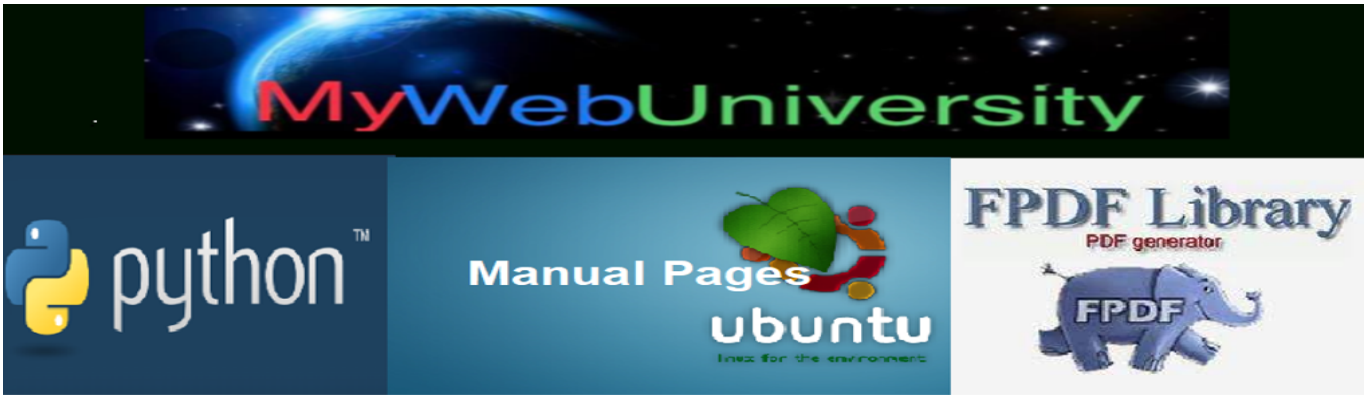

Rocky Enterprise Linux 9.2 Manual Pages on command 'sock_diag.7'

$ man sock_diag.7

SOCK_DIAG(7) Linux Programmer's Manual SOCK_DIAG(7)

NAME

sock_diag - obtaining information about sockets

SYNOPSIS

#include <sys/socket.h>

#include <linux/sock_diag.h>

#include <linux/unix_diag.h> /* for UNIX domain sockets */

#include <linux/inet_diag.h> /* for IPv4 and IPv6 sockets */

diag_socket = socket(AF_NETLINK, socket_type, NETLINK_SOCK_DIAG);

DESCRIPTION

The sock_diag netlink subsystem provides a mechanism for obtaining information about sock?

ets of various address families from the kernel. This subsystem can be used to obtain in?

formation about individual sockets or request a list of sockets.

In the request, the caller can specify additional information it would like to obtain

about the socket, for example, memory information or information specific to the address

family.

When requesting a list of sockets, the caller can specify filters that would be applied by

the kernel to select a subset of sockets to report. For now, there is only the ability to

filter sockets by state (connected, listening, and so on.)

Note that sock_diag reports only those sockets that have a name; that is, either sockets

bound explicitly with bind(2) or sockets that were automatically bound to an address

(e.g., by connect(2)). This is the same set of sockets that is available via

/proc/net/unix, /proc/net/tcp, /proc/net/udp, and so on.

Request

The request starts with a struct nlmsghdr header described in netlink(7) with nlmsg_type field set to SOCK_DIAG_BY_FAMILY. It is followed by a header specific to the address fam? ily that starts with a common part shared by all address families:

```
struct sock_diag_req {
    __u8 sdiag_family;
    __u8 sdiag_protocol;
};
```

The fields of this structure are as follows:

sdiag_family

An address family. It should be set to the appropriate AF_* constant.

sdiag_protocol

Depends on sdiag_family. It should be set to the appropriate IPPROTO_* constant for AF_INET and AF_INET6, and to 0 otherwise.

If the nlmsg_flags field of the struct nlmsghdr header has the NLM_F_DUMP flag set, it means that a list of sockets is being requested; otherwise it is a query about an individ? ual socket.

