NAME

unix, AF_UNIX, AF_LOCAL − Sockets for local interprocess communication

SYNOPSIS

#include <sys/socket.h>
#include <sys/un.h>

unix_socket **= socket(AF_UNIX, type, 0);**
error **= socketpair(AF_UNIX, type, 0, int ****sv***);**

DESCRIPTION

The **AF_UNIX** (also known as **AF_LOCAL**) socket family is used to communicate between processes on the same machine efficiently. Traditionally, Unix sockets can be either unnamed, or bound to a file system pathname (marked as being of type socket). Linux also supports an abstract namespace which is independent of the file system.

Valid types are: **SOCK_STREAM**, for a stream-oriented socket and **SOCK_DGRAM**, for a datagram-oriented socket that preserves message boundaries (as on most Unix implementations, Unix domain datagram sockets are always reliable and don’t reorder datagrams); and (since Linux 2.6.4) **SOCK_SEQPACKET**, for a connection-oriented socket that preserves message boundaries and delivers messages in the order that they were sent.

Unix sockets support passing file descriptors or process credentials to other processes using ancillary data.

Address Format

A Unix domain socket address is represented in the following structure:

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
    sa_family_t sun_family;               /* AF_UNIX */
    char        sun_path[UNIX_PATH_MAX];  /* pathname */
};
```

sun_family always contains **AF_UNIX**.

Three types of address are distinguished in this structure:

* *pathname*: a Unix domain socket can be bound to a null-terminated file system pathname using **bind**(2). When the address of the socket is returned by **getsockname**(2), **getpeername**(2), and **accept**(2), its length is *sizeof(sa_family_t) + strlen(sun_path) + 1*, and *sun_path* contains the null-terminated pathname.

* *unnamed*: A stream socket that has not been bound to a pathname using **bind**(2) has no name. Likewise, the two sockets created by **socketpair**(2) are unnamed. When the address of an unnamed socket is returned by **getsockname**(2), **getpeername**(2), and **accept**(2), its length is *sizeof(sa_family_t)*, and *sun_path* should not be inspected.

* *abstract*: an abstract socket address is distinguished by the fact that *sun_path[0]* is a null byte (’\0’). All of the remaining bytes in *sun_path* define the "name" of the socket. (Null bytes in the name have no special significance.) The name has no connection with file system pathnames. The socket’s address in this namespace is given by the rest of the bytes in *sun_path*. When the address of an abstract socket is returned by **getsockname**(2), **getpeername**(2), and **accept**(2), its length is *sizeof(struct sockaddr_un)*, and *sun_path* contains the abstract name. The abstract socket namespace is a non-portable Linux extension.

Socket Options

For historical reasons these socket options are specified with a **SOL_SOCKET** type even though they are **AF_UNIX** specific. They can be set with **setsockopt**(2) and read with **getsockopt**(2) by specifying

SOL_SOCKET as the socket family.

SO_PASSCRED
Enables the receiving of the credentials of the sending process ancillary message. When this option is set and the socket is not yet connected a unique name in the abstract namespace will be generated automatically. Expects an integer boolean flag.

Sockets API

The following paragraphs describe domain-specific details and unsupported features of the sockets API for Unix domain sockets on Linux.

Unix domain sockets do not support the transmission of out-of-band data (the **MSG_OOB** flag for **send**(2) and **recv**(2)).

The **send**(2) **MSG_MORE** flag is not supported by Unix domain sockets.

The use of **MSG_TRUNC** in the *flags* argument of **recv**(2) is not supported by Unix domain sockets.

The **SO_SNDBUF** socket option does have an effect for Unix domain sockets, but the **SO_RCVBUF** option does not. For datagram sockets, the **SO_SNDBUF** value imposes an upper limit on the size of outgoing datagrams. This limit is calculated as the doubled (see **socket**(7)) option value less 32 bytes used for overhead.

Ancillary Messages

Ancillary data is sent and received using **sendmsg**(2) and **recvmsg**(2). For historical reasons the ancillary message types listed below are specified with a **SOL_SOCKET** type even though they are **AF_UNIX** specific. To send them set the *cmsg_level* field of the struct *cmsghdr* to **SOL_SOCKET** and the *cmsg_type* field to the type. For more information see **cmsg**(3).

SCM_RIGHTS
Send or receive a set of open file descriptors from another process. The data portion contains an integer array of the file descriptors. The passed file descriptors behave as though they have been created with **dup**(2).

SCM_CREDENTIALS
Send or receive Unix credentials. This can be used for authentication. The credentials are passed as a *struct ucred* ancillary message. Thus structure is defined in <*sys/socket.h*> as follows:

