NAME

unix – 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 domain sockets can be either unnamed, or bound to a filesystem pathname (marked as being of type socket). Linux also supports an abstract namespace which is independent of the filesystem.

Valid socket types in the UNIX domain are: **SOCK_STREAM**, for a stream-oriented socket; **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 sequenced-packet socket that is connection-oriented, preserves message boundaries, and delivers messages in the order that they were sent.

UNIX domain 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:

```
struct sockaddr_un {
sa_family_t sun_family;               /* AF_UNIX */
char        sun_path[108];            /* pathname */
};
```

The *sun_family* field always contains **AF_UNIX**. On Linux *sun_path* is 108 bytes in size; see also NOTES, below.

Various systems calls (for example, bind(2), connect(2), and sendto(2)) take a *sockaddr_un* argument as input. Some other system calls (for example, getsockname(2), getpeername(2), recvfrom(2), and accept(2)) return an argument of this type.

Three types of address are distinguished in the *sockaddr_un* structure:

* *pathname*: a UNIX domain socket can be bound to a null-terminated filesystem pathname using bind(2). When the address of a pathname socket is returned (by one of the system calls noted above), its length is

  offsetof(struct sockaddr_un, sun_path) + strlen(sun_path) + 1

  and *sun_path* contains the null-terminated pathname. (On Linux, the above **offsetof**() expression equates to the same value as *sizeof(sa_family_t)*, but some other implementations include other fields before *sun_path*, so the **offsetof**() expression more portably describes the size of the address structure.)

  For further details of pathname sockets, see below.

* *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, its length is *sizeof(sa_family_t)*, and *sun_path* should not be inspected.

* *abstract*: an abstract socket address is distinguished (from a pathname socket) by the fact that *sun_path[0]* is a null byte ('\0'). The socket's address in this namespace is given by the additional bytes in *sun_path* that are covered by the specified length of the address structure. (Null bytes in the name have no special significance.) The name has no connection with filesystem pathnames. When the address of an abstract socket is returned, the returned *addrlen* is greater than *sizeof(sa_family_t)* (i.e., greater than 2), and the name of the socket is contained in the first *(addrlen – sizeof(sa_family_t))* bytes

of *sun_path*.

Pathname sockets

When binding a socket to a pathname, a few rules should be observed for maximum portability and ease of coding:

* The pathname in *sun_path* should be null-terminated.
* The length of the pathname, including the terminating null byte, should not exceed the size of *sun_path*.
* The *addrlen* argument that describes the enclosing *sockaddr_un* structure should have a value of at least:

  offsetof(struct sockaddr_un, sun_path)+strlen(addr.sun_path)+1

  or, more simply, *addrlen* can be specified as *sizeof(struct sockaddr_un)*.

There is some variation in how implementations handle UNIX domain socket addresses that do not follow the above rules. For example, some (but not all) implementations append a null terminator if none is present in the supplied *sun_path*.

When coding portable applications, keep in mind that some implementations have *sun_path* as short as 92 bytes.

Various system calls (accept(2), recvfrom(2), getsockname(2), getpeername(2)) return socket address structures. When applied to UNIX domain sockets, the value-result *addrlen* argument supplied to the call should be initialized as above. Upon return, the argument is set to indicate the *actual* size of the address structure. The caller should check the value returned in this argument: if the output value exceeds the input value, then there is no guarantee that a null terminator is present in *sun_path*. (See BUGS.)

Pathname socket ownership and permissions

In the Linux implementation, pathname sockets honor the permissions of the directory they are in. Creation of a new socket fails if the process does not have write and search (execute) permission on the directory in which the socket is created.

On Linux, connecting to a stream socket object requires write permission on that socket; sending a datagram to a datagram socket likewise requires write permission on that socket. POSIX does not make any statement about the effect of the permissions on a socket file, and on some systems (e.g., older BSDs), the socket permissions are ignored. Portable programs should not rely on this feature for security.

