRORS

errno.eperm The user is not the superuser.

errno.enoent No module by that name exists.

errno.einval Some image slot filled in incorrectly, image->name does not correspond to the original module name, some image->deps entry does not correspond to a loaded module, or some other similar inconsistency.

errno.ebusy The module's initialization routine failed.

errno.efault name or image is outside the program's accessible address space.

SEE ALSO

create_module(2), delete_module(2), query_module(2).

3.47 ioctl

```
    #macro ioctl( d, request, argp[] );

         #if( @elements( argp ) = 0 )

               ioctl2( d, request )

         #else

               ioctl3( d, request, @text( argp[0] ))

         #endif

    #endmacro;

// ioctl2 - two parameter form of the ioctl function.

procedure linux.ioctl2( d:int32;  request:int32 );
    @nodisplay;
begin ioctl2;

    linux.pushregs;
    mov( linux.sys_ioctl, eax );
    mov( d, ebx );
    mov( request, ecx );
    int( $80 );
    linux.popregs;

end ioctl2;

// ioctl3 - three parameter form of the ioctl function.

procedure linux.ioctl3( d:int32;  request:int32; argp:string );
    @nodisplay;
begin ioctl3;

    linux.pushregs;
    mov( linux.sys_ioctl, eax );
    mov( d, ebx );
    mov( request, ecx );
    mov( argp, edx );
    int( $80 );
    linux.popregs;

end ioctl3;
```

DESCRIPTION

The linux.ioctl macro manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with ioctl requests.The argument d must be an open file descriptor.

A linux.ioctl request has encoded in it whether the argument is an in parameter or out parameter, and the size of the argument argp in bytes. .

RETURN VALUE

Usually, on success zero is returned. A few ioctls use the return value as an output parameter and return a non negative value on success. On error, EAX contains a negative error code.

ERRORS

errno.ebadf	d is not a valid descriptor.
errno.efault	argp references an inaccessible memory area.
errno.enotty	d is not associated with a character special device.
errno.enotty	The specified request does not apply to the kind of object that the descriptor d references.
errno.einval	Request or argp is not valid.

CONFORMING TO

No single standard. Arguments, returns, and semantics of ioctl(2) vary according to the device driver in question (the call is used as a catch-all for operations that don't cleanly fit the Unix stream I/O model). See ioctl_list(2) for a list of many of the known ioctl calls. The ioctl function call appeared in Version 7 AT&T Unix.

SEE ALSO

execve(2), fcntl(2), ioctl_list(2), mt(4), sd(4), tty(4)

3.48 ioperm

```
// ioperm: sets the port access bits.

procedure linux.ioperm( from:dword; num:dword; turn_on:int32 );
    @nodisplay;
begin ioperm;

    linux.pushregs;
    mov( linux.sys_ioperm, eax );
    mov( from, ebx );
    mov( num, ecx );
    mov( turn_on, edx );
    int( $80 );
    linux.popregs;

end ioperm;
```

DESCRIPTION

linux.ioperm sets the port access permission bits for the process for num bytes starting from port address from to the value turn_on. The use of linux.ioperm requires root privileges.

Only the first $3ff I/O ports can be specified in this manner. For more ports, the linux.iopl function must be used. Permissions are not inherited on fork, but on exec they are. This is useful for giving port access permissions to non-privileged tasks.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

CONFORMING TO

linux.ioperm is Linux specific and should not be used in programs intended to be portable.

SEE ALSO

iopl(2)

3.49 iopl

```
// iopl: Changes the I/O privilege level.

procedure linux.iopl( level:dword );
    @nodisplay;
begin iopl;

    linux.pushregs;
    mov( linux.sys_iopl, eax );
    mov( level, ebx );
    int( $80 );
    linux.popregs;

end iopl;
```

DESCRIPTION

linux.iopl changes the I/O privilege level of the current pro cess, as specified in level.

This call is necessary to allow 8514-compatible X servers to run under Linux. Since these X servers require access to all 65536 I/O ports, the linux.ioperm call is not sufficient.

In addition to granting unrestricted I/O port access, running at a higher I/O privilege level also allows the process to disable interrupts. This will probably crash the system, and is not recommended.

Permissions are inherited by fork and exec.

The I/O privilege level for a normal process is 0.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.einval	level is greater than 3.
errno.eperm	The current user is not the super-user.

NOTES FROM THE KERNEL SOURCE

linux.iopl has to be used when you want to access the I/O ports beyond the $3ff range: to get the full 65536 ports bitmapped you'd need 8kB of bitmaps/process, which is a bit excessive.

CONFORMING TO

linux.iopl is Linux specific and should not be used in processes intended to be portable.

SEE ALSO

ioperm(2)

3.50 ipc

```
// ipc- interprocess communication.

procedure linux.ipc
    (
                theCall    :dword;
                first      :dword;
                second     :dword;
                third      :dword;
         var    ptr        :var;
                fifth      :dword
    );
    @nodisplay;
var
    parms:dword[6];

begin ipc;

    linux.pushregs;

    // Create a parameter block to pass to Linux:

    mov( theCall, eax );
    mov( first, ebx );
    mov( second, ecx );
    mov( third, edx );
    mov( ptr, esi );
    mov( fifth, edi );
    mov( eax, parms[0] );
    mov( ebx, parms[4] );
    mov( ecx, parms[8] );
    mov( edx, parms[12] );
    mov( esi, parms[16] );
    mov( edi, parms[20] );

    mov( linux.sys_ipc, eax );
    lea( ebx, parms );
    int( $80 );
    linux.popregs;

end ipc;
```

DESCRIPTION

linux.ipc is a common kernel entry point for the System V IPC calls for messages, semaphores, and shared memory. call determines which IPC function to invoke; the other arguments are passed through to the appropriate call.

User programs should call the appropriate functions by their usual names. Only standard library implementors and kernel hackers need to know about ipc. Note that if the particular ipc function requires five or more parameters, the ipc call must pass a pointer to the fifth and sixth parameters in EDI.

CONFORMING TO

linux.ipc is Linux specific, and should not be used in programs intended to be portable.

SEE ALSO

msgctl(2), msgget(2), msgrcv(2), msgsnd(2), semctl(2), semget(2), semop(2), shmat(2), shmctl(2), shmdt(2), shmget(2)

3.51 kill

```
// kill - sends a signal to a process.

procedure linux.kill( pid:linux.pid_t; sig:int32 );
   @nodisplay;
begin kill;

   linux.pushregs;
   mov( linux.sys_kill, eax );
   mov( pid, ebx );
   mov( sig, ecx );
   int( $80 );
   linux.popregs;

end kill;
```

DESCRIPTION

The kill system call can be used to send any signal to any process group or process.

If pid is positive, then signal sig is sent to pid.

If pid equals 0, then sig is sent to every process in the process group of the current process.

If pid equals -1, then sig is sent to every process except for the first one.

If pid is less than -1, then sig is sent to every process in the process group -pid.

If sig is 0, then no signal is sent, but error checking is still performed.

RETURN VALUE

On success, zero is returned. On error, EAX returns with a negative value.

ERRORS

errno.einval	An invalid signal was specified.
errno.esrch	The pid or process group does not exist. Note that an existing process might be a zombie, a process which already committed termination, but has not yet been wait()ed for.
errno.eperm	The process does not have permission to send the signal to any of the receiving processes. For a process to have permission to send a signal to process pid it must either have root privileges, or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the receiving process. In the case of SIGCONT it suffices when the sending and receiving processes belong to the same session.

BUGS

It is impossible to send a signal to task number one, the init process, for which it has not installed a signal handler. This is done to assure the system is not brought down accidentally.

CONFORMING TO

SVr4, SVID, POSIX.1, X/OPEN, BSD 4.3

SEE ALSO

_exit(2), exit(3), signal(2), signal(7)

3.52 link

```
// link - Create a hard link.

procedure linux.link( oldname:string; newname:string );
   @nodisplay;
begin link;

   linux.pushregs;
   mov( linux.sys_link, eax );
   mov( oldname, ebx );
   mov( newname, ecx );
   int( $80 );
   linux.popregs;

end link;
```

DESCRIPTION

linux.link creates a new link (also known as a hard link) to an existing file.

If newpath exists it will not be overwritten.

This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the `original'.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.exdev oldpath and newpath are not on the same filesystem.

errno.eperm The filesystem containing oldpath and newpath does not support the creation of hard links.

errno.efault oldpath or newpath points outside your accessible address space.

errno.eacces Write access to the directory containing newpath is not allowed for the process's effective uid, or one of the directories in oldpath or newpath did not allow search (execute) permission.

errno.enametoolong oldpath or newpath was too long.

errno.enoent A directory component in oldpath or newpath does not exist or is a dangling symbolic link.

errno.enotdir A component used as a directory in oldpath or newpath is not, in fact, a directory.

errno.enomem Insufficient kernel memory was available.

errno.erofs The file is on a read-only filesystem.

errno.eexist newpath already exists.

errno.emlink The file referred to by oldpath already has the maximum number of links to it.

errno.eloop Too many symbolic links were encountered in resolving oldpath or newpath.

errno.enospc The device containing the file has no room for the new directory entry.

errno.eperm	oldpath is a directory.
errno.eio	An I/O error occurred.

NOTES

Hard links, as created by link, cannot span filesystems. Use symlink if this is required.

CONFORMING TO

SVr4, SVID, POSIX, BSD 4.3, X/OPEN. SVr4 documents additional errno.enolink and errno.emultihop error conditions; POSIX.1 does not document errno.eloop. X/OPEN does not document errno.efault, errno.enomem or errno.eio.

BUGS

On NFS file systems, the return code may be wrong in case the NFS server performs the link creation and dies before it can say so. Use stat(2) to find out if the link got created.

SEE ALSO

symlink(2), unlink(2), rename(2), open(2), stat(2), ln(1)

3.53 llseek

```
// llseek - 64-bit version of lseek.

procedure linux.llseek
(
        fd:dword;
        offset_high:dword;
        offset_low:dword;
   var  theResult:linux.loff_t;
        whence:dword
);
   @nodisplay;
begin llseek;

   linux.pushregs;
   mov( linux.sys_llseek, eax );
   mov( fd, ebx );
   mov( offset_high, ecx );
   mov( offset_low, edx );
   mov( theResult, esi );
   mov( whence, edi );
   int( $80 );
   linux.popregs;

end llseek;
```

DESCRIPTION

The linux.llseek function repositions the offset of the file descriptor fd to (offset_high<<32) | offset_low bytes relative to the beginning of the file, the current position in the file, or the end of the file, depending on whether whence is linux.seek_set, linux.seek_cur, or linux.seek_end, respectively.

It returns the resulting file position in the argument result.

RETURN VALUE

Upon successful completion, linux.llseek returns 0. Otherwise, it returns a negative error code in EAX.

ERRORS

errno.ebadf	fd is not an open file descriptor.
errno.einval	whence is invalid.

CONFORMING TO

This function is Linux-specific, and should not be used in programs intended to be portable.

BUGS

The ext2 filesystem does not support files with a size of 2GB or more.

SEE ALSO

lseek(2)

3.54 lseek

```
// lseek - Reposition a file pointer.

procedure linux.lseek( fd:dword; offset:linux.off_t; origin:dword );
    @nodisplay;
begin lseek;

    linux.pushregs;
    mov( linux.sys_lseek, eax );
    mov( fd, ebx );
    mov( offset, ecx );
    mov( origin, edx );
    int( $80 );
    linux.popregs;

end lseek;
```

DESCRIPTION

The lseek function repositions the offset of the file descriptor fildes to the argument offset according to the directive whence as follows:

linux.seek_set	The offset is set to offset bytes.
linux.seek_cur	The offset is set to its current location plus offset bytes.
linux.seek_end	The offset is set to the size of the file plus offset bytes.

The lseek function allows the file offset to be set beyond the end of the existing end-of-file of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

RETURN VALUE

Upon successful completion, lseek returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a negative error value is returned in EAX.

ERRORS

errno.ebadf Fildes is not an open file descriptor.

errno.espipe Fildes is associated with a pipe, socket, or FIFO.

errno.einval Whence is not a proper value.

CONFORMING TO

SVr4, POSIX, BSD 4.3

RESTRICTIONS

Some devices are incapable of seeking and POSIX does not specify which devices must support it.

Linux specific restrictions: using lseek on a tty device returns errno.espipe. Other systems return the number of written characters, using linux.seek_set to set the counter. Some devices, e.g. /dev/null do not cause the error errno.espipe, but return a pointer which value is undefined.

NOTES

This document's use of whence is incorrect English, but maintained for historical reasons.

When converting old code, substitute values for whence with the following macros:

old	new
0	SEEK_SET
1	SEEK_CUR
2	SEEK_END
L_SET	SEEK_SET
L_INCR	SEEK_CUR
L_XTND	SEEK_END

SVR1-3 returns long instead of off_t, BSD returns int.

SEE ALSO

dup(2), open(2), fseek(3)

3.55 mkdir

```
// mkdir - creates a directory.

procedure linux.mkdir( pathname:string; mode:int32 );
    @nodisplay;
begin mkdir;

    linux.pushregs;
    mov( linux.sys_mkdir, eax );
    mov( pathname, ebx );
    mov( mode, ecx );
    int( $80 );
    linux.popregs;

end mkdir;
```

DESCRIPTION

mkdir attempts to create a directory named pathname.

mode specifies the permissions to use. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

The newly created directory will be owned by the effective uid of the process. If the directory containing the file has the set group id bit set, or if the filesystem is mounted with BSD group semantics, the new directory will inherit the group ownership from its parent; otherwise it will be owned by the effective gid of the process.

If the parent directory has the set group id bit set then so will the newly created directory.

RETURN VALUE

mkdir returns zero on success, or a negative error code in EAX.

ERRORS

errno.eperm The filesystem containing pathname does not support the creation of directories.

errno.eexist pathname already exists (not necessarily as a directory). This includes the case where pathname is a symbolic link, dangling or not.

errno.efault pathname points outside your accessible address space.

errno.eacces The parent directory does not allow write permission to the process, or one of the directories in pathname did not allow search (execute) permission.

errno.enametoolong pathname was too long.

errno.enoent A directory component in pathname does not exist or is a dangling symbolic link.

errno.enotdir A component used as a directory in pathname is not, in fact, a directory.

errno.enomem Insufficient kernel memory was available.

errno.erofs pathname refers to a file on a read-only filesystem.

errno.eloop Too many symbolic links were encountered in resolving pathname.

errno.enospc The device containing pathname has no room for the new directory.

errno.enospc The new directory cannot be created because the user's disk quota is exhausted.

CONFORMING TO

SVr4, POSIX, BSD, SYSV, X/OPEN. SVr4 documents additional EIO, EMULTIHOP and ENOLINK error conditions; POSIX.1 omits ELOOP.

There are many infelicities in the protocol underlying NFS. Some of these affect mkdir.

SEE ALSO

mkdir(1), chmod(2), mknod(2), mount(2), rmdir(2), stat(2), umask(2), unlink(2)

3.56 mknod

```
// mknod- Creates a special (device) file.

procedure linux.mknod( filename:string; mode:dword; dev:linux.dev_t );
    @nodisplay;
begin mknod;

    linux.pushregs;
    mov( linux.sys_mknod, eax );
    mov( filename, ebx );
    mov( mode, ecx );
    movzx( dev, edx );
    int( $80 );
    linux.popregs;

end mknod;
```

DESCRIPTION

linux.mknod attempts to create a filesystem node (file, device special file or named pipe) named pathname, specified by mode and dev.

mode specifies both the permissions to use and the type of node to be created.

It should be a combination (using bitwise OR) of one of the file types listed below and the permissions for the new node.

The permissions are modified by the process's umask in the usual way: the permissions of the created node are (mode & ~umask).

The file type should be one of S_IFREG, S_IFCHR, S_IFBLK and S_IFIFO to specify a normal file (which will be created empty), character special file, block special file or FIFO (named pipe), respectively, or zero, which will create a normal file.

If the file type is S_IFCHR or S_IFBLK then dev specifies the major and minor numbers of the newly created device special file; otherwise it is ignored.

If pathname already exists, or is a symlink, this call fails with an EEXIST error.

The newly created node will be owned by the effective uid of the process. If the directory containing the node has the set group id bit set, or if the filesystem is mounted with BSD group semantics, the new node will inherit the group ownership from its parent directory; otherwise it will be owned by the effective gid of the process.

RETURN VALUE

mknod returns zero on success, or a negative value in EAX if there is an error.

ERRORS

errno.eperm mode requested creation of something other than a FIFO (named pipe), and the caller is not the superuser; also returned if the filesystem containing pathname does not support the type of node requested.

errno.einval mode requested creation of something other than a normal file, device special file or FIFO.

errno.eexist pathname already exists.

errno.efault pathname points outside your accessible address space.

errno.eacces The parent directory does not allow write permission to the process, or one of the directories in pathname did not allow search (execute) permission.

errno.enametoolong pathname was too long.

errno.enoent A directory component in pathname does not exist or is a dangling symbolic link.

errno.enotdir A component used as a directory in pathname is not, in fact, a directory.

errno.enomem Insufficient kernel memory was available.

errno.erofs pathname refers to a file on a read-only filesystem.

errno.eloop Too many symbolic links were encountered in resolving pathname.

errno.enospc The device containing pathname has no room for the new node.

CONFORMING TO

SVr4 (but the call requires privilege and is thus not in POSIX), 4.4BSD. The Linux version differs from the SVr4 version in that it does not require root permission to create pipes, also in that no EMULTIHOP, ENOLINK, or EINTR error is documented.

NOTES

The Austin draft says: "The only portable use of mknod() is to create a FIFO-special file. If mode is not S_IFIFO or dev is not 0, the behavior of mknod() is unspecified."

Under Linux, this call cannot be used to create directories or socket files, and cannot be used to create normal files by users other than the superuser. One should make directories with mkdir, and FIFOs with mkfifo.

There are many infelicities in the protocol underlying NFS. Some of these affect mknod.

SEE ALSO

close(2), fcntl(2), mkdir(2), mount(2), open(2), read(2), socket(2), stat(2), umask(2), unlink(2), write(2), fopen(3), mkfifo(3)

3.57 mlock

```
// mlock - Disables paging for the specified memory region.

procedure linux.mlock( addr:dword; len:linux.size_t );
    @nodisplay;
begin mlock;

    linux.pushregs;
    mov( linux.sys_mlock, eax );
    mov( addr, ebx );
    mov( len, ecx );
    int( $80 );
    linux.popregs;

end mlock;
```

DESCRIPTION

mlock disables paging for the memory in the range starting at addr with length len bytes. All pages which contain a part of the specified memory range are guaranteed be resident in RAM when the mlock system call returns successfully and they are guaranteed to stay in RAM until the pages are unlocked by munlock or munlockall, or until the process terminates or starts another program with exec. Child processes do not inherit page locks across a fork.

Memory locking has two main applications: real-time algorithms and high-security data processing. Real-time applications require deterministic timing, and, like schedulng, paging is one major cause of unexpected program execution delays. Real-time applications will usually also switch to a real-time scheduler with sched_setscheduler. Cryptographic security software often handles critical bytes like passwords or secret keys as data structures. As a result of paging, these secrets could be transfered onto a persistent swap store medium, where they might be accessible to the enemy long after the security software has erased the secrets in RAM and terminated.

Memory locks do not stack, i.e., pages which have been locked several times by calls to mlock or mlockall will be unlocked by a single call to munlock for the corresponding range or by munlockall. Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process.

On POSIX systems on which mlock and munlock are available, _POSIX_MEMLOCK_RANGE is defined in <unistd.h> and the value PAGESIZE from <limits.h> indicates the number of bytes per page.

RETURN VALUE

On success, mlock returns zero. On error, EAX will contain a negative error code and no changes are made to any locks in the address space of the process.

ERRORS

errno.enomem Some of the specified address range does not correspond to mapped pages in the address space of the process or the process tried to exceed the maximum number of allowed locked pages.

errno.eperm The calling process does not have appropriate privileges. Only root processes are allowed to lock pages.

errno.einval len was not a positive number.

CONFORMING TO

POSIX.1b, SVr4. SVr4 documents an additional EAGAIN error code.

SEE ALSO

munlock(2), mlockall(2), munlockall(2)

3.58 mlockall

```
// mlockall - Disables paging for all pages in the current process.

procedure linux.mlockall( flags:dword );
    @nodisplay;
begin mlockall;

    linux.pushregs;
    mov( linux.sys_mlockall, eax );
    mov( flags, ebx );
    int( $80 );
    linux.popregs;

end mlockall;
```

DESCRIPTION

linux.mlockall disables paging for all pages mapped into the address space of the calling process. This includes the pages of the code, data and stack segment, as well as shared libraries, user space kernel data, shared memory and memory mapped files. All mapped pages are guaranteed to be resident in RAM when the mlockall system call returns successfully and they are guaranteed to stay in RAM until the pages are unlocked again by linux.munlock or linux.munlockall or until the process terminates or starts another program with exec. Child processes do not inherit page locks across a fork.

Memory locking has two main applications: real-time algorithms and high-security data processing. Real-time applications require deterministic timing, and, like scheduling, paging is one major cause of unexpected program execution delays. Real-time applications will usually also switch to a real-time scheduler with sched_setscheduler.

Cryptographic security software often handles critical bytes like passwords or secret keys as data structures. As a result of paging, these secrets could be transfered onto a persistent swap store medium, where they might be accessible to the enemy long after the security software has erased the secrets in RAM and terminated. For security applications, only small parts of memory have to be locked, for which mlock is available.

The flags parameter can be constructed from the bitwise OR of the following constants:

linux.mcl_current Lock all pages which are currently mapped into the address space of the process.

linux.mcl_future Lock all pages which will become mapped into the address space of the process in the future. These could be for instance new pages required by a growing heap and stack as well as new memory mapped files or shared memory regions.

If linux.mcl_future has been specified and the number of locked pages exceeds the upper limit of allowed locked pages, then the system call which caused the new mapping will fail with linux.enomem. If these new pages have been mapped by the the growing stack, then the kernel will deny stack expansion and send a signals.sigsegv.

Real-time processes should reserve enough locked stack pages before entering the time-critical section, so that no page fault can be caused by function calls. This can be achieved by calling a function which has a sufficiently large automatic variable and which writes to the memory occupied by this large array in order to touch these stack pages. This way, enough pages will be mapped for the stack and can be locked into RAM. The dummy writes ensure that not even copy-on-write page faults can occur in the critical section.

Memory locks do not stack, i.e., pages which have been locked several times by calls to mlockall or mlock will be unlocked by a single call to munlockall. Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process.

RETURN VALUE

On success, mlockall returns zero. On error, EAX returns with a negative error code.

ERRORS

errno.enomem	The process tried to exceed the maximum number of allowed locked pages.
errno.eperm	The calling process does not have appropriate privileges. Only root processes are allowed to lock pages.
errno.einval	Unknown flags were specified.

CONFORMING TO

POSIX.1b, SVr4. SVr4 documents an additional EAGAIN error code.

SEE ALSO

munlockall(2), mlock(2), munlock(2)

3.59 mmap, munmap

```
// mmap; Memory maps a file.
//      Note: must use @stdcall calling sequence on this one!

procedure linux.mmap
    (
        start    :dword;
        length   :linux.size_t;
        prot     :int32;
        flags    :dword;
        fd       :dword;
        offset   :linux.off_t
    );
    @nodisplay;

begin mmap;

    linux.pushregs;

    // Note: this code assumes the @stdcall calling sequence
    // so that the parameters will be in the right order on the
    // stack when we pass their address to Linux in EBX:

    lea( ebx, start );
    mov( linux.sys_mmap, eax );
    int( $80 );
    linux.popregs;

end mmap;

// munmap: closes a memory mapped file.

procedure linux.munmap( start:dword; length:linux.size_t );
    @nodisplay;
begin munmap;

    linux.pushregs;
    mov( linux.sys_munmap, eax );
    mov( start, ebx );
    mov( length, ecx );
    int( $80 );
    linux.popregs;

end munmap;
```

DESCRIPTION

The linux.mmap function asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor fd into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by linux.mmap. The prot argument describes the desired memory protection (and must not conflict with the open mode of the file). It has bits

linux.prot_exec	Pages may be executed.
linux.prot_read	Pages may be read.
linux.prot_write	Pages may be written.
linux.prot_none	Pages may not be accessed.

The flags parameter specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. It has bits

linux.map_fixed Do not select a different address than the one specified. If the specified address cannot be used, mmap will fail. If linux.map_fixed is specified, start must be a multiple of the pagesize. Use of this option is discouraged.

linux.map_shared Share this mapping with all other processes that map this object. Storing to the region is equivalent to writing to the file. The file may not actually be updated until msync(2) or munmap(2) are called.

linux.map_private Create a private copy-on-write mapping. Stores to the region do not affect the original file.

You must specify exactly one of linux.map_shared and linux.map_private.

The above three flags are described in POSIX.1b (formerly POSIX.4). Linux also knows about linux.map_denywrite, linux.map_executable, linux.map_noreserve, linux.map_locked, linux.map_growsdown and linux.map_anon(ymous).

offset should ordinarily be a multiple of the page size returned by getpagesize(2).

Memory mapped by mmap is preserved across fork(2), with the same attributes.

The linux.munmap system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

RETURN VALUE

On success, mmap returns a pointer to the mapped area. On error, EAX contains an appropriate error code. On success, munmap returns 0, on failure EAX contains an appropriate error code.

ERRORS

errno.ebadf fd is not a valid file descriptor (and map_anonymous was not set).

errno.eacces map_private was requested, but fd is not open for reading. or map_shared was requested and prot_write is set, but fd is not open in read/write (linux.o_rdwr) mode.

errno.einval We don't like start or length or offset. (E.g., they are too large, or not aligned on a pagesize boundary.) fd is open for writing.

errno.eagain The file has been locked, or too much memory has been locked.

errno. enomem No memory is available.

Use of a mapped region can result in these signals:

signals.sigsegv Attempted write into a region specified to mmap as read-only.

signals.sigbus Attempted access to a portion of the buffer that does not correspond to the file (for example, beyond the end of the file, including the case where another process has truncated the file).

CONFORMING TO

SVr4, POSIX.1b (formerly POSIX.4), 4.4BSD. Svr4 documents additional error codes ENXIO and ENODEV.

SEE ALSO

getpagesize(2), msync(2), shm_open(2), B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128-129 and 389-391.

3.60 modify_ldt

```
// modify_ldt- Lets the caller change the x86 LDT table.

procedure linux.modify_ldt( func:dword; var ptr:var; bytecount:dword );
    @nodisplay;
begin modify_ldt;

    linux.pushregs;

    mov( linux.sys_modify_ldt, eax );
    mov( func, ebx );
    mov( ptr, ecx );
    mov( bytecount, edx );
    int( $80 );
    linux.popregs;

end modify_ldt;
```

DESCRIPTION

modify_ldt reads or writes the local descriptor table (ldt) for a process. The ldt is a per-process memory management table used by the i386 processor. For more information on this table, see an Intel 386 processor handbook.

When func is 0, modify_ldt reads the ldt into the memory pointed to by ptr. The number of bytes read is the smaller of bytecount and the actual size of the ldt.

When func is 1, modify_ldt modifies one ldt entry. ptr points to a modify_ldt_ldt_s structure and bytecount must equal the size of this structure.

RETURN VALUE

On success, modify_ldt returns either the actual number of bytes read (for reading) or 0 (for writing). On failure, modify_ldt returns a negative error code in EAX.

ERRORS

errno.enosys func is neither 0 nor 1.

errno.einval ptr is 0, or func is 1 and bytecount is not equal
to the size of the structure modify_ldt_ldt_s, or
func is 1 and the new ldt entry has illegal values.

errno.efault ptr points outside the address space.

CONFORMING TO

This call in Linux-specfic and should not be used in programs intended to be portable.

SEE ALSO

vm86(2)

3.61 mount, umount

```
// mount- mounts a filesystem volume.

procedure linux.mount
(
        specialfile     :string;
        dir             :string;
        filesystemtype  :string;
        new_flags       :dword;
   var  data            :var
);
   @nodisplay;
begin mount;

   linux.pushregs;
   mov( linux.sys_mount, eax );
   mov( dev_name, ebx );
   mov( dir_name, ecx );
   mov( theType, edx );
   mov( new_flags, esi );
   mov( data, edi );
   int( $80 );
   linux.popregs;

end mount;

// umount - unmount a disk volume.

procedure linux.umount
(
        specialfile     :string;
        dir             :string;
        filesystemtype  :string;
        mountflags      :dword;
   var  data            :var
);
   @nodisplay;
begin umount;

   linux.pushregs;
   mov( linux.sys_umount, eax );
   mov( specialfile, ebx );
   mov( dir, ecx );
   mov( filesystemtype, edx );
   mov( mountflags, esi );
   mov( data, edi );
   int( $80 );
   linux.popregs;

end umount;
```

DESCRIPTION

mount attaches the filesystem specified by dev_name (which is often a device name) to the directory specified by dirname.

umount removes the attachment of the (topmost) filesystem mounted on dir.

