NAME

explain_lca2010 − No medium found: when it's time to stop trying to read *strerror*(3)'s mind.

MOTIVATION

The idea for libexplain occurred to me back in the early 1980s. Whenever a system call returns an error, the kernel knows exactly what went wrong... and compresses this into less that 8 bits of *errno*. User space has access to the same data as the kernel, it should be possible for user space to figure out exactly what happened to provoke the error return, and use this to write good error messages.

Could it be that simple?

Error messages as finesse

Good error messages are often those "one percent" tasks that get dropped when schedule pressure squeezes your project. However, a good error message can make a huge, disproportionate improvement to the user experience, when the user wanders into scarey unknown territory not usually encountered. This is no easy task.

As a larval programmer, the author didn't see the problem with (completely accurate) error messages like this one:

```
floating exception (core dumped)
```

until the alternative non-programmer interpretation was pointed out. But that isn't the only thing wrong with Unix error messages. How often do you see error messages like:

```
$ ./stupid
can't open file
$
```

There are two options for a developer at this point:

1. you can run a debugger, such as *gdb*(1), or
2. you can use *strace*(1) or *truss*(1) to look inside.

- Remember that your users may not even have access to these tools, let alone the ability to use them. (It's a very long time since *Unix beginner* meant "has only written *one* device driver".)

In this example, however, using *strace*(1) reveals

```
$ strace -e trace=open ./stupid
open("some/file", O_RDONLY) = -1 ENOENT (No such file or directory)
can't open file
$
```

This is considerably more information than the error message provides. Typically, the stupid source code looks like this

```
int fd = open("some/thing", O_RDONLY);
if (fd < 0)
{
    fprintf(stderr, "can't open file\n");
    exit(1);
}
```

The user isn't told *which* file, and also fails to tell the user *which* error. Was the file even there? Was there a permissions problem? It does tell you it was trying to open a file, but that was probably by accident.

Grab your clue stick and go beat the larval programmer with it. Tell him about *perror*(3). The next time you use the program you see a different error message:

```
$ ./stupid
open: No such file or directory
$
```

Progress, but not what we expected. How can the user fix the problem if the error message doesn't tell him what the problem was? Looking at the source, we see

```
int fd = open("some/thing", O_RDONLY);
if (fd < 0)
{
    perror("open");
    exit(1);
}
```

Time for another run with the clue stick. This time, the error message takes one step forward and one step back:

```
$ ./stupid
some/thing: No such file or directory
$
```

Now we know the file it was trying to open, but are no longer informed that it was *open*(2) that failed. In this case it is probably not significant, but it can be significant for other system calls. It could have been *creat*(2) instead, an operation implying that different permissions are necessary.

```
const char *filename = "some/thing";
int fd = open(filename, O_RDONLY);
if (fd < 0)
{
    perror(filename);
    exit(1);
}
```

The above example code is unfortunately typical of non-larval programmers as well. Time to tell our padawan learner about the *strerror*(3) system call.

```
$ ./stupid
open some/thing: No such file or directory
$
```

This maximizes the information that can be presented to the user. The code looks like this:

```
const char *filename = "some/thing";
int fd = open(filename, O_RDONLY);
if (fd < 0)
{
    fprintf(stderr, "open %s: %s\n", filename, strerror(errno));
    exit(1);
}
```

Now we have the system call, the filename, and the error string. This contains all the information that *strace*(1) printed. That's as good as it gets.

Or is it?

Limitations of `perror` and `strerror`

The problem the author saw, back in the 1980s, was that the error message is incomplete. Does "no such file or directory" refer to the "*some*" directory, or to the "*thing*" file in the "*some*" directory?

A quick look at the man page for *strerror*(3) is telling:

> strerror − return string describing error number

Note well: it is describing the error *number*, not the error.

On the other hand, the kernel *knows* what the error was. There was a specific point in the kernel code, caused by a specific condition, where the kernel code branched and said "no". Could a user-space program figure out the specific condition and write a better error message?

However, the problem goes deeper. What if the problem occurs during the *read*(2) system call, rather than the *open*(2) call? It is simple for the error message associated with *open*(2) to include the file name, it's right there. But to be able to include a file name in the error associated with the *read*(2) system call, you

have to pass the file name all the way down the call stack, as well as the file descriptor.

And here is the bit that grates: the kernel already knows what file name the file descriptor is associated with. Why should a programmer have to pass redundant data all the way down the call stack just to improve an error message that may never be issued? In reality, many programmers don't bother, and the resulting error messages are the worse for it.

But that was the 1980s, on a PDP11, with limited resources and no shared libraries. Back then, no flavor of Unix included `/proc` even in rudimentary form, and the *lsof*(1) program was over a decade away. So the idea was shelved as impractical.

Level Infinity Support

Imagine that you are level infinity support. Your job description says that you never *ever* have to talk to users. Why, then, is there still a constant stream of people wanting you, the local Unix guru, to decipher yet another error message?