When creating a new socket, the owner and group of the socket file are set according to the usual rules. The socket file has all permissions enabled, other than those that are turned off by the process umask(2).

The owner, group, and permissions of a pathname socket can be changed (using chown(2) and chmod(2)).

Abstract sockets

Socket permissions have no meaning for abstract sockets: the process umask(2) has no effect when binding an abstract socket, and changing the ownership and permissions of the object (via fchown(2) and fchmod(2)) has no effect on the accessibility of the socket.

Abstract sockets automatically disappear when all open references to the socket are closed.

The abstract socket namespace is a nonportable 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 in an 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.

Autobind feature

If a bind(2) call specifies *addrlen* as *sizeof(sa_family_t)*, or the **SO_PASSCRED** socket option was specified for a socket that was not explicitly bound to an address, then the socket is autobound to an abstract address. The address consists of a null byte followed by 5 bytes in the character set *[0−9a−f]*. Thus, there is a limit of 2^20 autobind addresses. (From Linux 2.1.15, when the autobind feature was added, 8 bytes were used, and the limit was thus 2^32 autobind addresses. The change to 5 bytes came in Linux 2.3.15.)

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.

Before Linux 3.4, the use of **MSG_TRUNC** in the *flags* argument of recv(2) was 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 (before including *any* header files) 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 real user ID, effective user ID, or saved set-user-ID (unless it has **CAP_SETUID**), and its real 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.

Ioctls

The following ioctl(2) calls return information in *value*. The correct syntax is:

int *value*;
error **= ioctl(***unix_socket***,** *ioctl_type***, &***value***);**

ioctl_type can be:

SIOCINQ

For **SOCK_STREAM** socket the function returns the amount of queued unread data in the receive buffer. The socket must not be in LISTEN state, otherwise an error (**EINVAL**) is returned. **SIOCINQ** is defined in *<linux/sockios.h>*. Alternatively, you can use the synonymous **FIONREAD**, defined in *<sys/ioctl.h>*. For **SOCK_DGRAM** socket, the returned value is the same as for Internet domain datagram socket; see udp(7).

ERRORS

EADDRINUSE

The specified local address is already in use or the filesystem socket object already exists.

EBADF

This error can occur for sendmsg(2) when sending a file descriptor as ancillary data over a UNIX domain socket (see the description of **SCM_RIGHTS**, above), and indicates that the file descriptor number that is being sent is not valid (e.g., it is not an open file descriptor).

ECONNREFUSED

The remote address specified by connect(2) was not a listening socket. This error can also occur if the target pathname 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 that the value **AF_UNIX** was not specified in the *sun_type* field of passed addresses, or the socket was 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.

ENOENT

The pathname in the remote address specified to connect(2) did not exist.

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 send(2) or sendmsg(2).

EPROTONOSUPPORT

Passed protocol is not **AF_UNIX**.

EPROTOTYPE

Remote socket does not match the local socket type (**SOCK_DGRAM** versus **SOCK_STREAM**).

ESOCKTNOSUPPORT

Unknown socket type.

ETOOMANYREFS

This error can occur for sendmsg(2) when sending a file descriptor as ancillary data over a UNIX domain socket (see the description of **SCM_RIGHTS**, above). It occurs if the number of "in-flight" file descriptors exceeds the **RLIMIT_NOFILE** resource limit and the caller does not have the **CAP_SYS_RESOURCE** capability. An in-flight file descriptor is one that has been sent using sendmsg(2) but has not yet been accepted in the recipient process using recvmsg(2).

This error is diagnosed since mainline Linux 4.5 (and in some earlier kernel versions where the fix has been backported). In earlier kernel versions, it was possible to place an unlimited number of file descriptors in flight, by sending each file descriptor with sendmsg(2) and then closing the file descriptor so that it was not accounted against the **RLIMIT_NOFILE** resource limit.