Only the super-user may mount and unmount filesystems.

The theType argument may take one of the values listed in /proc/filesystems (like "minix", "ext2", "msdos", "proc", "nfs", "iso9660" etc.).

The mountflags argument may have the magic number $C0ED in the top 16 bits, and various mount flags in the low order 16 bits:

linux.ms_rdonly	1	/* mount read-only */
linux.ms_nosuid	2	/* ignore suid and sgid bits */
linux.ms_nodev	4	/* no access to device special files */
linux.ms_noexec	8	/* no program execution */
linux.ms_synchronous	16	/* writes are synced at once */
linux.ms_remount	32	/* alter flags of a mounted fs */
linux.ms_mandlock	64	/* allow mandatory locks */
linux.ms_noatime	1024	/* do not update access times */
linux.ms_nodiratime	2048	/* do not update dir access times */
linux.ms_bind	4096	/* bind subtree elsewhere */

The data argument is interpreted by the different file systems.

RETURN VALUE

On success, zero is returned. On error, EAX will contain a negative error code.

ERRORS

The error values given below result from filesystem type independent errors. Each filesystem type may have its own special errors and its own special behavior. See the kernel source code for details.

errno.eperm The user is not the super-user.

errno.enodev filesystemtype not configured in the kernel.

errno.enotblk specialfile is not a block device (if a device was required).

errno.ebusy specialfile is already mounted. Or, it cannot be remounted read-only, because it still holds files open for writing. Or, it cannot be mounted on dir because dir is still busy (it is the working directory of some task, the mount point of another device, has open files, etc.).

errno.einval specialfile had an invalid superblock. Or, a remount was attempted, while specialfile was not already mounted on dir. Or, an umount was attempted, while dir was not a mount point.

errno.efault One of the pointer arguments points outside the user address space.

errno.enomem The kernel could not allocate a free page to copy filenames or data into.

errno.enametoolong A pathname was longer than linux.maxpathlen.

errno.enoent A pathname was empty or had a nonexistent component.

errno.enotdir The second argument, or a prefix of the first argument, is not a directory.

errno.eacces A component of a path was not searchable. Or, mounting a read-only filesystem was attempted without giving the ms_rdonly flag. Or, the block device Specialfile is located on a filesystem mounted with the ms_nodev option.

errno.enxio	The major number of the block device dev_name is out of range.
errno.emfile	(In case no block device is required:) Table of dummy devices is full.

CONFORMING TO

These functions are Linux-specific and should not be used in programs intended to be portable.

HISTORY

The original umount function was called as umount(device) and would return ENOTBLK when called with something other than a block device. In Linux 0.98p4 a call umount(dir) was added, in order to support anonymous devices. In Linux 2.3.99-pre7 the call umount(device) was removed, leaving only umount(dir) (since now devices can be mounted in more than one place, so specifying the device does not suffice).

The original MS_SYNC flag was renamed MS_SYNCHRONOUS in 1.1.69 when a different MS_SYNC was added to <mman.h>.

SEE ALSO

mount(8), umount(8)

3.62 mprotect

```
// mprotect- Control access to memory.

procedure linux.mprotect( var addr:var; len:linux.size_t; prot:dword );
    @nodisplay;
begin mprotect;

    linux.pushregs;

    mov( linux.sys_mprotect, eax );
    mov( addr, ebx );
    mov( len, ecx );
    mov( prot, edx );
    int( $80 );
    linux.popregs;

end mprotect;
```

DESCRIPTION

linux.mprotect controls how a section of memory may be accessed. If an access is disallowed by the protection given it, the program receives a signals.sigsegv.

prot is a bitwise-or of the following values:

linux.prot_none	The memory cannot be accessed at all.
linux.prot_read	The memory can be read.
linux.prot_write	The memory can be written to.
linux.prot_exec	The memory can contain executing code.

The new protection replaces any existing protection. For example, if the memory had previously been marked prot_read, and mprotect is then called with prot prot_write, it will no longer be readable.

RETURN VALUE

On success, mprotect returns zero. On error, returns a negative error code in EAX.

ERRORS

errno.einval addr is not a valid pointer, or not a multiple of linux.pagesize.

errno.efault The memory cannot be accessed.

errno.eacces The memory cannot be given the specified access. This can happen, for example, if you mmap(2) a file to which you have read-only access, then ask mprotect to mark it linux.prot_write.

errno.enomem Internal kernel structures could not be allocated.

CONFORMING TO

SVr4, POSIX.1b (formerly POSIX.4). SVr4 defines an additional error code EAGAIN. The SVr4 error conditions don'tmap neatly onto Linux's. POSIX.1b says that mprotect canbe used only on regions of memory obtained from mmap(2).

SEE ALSO

mmap(2)

3.63 mremap

```
// mremap - Remaps memory.

procedure linux.mremap
(
    old_address    :dword;
    old_size       :linux.size_t;
    new_size       :linux.size_t;
    flags          :dword
);
    @nodisplay;
begin mremap;

    linux.pushregs;
    mov( linux.sys_mremap, eax );
    mov( old_address, ebx );
    mov( old_size, ecx );
    mov( new_size, edx );
    mov( flags, esi );
    int( $80 );
    linux.popregs;

end mremap;
```

DESCRIPTION

linux.mremap expands (or shrinks) an existing memory mapping, potentially moving it at the same time (controlled by the flags argument and the available virtual address space).

old_address is the old address of the virtual memory block that you want to expand (or shrink). Note that old_address has to be page aligned. old_size is the old size of the virtual memory block. new_size is the requested size of the virtual memory block after the resize.

The flags argument is a bitmap of flags.

In Linux the memory is divided into pages. A user process has (one or) several linear virtual memory segments. Each virtual memory segment has one or more mappings to real memory pages (in the page table). Each virtual memory segment has its own protection (access rights), which may cause a segmentation violation if the memory is accessed incorrectly (e.g., writing to a read-only segment). Accessing virtual memory outside of the segments will also cause a segmentation violation.

linux.mremap uses the Linux page table scheme. linux.mremap changes the mapping between virtual addresses and memory pages. This can be used to implement a very efficient realloc.

FLAGS

linux.mremap_maymove indicates if the operation should fail, or change the virtual address if the resize cannot be done at the current virtual address.

RETURN VALUE

On success mremap returns a pointer to the new virtual memory area. On error, EAX returns with a negative error code.

ERRORS

errno.einval An invalid argument was given. Most likely old_address was not page aligned.

errno.efault "Segmentation fault." Some address in the range old_address to old_address+old_size is an invalid virtual memory address for this process. You can also get EFAULT even if there exist mappings that cover the whole address space requested, but those mappings are of different types.

errno.eagain The memory segment is locked and cannot be remapped.

errno.enomem The memory area cannot be expanded at the current virtual address, and the linux.mremap_maymove flag is not set in flags. Or, there is not enough (virtual) memory available.

CONFORMING TO

This call is Linux-specific, and should not be used in programs intended to be portable. 4.2BSD had a (never actually implemented) mremap(2) call with completely different semantics.

SEE ALSO

getpagesize(2), realloc(3), malloc(3), brk(2), sbrk(2), mmap(2)

Your favorite OS text book for more information on paged memory. (Modern Operating Systems by Andrew S. Tannen baum, Inside Linux by Randolf Bentson, The Design of the UNIX Operating System by Maurice J. Bach.)

3.64 msgctl

```
// msgctl - SysV message operation.

procedure linux.msgctl
(
        msqid    :dword;
        cmd      :dword;
    var buf      :linux.msqid_ds
);
    @nodisplay;
begin msgctl;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_msgctl, ebx );
    mov( msqid, ecx );
    mov( cmd, edx );
    mov( buf, esi );
    int( $80 );
    linux.popregs;

end msgctl;
```

DESCRIPTION

The function performs the control operation specified by cmd on the message queue with identifier msqid. Legal values for cmd are:

linux.ipc_stat Copy info from the message queue data structure into the structure pointed to by buf. The user must have read access privileges on

linux.ipc_set Write the values of some members of the msqid_ds structure pointed to by buf to the message queue data structure, updating also its msg_ctime member. Considered members from the user supplied struct msqid_ds pointed to by buf are

msg_perm.uid

msg_perm.gid

msg_perm.mode /* only lowest 9-bits */

msg_qbytes

The calling process effective user-ID must be one among super-user, creator or owner of the message queue. Only the super-user can raise the msg_qbytes value beyond the system parameter MSGMNB.

linux.ipc_rmid Remove immediately the message queue and its data structures awakening all waiting reader and writer processes (with an error return and errno set to EIDRM). The calling process effective user-ID must be one among super-user, creator or owner of the message queue.

RETURN VALUE

If successful, the return value will be 0, otherwise the function returns a negative error code in EAX.

ERRORS

For a failing return, errno will be set to one among the following values:

errno.eacces The argument cmd is equal to ipc_stat but the calling process has no read access permissions on the message queue msqid.

errno.efault The argument cmd has value ipc_set or ipc_stat but the address pointed to by buf isn't accessible.

errno.eidrm The message queue was removed.

errno.einval Invalid value for cmd or msqid.

errno.eperm The argument cmd has value ipc_set or ipc_rmid but the calling process effective user-ID has insufficient privileges to execute the command. Note this is also the case of a non super-user process trying to increase the msg_qbytes value beyond the value specified by the system parameter MSGMNB.

NOTES

The ipc_info, msg_stat and msg_info control calls are used by the ipcs(8) program to provide information on allocated resources. In the future these can be modified as needed or moved to a proc file system interface.

Various fields in a struct msqid_ds were shorts under Linux 2.2 and have become longs under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by a IPC_64 flag in cmd.)

CONFORMING TO

SVr4, SVID. SVID does not document the EIDRM error condition.

SEE ALSO

ipc(5), msgget(2), msgsnd(2), msgrcv(2)

3.65 msgget

```
// msgget - SysV message operation.

procedure linux.msgget
(
        key      :linux.key_t;
        msgflg:dword
);
   @nodisplay;
begin msgget;

   linux.pushregs;
   mov( linux.sys_ipc, eax );
   mov( linux.ipcop_msgget, ebx );
   mov( key, ecx );
   mov( msgflg, edx );
   int( $80 );
   linux.popregs;

end msgget;
```

DESCRIPTION

The function returns the message queue identifier associated to the value of the key argument. A new message queue is created if key has value ipc_private or key isn't ipc_private, no existing message queue is associated to key, and ipc_creat is asserted in msgflg (i.e. msgflg&ipc_creat is nonzero). The presence in msgflg of the fields ipc_creat and ipc_excl plays the same role, with respect to the existence of the message queue, as the presence of o_creat and o_excl in the mode argument of the open(2) system call: i.e. the msgget function fails if msgflg asserts both ipc_creat and ipc_excl and a message queue already exists for key.

Upon creation, the lower 9 bits of the argument msgflg define the access permissions of the message queue. These permission bits have the same format and semantics as the access permissions parameter in open(2) or creat(2) system calls. (The execute permissions are not used.)

Furthermore, while creating, the system call initializes the system message queue data structure msqid_ds as follows:

msg_perm.cuid and msg_perm.uid are set to the effective user-ID of the calling process.

msg_perm.cgid and msg_perm.gid are set to the effective group-ID of the calling process.

The lowest order 9 bits of msg_perm.mode are set to the lowest order 9 bit of msgflg.

msg_qnum, msg_lspid, msg_lrpid, msg_stime and msg_rtime are set to 0.

msg_ctime is set to the current time.

msg_qbytes is set to the system limit MSGMNB.

If the message queue already exists the access permissions are verified, and a check is made to see if it is marked for destruction.

RETURN VALUE

If successful, the return value will be the message queue identifier (a nonnegative integer), EAX will contain the negative error code.

ERRORS

For a failing return, EAX will be set to one among the following values:

errno.eacces	A message queue exists for key, but the calling
errno.eexist	A message queue exists for key and msgflg was asserting both ipc_creat and ipc_excl.
errno.eidrm	The message queue is marked for removal.
errno.enoent	No message queue exists for key and msgflg wasn't asserting ipc_creat.
errno.enomem	A message queue has to be created but the system has not enough memory for the new data structure.
errno.enospc	A message queue has to be created but the system limit for the maximum number of message queues (MSGMNI) would be exceeded.

NOTES

ipc_private isn't a flag field but a key_t type. If this special value is used for key, the system call ignores everything but the lowest order 9 bits of msgflg and creates a new message queue (on success).

The following is a system limit on message queue resources affecting a msgget call:

linux.msgmni	System wide maximum number of message queues: policy dependent.

BUGS

Use of ipc_private does not actually prohibit other processes from getting access to the allocated message queue.

As for the files, there is currently no intrinsic way for a process to ensure exclusive access to a message queue. Asserting both ipc_creat and ipc_excl in msgflg only ensures (on success) that a new message queue will be created, it doesn't imply exclusive access to the message queue.

CONFORMING TO

SVr4, SVID. SVr4 does not document the EIDRM error code.

SEE ALSO

ftok(3), ipc(5), msgctl(2), msgsnd(2), msgrcv(2)

3.66 msgrcv, msgsnd

```
// msgrcv - SysV message operation.

procedure linux.msgrcv
(
         msgid :dword;
    var  msgp  :linux.msgbuf;
         msgsz :linux.size_t;
         msgtyp:dword;
         msgflg:dword
);
    @nodisplay;
type
    ipc_kludge: record
         msgp  :dword;
         msgtyp:dword;
    endrecord;

var
    tmp  :ipc_kludge;

begin msgrcv;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_msgrcv, ebx );
    mov( msgid, ecx );
    mov( msgsz, edx );
    mov( msgflg, esi );

    // Set up "extra" parameters.

    mov( msgp, edi );
    mov( edi, tmp.msgp );
    mov( msgtyp, edi );
    mov( edi, tmp.msgtyp );
    lea( edi, tmp );

    int( $80 );
    linux.popregs;

end msgrcv;

// msgsnd - SysV message operation.

procedure linux.msgsnd
(
         msgid :dword;
    var  msgp  :linux.msgbuf;
         msgsz :linux.size_t;
         msgflag:dword
);
    @nodisplay;
begin msgsnd;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_msgsnd, ebx );
    mov( msgid, ecx );
    mov( msgp, edx );
    mov( msgsz, esi );
    mov( msgflag, edi );
    int( $80 );
    linux.popregs;

end msgsnd;
```

DESCRIPTION

To send or receive a message, the calling process allocates a structure that looks like the following

```
type
    msgbuf :record
         mtype     :dword;        /* message type, must be > 0 */
         mtext     :char[1];      /* message data */
    endrecord;
```

but with an array mtext of size msgsz, a non-negative integer value. The structure member mtype must have a strictly positive integer value that can be used by the receiving process for message selection (see the section about msgrcv).

The calling process must have write access permissions to send and read access permissions to receive a message on the queue.

The msgsnd system call enqueues a copy of the message pointed to by the msgp argument on the message queue whose identifier is specified by the value of the msqid argument.

The argument msgflg specifies the system call behaviour if enqueuing the new message will require more than msg_qbytes in the queue. Asserting linux.ipc_nowait the message will not be sent and the system call fails returning with errno set to errno.eagain. Otherwise the process is suspended until the condition for the suspension no longer exists (in which case the message is sent and the system call succeeds), or the queue is removed (in which case the system call fails with errno set to errno.eidrm), or the process receives a signal that has to be caught (in which case the system call fails with errno set to errno.eintr).

Upon successful completion the message queue data structure is updated as follows:

- msg_lspid is set to the process-ID of the calling process.
- msg_qnum is incremented by 1.
- msg_stime is set to the current time.

The system call linux.msgrcv reads a message from the message queue specified by msqid into the msgbuf pointed to by the msgp argument removing from the queue, on success, the read message.

The argument msgsz specifies the maximum size in bytes for the member mtext of the structure pointed to by the msgp argument. If the message text has length greater than msgsz, then if the msgflg argument asserts linux.msg_noerror, the message text will be truncated (and the truncated part will be lost), otherwise the message isn't removed from the queue and the system call fails returning with errno set to errno.e2big.

The argument msgtyp specifies the type of message requested as follows:

- If msgtyp is 0, then the message on the queue's front is read.
- If msgtyp is greater than 0, then the first message on the queue of type msgtyp is read if linux.msg_except isn't asserted by the msgflg argument, otherwise the first message on the queue of type not equal to msgtyp will be read.
- If msgtyp is less than 0, then the first message on the queue with the lowest type less than or equal to the absolute value of msgtyp will be read.

The msgflg argument asserts none, one or more (or-ing them) among the following flags:

linux.ipc_nowait For immediate return if no message of the requested type is on the queue. The system call fails with errno set to errno.enomsg.

linux.msg_except Used with msgtyp greater than 0 to read the first message on the queue with message type that differs from msgtyp.

linux.msg_noerror To truncate the message text if longer than msgsz bytes.

If no message of the requested type is available and linux.ipc_nowait isn't asserted in msgflg, the calling process is blocked until one of the following conditions occurs:

- A message of the desired type is placed on the queue.
- The message queue is removed from the system. In such a case the system call fails with errno set to errno.eidrm.
- The calling process receives a signal that has to be caught. In such a case the system call fails with EAX set to errno.eintr.

Upon successful completion the message queue data structure is updated as follows:

- msg_lrpid is set to the process-ID of the calling process.
- msg_qnum is decremented by 1.
- msg_rtime is set to the current time.

RETURN VALUE

On a failure both functions return a negative code in EAX indicating the error, otherwise msgsnd returns 0 and msgrvc returns the number of bytes actually copied into the mtext array.

ERRORS

When msgsnd fails, at return EAX will be set to one among the following values:

errno.eagain The message can't be sent due to the msg_qbytes limit for the queue and linux.ipc_nowait was asserted in mgsflg.

errno.eacces The calling process has no write access permissions on the message queue.

errno.efault The address pointed to by msgp isn't accessible.

errno.eidrm The message queue was removed.

errno.eintr Sleeping on a full message queue condition, the process received a signal that had to be caught.

errno.einval Invalid msqid value, or nonpositive mtype value, or invalid msgsz value (less than 0 or greater than the system value MSGMAX).

errno.enomem The system has not enough memory to make a copy of the supplied msgbuf.

When msgrcv fails, at return EAX will be set to one among the following values:

errno.e2big The message text length is greater than msgsz and linux.msg_noerror isn't asserted in msgflg.

errno.eacces The calling process has no read access permissions on the message queue.

errno.efault The address pointed to by msgp isn't accessible.

errno.eidrm While the process was sleeping to receive a message, the message queue was removed.

errno.eintr While the process was sleeping to receive a message, the process received a signal that had to be caught.

errno.einval Illegal msgqid value, or msgsz less than 0.

errno.enomsg linux.ipc_nowait was asserted in msgflg and no message of the requested type existed on the message queue.

NOTES

The followings are system limits affecting a msgsnd system call:

linux.msgmax Maximum size for a message text: the implementation set this value to 4080 bytes.

linux.msgmnb Default maximum size in bytes of a message queue: policy dependent. The super-user can increase the size of a message queue beyond linux.msgmnb by a msgctl system call.

The implementation has no intrinsic limits for the system wide maximum number of message headers (MSGTQL) and for the system wide maximum size in bytes of the message pool (MSGPOOL).

CONFORMING TO

SVr4, SVID.

SEE ALSO

ipc(5), msgctl(2), msgget(2), msgrcv(2), msgsnd(2)

3.67 msync

```
// msync - flushes to disk changes made to a memory mapped file.

procedure linux.msync( start:dword; length:linux.size_t; flags:dword );
    nodisplay;
begin msync;

    linux.pushregs;
    mov( linux.sys_msync, eax );
    mov( start, ebx );
    mov( length, ecx );
    mov( flags, edx );
    int( $80 );
    linux.popregs;

end msync;
```

DESCRIPTION

msync flushes changes made to the in-core copy of a file that was mapped into memory using mmap(2) back to disk. Without use of this call there is no guarantee that changes are written back before munmap(2) is called. To be more precise, the part of the file that corresponds to the memory area starting at start and having length length is updated. The flags argument may have the bits linux.ms_async, linux.ms_sync and linux.ms_invalidate set, but not both linux.ms_async and linux.ms_sync. linux.ms_async specifies that an update be scheduled, but the call returns immediately. linux.ms_sync asks for an update and waits for it to complete. linux.ms_invalidate asks to invalidate other mappings of the same file (so that they can be updated with the fresh values just written).

RETURN VALUE

On success, zero is returned. On error, EAX contains an appropriate negative error code.

ERRORS

errno.einval start is not a multiple of linux.pagesize, or any bit other than ms_async | ms_invalidate | ms_sync is set in flags.

errno.efault The indicated memory (or part of it) was not mapped.

CONFORMING TO

POSIX.1b (formerly POSIX.4)

SEE ALSO

mmap(2), B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128-129 and 389-391.

3.68 munlock

```
// munlock - Enables paging for the specified memory region.

procedure linux.munlock( addr:dword; len:linux.size_t );
    @nodisplay;
begin munlock;

    linux.pushregs;
    mov( linux.sys_munlock, eax );
    mov( addr, ebx );
    mov( len, ecx );
    int( $80 );
    linux.popregs;

end munlock;
```

DESCRIPTION

linux.munlock reenables paging for the memory in the range starting at addr with length len bytes. All pages which contain a part of the specified memory range can after calling munlock be moved to external swap space again by the kernel.

Memory locks do not stack, i.e., pages which have been locked several times by calls to mlock or mlockall will be unlocked by a single call to munlock for the corresponding range or by munlockall. Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process.

The value linux.pagesize indicates the number of bytes per page.

RETURN VALUE

On success, munlock returns zero. On error, EAX contains a negative error code and no changes are made to any locks in the address space of the process.

ERRORS

errno.enomem	Some of the specified address range does not correspond to mapped pages in the address space of the process.
errno.einval	len was not a positive number.

CONFORMING TO

POSIX.1b, SVr4

SEE ALSO

mlock(2), mlockall(2), munlockall(2)

3.69 munlockall

```
// munlockall - Enables paging for all pages in the current process.

procedure linux.munlockall;
    @nodisplay;
begin munlockall;

    linux.pushregs;
    mov( linux.sys_munlockall, eax );
    int( $80 );
    linux.popregs;

end munlockall;
```

DESCRIPTION

linux.munlockall reenables paging for all pages mapped into the address space of the calling process. Memory locks do not stack, i.e., pages which have been locked several times by calls to mlock or mlockall will be unlocked by a single call to munlockall. Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process.

RETURN VALUE

On success, munlockall returns zero. On error, EAX contains an appropriate negative error code.

CONFORMING TO

POSIX.1b, SVr4

SEE ALSO

mlockall(2), mlock(2), munlock(2)

3.70 nanosleep

```
// nanosleep - Sleeps for a specified number of nanoseconds.

procedure linux.nanosleep( var req:linux.timespec; var rem:linux.timespec );
    @nodisplay;
begin nanosleep;

    linux.pushregs;
    mov( linux.sys_nanosleep, eax );
    mov( req, ebx );
    mov( rem, ecx );
    int( $80 );
    linux.popregs;

end nanosleep;
```

DESCRIPTION

nanosleep delays the execution of the program for at least the time specified in req. The function can return earlier if a signal has been delivered to the process. In this case, it returns errno.eintr in EAX, and writes the remaining time into the structure pointed to by rem unless rem is NULL. The value of rem can then be used to call nanosleep again and complete the specified pause.

The structure timespec is used to specify intervals of time with nanosecond precision. It has the form

```
type
   timespec :record
        tv_sec    :linux.time_t          /* seconds */
        v_nsec    :dword;                /* nanoseconds */
   endrecord;
```

The value of the nanoseconds field must be in the range 0 to 999 999 999.

Compared to sleep(3) and usleep(3), nanosleep has the advantage of not affecting any signals, it is standardized by POSIX, it provides higher timing resolution, and it allows to continue a sleep that has been interrupted by a signal more easily.

ERRORS

In case of an error or exception, the nanosleep system call returns one of the following values in EAX:

errno.eintr The pause has been interrupted by a non-blocked signal that was delivered to the process. The remaining sleep time has been written into rem so that the process can easily call nanosleep again and continue with the pause.

errno.einval The value in the tv_nsec field was not in the range 0 to 999 999 999 or tv_sec was negative.

BUGS

The current implementation of nanosleep is based on the normal kernel timer mechanism, which has a resolution of 1/HZ s (i.e, 10 ms on Linux/i386 and 1 ms on Linux/Alpha). Therefore, nanosleep pauses always for at least the specified time, however it can take up to 10 ms longer than specified until the process becomes runnable again. For the same reason, the value returned in case of a delivered signal in rem is usually rounded to the next larger multiple of 1/HZ s.

As some applications require much more precise pauses (e.g., in order to control some time-critical hardware), nanosleep is also capable of short high-precision pauses. If the process is scheduled under a real-time policy like linux.sched_fifo or linux.sched_rr, then pauses of up to 2 ms will be performed as busy waits with microsecond precision.

CONFORMING TO

POSIX.1b (formerly POSIX.4).

SEE ALSO

sleep(3), usleep(3), sched_setscheduler(2), timer_create(2)

3.71 nice

```
// nice - Adjust the priority of a process.

procedure linux.nice( increment: int32 );
    @nodisplay;
begin nice;

    linux.pushregs;
    mov( linux.sys_nice, eax );
    mov( increment, ebx );
    int( $80 );
    linux.popregs;

end nice;
```

DESCRIPTION

linux.nice adds inc to the nice value for the calling pid. (A large nice value means a low priority.) Only the super user may specify a negative increment, or priority increase.

RETURN VALUE

On success, zero is returned. On error, EAX is returned with the appropriate error code.

ERRORS

errno.eperm A non-super user attempts to do a priority increase by supplying a negative inc.

CONFORMING TO

SVr4, SVID EXT, AT&T, X/OPEN, BSD 4.3. However, the Linux and glibc (earlier than glibc 2.2.4) return value is nonstandard, see below. SVr4 documents an additional EINVAL error code.

NOTES

Note that the routine is documented in SUSv2 to return the new nice value, while the Linux syscall and (g)libc (earlier than glibc 2.2.4) routines return 0 on success. The new nice value can be found using getpriority(2). Note that an implementation in which nice returns the new nice value can legitimately return negative values. To reliably detect an error, verify that EAX is less than or equal to -1024 (all error codes are less than or equal to -1024).

SEE ALSO

nice(1), getpriority(2), setpriority(2), fork(2), renice(8)

3.72 open

```
See Creat.
```

3.73 pause

```
// pause - sleep until a signal is received.

procedure linux.pause;
    @nodisplay;
begin pause;

    linux.pushregs;
    mov( linux.sys_pause, eax );
    int( $80 );
    linux.popregs;

end pause;
```

DESCRIPTION

The pause library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.

RETURN VALUE

The pause function only returns when a signal was caught and the signal-catching function returned. In this case pause returns errno.eintr in EAX.

errno.eintr a signal was caught and the signal-catching function returned.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

SEE ALSO

kill(2), select(2), signal(2)

3.74 personality

```
// Personality - Selects system call personality.

procedure linux.personality( persona:dword );
    @nodisplay;
begin personality;

    linux.pushregs;
    mov( linux.sys_personality, eax );
    mov( persona, ebx );
    int( $80 );
    linux.popregs;

end personality;
```

DESCRIPTION

Linux supports different execution domains, or personalities, for each process. Among other things, execution domains tell Linux how to map signal numbers into signal actions. The execution domain system allows Linux to provide limited support for binaries compiled under other Unix-like operating systems.

linux.personality will make the execution domain referenced by persona the new execution domain of the current process.

RETURN VALUE

On success, persona is made the new execution domain and the previous persona is returned. On error, EAX returns the appropriate negative error code.

ERRORS

errno.einval	persona does not refer to a valid execution domain.

CONFORMING TO

linux.personality is Linux-specific and should not be used in programs intended to be portable.

3.75 pipe

```
// pipe- creates a pipe.

procedure linux.pipe( fd:dword );
    @nodisplay;
begin pipe;

    linux.pushregs;
    mov( linux.sys_pipe, eax );
    mov( fd, ebx );
    int( $80 );
    linux.popregs;

end pipe;
```

DESCRIPTION

linux.pipe creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.emfile	Too many file descriptors are in use by the process.
errno.enfile	The system file table is full.
errno.efault	filedes is not valid.