Strangely, 25 years later, despite a simple permissions system, implemented with complete consistency, most Unix users still have no idea how to decode "No such file or directory", or any of the other cryptic error messages they see every day. Or, at least, cryptic to them.

Wouldn't it be nice if first level tech support didn't need error messages deciphered? Wouldn't it be nice to have error messages that users could understand without calling tech support?

These days `/proc` on Linux is more than able to provide the information necessary to decode the vast majority of error messages, and point the user to the proximate cause of their problem. On systems with a limited `/proc` implementation, the *lsof*(1) command can fill in many of the gaps.

In 2008, the stream of translation requests happened to the author way too often. It was time to re-examine that 25 year old idea, and libexplain is the result.

USING THE LIBRARY

The interface to the library tries to be consistent, where possible. Let's start with an example using *strerror*(3):

```
if (rename(old_path, new_path) < 0)
{
    fprintf(stderr, "rename %s %s: %s\n", old_path, new_path,
        strerror(errno));
    exit(1);
}
```

The idea behind libexplain is to provide a *strerror*(3) equivalent for **each** system call, tailored specifically to that system call, so that it can provide a more detailed error message, containing much of the information you see under the "ERRORS" heading of section 2 and 3 *man* pages, supplemented with information about actual conditions, actual argument values, and system limits.

The Simple Case

The *strerror*(3) replacement:

```
if (rename(old_path, new_path) < 0)
{
    fprintf(stderr, "%s\n", explain_rename(old_path, new_path));
    exit(1);
}
```

The Errno Case

It is also possible to pass an explicit *errno*(3) value, if you must first do some processing that would disturb *errno*, such as error recovery:

```
if (rename(old_path, new_path < 0))
{
    int old_errno = errno;
    ...code that disturbs errno...
```

```
        fprintf(stderr, "%s\n", explain_errno_rename(old_errno,
            old_path, new_path));
        exit(1);
    }
```

The Multi-thread Cases

Some applications are multi-threaded, and thus are unable to share libexplain's internal buffer. You can supply your own buffer using

```
if (unlink(pathname))
{
    char message[3000];
    explain_message_unlink(message, sizeof(message), pathname);
    error_dialog(message);
    return -1;
}
```

And for completeness, both *errno*(3) and thread-safe:

```
ssize_t nbytes = read(fd, data, sizeof(data));
if (nbytes < 0)
{
    char message[3000];
    int old_errno = errno;
    ...error recovery...
    explain_message_errno_read(message, sizeof(message),
        old_errno, fd, data, sizeof(data));
    error_dialog(message);
    return -1;
}
```

These are replacements for *strerror_r*(3), on systems that have it.

Interface Sugar

A set of functions added as convenience functions, to woo programmers to use the libexplain library, turn out to be the author's most commonly used libexplain functions in command line programs:

```
int fd = explain_creat_or_die(filename, 0666);
```

This function attempts to create a new file. If it can't, it prints an error message and exits with EXIT_FAILURE. If there is no error, it returns the new file descriptor.

A related function:

```
int fd = explain_creat_on_error(filename, 0666);
```

will print the error message on failure, but also returns the original error result, and *errno*(3) is unmolested, as well.

All the other system calls

In general, every system call has its own include file

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

that defines function prototypes for six functions:

- `explain_`*name*,
- `explain_errno_`*name*,
- `explain_message_`*name*,
- `explain_message_errno_`*name*,
- `explain_`*name*`_or_die` and

- `explain_`*name*`_on_error`.

Every function prototype has Doxygen documentation, and this documentation *is not* stripped when the include files are installed.

The *wait*(2) system call (and friends) have some extra variants that also interpret failure to be an exit status that isn't EXIT_SUCCESS. This applies to *system*(3) and *pclose*(3) as well.

Coverage includes 221 system calls and 547 ioctl requests. There are many more system calls yet to implement. System calls that never return, such as *exit*(2), are not present in the library, and will never be. The *exec* family of system calls *are* supported, because they return when there is an error.

Cat

This is what a hypothetical "cat" program could look like, with full error reporting, using libexplain.

```
#include <libexplain/libexplain.h>
#include <stdlib.h>
#include <unistd.h>
```

There is one include for libexplain, plus the usual suspects. (If you wish to reduce the preprocessor load, you can use the specific `<libexplain/`*name*`.h>` includes.)

```
static void
process(FILE *fp)
{
    for (;;)
    {
        char buffer[4096];
        size_t n = explain_fread_or_die(buffer, 1, sizeof(buffer), fp);
        if (!n)
            break;
        explain_fwrite_or_die(buffer, 1, n, stdout);
    }
}
```

The *process* function copies a file stream to the standard output. Should an error occur for either reading or writing, it is reported (and the pathname will be included in the error) and the command exits with EXIT_FAILURE. We don't even worry about tracking the pathnames, or passing them down the call stack.

```
int
main(int argc, char **argv)
{
    for (;;)
    {
        int c = getopt(argc, argv, "o:");
        if (c == EOF)
            break;
        switch (c)
        {
        case 'o':
            explain_freopen_or_die(optarg, "w", stdout);
            break;
```

