NAME

File::FcntlLock - File locking with fcntl(2)

This text also documents the following sub-packages:

File::FcntlLock::XS
File::FcntlLock::Pure
File::FcntlLock::Inline

SYNOPSIS

```
use File::FcntlLock;

my $fs = new File::FcntlLock;
$fs->l_type( F_RDLCK );
$fs->l_whence( SEEK_CUR );
$fs->l_start( 100 );
$fs->l_len( 123 );

open my $fh, '<', 'file_name' or die "Can't open file: $!\n";
$fs->lock( $fh, F_SETLK )
or print "Locking failed: " . $fs->error . "\n";
$fs->l_type( F_UNLCK );
$fs->lock( $fh, F_SETLK )
or print "Unlocking failed: " . $fs->error . "\n";
```

DESCRIPTION

File locking in Perl is usually done using the `flock` function. Unfortunately, this only allows locks on whole files and is often implemented in terms of the *flock(2)* system function which has some shortcomings (especially concerning locks on remotely mounted file systems) and slightly different behaviour than *fcntl(2)*.

Using this module file locking via *fcntl(2)* can be done (obviously, this restricts the use of the module to systems that have a *fcntl(2)* system call). Before a file (or parts of a file) can be locked, an object simulating a flock structure, containing information in a binary format to be passed to *fcntl(2)* for locking requests, must be created and its properties set. Afterwards, by calling the *lock()* method a lock can be set and removed or it can be determined if and which process currently holds the lock.

File::FcntlLock (or its alias File::FcntlLock::XS) uses a shared library, build during installation, to call the *fcntl(2)* system function directly. If this is unsuitable there are two alternatives, File::FcntlLock::Pure and File::FcntlLock::Inline. Both call the Perl `fcntl` function instead and use Perl code to assemble and disassemble the structure. For this at some time the (system-dependent) binary layout of the flock structure must have been determined via a program written in C. The difference between File::FcntlLock::Pure and File::FcntlLock::Inline is that for the former this happened when the package is installed while for the latter it is done each time the package is loaded (e.g., with `use`). Thus, for File::FcntlLock::Inline to work a C compiler must be available. There are some minor differences in the functionality and the behaviour on passing the method for locking invalid arguments to be described below.

Creating objects

new()

To create a new object, representing a flock structure, call *new()*:

```
$fs = new File::FcntlLock;
```

The object has a number of properties, reflecting the members of the flock structure to be passed to *fcntl(2)* (see below). Per default on object creation the l_type property is set to `F_RDLCK`, l_whence to `SEEK_SET`, and both l_start and l_len to 0, i.e., the settings for a read lock on the whole file.

These defaults can be overruled by passing the *new()* method a set of key-value pairs to initialize the objects properties, e.g. use

```
$fs = new File::FcntlLock( l_type => F_WRLCK,
l_whence => SEEK_SET,
l_start => 0,
l_len => 100 );
```

if you intend to obtain a write lock for the first 100 bytes of a file.

Object properties

Once the object simulating the flock structure has been created the following methods allow to query and, in most cases, to also modify its properties.

`l_type()`

If called without an argument the method returns the current setting of the lock type, otherwise the lock type is set to the argument's value which must be either `F_RDLCK`, `F_WRLCK` or `F_UNLCK` (for read lock, write lock or unlock).

`l_whence()`

This method sets, when called with an argument, the l_whence property of the flock object, determining if the l_start value is relative to the start of the file, to the current position in the file or to the end of the file. These values are `SEEK_SET`, `SEEK_CUR` and `SEEK_END` (also see the man page for *lseek(2)*). If called with no argument the current value of the property is returned.

`l_start()`

Queries or sets the start position (offset) of the lock in the file according to the mode selected by the l_whence member. See also the man page for *lseek(2)*.

`l_len()`

Queries or sets the length of the region (in bytes) in the file to be locked. A value of 0 is interpreted to mean a lock, starting at `l_start`, to the end of the file. E.g., a lock obtained with l_whence set to `SEEK_SET` and both l_start and l_len set to 0 locks the complete file.

According to SUSv3 support for negative values for l_len are permitted, resulting in a lock ranging from `l_start+l_len` up to and including `l_start-1`. But not all systems support negative values for l_len and will return an error when you try to obtain such a lock, so please read the *fcntl(2)* man page of the system carefully for details.

`l_pid()`

If a call of the *lock()* method with `F_GETLK` indicates that another process is holding the lock (in which case the l_type property will be either `F_WRLCK` or `F_RDLCK`) a call of the *l_pid()* method returns the PID of the process holding the lock. This method does not accept any arguments.