CONFORMING TO

SVr4, SVID, AT&T, POSIX, X/OPEN, BSD 4.3

SEE ALSO

read(2), write(2), fork(2), socketpair(2)

3.76 poll

```
// poll - Checks to see if input is available from a device.

procedure linux.poll( var ufds:linux.pollfd; nfds:dword; timeout:dword );
    @nodisplay;
begin poll;

    linux.pushregs;
    mov( linux.sys_poll, eax );
    mov( ufds, ebx );
    mov( nfds, ecx );
    mov( timeout, edx );
    int( $80 );
    linux.popregs;

end poll;
```

DESCRIPTION

linux.poll is a variation on the theme of select. It specifies an array of nfds structures of type

```
type
    pollfd : record
          fd       :dword;           /* file descriptor */
          events   :word;            /* requested events */
          revents  :word;            /* returned events */
    endrecord;
```

and a timeout in milliseconds. A negative value means infinite timeout. The field fd contains a file descriptor for an open file. The field events is an input parameter, a bitmask specifying the events the application is interested in. The field revents is an output parameter, filled by the kernel with the events that actually occurred, either of the type requested, or of one of the types linux.pollerr or linux.pollhup or linux.pollnval. (These three bits are meaningless in the events field, and will be set in the revents field whenever the corresponding condition is true.) If none of the events requested (and no error) has occurred for any of the file descriptors, the kernel waits for timeout milliseconds for one of these events to occur.

The following possible bits in these masks are defined in linux.hhf:

linux.pollin	$0001	/* There is data to read */
linux.pollpri	$0002	/* There is urgent data to read */
linux.pollout	$0004	/* Writing now will not block */
linux.pollerr	$0008	/* Error condition */
linux.pollhup	$0010	/* Hung up */
linux.pollnval	$0020	/* Invalid request: fd not open */

linux.hhf also the values linux.pollrdnorm, linux.pollrdband,

linux.pollwrnorm, linux.pollwrband and linux.pollmsg.

RETURN VALUE

On success, a positive number is returned, where the number returned is the number of structures which have non-zero revents fields (in other words, those descriptors with events or errors reported). A value of 0 indicates that the call timed out and no file descriptors have been selected. On error, EAX contains an appropriate negative error code.

ERRORS

errno.EBADF	An invalid file descriptor was given in one of the sets.
errno.ENOMEM	There was no space to allocate file descriptor tables.
errno.EFAULT	The array given as argument was not contained in the calling program's address space.
errno.EINTR	A signal occurred before any requested event.

CONFORMING TO

XPG4-UNIX.

SEE ALSO

select(2)

3.77 prctl

```
// prctl - Process control.

procedure linux.prctl
(
    option:dword;
    arg2 :dword;
    arg3 :dword;
    arg4 :dword;
    arg5 :dword
);
    @nodisplay;
begin prctl;

    linux.pushregs;
    mov( linux.sys_prctl, eax );
    mov( option, ebx );
    mov( arg2, ecx );
    mov( arg3, edx );
    mov( arg4, esi );
    mov( arg5, edi );
    int( $80 );
    linux.popregs;

end prctl;
```

DESCRIPTION

linux.prctl is called with a first argument describing what to do (with values defined in linux.hhf), and further parameters with a significance depending on the first one.

The first argument can be:

linux.pr_set_pdeathsig (since Linux 2.1.57) Set the parent process death signal of the current process to arg2 (either a signal value in the range 1..maxsig, or 0 to clear). This is the signal that the current process will get when its parent dies. This value is cleared upon a fork().

linux.pr_get_pdeathsig (since Linux 2.3.15) Read the current value of the parent process death signal into the memory address whose pointer is passed in arg2.

RETURN VALUE

On success, zero is returned. On error, EAX contains an appropriate negative error code.

ERRORS

errno.einval The value of option is not recognized, or it is linux.pr_set_pdeathsig and arg2 is not zero or a signal number.

CONFORMING TO

This call is Linux-specific.

SEE ALSO

signal(2)

3.78 pread, pwrite

```
// pread - Read data from a file w/o advancing file ptr.

procedure linux.pread
(
         fd       :dword;
    var  buf      :var;
         count :linux.size_t;
         offset:linux.off_t
);
    @nodisplay;
begin pread;

    linux.pushregs;
    mov( linux.sys_pread, eax );
    mov( fd, ebx );
    mov( buf, ecx );
    mov( count, edx );
    mov( offset, esi );
    int( $80 );
    linux.popregs;

end pread;

/ pwrite - Write data to a file w/o advancing file ptr.

procedure linux.pwrite
(
         fd       :dword;
    var  buf      :var;
         count :linux.size_t;
         offset:linux.off_t
);
    @nodisplay;
begin pwrite;

    linux.pushregs;
    mov( linux.sys_pwrite, eax );
    mov( fd, ebx );
    mov( buf, ecx );
    mov( count, edx );
    mov( offset, esi );
    int( $80 );
    linux.popregs;

end pwrite;
```

DESCRIPTION

linux.pread() reads up to count bytes from file descriptor fd at offset offset (from the start of the file) into the buffer starting at buf. The file offset is not changed.

linux.pwrite() writes up to count bytes from the buffer starting at buf to the file descriptor fd at offset offset. The file offset is not changed.

The file referenced by fd must be capable of seeking.

RETURN VALUE

On success, the number of bytes read or written is returned (zero indicates that nothing was written, in the case of pwrite, or end of file, in the case of pread), or EAX will contain an appropriate (negative) error code.

ERRORS

linux.pread can fail and set EAX to any error specified for read(2) or lseek(2). linux.pwrite can fail and set errno to any error specified for write(2) or lseek(2).

CONFORMING TO

Unix98

SEE ALSO

read(2), write(2), lseek(2)

3.79 ptrace

```
// ptrace - Retrives process info for use by a debugger.

procedure linux.ptrace( request:dword; pid:dword; addr:dword; data:dword );
    @nodisplay;
begin ptrace;

    linux.pushregs;
    mov( linux.sys_ptrace, eax );
    mov( request, ebx );
    mov( pid, ecx );
    mov( addr, edx );
    mov( data, esi );
    int( $80 );
    linux.popregs;

end ptrace;
```

DESCRIPTION

The ptrace system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.

The parent can initiate a trace by calling fork(2) and having the resulting child do a PTRACE_TRACEME, followed (typically) by an exec(2). Alternatively, the parent may commence trace of an existing process using PTRACE_ATTACH.

While being traced, the child will stop each time a signal is delivered, even if the signal is being ignored. (The exception is SIGKILL, which has its usual effect.) The parent will be notified at its next wait(2) and may inspect and modify the child process while it is stopped. The parent then causes the child to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

When the parent is finished tracing, it can terminate the child with PTRACE_KILL or cause it to continue executing in a normal, untraced mode via PTRACE_DETACH.

The value of request determines the action to be performed:

linux.ptrace_traceme Indicates that this process is to be traced by its parent. Any signal (except SIGKILL) delivered to this process will cause it to stop and its parent to be notified via wait. Also, all subsequent calls to exec by this process will cause a

SIGTRAP to be sent to it, giving the parent a chance to gain control before the new program begins execution. A process probably shouldn't make this request if its parent isn't expecting to trace it. (pid, addr, and data are ignored.)

The above request is used only by the child process; the rest are used only by the parent. In the following requests, pid specifies the child process to be acted on. For requests other than linux.ptrace_kill, the child process must be stopped.

linux.ptrace_peektext,

linux.ptrace_peekdata Reads a word at the location addr in the child's memory, returning the word as the result of the ptrace call. Linux does not have separate text and data address spaces, so the two requests are currently equivalent. (data is ignored.)

linux.ptrace_peekuser Reads a word at offset addr in the child's USER area, which holds the registers and other information about the process. The word is returned as the result of the ptrace call. Typically the offset must be word-aligned, though this might vary by architecture. (data is ignored.)

linux.ptrace_poketext,

linux.ptrace_pokedata Copies a word from location data in the parent's memory to location addr in the child's memory. As above, the two requests are currently equivalent.

linux.ptrace_pokeuser Copies a word from location data in the parent's memory to offset addr in the child's USER area. As above, the offset must typically be word-aligned. In order to maintain the integrity of the kernel, some modifications to the USER area are disallowed.

linux.ptrace_getregs,

linux.ptrace_getfpregs Copies the child's general purpose or floating-point registers, respectively, to location data in the parent. See linux.hhf for information on the format of this data. (addr is ignored.)

linux. ptrace_setregs,

linux.ptrace_setfpregs Copies the child's general purpose or floating-point registers, respectively, from location data in the parent. As for linux.ptrace_pokeuser, some general purpose register modifications may be disallowed. (addr is ignored.)

linux.ptrace_cont Restarts the stopped child process. If data is non-zero and not signals.sigstop, it is interpreted as a signal to be delivered to the child; otherwise, no signal is delivered. Thus, for example, the parent can control whether a signal sent to the child is delivered or not. (addr is ignored.)

linux.ptrace_syscall,

linux.ptrace_singlestep Restarts the stopped child as for linux.ptrace_cont, but arranges for the child to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively. (The child will also, as usual, be stopped upon receipt of a signal.) From the parent's perspective, the child will appear to have been stopped by receipt of a signals.sigtrap. So, for linux.ptrace_syscall, for example, the idea is to inspect the arguments to the system call at the first stop, then do another linux.ptrace_syscall and inspect the return value of the system call at the second stop. (addr is ignored.)

linux.ptrace_kill Sends the child a signals.sigkill to terminate it. (addr and data are ignored.)

linux.ptrace_attach Attaches to the process specified in pid, making it a traced "child" of the current process; the behavior of the child is as if it had done a linux.ptrace_traceme. The current process actually becomes the parent of the child process for most purposes (e.g., it will receive notification of child events and appears in ps(1) output as the child's parent), but a getpid(2) by the child will still return the pid of the original parent. The child is sent a signals.sigstop, but will not necessarily have stopped by the completion of this call; use linux.wait to wait for the child to stop. (addr and data are ignored.)

linux.ptrace_detach Restarts the stopped child as for linux.ptrace_cont, but first detaches from the process, undoing the reparenting effect of linux.ptrace_attach, and the effects of linux.ptrace_traceme. Although perhaps not intended, under Linux a traced child can be detached in this way regardless of which method was used to initiate tracing. (addr is ignored.)

NOTES

init(8), the process with pid 1, may not be traced.

The layout of the contents of memory and the USER area are quite OS- and architecture-specific.

The size of a "word" is determined by the OS variant (e.g., for 32-bit Linux it's 32 bits, etc.).

Tracing causes a few subtle differences in the semantics of traced processes. For example, if a process is attached to with linux.ptrace_attach, its original parent can no longer receive notification via wait when it stops, and there is no way for the new parent to effectively simulate this notification.

This page documents the way the ptrace call works currently in Linux. Its behavior differs noticeably on other flavors of Unix. In any case, use of ptrace is highly OS- and architecture-specific.

RETURN VALUE

On success, linux.ptrace_peek* requests return the requested data, while other requests return zero. On error, all requests an appropriate negative error code in EAX. Since the value returned by a successful linux.ptrace_peek* request may be -1, the caller must check EAX upon return to verify that it’s a value error number (less than or equal to -1024).

ERRORS

errno.eperm The specified process cannot be traced. This could be because the parent has insufficient privileges; non-root processes cannot trace processes that they cannot send signals to or those running setuid/setgid programs, for obvious reasons. Alternatively, the process may already be being traced, or be init (pid 1).

errno.esrch The specified process does not exist, or is not currently being traced by the caller, or is not stopped (for requests that require that).

errno.eio request is invalid, or an attempt was made to read from or write to an invalid area in the parent's or child's memory, or there was a word-alignment violation, or an invalid signal was specified during a restart request.

errno.efault There was an attempt to read from or write to an invalid area in the parent's or child's memory, probably because the area wasn't mapped or accessible. Unfortunately, under Linux, different variations of this fault will return errno.eio or errno.efault more or less arbitrarily.

CONFORMING TO

SVr4, SVID EXT, AT&T, X/OPEN, BSD 4.3

SEE ALSO

exec(3), wait(2), signal(2), fork(2), gdb(1), strace(1)

3.80 pwrite

See pread.

3.81 query_module

```
// query_module - Tests for a device driver module.

procedure linux.query_module
(
         theName  :string;
         which    :dword;
    var  buf      :var;
         bufsize  :linux.size_t;
    var  retval   :linux.size_t
);
    @nodisplay;
begin query_module;

    linux.pushregs;
    mov( linux.sys_query_module, eax );
    mov( theName, ebx );
    mov( which, ecx );
    mov( buf, edx );
    mov( bufsize, esi );
    mov( retval, esi );
    int( $80 );
    linux.popregs;

end query_module;
```

DESCRIPTION

linux.query_module requests information related to loadable modules from the kernel. The precise nature of the information and its format depends on the which sub unction. Some functions require name to name a currently loaded module, some allow name to be NULL indicating the kernel proper.

VALUES OF WHICH

0 Always returns success. Used to probe for the system call.

kernel.qm_modules Returns the names of all loaded modules. The output buffer format is adjacent null-terminated strings; ret is set to the number of modules.

kernel.qm_deps Returns the names of all modules used by the indicated module. The output buffer format is adjacent null-terminated strings; ret is set to the number of modules.

kernel.qm_refs Returns the names of all modules using the indicated module. This is the inverse of QM_DEPS. The output buffer format is adjacent null-terminated strings; ret is set to the number of modules.

kernel.qm_symbols Returns the symbols and values exported by the kernel or the indicated module. The buffer format is an array of:

```
type
    module_symbol :record
    _value :dword;
    _name   :dword;
endrecord;
```

followed by null-terminated strings. The value of _name is the character offset of the string relative to the start of buf; ret is set to the number of symbols.

kernel.qm_info Returns miscellaneous information about the indicated module. The output buffer format is:

```
type
        module_info: record
            address : dword;
            size :dword;
            flags : dword;
        endrecord;
```

where address is the kernel address at which the module resides, size is the size of the module in bytes, and flags is a mask of kernel.mod_running, kernel.mod_autoclean, et al that indicates the current status of the module. retval is set to the size of the module_info struct.

RETURN VALUE

On success, zero is returned. On error, EAX contains an appropriate negative error code.

ERRORS

errno.enoent No module by that name exists.

errno.einval Invalid which, or name indicates the kernel for an inappropriate sub function.

errno.enospc The buffer size provided was too small. retval is set to the minimum size needed.

errno.efault At least one of theName, buf, or ret was outside the program's accessible address space.

SEE ALSO

create_module(2), init_module(2), delete_module(2).

3.82 quotactl

```
// access - Manipulates disk quotas.

procedure linux.quotactl
(
    cmd        :dword;
    special    :string;
    id         :dword;
    addr       :linux.caddr_t
);
    @nodisplay;
begin quotactl;

    linux.pushregs;
    mov( linux.sys_quotactl, eax );
    mov( cmd, ebx );
    mov( special, ecx );
    mov( id, edx );
    mov( addr, esi );
    int( $80 );
    linux.popregs;

end quotactl;
```

DESCRIPTION

The linux.quotactl call manipulates disk quotas. cmd indicates a command to be applied to UID id or GID id. To set the type of quota use the QCMD(cmd, type) macro. special is a pointer to a null-terminated string containing the path name of the block special device for the filesystem being manipulated. addr is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of addr is given with each command below.

linux.q_quotaon Turn on quotas for a filesystem. addr points to the path name of file containing the quotas for the filesystem. The quota file must exist; it is normally created with the quotacheck(8) program. This call is restricted to the super-user.

linux.q_quotaoff Turn off quotas for a filesystem. addr and id are ignored. This call is restricted to the super-user.

linux.q_getquota Get disk quota limits and current usage for user or group id. addr is a pointer to a mem_dqblk structure (defined inlinux.hhf). Only the super-user may get the quotas of a user other than himself.

linux.q_setquota Set disk quota limits and current usage for user or group id. addr is a pointer to a mem_dqblk structure (defined in linux.hhf). This call is restricted to the super-user.

linux.q_setqlim Set disk quota limits for user or group id. addr is a pointer to a mem_dqblk structure (defined inlinux.hhf). This call is restricted to the super-user.

linux.q_setuse Set current usage for user or group id. addr is a pointer to a mem_dqblk structure (defined in linux.hhf). This call is restricted to the super-user.

linux.q_sync Update the on-disk copy of quota usages for a filesystem. If special is null then all filesystems with active quotas are sync'ed. addr and id are ignored.

linux. q_getstats Get statistics and other generic information about quota subsystem. addr should be a pointer to dqstats structure (defined in linux.hhf) in which data should be stored. special and id are ignored.

New quota format also allows following additional calls:

linux.q_getinfo Get information (like grace times) about quotafile. addr should be a pointer to mem_dqinfo structure (defined in linux.hhf). id is ignored.

linux.q_setinfo Set information about quotafile. addr should be a pointer to mem_dqinfo structure (defined in linux.hhf). id is ignored. This operation is restricted to super-user.

linux.q_setgrace Set grace times in information about quotafile. addr should be a pointer to mem_dqinfo structure (defined in linux.hhf). id is ignored. This operation is restricted to super-user.

linux.q_setflags Set flags in information about quotafile. These flags are defined in linux.hhf. Note that there are currently no defined flags. addr should be a pointer to mem_dqinfo structure (defined in linux.hhf). id is ignored. This operation is restricted to super-user.

For XFS filesystems making use of the XFS Quota Manager (XQM), the above commands are bypassed and the following commands are used:

linux.q_xquotaon Turn on quotas for an XFS filesystem. XFS provides the ability to turn on/off quota limit enforcement with quota accounting. Therefore, XFS expects the addr to be a pointer to an unsigned int that contains either the flags XFS_QUOTA_UDQ_ACCT and/or XFS_QUOTA_UDQ_ENFD (for user quota), or XFS_QUOTA_GDQ_ACCT and/or XFS_QUOTA_GDQ_ENFD (for group quota), as defined in linux.hhf. This call is restricted to the superuser.

linux.q_xquotaoff Turn off quotas for an XFS filesystem. As in Q_QUOTAON, XFS filesystems expect a pointer to an unsigned int that specifies whether quota accounting and/or limit enforcement need to be turned off. This call is restricted to the superuser.

linux.q_xgetquota Get disk quota limits and current usage for user id. addr is a pointer to a fs_disk_quota structure (defined in linux.hhf). Only the superuser may get the quotas of a user other than himself.

linux.q_xsetqlim Set disk quota limits for user id. addr is a pointer to a fs_disk_quota structure (defined in linux.hhf). This call is restricted to the superuser.

linux.q_xgetqstat Returns a fs_quota_stat structure containing XFS filesystem specific quota information. This is useful in finding out how much space is spent to store quota information, and also to get quotaon/off status of a given local XFS filesystem.

linux.q_xquotarm Free the disk space taken by disk quotas. Quotas must have already been turned off.

There is no command equivalent to Q_SYNC for XFS since sync(1) writes quota information to disk (in addition to the other filesystem metadata it writes out).

RETURN VALUES

quotactl() returns: zero on success and an appropriate negative error code in EAX on error.

ERRORS

errno.efault addr or special are invalid.

errno.einval The kernel has not been compiled with the QUOTA option. cmd is invalid.

errno.enoent The file specified by special or addr does not exist.

errno.enotblk special is not a block device.

errno.eperm The call is privileged and the caller was not the super-user.

errno.esrch No disk quota is found for the indicated user. Quotas have not been turned on for this filesystem.

errno.eusers The quota table is full.

If cmd is linux.q_quotaon, quotactl() may set errno to:

errno.eacces The quota file pointed to by addr exists but is not a regular file.

The quota file pointed to by addr exists but is not on the filesystem pointed to by special.

errno.ebusy linux.q_quotaon attempted while another linux.q_quotaon has already taken place.

SEE ALSO

quota(1), getrlimit(2), quotacheck(8), quotaon(8)

3.83 read

```
// read - reads data via a file handle.

procedure linux.read( fd:dword; var buf:var; count:linux.size_t );
    @nodisplay;
begin read;

    linux.pushregs;
    mov( linux.sys_read, eax );
    mov( fd, ebx );
    mov( buf, ecx );
    mov( count, edx );
    int( $80 );
    linux.popregs;

end read;
```

DESCRIPTION

linux.read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. If count is zero, read() returns zero and has no other results. If count is greater than linux.ssize_max, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, EAX contains the negative error code. In this case it is left unspecified whether the file position (if any) changes.

ERRORS

errno.eintr The call was interrupted by a signal before any data was read.

errno.eagain Non-blocking I/O has been selected using linux.o_nonblock and no data was immediately available for reading.

errno.eio I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling tty, and either it is ignoring or blocking signals.sigttin or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape.

errno.eisdir fd refers to a directory.

errno.ebadf fd is not a valid file descriptor or is not open for reading.

errno.einval fd is attached to an object which is unsuitable for reading.

errno.efault buf is outside your accessible address space.

Other errors may occur, depending on the object connected to fd. POSIX allows a read that is interrupted after reading some data to return errno.eintr or to return the number of bytes already read.

CONFORMING TO

SVr4, SVID, AT&T, POSIX, X/OPEN, BSD 4.3

RESTRICTIONS

On NFS file systems, reading small amounts of data will only update the time stamp the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave atime updates to the server and client side reads satisfied from the client's cache will not cause atime updates on the server as there are no server side reads. UNIX semantics can be obtained by disabling client side attribute caching, but in most situations this will substantially increase server load and decrease performance.

SEE ALSO

close(2), fcntl(2), ioctl(2), lseek(2), readdir(2), readlink(2), select(2), write(2), fread(3)

3.84 readlink

```
// readlink: Extract the text for a symbolic link.

procedure linux.readlink( path:string; var buf:var; bufsize:linux.size_t );
    @nodisplay;
begin readlink;

    linux.pushregs;
    mov( linux.sys_readlink, eax );
    mov( path, ebx );
    mov( buf, ecx );
    mov( bufsize, edx );
    int( $80 );
    linux.popregs;

end readlink;
```

DESCRIPTION

linux.readlink places the contents of the symbolic link path in the buffer buf, which has size bufsiz. readlink does not append a NUL character to buf. It will truncate the contents (to a length of bufsiz characters), in case the buffer is too small to hold all of the contents.

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a negative error code in EAX.

ERRORS

errno.enotdir	A component of the path prefix is not a directory.
errno. einval	bufsiz is not positive.
errno.enametoolong	A pathname, or a component of a pathname, was too long.
errno.enoent	The named file does not exist.
errno.eacces	Search permission is denied for a component of the path prefix.
errno.eloop	Too many symbolic links were encountered in translating the pathname.
errno.einval	The named file is not a symbolic link.
errno.eio	An I/O error occurred while reading from the file system.
errno.efault	buf extends outside the process's allocated address space.
errno.enomem	Insufficient kernel memory was available.

CONFORMING TO

X/OPEN, 4.4BSD (the readlink function call appeared in 4.2BSD).

SEE ALSO

stat(2), lstat(2), symlink(2)

3.85 readv, writev

```
// readv - Scatter/gather input operation.

procedure linux.readv( fd:dword; var vector:var; count:int32 );
    @nodisplay;
begin readv;

    linux.pushregs;
    mov( linux.sys_readv, eax );
    mov( fd, ebx );
    mov( vector, ecx );
    mov( count, edx );
    int( $80 );
    linux.popregs;

end readv;

// writev - Scatter/gather output operation.

procedure linux.writev( fd:dword; var vector:var; count:int32 );
    @nodisplay;
begin writev;

    linux.pushregs;
    mov( linux.sys_writev, eax );
    mov( fd, ebx );
    mov( vector, ecx );
    mov( count, edx );
    int( $80 );
    linux.popregs;

end writev;
```

DESCRIPTION

linux.readv reads data from file descriptor fd, and puts the result in the buffers described by vector. The number of buffers is specified by count. The buffers are filled in the order specified. Operates just like linux.read except that data is put in vector instead of a contiguous buffer.

linux.writev writes data to file descriptor fd, and from the buffers described by vector. The number of buffers is specified by count. The buffers are used in the order specified. Operates just like linux.write except that data is taken from vector instead of a contiguous buffer.

RETURN VALUE

On success readv returns the number of bytes read. On success writev returns the number of bytes written. On error, EAX contains the appropriate error code.

ERRORS

errno.einval An invalid argument was given. For instance count might be greater than linux.max_iovec, or zero. fd could also be attached to an object which is unsuitable for reading (for readv) or writing (for writev).

errno.efault "Segmentation fault." Most likely vector or some of the iov_base pointers points to memory that is not properly allocated.

errno.ebadf The file descriptor fd is not valid.

errno.eintr The call was interrupted by a signal before any data was read/written.

errno.eagain Non-blocking I/O has been selected using linux.o_nonblock and no data was immediately available for reading. (Or the file descriptor fd is for an object that is locked.)

errno.eisdir fd refers to a directory.

errno.eopnotsup fd refers to a socket or device that does not support reading/writing.

errno.enomem Insufficient kernel memory was available.

Other errors may occur, depending on the object connected to fd.

CONFORMING TO

4.4BSD (the readv and writev functions first appeared in BSD 4.2), Unix98. Linux libc5 uses size_t as the type of the count parameter, which is logical but non-standard.

SEE ALSO

read(2), write(2), fprintf(3), fscanf(3)

3.86 reboot

```
// reboot: Reboots the system.

procedure linux.reboot( magic:dword; magic2:dword; flag:dword; var arg:var );
    @nodisplay;
begin reboot;

    linux.pushregs;
    mov( linux.sys_reboot, eax );
    mov( magic, ebx );
    mov( magic2, ecx );
    mov( flag, edx );
    mov( arg, esi );
    int( $80 );
    linux.popregs;

end reboot;
```

DESCRIPTION

The reboot call reboots the system, or enables/disables the reboot keystroke (abbreviated CAD, since the default is Ctrl-Alt-Delete; it can be changed using loadkeys(1)).

This system call will fail (with errno.einval) unless magic equals LINUX_REBOOT_MAGIC1 (that is, $fee1_dead) and magic2 equals LINUX_REBOOT_MAGIC2 (that is, 672274793). However, since 2.1.17 also LINUX_REBOOT_MAGIC2A (that is, 85072278) and since 2.1.97 also LINUX_REBOOT_MAGIC2B (that is, 369367448) are permitted as value for magic2. (The hexadecimal values ofthese constants are meaningful.) The flag argument can have the following values:

linux.reboot_cmd_restart The message `Restarting system. 'is printed, and a default restart is performed immediately. If not preceded by a sync(2), data will be lost.

linux.reboot_cmd_halt The message `System halted.' is printed, and the system is halted. Control is given to the ROM monitor, if there is one. If not preceded by a sync(2), data will be lost.

linux.reboot_cmd_power_off The message `Power down.' is printed, the system is stopped, and all power is removed from the system, if possible. If not preceded by a sync(2), data will be lost.

linux.reboot_cmd_restart2 The message `Restarting system with command '%s'' is printed, and a restart (using the command string given in arg) is performed immediately. If not preceded by a sync(2), data will be lost.

linux.reboot_cmd_cad_on CAD is enabled. This means that the CAD keystroke will immediately cause the action associated to linux.reboot_cmd_restart.

linux.reboot_cmd_cad_off CAD is disabled. This means that the CAD keystroke will cause a signals.sigint signal to be sent to init (process 1), whereupon this process may decide upon a proper action (maybe: kill all processes, sync, reboot).

Only the super-user may use this function.

The precise effect of the above actions depends on the architecture. For the i386 architecture, the additional argument does not do anything at present (2.1.122), but the type of reboot can be determined by kernel command line arguments (`reboot=...') to be either warm or cold, and either hard or through the BIOS.

RETURN VALUE

On success, zero is returned. On error, EAX contains an appropriate negative error code

ERRORS

errno.einval Bad magic numbers or flag.

errno.eperm A non-root user attempts to call reboot.

CONFORMING TO

reboot is Linux specific, and should not be used in programs intended to be portable.

SEE ALSO

sync(2), bootparam(7), ctrlaltdel(8), halt(8), reboot(8)

3.87 rename

```
// rename - renames a file.

procedure linux.rename( oldpath:string; newpath:string );
    @nodisplay;
begin rename;

    linux.pushregs;
    mov( linux.sys_rename, eax );
    mov( oldpath, ebx );
    mov( newpath, ecx );
    int( $80 );
    linux.popregs;

end rename;
```

DESCRIPTION

rename renames a file, moving it between directories if required.

Any other hard links to the file (as created using link(2)) are unaffected.

If newpath already exists it will be atomically replaced (subject to a few conditions - see ERRORS below), so that there is no point at which another process attempting to access newpath will find it missing.

If newpath exists but the operation fails for some reason rename guarantees to leave an instance of newpath in place.

However, when overwriting there will probably be a window in which both oldpath and newpath refer to the file being renamed.

