NAME

new system call − How to add a new system call to libexplain

DESCRIPTION

Adding a new system call to libexplain is both simple and tedious.

In this example, the system call is called *example*, and takes two arguments, *pathname* and *flags*.

```
example(const char *pathname, int flags);
```

The libexplain library presents a C interface to the user, and explains the C system calls. It tries to avoid dynamic memory, and has several helper functions and structures to make this simpler.

Naming Conventions

In general, one function per file. This gives the static linker more opportunity to leave things out, thus producing smaller executables. Exceptions to make use of `static` common functions are acceptable. No savings for shared libraries, of course.

Functions that write their output into a *explain_string_buffer_t* via the `explain_string_buffer_*` functions, all have a filename of `libexplain/buffer/`*something*.

Functions that write their output to a *message*, *message_size* pair have a `message` path component in their file name.

Functions that accept an *errno* value as an argument have an `errno` path component in their file name, called `errnum`. If a function has both a buffer and an errno, the buffer comes first, both in the argument list, and the file's name. If a function has both a message+size and an errno, the message comes first, both in the argument list, and the file's name.

MODIFIED FILES

Note that the *codegen* command does most of the work for you. Pass it the function prototype (in single quotes) and it will do most of the work.

```
$ bin/codegen 'example(const char *pathname, int flags);'
creating catalogue/example
$
```

then you mast edit the `catalogue/example` file to make any adjustment necessary. This file is then used to do the boring stuff:

```
$ bin/codegen example
creating explain/syscall/example.c
creating explain/syscall/example.h
creating libexplain/buffer/errno/example.c
creating libexplain/buffer/errno/example.h
creating libexplain/example.c
creating libexplain/example.h
creating libexplain/example_or_die.c
creating man/man3/explain_example.3
creating man/man3/explain_example_or_die.3
creating test_example/main.c
modify explain/syscall.c
modify libexplain/libexplain.h
modify man/man1/explain.1
modify man/man3/explain.3
$
```

All of these files have been added to the Aegis change set. Edit the last 4 to place the appended line in their correct positions within the files, respecting the symbol sort ordering of each file.

libexplain/libexplain.h

The `libexplain/libexplain.h` include file defines the user API. It, and any files it includes, are installed into `$(prefix)/include` by *make install*.

This file needs another include line. This means that the entire API is available to the user as a single

include directive.

```
#include <libexplain/example.h>
```

This file is also used to decide which files are installed by the *make install* command.

Take care that none of those files, directly or indirectly, wind up including `libexplain/config.h` which is generated by the *configure* script, and has **no** namespace protection.

This means you can't `#include <stddef.h>`, or use any of the types it defines, because on older systems *configure* works quite hard to cope with its absence. Ditto `<unistd.h>` and `<sys/types.h>`.

explain/main.c
Include the include file for the new function, and add the function to the table.

man/man1/explain.1
Add a description of the new system call.

man/man3/libexplain.3
Add your new man pages, man/man3/explain_*example*.3 and man/man3/explain_*example*_or_die.3, to the list. Keep the list sorted.

NEW FILES

Note that the *codegen* command does most of the work for you. Pass it the function prototype (in single quotes) and it will do most of the work.

libexplain/buffer/errno/*example*.c
The central file for adding a new example is `libexplain/buffer/errno/`*example*`.c` Which defines a function

```
void explain_buffer_errno_example(explain_string_buffer_t *buffer,
int errnum, const char *pathname, int flags);
```

The `errnum` argument holds the *errno* value. Note that calling *errno* usually has problems because many systems have *errno* as a macro, which makes the compiler barf, and because there are times you want access to the global *errno*, and having it shadowed by the argument is a nuisance.

This function writes its output into the buffer via the `explain_string_buffer_printf`, *etc*, functions. First the argument list is reprinted.

The `explain_string_buffer_puts_quoted` function should be used to print pathnames, because it uses full C quoting and escape sequences.

If an argument is a file descriptor, it should be called *fildes*, short for "file descriptor". On systems capable of it, the file descriptor can be mapped to a pathname using the `explain_buffer_fildes_to_pathname` function. This makes explanations for system calls like *read* and *write* much more informative.

Next comes a switch on the errnum value, and additional explanation is given for each errno value documented (or sometimes undocumented) for that system call. Copy-and-paste of the man page is often useful as a basis for the text of the explanation, but be sure it is open source documentation, and not Copyright proprietary text.