Other errors can be generated by the generic socket layer or by the filesystem while generating a filesystem 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

Binding to a socket with a filename creates a socket in the filesystem 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 filesystem when the last reference to it is closed.

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

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

BUGS

When binding a socket to an address, Linux is one of the implementations that appends a null terminator if none is supplied in *sun_path*. In most cases this is unproblematic: when the socket address is retrieved, it will be one byte longer than that supplied when the socket was bound. However, there is one case where confusing behavior can result: if 108 non-null bytes are supplied when a socket is bound, then the addition of the null terminator takes the length of the pathname beyond *sizeof(sun_path)*. Consequently, when retrieving the socket address (for example, via accept(2)), if the input *addrlen* argument for the retrieving call is specified as *sizeof(struct sockaddr_un)*, then the returned address structure *won't* have a null terminator in *sun_path*.

In addition, some implementations don't require a null terminator when binding a socket (the *addrlen* argument is used to determine the length of *sun_path*) and when the socket address is retrieved on these implementations, there is no null terminator in *sun_path*.

Applications that retrieve socket addresses can (portably) code to handle the possibility that there is no null terminator in *sun_path* by respecting the fact that the number of valid bytes in the pathname is:

strnlen(addr.sun_path, addrlen − offsetof(sockaddr_un, sun_path))

Alternatively, an application can retrieve the socket address by allocating a buffer of size *sizeof(struct sockaddr_un)+1* that is zeroed out before the retrieval. The retrieving call can specify *addrlen* as *sizeof(struct sockaddr_un)*, and the extra zero byte ensures that there will be a null terminator for the string returned in *sun_path*:

```
void *addrp;

addrlen = sizeof(struct sockaddr_un);
addrp = malloc(addrlen + 1);
if (addrp == NULL)
/* Handle error */ ;
memset(addrp, 0, addrlen + 1);

if (getsockname(sfd, (struct sockaddr *) addrp, &addrlen)) == -1)
/* handle error */ ;
```

```
printf("sun_path = %s\n", ((struct sockaddr_un *) addrp)->sun_path);
```

This sort of messiness can be avoided if it is guaranteed that the applications that *create* pathname sockets follow the rules outlined above under *Pathname sockets*.

EXAMPLE

The following code demonstrates the use of sequenced-packet sockets for local interprocess communication. It consists of two programs. The server program waits for a connection from the client program. The client sends each of its command-line arguments in separate messages. The server treats the incoming messages as integers and adds them up. The client sends the command string "END". The server sends back a message containing the sum of the client's integers. The client prints the sum and exits. The server waits for the next client to connect. To stop the server, the client is called with the command-line argument "DOWN".

The following output was recorded while running the server in the background and repeatedly executing the client. Execution of the server program ends when it receives the "DOWN" command.

Example output

```
$ ./server &
[1] 25887
$ ./client 3 4
Result = 7
$ ./client 11 -5
Result = 6
$ ./client DOWN
Result = 0
[1]+  Done                    ./server
$
```