If oldpath refers to a symbolic link the link is renamed; if newpath refers to a symbolic link the link will be overwritten.

RETURN VALUE

On success, zero is returned. On EAX contains an appropriate negative error code.

ERRORS

errno.eisdir newpath is an existing directory, but oldpath is not a directory.

errno.exdev oldpath and newpath are not on the same filesystem.

errno.enotempty or

errno.eexist newpath is a non-empty directory, i.e., contains entries other than "." and "..".

errno.ebusy The rename fails because oldpath or newpath is a directory that is in use by some process (perhaps as current working directory, or as root directory, or because it was open for reading) or is in use by the system (for example as mount point), while the system considers this an error. (Note that there is no requirement to return errno.ebusy in such cases- there is nothing wrong with doing the rename anyway - but it is allowed to return errno.ebusy if the system cannot otherwise handle such situations.)

errno.einval The new pathname contained a path prefix of the old, or, more generally, an attempt was made to make a directory a subdirectory of itself.

errno.emlink oldpath already has the maximum number of links to it, or it was a directory and the directory containing newpath has the maximum number of links.

errno.enotdir A component used as a directory in oldpath or newpath is not, in fact, a directory. Or, oldpath is a directory, and newpath exists but is not a directory.

errno.efault oldpath or newpath points outside your accessible address space.

errno.eacces Write access to the directory containing oldpath or newpath is not allowed for the process's effective uid, or one of the directories in oldpath or newpath did not allow search (execute) permission, or oldpath was a directory and did not allow write permission (needed to update the .. entry).

errno.eperm or

errno.eacces The directory containing oldpath has the sticky bit set and the process's effective uid is neither that of root nor the uid of the file to be deleted nor that of the directory containing it, or newpath is an existing file and the directory containing it has the sticky bit set and the process's effective uid is neither that of root nor the uid of the file to be replaced nor that of the directory containing it, or the filesystem containing pathname does not support renaming of the type requested.

errno. enametoolong oldpath or newpath was too long.

errno.enoent A directory component in oldpath or newpath does not exist or is a dangling symbolic link.

errno.enomem Insufficient kernel memory was available.

errno.erofs The file is on a read-only filesystem.

errno. eloop Too many symbolic links were encountered in resolving oldpath or newpath.

errno.enospc The device containing the file has no room for the new directory entry.

CONFORMING TO

POSIX, 4.3BSD, ANSI C

BUGS

On NFS filesystems, you can not assume that if the operation failed the file was not renamed. If the server does

the rename operation and then crashes, the retransmitted RPC which will be processed when the server is up again causes a failure. The application is expected to deal with this. See link(2) for a similar problem.

SEE ALSO

link(2), unlink(2), symlink(2), mv(1)

3.88 rmdir

```
// rmdir - removes a directory.

procedure linux.rmdir( pathname:string );
   @nodisplay;
begin rmdir;

   linux.pushregs;
   mov( linux.sys_rmdir, eax );
   mov( pathname, ebx );
   int( $80 );
   linux.popregs;

end rmdir;
```

DESCRIPTION

rmdir deletes a directory, which must be empty.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.eperm The filesystem containing pathname does not support the removal of directories.

errno.efault pathname points outside your accessible address space.

errno.eacces Write access to the directory containing pathname was not allowed for the process's effective uid, or one of the directories in pathname did not allow search (execute) permission.

errno.eperm The directory containing pathname has the sticky-bit (linux.s_isvtx) set and the process's effective uid is neither the uid of the file to be deleted nor that of the directory containing it.

errno.enametoolong pathname was too long.

errno.enoent A directory component in pathname does not exist or is a dangling symbolic link.

errno. enotdir pathname, or a component used as a directory in pathname, is not, in fact, a directory.

errno.enotempty pathname contains entries other than . and .. .

errno.ebusy pathname is the current working directory or root directory of some process.

errno.enomem Insufficient kernel memory was available.

errno.erofs pathname refers to a file on a read-only filesystem.

errno.eloop Too many symbolic links were encountered in resolving pathname.

CONFORMING TO

SVr4, SVID, POSIX, BSD 4.3

BUGS

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of directories which are still being used.

SEE ALSO

rename(2), mkdir(2), chdir(2), unlink(2), rmdir(1), rm(1)

3.89 sched_getparam, sched_getparam

```
// sched_getparam - Gets scheduling parameters for a process.

procedure linux.sched_getparam( pid:linux.pid_t; var p:linux.sched_param_t );
    @nodisplay;
begin sched_getparam;

    linux.pushregs;
    mov( linux.sys_sched_getparam, eax );
    mov( pid, ebx );
    mov( p, ecx );
    int( $80 );
    linux.popregs;

end sched_getparam;

// sched_setparam - Sets scheduling parameters for a process.

procedure linux.sched_setparam( pid:linux.pid_t; var p:linux.sched_param_t );
    @nodisplay;
begin sched_setparam;

    linux.pushregs;
    mov( linux.sys_sched_setparam, eax );
    mov( pid, ebx );
    mov( p, ecx );
    int( $80 );
    linux.popregs;

end sched_setparam;
```

DESCRIPTION

linux.sched_setparam sets the scheduling parameters associated with the scheduling policy for the process identified by pid. If pid is zero, then the parameters of the current process are set. The interpretation of the parameter p depends on the selected policy. Currently, the following three scheduling policies are supported under Linux:

- linux.sched_fifo,
- linux.sched_rr, and
- linxux.sched_other.

linux.sched_getparam retrieves the scheduling parameters for the process identified by pid. If pid is zero, then the parameters of the current process are retrieved.

sched_setparam checks the validity of p for the scheduling policy of the process. The parameter p->sched_priority must lie within the range given by sched_get_priority_min and sched_get_priority_max.

RETURN VALUE

On success, sched_setparam and sched_getparam return 0. On error, EAX contains an appropriate negative error code.

ERRORS

errno.esrch	The process whose ID is pid could not be found.

errno.eperm The calling process does not have appropriate privileges. The process calling sched_setparam needs an effective uid equal to the euid or uid of the process identified by pid, or it must be a superuser process.

errno.einval The parameter p does not make sense for the current scheduling policy.

CONFORMING TO

POSIX.1b (formerly POSIX.4)

SEE ALSO

sched_setscheduler(2), sched_getscheduler(2), sched_get_priority_max(2), sched_get_priority_min(2), nice(2), setpriority(2), getpriority(2),

sched_setscheduler(2) has a description of the Linux scheduling scheme.

Programming for the real world - POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0 IEEE Std 1003.1b-1993 (POSIX.1b standard) ISO/IEC 9945-1:1996

3.90 sched_getscheduler, sched_setscheduler

```
// sched_getscheduler - Retrieves scheduling policy for a process.

procedure linux.sched_getscheduler( pid:linux.pid_t );
    @nodisplay;
begin sched_getscheduler;

    linux.pushregs;
    mov( linux.sys_sched_getscheduler, eax );
    mov( pid, ebx );
    int( $80 );
    linux.popregs;

end sched_getscheduler;

// sched_setscheduler - Sets scheduling policy for a process.

procedure linux.sched_setscheduler
(
        pid      :linux.pid_t;
        policy:dword;
    var p      :linux.sched_param_t
);
    @nodisplay;
begin sched_setscheduler;

    linux.pushregs;
    mov( linux.sys_sched_setscheduler, eax );
    mov( pid, ebx );
    mov( policy, ecx );
    mov( p, edx );
    int( $80 );
    linux.popregs;

end sched_setscheduler;
```

DESCRIPTION

linux.sched_setscheduler sets both the scheduling policy and the associated parameters for the process identified by pid. If pid equals zero, the scheduler of the calling process will be set. The interpretation of the parameter p depends on the selected policy. Currently, the following three scheduling policies are supported under Linux: linux.sched_fifo, linux.sched_rr, and linux.sched_other; their respective semantics is described below.

sched_getscheduler queries the scheduling policy currently applied to the process identified by pid. If pid equals zero, the policy of the calling process will be retrieved.

Scheduling Policies

The scheduler is the kernel part that decides which runnable process will be executed by the CPU next. The Linux scheduler offers three different scheduling policies, one for normal processes and two for real-time applications. A static priority value sched_priority is assigned to each process and this value can be changed only via system calls. Conceptually, the scheduler maintains a list of runnable processes for each possible sched_priority value, and sched_priority can have a value in the range 0 to 99. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines for each process, where it will be inserted into the list of processes with equal static priority and how it will move inside this list.

linux.sched_other is the default universal time-sharing scheduler policy used by most processes, linux.sched_fifo and linux.sched_rr are intended for special time-critical applications that need precise control over the way in which runnable processes are selected for execution. Processes scheduled with linux.sched_other must be assigned the static priority 0, processes scheduled under linux.sched_fifo or linux.sched_rr can have a static priority in the range 1 to 99. Only processes with superuser privileges can get a static priority higher than 0 and can therefore be scheduled under linux.sched_fifo or linux.sched_rr. The system calls sched_get_priority_min and sched_get_priority_max can be used to to find out the valid priority range for a scheduling policy in a portable way on all POSIX.1b conforming systems.

All scheduling is preemptive: If a process with a higher static priority gets ready to run, the current process will be preempted and returned into its wait list. The scheduling policy only determines the ordering within the list of runnable processes with equal static priority.

linux.sched_fifo

First In-First out scheduling linux.sched_fifo can only be used with static priorities higher than 0, that means that when a linux.sched_fifo processes becomes runnable, it will always preempt immediately any currently running normal linux.sched_other process. linux.sched_fifo is a simple scheduling algorithm without time slicing. For processes scheduled under the linux.sched_fifo policy, the following rules are applied: A linux.sched_fifo process that has been preempted by another process of higher priority will stay at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again. When a linux.sched_fifo process becomes runnable, it will be inserted at the end of the list for its priority. A call to linux.sched_setscheduler or linux.sched_setparam will put the linux.sched_fifo process identified by pid at the end of the list if it was runnable. A process calling linux.sched_yield will be put at the end of the list. No other events will move a process scheduled under the linux.sched_fifo policy in the wait list of runnable processes with equal static priority. A linux.sched_fifo process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, or it calls linux.sched_yield.

linux.sched_rr

Round Robin scheduling linux.sched_rr is a simple enhancement of linux.sched_fifo. Everything described above for linux.sched_fifo also applies to linux.sched_rr, except that each process is only allowed to run for a maximum time quantum. If a linux.sched_rr process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A linux.sched_rr process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum. The length of the time quantum can be retrieved by linux.sched_rr_get_interval.

linux.sched_other Default Linux time-sharing scheduling linux.sched_other can only be used at static priority 0. linux.sched_other is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice level (set by the nice or setpriority system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all linux.sched_other processes.

Response time

A blocked high priority process waiting for the I/O has a certain response time before it is scheduled again. The device driver writer can greatly reduce this response time by using a "slow interrupt" interrupt handler as described in request_irq(9).

Miscellaneous

Child processes inherit the scheduling algorithm and parameters across a fork.

Memory locking is usually needed for real-time processes to avoid paging delays, this can be done with mlock or mlockall.

As a non-blocking end-less loop in a process scheduled under linux.sched_fifo or linux.sched_rr will block all processes with lower priority forever, a software developer should always keep available on the console a shell scheduled under a higher static priority than the tested application. This will allow an emergency kill of tested real-time applications that do not block or terminate as expected. As linux.sched_fifo and linux.sched_rr processes can preempt other processes forever, only root processes are allowed to activate these policies under Linux.

RETURN VALUE

On success, linux.sched_setscheduler returns zero. On success, linux.sched_getscheduler returns the policy for the process (a non-negative integer). On error, EAX will contain a negative error code

ERRORS

errno.esrch The process whose ID is pid could not be found.

errno.eperm The calling process does not have appropriate privileges. Only root processes are allowed to activate the linux.sched_fifo and linux.sched_rr policies. The process calling linux.sched_setscheduler needs an effective uid equal to the euid or uid of the process identified by pid, or it must be a superuser process.

errno.einval The scheduling policy is not one of the recognized policies, or the parameter p does not make sense for the policy.

CONFORMING TO

POSIX.1b (formerly POSIX.4)

BUGS

As of linux-1.3.81, linux.sched_rr has not yet been tested carefully and might not behave exactly as described or required by POSIX.1b.

NOTE

Standard Linux is a general-purpose operating system and can handle background processes, interactive applications, and soft real-time applications (applications that need to usually meet timing deadlines). This man page is directed at these kinds of applications.

Standard Linux is not designed to support hard real-time applications, that is, applications in which deadlines (often much shorter than a second) must be guaranteed or the system will fail catastrophically. Like all general-purpose operating systems, Linux is designed to maximize average case performance instead of worst case performance. Linux's worst case performance for interrupt handling is much poorer than its average case, its various kernel locks (such as for SMP) produce long maximum wait times, and many of its performance improvement techniques decrease average time by increasing worst-case time. For most situations, that's what you want, but if you truly are developing a hard real-time application, consider using hard real-time extensions to Linux such as RTLinux (http://www.rtlinux.org) or use a different operating system designed specifically for hard real-time applications.

SEE ALSO

sched_setparam(2), sched_getparam(2), sched_yield(2), sched_get_priority_max(2), sched_get_priority_min(2), nice(2), setpriority(2), getpriority(2), mlockall(2), munlockall(2), mlock(2), munlock(2)

Programming for the real world - POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0 IEEE Std 1003.1b-1993 (POSIX.1b standard) ISO/IEC9945-1:1996 -This is the new 1996 revision of POSIX.1 which contains in one single standard POSIX.1(1990), POSIX.1b(1993), POSIX.1c(1995), and POSIX.1i(1995).

3.91 sched_get_priority_max, sched_get_priority_min

```
// sched_get_priority_max - Retrieves the maximum priority value.

procedure linux.sched_get_priority_max( policy:dword );
    @nodisplay;
begin sched_get_priority_max;

    linux.pushregs;
    mov( linux.sys_sched_get_priority_max, eax );
    mov( policy, ebx );
    int( $80 );
    linux.popregs;

end sched_get_priority_max;

// sched_get_priority_min - Retrieves the minimum priority value.

procedure linux.sched_get_priority_min( policy:dword );
    @nodisplay;
begin sched_get_priority_min;

    linux.pushregs;
    mov( linux.sys_sched_get_priority_min, eax );
    mov( policy, ebx );
    int( $80 );
    linux.popregs;

end sched_get_priority_min;
```

DESCRIPTION

linux.sched_get_priority_max returns the maximum priority value that can be used with the scheduling algorithm identified by policy. sched_get_priority_min returns the minimum priority value that can be used with the scheduling algorithm identified by policy. Supported policy values are linux.sched_fifo, linux.sched_rr, and linux.sched_other.

Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. Thus, the value returned by sched_get_priority_max will be greater than the value returned by sched_get_priority_min.

Linux allows the static priority value range 1 to 99 for linux.sched_fifo and linux.sched_rr and the priority 0 for linux.sched_other. Scheduling priority ranges for the various policies are not alterable.

The range of scheduling priorities may vary on other POSIX systems, thus it is a good idea for portable applications to use a virtual priority range and map it to the interval given by sched_get_priority_max and sched_get_priority_min. POSIX.1b requires a spread of at least 32 between the maximum and the minimum values for linux.sched_fifo and linux.sched_rr.

RETURN VALUE

On success, sched_get_priority_max and sched_get_priority_min return the maximum/minimum priority value for the named scheduling policy. On error, EAX contains a negative error code.

ERRORS

errno.einval	The parameter policy does not identify a defined scheduling policy.

CONFORMING TO

POSIX.1b (formerly POSIX.4)

SEE ALSO

sched_setscheduler(2), sched_getscheduler(2), sched_setparam(2), sched_getparam(2) sched_setscheduler(2) has a description of the Linux scheduling scheme.

Programming for the real world - POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0 IEEE Std 1003.1b-1993 (POSIX.1b standard) ISO/IEC 9945-1:1996

3.92 sched_rr_get_interval

```
// sched_rr_get_interval - Retrieves the timeslice interval.

procedure linux.sched_rr_get_interval( pid:linux.pid_t; var tp:linux.timespec );
    @nodisplay;
begin sched_rr_get_interval;

    linux.pushregs;
    mov( linux.sys_sched_rr_get_interval, eax );
    mov( pid, ebx );
    mov( tp, ecx );
    int( $80 );
    linux.popregs;

end sched_rr_get_interval;
```

DESCRIPTION

linux.sched_rr_get_interval writes into the timespec structure pointed to by tp the round robin time quantum for the process identified by pid. If pid is zero, the time quantum for the calling process is written into *tp. The identified process should be running under the linux.sched_rr scheduling policy.

The round robin time quantum value is not alterable under Linux 1.3.81.

RETURN VALUE

On success, linux.sched_rr_get_interval returns 0. On error, it returns a negative error code in EAX.

ERRORS

errno.esrch	The process whose ID is pid could not be found.
errno.enosys	The system call is not yet implemented.

CONFORMING TO

POSIX.1b (formerly POSIX.4)

BUGS

As of Linux 1.3.81 linux.sched_rr_get_interval returns with error errno.enosys, because linux.sched_rr has not yet been fully implemented and tested properly.

SEE ALSO

sched_setscheduler(2) has a description of the Linux scheduling scheme.

Programming for the real world - POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0 IEEE Std 1003.1b-1993 (POSIX.1b standard, formerly POSIX.4) ISO/IEC 9945-1:1996

3.93 sched_setparam

See “sched_getparam, sched_getparam” on page 132.

3.94 sched_setscheduler

See “sched_getscheduler, sched_setscheduler” on page 133.

3.95 sched_yield

```
// sched_yield - Yields the time quantum.

procedure linux.sched_yield;
    @nodisplay;
begin sched_yield;

    linux.pushregs;
    mov( linux.sys_sched_yield, eax );
    int( $80 );
    linux.popregs;

end sched_yield;
```

DESCRIPTION

A process can relinquish the processor voluntarily without blocking by calling linux.sched_yield. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Note: If the current process is the only process in the highest priority list at that time, this process will continue to run after a call to linux.sched_yield.

RETURN VALUE

On success, sched_yield returns 0. On error, EAX will contain a negative error code.

CONFORMING TO

POSIX.1b (formerly POSIX.4)

SEE ALSO

sched_setscheduler(2) for a description of Linux scheduling.

Programming for the real world - POSIX.4 by Bill O. Gallmeister, O'Reilly & Associates, Inc., ISBN 1-56592-074-0 IEEE Std 1003.1b-1993 (POSIX.1b standard) ISO/IEC 9945-1:1996

3.96 select

```
// select - polls devices.

procedure linux.select
(
        n            :int32;
    var readfds      :linux.fd_set;
    var writefds     :linux.fd_set;
    var  exceptfds   :linux.fd_set;
    var  timeout     :linux.timespec;
    var  sigmask     :linux.sigset_t
);
    @nodisplay;
begin select;

    linux.pushregs;
    mov( linux.sys_select, eax );
    mov( n, ebx );
    mov( readfds, ecx );
    mov( exceptfds, edx );
    mov( timeout, esi );
    mov( sigmask, edi );
    int( $80 );
    linux.popregs;

end select;
```

DESCRIPTION

The select function waits for a number of file descriptors to change status.

Things to note about select:

(i) The select function uses a timeout that is a struct timeval (with seconds and microseconds)

(ii) The select function may update the timeout parameter to indicate how much time was left.

Three independent sets of descriptors are watched. Those listed in readfds will be watched to see if characters become available for reading (more precisely, to see if a read will not block - in particular, a file descriptor is also ready on end-of-file), those in writefds will be watched to see if a write will not block, and those in exceptfds will be watched for exceptions. On exit, the sets are modified in place to indicate which descriptors actually changed status.

Note that you may use the BTS, BTC, and BTR instructions to manipulate the sets.

n is the highest-numbered descriptor in any of the three sets, plus 1.

timeout is an upper bound on the amount of time elapsed before select returns. It may be zero, causing select to return immediately. (This is useful for polling.) If timeout is NULL (no timeout), select can block indefinitely.

RETURN VALUE

On success, select returns the number of descriptors contained in the descriptor sets, which may be zero if the timeout expires before anything interesting happens. On error, -1 is returned, and EAX will contain a negative error code; the sets and timeout become undefined, so do not rely on their contents after an error.

ERRORS

errno.ebadf	An invalid file descriptor was given in one of the sets.
errno.eintr	A non blocked signal was caught.
errno.einval	n is negative.
errno.enomem	select was unable to allocate memory for internal tables.

NOTES

Some code calls select with all three sets empty, n zero, and a non-null timeout as a fairly portable way to sleep with subsecond precision.

On Linux, timeout is modified to reflect the amount of time not slept; most other implementations do not do this. This causes problems both when Linux code which reads timeout is ported to other operating systems, and when code is ported to Linux that reuses a struct timeval for multiple selects in a loop without reinitializing it. Consider timeout to be undefined after select returns.

CONFORMING TO

4.4BSD (the select function first appeared in 4.2BSD). Generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants).However, note that the System V variant typically sets the timeout variable before exit, but the BSD variant does not.

SEE ALSO

accept(2), connect(2), poll(2), read(2), recv(2), send(2), sigprocmask(2), write(2)

3.97 semctl

```
// semctl - SysV semaphore operation.

procedure linux.semctl( semid:dword; semnum:int32; cmd:dword; arg:linux.semun );
    @nodisplay;
begin semctl;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_semctl, ebx );
    mov( semid, ecx );
    mov( semnum, edx );
    mov( cmd, esi );
    mov( arg, edi );
    int( $80 );
    linux.popregs;

end semctl;
```

DESCRIPTION

The function performs the control operation specified by cmd on the semaphore set (or on the semnum-th semaphore of the set) identified by semid. The first semaphore of the set is indicated by a value 0 for semnum.

Legal values for cmd are

linux.ipc_stat Copy info from the semaphore set data structure into the structure pointed to by arg.buf. The argument semnum is ignored. The calling process must have read access privileges on the semaphore set.

linux.ipc_set Write the values of some members of the semid_ds structure pointed to by arg.buf to the semaphore set data structure, updating also its sem_ctime member. Considered members from the user supplied struct semid_ds pointed to by arg.buf are

sem_perm.uid

sem_perm.gid

sem_perm.mode /* only lowest 9-bits */

The calling process effective user-ID must be one among super-user, creator or owner of the semaphore set. The argument semnum is ignored.

linux.ipc_rmid Remove immediately the semaphore set and its data structures awakening all waiting processes (with an error return and EAX set to errno.eidrm). The calling process effective user-ID must be one among super-user, creator or owner of the semaphore set. The argument semnum is ignored.

linux.getall Return semval for all semaphores of the set into arg.array. The argument semnum is ignored. The calling process must have read access privileges on the semaphore set.

linux.getncnt The system call returns the value of semncnt for the semnum-th semaphore of the set (i.e. the number of processes waiting for an increase of semval for the semnum-th semaphore of the set). The calling process must have read access privileges on the semaphore set.

linux.getpid The system call returns the value of sempid for the semnum-th semaphore of the set (i.e. the pid of the process that executed the last linux.semop call for the semnum-th semaphore of the set). The calling process must have read access privileges on the semaphore set.

linux.getval The system call returns the value of semval for the semnum-th semaphore of the set. The calling process must have read access privileges on the semaphore set.

linux.getzcnt The system call returns the value of semzcnt for the semnum-th semaphore of the set (i.e. the number of processes waiting for semval of the semnum-th semaphore of the set to become 0). The calling process must have read access privileges on the semaphore set.

linux. setall Set semval for all semaphores of the set using arg.array, updating also the sem_ctime member of the semid_ds structure associated to the set. Undo entries are cleared for altered semaphores in all processes. Processes sleeping on the wait queue are awakened if some semval becomes 0 or increases. The argument semnum is ignored. The calling process must have alter access privileges on the semaphore set.

linux.setval Set the value of semval to arg.val for the semnum-th semaphore of the set, updating also the sem_ctime member of the semid_ds structure associated to the set. Undo entry is cleared for altered semaphore in all processes. Processes sleeping on the wait queue are awakened if semval becomes 0 or increases. The calling process must have alter access privileges on the semaphore set.

RETURN VALUE

On fail the system call returns EAX containing a negative error code. Otherwise the system call returns a non-negative value depending on cmd as follows:

linux.sem_getncnt	the value of semncnt.
linux.sem_getpid	the value of sempid.
linux.sem_getval	the value of semval.
linux.sem_getzcnt	the value of semzcnt.

ERRORS

For a failing return, EAX will be set to one among the following values:

errno.eacces The calling process has no access permissions needed to execute cmd.

errno.efault The address pointed to by arg.buf or arg.array isn't accessible.

errno.eidrm The semaphore set was removed.

errno.einval Invalid value for cmd or semid.

errno.eperm The argument cmd has value IPC_SET or IPC_RMID but the calling process effective user-ID has insufficient privileges to execute the command.

errno.erange The argument cmd has value SETALL or SETVAL and the value to which semval has to be set (for some semaphore of the set) is less than 0 or greater than the implementation value SEMVMX.

NOTES

The IPC_INFO, SEM_STAT and SEM_INFO control calls are used by the ipcs(8) program to provide information on allocated resources. In the future these can be modified as needed or moved to a proc file system interface.

Various fields in a struct semid_ds were shorts under Linux 2.2 and have become longs under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by a IPC_64 flag in cmd.)

The following system limit on semaphore sets affects a semctl call:

linux.semvmx Maximum value for semval: implementation dependent (32767).

CONFORMING TO

SVr4, SVID. SVr4 documents more error conditions errno.einval and errno.eoverflow.

SEE ALSO

ipc(5), shmget(2), shmat(2), shmdt(2)

3.98 semget

```
// semget - SysV semaphore operation.

procedure linux.semget( key:linux.key_t; nsyms:int32; semflg:dword );
    @nodisplay;
begin semget;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_semget, ebx );
    mov( key, ecx );
    mov( nsyms, edx );
    mov( semflg, esi );
    int( $80 );
    linux.popregs;

end semget;
```

DESCRIPTION

The function returns the semaphore set identifier associated to the value of the argument key. A new set of nsems semaphores is created if key has value IPC_PRIVATE or key isn't IPC_PRIVATE, no existing semaphore set is associated to key, and IPC_CREAT is asserted in semflg (i.e. semflg & IPC_CREAT isn't zero). The presence in semflg of the fields IPC_CREAT and IPC_EXCL plays the same role, with respect to the existence of the semaphore set, as the presence of O_CREAT and O_EXCL in the mode argument of the open(2) system call: i.e. the semget function fails if semflg asserts both IPC_CREAT and IPC_EXCL and a semaphore set already exists for key.

Upon creation, the lower 9 bits of the argument semflg define the access permissions (for owner, group and others) to the semaphore set in the same format, and with the same meaning, as for the access permissions parameter in the open(2) or creat(2) system calls (though the execute permissions are not used by the system, and write permis sions, for a semaphore set, effectively means alter permissions).

Furthermore, while creating, the system call initializes the system semaphore set data structure semid_ds as follows:

- sem_perm.cuid and sem_perm.uid are set to the effective user-ID of the calling process.
- sem_perm.cgid and sem_perm.gid are set to the effective group-ID of the calling process.
- The lowest order 9 bits of sem_perm.mode are set to the lowest order 9 bit of semflg.
- sem_nsems is set to the value of nsems.
- sem_otime is set to 0.
- sem_ctime is set to the current time.

The argument nsems can be 0 (a don't care) when the system call isn't a create one. Otherwise nsems must be greater than 0 and less or equal to the maximum number of semaphores per semid, (SEMMSL).

If the semaphore set already exists, the access permissions are verified, and a check is made to see if it is marked for destruction.

RETURN VALUE

If successful, the return value will be the semaphore set identifier (a positive integer), otherwise EAX will contain a negative error code.

ERRORS

For a failing return, EAX will be set to one among the following values:

errno.eacces A semaphore set exists for key, but the calling process has no access permissions to the set.

errno.eexist A semaphore set exists for key and semflg was asserting both IPC_CREAT and IPC_EXCL.

errno.eidrm The semaphore set is marked as to be deleted.

errno.enoent No semaphore set exists for key and semflg wasn't asserting IPC_CREAT.

errno.enomem A semaphore set has to be created but the system has not enough memory for the new data structure.

errno.enospc A semaphore set has to be created but the system limit for the maximum number of semaphore sets (SEMMNI), or the system wide maximum number of semaphores (SEMMNS), would be exceeded.

NOTES

linux.ipc_private isn't a flag field but a linux.key_t type. If this special value is used for key, the system call ignores everything but the lowest order 9 bits of semflg and creates a new semaphore set (on success). The following are limits on semaphore set resources affecting a semget call:

linux.semmni System wide maximum number of semaphore sets: policy dependent.

linux.semmsl Maximum number of semaphores per semid: implementation dependent (500 currently).

linux.semmns System wide maximum number of semaphores: policy dependent. Values greater than (linux.semmsl * linux.semmni) makes it irrelevant.

