Sun UNIX Modifications to use the Sun Network File Service

Bill Joy

Company Confidential

ABSTRACT

The Sun Network File Service provides shared access to a set of files. The implementation consists of three main parts: modifications to UNIX to support several drivers for file systems, a file system driver which speaks to the Network File Service, and the server code which implements the Network File Service.

This document describes the virtual file system interface and includes an appendix with the draft code for the file system implementation-independent code to be included in UNIX.

It also includes a section answering specific questions about the functional divisions between the base-level UNIX system, the driver which speaks to the Network File Service and the Network File Service itself, assuming that the latter two are the client and server-side implementations of the standard Sun Network File Service.

Overview

Sun version 1.0 of UNIX provides transport level network services which allow remote login and file transfer. It is easy to write additional UNIX application programs to provide other network services. The system supports a ***network disk*** protocol which allows nodes to operate without local disks. This protocol does not support shared access to network files.

To support shared access to network files we are modifying UNIX to support multiple implementations of its file system. Each implementation acts as a file system driver called by implementation independent routines in the mainline system. A driver for the file system can then be written to talk to the Sun Network File Service without modifying the mainline UNIX code.

To the UNIX user, file systems accessed over the network and local file systems are indistinguishable. Both local and remote file systems can be ***mounted*** into the file system hierarchy.

The implementation of the Sun Network File Service consists of three main parts: the modifications to UNIX to support the virtual file system concept, the writing of a UNIX file system driver to talk to the Sun Network File Service, and the implementation of the code to run within UNIX to implement the Sun Network File Service.

1. Facilities in the 1.0 release of UNIX

The current version of UNIX contains a file system implementation which is present in each workstation, and which interfaces to the disk drivers in the system. Each disk driver provides read and write operations on its underlying disk drive. A disk driver exists which accesses a

disk partition on another machine. This network disk driver allows diskless operation.

The interface between the mainline UNIX system and the disk driver is managed by a set of buffer management routines. These routines allow both cached and uncached input/output operations to occur.

A higher-level concept in the UNIX system is the ***inode***, which is the central input/output system abstraction. Each open file, directory, or device is represented in the system as an inode. UNIX system calls, such as read and writes to files or devices and operations which create and remove files all apply to inodes.

Inodes are uniquely identified by the ***device*** in which they are contained and a small integer ***number***. Each file system contains a set of inodes with numbers from 1 to a fixed upper bound, typically a few tens of thousands.

The system maintains a table of active inodes. In particular, the current working directory of a process is an inode, and an operation ***namei*** exists which takes a pathname and searches the set of active file systems for the named file or device. *Namei* converts the pathname into a pointer to a locked inode, with side effects which are usable in creating and removing files.

A typical system call which takes a pathname will call ***namei*** to translate the pathname, and then perform operations on the inode which is returned. If the operation is creating a file, then ***namei*** will return with the containing directory locked, and return information to allow the new entry to be created. *Namei* is both a central and complex routine.

User programs running in UNIX make system calls to perform operations on both open files and on files whose names are specified relative to the root directory of the file system or to the current working directory. Information about open files and active communications channels is maintained in an open ***file table***. Stored in this table for active inodes is a pointer to an inode. The file table implementation is object oriented, so that each file table structure references a set of functions for performing operations on the contained object, such as reading and writing.

Operations which apply to named files use the ***namei*** routine to translate the pathname, and then use some low level information returned by the ***namei*** routine, or inode-level operations to achieve their desired result. As an example, removal of a file is achieved by calling ***namei*** giving a path name and an indication that the named is to be removed. *Namei* searches the directories involved in the file name, and when it finds the entry in the final directory it leaves information as to the offset and size of the name to be removed from the directory in a process-global data structure. The system call handler then calls a sibling routine to ***namei*** to remove the entry from the directory structure. If this is the last directory reference to the underlying ***inode***, inode-level routines are invoked to free the space associated with the inode and make the ***inode*** slot available for reallocation.

The system handles pathnames involving multiple file systems. The ***mount*** command indicates that a directory within one file system, is to be overlaid with the root of another file system. A call:

```
# /etc/mount /dev/sd0g /usr
```

indicates that the file system structure contained in the disk partition "sd0g" is to be made available beginning at the path name "/usr." Internally this is implemented by marking the inode for the "/usr" directory (which is in the root file system) as being mounted on. When a pathname search encounters a name beginning with "/usr", the search encounters the inode for the "/usr" directory and notices that it is mounted on. This inode indicates the new inode at which the search is to continue, and the search is continued with inodes from the newly mounted "sd0g" file system. UNIX uses the fact that the root inode of each file system has a fixed

number ROOTINO to perform the indirection.

