NFS

The NFS module implements the client side of the Network File System, using RPC and running on top of UDP. The process requires no configuration. However, it does assume the presence of both a DNS and a NIS module in the system.

Caveat

In its current implementation, the NFS code does not follow symbolic links automatically, because of the various possibilities of remote-mount points across systems. Also, although the *DEV_LIST* message is supported and returns a list of mount points, the device attributes of the mount points are not returned in the reply.

Process Information

Prototype Name	nfs
Link Order	does not matter
Process Name	“nfs”

Process Operation

The module has only a main process. It accepts the *Standard* messageset for dataflow communications and the *Files* messageset for filing system operations.

Protocol Design

The following table shows how the NFS protocol elements are mapped to the corresponding shared library routines and ROME messages.

command	level	stdlib	message	ROME routine
GETATTR	[2,3]	*stat*	FSLIB_ATTR_GET	*fslib_get_attr*
SETATTR	[2,3]	*chown, chmod*	FSLIB_ATTR_SET	*fslib_set_attr*
ROOT	[2]			—[1]
LOOKUP	[2,3]			*fopen*[2]
ACCESS	[3]	*access*		—[3]
READLINK	[2,3]	*readlink*	FSLIB_READLINK	*readlink*
READ	[2,3]	*read*	*Standard*	*fread etc.*
WRITECACHE	[2]			—[4]
WRITE	[2,3]	*write*	*Standard*	*fwrite etc.*
CREATE	[2,3]	*open*	OPEN	*fopen*[2]
REMOVE	[2,3]	*unlink*	FSLIB_DELETE	*delete*
RENAME	[2,3]	*rename*	FSLIB_RENAME	*rename*
LINK	[2,3]	*link*	FSLIB_LINK	*link*
SYMLINK	[2,3]	*symlink*	FSLIB_LINK	*l*ink
MKDIR	[2,3]	*mkdir*	FSLIB_MKDIR	*mkdir*
RMDIR	[2,3]	*rmdir*	FSLIB_RMDIR	*rmdir*
READDIR	[2,3]		FSLIB_READDIR	*readdir*[5]
READDIRPLUS	[3]		FSLIB_READDIR	*readdir*[5]
STATFS	[3]		FSLIB_DEV_ATTR	*fslib_get_device_attr*[6]
FSSTAT	[3]		FSLIB_DEV_ATTR	*fslib_get_device_attr*[6]
FSINFO	[3]		FSLIB_DEV_ATTR	*fslib_get_device_attr*[6]
MKNOD	[3]	*mknod*		—[7]
PATHCONF	[3]	*fstat64*	FSLIB_READDIR	*readdir*[5]
COMMIT	[3]	*sync*		—[8]

Notes

1. The ROOT operation is obsolete and has been replaced by the *MOUNT* protocol, which is invoked as needed during *OPEN* processing.

2. The NFS server is (almost) stateless with no explicit *OPEN* or *CLOSE* operations. The *LOOKUP* and *CREATE* operations are performed during *OPEN* processing, according to the read or write options on the *OPEN* message.

3. There is no direct support for the *access* routine, the information can be derived from the directory-level operations.

4. The *WRITECACHE* operation is obsolete.

5. The generic *readdir* operation is a superset of the level-2 and level-3 functionality.

6. The generic device interrogation interface combines these three routines.

7. The current ROME implementation does not support the creation of special device nodes on the remote machine

8. The NFS level-3 caching operations are not supported.

Authorisation

Although some operations (for example reading the export list from a server) can be performed without any authentication, more NFS RPCs require authorisation. In the current implementation of the NFS client, the intention is that the applications all run under the same *UNIX* userid and group, these values must be set globally before any of the NFS operations are invoked. This is done by calling the *nfs_authorize* routine.