BUGS

Use of IPC_PRIVATE doesn't inhibit to other processes the access to the allocated semaphore set.

As for the files, there is currently no intrinsic way for a process to ensure exclusive access to a semaphore set. Asserting both linux.ipc_creat and linux.ipc_excl in semflg only ensures (on success) that a new semaphore set will be created, it doesn't imply exclusive access to the semaphore set.

The data structure associated with each semaphore in the set isn't initialized by the system call. In order to initialize those data structures, one has to execute a subsequent call to semctl(2) to perform a linux.sem_setval or a linux.sem_setall command on the semaphore set.

CONFORMING TO

SVr4, SVID. SVr4 documents additional error conditions EINVAL, EFBIG, E2BIG, EAGAIN, ERANGE, EFAULT.

SEE ALSO

ftok(3), ipc(5), semctl(2), semop(2)

3.99 semop

```
// semop - SysV semaphore operation.

procedure linux.semop( semid:dword; var sops:linux.sembuf; nsops:dword );
    @nodisplay;
begin semop;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_semop, ebx );
    mov( semid, ecx );
    mov( sops, edx );
    mov( nsops, esi );
    int( $80 );
    linux.popregs;

end semop;
```

DESCRIPTION

The function performs operations on selected members of the semaphore set indicated by semid. Each of the nsops elements in the array pointed to by sops specify an operation to be performed on a semaphore by a struct sembuf including the following members:

```
sem_num   :word;         /* semaphore number: 0 = first */
sem_op    :word;         /* semaphore operation */
sem_flg   :word;         /* operation flags */
```

Flags recognized in sem_flg are linux.ipc_nowait and linux.sem_undo. If an operation asserts linux.sem_undo, it will be undone when the process exits.

The system call semantic assures that the operations will be performed if and only if all of them will succeed. Each operation is performed on the sem_num-th semaphore of the semaphore set - where the first semaphore of the set is semaphore 0 - and is one among the following three.

If sem_op is a positive integer, the operation adds this value to semval. Furthermore, if linux.sem_undo is asserted for this operation, the system updates the process undo count for this semaphore. The operation always goes through, so no process sleeping can happen. The calling process must have alter permissions on the semaphore set. If sem_op is zero, the process must have read access permissions on the semaphore set. If semval is zero, the operation goes through. Otherwise, if linux.ipc_nowait is asserted in sem_flg, the system call fails (undoing all previous actions performed) with errno set to errno.eagain. Otherwise semzcnt is incremented by one and the process sleeps until one of the following occurs:

- semval becomes 0, at which time the value of semzcnt is decremented.
- The semaphore set is removed: the system call fails with errno set to errno.eidrm.
- The calling process receives a signal that has to be caught: the value of semzcnt is decremented and the system call fails with errno set to errno.eintr.

If sem_op is less than zero, the process must have alter permissions on the semaphore set. If semval is greater than or equal to the absolute value of sem_op, the absolute value of sem_op is subtracted by semval. Furthermore, if linux.sem_undo is asserted for this operation, the system updates the process undo count for this semaphore. Then the operation goes through. Otherwise, if linux.ipc_nowait is asserted in sem_flg, the system call fails (undoing all previous actions performed) with errno set to errno.eagain. Otherwise semncnt is incremented by one and the process sleeps until one of the following occurs:

- semval becomes greater or equal to the absolute value of sem_op, at which time the value of semncnt is decremented, the absolute value of sem_op is subtracted from semval and, if linux.sem_undo is asserted for this operation, the system updates the process undo count for this semaphore.

- The semaphore set is removed from the system: the system call fails with errno set to linux.eidrm.
- The calling process receives a signal that has to be caught: the value of semncnt is decremented and the system call fails with errno set to errno.eintr.

In case of success, the sempid member of the structure sem for each semaphore specified in the array pointed to by sops is set to the process-ID of the calling process. Furthermore both sem_otime and sem_ctime are set to the current time.

RETURN VALUE

If successful the system call returns 0, otherwise it returns a negative error code in EAX.

ERRORS

For a failing return, errno will be set to one among the following values:

errno.e2big The argument nsops is greater than linux.semopm, the maximum number of operations allowed per system call.

errno.eacces The calling process has no access permissions on the semaphore set as required by one of the specified operations.

errno.eagain An operation could not go through and linux.ipc_nowait was asserted in its sem_flg.

errno. efault The address pointed to by sops isn't accessible.

errno.efbig For some operation the value of sem_num is less than 0 or greater than or equal to the number of semaphores in the set.

errno.eidrm The semaphore set was removed.

errno.eintr Sleeping on a wait queue, the process received a signal that had to be caught.

errno.einval The semaphore set doesn't exist, or semid is less than zero, or nsops has a non-positive value.

errno.enomem The sem_flg of some operation asserted linux.sem_undo and the system has not enough memory to allocate the undo structure.

errno.erange For some operation semop+semval is greater than linux.semvmx, the implementation dependent maximum value for semval.

NOTES

The sem_undo structures of a process aren't inherited by its child on execution of a fork(2) system call. They are instead inherited by the substituting process resulting by the execution of the execve(2) system call.

The followings are limits on semaphore set resources affecting a semop call:

linux.semopm Maximum number of operations allowed for one semop call: policy dependent.

linux.semvmx Maximum allowable value for semval: implementation dependent (32767).

The implementation has no intrinsic limits for the adjust on exit maximum value (linux.semaem), the system wide maximum number of undo structures (linux.semmnu) and the per process maximum number of undo entries system parameters.

BUGS

The system maintains a per process sem_undo structure for each semaphore altered by the process with undo requests. Those structures are free at process exit. One major cause for unhappiness with the undo mechanism is that it does not fit in with the notion of having an atomic set of operations an array of semaphores. The undo requests for an array and each semaphore therein may have been accumulated over many semopt calls. Should the process sleep when exiting, or should all undo operations be applied with the linux.ipc_nowait flag in effect? Currently those undo operations which go through immediately are applied, and those that require a wait are ignored silently. Thus harmless undo usage is guaranteed with private semaphores only.

CONFORMING TO

SVr4, SVID. SVr4 documents additional error conditions EINVAL, EFBIG, ENOSPC.

SEE ALSO

ipc(5), semctl(2), semget(2)

3.100 sendfile

```
// sendfile - Transmits a file to a device.

procedure linux.sendfile
(
        out_fd    :dword;
        in_fd     :dword;
    var offset    :linux.off_t;
        count     :linux.size_t
);
    @nodisplay;
begin sendfile;

    linux.pushregs;
    mov( linux.sys_sendfile, eax );
    mov( out_fd, ebx );
    mov( in_fd, ecx );
    mov( offset, edx );
    mov( count, esi );
    int( $80 );
    linux.popregs;

end sendfile;
```

DESCRIPTION

linux.sendfile copies data between one file descriptor and another. Either or both of these file descriptors may refer to a socket (but see below). in_fd should be a file descriptor opened for reading and out_fd should be a descriptor opened for writing. offset is a pointer to a variable holding the input file pointer position from which sendfile() will start reading data. When sendfile() returns, this variable will be set to the offset of the byte following the last byte that was read. count is the number of bytes to copy between file descriptors.

Because this copying is done within the kernel, sendfile() does not need to spend time transferring data to and from user space.

NOTES

Sendfile does not modify the current file pointer of in_fd, but does for out_fd.

If you plan to use sendfile for sending files to a TCP socket, but need to send some header data in front of the file contents, please see the TCP_CORK option in tcp(7) to minimize the number of packets and to tune performance.

Presently the descriptor from which data is read cannot correspond to a socket, it must correspond to a file which supports mmap()-like operations.

RETURN VALUE

If the transfer was successful, the number of bytes written to out_fd is returned. On error, EAX contains a negative error code.

ERRORS

errno.ebadf	The input file was not opened for reading or the output file was not opened for writing.
errno.einval	Descriptor is not valid or locked.
errno.enomem	Insufficient memory to read from in_fd.
errno.eio	Unspecified error while reading from in_fd.

VERSIONS

sendfile is a new feature in Linux 2.2. The include file <sys/sendfile.h> is present since glibc2.1. Other Unixes often implement sendfile with different semantics and prototypes. It should not be used in portable programs.

SEE ALSO

socket(2), open(2)

3.101 setdomainname

```
// setdomainname-

procedure linux.setdomainname( domainName:string; len:linux.size_t );
    @nodisplay;
begin setdomainname;

    linux.pushregs;

    mov( linux.sys_setdomainname, eax );
    mov( domainName, ebx );
    mov( len, ecx );
    int( $80 );
    linux.popregs;

end setdomainname;
```

DESCRIPTION

linux.setdomainname changes the domain name of the current processor. To obtain this name, see linux.uname.

RETURN VALUE

On success, zero is returned. On error, EAX contains an appropriate, negative, error code.

ERRORS

errno.einval	len was negative or too large.
errno.eperm	the caller was not the superuser.
errno.efault	name pointed outside of user address space.

CONFORMING TO

POSIX does not specify these calls.

SEE ALSO

gethostname(2), sethostname(2), uname(2)

3.102 setfsgid

```
// setfsuid - Sets the GID that Linux uses.

procedure linux.setfsgid( fsgid:linux.gid_t );
    @nodisplay;
begin setfsgid;

    linux.pushregs;
    mov( linux.sys_setfsgid, eax );
    movzx( fsgid, ebx );
    int( $80 );
    linux.popregs;

end setfsgid;
```

DESCRIPTION

setfsgid sets the group ID that the Linux kernel uses to check for all accesses to the file system. Normally, the value of fsgid will shadow the value of the effective group ID. In fact, whenever the effective group ID is changed, fsgid will also be changed to new value of effective group ID.

An explicit call to setfsgid is usually only used by programs such as the Linux NFS server that need to change what group ID is used for file access without a corresponding change in the real and effective group IDs. A change in the normal group IDs for a program such as the NFS server is a security hole that can expose it to unwanted signals from other group IDs.

setfsgid will only succeed if the caller is the superuser or if fsgid matches either the real group ID, effective group ID, saved set-group-ID, or the current value of fsgid.

RETURN VALUE

On success, the previous value of fsgid is returned. On error, the current value of fsgid is returned.

CONFORMING TO

setfsgid is Linux specific and should not be used in programs intended to be portable.

BUGS

No error messages of any kind are returned to the caller. At the very least, errno.eperm should be returned when the call fails.

NOTE

When glibc determines that the argument is not a valid gid, it will return -1 and set errno to errno.einval without attempting the system call.

SEE ALSO

setfsuid(2)

3.103 setfsuid

```
// setfsuid - Sets the UID that Linux uses.

procedure linux.setfsuid( fsuid:linux.uid_t );
    @nodisplay;
begin setfsuid;

    linux.pushregs;
    mov( linux.sys_setfsuid, eax );
    movzx( fsuid, ebx );
    int( $80 );
    linux.popregs;

end setfsuid;
```

DESCRIPTION

setfsuid sets the user ID that the Linux kernel uses to check for all accesses to the file system. Normally, the value of fsuid will shadow the value of the effective user ID. In fact, whenever the effective user ID is changed, fsuid will also be changed to new value of effective user ID.

An explict call to setfsuid is usually only used by programs such as the Linux NFS server that need to change what user ID is used for file access without a corresponding change in the real and effective user IDs. A change in the normal user IDs for a program such as the NFS server is a security hole that can expose it to unwanted signals from other user IDs.

setfsuid will only succeed if the caller is the superuser or if fsuid matches either the real user ID, effective user ID, saved set-user-ID, or the current value of fsuid.

RETURN VALUE

On success, the previous value of fsuid is returned. On error, the current value of fsuid is returned.

CONFORMING TO

setfsuid is Linux specific and should not be used in programs intended to be portable.

BUGS

No error messages of any kind are returned to the caller. At the very least, errno.eperm should be returned when the call fails.

SEE ALSO

setfsgid(2)

3.104 setgid

```
// setgid - sets the effective group ID for the current process.

procedure linux.setgid( gid:linux.gid_t );
    @nodisplay;
begin setgid;

    linux.pushregs;
    mov( linux.sys_setgid, eax );
    movzx( gid, ebx );
    int( $80 );
    linux.popregs;

end setgid;
```

DESCRIPTION

setgid sets the effective group ID of the current process. If the caller is the superuser, the real and saved group ID's are also set.

Under Linux, setgid is implemented like the POSIX version with the _POSIX_SAVED_IDS feature. This allows a setgid (other than root) program to drop all of its group privileges, do some un-privileged work, and then re-engage the original effective group ID in a secure manner.

If the user is root or the program is setgid root, special care must be taken. The setgid function checks the effective gid of the caller and if it is the superuser, all process related group ID's are set to gid. After this has occurred, it is impossible for the program to regain root privileges.

Thus, a setgid-root program wishing to temporarily drop root privileges, assume the identity of a non-root group, and then regain root privileges afterwards cannot use setgid. You can accomplish this with the (non-POSIX, BSD) call setegid.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code,

ERRORS

errno.**eperm**	The user is not the super-user, and gid does not match the effective group ID or saved set-group-ID of the calling process.

CONFORMING TO

SVr4, SVID.

SEE ALSO

getgid(2), setregid(2), setegid(2)

3.105 setgroups

See “getgroups, setgroups” on page 58

3.106 sethostname

```
// sethostname - sets the host name of the current CPU.

procedure linux.sethostname( theName:string; len:linux.size_t );
    @nodisplay;
begin sethostname;

    linux.pushregs;
    mov( linux.sys_sethostname, eax );
    mov( theName, ebx );
    mov( len, ecx );
    int( $80 );
    linux.popregs;

end sethostname;
```

DESCRIPTION

linux.sethostname is used to change the host name of the current processor.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.einval len is negative or len is larger than the maximum allowed size.

errno.eperm The caller was not the superuser.

errno.efault name is an invalid address.

CONFORMING TO

SVr4, 4.4BSD (this function first appeared in 4.2BSD). POSIX.1 does not define these functions, but ISO/IEC 9945-1:1990 mentions them in B.4.4.1.

NOTE

To obtain the host name, see linux.uname.

SEE ALSO

getdomainname(2), setdomainname(2), uname(2)

3.107 setitimer

See “getitimer, setitimer” on page 60.

3.108 setpriority

See “getpriority, setpriority” on page 65.

3.109 setregid, setreuid

```
// setregid - Sets the real and effective group IDs for this process.

procedure linux.setregid( rgid:linux.gid_t; egid:linux.gid_t );
    @nodisplay;
begin setregid;

    linux.pushregs;
    mov( linux.sys_setregid, eax );
    movzx( rgid, ebx );
    movzx( egid, ecx );
    int( $80 );
    linux.popregs;

end setregid;

// setreuid - Sets the real and effective user IDs for this process.

procedure linux.setreuid( ruid:linux.uid_t; euid:linux.uid_t );
    @nodisplay;
begin setreuid;

    linux.pushregs;
    mov( linux.sys_setreuid, eax );
    movzx( ruid, ebx );
    movzx( euid, ecx );
    int( $80 );
    linux.popregs;

end setreuid;
```

DESCRIPTION

linux.setreuid sets real and effective user IDs of the current process. Unprivileged users may only set the real user ID to the real user ID or the effective user ID, and may only set the effective user ID to the real user ID, the effective user ID or the saved user ID.

Supplying a value of -1 for either the real or effective user ID forces the system to leave that ID unchanged.

If the real user ID is set or the effective user ID is set to a value not equal to the previous real user ID, the saved user ID will be set to the new effective user ID.

Completely analogously, linux.setregid sets real and effective group ID's of the current process, and all of the above holds with "group" instead of "user".

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.eperm The current process is not the super-user and changes other than (i) swapping the effective user (group) ID with the real user (group) ID, or (ii) setting one to the value of the other or (iii) setting the effective user (group) ID to the value of the saved user (group) ID was specified.

NOTES

Setting the effective user (group) ID to the saved user ID is possible since Linux 1.1.37 (1.1.38).

CONFORMING TO

BSD 4.3 (the setreuid and setregid function calls first appeared in 4.2BSD).

SEE ALSO

getuid(2), getgid(2), setuid(2), setgid(2), seteuid(2), setresuid(2)

3.110 setresgid, setresuid

```
// setresgid - Sets all the group IDs.

procedure linux.setresgid
(
    rgid:linux.gid_t;
    egid:linux.gid_t;
    sgid:linux.gid_t
);
    @nodisplay;
begin setresgid;

    linux.pushregs;
    mov( linux.sys_setresgid, eax );
    movzx( rgid, ebx );
    movzx( egid, ecx );
    movzx( sgid, edx );
    int( $80 );
    linux.popregs;

end setresgid;

// setresuid - Sets the various user IDs for a process.

procedure linux.setresuid( ruid:linux.uid_t; euid:linux.uid_t; suid:linux.uid_t );
    @nodisplay;
begin setresuid;

    linux.pushregs;
    mov( linux.sys_setresuid, eax );
    movzx( ruid, ebx );
    movzx( euid, ecx );
    movzx( suid, edx );
    int( $80 );
    linux.popregs;

end setresuid;
```

DESCRIPTION

setresuid (introduced in Linux 2.1.44) sets the real user ID, the effective user ID, and the saved set-user-ID of the current process.

Unprivileged user processes (i.e., processes with each of real, effective and saved user ID nonzero) may change the real, effective and saved user ID, each to on of: the current uid, the current effective uid or the current saved uid.

The super-user may set real, effective and saved user ID to arbitrary values.

If one of the parameters equals -1, the corresponding value is not changed.

Completely analogously, setresgid sets the real, effective and saved group ID's of the current process, with the same restrictions for processes with each of real, effective and saved user ID nonzero.

RETURN VALUE

On success, zero is returned. On error, EAX contains an appropriate negative error code.

ERRORS

errno.**eperm**	The current process was not privileged and tried to change the IDs to inappropriate values.

CONFORMING TO

This call is Linux-specific.

SEE ALSO

getuid(2), setuid(2), getreuid(2), setreuid(2), getresuid(2)

3.111 setsid

```
// setsid - creates a new session.

procedure linux.setsid;
   @nodisplay;
begin setsid;

   linux.pushregs;
   mov( linux.sys_setsid, eax );
   int( $80 );
   linux.popregs;

end setsid;
```

DESCRIPTION

linux.setsid() creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session, the process group leader of the new process group, and has no controlling tty. The process group ID and session ID of the calling process are set to the PID of the calling process. The calling process will be the only process in this new process group and in this new session.

RETURN VALUE

The session ID of the calling process.

ERRORS

On error, errno.eperm is returned (the only possible error). It is returned when the process group ID of any process equals the PID of the calling process. Thus, in particular, setsid fails if the calling process is already a process group leader.

NOTES

A process group leader is a process with process group ID equal to its PID. In order to be sure that setsid will succeed, fork and exit, and have the child do setsid().

CONFORMING TO

POSIX, SVr4.

SEE ALSO

setpgid(2), setpgrp(2)

3.112 setuid

```
// setuid - Sets the userID for a given process.

procedure linux.setuid( uid:linux.uid_t );
    @nodisplay;
begin setuid;

    linux.pushregs;
    mov( linux.sys_setuid, eax );
    movzx( uid, ebx );
    int( $80 );
    linux.popregs;

end setuid;
```

DESCRIPTION

setuid sets the effective user ID of the current process. If the effective userid of the caller is root, the real and saved user ID's are also set.

Under Linux, setuid is implemented like the POSIX version with the _POSIX_SAVED_IDS feature. This allows a setuid (other than root) program to drop all of its user privileges, do some un-privileged work, and then re-engage the original effective user ID in a secure manner.

If the user is root or the program is setuid root, special care must be taken. The setuid function checks the effective uid of the caller and if it is the superuser, all process related user ID's are set to uid. After this has occurred, it is impossible for the program to regain root privileges.

Thus, a setuid-root program wishing to temporarily drop root privileges, assume the identity of a non-root user, and then regain root privileges afterwards cannot use setuid. You can accomplish this with the (non-POSIX, BSD) call seteuid.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.eperm	The user is not the super-user, and uid does not match the real or saved user ID of the calling process.

CONFORMING TO

SVr4, SVID, POSIX.1. Not quite compatible with the 4.4BSD call, which sets all of the real, saved, and effective user IDs. SVr4 documents an additional EINVAL error condition.

LINUX-SPECIFIC REMARKS

Linux has the concept of filesystem user ID, normally equal to the effective user ID. The setuid call also sets the filesystem user ID of the current process. See setfsuid(2).

If uid is different from the old effective uid, the process will be forbidden from leaving core dumps.

SEE ALSO

getuid(2), setreuid(2), seteuid(2), setfsuid(2)

3.113 sgetmask

```
// sgetmask - Retrives the signal mask.

procedure linux.sgetmask;
    @nodisplay;
begin sgetmask;

    linux.pushregs;
    mov( linux.sys_sgetmask, eax );
    int( $80 );
    linux.popregs;

end sgetmask;
```

DESCRIPTION

linux.setuid sets the effective user ID of the current process. If the effective userid of the caller is root, the real and saved user ID's are also set.

Under Linux, setuid is implemented like the POSIX version with the _POSIX_SAVED_IDS feature. This allows a setuid (other than root) program to drop all of its user privileges, do some un-privileged work, and then re-engage the original effective user ID in a secure manner.

If the user is root or the program is setuid root, special care must be taken. The setuid function checks the effective uid of the caller and if it is the superuser, all process related user ID's are set to uid. After this has occurred, it is impossible for the program to regain root privileges.

Thus, a setuid-root program wishing to temporarily drop root privileges, assume the identity of a non-root user, and then regain root privileges afterwards cannot use setuid. You can accomplish this with the (non-POSIX, BSD) call seteuid.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.eperm	The user is not the super-user, and uid does not match the real or saved user ID of the calling process.

CONFORMING TO

SVr4, SVID, POSIX.1. Not quite compatible with the 4.4BSD call, which sets all of the real, saved, and effective user IDs. SVr4 documents an additional EINVAL error condition.

LINUX-SPECIFIC REMARKS

Linux has the concept of filesystem user ID, normally equal to the effective user ID. The setuid call also sets the filesystem user ID of the current process. See setfsuid(2).

If uid is different from the old effective uid, the process will be forbidden from leaving core dumps.

SEE ALSO

getuid(2), setreuid(2), seteuid(2), setfsuid(2)

3.114 shmat, shmdt

```
// shmat - SysV share memory attach.

procedure linux.shmat
(
        shmid    :dword;
        shmaddr  :dword;
        shmflg   :dword
);
   @nodisplay;
begin shmat;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_shmat, ebx );
    mov( shmid, ecx );
    mov( shmaddr, edx );
    mov( shmflg, esi );
    int( $80 );
    linux.popregs;

end shmat;

// shmdt - SysV share memory detach.

procedure linux.shmdt
(
        shmaddr:dword
);
   @nodisplay;
begin shmdt;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_shmdt, ebx );
    mov( shmaddr, ecx );
    int( $80 );
    linux.popregs;

end shmdt;
```

DESCRIPTION

linux.shmat attaches the shared memory segment identified by shmid to the data segment of the calling process. The attaching address is specified by shmaddr with one of the following criteria:

If shmaddr is 0, the system tries to find an unmapped region in the range 1 - 1.5G starting from the upper value and coming down from there.

If shmaddr isn't 0 and linux.shm_rnd is asserted in shmflg, the attach occurs at address equal to the rounding down of shmaddr to a multiple of linux.shmlba. Otherwise shmaddr must be a page aligned address at which the attach occurs.

If linux.shm_rdonly is asserted in shmflg, the segment is attached for reading and the process must have read access permissions to the segment. Otherwise the segment is attached for read and write and the process must have read and write access permissions to the segment. There is no notion of write-only shared memory segment.

The brk value of the calling process is not altered by the attach. The segment will automatically detached at process exit. The same segment may be attached as a read and as a read-write one, and more than once, in the process's address space.

On a successful linux.shmat call the system updates the members of the structure shmid_ds associated to the shared memory segment as follows:

- shm_atime is set to the current time.
- shm_lpid is set to the process-ID of the calling process.
- shm_nattch is incremented by one.

Note that the attach succeeds also if the shared memory segment is marked as to be deleted.

The function linux.shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by shmaddr. The detaching shared memory segment must be one among the currently attached ones (to the process's address space) with shmaddr equal to the value returned by the its attaching linux.shmat call.

On a successful shmdt call the system updates the members of the structure shmid_ds associated to the shared memory segment as follows:

- shm_dtime is set to the current time.
- shm_lpid is set to the process-ID of the calling process.
- shm_nattch is decremented by one. If it becomes 0 and the segment is marked for deletion, the segment is deleted.

The occupied region in the user space of the calling process is unmapped.

SYSTEM CALLS

fork() After a fork() the child inherits the attached shared memory segments.

exec() After an exec() all attached shared memory segments are detached (not destroyed).

exit() Upon exit() all attached shared memory segments are detached (not destroyed).

RETURN VALUE

On a failure both functions return -1 with EAX indicating the error, otherwise shmat returns the address of the attached shared memory segment, and shmdt returns 0.

ERRORS

When shmat fails, at return EAX will be set to one among the following negative values:

errno.eacces	The calling process has no access permissions for the requested attach type.
errno.einval	Invalid shmid value, unaligned (i.e., not page-aligned and linux.shm_rnd was not specified) or invalid shmaddr value, or failing attach at brk.
errno.enomem	Could not allocate memory for the descriptor or for the page tables.

The function shmdt can fails only if there is no shared memory segment attached at shmaddr, in such a case at return EAX will be set to errno.einval.

NOTES

On executing a fork(2) system call, the child inherits all the attached shared memory segments.

The shared memory segments attached to a process executing an execve(2) system call will not be attached to the resulting process.

The following is a system parameter affecting a shmat system call:

linux.shmlba Segment low boundary address multiple. Must be page aligned. For the current implementation the linux.shmbla value is linux.page_size.

The implementation has no intrinsic limit to the per process maximum number of shared memory segments (linux.shmseg)

CONFORMING TO

SVr4, SVID. SVr4 documents an additional error condition EMFILE. In SVID-v4 the type of the shmaddr argument was changed from char * into const void *, and the returned type of shmat() from char * into void *. (Linux libc4 and libc5 have the char * prototypes; glibc2 has void *.)

SEE ALSO

ipc(5), shmctl(2), shmget(2)

3.115 shmctl

```
// shmctl - SysV share memory control.

procedure linux.shmctl
(
          shmid :dword;
          cmd       :dword;
    var   buf       :linux.shmid_ds
);
    @nodisplay;
begin shmctl;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_shmget, ebx );
    mov( shmid, ecx );
    mov( cmd, edx );
    mov( buf, esi );
    int( $80 );
    linux.popregs;

end shmctl;
```

DESCRIPTION

shmctl() allows the user to receive information on a shared memory segment, set the owner, group, and permissions of a shared memory segment, or destroy a segment. The information about the segment identified by shmid is returned in a shmid_ds structure:

```
type
    shmid_ds: record
         shm_perm        :ipc_perm;
         shm_segsz       :dword;
         shm_atime       :time_t;
         shm_dtime       :time_t;
         shm_ctime       :time_t;
         shm_cpid        :pid_t;
         shm_lpid        :pid_t;
         shm_nattch      :word;
         __shm_npages    :word;
         __shm_pages     :dword;    // pointer to array of frames.
         __attaches      :dword;    // pointer to descriptors.
    endrecord;
```

The fields in the field shm_perm can be set:

```
type
    ipc_perm: record
         __key    :key_t;
         uid          :uid_t;
         gid          :gid_t;
         cuid     :uid_t;
         cgid     :gid_t;
         mode     :word;
         __pad1   :word;
         __seq    :word;
         __pad2   :word;
         __unused1:dword;
         __unused2:dword;
    endrecord;
```

The following cmds are available:

linux.ipc_stat is used to copy the information about the shared memory segment into the buffer buf. The user must have read access to the shared memory segment.

linux.ipc_set is used to apply the changes the user has made to the uid, gid, or mode members of the shm_perms field. Only the lowest 9 bits of mode are used. The shm_ctime member is also updated. The user must be the owner, creator, or the super-user.

linux.ipc_rmid is used to mark the segment as destroyed. It will actually be destroyed after the last detach. (I.e., when the shm_nattch member of the associated structure shmid_ds is zero.) The user must be the owner, creator, or the super-user.

The user must ensure that a segment is eventually destroyed; otherwise its pages that were faulted in will remain in memory or swap.

In addition, the super-user can prevent or allow swapping of a shared memory segment with the following cmds: (Linux only)

linux.shm_lock prevents swapping of a shared memory segment. The user must fault in any pages that are required to be present after locking is enabled.

linux.shm_unlock allows the shared memory segment to be swapped out.