The fun part of this code is that libexplain can report errors *including the pathname* even if you **don't** explicitly re-open stdout as is done here. We don't even worry about tracking the file name.

```
        default:
            fprintf(stderr, "Usage: %ss [ -o <filename> ] <filename>...\n",
                argv[0]);
            return EXIT_FAILURE;
        }
    }
```

```
    if (optind == argc)
        process(stdin);
    else
    {
        while (optind < argc)
        {
            FILE *fp = explain_fopen_or_die(argv[optind]++, "r");
            process(fp);
            explain_fclose_or_die(fp);
        }
    }
```

The standard output will be closed implicitly, but too late for an error report to be issued, so we do that here, just in case the buffered I/O hasn't written anything yet, and there is an ENOSPC error or something.

```
    explain_fflush_or_die(stdout);
    return EXIT_SUCCESS;
}
```

That's all. Full error reporting, clear code.

Rusty's Scale of Interface Goodness

For those of you not familiar with it, Rusty Russel's "How Do I Make This Hard to Misuse?" page is a must-read for API designers.
http://ozlabs.org/~rusty/index.cgi/tech/2008-03-30.html

10. It's impossible to get wrong.

Goals need to be set high, ambitiously high, lest you accomplish them and think you are finished when you are not.

The libexplain library detects bogus pointers and many other bogus system call parameters, and generally tries to avoid segfaults in even the most trying circumstances.

The libexplain library is designed to be thread safe. More real-world use will likely reveal places this can be improved.

The biggest problem is with the actual function names themselves. Because C does not have name-spaces, the libexplain library always uses an `explain_` name prefix. This is the traditional way of creating a pseudo-name-space in order to avoid symbol conflicts. However, it results in some unnatural-sounding names.

9. The compiler or linker won't let you get it wrong.

A common mistake is to use `explain_open` where `explain_open_or_die` was intended. Fortunately, the compiler will often issue a type error at this point (*e.g.* can't assign `const char *` rvalue to an `int` lvalue).

8. The compiler will warn if you get it wrong.

If `explain_rename` is used when `explain_rename_or_die` was intended, this can cause other problems. GCC has a useful `warn_unused_result` function attribute, and the libexplain library attaches it to all the `explain_`*name* function calls to produce a warning when you make this mistake. Combine this with *gcc −Werror* to promote this to level 9 goodness.

7. The obvious use is (probably) the correct one.

The function names have been chosen to convey their meaning, but this is not always successful. While `explain_`*name*`_or_die` and `explain_`*name*`_on_error` are fairly descriptive, the less-used thread safe variants are harder to decode. The function prototypes help the compiler towards understanding, and the Doxygen comments in the header files help the user towards understanding.

6. The name tells you how to use it.

It is particularly important to read `explain_`*name*`_or_die` as "explain (*name* or die)". Using a

consistent `explain_` name-space prefix has some unfortunate side-effects in the obviousness department, as well.

The order of words in the names also indicate the order of the arguments. The argument lists always *end* with the same arguments as passed to the system call; *all of them*. If `_errno_` appears in the name, its argument always precedes the system call arguments. If `_message_` appears in the name, its two arguments always come first.

5. Do it right or it will break at runtime.

The libexplain library detects bogus pointers and many other bogus system call parameters, and generally tries to avoid segfaults in even the most trying circumstances. It should never break at runtime, but more real-world use will no doubt improve this.

Some error messages are aimed at developers and maintainers rather than end users, as this can assist with bug resolution. Not so much "break at runtime" as "be informative at runtime" (after the system call barfs).

4. Follow common convention and you'll get it right.

Because C does not have name-spaces, the libexplain library always uses an `explain_` name prefix. This is the traditional way of creating a pseudo-name-space in order to avoid symbol conflicts.

The trailing arguments of all the libexplain call are identical to the system call they are describing. This is intended to provide a consistent convention in common with the system calls themselves.

3. Read the documentation and you'll get it right.

The libexplain library aims to have complete Doxygen documentation for each and every public API call (and internally as well).

MESSAGE CONTENT

Working on libexplain is a bit like looking at the underside of your car when it is up on the hoist at the mechanic's. There's some ugly stuff under there, plus mud and crud, and users rarely see it. A good error message needs to be informative, even for a user who has been fortunate enough not to have to look at the under-side very often, and also informative for the mechanic listening to the user's description over the phone. This is no easy task.

Revisiting our first example, the code would like this if it uses libexplain:

```
int fd = explain_open_or_die("some/thing", O_RDONLY, 0);
```

will fail with an error message like this

```
open(pathname = "some/file", flags = O_RDONLY) failed, No such
file or directory (2, ENOENT) because there is no "some" directory
in the current directory
```

This breaks down into three pieces

system-call `failed,` *system-error* `because`
explanation

Before Because

It is possible to see the part of the message before "because" as overly technical to non-technical users, mostly as a result of accurately printing the system call itself at the beginning of the error message. And it looks like *strace*(1) output, for bonus geek points.

```
open(pathname = "some/file", flags = O_RDONLY) failed, No such
file or directory (2, ENOENT)
```

This part of the error message is essential to the developer when he is writing the code, and equally important to the maintainer who has to read bug reports and fix bugs in the code. It says exactly what failed.