Response

The response starts with a struct nlmsghdr header and is followed by an array of objects specific to the address family. The array is to be accessed with the standard NLMSG_* macros from the netlink(3) API.

Each object is the NLA (netlink attributes) list that is to be accessed with the RTA_* macros from rtnetlink(3) API.

UNIX domain sockets

For UNIX domain sockets the request is represented in the following structure:

```
struct unix_diag_req {
    __u8    sdiag_family;
    __u8    sdiag_protocol;
    __u16   pad;
    __u32   udiag_states;
    __u32   udiag_ino;
    __u32   udiag_show;
    __u32   udiag_cookie[2];
```

};

The fields of this structure are as follows:

sdiag_family

The address family; it should be set to AF_UNIX.

sdiag_protocol

pad These fields should be set to 0.

udiag_states

This is a bit mask that defines a filter of sockets states. Only those sockets whose states are in this mask will be reported. Ignored when querying for an indi? vidual socket. Supported values are:

1 << TCP_ESTABLISHED

1 << TCP_LISTEN

udiag_ino

This is an inode number when querying for an individual socket. Ignored when querying for a list of sockets.

udiag_show

This is a set of flags defining what kind of information to report. Each requested kind of information is reported back as a netlink attribute as described below:

UDIAG_SHOW_NAME

The attribute reported in answer to this request is UNIX_DIAG_NAME. The payload associated with this attribute is the pathname to which the socket was bound (a sequence of bytes up to UNIX_PATH_MAX length).

UDIAG_SHOW_VFS

The attribute reported in answer to this request is UNIX_DIAG_VFS. The pay? load associated with this attribute is represented in the following struc? ture:

```
struct unix_diag_vfs {
    __u32 udiag_vfs_dev;
    __u32 udiag_vfs_ino;
};
```

The fields of this structure are as follows:

udiag_vfs_dev

The device number of the corresponding on-disk socket inode.

udiag_vfs_ino

The inode number of the corresponding on-disk socket inode.

UDIAG_SHOW_PEER

The attribute reported in answer to this request is UNIX_DIAG_PEER. The payload associated with this attribute is a __u32 value which is the peer's inode number. This attribute is reported for connected sockets only.

UDIAG_SHOW_ICONS

The attribute reported in answer to this request is UNIX_DIAG_ICONS. The payload associated with this attribute is an array of __u32 values which are inode numbers of sockets that has passed the connect(2) call, but hasn't been processed with accept(2) yet. This attribute is reported for listening sockets only.

UDIAG_SHOW_RQLEN

The attribute reported in answer to this request is UNIX_DIAG_RQLEN. The payload associated with this attribute is represented in the following structure:

```
struct unix_diag_rqlen {
    __u32 udiag_rqueue;
    __u32 udiag_wqueue;
};
```

The fields of this structure are as follows:

udiag_rqueue

For listening sockets: the number of pending connections. The length of the array associated with the UNIX_DIAG_ICONS response attribute is equal to this value.

For established sockets: the amount of data in incoming queue.

udiag_wqueue

For listening sockets: the backlog length which equals to the value passed as the second argument to listen(2).

For established sockets: the amount of memory available for sending.

UDIAG_SHOW_MEMINFO

The attribute reported in answer to this request is UNIX_DIAG_MEMINFO. The payload associated with this attribute is an array of __u32 values described

below in the subsection "Socket memory information".

The following attributes are reported back without any specific request:

UNIX_DIAG_SHUTDOWN

The payload associated with this attribute is __u8 value which represents bits of shutdown(2) state.

udiag_cookie

This is an array of opaque identifiers that could be used along with udiag_ino to specify an individual socket. It is ignored when querying for a list of sockets, as well as when all its elements are set to -1.

The response to a query for UNIX domain sockets is represented as an array of

```
struct unix_diag_msg {
    __u8    udiag_family;
    __u8    udiag_type;
    __u8    udiag_state;
    __u8    pad;
    __u32   udiag_ino;
    __u32   udiag_cookie[2];
};
```

followed by netlink attributes.