The linux.ipc_info, linux.shm_stat and linux.shm_info control calls are used by the ipcs(8) program to provide information on allocated resources. In the future, these man be modified as needed or moved to a proc file system interface.

SYSTEM CALLS

fork() After a fork() the child inherits the attached shared memory segments.

exec() After an exec() all attached shared memory segments are detached (not destroyed).

exit() Upon exit() all attached shared memory segments are detached (not destroyed).

RETURN VALUE

0 is returned on success, -1 on error.

ERRORS

On error, EAX will be set to one of the following:

errno.eacces is returned if linux.ipc_stat is requested and shm_perm.modes does not allow read access for msqid.

errno.efault The argument cmd has value linux.ipc_set or linux.ipc_stat but the address pointed to by buf isn't accessible.

errno.einval is returned if shmid is not a valid identifier, or cmd is not a valid command.

errno.eidrm is returned if shmid points to a removed identifier.

errno.eperm is returned if linux.ipc_set or linux.ipc_rmid is attempted, and the user is not the creator, the owner, or the super-user, and the user does not have permission granted to their group or to the world.

NOTE

Various fields in a struct shmid_ds were shorts under Linux 2.2 and have become longs under Linux 2.4. To take advantage of this, a recompilation under glibc-2.1.91 or later should suffice. (The kernel distinguishes old and new calls by a IPC_64 flag in cmd.)

CONFORMING TO

SVr4, SVID. SVr4 documents additional error conditions EINVAL, ENOENT, ENOSPC, ENOMEM, EEXIST. Neither SVr4 nor SVID documents an EIDRM error condition.

SEE ALSO

shmget(2), shmop(2)

3.116 shmget

```
// shmget - SysV share memory get.

procedure linux.shmget
(
        key      :linux.key_t;
        size  :dword;
        shmflg:dword
);
   @nodisplay;
begin shmget;

    linux.pushregs;
    mov( linux.sys_ipc, eax );
    mov( linux.ipcop_shmget, ebx );
    mov( key, ecx );
    mov( size, edx );
    mov( shmflg, esi );
    int( $80 );
    linux.popregs;

end shmget;
```

DESCRIPTION

linux.shmget() returns the identifier of the shared memory segment associated to the value of the argument key. A new shared memory segment, with size equal to the round up of size to a multiple of linux.page_size, is created if key has value linux.ipc_private or key isn't linux.ipc_private, no shared memory segment is associated to key, and linux.ipc_creat is asserted in shmflg (i.e. shmflg & linux.ipc_creat isn't zero). The presence in shmflg is composed of:

linux.ipc_creat to create a new segment. If this flag is not used, then shmget() will find the segment associated with key, check to see if the user has permission to receive the shmid associated with the segment, and ensure the segment is not marked for destruction.

linux.ipc_excl used with linux.ipc_creat to ensure failure if the segment exists.

mode_flags (lowest 9 bits) specifying the permissions granted to the owner, group, and world. Presently, the execute permissions are not used by the system.

If a new segment is created, the access permissions from shmflg are copied into the shm_perm member of the shmid_ds structure that defines the segment. The shmid_ds structure:

```
type
   shmid_ds: record
        shm_perm        :ipc_perm;
        shm_segsz       :dword;
        shm_atime       :time_t;
        shm_dtime       :time_t;
        shm_ctime       :time_t;
        shm_cpid        :pid_t;
        shm_lpid        :pid_t;
        shm_nattch      :word;
        __shm_npages    :word;
        __shm_pages     :dword;    // pointer to array of frames.
        __attaches      :dword;    // pointer to descriptors.
   endrecord;
```

The fields in the field shm_perm can be set:

```
type
   ipc_perm: record
        __key    :key_t;
        uid          :uid_t;
        gid          :gid_t;
        cuid     :uid_t;
        cgid     :gid_t;
        mode     :word;
        __pad1   :word;
        __seq    :word;
        __pad2   :word;
        __unused1:dword;
        __unused2:dword;
   endrecord;
```

Furthermore, while creating, the system call initializes the system shared memory segment data structure shmid_ds as follows:

- shm_perm.cuid and shm_perm.uid are set to the effective user-ID of the calling process.
- shm_perm.cgid and shm_perm.gid are set to the effective group-ID of the calling process.
- The lowest order 9 bits of shm_perm.mode are set to the lowest order 9 bit of shmflg.
- shm_segsz is set to the value of size.
- shm_lpid, shm_nattch, shm_atime and shm_dtime are set to 0.
- shm_ctime is set to the current time.

If the shared memory segment already exists, the access permissions are verified, and a check is made to see if it is marked for destruction.

SYSTEM CALLS

fork() After a fork() the child inherits the attached shared memory segments.

exec() After an exec() all attached shared memory segments are detached (not destroyed).

exit() Upon exit() all attached shared memory segments are detached (not destroyed).

RETURN VALUE

A valid segment identifier, shmid, is returned on success, an appropriate negative error code on error.

ERRORS

On failure, EAX is set to one of the following:

errno.einval is returned if a new segment was to be created and size < SHMMIN or size > SHMMAX, or no new segment was to be created, a segment with given key existed, but size is greater than the size of that segment.

errno.eexist is returned if IPC_CREAT | IPC_EXCL was specified and the segment exists.

errno. eidrm is returned if the segment is marked as destroyed, or was removed.

errno. enospc is returned if all possible shared memory id's have been taken (SHMMNI) or if allocating a segment of the requested size would cause the system to exceed the system-wide limit on shared memory (SHMALL).

errno.enoent is returned if no segment exists for the given key, and IPC_CREAT was not specified.

errno.eacces is returned if the user does not have permission to access the shared memory segment.

errno.enomem is returned if no memory could be allocated for segment overhead.

NOTES

linux.ipc_private isn't a flag field but a key_t type. If this special value is used for key, the system call ignores everything but the lowest order 9 bits of shmflg and creates a new shared memory segment (on success).

The followings are limits on shared memory segment resources affecting a shmget call:

linux.shmall System wide maximum of shared memory pages: policy dependent.

linux.shmmax Maximum size in bytes for a shared memory segment: implementation dependent (currently 4M).

linux.shmmin Minimum size in bytes for a shared memory segment: implementation dependent (currently 1 byte, though PAGE_SIZE is the effective minimum size).

linux.shmmni System wide maximum number of shared memory segments: implementation dependent (currently 4096).

The implementation has no specific limits for the per process maximum number of shared memory segments (linux.shmseg).

BUGS

Use of linux.ipc_private doesn't inhibit to other processes the access to the allocated shared memory segment.

As for the files, there is currently no intrinsic way for a process to ensure exclusive access to a shared memory segment. Asserting both linux.ipc_creat and linux.ipc_excl in shmflg only ensures (on success) that a new shared memory segment will be created, it doesn't imply exclusive access to the segment.

CONFORMING TO

SVr4, SVID. SVr4 documents an additional error condition EEXIST. Neither SVr4 nor SVID documents an EIDRM condition.

SEE ALSO

ftok(3), ipc(5), shmctl(2), shmat(2), shmdt(2)

3.117 sigaction, sigprocmask, sigpending, sigsuspend

```
// sigaction - sets up the action for a given signal.

procedure linux.sigaction
(
        signum   :int32;
    var  act          :linux.sigaction_t;
    var oldaction:linux.sigaction_t
);
    @nodisplay;
begin sigaction;

    linux.pushregs;
    mov( linux.sys_sigaction, eax );
    mov( signum, ebx );
    mov( act, ecx );
    mov( oldaction, edx );
    int( $80 );
    linux.popregs;

end sigaction;


// sigprocmask- Sets up signal masks.

procedure linux.sigprocmask
(
        how      :dword;
    var  set      :linux.sigset_t;
    var  oldset:linux.sigset_t
);
    @nodisplay;
begin sigprocmask;

    linux.pushregs;

    mov( linux.sys_sigprocmask, eax );
    mov( how, ebx );
    mov( set, ecx );
    mov( oldset, edx );
    int( $80 );
    linux.popregs;

end sigprocmask;
```

```
// sigsuspend - Retrieves signals that are pending while blocked.

procedure linux.sigpending( var set:linux.sigset_t );
    @nodisplay;
begin sigpending;

    linux.pushregs;
    mov( linux.sys_sigpending, eax );
    mov( set, ebx );
    int( $80 );
    linux.popregs;

end sigpending;



// sigsuspend - Temporarily sets the signal mask (as specified)
//                 and then suspends the process pending a signal.

procedure linux.sigsuspend( var mask:linux.sigset_t );
    @nodisplay;
begin sigsuspend;

    linux.pushregs;
    mov( linux.sys_sigsuspend, eax );
    mov( mask, ebx );
    int( $80 );
    linux.popregs;

end sigsuspend;
```

DESCRIPTION

The linux.sigaction system call is used to change the action taken by a process on receipt of a specific signal.

signum specifies the signal and can be any valid signal except signals.sigkill and signals.sigstop.

If act is non-null, the new action for signal signum is installed from act. If oldact is non-null, the previous action is saved in oldact.

The signals.sigaction_t structure is defined as something like

```
type
        sigaction_t:record
            sa_sigaction    :procedure
                            (
                                        signum:int32;
                                var     siginfo:siginfo_t;
                                var     buf:var
                            );
            sa_mask         :sigset_t;
            sa_flags        :dword;
            sa_restorer     :procedure;
        endrecord;
```

The sa_restorer element is obsolete and should not be used. POSIX does not specify a sa_restorer element.

sa_handler specifies the action to be associated with signum and may be signals.sig_dfl for the default action, signals.sig_ign to ignore this signal, or a pointer to a signal handling function.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the signals.sa_nodefer or signals.sa_nomask flags are used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

linux.sa_nocldstop If signum is signals.sigchld, do not receive notification when child processes stop (i.e., when child processes receive one of signals.sigstop, signals.sigtstp, signals.sigttin or signals.sigttou).

linux.sa_oneshot or

linux.sa_resethand Restore the signal action to the default state once the signal handler has been called. (This is the default behavior of the signal(2) system call.)

linux. sa_restart Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals.

linux.sa_nomask or

linux.sa_nodefer Do not prevent the signal from being received from within its own signal handler.

linux.sa_siginfo The signal handler takes 3 arguments, not one. In this case, sa_sigaction should be set instead of sa_handler. (The sa_sigaction field was added in Linux 2.1.86.)

The signals.siginfo_t parameter to sa_sigaction is a record with the following elements

```
type
        siginfo_t:record
                si_signo :int32;
                si_errno :int32;
                si_code         :int32;

                _sifields:
                        union

                                _pad    :dword[ 29 ];

                                /* kill() */

                                _kill:
                                        record
                                                _pid    :@global:linux.pid_t;
                                                _uid    :@global:linux.uid_t;
                                        endrecord;

                                /* POSIX.1b timers */

                                _timer:
                                        record
```

```
                    _timer1   :uns32;
                    _timer2   :uns32;
                endrecord;

            /* POSIX.1b signals */

            _rt:
                record
                    _pid      :@global:linux.pid_t;
                    _uid      :@global:linux.uid_t;
                    _sigval   :dword;
                endrecord;

            /* SIGCHLD */

            _sigchld:
                record
                    _pid      :@global:linux.pid_t;
                    _uid      :@global:linux.uid_t;
                    _status   :int32;
                    _utime    :@global:linux.clock_t;
                    _stime    :@global:linux.clock_t;
                endrecord;

            /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */

            _sigfault:
                record
                    _addr     :dword;
                endrecord;

            /* SIGPOLL */

            _sigpoll:
                record
                    _band     :int32;
                    _fd                 :int32;
                endrecord;

        endunion;

endrecord;
```

si_signo, si_errno and si_code are defined for all signals. The rest of the record may be a union, so that one should only read the fields that are meaningful for the given signal. kill(2), POSIX.1b signals and signals.sigchld fill in si_pid and si_uid. signals.sigchld also fills in si_status, si_utime and si_stime. si_int and si_ptr are specified by the sender of the POSIX.1b signal. signals.sigill, signals.sigfpe, signals.sigsegv and signals.sigbus fill in si_addr with the address of the fault. signals.sigpoll fills in si_band and si_fd.

si_code indicates why this signal was sent. It is a value, not a bitmask. The values which are possible for any signal are listed in this table (see linux.hhf for the actual names, generally they are all lower case with a "signals." prefix):

si_code	
Value	Signal origin
SI_USER	kill, sigsend or raise
SI_KERNEL	The kernel
SI_QUEUE	sigqueue
SI_TIMER	timer expired
SI_MESGQ	mesq state changed
SI_ASYNCIO	AIO completed
SI_SIGIO	queued SIGIO

SIGILL	
ILL_ILLOPC	illegal opcode
ILL_ILLOPN	illegal operand
ILL_ILLADR	illegal addressing mode
ILL_ILLTRP	illegal trap
ILL_PRVOPC	privileged opcode
ILL_PRVREG	privileged register
ILL_COPROC	coprocessor error
ILL_BADSTK	internal stack error

SIGFPE	
FPE_INTDIV	integer divide by zero
FPE_INTOVF	integer overflow
FPE_FLTDIV	floating point divide by zero
FPE_FLTOVF	floating point overflow
FPE_FLTUND	floating point underflow
FPE_FLTRES	floating point inexact result
FPE_FLTINV	floating point invalid operation
FPE_FLTSUB	subscript out of range

SIGSEGV	
SEGV_MAPERR	address not mapped to object
SEGV_ACCERR	invalid permissions for mapped object

SIGBUS	
BUS_ADRALN	invalid address alignment
BUS_ADRERR	non-existant physical address
BUS_OBJERR	object specific hardware error

SIGTRAP	
TRAP_BRKPT	process breakpoint
TRAP_TRACE	process trace trap

SIGCHLD	
CLD_EXITED	child has exited
CLD_KILLED	child was killed
CLD_DUMPED	child terminated abnormally
CLD_TRAPPED	traced child has trapped
CLD_STOPPED	child has stopped
CLD_CONTINUED	stopped child has continued

SIGPOLL	
POLL_IN	data input available
POLL_OUT	output buffers available
POLL_MSG	input message available
POLL_ERR	i/o error
POLL_PRI	high priority input available
POLL_HUP	device disconnected

The slinux.igprocmask call is used to change the list of currently blocked signals. The behaviour of the call is dependent on the value of how, as follows.

linux.sig_block The set of blocked signals is the union of the current set and the set argument.

linux.sig_unblock The signals in set are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked.

linux.sig_setmask The set of blocked signals is set to the argument set.

If oldset is non-null, the previous value of the signal mask is stored in oldset.

The linux.sigpending call allows the examination of pending signals (ones which have been raised while blocked). The signal mask of pending signals is stored in set.

The linux.sigsuspend call temporarily replaces the signal mask for the process with that given by mask and then suspends the process until a signal is received.

RETURN VALUE

The functions sigaction, sigprocmask, sigpending and sigsuspend return 0 on success and an appropriate error code in EAX on error. (In the case of sigsuspend there will be no success, and only the error return with errno.eintr is possible.)

ERRORS

errno.einval An invalid signal was specified. This will also be generated if an attempt is made to change the action for signals.sigkill or signals.sigstop, which cannot be caught.

errno.efault act, oldact, set or oldset point to memory which is not a valid part of the process address space.

errno.eintr System call was interrupted.

NOTES

It is not possible to block signals.sigkill or signals.sigstop with the sigprocmask call. Attempts to do so will be silently ignored.

According to POSIX, the behaviour of a process is undefined after it ignores a signals.sigfpe, signals.sigill, or signals.sigsegv signal that was not generated by the kill() or the raise() functions. Integer division by zero has undefined result. On some architectures it will generate a signals.sigfpe signal. (Also dividing the most negative integer by -1 may generate signals.sigfpe.) Ignoring this signal might lead to an endless loop.

POSIX (B.3.3.1.3) disallows setting the action for signals.sigchld to signals.sig_ign. The BSD and SYSV behaviours differ, causing BSD software that sets the action for signals.sigchld to signals.sig_ign to fail on Linux.

The POSIX spec only defines signals.sa_nocldstop. Use of other sa_flags is non-portable.

The signals.sa_resethand flag is compatible with the SVr4 flag of the same name.

The signals.sa_nodefer flag is compatible with the SVr4 flag of the same name under kernels 1.3.9 and newer. On older kernels the Linux implementation allowed the receipt of any signal, not just the one we are installing (effectively overriding any sa_mask settings).

The signals.sa_resethand and signals.sa_nodefer names for SVr4 compatibility are present only in library versions 3.0.9 and greater.

The signals.sa_siginfo flag is specified by POSIX.1b. Support for it was added in Linux 2.2.

sigaction can be called with a null second argument to query the current signal handler. It can also be used to check whether a given signal is valid for the current machine by calling it with null second and third arguments.

See sigsetops(3) for details on manipulating signal sets.

CONFORMING TO

POSIX, SVr4. SVr4 does not document the EINTR condition.

UNDOCUMENTED

Before the introduction of signals.sa_siginfo it was also possible to get some additional information namely by using a sa_handler with second argument of type struct sigcontext. See the relevant kernel sources for details. This use is obsolete now.

SEE ALSO

kill(1), kill(2), killpg(2), pause(2), raise(3), siginterrupt(3), signal(2), signal(7), sigsetops(3), sigvec(2)

3.118 sigaltstack

```
// sigaltstack - Specifies an alternate stack for processing signals.

procedure linux.sigaltstack( var sss:linux.stack_t; var oss:linux.stack_t );
    @nodisplay;
begin sigaltstack;

    linux.pushregs;
    mov( linux.sys_sigaltstack, eax );
    mov( sss, ebx );
    mov( oss, ecx );
    int( $80 );
    linux.popregs;

end sigaltstack;
```

DESCRIPTION

sigaction(2) may indicate that a signal should execute on an alternate stack. Where this is the case, sigaltstack(2) stores the signal in an alternate stack structure ss where its execution status may be examined prior to processing.

The signals.stack_t struct is defined as follows:

```
type
    stack_t:record
         ss_sp    :dword;
         ss_flags :dword;
         ss_size  :@global:linux.size_t;
    endrecord;
```

ss_sp — points to the stack structure.

ss_flags — specifies the stack state to signals.ss_disable or signals.ss_onstack as follows:

If ss is not NULL,the new state may be set to signals.ss_disable, which specifies that the stack is to be disabled and ss_sp and ss_size are ignored. If signals.ss_disable is not set, the stack will be enabled.

If oss is not NULL, the stack state may be either signals.ss_onstack or signals.ss_disable. The value signals.ss_onstack indicates that the process is currently executing on the alternate stack and that any attempt to modify it during execution will fail. The value signals.ss_disable indicates that the current signal stack is disabled.

ss_size — specifies the size of the stack.

The value signals.sigstksz defines the average number of bytes used when allocating an alternate stack area. The value signals.minsigstksz defines the minimum stack size for a signal handler. When processing an alternate stack size, your program should include these values in the stack requirement to plan for the overhead of the operating system.

RETURN VALUES

sigaltstack(2) returns 0 on success and an appropriate negative error code in EAX on error.

ERRORS

sigaltstack(2) sets EAX for the following conditions:

errno.einval ss is not a null pointer the ss_flags member pointed to by ss contains flags other than signals.ss_disable.

errno.enomem The size of the alternate stack area is less than signals.minsigstksz.

errno.eperm An attempt was made to modify an active stack.

STANDARDS

This function comforms to: XPG4-UNIX.

SEE ALSO

getcontext(2), sigaction(2), sigsetjmp(3).

3.119 signal

```
// signal - installs a new signal handler.

procedure linux.signal( signum:int32; sighandler:procedure( signum:int32) );
    @nodisplay;
begin signal;

    linux.pushregs;
    mov( linux.sys_signal, eax );
    mov( signum, ebx );
    mov( sighandler, ecx );
    int( $80 );
    linux.popregs;

end signal;
```

DESCRIPTION

The linux.signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either signals.sig_ign or signals.sig_dfl.

Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to signals.sig_ign, then the signal is ignored. If the handler is set to signals.sig_dfl, then the default action associated to the signal (see signal(7)) occurs. Finally, if the handler is set to a function sighandler then first either the handler is reset to signals.sig_dfl or an implementation-dependent blocking of the signal is performed and next sighandler is called with argument signum.

Using a signal handler function for a signal is called "catching the signal". The signals signals.sigkill and signals.sigstop cannot be caught or ignored.

RETURN VALUE

The function signal() returns the previous value of the signal handler, or signals.sig_err on error.

PORTABILITY

The original Unix signal() would reset the handler to signals.sig_dfl, and System V (and the Linux kernel and libc4,5) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler.

Trying to change the semantics of this call using defines and includes is not a good idea. It is better to avoid signal altogether, and use sigaction(2) instead.

NOTES

According to POSIX, the behaviour of a process is undefined after it ignores a signals.sigfpe, signals.sigill, or signals.sigsegv signal that was not generated by the kill() or the raise() functions. Integer division by zero has undefined result. On some architectures it will generate a signals.sigfpe signal. (Also dividing the most negative integer by -1 may generate signals.sigfpe.) Ignoring this signal might lead to an endless loop.

According to POSIX (3.3.1.3) it is unspecified what happens when signals.sigchld is set to signals.sig_ign. Here the BSD and SYSV behaviours differ, causing BSD software that sets the action for signals.sigchld to signals.sig_ign to fail on Linux.

CONFORMING TO

ANSI C

SEE ALSO

kill(1), kill(2), killpg(2), pause(2),raise(3), sigaction(2), signal(7), sigsetops(3), sigvec(2), alarm(2)

3.120 sigpending

See “sigaction, sigprocmask, sigpending, sigsuspend” on page 167.

3.121 sigprocmask

See “sigaction, sigprocmask, sigpending, sigsuspend” on page 167.

3.122 sigreturn

```
// sigreturn: return to statement that threw the signal.

procedure linux.sigreturn( unused:dword );
    @nodisplay;
begin sigreturn;

    linux.pushregs;

    mov( linux.sys_sigreturn, eax );
    mov( unused, ebx );
    int( $80 );
    linux.popregs;

end sigreturn;
```

DESCRIPTION

When the Linux kernel creates the stack frame for a signal handler, a call to sigreturn is inserted into the stack frame so that the the signal handler will call sigreturn upon return. This inserted call to sigreturn cleans up the stack so that the process can restart from where it was interrupted by the signal.

RETURN VALUE

sigreturn never returns.

WARNING

The sigreturn call is used by the kernel to implement signal handlers. It should never be called directly. Better yet, the specific use of the __unused argument varies depending on the architecture.

CONFORMING TO

sigreturn is specific to Linux and should not be used in programs intended to be portable.

SEE ALSO

kill(2), signal(2), signal(7)

3.123 sigsuspend

See “sigaction, sigprocmask, sigpending, sigsuspend” on page 167.

3.124 socketcall

```
// socketcall: TCP/IP--socket invocation.

procedure linux.socketcall( callop:dword; var args:var );
    @nodisplay;
begin socketcall;

    linux.pushregs;
    mov( linux.sys_socketcall, eax );
    mov( callop, ebx );
    mov( args, ecx );
    int( $80 );
    linux.popregs;

end socketcall;
```

DESCRIPTION

socketcall is a common kernel entry point for the socket system calls. call determines which socket function to invoke. args points to a block containing the actual arguments, which are passed through to the appropriate call.

User programs should call the appropriate functions by their usual names. Only standard library implementors and kernel hackers need to know about socketcall.

CONFORMING TO

This call is specific to Linux, and should not be used in programs intended to be portable.

SEE ALSO

accept(2), bind(2), connect(2), getpeername(2), getsockname(2), getsockopt(2), listen(2), recv(2), recvfrom(2), send(2), sendto(2), setsockopt(2), shutdown(2), socket(2), socketpair(2)

3.125 ssetmask

```
// ssetmask - Sets the signal mask.

procedure linux.ssetmask( mask:dword );
    @nodisplay;
begin ssetmask;

    linux.pushregs;
    mov( linux.sys_ssetmask, eax );
    mov( mask, ebx );
    int( $80 );
    linux.popregs;

end ssetmask;
```

(no man page???)

3.126 stat

See “fstat, lstat, stat” on page 46.

3.127 statfs

See “fstatfs, statfs” on page 49.

3.128 stime

```
// stime - Retrives time in seconds.

procedure linux.stime( var tptr:int32 );
    @nodisplay;
begin stime;

    linux.pushregs;
    mov( linux.sys_stime, eax );
    mov( tptr, ebx );
    int( $80 );
    linux.popregs;

end stime;
```

DESCRIPTION

stime sets the system's idea of the time and date. Time, pointed to by t, is measured in seconds from 00:00:00 GMT January 1, 1970. stime() may only be executed by the super user.

RETURN VALUE

On success, zero is returned. On error, EAX returns an appropriate negative error code.

ERRORS

errno.eperm	The caller is not the super-user.

CONFORMING TO

SVr4, SVID, X/OPEN

SEE ALSO

date(1), settimeofday(2)

3.129 swapoff, swapon

```
// swapoff: Disables swapping to file.

procedure linux.swapoff( path:string );
    @nodisplay;
begin swapoff;

    linux.pushregs;
    mov( linux.sys_swapoff, eax );
    mov( path, ebx );
    int( $80 );
    linux.popregs;

end swapoff;

// swapon: Sets the swap area to the specified file.

procedure linux.swapon( path:string; swapflags:dword );
    @nodisplay;
begin swapon;

    linux.pushregs;
    mov( linux.sys_swapon, eax );
    mov( path, ebx );
    mov( swapflags, ecx );
    int( $80 );
    linux.popregs;

end swapon;
```

DESCRIPTION

swapon sets the swap area to the file or block device specified by path. swapoff stops swapping to the file or block device specified by path.

swapon takes a swapflags argument. If swapflags has the SWAP_FLAG_PREFER bit turned on, the new swap area will have a higher priority than default. The priority is encoded as:

(prio << SWAP_FLAG_PRIO_SHIFT) & SWAP_FLAG_PRIO_MASK

These functions may only be used by the super-user.

PRIORITY

Each swap area has a priority, either high or low. The default priority is low. Within the low-priority areas, newer areas are even lower priority than older areas.

All priorities set with swapflags are high-priority, higher than default. They may have any non-negative value chosen by the caller. Higher numbers mean higher priority.

Swap pages are allocated from areas in priority order, highest priority first. For areas with different priorities, a higher-priority area is exhausted before using a lower-priority area. If two or more areas have the same priority, and it is the highest priority available, pages are allocated on a round-robin basis between them.

As of Linux 1.3.6, the kernel usually follows these rules, but there are exceptions.

RETURN VALUE

On success, zero is returned. On error, EAX will contain an appropriate negative error code.

ERRORS

Many other errors can occur if path is not valid.

errno.eperm The user is not the super-user, or more than linux.max_swapfiles (defined to be 8 in Linux 1.3.6) are in use.

errno.einval is returned if path exists, but is neither a regular path nor a block device.

errno.enoent is returned if path does not exist.

errno.enomem is returned if there is insufficient memory to start swapping.

CONFORMING TO

These functions are Linux specific and should not be used in programs intended to be portable. The second `swapflags' argument was introduced in Linux 1.3.2.

NOTES

The partition or path must be prepared with mkswap(8).

SEE ALSO

mkswap(8), swapon(8), swapoff(8)

3.130 symlink

```
// symlink: Create a symbolic link.

procedure linux.symlink( oldpath:string; newpath:string );
   @nodisplay;
begin symlink;

   linux.pushregs;
   mov( linux.sys_symlink, eax );
   mov( oldpath, ebx );
   mov( newpath, ecx );
   int( $80 );
   linux.popregs;

end symlink;
```

DESCRIPTION

linux.symlink creates a symbolic link named newpath which contains the string oldpath.

Symbolic links are interpreted at run-time as if the contents of the link had been substituted into the path being followed to find a file or directory.

Symbolic links may contain .. path components, which (if used at the start of the link) refer to the parent directories of that in which the link resides.

A symbolic link (also known as a soft link) may point to an existing file or to a nonexistent one; the latter case is known as a dangling link.

The permissions of a symbolic link are irrelevant; the ownership is ignored when following the link, but is checked when removal or renaming of the link is requested and the link is in a directory with the sticky bit set.

If newpath exists it will not be overwritten.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.eperm The filesystem containing newpath does not support the creation of symbolic links.

errno.efault oldpath or newpath points outside your accessible address space.

errno.eacces Write access to the directory containing newpath is not allowed for the process's effective uid, or one of the directories in newpath did not allow search (execute) permission.

errno.enametoolong oldpath or newpath was too long.

errno.enoent A directory component in newpath does not exist or is a dangling symbolic link, or oldpath is the empty string.

errno.enotdir A component used as a directory in newpath is not, in fact, a directory.

errno.enomem Insufficient kernel memory was available.

errno.erofs newpath is on a read-only filesystem.

errno.eexist newpath already exists.

errno.eloop Too many symbolic links were encountered in resolving newpath.

errno.enospc The device containing the file has no room for the new directory entry.

errno.eio An I/O error occurred.