If this text is not presented to the user then the user cannot copy-and-paste it into a bug report, and if it isn't in the bug report the maintainer can't know what actually went wrong.

Frequently tech staff will use *strace*(1) or *truss*(1) to get this exact information, but this avenue is not open

when reading bug reports. The bug reporter's system is far far away, and, by now, in a far different state. Thus, this information needs to be in the bug report, which means it must be in the error message.

The system call representation also gives context to the rest of the message. If need arises, the offending system call argument may be referred to by name in the explanation after "because". In addition, all strings are fully quoted and escaped C strings, so embedded newlines and non-printing characters will not cause the user's terminal to go haywire.

The *system-error* is what comes out of *strerror*(2), plus the error symbol. Impatient and expert sysadmins could stop reading at this point, but the author's experience to date is that reading further is rewarding. (If it isn't rewarding, it's probably an area of libexplain that can be improved. Code contributions are welcome, of course.)

After Because

This is the portion of the error message aimed at non-technical users. It looks beyond the simple system call arguments, and looks for something more specific.

```
there is no "some" directory in the current directory
```

This portion attempts to explain the proximal cause of the error in plain language, and it is here that internationalization is essential.

In general, the policy is to include as much information as possible, so that the user doesn't need to go looking for it (and doesn't leave it out of the bug report).

Internationalization

Most of the error messages in the libexplain library have been internationalized. There are no localizations as yet, so if you want the explanations in your native language, please contribute.

The "most of" qualifier, above, relates to the fact that the proof-of-concept implementation did not include internationalization support. The code base is being revised progressively, usually as a result of refactoring messages so that each error message string appears in the code exactly once.

Provision has been made for languages that need to assemble the portions of

system-call `failed,` *system-error* `because` *explanation*

in different orders for correct grammar in localized error messages.

Postmortem

There are times when a program has yet to use libexplain, and you can't use *strace*(1) either. There is an *explain*(1) command included with libexplain that can be used to decipher error messages, if the state of the underlying system hasn't changed too much.

```
$ explain rename foo /tmp/bar/baz -e ENOENT
rename(oldpath = "foo", newpath = "/tmp/bar/baz") failed, No such
file or directory (2, ENOENT) because there is no "bar" directory
in the newpath "/tmp" directory
$
```

Note how the path ambiguity is resolved by using the system call argument name. Of course, you have to know the error and the system call for *explain*(1) to be useful. As an aside, this is one of the ways used by the libexplain automatic test suite to verify that libexplain is working.

Philosophy

"Tell me everything, including stuff I didn't know to look for."

The library is implemented in such a way that when statically linked, only the code you actually use will be linked. This is achieved by having one function per source file, whenever feasible.

When it is possible to supply more information, libexplain will do so. The less the user has to track down for themselves, the better. This means that UIDs are accompanied by the user name, GIDs are accompanied by the group name, PIDs are accompanied by the process name, file descriptors and streams are accompanied by the pathname, *etc*.

When resolving paths, if a path component does not exist, libexplain will look for similar names, in order to

suggest alternatives for typographical errors.

The libexplain library tries to use as little heap as possible, and usually none. This is to avoid perturbing the process state, as far as possible, although sometimes it is unavoidable.

The libexplain library attempts to be thread safe, by avoiding global variables, keeping state on the stack as much as possible. There is a single common message buffer, and the functions that use it are documented as not being thread safe.

The libexplain library does not disturb a process's signal handlers. This makes determining whether a pointer would segfault a challenge, but not impossible.

When information is available via a system call as well as available through a `/proc` entry, the system call is preferred. This is to avoid disturbing the process's state. There are also times when no file descriptors are available.

The libexplain library is compiled with large file support. There is no large/small schizophrenia. Where this affects the argument types in the API, and error will be issued if the necessary large file defines are absent.

FIXME: Work is needed to make sure that file system quotas are handled in the code. This applies to some *getrlimit*(2) boundaries, as well.

There are cases when relatives paths are uninformative. For example: system daemons, servers and background processes. In these cases, absolute paths are used in the error explanations.

PATH RESOLUTION

Short version: see *path_resolution*(7).

Long version: Most users have never heard of *path_resolution*(7), and many advanced users have never read it. Here is an annotated version:

Step 1: Start of the resolution process

If the pathname starts with the slash ("/") character, the starting lookup directory is the root directory of the calling process.

If the pathname does not start with the slash("/") character, the starting lookup directory of the resolution process is the current working directory of the process.

Step 2: Walk along the path

Set the current lookup directory to the starting lookup directory. Now, for each non-final component of the pathname, where a component is a substring delimited by slash ("/") characters, this component is looked up in the current lookup directory.