Program source

```
/*
* File connection.h
*/

#define SOCKET_NAME "/tmp/9Lq7BNBnBycd6nxy.socket"
#define BUFFER_SIZE 12

/*
* File server.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include "connection.h"

int
main(int argc, char *argv[])
{
struct sockaddr_un name;
int down_flag = 0;
int ret;
int connection_socket;
int data_socket;
int result;
```

```
char buffer[BUFFER_SIZE];

/*
* In case the program exited inadvertently on the last run,
* remove the socket.
*/

unlink(SOCKET_NAME);

/* Create local socket. */

connection_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
if (connection_socket == -1) {
perror("socket");
exit(EXIT_FAILURE);
}

/*
* For portability clear the whole structure, since some
* implementations have additional (nonstandard) fields in
* the structure.
*/

memset(&name, 0, sizeof(struct sockaddr_un));

/* Bind socket to socket name. */

name.sun_family = AF_UNIX;
strncpy(name.sun_path, SOCKET_NAME, sizeof(name.sun_path) - 1);

ret = bind(connection_socket, (const struct sockaddr *) &name,
sizeof(struct sockaddr_un));
if (ret == -1) {
perror("bind");
exit(EXIT_FAILURE);
}

/*
* Prepare for accepting connections. The backlog size is set
* to 20. So while one request is being processed other requests
* can be waiting.
*/

ret = listen(connection_socket, 20);
if (ret == -1) {
perror("listen");
exit(EXIT_FAILURE);
}

/* This is the main loop for handling connections. */

for (;;) {

/* Wait for incoming connection. */

data_socket = accept(connection_socket, NULL, NULL);
if (data_socket == -1) {
perror("accept");
exit(EXIT_FAILURE);
}

result = 0;
```

```
for(;;) {

/* Wait for next data packet. */

ret = read(data_socket, buffer, BUFFER_SIZE);
if (ret == -1) {
perror("read");
exit(EXIT_FAILURE);
}

/* Ensure buffer is 0-terminated. */

buffer[BUFFER_SIZE - 1] = 0;

/* Handle commands. */

if (!strncmp(buffer, "DOWN", BUFFER_SIZE)) {
down_flag = 1;
break;
}

if (!strncmp(buffer, "END", BUFFER_SIZE)) {
break;
}

/* Add received summand. */

result += atoi(buffer);
}

/* Send result. */

sprintf(buffer, "%d", result);
ret = write(data_socket, buffer, BUFFER_SIZE);

if (ret == -1) {
perror("write");
exit(EXIT_FAILURE);
}

/* Close socket. */

close(data_socket);

/* Quit on DOWN command. */

if (down_flag) {
break;
}
}

close(connection_socket);

/* Unlink the socket. */

unlink(SOCKET_NAME);

exit(EXIT_SUCCESS);
}

/*
* File client.c
*/

#include <errno.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include "connection.h"

int
main(int argc, char *argv[])
{
struct sockaddr_un addr;
int i;
int ret;
int data_socket;
char buffer[BUFFER_SIZE];

/* Create local socket. */

data_socket = socket(AF_UNIX, SOCK_SEQPACKET, 0);
if (data_socket == -1) {
perror("socket");
exit(EXIT_FAILURE);
}

/*
* For portability clear the whole structure, since some
* implementations have additional (nonstandard) fields in
* the structure.
*/

memset(&addr, 0, sizeof(struct sockaddr_un));

/* Connect socket to socket address */

addr.sun_family = AF_UNIX;
strncpy(addr.sun_path, SOCKET_NAME, sizeof(addr.sun_path) - 1);

ret = connect (data_socket, (const struct sockaddr *) &addr,
sizeof(struct sockaddr_un));
if (ret == -1) {
fprintf(stderr, "The server is down.\n");
exit(EXIT_FAILURE);
}

/* Send arguments. */

for (i = 1; i < argc; ++i) {
ret = write(data_socket, argv[i], strlen(argv[i]) + 1);
if (ret == -1) {
perror("write");
break;
}
}

/* Request result. */

strcpy (buffer, "END");
ret = write(data_socket, buffer, strlen(buffer) + 1);
if (ret == -1) {
```

```
perror("write");
exit(EXIT_FAILURE);
}

/* Receive result. */

ret = read(data_socket, buffer, BUFFER_SIZE);
if (ret == -1) {
perror("read");
exit(EXIT_FAILURE);
}

/* Ensure buffer is 0-terminated. */

buffer[BUFFER_SIZE - 1] = 0;

printf("Result = %s\n", buffer);

/* Close socket. */

close(data_socket);

exit(EXIT_SUCCESS);
}
```

For an example of the use of **SCM_RIGHTS** see cmsg(3).

SEE ALSO

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

COLOPHON

This page is part of release 4.16 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.