NOTES

No checking of oldpath is done.

Deleting the name referred to by a symlink will actually delete the file (unless it also has other hard links). If this behaviour is not desired, use link.

CONFORMING TO

SVr4, SVID, POSIX, BSD 4.3. SVr4 documents additional error codes SVr4, SVID, BSD 4.3, X/OPEN. SVr4 documents additional error codes EDQUOT and ENOSYS. See open(2) re multiple files with the same name, and NFS.

SEE ALSO

readlink(2), link(2), unlink(2), rename(2), open(2), lstat(2), ln(1)

3.131 sync

```
// sync - flushes file data to disk.

procedure linux.sync;
    @nodisplay;
begin sync;

    linux.pushregs;
    mov( linux.sys_sync, eax );
    int( $80 );
    linux.popregs;

end sync;
```

DESCRIPTION

linux.sync first commits inodes to buffers, and then buffers to disk.

RETURN VALUE

sync always returns 0.

CONFORMING TO

SVr4, SVID, X/OPEN, BSD 4.3

BUGS

According to the standard specification (e.g., SVID), sync() schedules the writes, but may return before the actual writing is done. However, since version 1.3.20 Linux does actually wait. (This still does not guarantee data integrity: modern disks have large caches.)

SEE ALSO

bdflush(2), fsync(2), fdatasync(2), update(8), sync(8)

3.132 sysctl

```
// sysctl - Reads and writes kernel parameters.

procedure linux.sysctl( var args:linux.__sysctl_args );
    @nodisplay;
begin sysctl;

    linux.pushregs;
    mov( linux.sys_sysctl, eax );
    mov( args, ebx );
    int( $80 );
    linux.popregs;

end sysctl;
```

DESCRIPTION

The linux.sysctl call reads and/or writes kernel parameters. For example, the hostname, or the maximum number of open files. The argument has the form

```
type
        __sysctl_args:
              record
                  theName    :pointer to char;
                  nlen       :int32;
                  oldval     :dword;
                  oldlenp    :pointer to size_t;
                  newval     :dword;
                  newlen     :size_t;
                  __unused   :dword[4];
              endrecord;
```

This call does a search in a tree structure, possibly resembling a directory tree under /proc/sys, and if the requested item is found calls some appropriate routine to read or modify the value.

RETURN VALUE

Upon successful completion, linux.sysctl returns 0. Otherwise,it returns a negative error code in EAX.

ERRORS

errno.enotdir name was not found.

errno.eperm No search permission for one of the encountered `directories', or no read permission where oldval was nonzero, or no write permission where newval was nonzero.

errno.efault The invocation asked for the previous value by setting oldval non-NULL, but allowed zero room in oldlenp.

CONFORMING TO

This call is Linux-specific, and should not be used in programs intended to be portable. A sysctl call has been present in Linux since version 1.3.57. It originated in 4.4BSD. Only Linux has the /proc/sys mirror, and the object naming schemes differ between Linux and BSD 4.4, but the declaration of the sysctl(2) function is the same in both.

BUGS

The object names vary between kernel versions. THIS MAKES THIS SYSTEM CALL WORTHLESS FOR APPLICATIONS. Use the /proc/sys interface instead. Not all available objects are properly documented. It is not yet possible to change operating system by writing to /proc/sys/kernel/ostype.

SEE ALSO

proc(5)

3.133 sysfs

```
// bdflush - Tunes the buffer dirty flush daemon.

#macro sysfs( option, args[] );

    #if( @elements( args ) = 0 )

         sysfs1( option )

    #elseif( @elements( args ) = 1 )

         sysfs2( option, @text( args[0] ) )

    #else

         sysfs3( option, @text( args[0] ), @text( args[1] ))

    #endif

#endmacro;


procedure linux.sysfs1( option:dword );
    @nodisplay;
begin sysfs1;

    linux.pushregs;
    mov( linux.sys_sysfs, eax );
    mov( option, ebx );
    int( $80 );
    linux.popregs;

end sysfs1;

// sysfs2 - Two parameter version of sysfs.

procedure linux.sysfs2( option:dword; fsname:string );
    @nodisplay;
begin sysfs2;

    linux.pushregs;
    mov( linux.sys_sysfs, eax );
    mov( option, ebx );
    mov( fsname, ecx );
    int( $80 );
    linux.popregs;

end sysfs2;


// sysfs3 - Three parameter version of sysfs.

procedure linux.sysfs3( option:dword; fs_index:dword; var buf:var );
    @nodisplay;
begin sysfs3;

    linux.pushregs;
    mov( linux.sys_sysfs, eax );
    mov( option, ebx );
    mov( fs_index, ecx );
    mov( buf, edx );
    int( $80 );
    linux.popregs;

end sysfs3;
```

DESCRIPTION

linux.sysfs returns information about the file system types currently present in the kernel. The specific form of the sysfs call and the information returned depends on the option in effect:

1. Translate the file-system identifier string fsname into a file-system type index.

2. Translate the file-system type index fs_index into a null-terminated file-system identifier string. This string will be written to the buffer pointed to by buf. Make sure that buf has enough space to accept the string.

3. Return the total number of file system types currently present in the kernel.

The numbering of the file-system type indexes begins with zero.

RETURN VALUE

On success, linux.sysfs returns the file-system index for option 1, zero for option 2, and the number of currently configured file systems for option 3. On error, EAX contains a negative error code.

ERRORS

errno.einval fsname is not a valid file-system type identifier; fs_index is out-of-bounds; option is invalid.

errno.efault Either fsname or buf is outside your accessible address space.

CONFORMING TO

SVr4.

NOTE

On Linux with the proc filesystem mounted on /proc, the same information can be derived from /proc/filesystems.

BUGS

There is no way to guess how large buf should be.

3.134 sysinfo

```
// sysinfo: Returns system information.

procedure linux.sysinfo( var info:linux.sysinfo_t );
    @nodisplay;
begin sysinfo;

    linux.pushregs;
    mov( linux.sys_sysinfo, eax );
    mov( info, ebx );
    int( $80 );
    linux.popregs;

end sysinfo;
```

DESCRIPTION

linux.sysinfo used to return information in the following structure:

```
sysinfo_t:
      record
          uptime :int32;
          loads  :uns32[3];
          totalram:uns32;
          freeram:uns32;
          shardram:uns32;
          bufferram:uns32;
          totalswap:uns32;
          freeswap:uns32;
          procs  :uns16;
          align(64);
      endrecord;
```

and the sizes are given as multiples of mem_unit bytes.

sysinfo provides a simple way of getting overall system statistics. This is more portable than reading /dev/kmem. For an example of its use, see intro(2).

RETURN VALUE

On success, zero is returned. On error, EAX contains the negative error code.

ERRORS

errno.efault pointer to struct sysinfo is invalid.

CONFORMING TO

This function is Linux-specific, and should not be used in programs intended to be portable.

The Linux kernel has a sysinfo system call since 0.98.pl6. Linux libc contains a sysinfo() routine since 5.3.5, and glibc has one since 1.90.

SEE ALSO

proc(5)

3.135 syslog

```
// syslog: TCP/IP--socket invocation.

procedure linux.syslog( theType:dword; var bufp:var; len:dword );
    @nodisplay;
begin syslog;

    linux.pushregs;
    mov( linux.sys_syslog, eax );
    mov( theType, ebx );
    mov( bufp, ecx );
    mov( len, edx );
    int( $80 );
    linux.popregs;

end syslog;
```

DESCRIPTION

The theType argument determines the action taken by syslog.

Quoting from kernel/printk.c:

```
* Commands to sys_syslog:
*
*      0 -- Close the log.  Currently a NOP.
*      1 -- Open the log. Currently a NOP.
*      2 -- Read from the log.
*      3 -- Read up to the last 4k of messages in the ring buffer.
*      4 -- Read and clear last 4k of messages in the ring buffer
*      5 -- Clear ring buffer.
*      6 -- Disable printk's to console
*      7 -- Enable printk's to console
*      8 -- Set level of messages printed to console
```

Only function 3 is allowed to non-root processes.

The kernel log buffer The kernel has a cyclic buffer of length LOG_BUF_LEN (4096, since 1.3.54: 8192, since 2.1.113: 16384) in which messages given as argument to the kernel function printk() are stored (regardless of their loglevel).

The call syslog (2,buf,len) waits until this kernel log buffer is nonempty, and then reads at most len bytes into the buffer buf. It returns the number of bytes read. Bytes read from the log disappear from the log buffer: the information can only be read once. This is the function executed by the kernel when a user program reads /proc/kmsg.

The call syslog (3,buf,len) will read the last len bytes from the log buffer (nondestructively), but will not read more than was written into the buffer since the last `clear ring buffer' command (which does not clear the buffer at all). It returns the number of bytes read.

The call syslog (4,buf,len) does precisely the same, but also executes the `clear ring buffer' command.

The call syslog (5,dummy,idummy) only executes the `clear ring buffer' command.

The loglevel:

The kernel routine printk() will only print a message on the console, if it has a loglevel less than the value of the variable console_loglevel (initially DEFAULT_CONSOLE_LOGLEVEL (7), but set to 10 if the kernel commandline contains the word `debug', and to 15 in case of a kernel fault - the 10 and 15 are just silly, and equivalent to 8). This variable is set (to a value in the range 1-8) by the call syslog

(8,dummy,value). The calls syslog (type,dummy,idummy) with type equal to 6 or 7, set it to 1 (kernel panics only) or 7 (all except debugging messages), respectively.

Every text line in a message has its own loglevel. This level is DEFAULT_MESSAGE_LOGLEVEL - 1 (6) unless the line starts with <d> where d is a digit in the range 1-7, in which case the level is d. The conventional meaning of the loglevel is defined in <linux/kernel.h> as follows:

```
#define KERN_EMERG    "<0>"  /* system is unusable*/
#define KERN_ALERT    "<1>"  /* action must be taken immediately */
#define KERN_CRIT     "<2>"  /* critical conditions*/
#define KERN_ERR      "<3>"  /* error conditions*/
#define KERN_WARNING  "<4>"  /* warning conditions*/
#define KERN_NOTICE   "<5>"  /* normal but significant condition */
#define KERN_INFO     "<6>"  /* informational*/
#define KERN_DEBUG    "<7>"  /* debug-level messages*/
```

RETURN VALUE

In case of error, EAX contains a negative error code. Otherwise, for type equal to 2, 3 or 4, syslog() returns the number of bytes read, and otherwise 0.

ERRORS

errno.eperm An attempt was made to change console_loglevel or clear the kernel message ring buffer by a process without root permissions.

errno.einval Bad parameters.

errno.erestartsys System call was interrupted by a signal - nothing was read.

CONFORMING TO

This system call is Linux specific and should not be used in programs intended to be portable.

SEE ALSO

syslog(3)

3.136 time

```
// time - Return the system time.

procedure linux.time( var tloc:dword );
    @nodisplay;
begin time;

    linux.pushregs;
    mov( linux.sys_time, eax );
    mov( tloc, ebx );
    int( $80 );
    linux.popregs;

end time;
```

DESCRIPTION

linux.time returns the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

If t is non-NULL, the return value is also stored in the memory pointed to by t.

RETURN VALUE

On success, the value of time in seconds since the Epoch is returned. On error, ((time_t)-1) is returned, and EAX contains a negative error code.

ERRORS

errno.efault	t points outside your accessible address space.

NOTES

POSIX.1 defines seconds since the Epoch as a value to be interpreted as the number of seconds between a specified time and the Epoch, according to a formula for conversion from UTC equivalent to conversion on the naive basis that leap seconds are ignored and all years divisible by 4 are leap years. This value is not the same as the actual number of seconds between the time and the Epoch, because of leap seconds and because clocks are not required to be synchronised to a standard reference. The intention is that the interpretation of seconds since the Epoch values be consistent; see POSIX.1 Annex B 2.2.2 for further rationale.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3 Under BSD 4.3, this call is obsoleted by gettimeofday(2). POSIX does not specify any error conditions.

SEE ALSO

ctime(3), date(1), ftime(3), gettimeofday(2)

3.137 times

```
// times - retrieves execution times for the current process.

procedure linux.times( var buf:linux.tms );
    @nodisplay;
begin times;

    linux.pushregs;
    mov( linux.sys_times, eax );
    mov( buf, ebx );
    int( $80 );
    linux.popregs;

end times;
```

DESCRIPTION

The linux.times() function stores the current process times in the record linux.tms that buf points to. linux.tms is as defined in linux.hhf:

```
    tms:record
         tms_utime    :clock_t;
         tms_stime    :clock_t;
         tms_cutime   :clock_t;
         tms_cstime   :clock_t;
    endrecord;
```

The tms_utime field contains the CPU time spent executing instructions of the calling process. The tms_stime field contains the CPU time spent in the system while executing tasks on behalf of the calling process. The tms_cutime field contains the sum of the tms_utime and tms_cutime values for all waited-for terminated children. The tms_cstime field contains the sum of the tms_stime and tms_cstime values for all waited-for terminated children.

Times for terminated children (and their descendants) is added in at the moment wait(2) or waitpid(2) returns their process ID. In particular, times of grandchildren that the children did not wait for are never seen.

All times reported are in clock ticks.

RETURN VALUE

The function times returns the number of clock ticks that have elapsed since an arbitrary point in the past. For Linux this point is the moment the system was booted. This return value may overflow the possible range of type linux.clock_t. On error, EAX returns with a negative error code.

NOTE

The number of clock ticks per second can be obtained using sysconf(_SC_CLK_TCK); In POSIX-1996 the symbol CLK_TCK (defined in <time.h>) is mentioned as obsolescent. It is obsolete now.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

HISTORICAL NOTES

SVr1-3 returns long and the struct members are of type time_t although they store clock ticks, not seconds since the epoch. V7 used long for the struct members, because it had no type time_t yet.

On older systems the number of clock ticks per second is given by the variable HZ.

SEE ALSO

time(1), getrusage(2), wait(2), clock(3), sysconf(3)

3.138 truncate

See “ftruncate, truncate” on page 53.

3.139 umask

```
// umask - sets the default permissions mask.

procedure linux.umask( mask:linux.mode_t );
   @nodisplay;
begin umask;

   linux.pushregs;
   mov( linux.sys_umask, eax );
   mov( mask, ebx );
   int( $80 );
   linux.popregs;

end umask;
```

DESCRIPTION

linux.umask sets the umask to mask & $1FF.

The umask is used by open(2) to set initial file permissions on a newly-created file. Specifically, permissions in the umask are turned off from the mode argument to open(2) (so, for example, the common umask default value of 022 results in new files being created with permissions 0666 & ~022 = 0644 = rw-r--r-- in the usual case where the mode is specified as 0666).

RETURN VALUE

This system call always succeeds and the previous value of the mask is returned.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

SEE ALSO

creat(2), open(2)

3.140 umount

See “mount, umount” on page 93.

3.141 uname

```
// uname: Retrieves system info.

procedure linux.uname( var buf:linux.utsname );
    @nodisplay;
begin uname;

    linux.pushregs;
    mov( linux.sys_uname, eax );
    mov( buf, ebx );
    int( $80 );
    linux.popregs;

end uname;
```

DESCRIPTION

linux.uname returns system information in the structure pointed to by buf. The utsname record is as defined in linux.hhf:

```
type
        utsname:
              record
                  sysname    :char[65];
                  nodename   :char[65];
                  release    :char[65];
                  version    :char[65];
                  machine    :char[65];
                  domainname :char[65];
              endrecord;
```

RETURN VALUE

On success, zero is returned. On error, EAX will contain a negative error code.

ERRORS

errno.efault buf is not valid.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN

The domainname member is a GNU extension.

SEE ALSO

uname(1), getdomainname(2), gethostname(2)

3.142 unlink

```
// unlink - Delete a file/remove a link.

procedure linux.unlink( pathname:string );
    @nodisplay;
begin unlink;

    linux.pushregs;
    mov( linux.sys_unlink, eax );
    mov( pathname, ebx );
    int( $80 );
    linux.popregs;

end unlink;
```

DESCRIPTION

unlink deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open the file will remain in existence until the last file descriptor referring to it is closed.

If the name referred to a symbolic link the link is removed.

If the name referred to a socket, fifo or device the name for it is removed but processes which have the object open may continue to use it.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

errno.eacces Write access to the directory containing pathname is not allowed for the process's effective uid, or one of the directories in pathname did not allow search (execute) permission.

errno.eperm or

errno.eacces The directory containing pathname has the sticky-bit (S_ISVTX) set and the process's effective uid is neither the uid of the file to be deleted nor that of the directory containing it.

errno.eperm (Linux only) The filesystem does not allow unlinking of files.

errno.eperm The system does not allow unlinking of directories, or unlinking of directories requires privileges that the current process doesn't have. (This is the POSIX prescribed error return.)

errno.eisdir pathname refers to a directory. (This is the non-POSIX value returned by Linux since 2.1.132.)

errno.ebusy (not on Linux) The file pathname cannot be unlinked because it is being used by the system or another process and the implementation considers this an error.

errno.efault pathname points outside your accessible address space.

errno.enametoolong pathname was too long.

errno.enoent A directory component in pathname does not exist or is a dangling symbolic link.

errno.enotdir A component used as a directory in pathname is not, in fact, a directory.

errno.enomem Insufficient kernel memory was available.

errno.erofs pathname refers to a file on a read-only filesystem.

errno.eloop Too many symbolic links were encountered in translating pathname.

errno.eio An I/O error occurred.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, 4.3BSD. SVr4 documents additional error conditions EINTR, EMULTIHOP, ETXTBSY, ENOLINK.

BUGS

Infelicities in the protocol underlying NFS can cause the unexpected disappearance of files which are still being used.

SEE ALSO

link(2), rename(2), open(2), rmdir(2), mknod(2), mkfifo(3), remove(3), rm(1)

3.143 uselib

```
// uselib: Specifies a dynamic linking library module.

procedure linux.uselib( library:string );
    @nodisplay;
begin uselib;

    linux.pushregs;
    mov( linux.sys_uselib, eax );
    mov( library, ebx );
    int( $80 );
    linux.popregs;

end uselib;
```

DESCRIPTION

linux.uselib selects the shared library binary that will be used by the calling process.

RETURN VALUE

On success, zero is returned. On error, EAX contains an appropriate error code.

ERRORS

In addition to all of the error codes returned by open(2) and mmap(2), the following may also be returned:

errno.enoexec The file specified by library is not executable, or does not have the correct magic numbers.

errno.eacces The library specified by library is not readable.

CONFORMING TO

uselib() is Linux specific, and should not be used in programs intended to be portable.

SEE ALSO

ar(1), gcc(1), ld(1), ldd(1), mmap(2), open(2), ld.so(8)

3.144 ustat

```
// ustat - returns information about a mounted file system.

procedure linux.ustat( dev:linux.dev_t; var ubuf:linux.ustat_t );
    @nodisplay;
begin ustat;

    linux.pushregs;
    mov( linux.sys_ustat, eax );
    movzx( dev, ebx );
    mov( ubuf, ecx );
    int( $80 );
    linux.popregs;

end ustat;
```

DESCRIPTION

linux.ustat returns information about a mounted file system. dev is a device number identifying a device containing a mounted file system. ubuf is a pointer to a ustat_t structure that contains the following members:

```
type
    ustat_t: record
         f_tfree   :@global:kernel.__kernel_daddr_t;
         f_tinode:@global:kernel.__kernel_ino_t;
         f_fname   :char[6];
         f_fpack   :char[6];
    endrecord;
```

The last two fields, f_fname and f_fpack, are not implemented and will always be filled with null characters.

RETURN VALUE

On success, zero is returned and the ustat_t structure pointed to by ubuf will be filled in. On error, EAX will contain an appropriate error code.

ERRORS

errno.einval dev does not refer to a device containing a mounted file system.

errno.efault ubuf points outside of your accessible address space.

errno.enosys The mounted file system referenced by dev does not support this operation, or any version of Linux before 1.3.16.

NOTES

ustat has only been provided for compatibility. All new programs should use statfs(2) instead.

CONFORMING TO

SVr4. SVr4 documents additional error conditions ENOLINK, ECOMM, and EINTR but has no ENOSYS condition.

SEE ALSO

statfs(2), stat(2)

3.145 utime

```
// utime - Change the access and modification times of a file.

procedure linux.utime( filename:string; var times: linux.utimbuf );
    @nodisplay;
begin utime;

    linux.pushregs;
    mov( linux.sys_utime, eax );
    mov( filename, ebx );
    mov( times, ecx );
    int( $80 );
    linux.popregs;

end utime;
```

DESCRIPTION

linux.utime changes the access and modification times of the inode specified by filename to the actime and modtime fields of buf respectively. If buf is NULL, then the access and modification times of the file are set to the current time. The utimbuf structure is:

```
type
         utimbuf:
               record
                   actime     :time_t;
                   modtime    :time_t;
               endrecord;
```

In the Linux DLL 4.4.1 libraries, utimes is just a wrapper for linux.utime: tvp[0].tv_sec is actime, and tvp[1].tv_sec is modtime. The timeval structure is:

```
type
    timeval: record
         tv_sec    :time_t;
         tv_usec   :suseconds_t;
    endrecord;
```

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

Other errors may occur.

errno.eacces	Permission to write the file is denied.
errno. enoent	filename does not exist.

CONFORMING TO

utime: SVr4, SVID, POSIX. SVr4 documents additional error conditions EFAULT, EINTR, ELOOP, EMULTIHOP, ENAMETOOLONG, ENOLINK, ENOTDIR, ENOLINK, ENOTDIR, EPERM, EROFS. utimes: BSD 4.3

SEE ALSO

stat(2)

3.146 vfork

```
// vfork - Special version of fork (no data copy).

procedure linux.vfork;
    @nodisplay;
begin vfork;

    linux.pushregs;
    mov( linux.sys_vfork, eax );
    int( $80 );
    linux.popregs;

end vfork;
```

STANDARD DESCRIPTION

(From XPG4 / SUSv2 / POSIX draft.) The vfork() function has the same effect as fork(), except that the behaviour is undefined if the process created by vfork() either modifies any data other than a variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling _exit() or one of the exec family of functions.

ERRORS

errno.EAGAIN	Too many processes - try again.
errno.ENOMEM	There is insufficient swap space for the new process.

LINUX DESCRIPTION

linux.vfork, just like linux.fork(2), creates a child process of the calling process. For details and return value and errors, see fork(2).

linux.vfork() is a special case of linux.clone(2). It is used to create new processes without copying the page tables of the parent process. It may be useful in performance sensitive applications where a child will be created which then immediately issues a linux.execve().

linux.vfork() differs from fork in that the parent is suspended until the child makes a call to execve(2) or _exit(2). The child shares all memory with its parent, including the stack, until execve() is issued by the child. The child must not return from the current function or call exit(), but may call _exit().

Signal handlers are inherited, but not shared. Signals to the parent arrive after the child releases the parent.

HISTORIC DESCRIPTION

Under Linux, fork() is implemented using copy-on-write pages, so the only penalty incurred by fork() is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. However, in the bad old days a fork() would require making a complete copy of the caller's data space, often needlessly, since usually immediately afterwards an exec() is done. Thus, for greater efficiency, BSD introduced the vfork system call, that did not fully copy the address space of the parent process, but borrowed the parent's memory and thread of control until a call to execve() or an exit occurred. The parent process was suspended while the child was using its resources. The use of vfork was tricky - for example, not modifying data in the parent process depended on knowing which variables are held in a register.

BUGS

It is rather unfortunate that Linux revived this spectre from the past. The BSD manpage states: "This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork as it will, in that case, be made synonymous to fork."

Formally speaking, the standard description given above does not allow one to use vfork() since a following exec might fail, and then what happens is undefined.

Details of the signal handling are obscure and differ between systems. The BSD manpage states: "To avoid a possible deadlock situation, processes that are children in the middle of a vfork are never sent SIGTTOU or SIGTTIN signals; rather, output or ioctls are allowed and input attempts result in an end-of-file indication."

Currently (Linux 2.3.25), strace(1) cannot follow vfork() and requires a kernel patch.

HISTORY

The vfork() system call appeared in 3.0BSD. In BSD 4.4 it was made synonymous to fork(), but NetBSD introduced it again, cf. http://www.netbsd.org/Documentation/kernel/vfork.html . In Linux, it has been equivalent to fork() until 2.2.0-pre6 or so. Since 2.2.0-pre9 (on i386, somewhat later on other architectures) it is an independent system call. Support was added in glibc 2.0.112.

CONFORMING TO

The vfork call may be a bit similar to calls with the same name in other operating systems. The requirements put on vfork by the standards are weaker than those put on fork, so an implementation where the two are synonymous is compliant. In particular, the programmer cannot rely on the parent remaining blocked until a call of execve() or _exit() and cannot rely on any specific behaviour w.r.t. shared memory.

SEE ALSO

clone(2), execve(2), fork(2), wait(2)

3.147 vhangup

```
// vhangup: Virtual hang-up of a console.

procedure linux.vhangup;
    @nodisplay;
begin vhangup;

    linux.pushregs;
    mov( linux.sys_vhangup, eax );
    int( $80 );
    linux.popregs;

end vhangup;
```

DESCRIPTION

vhangup simulates a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.

RETURN VALUE

On success, zero is returned. On error, EAX returns with a negative error code.

ERRORS

errno.eperm The user is not the super-user.

CONFORMING TO

This call is Linux-specific, and should not be used in programs intended to be portable.

SEE ALSO

init(8)

3.148 vm86

```
// vm86 - vm86 call for DOS emu.

procedure linux.vm86( fn:dword; var vm86pss:linux.vm86plus_struct );
    @nodisplay;
begin vm86;

    linux.pushregs;
    mov( linux.sys_vm86, eax );
    mov( fn, ebx );
    mov( vm86pss, ecx );
    int( $80 );
    linux.popregs;

end vm86;
```

DESCRIPTION

The system call vm86 was introduced in Linux 0.97p2. In Linux 2.1.15 and 2.0.28 it was renamed to vm86old, and a new vm86 was introduced. The definition of `struct vm86_struct' was changed in 1.1.8 and 1.1.9.

These calls cause the process to enter VM86 mode, and are used by dosemu.

RETURN VALUE

On success, zero is returned. On error, EAX contains a negative error code.

ERRORS

(for vm86old)

errno.eperm Saved kernel stack exists. (This is a kernel sanity check; the saved stack should only exist within vm86 mode itself.)

CONFORMING TO

This call is specific to Linux on Intel processors, and should not be used in programs intended to be portable.

3.149 wait4

```
// idle: Sets the current process as the idle process.

procedure linux.wait4
(
        pid      :linux.pid_t;
        status:dword;
        options:dword;
    var rusage:linux.rusage_t
);
    @nodisplay;
begin wait4;

    linux.pushregs;
    mov( linux.sys_wait4, eax );
    mov( pid, ebx );
    mov( status, ecx );
    mov( options, edx );
    mov( rusage, esi );
    int( $80 );
    linux.popregs;

end wait4;
```

DESCRIPTION

The wait4 function suspends execution of the current process until a child as specified by the pid argument has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child as requested by pid has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed.

The value of pid can be one of:

< -1 which means to wait for any child process whose process group ID is equal to the absolute value of pid.

-1 which means to wait for any child process; this is equivalent to calling wait3.

0 which means to wait for any child process whose process group ID is equal to that of the calling process.

> 0 which means to wait for the child whose process ID is equal to the value of pid.

The value of options is a bitwise OR of zero or more of the following constants:

linux.wnohang which means to return immediately if no child is there to be waited for.

linux.wuntraced which means to also return for children which are stopped, and whose status has not been reported.

If status is not NULL, wait3 or wait4 store status information in the location pointed to by status.

This status can be evaluated with the following macros (these macros take the stat buffer (an int) as an argument -- not a pointer to the buffer!):

linux.wifexited(status) is non-zero if the child exited normally.

linux.wexitstatus(status) evaluates to the least significant eight bits of the return code of the child which terminated, which may have been set as the argument to a call to exit() or as the argument for a return statement in the main program. This macro can only be evaluated if linux.wifexited returned non-zero.

linux.wifsignaled(status) returns true if the child process exited because of a signal which was not caught.

linux.wtermsig(status) returns the number of the signal that caused the child process to terminate. This macro can only be evaluated if linux.wifsignaled returned non-zero.

linux.wifstopped(status) returns true if the child process which caused the return is currently stopped; this is only possible if the call was done using linux.wuntraced.

linux.wstopsig(status) returns the number of the signal which caused the child to stop. This macro can only be evaluated if linux.wifstopped returned non-zero.