URL Processing

Handling file names is particularly complicated in NFS because of the lack of state and the various possibilities for mounting and accessing remote filing systems. The URL passed on the *OPEN* message contains a full path to a file, for example:

nfs://anlicus.nec-lab.com/export/home/ljf/max

or

nfs://~ljf/max

which must be converted into an NFS handle for subsequent operations.The first stage is to resolve usernames into file-system mount points. Firstly the *passwd.byname* map is used to retrieve the user's home directory. Then the *auto_home* map resolves the automount point to a host and filing system. After these operations, all filenames are in the first format given above, with an explicit hostname and path.

The next stage is to find the appropriate mount point on the remote filesystem. If this is the first time a file has been accessed from this host, the *MOUNTPROG* service is used to read the list of exported filing systems on that machine, and the returned list is cached for future reference. Once the list is known, it is scanned to find a matching directory prefix with the supplied path.

If the directory has not previously been accessed the *MNT* procedure of the mount protocol is used to retrieve the handle for the mount point. This leaves a path and terminal filename relative to a known file handle on the remote system. For example:

nfs://anlicus.nec-lab.com/export/home/ljf/max = File-Handle + /ljf/ + max

The path is resolved into a sequence of File Handles using the *LOOKUP* procedure of the main *NFS* protocol, and checking that each returned handle is to a directory. Finally, the last component is resolved relative to its containing directory. If this file does not exist, and the *OPEN* was for writing, a new file is made with the *CREATE* procedure on the remote system, otherwise an error is returned.

The expected result of this processing is to generate a File-Handle for the destination file and have a datapath open to the remote NFS server for subsequent operations.

Messages

CLOSE messages caused any buffered output to be written to the remote device, and the data structures associated with the file are freed.

FETMBLK/GETMBLK messages cause internally-buffered data to be transferred to the application's buffer or the *READ* procedure to be invoked remotely to transfer data to the local machine. *GETMBLK* replies contain the returns from remote procedure calls and are handled within the RPC layer, as described in the RPC module documentation.

FLUSH messages force any buffered output to be written to the remote file.

FSLIB_ATTR_GET messages return the filing system attributes for the URL passed in the *name* parameter of the message.

FSLIB_ATTR_SET messages set the attributes for the URL passed in the *name* parameter. Currently only the *mode, uid* and *gid* parameters are updated.

FSLIB_DELETE messages first locate the file handle using the same method as for *OPEN* and then issue the *REMOVE* procedure on that handle.

FSLIB_DEV_ATTR messages return the device attributes for the file system specified in the *devspec* parameter. The values returned are for the mount point on which the file resides.

FSLIB_DEV_LIST messages return a list of the mount points for the hostname specified in the *devspec* parameter.

FSLIB_LINK messages create either hard or symbolc links in the filing system.

FSLIB_MKDIR messages first locate the parent directory file handle using the same method as for *OPEN* and then issue the *MKDIR* procedure on the final component.

FSLIB_READDIR messages use the *READDIR* procedure to retrieve directory entries from the remote directory. The 'cookie' value is saved between calls to allow multiple calls to retrieve a list of entries in sequence.

FSLIB_RENAME messages use the RENAME procedure to move the file. Renaming a file across different directories on the same device is supported.

FSLIB_RMDIR messages first locate the parent directory file handle using the same method as for *OPEN* and then issue the *RMDIR* procedure on the final component.

FSLIB_SEEK messages set the file position which will be used to read or write data on subsequent calls.

FSLIB_TELL messages return the current file position.

FSLIB_TRUNCATE messages set the size of the file to the current file position by updating the file size attribute directly.

OPEN messages are handled as described in the 'URL' section above.

OUTMBLK/PUTMBLK messages invoke the *WRITE* procedure to transfer data to the remote file.

NEWMBLK messages return buffers for applications to use

RETMBLK messages just free the associated buffers.

TIMEOUT messages are used to trigger re-transmissions at the RPC layer for requests originating from this process.

Shared Library Macros and Routines

nfs_authorize

void *nfs_authorize*(
char **host*,
uint *uid*,
uint *gid*)

The *nfs_authorize* routine sets the *UNIX* authorisation for subsequent NFS calls. It is assumed that only one userid is active for all processes within the system.