The fields of this structure are as follows:

udiag_family

This field has the same meaning as in struct unix_diag_req.

udiag_type

This is set to one of SOCK_PACKET, SOCK_STREAM, or SOCK_SEQPACKET.

udiag_state

This is set to one of TCP_LISTEN or TCP_ESTABLISHED.

pad This field is set to 0.

udiag_ino

This is the socket inode number.

udiag_cookie

This is an array of opaque identifiers that could be used in subsequent queries.

IPv4 and IPv6 sockets

For IPv4 and IPv6 sockets, the request is represented in the following structure:

```
struct inet_diag_req_v2 {
    __u8    sdiag_family;
    __u8    sdiag_protocol;
    __u8    idiag_ext;
    __u8    pad;
    __u32   idiag_states;
    struct inet_diag_sockid id;
};
```

where struct inet_diag_sockid is defined as follows:

```
struct inet_diag_sockid {
    __be16  idiag_sport;
    __be16  idiag_dport;
    __be32  idiag_src[4];
    __be32  idiag_dst[4];
    __u32   idiag_if;
    __u32   idiag_cookie[2];
};
```

The fields of struct inet_diag_req_v2 are as follows:

sdiag_family

This should be set to either AF_INET or AF_INET6 for IPv4 or IPv6 sockets respec? tively.

sdiag_protocol

This should be set to one of IPPROTO_TCP, IPPROTO_UDP, or IPPROTO_UDPLITE.

idiag_ext

This is a set of flags defining what kind of extended information to report. Each requested kind of information is reported back as a netlink attribute as described below:

INET_DIAG_TOS

The payload associated with this attribute is a __u8 value which is the TOS of the socket.

INET_DIAG_TCLASS

The payload associated with this attribute is a __u8 value which is the TClass of the socket. IPv6 sockets only. For LISTEN and CLOSE sockets,

this is followed by INET_DIAG_SKV6ONLY attribute with associated __u8 pay?
load value meaning whether the socket is IPv6-only or not.

INET_DIAG_MEMINFO

The payload associated with this attribute is represented in the following structure:

```
struct inet_diag_meminfo {
    __u32 idiag_rmem;
    __u32 idiag_wmem;
    __u32 idiag_fmem;
    __u32 idiag_tmem;
};
```

The fields of this structure are as follows:

idiag_rmem The amount of data in the receive queue.

idiag_wmem The amount of data that is queued by TCP but not yet sent.

idiag_fmem The amount of memory scheduled for future use (TCP only).

idiag_tmem The amount of data in send queue.

INET_DIAG_SKMEMINFO

The payload associated with this attribute is an array of __u32 values de? scribed below in the subsection "Socket memory information".

INET_DIAG_INFO

The payload associated with this attribute is specific to the address fam? ily. For TCP sockets, it is an object of type struct tcp_info.

INET_DIAG_CONG

The payload associated with this attribute is a string that describes the congestion control algorithm used. For TCP sockets only.

pad This should be set to 0.

idiag_states

This is a bit mask that defines a filter of socket states. Only those sockets whose states are in this mask will be reported. Ignored when querying for an indi? vidual socket.

id This is a socket ID object that is used in dump requests, in queries about individ? ual sockets, and is reported back in each response. Unlike UNIX domain sockets, IPv4 and IPv6 sockets are identified using addresses and ports. All values are in

network byte order.

The fields of struct inet_diag_sockid are as follows:

idiag_sport

The source port.

idiag_dport

The destination port.

idiag_src

The source address.

idiag_dst

The destination address.

idiag_if

The interface number the socket is bound to.

idiag_cookie

This is an array of opaque identifiers that could be used along with other fields of this structure to specify an individual socket. It is ignored when querying for a list of sockets, as well as when all its elements are set to -1.