If rusage is not NULL, the struct rusage as defined in linux.hhf it points to will be filled with accounting information. See getrusage(2) for details.

RETURN VALUE

The process ID of the child which exited, a negative error code in EAX on error, when no unwaited-for child processes of the specified kind exist) or zero if linux.wnohang was used and no child was available yet. In the latter two cases EAX will be set appropriately.

ERRORS

errno.echild No unwaited-for child process as specified does exist.

errno.erestartsys if linux.wnohang was not set and an unblocked signal or a signals.sigchld was caught. This error is returned by the system call. The library interface is not allowed to return errno.erestartsys, but will return errno.eintr.

CONFORMING TO

SVr4, POSIX.1

SEE ALSO

signal(2), getrusage(2), wait(2), signal(7)

3.150 waitpid

```
// waitpid - Linux process wait call.

procedure linux.waitpid
(
        pid          :linux.pid_t;
    var stat_addr:dword;
        options   :dword
);
    @nodisplay;
begin waitpid;

    linux.pushregs;
    mov( linux.sys_waitpid, eax );
    mov( pid, ebx );
    mov( stat_addr, ecx );
    mov( options, edx );
    int( $80 );
    linux.popregs;

end waitpid;
```

DESCRIPTION

The waitpid function suspends execution of the current process until a child as specified by the pid argument has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child as requested by pid has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed.

The value of pid can be one of:

< -1 which means to wait for any child process whose process group ID is equal to the absolute value of pid.

-1 which means to wait for any child process; this is the same behaviour which wait exhibits.

0 which means to wait for any child process whose process group ID is equal to that of the calling process.

> 0 which means to wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following constants:

linux.wnohang which means to return immediately if no child has exited.

linux.wuntraced which means to also return for children which are stopped, and whose status has not been reported.

If status is not NULL, wait or waitpid store status information in the location pointed to by status.

This status can be evaluated with the following macros (these macros take the stat buffer (an int) as an argument -- not a pointer to the buffer!):

linux.**wifexited**(status) is non-zero if the child exited normally.

linux.**wexitstatus**(status) evaluates to the least significant eight bits of the return code of the child which terminated, which may have been set as the argument to a call to exit() or as the argument for a return statement in the main program. This macro can only be evaluated if linux.wifexited returned non-zero.

linux.**wifsignaled**(status) returns true if the child process exited because of a signal which was not caught.

linux.**wtermsig**(status) returns the number of the signal that caused the child process to terminate. This macro can only be evaluated if linux.wifsignaled returned non-zero.

linux.**wifstopped**(status) returns true if the child process which caused the return is currently stopped; this is only possible if the call was done using linux.wuntraced.

linux.**wstopsig**(status) returns the number of the signal which caused the child to stop. This macro can only be evaluated if linux.wifstopped returned non-zero.

RETURN VALUE

The process ID of the child which exited, or zero if linux.wnohang was used and no child was available, or a negative error code.

ERRORS

errno.ECHILD if the process specified in pid does not exist or is not a child of the calling process. (This can happen for one's own child if the action for SIGCHLD is set to SIG_IGN. See also the NOTES section about threads.)

errno.EINVAL if the options argument was invalid.

errno.ERESTARTSYS if linux.wnohang was not set and an unblocked signal or a signals.sigchld was caught. This error is returned by the system call. The library interface is not allowed to return errno.erestartsys, but will return errno.eintr.

In the Linux kernel, a kernel-scheduled thread is not a distinct construct from a process. Instead, a thread is simply a process that is created using the Linux-unique clone(2) system call; other routines such as the portable pthread_create(3) call are implemented using clone(2). Thus, if two threads A and B are siblings, then thread A cannot wait on any processes forked by thread B or its descendents, because an uncle cannot wait on his nephews. In some other Unix-like systems where multiple threads are implemented as belonging to a single process, thread A can wait on any processes forked by sibling thread B; you will have to rewrite any code that makes this assumption for it to work on Linux.

CONFORMING TO

SVr4, POSIX.1

SEE ALSO

clone(2), signal(2), wait4(2), pthread_create(3), signal(7)

3.151 write

```
// write - writes data via a file handle.

procedure linux.write( fd:dword; var buf:var; count:linux.size_t );
    @nodisplay;
begin write;

    linux.pushregs;
    mov( linux.sys_write, eax );
    mov( fd, ebx );
    mov( buf, ecx );
    mov( count, edx );
    int( $80 );
    linux.popregs;

end write;
```

DESCRIPTION

write writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data. Note that not all file systems are POSIX conforming.

RETURN VALUE

On success, the number of bytes written are returned (zero indicates nothing was written). On error, EAX will contain a negative error code. If count is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect. For a special file, the results are not portable.

ERRORS

errno.EBADF fd is not a valid file descriptor or is not open for writing.

errno. EINVAL fd is attached to an object which is unsuitable for writing.

errno. EFAULT buf is outside your accessible address space.

errno.EFBIG An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process' file size limit, or to write at a position past than the maximum allowed offset.

errno.EPIPE fd is connected to a pipe or socket whose reading end is closed. When this happens the writing process will receive a SIGPIPE signal; if it catches, blocks or ignores this the error EPIPE is returned.

errno.EAGAIN Non-blocking I/O has been selected using O_NONBLOCK and the write would block.

errno.EINTR The call was interrupted by a signal before any data was written.

errno.ENOSPC The device containing the file referred to by fd has no room for the data.

errno.EIO A low-level I/O error occurred while modifying the inode.

Other errors may occur, depending on the object connected to fd.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, 4.3BSD. SVr4 documents additional error conditions EDEADLK, ENOLCK, ENOLNK, ENOSR, ENXIO, EPIPE, or ERANGE. Under SVr4 a write may be interrupted and return EINTR at any point, not just before any data is written.

SEE ALSO

open(2), read(2), fcntl(2), close(2), lseek(2), select(2), ioctl(2), fsync(2), fwrite(3)

3.152 writev

See “readv, writev” on page 126.

4 Appendix A: Linux System Call Opcodes, Sorted Alphabetically

This appendix lists the Linux system calls alphabetically. This table also provides a prototype for the HLA wrapper function and a short description of the function.

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

access	033	`procedure access( pathname:string; mode:int32 );`
		`returns( "eax" );`

acct	051	`procedure acct( filename: string );`
		`returns( "eax" );`

adjtimex	124	`procedure adjtimex( var buf:timex );`
		`returns( "eax" );`

alarm	027	`procedure alarm( seconds:uns32 );`
		`returns( "eax" );`

bdflush	134	`procedure bdflush( func:dword; address:dword );`
		`returns( "eax" );`

brk	045	`procedure brk( end_data_segment:dword );`
		`returns( "eax" );`

capget	184	(no current HLA prototype)
capset	185	(no current HLA prototype)

chdir	012	`procedure chdir( filename:string );`
		`returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

chmod	015	`procedure chmod( filename:string; mode:mode_t );`
		`returns( "eax" );`

chown	182	`procedure chown`
		`( path:string; owner:uid_t; group:gid_t);`
		`returns( "eax" );`

chroot	061	`procedure chroot( path:string );`
		`returns( "eax" );`

clone	120	`type process:procedure(var p:var );`
		`procedure clone`
		`(`
		`fn       :process;`
		`varchild_stack:var;`
		`flags :dword;`
		`var arg  :var`
		`);`
		`returns( "eax" );`

close	006	`procedure close( fd:dword );`
		`returns( "eax" );`

creat	008	`procedure creat( pathname:string; mode:mode_t );`
		`returns( "eax" );`

create_module	127	`procedure create_module`
		`( theName:string; size:size_t );`
		`returns( "eax" );`

delete_module	129	`procedure delete_module( theName:string );`
		`returns( "eax" );`

dup	041	`procedure dupfd( oldfd:dword );`
		`returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

dup2	063	`procedure dup2( oldfd:dword; newfd:dword );`
		`returns( "eax" );`

execve	011	`procedure execve`
		`( filename:string; var argv:var; var envp:var );`
		`returns( "eax" );`

exit	001	`procedure _exit( status:int32 );`

fchdir	133	`procedure fchdir( fd:dword );`
		`returns( "eax" );`

fchmod	094	`procedure fchmod( fd:dword; mode:mode_t );`
		`returns( "eax" );`

fchown	095	`procedure fchown( fd:dword; owner:uid_t; group:gid_t );`
		`returns( "eax" );`

fcntl	055	`procedure fcntl2( fd:dword; cmd:dword );`
		`returns( "eax" );`
		
		`procedure fcntl3( fd:dword; cmd:dword; arg:dword );`
		`returns( "eax" );`
		
		`macro fcntl( fd, cmd, arg[] );`
		
		`#if( @elements( arg ) = 0 )`
		
		`fcntl2( fd, cmd )`
		
		`#else`
		
		`fcntl3( fd, cmd, @text( arg[0] ))`
		
		`#endif`
		
		`endmacro;`

fdatasync	148	`procedure fdatasync( fd:dword );`
		`returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

flock	143	`procedure flock( fd:dword; operation:int32 );`
		`  returns( "eax" );`

fork	002	`procedure fork;`
		`  returns( "eax" );`

fstat	028	(obsolete version)

fstat	108	`procedure fstat( fd:dword; var buf:stat_t );`
		`  returns( "eax" );`

fstatfs	100	`procedure fstatfs( fd:dword; var buf:statfs_t );`
		`  returns( "eax" );`

fsync	118	`procedure fsync( fd:dword );`
		`  returns( "eax" );`

ftruncate	093	`procedure ftruncate( fd:dword; length:off_t );`
		`  returns( "eax" );`

getcwd	183	`procedure getcwd( var buf:var; maxlen:size_t );`
		`  returns( "eax" );`

getdents	141	`procedure getdents`
		`( fd:dword; var dirp:dirent; count:int32 );`
		`  returns( "eax" );`

getegid	050	`procedure getegid;`
		`  returns( "eax" );`

geteuid	049	`procedure geteuid;`
		`  returns( "eax" );`

getgid	047	`procedure getgid;`
		`  returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

getgroups	080	procedure getgroups(size:dword; var list:var);
		returns("eax");

getitimer	105	procedure getitimer
		(which:dword; var theValue:itimerval);
		returns("eax");

get_kernel_syms	130	procedure get_kernel_syms(var table:kernel_sym);
		returns("eax");

getpgid	132	procedure getpgid(pid:pid_t);
		returns("eax");

getpgrp	065	procedure getpgrp;
		returns("eax");

getpid	020	procedure getpid;
		returns("eax");

getppid	064	procedure getppid;
		returns("eax");

getpriority	096	procedure getpriority(which:dword; who:dword);
		returns("eax");

getresgid	171	procedure getresgid
		(
		var rgid:gid_t;
		var egid:gid_t;
		var sgid:gid_t
		);
		returns("eax");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

getresuid	165	`procedure getresuid`
		`(`
		`  var ruid:uid_t;`
		`  var euid:uid_t;`
		`  var suid:uid_t`
		`);`
		`  returns( "eax" );`

getrlimit	076	`procedure getrlimit( resource:dword; var rlim:rlimit );`
		`  returns( "eax" );`

getrusage	077	`procedure getrusage( who:dword; var usage:rusage_t );`
		`  returns( "eax" );`

getsid	147	`procedure getsid( pid:pid_t );`
		`  returns( "eax" );`

gettimeofday	078	`procedure gettimeofday`
		`( var tv:timeval; var tz:timezone );`
		`  returns( "eax" );`

getuid	024	`procedure getuid;`
		`  returns( "eax" );`

idle	112	`procedure idle;`
		`  returns( "eax" );`

init_module	128	`procedure init_module`
		`( theName:string; var image:module_t );`
		`  returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

ioctl	054	procedure ioctl2(d:int32; request:int32);
		  returns("eax");
		
		procedure ioctl3(d:int32; request:int32; argp:string
		);
		  returns("eax");
		
		macro ioctl(d, request, argp[]);
		
		  #if(@elements(argp) = 0)
		
		    ioctl2(d, request)
		
		  #else
		
		    ioctl3(d, request, @text(argp[0]))
		
		  #endif
		
		endmacro;

ioperm	101	procedure ioperm
		(from:dword; num:dword; turn_on:int32);
		  returns("eax");

iopl	110	procedure iopl(level:dword);
		  returns("eax");

ipc	117	procedure ipc
		(
		    theCall:dword;
		    first:dword;
		    second:dword;
		    third:dword;
		  var ptr :var;
		    fifth:dword
		);
		  returns("eax");

kill	037	procedure kill(pid:pid_t; sig:int32);
		  returns("eax");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

lchown	016	procedure lchown
		(filename:string; user:uid_t; group:gid_t);
		returns("eax");

link	009	procedure link(oldname:string; newname:string);
		returns("eax");

llseek	140	procedure llseek
		(
		fd:dword;
		offset_high:dword;
		offset_low:dword;
		var theResult:loff_t;
		whence:dword
		);
		returns("eax");

lseek	019	procedure lseek
		(fd:dword; offset:off_t; origin:dword);
		returns("eax");

lstat	084	(obsolete call)

lstat	107	procedure lstat(filename:string; var buf:stat_t);
		returns("eax");

mkdir	039	procedure mkdir(pathname:string; mode:int32);
		returns("eax");

mknod	014	procedure mknod
		(filename:string; mode:dword; dev:dev_t);
		returns("eax");

mlock	150	procedure mlock(addr:dword; len:size_t);
		returns("eax");

mlockall	152	procedure mlockall(flags:dword);
		returns("eax");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

mmap	090	`procedure mmap`
		`(`
		`  start:dword;`
		`  length:size_t;`
		`  prot:int32;`
		`  flags:dword;`
		`  fd:dword;`
		`  offset:off_t`
		`);`
		`  returns( "eax" );`

modify_ldt	123	`procedure modify_ldt`
		`( func:dword; var ptr:var; bytecount:dword );`
		`  returns( "eax" );`

mount	021	`procedure mount`
		`(`
		`    dev_name:string;`
		`    dir_name:string;`
		`    theType:string;`
		`    new_flags:dword;`
		`  var data:var`
		`);`
		`  returns( "eax" );`

mprotect	125	`procedure mprotect`
		`( var addr:var; len:size_t; prot:dword );`
		`  returns( "eax" );`

mremap	163	`procedure mremap`
		`(`
		`  old_address:dword;`
		`  old_size:size_t;`
		`  new_size:size_t;`
		`  flags :dword`
		`);`
		`  returns( "eax" );`

msync	144	`procedure msync`
		`( start:dword; length:size_t; flags:dword );`
		`  returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

munlock	151	`procedure munlock( addr:dword; len:size_t );`
		`returns( "eax" );`

munlockall	153	`procedure munlockall;`
		`returns( "eax" );`

munmap	091	`procedure munmap( start:dword; length:size_t );`
		`returns( "eax" );`

nanosleep	162	`procedure nanosleep`
		`( var req:timespec; var rem:timespec );`
		`returns( "eax" );`

nfsservctl	169	(no HLA prototype exists)

nice	034	`procedure nice( increment: int32 );`
		`returns( "eax" );`

oldmount	022	(obsolete version of mount)
old_select	082	(obsolete version of select)
olduname	059	(obsolete version of uname)

open	005	`procedure open`
		`( filename:string; flags:dword; mode:mode_t );`
		`returns( "eax" );`

pause	029	`procedure pause;`
		`returns( "eax" );`

personality	136	`procedure personality( persona:dword );`
		`returns( "eax" );`

pipe	042	`procedure pipe( fd:dword );`
		`returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

poll	168	`procedure poll`
		`( var ufds:pollfd; nfds:dword; timeout:dword );`
		`  returns( "eax" );`

prctl	172	`procedure prctl`
		`(`
		`  option:dword;`
		`  arg2:dword;`
		`  arg3:dword;`
		`  arg4:dword;`
		`  arg5:dword`
		`);`
		`  returns( "eax" );`

pread	180	`procedure pread`
		`(`
		`     fd    :dword;`
		`  var buf  :var;`
		`     count:size_t;`
		`     offset:off_t`
		`);`
		`  returns( "eax" );`

ptrace	026	`procedure ptrace`
		`( request:dword; pid:dword; addr:dword; data:dword );`
		`  returns( "eax" );`

pwrite	181	`procedure pwrite`
		`(`
		`     fd    :dword;`
		`  var buf  :var;`
		`     count:size_t;`
		`     offset:off_t`
		`);`
		`  returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

query_module	167	procedure query_module
		(
		theName:string;
		which:dword;
		var buf :var;
		bufsize:size_t;
		var retval:size_t
		);
		returns("eax");

quotactl	131	procedure quotactl
		(cmd:dword; special:string; id:dword; addr:caddr_t);
		returns("eax");

read	003	procedure read(fd:dword; var buf:var; count:size_t);
		returns("eax");

readdir	089	(No HLA prototype available)

readlink	085	procedure readlink
		(path:string; var buf:var; bufsize:size_t);
		returns("eax");

readv	145	procedure readv
		(fd:dword; var vector:var; count:int32);
		returns("eax");

reboot	088	procedure reboot
		(
		magic:dword;
		magic2:dword;
		flag:dword;
		var arg:var
		);
		returns("eax");

rename	038	procedure rename(oldpath:string; newpath:string);
		returns("eax");

rmdir	040	procedure rmdir(pathname:string);
		returns("eax");
		external("linux_rmdir");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype
rt_sigaction	174	(No HLA prototype yet)
rt_sigpending	176	(No HLA prototype yet)
rt_sigprocmask	175	(No HLA prototype yet)
rt_sigqueueinfo	178	(No HLA prototype yet)
rt_sigreturn	173	(No HLA prototype yet)
rt_sigsuspend	179	(No HLA prototype yet)
rt_sigtimedwait	177	(No HLA prototype yet)

sched_getparam	155	`procedure sched_getparam`
		`( pid:pid_t; var p:sched_param_t );`
		`returns( "eax" );`

sched_get_priority_max	159	`procedure sched_get_priority_max( policy:dword );`
		`returns( "eax" );`

sched_get_priority_min	160	`procedure sched_get_priority_min( policy:dword );`
		`returns( "eax" );`

sched_getscheduler	157	`procedure sched_getscheduler( pid:pid_t );`
		`returns( "eax" );`

sched_rr_get_interval	161	`procedure sched_rr_get_interval`
		`( pid:pid_t; var tp:timespec );`
		`returns( "eax" );`

sched_setparam	154	`procedure sched_setparam`
		`( pid:pid_t; var p:sched_param_t );`
		`returns( "eax" );`

sched_setscheduler	156	`procedure sched_setscheduler`
		`( pid:pid_t; policy:dword; var p:sched_param_t );`
		`returns( "eax" );`

sched_yield	158	`procedure sched_yield;`
		`returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

select	142	procedure select
		(
		    n :int32;
		  var readfds:fd_set;
		  var writefds:fd_set;
		  var exceptfds:fd_set;
		  var timeout:timespec;
		  var sigmask:sigset_t
		);
		  returns("eax");

sendfile	187	procedure sendfile
		(
		    out_fd:dword;
		    in_fd:dword;
		  var offset:off_t;
		    count:size_t
		);
		  returns("eax");

setdomainname	121	procedure setdomainname
		(domainName:string; len:size_t);
		  returns("eax");

setfsgid	139	procedure setfsgid(fsgid:gid_t);
		  returns("eax");

setfsuid	138	procedure setfsuid(fsuid:uid_t);
		  returns("eax");

setgid	046	procedure setgid(gid:gid_t);
		  returns("eax");

setgroups	081	procedure setgroups(size:size_t; var list:var);
		  returns("eax");

sethostname	074	procedure sethostname(theName:string; len:size_t);
		  returns("eax");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

setitimer	104	procedure setitimer
		(
		which:dword;
		var ivalue:itimerval;
		var ovalue:itimerval
		);
		returns("eax");

setpgid	057	procedure setpgid(pid:pid_t; pgid:pid_t);
		returns("eax");

setpriority	097	procedure setpriority(which:dword; who:dword);
		returns("eax");

setregid	071	procedure setregid(rgid:gid_t; egid:gid_t);
		returns("eax");

setresgid	170	procedure setresgid
		(rgid:gid_t; egid:gid_t; sgid:gid_t);
		returns("eax");

setresuid	164	procedure setresuid
		(ruid:uid_t; euid:uid_t; suid:uid_t);
		returns("eax");

setreuid	070	procedure setreuid(rgid:gid_t; egid:gid_t);
		returns("eax");

setrlimit	075	procedure setrlimit
		(resource:dword; var rlim:rlimit);
		returns("eax");

setsid	066	procedure setsid;
		returns("eax");
		external("linux_setsid");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

settimeofday	079	procedure settimeofday
		(var tv:timeval; var tz:timezone);
		  returns("eax");

setuid	023	procedure setuid(uid:uid_t);
		  returns("eax");

sgetmask	068	procedure sgetmask;
		  returns("eax");

sigaction	067	procedure sigaction
		(
		    signum:int32;
		  var act :sigaction_t;
		  var oldaction:sigaction_t
		);
		  returns("eax");

sigaltstack	186	procedure sigaltstack
		(var sss:stack_t; var oss:stack_t);
		  returns("eax");

signal	048	procedure signal
		(signum:int32; sighandler:procedure(signum:int32));
		  returns("eax");

sigpending	073	procedure sigpending(var set:sigset_t);
		  returns("eax");

sigprocmask	126	procedure sigprocmask
		(
		    how :dword;
		  var set :sigset_t;
		  var oldset:sigset_t
		);
		  returns("eax");

sigreturn	119	procedure sigreturn(unused:dword);
		    returns("eax");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

sigsuspend	072	`procedure sigsuspend( var mask:sigset_t );`
		`  returns( "eax" );`

socketcall	102	`procedure socketcall( callop:dword; var args:var );`
		`  returns( "eax" );`

ssetmask	069	`procedure ssetmask( mask:dword );`
		`  returns( "eax" );`

stat	018	(Obsolete call, don’t use)

stat	106	`procedure stat( filename:string; var buf:stat_t );`
		`  returns( "eax" );`

statfs	098	`procedure statfs( path:string; var buf:statfs_t );`
		`  returns( "eax" );`

stime	025	`procedure stime( var tptr:int32 );`
		`  returns( "eax" );`

swapoff	115	`procedure swapoff( path:string );`
		`  returns( "eax" );`

swapon	087	`procedure swapon( path:string; swapflags:dword );`
		`  returns( "eax" );`

symlink	083	`procedure symlink`
		`( oldpath:string; newpath:string );`
		`  returns( "eax" );`

sync	036	`procedure sync;`
		`  returns( "eax" );`

sysctl	149	`procedure sysctl( var args:__sysctl_args );`
		`  returns( "eax" );`

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

sysfs	135	procedure sysfs1(option:dword);
		  returns("eax");
		
		procedure sysfs2(option:dword; fsname:string);
		  returns("eax");
		
		procedure sysfs3
		(option:dword; fs_index:dword; var buf:var);
		  returns("eax");
		
		macro sysfs(option, args[]);
		
		  #if(@elements(args) = 0)
		
		    sysfs1(option)
		
		  #elseif(@elements(args) = 1)
		
		    sysfs2(option, @text(args[0]))
		
		  #else
		
		    sysfs3(option, @text(args[0]), @text(args[1]))
		
		  #endif
		
		endmacro;

sysinfo	116	procedure sysinfo(var info:sysinfo_t);
		  returns("eax");

syslog	103	procedure syslog
		(theType:dword; var bufp:var; len:dword);
		  returns("eax");

time	013	procedure time(var tloc:dword);
		  returns("eax");

times	043	procedure times(var buf:tms);
		  returns("eax");

truncate	092	procedure truncate(path:string; length:off_t);
		  returns("eax");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype
umask	060	

umount	052	procedure umount
		(
		specialfile:string;
		dir :string;
		filesystemtype:string;
		mountflags:dword;
		var data :var
		);
		returns("eax");
		external("linux_umount");

uname	109	procedure umask(mask:mode_t);
		returns("eax");

uname	122	procedure uname(var buf:utsname);
		returns("eax");

unlink	010	procedure unlink(pathname:string);
		returns("eax");

uselib	086	procedure uselib(library:string);
		returns("eax");

ustat	062	procedure ustat(dev:dev_t; var ubuf:ustat_t);
		returns("eax");

utime	030	procedure utime
		(filename:string; var times: utimbuf);
		returns("eax");

vfork	190	procedure vfork;
		returns("eax");

vhangup	111	procedure vhangup;
		returns("eax");

Table 1: Linux System Calls Sorted by Name

Name	System Opcode	HLA Prototype

vm86	166	procedure vm86
		(fn:dword; var vm86pss:vm86plus_struct);
		  returns("eax");

vm86old	113	(obsolete, don't use)

wait4	114	procedure wait4
		  (
		    pid :pid_t;
		    status:dword;
		    options:dword;
		  var rusage:rusage_t
		);
		  returns("eax");

waitpid	007	procedure waitpid
		(pid:pid_t; var stat_addr:dword; options:dword);
		  returns("eax");

write	004	procedure write
		(fd:dword; var buf:var; count:size_t);
		  returns("eax");

writev	146	procedure writev
		(fd:dword; var vector:var; count:int32);
		  returns("eax");

5 Appendix B: Linux System Call Opcodes, Sorted Numerically

Table 2 Linux System Calls by Number

Linux System Opcode	Function
001	exit
002	fork
003	read
004	write

Table 2 Linux System Calls by Number

Linux System Opcode	Function
005	open
006	close
007	waitpid
008	creat
009	link
010	unlink
011	execve
012	chdir
013	time
014	mknod
015	chmod
016	lchown
018	stat
019	lseek
020	getpid
021	mount
022	oldmount
023	setuid
024	getuid
025	stime
026	ptrace
027	alarm
028	fstat
029	pause
030	utime
033	access
034	nice
036	sync
037	kill
038	rename
039	mkdir

Table 2 Linux System Calls by Number

Linux System Opcode	Function
040	rmdir
041	dup
042	pipe
043	times
045	brk
046	setgid
047	getgid
048	signal
049	geteuid
050	getegid
051	acct
052	umount
054	ioctl
055	fcntl
057	setpgid
059	olduname
060	umask
061	chroot
062	ustat
063	dup2
064	getppid
065	getpgrp
066	setsid
067	sigaction
068	sgetmask
069	ssetmask
070	setreuid
071	setregid
072	sigsuspend
073	sigpending
074	sethostname

Table 2 Linux System Calls by Number

Linux System Opcode	Function
075	setrlimit
076	getrlimit
077	getrusage
078	gettimeofday
079	settimeofday
080	getgroups
081	setgroups
082	old_select
083	symlink
084	lstat
085	readlink
086	uselib
087	swapon
088	reboot
089	readdir
090	mmap
091	munmap
092	truncate
093	ftruncate
094	fchmod
095	fchown
096	getpriority
097	setpriority
098	statfs
100	fstatfs
101	ioperm
102	socketcall
103	syslog
104	setitimer
105	getitimer
106	stat

Table 2 Linux System Calls by Number

Linux System Opcode	Function
107	lstat
108	fstat
109	uname
110	iopl
111	vhangup
112	idle
113	vm86old
114	wait4
115	swapoff
116	sysinfo
117	ipc
118	fsync
119	sigreturn
120	clone
121	setdomainname
122	uname
123	modify_ldt
124	adjtimex
125	mprotect
126	sigprocmask
127	create_module
128	init_module
129	delete_module
130	get_kernel_syms
131	quotactl
132	getpgid
133	fchdir
134	bdflush
135	sysfs
136	personality
138	setfsuid

Table 2 Linux System Calls by Number

Linux System Opcode	Function
139	setfsgid
140	llseek
141	getdents
142	select
143	flock
144	msync
145	readv
146	writev
147	getsid
148	fdatasync
149	sysctl
150	mlock
151	munlock
152	mlockall
153	munlockall
154	sched_setparam
155	sched_getparam
156	sched_setscheduler
157	sched_getscheduler
158	sched_yield
159	sched_get_priority_max
160	sched_get_priority_min
161	sched_rr_get_interval
162	nanosleep
163	mremap
164	setresuid
165	getresuid
166	vm86
167	query_module
168	poll
169	nfsservctl

Table 2 Linux System Calls by Number

Linux System Opcode	Function
170	setresgid
171	getresgid
172	prctl
173	rt_sigreturn
174	rt_sigaction
175	rt_sigprocmask
176	rt_sigpending
177	rt_sigtimedwait
178	rt_sigqueueinfo
179	rt_sigsuspend
180	pread
181	pwrite
182	chown
183	getcwd
184	capget
185	capset
186	sigaltstack
187	sendfile
190	vfork

6 Appendix C: LINUX.HHF Header File (as of 3/22/2002)