If the process does not have search permission on the current lookup directory, an EACCES error is returned ("Permission denied").

```
open(pathname = "/home/archives/.ssh/private_key", flags =
O_RDONLY) failed, Permission denied (13, EACCES) because the
process does not have search permission to the pathname "/home/ar-
chives/.ssh" directory, the process effective GID 1000 "pmiller"
does not match the directory owner 1001 "archives" so the owner
permission mode "rwx" is ignored, the others permission mode is
"---", and the process is not privileged (does not have the
DAC_READ_SEARCH capability)
```

If the component is not found, an ENOENT error is returned ("No such file or directory").

```
unlink(pathname = "/home/microsoft/rubbish") failed, No such file
or directory (2, ENOENT) because there is no "microsoft" directory
in the pathname "/home" directory
```

There is also some support for users when they mis-type pathnames, making suggestions when ENOENT is returned:

```
open(pathname = "/user/include/fcntl.h", flags = O_RDONLY) failed,
No such file or directory (2, ENOENT) because there is no "user"
directory in the pathname "/" directory, did you mean the "usr"
directory instead?
```

If the component is found, but is neither a directory nor a symbolic link, an ENOTDIR error is returned ("Not a directory").

```
open(pathname = "/home/pmiller/.netrc/lca", flags = O_RDONLY)
failed, Not a directory (20, ENOTDIR) because the ".netrc" regular
file in the pathname "/home/pmiller" directory is being used as a
directory when it is not
```

If the component is found and is a directory, we set the current lookup directory to that directory, and go to the next component.

If the component is found and is a symbolic link (symlink), we first resolve this symbolic link (with the current lookup directory as starting lookup directory). Upon error, that error is returned. If the result is not a directory, an ENOTDIR error is returned.

```
unlink(pathname = "/tmp/dangling/rubbish") failed, No such file or
directory (2, ENOENT) because the "dangling" symbolic link in the
pathname "/tmp" directory refers to "nowhere" that does not exist
```

If the resolution of the symlink is successful and returns a directory, we set the current lookup directory to that directory, and go to the next component. Note that the resolution process here involves recursion. In order to protect the kernel against stack overflow, and also to protect against denial of service, there are limits on the maximum recursion depth, and on the maximum number of symbolic links followed. An ELOOP error is returned when the maximum is exceeded ("Too many levels of symbolic links").

```
open(pathname = "/tmp/dangling", flags = O_RDONLY) failed, Too
many levels of symbolic links (40, ELOOP) because a symbolic link
loop was encountered in pathname, starting at "/tmp/dangling"
```

It is also possible to get an ELOOP or EMLINK error if there are too many symlinks, but no loop was detected.

```
open(pathname = "/tmp/rabbit-hole", flags = O_RDONLY) failed, Too
many levels of symbolic links (40, ELOOP) because too many sym-
bolic links were encountered in pathname (8)
```

Notice how the actual limit is also printed.

Step 3: Find the final entry

The lookup of the final component of the pathname goes just like that of all other components, as described in the previous step, with two differences:

(i) The final component need not be a directory (at least as far as the path resolution process is concerned. It may have to be a directory, or a non-directory, because of the requirements of the specific system call).

(ii) It is not necessarily an error if the final component is not found; maybe we are just creating it. The details on the treatment of the final entry are described in the manual pages of the specific system calls.

(iii) It is also possible to have a problem with the last component if it is a symbolic link and it should not be followed. For example, using the *open*(2) O_NOFOLLOW flag:

```
open(pathname = "a-symlink", flags = O_RDONLY | O_NOFOLLOW) failed,
Too many levels of symbolic links (ELOOP) because O_NOFOLLOW was
specified but pathname refers to a symbolic link
```

(iv) It is common for users to make mistakes when typing pathnames. The libexplain library attempts to make suggestions when ENOENT is returned, for example:

```
open(pathname = "/usr/include/filecontrl.h", flags = O_RDONLY) failed, No such file or directory (2, ENOENT) because there is no "filecontrl.h" regular file in the pathname "/usr/include" directory, did you mean the "fcntl.h" regular file instead?
```

(v) It is also possible that the final component is required to be something other than a regular file:

```
readlink(pathname = "just-a-file", data = 0x7F930A50, data_size = 4097) failed, Invalid argument (22, EINVAL) because pathname is a regular file, not a symbolic link
```

(vi) FIXME: handling of the "t" bit.

Limits

There are a number of limits with regards to pathnames and filenames.