The response to a query for IPv4 or IPv6 sockets is represented as an array of

```
struct inet_diag_msg {
    __u8    idiag_family;
    __u8    idiag_state;
    __u8    idiag_timer;
    __u8    idiag_retrans;
    struct inet_diag_sockid id;
    __u32   idiag_expires;
    __u32   idiag_rqueue;
    __u32   idiag_wqueue;
    __u32   idiag_uid;
    __u32   idiag_inode;
};
```

followed by netlink attributes.

The fields of this structure are as follows:

idiag_family

This is the same field as in struct inet_diag_req_v2.

idiag_state

This denotes socket state as in struct inet_diag_req_v2.

idiag_timer

For TCP sockets, this field describes the type of timer that is currently active for the socket. It is set to one of the following constants:

0 no timer is active

1 a retransmit timer

2 a keep-alive timer

3 a TIME_WAIT timer

4 a zero window probe timer

For non-TCP sockets, this field is set to 0.

idiag_retrans

For idiag_timer values 1, 2, and 4, this field contains the number of retransmits.

For other idiag_timer values, this field is set to 0.

idiag_expires

For TCP sockets that have an active timer, this field describes its expiration time in milliseconds. For other sockets, this field is set to 0.

idiag_rqueue

For listening sockets: the number of pending connections.

For other sockets: the amount of data in the incoming queue.

idiag_wqueue

For listening sockets: the backlog length.

For other sockets: the amount of memory available for sending.

idiag_uid

This is the socket owner UID.

idiag_inode

This is the socket inode number.

Socket memory information

The payload associated with UNIX_DIAG_MEMINFO and INET_DIAG_SKMEMINFO netlink attributes is an array of the following __u32 values:

SK_MEMINFO_RMEM_ALLOC

The amount of data in receive queue.

SK_MEMINFO_RCVBUF

The receive socket buffer as set by SO_RCVBUF.

SK_MEMINFO_WMEM_ALLOC

The amount of data in send queue.

SK_MEMINFO_SNDBUF

The send socket buffer as set by SO_SNDBUF.

SK_MEMINFO_FWD_ALLOC

The amount of memory scheduled for future use (TCP only).

SK_MEMINFO_WMEM_QUEUED

The amount of data queued by TCP, but not yet sent.

SK_MEMINFO_OPTMEM

The amount of memory allocated for the socket's service needs (e.g., socket fil?

ter).

SK_MEMINFO_BACKLOG

The amount of packets in the backlog (not yet processed).

VERSIONS

NETLINK_INET_DIAG was introduced in Linux 2.6.14 and supported AF_INET and AF_INET6 sock?

ets only. In Linux 3.3, it was renamed to NETLINK_SOCK_DIAG and extended to support

AF_UNIX sockets.

UNIX_DIAG_MEMINFO and INET_DIAG_SKMEMINFO were introduced in Linux 3.6.

CONFORMING TO

The NETLINK_SOCK_DIAG API is Linux-specific.

EXAMPLES

The following example program prints inode number, peer's inode number, and name of all

UNIX domain sockets in the current namespace.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <linux/sock_diag.h>
```

```
#include <linux/unix_diag.h>
static int
send_query(int fd)
{
    struct sockaddr_nl nladdr = {
        .nl_family = AF_NETLINK
    };
    struct
    {
        struct nlmsghdr nlh;
        struct unix_diag_req udr;
    } req = {
        .nlh = {
            .nlmsg_len = sizeof(req),
            .nlmsg_type = SOCK_DIAG_BY_FAMILY,
            .nlmsg_flags = NLM_F_REQUEST | NLM_F_DUMP
        },
        .udr = {
            .sdiag_family = AF_UNIX,
            .udiag_states = -1,
            .udiag_show = UDIAG_SHOW_NAME | UDIAG_SHOW_PEER
        }
    };
    struct iovec iov = {
        .iov_base = &req,
        .iov_len = sizeof(req)
    };
    struct msghdr msg = {
        .msg_name = &nladdr,
        .msg_namelen = sizeof(nladdr),
        .msg_iov = &iov,
        .msg_iovlen = 1
    };
```