```
struct ucred {
    pid_t pid;    /* process ID of the sending process */
    uid_t uid;    /* user ID of the sending process */
    gid_t gid;    /* group ID of the sending process */
};
```

Since glibc 2.8, the **_GNU_SOURCE** feature test macro must be defined in order to obtain the definition of this structure.

The credentials which the sender specifies are checked by the kernel. A process with effective user ID 0 is allowed to specify values that do not match its own. The sender must specify its own process ID (unless it has the capability **CAP_SYS_ADMIN**), its user ID, effective user ID, or saved set-user-ID (unless it has **CAP_SETUID**), and its group ID, effective group ID, or saved set-group-ID (unless it has **CAP_SETGID**). To receive a *struct ucred* message the **SO_PASSCRED** option must be enabled on the socket.

ERRORS

EADDRINUSE
Selected local address is already taken or file system socket object already exists.

ECONNREFUSED
: **connect**(2) called with a socket object that isn't listening. This can happen when the remote socket does not exist or the filename is not a socket.

ECONNRESET
: Remote socket was unexpectedly closed.

EFAULT
: User memory address was not valid.

EINVAL
: Invalid argument passed. A common cause is the missing setting of AF_UNIX in the *sun_type* field of passed addresses or the socket being in an invalid state for the applied operation.

EISCONN
: **connect**(2) called on an already connected socket or a target address was specified on a connected socket.

ENOMEM
: Out of memory.

ENOTCONN
: Socket operation needs a target address, but the socket is not connected.

EOPNOTSUPP
: Stream operation called on non-stream oriented socket or tried to use the out-of-band data option.

EPERM
: The sender passed invalid credentials in the *struct ucred*.

EPIPE Remote socket was closed on a stream socket. If enabled, a **SIGPIPE** is sent as well. This can be avoided by passing the **MSG_NOSIGNAL** flag to **sendmsg**(2) or **recvmsg**(2).

EPROTONOSUPPORT
: Passed protocol is not AF_UNIX.

EPROTOTYPE
: Remote socket does not match the local socket type (**SOCK_DGRAM** vs. **SOCK_STREAM**)

ESOCKTNOSUPPORT
: Unknown socket type.

Other errors can be generated by the generic socket layer or by the file system while generating a file system socket object. See the appropriate manual pages for more information.

VERSIONS

SCM_CREDENTIALS and the abstract namespace were introduced with Linux 2.2 and should not be used in portable programs. (Some BSD-derived systems also support credential passing, but the implementation details differ.)

NOTES

In the Linux implementation, sockets which are visible in the file system honor the permissions of the directory they are in. Their owner, group and their permissions can be changed. Creation of a new socket will fail if the process does not have write and search (execute) permission on the directory the socket is created in. Connecting to the socket object requires read/write permission. This behavior differs from many BSD-derived systems which ignore permissions for Unix sockets. Portable programs should not rely on this feature for security.

Binding to a socket with a filename creates a socket in the file system that must be deleted by the caller when it is no longer needed (using **unlink**(2)). The usual Unix close-behind semantics apply; the socket can be unlinked at any time and will be finally removed from the file system when the last reference to it is closed.

To pass file descriptors or credentials over a **SOCK_STREAM**, you need to send or receive at least one byte of non-ancillary data in the same **sendmsg**(2) or **recvmsg**(2) call.

Unix domain stream sockets do not support the notion of out-of-band data.

EXAMPLE

See **bind**(2).

SEE ALSO

recvmsg(2), **sendmsg**(2), **socket**(2), **socketpair**(2), **cmsg**(3), **capabilities**(7), **credentials**(7), **socket**(7)

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.