NAME

VNET - network subsystem virtualization infrastructure

SYNOPSIS

options VIMAGE
options VNET_DEBUG

#include <net/vnet.h>

Constants and Global Variables

VNET_SETNAME VNET_SYMPREFIX
*extern struct vnet *vnet0;*

Variable Declaration

VNET(*name*);

VNET_NAME(*name*);

VNET_DECLARE(*type*, *name*);

VNET_DEFINE(*type*, *name*);

VNET_DEFINE_STATIC(*type*, *name*);

```
#define    V_name   VNET(name)
```

Virtual Instance Selection

CRED_TO_VNET(*struct ucred **);

TD_TO_VNET(*struct thread **);

P_TO_VNET(*struct proc **);

IS_DEFAULT_VNET(*struct vnet **);

VNET_ASSERT(*exp*, *msg*);

CURVNET_SET(*struct vnet **);

CURVNET_SET_QUIET(*struct vnet **);

CURVNET_RESTORE();

VNET_ITERATOR_DECL(*struct vnet **);

VNET_FOREACH(*struct vnet **);

Locking

VNET_LIST_RLOCK();

VNET_LIST_RUNLOCK();

VNET_LIST_RLOCK_NOSLEEP();

VNET_LIST_RUNLOCK_NOSLEEP();

Startup and Teardown Functions

*struct vnet **
vnet_alloc(*void*);

void
vnet_destroy(*struct vnet **);

VNET_SYSINIT(*ident*, *enum sysinit_sub_id subsystem*, *enum sysinit_elem_order order*, *sysinit_cfunc_t func*, *const void *arg*);

VNET_SYSUNINIT(*ident*, *enum sysinit_sub_id subsystem*, *enum sysinit_elem_order order*, *sysinit_cfunc_t func*, *const void *arg*);

Eventhandlers

VNET_GLOBAL_EVENTHANDLER_REGISTER(*const char *name*, *void *func*, *void *arg*, *int priority*);

VNET_GLOBAL_EVENTHANDLER_REGISTER_TAG(*eventhandler_tag tag*, *const char *name*, *void *func*, *void *arg*, *int priority*);

Sysctl Handling

SYSCTL_VNET_INT(*parent*, *nbr*, *name*, *access*, *ptr*, *val*, *descr*);

SYSCTL_VNET_PROC(*parent*, *nbr*, *name*, *access*, *ptr*, *arg*, *handler*, *fmt*, *descr*);

SYSCTL_VNET_STRING(*parent*, *nbr*, *name*, *access*, *arg*, *len*, *descr*);

SYSCTL_VNET_STRUCT(*parent*, *nbr*, *name*, *access*, *ptr*, *type*, *descr*);

SYSCTL_VNET_UINT(*parent*, *nbr*, *name*, *access*, *ptr*, *val*, *descr*);

VNET_SYSCTL_ARG(*req*, *arg1*);

DESCRIPTION

VNET is the name of a technique to virtualize the network stack. The basic idea is to change global resources most notably variables into per network stack resources and have functions, sysctls, eventhandlers, etc. access and handle them in the context of the correct instance. Each (virtual) network stack is attached to a *prison*, with *vnet0* being the unrestricted default network stack of the base system.

The global defines for VNET_SETNAME and VNET_SYMPREFIX are shared with kvm(3) to access internals for debugging reasons.

Variable Declaration

Variables are virtualized by using the **VNET_DEFINE**() macro rather than writing them out as *type name*. One can still use static initialization, e.g.,

```
VNET_DEFINE(int, foo) = 1;
```

Variables declared with the static keyword can use the **VNET_DEFINE_STATIC**() macro, e.g.,

```
VNET_DEFINE_STATIC(SLIST_HEAD(, bar), bars);
```

Static initialization is not possible when the virtualized variable would need to be referenced, e.g., with "TAILQ_HEAD_INITIALIZER()". In that case a **VNET_SYSINIT**() based initialization function must be used.

External variables have to be declared using the **VNET_DECLARE**() macro. In either case the convention is to define another macro, that is then used throughout the implementation to access that variable. The variable name is usually prefixed by *V_* to express that it is virtualized. The **VNET**() macro will then translate accesses to that variable to the copy of the currently selected instance (see the *Virtual instance selection* section):

```
#define V_name          VNET(name)
```

NOTE: Do not confuse this with the convention used by VFS(9).

The **VNET_NAME**() macro returns the offset within the memory region of the virtual network stack instance. It is usually only used with **SYSCTL_VNET_***() macros.

Virtual Instance Selection

There are three different places where the current virtual network stack pointer is stored and can be taken from:

1. a *prison*:

    (struct prison *)->pr_vnet

    For convenience the following macros are provided:

    CRED_TO_VNET(*struct ucred **)
    TD_TO_VNET(*struct thread **)
    P_TO_VNET(*struct proc **)

2. a *socket*:

    (struct socket *)->so_vnet

3. an *interface*:

    (struct ifnet *)->if_vnet

In addition the currently active instance is cached in "curthread->td_vnet" which is usually only accessed through the curvnet macro.