One other facility supports the use of mounted file systems. It is a convention in each directory that the pathname ".." refers to the parent directory of the current directory. This is possible because UNIX enforces the structuring of the directory hierarchy into a tree, so that each directory has a unique parent. However, in the root directory of each mounted file system the ".." entry refers to the directory itself, since directory entries cannot reference across file system boundaries. The interpretation of this entry as a reference to its parent can occur only dynamically in the ***namei*** routine, as it encounters a ".." in the root directory of a mounted file system.

An important point here is that the relationship between different file systems is maintained ***actively*** by the file system. That is, the "sd0g" disk partition does not permanently have the name "/usr". Its files are accessible at that pathname only because the ***mount*** command made them available there.

2. Structure for the new release

In the new release of UNIX each workstation contains code which translates system calls on files into operations on a set of "virtual inodes" or ***vnodes***. Each ***vnode*** contains a pointer to an array of functions which implement the operations necessary to support an abstract file system. With each workstation is also loaded one or more virtual file system (or ***vfs***) implementors who supply the vnode operations.

The two implementors for the first new release are the local file system implementation ***unixfs*** which implements a file system in a disk partition, as before, and ***netfs*** which accesses a file system on a remote machine which supports the Sun Network File Service protocol. A workstation containing the ***netfs*** implementation can access files shared with other workstations.

The approach described here allows UNIX to talk not just to a particular file server, but makes it possible to write virtual file system drivers for other file servers and to access files stored on other operating systems.

The interface between the mainline UNIX system and the virtual file system uses the ***vnode*** data structure. Each workstation maintains a table of vnodes, with the pair (vfs,number) logically replacing the (dev,number) pair of the current release.

The current working directory of a process is given by a ***vnode***. Operations involving path names are broken down by the system into a number of operations involving a directory and a component name. Thus to translate the pathname "/bin/sh" into a ***vnode*** pointer, the system first takes the ***vnode*** pointer for the root directory "/", and invokes the LOOKUP operation of this ***vnode***'s implementor, giving it as argument the character string "bin". If this operation succeeds, it will return a ***vnode*** for the "/bin" directory. This ***vnode*** can then be asked to LOOKUP the string "sh", and the resulting ***vnode*** is a reference to "/bin/sh". The path traversal and series of ***vnode*** operations is performed by the routine ***apply*** which is part of the base UNIX system.

The ***apply*** routine is, in many ways, a logical replacement for the ***namei*** routine of the previous system. It differs from ***namei***, however, in that it does not return ***vnodes*** locked, and does not leave information as side effects which can be used to perform further operations on the locked ***vnodes***. Rather than having a number of operations on ***vnodes*** result from each UNIX-level operation, the ***vnode*** abstraction is designed so that each UNIX-level operation can be resolved by a sequence of calls on the ***vnode*** primitives so that its effect is then achieved by a single ***vnode*** operation. The preliminary operations can be used to translate pathnames, but the final operation atomically performs all of the permanent actions.

A typical system call which takes a pathname will thus call ***apply*** to translate the pathname, passing it the operation and arguments to the operation which is to be performed on the named file. If the operation is one which applies to the directory containing the file (e.g. remove a file, or create a file), then ***apply*** will ultimately invoke the operation on the containing directory. If the operation is one which applies to the file itself (e.g. change its owner), then ***apply*** insists that the file exist, and if it does invokes the operation on the file itself.

In both cases the operation may encounter symbolic links. These are files marked as being special and which cause their contents to be substituted into the pathname as it is translated. The base-level UNIX system code is designed so that such symbolic links are always followed if they occur at any level in the pathname translation except at the final entry. If an operation is applied to a symbolic link but decides it would rather operate on the file referenced by the link, it can return a characteristic error to the ***apply*** routine and ask it to further interpret the contents of this link.

As in the current system, the system handles pathnames involving multiple virtual file systems. The ***netmount*** command, in addition to the ***mount*** command can be used to indicate that a directory withing one file system is to be overlaid with the root of another file system. In the new system it is the ***vnode***, rather than the ***inode*** which is marked as being overlaid, and the ***vfs*** table contains a function which can be invoked to return the root ***vnode*** of the mounted file system. We use a function rather than storing a pointer to a ***vnode*** because we expect that the ***vnode*** for a remote mounted file system will only be initialized upon use, since the remote system may not be available when the ***netmount*** occurs.

