NAME

tun - tunnel software network interface

SYNOPSIS

To compile this driver into the kernel, place the following line in your kernel configuration file:

```
device tuntap
```

Alternatively, to load the driver as a module at boot time, place the following lines in loader.conf(5):

```
if_tuntap_load="YES"
```

DESCRIPTION

The **tun** interface is a software loopback mechanism that can be loosely described as the network interface analog of the pty(4), that is, **tun** does for network interfaces what the pty(4) driver does for terminals.

The **tun** driver, like the pty(4) driver, provides two interfaces: an interface like the usual facility it is simulating (a network interface in the case of **tun**, or a terminal for pty(4)), and a character-special device "control" interface. A client program transfers IP (by default) packets to or from the **tun** "control" interface. The tap(4) interface provides similar functionality at the Ethernet layer: a client will transfer Ethernet frames to or from a tap(4) "control" interface.

The network interfaces are named "tun0", "tun1", etc., one for each control device that has been opened. These network interfaces persist until the *if_tuntap.ko* module is unloaded, or until removed with the ifconfig(8) command.

tun devices are created using interface cloning. This is done using the "ifconfig tun**N** create" command. This is the preferred method of creating **tun** devices. The same method allows removal of interfaces. For this, use the "ifconfig tun**N** destroy" command.

If the sysctl(8) variable *net.link.tun.devfs_cloning* is non-zero, the **tun** interface permits opens on the special control device */dev/tun*. When this device is opened, **tun** will return a handle for the lowest unused **tun** device (use devname(3) to determine which).

Disabling the legacy devfs cloning functionality may break existing applications which use **tun***, such as ppp(8) and ssh(1). It therefore defaults to being enabled until further notice.*

Control devices (once successfully opened) persist until *if_tuntap.ko* is unloaded in the same way that network interfaces persist (see above).

Each interface supports the usual network-interface ioctl(2)s, such as SIOCAIFADDR and thus can be used with ifconfig(8) like any other interface. At boot time, they are POINTOPOINT interfaces, but this can be changed; see the description of the control device, below. When the system chooses to transmit a packet on the network interface, the packet can be read from the control device (it appears as "input" there); writing a packet to the control device generates an input packet on the network interface, as if the (non-existent) hardware had just received it.

The tunnel device (*/dev/tunN*) is exclusive-open (it cannot be opened if it is already open). A read(2) call will return an error (EHOSTDOWN) if the interface is not "ready" (which means that the control device is open and the interface's address has been set).

Once the interface is ready, read(2) will return a packet if one is available; if not, it will either block until one is or return EWOULDBLOCK, depending on whether non-blocking I/O has been enabled. If the packet is longer than is allowed for in the buffer passed to read(2), the extra data will be silently dropped.

If the TUNSLMODE ioctl has been set, packets read from the control device will be prepended with the destination address as presented to the network interface output routine, **tunoutput**(). The destination address is in *struct sockaddr* format. The actual length of the prepended address is in the member *sa_len*. If the TUNSIFHEAD ioctl has been set, packets will be prepended with a four byte address family in network byte order. TUNSLMODE and TUNSIFHEAD are mutually exclusive. In any case, the packet data follows immediately.

A write(2) call passes a packet in to be "received" on the pseudo-interface. If the TUNSIFHEAD ioctl has been set, the address family must be prepended, otherwise the packet is assumed to be of type AF_INET. Each write(2) call supplies exactly one packet; the packet length is taken from the amount of data provided to write(2) (minus any supplied address family). Writes will not block; if the packet cannot be accepted for a transient reason (e.g., no buffer space available), it is silently dropped; if the reason is not transient (e.g., packet too large), an error is returned.

The following ioctl(2) calls are supported (defined in <*net/if_tun.h*>):

TUNSDEBUG The argument should be a pointer to an *int*; this sets the internal debugging variable to that value. What, if anything, this variable controls is not documented here; see the source code.

TUNGDEBUG The argument should be a pointer to an *int*; this stores the internal debugging variable's value into it.

TUNSIFINFO The argument should be a pointer to an *struct tuninfo* and allows setting the MTU and

the baudrate of the tunnel device. The type must be the same as returned by TUNGIFINFO or set to IFT_PPP else the ioctl(2) call will fail. The *struct tuninfo* is declared in <*net/if_tun.h*>.

The use of this ioctl is restricted to the super-user.

TUNGIFINFO The argument should be a pointer to an *struct tuninfo*, where the current MTU, type, and baudrate will be stored.

TUNSIFMODE The argument should be a pointer to an *int*; its value must be either IFF_POINTOPOINT or IFF_BROADCAST and should have IFF_MULTICAST OR'd into the value if multicast support is required. The type of the corresponding "tun*N*" interface is set to the supplied type. If the value is outside the above range, an EINVAL error is returned. The interface must be down at the time; if it is up, an EBUSY error is returned.

TUNSLMODE The argument should be a pointer to an *int*; a non-zero value turns off "multi-af" mode and turns on "link-layer" mode, causing packets read from the tunnel device to be prepended with the network destination address (see above).

TUNSIFPID Will set the pid owning the tunnel device to the current process's pid.

TUNSIFHEAD The argument should be a pointer to an *int*; a non-zero value turns off "link-layer" mode, and enables "multi-af" mode, where every packet is preceded with a four byte address family.

TUNGIFHEAD The argument should be a pointer to an *int*; the ioctl sets the value to one if the device is in "multi-af" mode, and zero otherwise.

FIONBIO Turn non-blocking I/O for reads off or on, according as the argument *int*'s value is or is not zero. (Writes are always non-blocking.)

FIOASYNC Turn asynchronous I/O for reads (i.e., generation of SIGIO when data is available to be read) off or on, according as the argument *int*'s value is or is not zero.

FIONREAD If any packets are queued to be read, store the size of the first one into the argument *int*; otherwise, store zero.

TIOCSPGRP Set the process group to receive SIGIO signals, when asynchronous I/O is enabled, to the argument *int* value.

TIOCGPGRP Retrieve the process group value for SIGIO signals into the argument *int* value.

The control device also supports select(2) for read; selecting for write is pointless, and always succeeds, since writes are always non-blocking.

On the last close of the data device, by default, the interface is brought down (as if with **ifconfig** *tunN* **down**). All queued packets are thrown away. If the interface is up when the data device is not open output packets are always thrown away rather than letting them pile up.

SEE ALSO

ioctl(2), read(2), select(2), write(2), devname(3), inet(4), intro(4), pty(4), tap(4), ifconfig(8)

AUTHORS

This manual page was originally obtained from NetBSD.