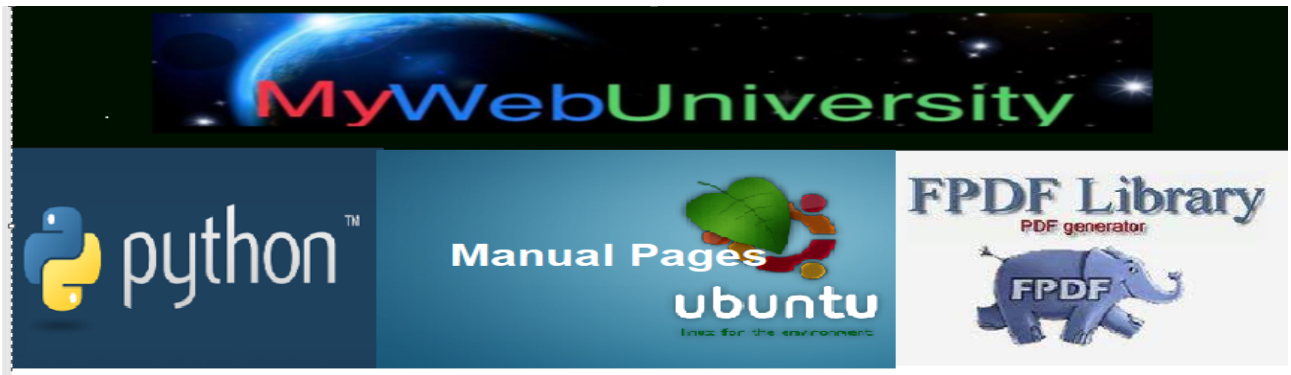

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'fsopen.2'

$ man fsopen.2

fsopen(2) System Calls Manual fsopen(2)

NAME

fsopen - create a new filesystem context

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

#include <sys/mount.h>

int fsopen(const char *fsname, unsigned int flags);

DESCRIPTION

The fsopen() system call is part of the suite of file-descriptor-based mount facilities in Linux.

fsopen() creates a blank filesystem configuration context within the kernel for the filesys? tem named by fsname and places it into creation mode. A new file descriptor associated with the filesystem configuration context is then returned. The calling process must have the CAP_SYS_ADMIN capability in order to create a new filesystem configuration context.

A filesystem configuration context is an in-kernel representation of a pending transaction, containing a set of configuration parameters that are to be applied when creating a new in? stance of a filesystem (or modifying the configuration of an existing filesystem instance, such as when using fspick(2)).

After obtaining a filesystem configuration context with fsopen(), the general workflow for operating on the context looks like the following:

(1) Pass the filesystem context file descriptor to fsconfig(2) to specify any desired filesystem parameters. This may be done as many times as necessary.

(2) Pass the same filesystem context file descriptor to fsconfig(2) with FSCONFIG_CMD_CREATE to create an instance of the configured filesystem.

(3) Pass the same filesystem context file descriptor to fsmount(2) to create a new detached mount object for the root of the filesystem instance, which is then attached to a new file descriptor. (This also places the filesystem context file descriptor into recon? figuration mode, similar to the mode produced by fspick(2).) Once a mount object has been created with fsmount(2), the filesystem context file descriptor can be safely closed.

(4) Now that a mount object has been created, you may

? use the detached mount object file descriptor as a dirfd argument to "*at()" system calls; and/or

? attach the mount object to a mount point by passing the mount object file descriptor to move_mount(2). This will also prevent the mount object from being unmounted and destroyed when the mount object file descriptor is closed.

The mount object file descriptor will remain associated with the mount object even af? ter doing the above operations, so you may repeatedly use the mount object file de? scriptor with move_mount(2) and/or "*at()" system calls as many times as necessary.

A filesystem context will move between different modes throughout its lifecycle (such as the creation phase when created with fsopen(), the reconfiguration phase when an existing filesystem instance is selected with fspick(2), and the intermediate "awaiting-mount" phase between FSCONFIG_CMD_CREATE and fsmount(2)), which has an impact on what operations are per? mitted on the filesystem context.

The file descriptor returned by fsopen() also acts as a channel for filesystem drivers to provide more comprehensive diagnostic information than is normally provided through the standard errno(3) interface for system calls. If an error occurs at any time during the workflow mentioned above, calling read(2) on the filesystem context file descriptor will re? trieve any ancillary information about the encountered errors. (See the "Message retrieval interface" section for more details on the message format.)

flags can be used to control aspects of the creation of the filesystem configuration context file descriptor. A value for flags is constructed by bitwise ORing zero or more of the fol? lowing constants:

FSOPEN_CLOEXEC

Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor. See the

description of the O_CLOEXEC flag in open(2) for reasons why this may be use?
ful.

A list of filesystems supported by the running kernel (and thus a list of valid values for fsname) can be obtained from /proc/filesystems. (See also proc_filesystems(5).)

Message retrieval interface

When doing operations on a filesystem configuration context, the filesystem driver may choose to provide ancillary information to userspace in the form of message strings.

The filesystem context file descriptors returned by fsopen() and fspick(2) may be queried for message strings at any time by calling read(2) on the file descriptor. Each call to read(2) will return a single message, prefixed to indicate its class:

e message

An error message was logged. This is usually associated with an error being returned from the corresponding system call which triggered this message.

w message

A warning message was logged.

i message

An informational message was logged.