Pathname length limit

There is a maximum length for pathnames. If the pathname (or some intermediate pathname obtained while resolving symbolic links) is too long, an ENAMETOOLONG error is returned ("File name too long"). Notice how the system limit is included in the error message.

```
open(pathname = "very...long", flags = O_RDONLY) failed, File name too long (36, ENAMETOOLONG) because pathname exceeds the system maximum path length (4096)
```

Filename length limit

Some Unix variants have a limit on the number of bytes in each path component. Some of them deal with this silently, and some give ENAMETOOLONG; the libexplain library uses *pathconf*(3) _PC_NO_TRUNC to tell which. If this error happens, the libexplain library will state the limit in the error message, the limit is obtained from *pathconf*(3) _PC_NAME_MAX. Notice how the system limit is included in the error message.

```
open(pathname = "system7/only-had-14-characters", flags = O_RDONLY) failed, File name too long (36, ENAMETOOLONG) because "only-had-14-characters" component is longer than the system limit (14)
```

Empty pathname

In the original Unix, the empty pathname referred to the current directory. Nowadays POSIX decrees that an empty pathname must not be resolved successfully.

```
open(pathname = "", flags = O_RDONLY) failed, No such file or directory (2, ENOENT) because POSIX decrees that an empty pathname must not be resolved successfully
```

Permissions

The permission bits of a file consist of three groups of three bits. The first group of three is used when the effective user ID of the calling process equals the owner ID of the file. The second group of three is used when the group ID of the file either equals the effective group ID of the calling process, or is one of the supplementary group IDs of the calling process. When neither holds, the third group is used.

```
open(pathname = "/etc/passwd", flags = O_WRONLY) failed, Permission denied (13, EACCES) because the process does not have write permission to the "passwd" regular file in the pathname "/etc" directory, the process effective UID 1000 "pmiller" does not match the regular file owner 0 "root" so the owner permission mode "rw-" is ignored, the others permission mode is "r--", and the process is not privileged (does not have the DAC_OVERRIDE capability)
```

Some considerable space is given to this explanation, as most users do not know that this is how the permissions system works. In particular: the owner, group and other permissions are exclusive, they are not

"OR"ed together.

STRANGE AND INTERESTING SYSTEM CALLS

The process of writing a specific error handler for each system call often reveals interesting quirks and boundary conditions, or obscure *errno*(3) values.

ENOMEDIUM, No medium found

The act of copying a CD was the source of the title for this paper.

```
$ dd if=/dev/cdrom of=fubar.iso
dd: opening "/dev/cdrom": No medium found
$
```

The author wondered why his computer was telling him there is no such thing as a psychic medium. Quite apart from the fact that huge numbers of native English speakers are not even aware that "media" is a plural, let alone that "medium" is its singular, the string returned by *strerror*(3) for ENOMEDIUM is so terse as to be almost completely free of content.

When *open*(2) returns ENOMEDIUM it would be nice if the libexplain library could expand a little on this, based on the type of drive it is. For example:

... because there is no disk in the floppy drive
... because there is no disc in the CD-ROM drive
... because there is no tape in the tape drive
... because there is no memory stick in the card reader

And so it came to pass...

```
open(pathname = "/dev/cdrom", flags = O_RDONLY) failed, No medium
found (123, ENOMEDIUM) because there does not appear to be a disc
in the CD-ROM drive
```

The trick, that the author was previously unaware of, was to open the device using the O_NONBLOCK flag, which will allow you to open a drive with no medium in it. You then issue device specific *ioctl*(2) requests until you figure out what the heck it is. (Not sure if this is POSIX, but it also seems to work that way in BSD and Solaris, according to the *wodim*(1) sources.)

Note also the differing uses of "disk" and "disc" in context. The CD standard originated in France, but everything else has a "k".

EFAULT, Bad address

Any system call that takes a pointer argument can return EFAULT. The libexplain library can figure out which argument is at fault, and it does it without disturbing the process (or thread) signal handling.

When available, the *mincore*(2) system call is used, to ask if the memory region is valid. It can return three results: mapped but not in physical memory, mapped and in physical memory, and not mapped. When testing the validity of a pointer, the first two are "yes" and the last one is "no".

Checking C strings are more difficult, because instead of a pointer and a size, we only have a pointer. To determine the size we would have to find the NUL, and that could segfault, catch-22.

To work around this, the libexplain library uses the *lstat*(2) sysem call (with a known good second argument) to test C strings for validity. A failure return && errno == EFAULT is a "no", and anythng else is a "yes". This, of course limits strings to PATH_MAX characters, but that usually isn't a problem for the libexplain library, because that is almost always the longest strings it cares about.

EMFILE, Too many open files

This error occurs when a process already has the maximum number of file descriptors open. If the actual limit is to be printed, and the libexplain library tries to, you can't open a file in `/proc` to read what it is.

```
open_max = sysconf(_SC_OPEN_MAX);
```

This one wan't so difficult, there is a *sysconf*(3) way of obtaining the limit.

ENFILE, Too many open files in system

This error occurs when the system limit on the total number of open files has been reached. In this case there is no handy *sysconf*(3) way of obtain the limit.

Digging deeper, one may discover that on Linux there is a `/proc` entry we could read to obtain this value. Catch-22: we are out of file descriptors, so we can't open a file to read the limit.