The ".." convention for remote file systems is supported by the ***apply*** routine, by looking at flags which indicate when a ***vnode*** is the root of a mounted file system.

The relationship between mounted file systems is maintained on each workstation, so that it is possible for each workstation to have access to a different set of file systems.

3. Functional Division

This section discusses where and how various facilities are implemented to support shared access to a file from this new release of the system to a Sun Network File Server. Most of the answers are determined not by the ***vfs*** and ***vnode*** structures described here, but by more global network architecture. We include the discussion for completeness.

How is booting performed?

A program resident in a prom in the workstation loads UNIX from a boot server. UNIX then locates its root file system on the network, using a broadcast-based datagram protocol. UNIX can then begin to access files on this file server in the same way that it does after any other ***netmount*** command.

How do more network file systems become available?

When UNIX boots it runs a standard startup program ***/etc/init***. Shell scripts run by ***/etc/init*** invoke further ***netmount*** commands to make various file systems on the network available.

Who guarantees integrity of the network file systems?

A machine which implements the Sun Network File Service protocol retains control over the integrity of its file system. The operations which are specified by the remote clients of such a file system are high-level and cannot affect the integrity of the server file systems. The server operates as a normal UNIX system and checks all of its file systems each time it reboots. The

results of this check are not affected by the fact that remote clients were accessing the servers files.

What happens when the server crashes?

A server crash results in the loss of some state in the Network File Service data structures on that server. This will cause each client workstation to have to do additional work when the server reboots to reestablish authentication and a secure communications channel. Since the File Service has no volatile state which can not be reconstructed by use of the File Service protocol, machines accessing the service will not lose state as a result of a File Service crash. Rather, they will only see a service interruption.

How are network files protected?

Network files are subject to normal UNIX protection checks. Each client workstation has some set of user-id's which are authenticated for use on the remote server. Other id's access the remote server as an outlandish user-id which normally denies all access. In particular, most workstations will not have access as ***root***, the privileged user, to the files stored on the File Service.

What if a server is down when UNIX wishes to boot?

If the server which is down is the server for a critical file system, e.g. the ***root*** file system of the client workstation, then the workstation will not be able to boot until the server becomes available again. Non-critical file systems on other machines will become available as soon as the machine on which they are mounted is available.

How can a new file system be added to the network?

Any disk partition containing a UNIX file system can be made available to other machines on the network by the machine which is attached to the disk containing the partition. The only preparation the UNIX machine must make is to mount the file system when it boots.

4. Description of draft code

As an appendix this document contains listings of a number of files which define and implement the ***vfs*** and ***vnode*** operations for the base-level system. These files are:

vfs.h

This file defines the structure per virtual file system. It shows the operations which each virtual file system must supply, and the structure whereby arguments to a generic mount routine are passed to a particular ***vfs*** implementor.

vfs_pathname.h

This file shows the structure used to pass pathnames between the base-level system routines, e.g as an argument to ***apply***.

vnode.h

This file defines the very important ***vnode*** structure. The structure itself has many fields in common with the older ***inode*** structure of the current 1.0 system. The most important changes are the deletion from the structure of the implementation-specific fields, and the addition of the ***v_op*** procedures to perform operations.

Also defined in this file is the ***vattr*** structure. Operations which lookup names in directories return a ***vnode*** pointer and a ***vattr*** structure containing information about the file or directory referenced by the ***vnode***.

The final structure defined here is the ***vopargres*** structure, which is passed as argument to each of the ***v_op*** routines, and in which each of those routines returns a result. Not all the fields in the structure are used by each routine.

vfs_apply.c

This file contains the important ***apply*** routine, which processes the semantics for pathname translation and operations.

vfs_mount.c

This file contains the generic implementations of the ***mount*** and ***netmount*** system calls.

vfs_pathname.c

This file contains the routines supporting use of the ***pathname*** data structure defined in vfs_pathname.h.

vfs_syscalls.c

This file contains the base-level system's implementation of the system calls which apply to files (and thus to vnodes), except for read, write and ioctl.

vnode.c

This file contains the support routines for maintaining the ***vnode*** data structure instances.

vnode_rwios.c

This file contains the routines which implement read, write, ioctl and select on vnodes, as well as the system locking facilities.