Messages are removed from the queue as they are read. Note that the message queue has lim?
ited depth, so it is possible for messages to get lost. If there are no messages in the message queue, read(2) will return -1 and errno will be set to ENODATA. If the buf argument to read(2) is not large enough to contain the entire message, read(2) will return -1 and er?
rno will be set to EMSGSIZE. (See BUGS.)

If there are multiple filesystem contexts referencing the same filesystem instance (such as if you call fspick(2) multiple times for the same mount), each one gets its own independent message queue. This does not apply to multiple file descriptors that are tied to the same underlying open file description (such as those created with dup(2)).

Message strings will usually be prefixed by the name of the filesystem or kernel subsystem that logged the message, though this may not always be the case. See the Linux kernel source code for details.

RETURN VALUE

On success, a new file descriptor is returned. On error, -1 is returned, and errno is set to indicate the error.

ERRORS

EFAULT fsname is NULL or a pointer to a location outside the calling process's accessible address space.

EINVAL flags had an invalid flag set.

EMFILE The calling process has too many open files to create more.

ENFILE The system has too many open files to create more.

ENODEV The filesystem named by fsname is not supported by the kernel.

ENOMEM The kernel could not allocate sufficient memory to complete the operation.

EPERM The calling process does not have the required CAP_SYS_ADMIN capability.

STANDARDS

Linux.

HISTORY

Linux 5.2. glibc 2.36.

BUGS

Message retrieval interface and EMSGSIZE

As described in the "Message retrieval interface" subsection above, calling read(2) with too small a buffer to contain the next pending message in the message queue for the filesystem configuration context will cause read(2) to return -1 and set errno(3) to EMSGSIZE. However, this failed operation still consumes the message from the message queue. This effectively discards the message silently, as no data is copied into the read(2) buffer. Programs should take care to ensure that their buffers are sufficiently large to contain any reasonable message string, in order to avoid silently losing valuable diagnostic information.

EXAMPLES

To illustrate the workflow for creating a new mount, the following is an example of how to mount an ext4(5) filesystem stored on /dev/sdb1 onto /mnt.

```
int fsfd, mntfd;
fsfd = fsopen("ext4", FSOPEN_CLOEXEC);
fsconfig(fsfd, FSCONFIG_SET_FLAG, "ro", NULL, 0);
fsconfig(fsfd, FSCONFIG_SET_PATH, "source", "/dev/sdb1", AT_FDCWD);
fsconfig(fsfd, FSCONFIG_SET_FLAG, "noatime", NULL, 0);
fsconfig(fsfd, FSCONFIG_SET_FLAG, "acl", NULL, 0);
fsconfig(fsfd, FSCONFIG_SET_FLAG, "user_xattr", NULL, 0);
fsconfig(fsfd, FSCONFIG_SET_FLAG, "iversion", NULL, 0)
```

```
fsconfig(fsfd, FSCONFIG_CMD_CREATE, NULL, NULL, 0);
mntfd = fsmount(fsfd, FSMOUNT_CLOEXEC, MOUNT_ATTR_RELATIME);
move_mount(mntfd, "", AT_FDCWD, "/mnt", MOVE_MOUNT_F_EMPTY_PATH);
```

First, an ext4 configuration context is created and attached to the file descriptor fsfd. Then, a series of parameters (such as the source of the filesystem) are provided using fsconfig(2), followed by the filesystem instance being created with FSCONFIG_CMD_CREATE. fsmount(2) is then used to create a new mount object attached to the file descriptor mntfd, which is then attached to the intended mount point using move_mount(2).

The above procedure is functionally equivalent to the following mount operation using mount(2):

```
mount("/dev/sdb1", "/mnt", "ext4", MS_RELATIME,
      "ro,noatime,acl,user_xattr,iversion");
```

And here's an example of creating a mount object of an NFS server share and setting a Smack security module label. However, instead of attaching it to a mount point, the program uses the mount object directly to open a file from the NFS share.

```
int fsfd, mntfd, fd;
fsfd = fsopen("nfs", 0);
fsconfig(fsfd, FSCONFIG_SET_STRING, "source", "example.com/pub", 0);
fsconfig(fsfd, FSCONFIG_SET_STRING, "nfsvers", "3", 0);
fsconfig(fsfd, FSCONFIG_SET_STRING, "rsize", "65536", 0);
fsconfig(fsfd, FSCONFIG_SET_STRING, "wsize", "65536", 0);
fsconfig(fsfd, FSCONFIG_SET_STRING, "smackfsdef", "foolabel", 0);
fsconfig(fsfd, FSCONFIG_SET_FLAG, "rdma", NULL, 0);
fsconfig(fsfd, FSCONFIG_CMD_CREATE, NULL, NULL, 0);
mntfd = fsmount(fsfd, 0, MOUNT_ATTR_NODEV);
fd = openat(mntfd, "src/linux-5.2.tar.xz", O_RDONLY);
```

Unlike the previous example, this operation has no trivial equivalent with mount(2), as it was not previously possible to create a mount object that is not attached to any mount point.

SEE ALSO

fsconfig(2), fsmount(2), fspick(2), mount(2), mount_setattr(2), move_mount(2), open_tree(2), mount_namespaces(7)