Locking

After having set up the object representing a flock structure one can then try to obtain a lock, release it or determine the current holder of the lock by invoking the *lock()* method:

`lock()`

This method expects two arguments. The first one is a file handle (or typeglob). File::FcntlLock, and thus File::FcntlLock::XS (**but neither** File::FcntlLock::Pure **nor** File::FcntlLock::Inline), also accepts a "raw" integer file descriptor. The second argument is a flag indicating the action to be taken. So call it as in

```
$fs->lock( $fh, F_SETLK );
```

There are three values that can be used as the second argument:

`F_SETLK`
: With `F_SETLK` the *lock()* method tries to obtain a lock (when l_type is set to either `F_WRLCK` or `F_RDLCK`) or releases it (if l_type is set to `F_UNLCK`). If an attempt is made to obtain a lock but a lock is already being held by some other process the method returns `undef` and `errno` is set to `EACCESS` or `EAGAIN` (please see the the man page for *fcntl(2)* for more details).

`F_SETLKW`
: is similar to `F_SETLK`, but instead of returning an error if the lock can't be obtained immediately it puts the calling process to sleep, i.e., it blocks, until the lock is obtained at some later time. If a signal is received while waiting for the lock the method returns `undef` and `errno` is set to `EINTR`.

`F_GETLK`
: With `F_GETLK` the *lock()* method determines if and which process currently is holding the lock. If there's no other lock the l_type property will be set to `F_UNLCK`. Otherwise the flock structure object is set to the values that would prevent us from obtaining a lock. There may be several processes that keep us from getting a lock, including some that themselves are blocked waiting to obtain a lock. `F_GETLK` will only make details of one of these processes visible, and one has no control over which process this is.

On success the *lock()* method returns the string "0 but true", i.e., a value that is true in boolean but 0 in numeric context. If the method fails (as indicated by an `undef` return value) you can either immediately evaluate the error number (using $!, `$ERRNO` or `$OS_ERROR`) or check for it via the methods discussed below at some later time.

Error handling

There are minor differences between File::FcntlLock on the one hand and File::FcntlLock::Pure and File::FcntlLock::Inline on the other, due to the first calling the system function *fcntl(2)* directly while the latter two invoke the Perl `fcntl` function. Perl's `fcntl` function already returns a Perl error on some types of invalid arguments. In contrast File::FcntlLock passes them on to the *fcntl(2)* system call and then returns the systems response to the caller.

There are three methods for obtaining information about the reason the a call of the *lock()* method failed:

`lock_errno()`
: Returns the `errno` error number from the latest call of *lock()*. If the last call did not result in an error `undef` is returned.

`error()`
: Returns a short description of the error that happened during the latest call of *lock()*. Please take the messages with a grain of salt, they represent what SUSv3 (IEEE 1003.1-2001) and the Linux, TRUE64, OpenBSD3 and Solaris8 man pages tell what the error numbers mean. There could be differences (and additional error numbers) on other systems. If there was no error the method returns `undef`.

`system_error()`
: While the *error()* method tries to return a string with some direct relevance to the locking operation (i.e., "File or segment already locked by other process(es)" instead of "Permission denied") this method returns the "normal" system error message associated with `errno`. The method returns `undef` if there was no error.

EXPORT

The package exports the following constants:

F_GETLK F_SETLK F_SETLKW
F_RDLCK F_WRLCK F_UNLCK

SEEK_SET SEEK_CUR SEEK_END

INCOMPATIBILITIES

Obviously, this module requires that there's a *fcntl(2)* system call. Note also that under certain circumstances the File::FcntlLock::Pure and File::FcntlLock::Inline modules may not have been installed. This happens on 32-bit systems that use 64-bit integers in their flock structure but where the installed Perl version doesn't support the 'q' format for its `pack` and `unpack` functions.

CREDITS

Thanks to Mark Jason Dominus and Benjamin Goldberg for helpful discussions, code examples and encouragement. Glenn Herteg pointed out several problems and also helped improve the documentation. Julian Moreno Patino helped correcting the documentation and pointed out problems arising on GNU Hurd which seems to have only very rudimentary support for locking with *fcntl(2)*. Niko Tyni and Guillem Jover encouraged and helped with implementing alternatives to an XS-only approach which hopefully will make the module more useful under certain circumstances.

AUTHOR

Jens Thoms Toerring <jt@toerring.de>

SEE ALSO

perl(1), *fcntl(2)*, *lseek(2)*.

LICENSE

This library is free software. You can redistribute it and/or modify it under the same terms as Perl itself.