On Linux there is a system call to obtain it, but it has no [e]glibc wrapper function, so you have to all it very carefully:

```
long
explain_maxfile(void)
{
#ifdef __linux__
    struct __sysctl_args args;
    int32_t maxfile;
    size_t maxfile_size = sizeof(maxfile);
    int name[] = { CTL_FS, FS_MAXFILE };
    memset(&args, 0, sizeof(struct __sysctl_args));
    args.name = name;
    args.nlen = 2;
    args.oldval = &maxfile;
    args.oldlenp = &maxfile_size;
    if (syscall(SYS__sysctl, &args) >= 0)
        return maxfile;
#endif
    return -1;
}
```

This permits the limit to be included in the error message, when available.

EINVAL "Invalid argument" *vs* **ENOSYS "Function not implemented"**

Unsupported actions (such as *symlink*(2) on a FAT file system) are not reported consistently from one system call to the next. It is possible to have either EINVAL or ENOSYS returned.

As a result, attention must be paid to these error cases to get them right, particularly as the EINVAL could also be referring to problems with one or more system call arguments.

Note that *errno*(3) is not always set

There are times when it is necessary to read the [e]glibc sources to determine how and when errors are returned for some system calls.

feof(3), *fileno*(3)

It is often assumed that these functions cannot return an error. This is only true if the *stream* argument is valid, however they are capable of detecting an invalid pointer.

fpathconf(3), *pathconf*(3)

The return value of *fpathconf*(2) and *pathconf*(2) could legitimately be −1, so it is necessary to see if *errno*(3) has been explicitly set.

ioctl(2)

The return value of *ioctl*(2) could legitimately be −1, so it is necessary to see if *errno*(3) has been explicitly set.

readdir(3)

The return value of *readdir*(3) is NULL for both errors and end-of-file. It is necessary to see if *errno*(3) has been explicitly set.

setbuf(3), *setbuffer*(3), *setlinebuf*(3), *setvbuf*(3)

All but the last of these functions return void. And *setvbuf*(3) is only documented as returning "non-zero" on error. It is necessary to see if *errno*(3) has been explicitly set.

strtod(3), *strtol*(3), *strtold*(3), *strtoll*(3), *strtoul*(3), *strtoull*(3)
: These functions return 0 on error, but that is also a legitimate return value. It is necessary to see if *errno*(3) has been explicitly set.

ungetc(3)
: While only a single character of backup is mandated by the ANSI C standard, it turns out that [e]glibc permits more... but that means it can fail with ENOMEM. It can also fail with EBADF if *fp* is bogus. Most difficult of all, if you pass EOF an error return occurs, but errno is not set.

The libexplain library detects all of these errors correctly, even in cases where the error values are poorly documented, if at all.

ENOSPC, No space left on device

When this error refers to a file on a file system, the libexplain library prints the mount point of the file system with the problem. This can make the source of the error much clearer.

```
write(fildes = 1 "example", data = 0xbfff2340, data_size = 5)
failed, No space left on device (28, ENOSPC) because the file sys-
tem containing fildes ("/home") has no more space for data
```

As more special device support is added, error messages are expected to include the device name and actual size of the device.

EROFS, Read-only file system

When this error refers to a file on a file system, the libexplain library prints the mount point of the file system with the problem. This can make the source of the error much clearer.

As more special device support is added, error messages are expected to include the device name and type.

```
open(pathname = "/dev/fd0", O_RDWR, 0666) failed, Read-only file
system (30, EROFS) because the floppy disk has the write protect
tab set
```

...because a CD-ROM is not writable
...because the memory card has the write protect tab set
...because the ½ inch magnetic tape does not have a write ring

rename

The *rename*(2) system call is used to change the location or name of a file, moving it between directories if required. If the destination pathname already exists it will be atomically replaced, so that there is no point at which another process attempting to access it will find it missing.

There are limitations, however: you can only rename a directory on top of another directory if the destination directory is not empty.

```
rename(oldpath = "foo", newpath = "bar") failed, Directory not
empty (39, ENOTEMPTY) because newpath is not an empty directory;
that is, it contains entries other than "." and ".."
```

You can't rename a directory on top of a non-directory, either.

```
rename(oldpath = "foo", newpath = "bar") failed, Not a directory
(20, ENOTDIR) because oldpath is a directory, but newpath is a
regular file, not a directory
```

Nor is the reverse allowed

```
rename(oldpath = "foo", newpath = "bar") failed, Is a directory
(21, EISDIR) because newpath is a directory, but oldpath is a reg-
ular file, not a directory
```

This, of course, makes the libexplain library's job more complicated, because the *unlink*(2) or *rmdir*(2) system call is called implicitly by *rename*(2), and so all of the *unlink*(2) or *rmdir*(2) errors must be detected and handled, as well.

dup2

The *dup2*(2) system call is used to create a second file descriptor that references the same object as the first file descriptor. Typically this is used to implement shell input and output redirection.

The fun thing is that, just as *rename*(2) can atomically rename a file on top of an existing file and remove the old file, *dup2*(2) can do this onto an already-open file descriptor.