Don't forget to check the existing `libexplain/buffer/e*.h` files for pre-canned explanations for common errors. Some pre-canned explanations include

EACCES	explain_buffer_eacces
EADDRINUSE	explain_buffer_eaddrinuse
EAFNOSUPPORT	explain_buffer_eafnosupport
EBADF	explain_buffer_ebadf
EFAULT	explain_buffer_efault
EFBIG	explain_buffer_efbig
EINTR	explain_buffer_eintr
EINVAL	explain_buffer_einval_vague, *etc*

EIO	explain_buffer_eio
ELOOP	explain_buffer_eloop
EMFILE	explain_buffer_emfile
EMLINK	explain_buffer_emlink
ENAMETOOLONG	explain_buffer_enametoolong
ENFILE	explain_buffer_enfile
ENOBUFS	explain_buffer_enobufs
ENOENT	explain_buffer_enoent
ENOMEM	explain_buffer_enomem
ENOTCONN	explain_buffer_enotconn
ENOTDIR	explain_buffer_enotdir
ENOTSOCK	explain_buffer_enotsock
EROFS	explain_buffer_erofs
ETXTBSY	explain_buffer_etxtbsy
EXDEV	explain_buffer_exdev

libexplain/buffer/errno/example.h
This file holds the function prototype for the above function definition.

libexplain/example.h
The file contains the user visible API for the *example* system call. There are five function prototypes declared in this file:

```
void explain_example_or_die(const char *pathname, int flags);
void explain_example( const char *pathname, int flags);
void explain_errno_example(int errnum, const char *pathname, int flags);
void explain_message_example(const char *message, int message_size,
const char *pathname, int flags);
void explain_message_errno_example(const char *message, int mes-
sage_size, int errnum, const char *pathname, int flags);
```

The function prototypes for these appear in the `libexplain/`*example*`.h include file.`

Each function prototype shall be accompanied by thorough Doxygen style comments. These are extracted and placed on the web site.

The buffer functions are **never** part of the user visible API.

libexplain/*example***_or_die.c**
One function per file, `explain_`*example*`_or_die` in this case. It simply calls *example* and then, if fails, `explain_`*example* to print why, and then exit(EXIT_FAILURE).

libexplain/example.c
One function per file, `explain_`*example* in this case. It simply calls `explain_errno_`*example* to pass in the global *errno* value.

libexplain/errno/example.c
One function per file, `explain_errno_`*example* in this case. It calls `explain_message_errno_`*example*, using the `<libexplain/global_message_buffer.h>` to hold the string.

libexplain/message/example.c
One function per file, `explain_message_`*example* in this case. It simply calls `explain_message_errno_`*example* to pass in the global *errno* value.

libexplain/message/errno/example.c
One function per file, `explain_message_errno_`*example* in this case. It declares and initializes a `explain_string_buffer_t` instance, which ensures that the message buffer will not be exceeded, and passes that buffer to the `explain_buffer_errno_`*example* function.

man/man3/explain_example.3
This file also documents the error explanations functions, except `explain_`*example*`_or_dir`. Use the same text as you did in `libexplain/`*example*`.h`

man/man3/explain_example_or_die.3
This file also documents the helper function. Use the same text as you did in `libexplain/`*example*`.h`

explain/example.c
Glue to turn the command line into arguments to a call to `explain_`*example*

explain/example.h
Function prototype for the above.

test_example/main.c
This program should call `explain_`*explain*`_or_die`.

NEW IOCTL REQUESTS

Each different *ioctl*(2) request is, in effect, yet another system call. Except that they all have appallingly bad type safety. I have seen fugly C++ classes with less overloading than *ioctl*(2).

libexplain/iocontrol/request_by_number.c
This file has one include line for each *ioctl*(2) request. There is a `table` array that contains a pointer to the explain_iocontrol_t variable declared in the include file (see next). Keep both sets of lines sorted alphabetically, it makes it easier to detect duplicates.

libexplain/iocontrol/*name*.h
Where *name* is the name of the *ioctl*(2) request in lower case. This declares an global const variable describing how to handle it.

libexplain/iocontrol/*name*.c
This defines the above global variable, and defines any static glue functions necessary to print a representation of it. You will probably have to read the kernel source to discover the errors the ioctl can return, and what causes them, in order to write the explanation function; they are almost never described in the man pages.

TESTS

Write at least one separate test for each case in the errnum switch.

Debian Notes

You can check that the Debian stuff builds by using
apt-get install pbuilder
pbuiler create
pbuilder login
now copy the files from *web-site/debian/* into the chroot
cd libexplain−*
dpkg−checkbuilddeps
apt−get install *what dpkg−checkbuilddeps said*
apt−get install devscripts
debuild
This should report success.

COPYRIGHT

AUTHOR

Written by Peter Miller <pmiller@opensource.org.au>