To set the correct context of the current virtual network instance, use the **CURVNET_SET**() or **CURVNET_SET_QUIET**() macros. The **CURVNET_SET_QUIET**() version will not record vnet recursions in case the kernel was compiled with **options VNET_DEBUG** and should thus only be used in well known cases, where recursion is unavoidable. Both macros will save the previous state on the stack and it must be restored with the **CURVNET_RESTORE**() macro.

NOTE: As the previous state is saved on the stack, you cannot have multiple **CURVNET_SET**() calls in the same block.

NOTE: As the previous state is saved on the stack, a **CURVNET_RESTORE**() call has to be in the same block as the **CURVNET_SET**() call or in a subblock with the same idea of the saved instances as the outer block.

NOTE: As each macro is a set of operations and, as previously explained, cannot be put into its own block when defined, one cannot conditionally set the current vnet context. The following will *not* work:

```
      if (condition)
              CURVNET_SET(vnet);
```

nor would this work:

```
      if (condition) {
              CURVNET_SET(vnet);
      }
      CURVNET_RESTORE();
```

Sometimes one needs to loop over all virtual instances, for example to update virtual from global state, to run a function from a callout(9) for each instance, etc. For those cases the **VNET_ITERATOR_DECL**() and **VNET_FOREACH**() macros are provided. The former macro defines the variable that iterates over the loop, and the latter loops over all of the virtual network stack instances. See *Locking* for how to savely traverse the list of all virtual instances.

The **IS_DEFAULT_VNET**() macro provides a safe way to check whether the currently active instance is the unrestricted default network stack of the base system (*vnet0*).

The **VNET_ASSERT**() macro provides a way to conditionally add assertions that are only active with **options VIMAGE** compiled in and either **options VNET_DEBUG** or **options INVARIANTS** enabled as well. It uses the same semantics as KASSERT(9).

Locking

For public access to the list of virtual network stack instances e.g., by the **VNET_FOREACH**() macro, read locks are provided. Macros are used to abstract from the actual type of the locks. If a caller may sleep while traversing the list, it must use the **VNET_LIST_RLOCK**() and **VNET_LIST_RUNLOCK**() macros. Otherwise, the caller can use **VNET_LIST_RLOCK_NOSLEEP**() and **VNET_LIST_RUNLOCK_NOSLEEP**().

Startup and Teardown Functions

To start or tear down a virtual network stack instance the internal functions **vnet_alloc**() and **vnet_destroy**() are provided and called from the jail framework. They run the publicly provided methods to handle network stack startup and teardown.

For public control, the system startup interface has been enhanced to not only handle a system boot but to also handle a virtual network stack startup and teardown. To the base system the **VNET_SYSINIT**() and **VNET_SYSUNINIT**() macros look exactly as if there were no virtual network stack. In fact, if **options VIMAGE** is not compiled in they are compiled to the standard **SYSINIT**() macros. In addition to that they are run for each virtual network stack when starting or, in reverse order, when shutting down.

Eventhandlers

Eventhandlers can be handled in two ways:

1. save the *tags* returned in each virtual instance and properly free the eventhandlers on teardown using those, or
2. use one eventhandler that will iterate over all virtual network stack instances.

For the first case one can just use the normal EVENTHANDLER(9) functions, while for the second case the **VNET_GLOBAL_EVENTHANDLER_REGISTER**() and **VNET_GLOBAL_EVENTHANDLER_REGISTER_TAG**() macros are provided. These differ in that **VNET_GLOBAL_EVENTHANDLER_REGISTER_TAG**() takes an extra first argument that will carry the *tag* upon return. Eventhandlers registered with either of these will not run *func* directly but *func* will be called from an internal iterator function for each vnet. Both macros can only be used for eventhandlers that do not take additional arguments, as the variadic arguments from an EVENTHANDLER_INVOKE(9) call will be ignored.

Sysctl Handling

A sysctl(9) can be virtualized by using one of the **SYSCTL_VNET_***() macros.

They take the same arguments as the standard sysctl(9) functions, with the only difference, that the *ptr* argument has to be passed as ‘&VNET_NAME(foo)’ instead of ‘&foo’ so that the variable can be selected from the correct memory region of the virtual network stack instance of the caller.

For the very rare case a sysctl handler function would want to handle *arg1* itself the **VNET_SYSCTL_ARG**(*req*, *arg1*) is provided that will translate the *arg1* argument to the correct memory address in the virtual network stack context of the caller.

SEE ALSO

jail(2), kvm(3), EVENTHANDLER(9), KASSERT(9), sysctl(9)

Marko Zec, Implementing a Clonable Network Stack in the FreeBSD Kernel, USENIX ATC’03, June 2003, Boston

HISTORY

The virtual network stack implementation first appeared in FreeBSD 8.0.

AUTHORS

The **VNET** framework was designed and implemented at the University of Zagreb by Marko Zec under sponsorship of the FreeBSD Foundation and NLnet Foundation, and later extended and refined by Bjoern A. Zeeb (also under FreeBSD Foundation sponsorship), and Robert Watson.

This manual page was written by Bjoern A. Zeeb, CK Software GmbH, under sponsorship from the FreeBSD Foundation.