Once again, this makes the libexplain library's job more complicated, because the *close*(2) system call is called implicitly by *dup2*(2), and so all of *close*(2)'s errors must be detected and handled, as well.

ADVENTURES IN IOCTL SUPPORT

The *ioctl*(2) system call provides device driver authors with a way to communicate with user-space that doesn't fit within the existing kernel API. See *ioctl_list*(2).

Decoding Request Numbers

From a cursory look at the *ioctl*(2) interface, there would appear to be a large but finite number of possible *ioctl*(2) requests. Each different *ioctl*(2) request is effectively another system call, but without any type-safety at all − the compiler can't help a programmer get these right. This was probably the motivation behind *tcflush*(3) and friends.

The initial impression is that you could decode *ioctl*(2) requests using a huge switch statement. This turns out to be infeasible because one very rapidly discovers that it is impossible to include all of the necessary system headers defining the various *ioctl*(2) requests, because they have a hard time playing nicely with each other.

A deeper look reveals that there is a range of "private" request numbers, and device driver authors are encouraged to use them. This means that there is a far larger possible set of requests, with ambiguous request numbers, than are immediately apparent. Also, there are some historical ambiguities as well.

We already knew that the switch was impractical, but now we know that to select the appropriate request name and explanation we must consider not only the request number but also the file descriptor.

The implementation of *ioctl*(2) support within the libexplain library is to have a table of pointers to *ioctl*(2) request descriptors. Each of these descriptors includes an optional pointer to a disambiguation function.

Each request is actually implemented in a separate source file, so that the necessary include files are relieved of the obligation to play nicely with others.

Representation

The philosophy behind the libexplain library is to provide as much information as possible, including an accurate representation of the system call. In the case of *ioctl*(2) this means printing the correct request number (by name) and also a correct (or at least useful) representation of the third argument.

The *ioctl*(2) prototype looks like this:

```
int ioctl(int fildes, int request, ...);
```

which should have your type-safety alarms going off. Internal to [e]glibc, this is turned into a variety of forms:

```
int __ioctl(int fildes, int request, long arg);
int __ioctl(int fildes, int request, void *arg);
```

and the Linux kernel syscall interface expects

```
asmlinkage long sys_ioctl(unsigned int fildes, unsigned int
request, unsigned long arg);
```

The extreme variability of the third argument is a challenge, when the libexplain library tries to print a representation of that third argument. However, once the request number has been disambiguated, each entry in the the libexplain library's ioctl table has a custom `print_data` function (OO done manually).

Explanations

There are fewer problems determining the explanation to be used. Once the request number has been disambiguated, each entry in the libexplain library's ioctl table has a custom `print_explanation` function (again, OO done manually).

Unlike section 2 and section 3 system calls, most *ioctl*(2) requests have no errors documented. This means, to give good error descriptions, it is necessary to read kernel sources to discover

- what *errno*(3) values may be returned, and
- the cause of each error.

Because of the OO nature of function call dispatching withing the kernel, you need to read *all* sources implementing that *ioctl*(2) request, not just the generic implementation. It is to be expected that different kernels will have different error numbers and subtly different error causes.

EINVAL *vs* **ENOTTY**

The situation is even worse for *ioctl*(2) requests than for system calls, with EINVAL and ENOTTY both being used to indicate that an *ioctl*(2) request is inappropriate in that context, and occasionally ENOSYS, ENOTSUP and EOPNOTSUPP (meant to be used for sockets) as well. There are comments in the Linux kernel sources that seem to indicate a progressive cleanup is in progress. For extra chaos, BSD adds ENOIOCTL to the confusion.

As a result, attention must be paid to these error cases to get them right, particularly as the EINVAL could also be referring to problems with one or more system call arguments.

intptr_t

The C99 standard defines an integer type that is guaranteed to be able to hold any pointer without representation loss.

The above function syscall prototype would be better written

```
long sys_ioctl(unsigned int fildes, unsigned int request, intptr_t
arg);
```

The problem is the cognitive dissonance induced by device-specific or file-system-specific *ioctl*(2) implementations, such as:

```
long vfs_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg);
```

The majority of *ioctl*(2) requests actually have an `int *arg` third argument. But having it declared `long` leads to code treating this as `long *arg`. This is harmless on 32-bits (`sizeof(long) == sizeof(int)`) but nasty on 64-bits (`sizeof(long) != sizeof(int)`). Depending on the endian-ness, you do or don't get the value you expect, but you *always* get a memory scribble or stack scribble as well.

Writing all of these as

```
int ioctl(int fildes, int request, ...);
int __ioctl(int fildes, int request, intptr_t arg);
long sys_ioctl(unsigned int fildes, unsigned int request, intptr_t
arg);
long vfs_ioctl(struct file *filp, unsigned int cmd, intptr_t arg);
```

emphasizes that the integer is only an integer to represent a quantity that is almost always an unrelated pointer type.

CONCLUSION

Use libexplain, your users will like it.

COPYRIGHT

AUTHOR

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