```
    for (;;) {
        if (sendmsg(fd, &msg, 0) < 0) {
            if (errno == EINTR)
                continue;
            perror("sendmsg");
            return -1;
        }
        return 0;
    }
}
static int
print_diag(const struct unix_diag_msg *diag, unsigned int len)
{
    if (len < NLMSG_LENGTH(sizeof(*diag))) {
        fputs("short response\n", stderr);
        return -1;
    }
    if (diag->udiag_family != AF_UNIX) {
        fprintf(stderr, "unexpected family %u\n", diag->udiag_family);
        return -1;
    }
    unsigned int rta_len = len - NLMSG_LENGTH(sizeof(*diag));
    unsigned int peer = 0;
    size_t path_len = 0;
    char path[sizeof(((struct sockaddr_un *) 0)->sun_path) + 1];
    for (struct rtattr *attr = (struct rtattr *) (diag + 1);
         RTA_OK(attr, rta_len); attr = RTA_NEXT(attr, rta_len)) {
        switch (attr->rta_type) {
        case UNIX_DIAG_NAME:
            if (!path_len) {
                path_len = RTA_PAYLOAD(attr);
                if (path_len > sizeof(path) - 1)
                    path_len = sizeof(path) - 1;
```

```
                memcpy(path, RTA_DATA(attr), path_len);
                path[path_len] = '\0';
            }
            break;
        case UNIX_DIAG_PEER:
            if (RTA_PAYLOAD(attr) >= sizeof(peer))
                peer = *(unsigned int *) RTA_DATA(attr);
            break;
        }
    }
    printf("inode=%u", diag->udiag_ino);
    if (peer)
        printf(", peer=%u", peer);
    if (path_len)
        printf(", name=%s%s", *path ? "" : "@",
                *path ? path : path + 1);
    putchar('\n');
    return 0;
}
static int
receive_responses(int fd)
{
    long buf[8192 / sizeof(long)];
    struct sockaddr_nl nladdr = {
        .nl_family = AF_NETLINK
    };
    struct iovec iov = {
        .iov_base = buf,
        .iov_len = sizeof(buf)
    };
    int flags = 0;
    for (;;) {
        struct msghdr msg = {
```

```
    .msg_name = &nladdr,
    .msg_namelen = sizeof(nladdr),
    .msg_iov = &iov,
    .msg_iovlen = 1
};
ssize_t ret = recvmsg(fd, &msg, flags);
if (ret < 0) {
    if (errno == EINTR)
        continue;
    perror("recvmsg");
    return -1;
}
if (ret == 0)
    return 0;
const struct nlmsghdr *h = (struct nlmsghdr *) buf;
if (!NLMSG_OK(h, ret)) {
    fputs("!NLMSG_OK\n", stderr);
    return -1;
}
for (; NLMSG_OK(h, ret); h = NLMSG_NEXT(h, ret)) {
    if (h->nlmsg_type == NLMSG_DONE)
        return 0;
    if (h->nlmsg_type == NLMSG_ERROR) {
        const struct nlmsgerr *err = NLMSG_DATA(h);
        if (h->nlmsg_len < NLMSG_LENGTH(sizeof(*err))) {
            fputs("NLMSG_ERROR\n", stderr);
        } else {
            errno = -err->error;
            perror("NLMSG_ERROR");
        }
        return -1;
    }
    if (h->nlmsg_type != SOCK_DIAG_BY_FAMILY) {
```

```
                fprintf(stderr, "unexpected nlmsg_type %u\n",
                        (unsigned) h->nlmsg_type);
                return -1;
            }
            if (print_diag(NLMSG_DATA(h), h->nlmsg_len))
                return -1;
        }
    }
}
int
main(void)
{
    int fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_SOCK_DIAG);
    if (fd < 0) {
        perror("socket");
        return 1;
    }
    int ret = send_query(fd) || receive_responses(fd);
    close(fd);
    return ret;
}
```

SEE ALSO

netlink(3), rtnetlink(3), netlink(7), tcp(7)

COLOPHON

This page is part of release 5.10 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/.