sun microsystems, inc.
internal memorandum

To: File

From: Bill Joy, x7254; Steve Kleiman x7295

Subject: NFSBOX: Design Overview

Date: February 19, 1985

Introduction

Goals

The NFSBOX is designed to be a fast, autonomous NFS server which can run with less hardware than a full UNIX implementation.

Speed is achieved by having simple data structures and a runtime environment tuned to i/o processing. By running NFSBOX in a standalone (non-UNIX) environment, UNIX CPU and memory overhead is eliminated, and simplifications not possible under a full-function UNIX are possible.

Autonomy is the ability to make independent decisions. It is achieved in NFSBOX by having labels on each disk sector, and always giving precedence to information in the disk label over any other information.

A NFSBOX is easier to administer than a UNIX-based NFS server because it can run with the code in prom's, and does not require user-intervention except in the case of hardware failure.

Data Structures

Free Storage

A NFSBOX disk is organized into 8.5kb blocks, consisting of a 512 byte header and 8kb of data. At the beginning of each disk are two copies of the free list for the disk, each containing a 2^16 bits indicating which blocks are available. This allows disks up to 2^29 bytes in size. The ''pack label'' for the disk drive is contained in the label blocks for the free list.

```
#define HEADER_SIZE     512
#define BLOCK_SIZE      8192

typedef struct freemap_block {
        int     bits[BLOCK_SIZE/sizeof(int)];
} FREEMAP;
```

Directories

A NFSBOX directory is a file, which contains an array of 256-byte entries. Each entry is a record containing information about the file and the file's name.

```
#define NDADDR  12
#define NIADDR  3

typedef struct directory_block {
        char     name[128];
        int      namelen;
        u_short  mode;
        short    uid;
        short    gid;
        long     size
        struct   timeval atime;
        struct   timeval mtime;
        struct   timeval ctime;
        daddr_t  db[NDADDR];
        daddr_t  ib[NIADDR];
        long     flags;
        long     blocks;
        long     gen;
};
```

File Data

File data is stored as in UNIX 4.2, with direct and indirect data blocks.

Labels

Each disk block has a label, indicating that it is either a free map block, a directory block, a file block, an indirect block, a double indirect block, a data block, or free. The label is the final arbiter of the status of the block.

The label also contains the file's high-level name, and the disk address of the directory block in which the name is stored. This information is used to reconstruct directory blocks which are lost.

Each file is always allocated at least one block, and it is the disk address of this block which is the primary low-level name (fhandle) for the file.

The size of a file is stored in each label_block. Files which are active may not yet have their size fully reflected in their directory entries. In this case the sizes reflected in the last few data blocks must be reflected back to the main directory entry before the length there is correct. After a crash, this may require some extra work (to be described).

```
typedef struct label_block {
        char     name[128];
        int      namelen;
        daddr_t  dirblock;
        int      dirslot;
        enum     label_type {
                 freemap,
                 directory,
                 file,
                 indirect,
                 doubleindirect,
                free
        } type;
        int     size;
        daddr_t parent;                 /* for directory */
} LABEL_BLOCK;
```

Operations

Formatting

Formatting a disk requires writing the free maps and all the headers to mark all the blocks on the disk free. In addition a ''root directory'' for the disk is created with block #2 as its first data block.

Block Allocation

The allocation map on the disk is always a superset of the available space. Allocation requires selecting a free bit from this map and checking the disk lable to insure that it is really free. This means that the map is always a superset of the space available. WHY TWO COPIES OF THE MAP?

Block Free

Consider removing a file. Allocate a block, clear it, and then collect the set of bits which are the blocks in the file. Then write this free map out. Then mark the file as ''free in progress''. This committs the deletion of the file. Then read each block of the file and mark the block as free if it was in the file. Then remove the ''free in progress'' mark on the file.

File Allocation

A file is allocated by finding a slot in a directory and initializing it as ''create in progress'' and then allocating a disk block and writing it out pointing back at the file. This is the committ. We then clear the ''create in progress'' bit.

File Free

See block free. Then erase the directory entry.

Directory Allocation

Same as file allocation.

Directory Free

Same as file allocation.

Continuation After Crashes

Block Allocation in progress

A file which is actively allocating blocks is marked allocation in progress. If we are allocating within 100 blocks of the last block in the file then we don't require that the direct/indirect pointer be immediately updated, but require these blocks to be scanned on first access when restarting after a crash. The search can be restricted to free disk blocks, and each free block need be examined at most once.

Block Free in Progress

Blocks which are in the free list may not be freed, because the file they are in is deletion in progress. If we find a block in the free list which is not actually free on the disk we check the directory entry for that file. If the directory entry indicates free in progress, we start up an asynchronous operation to finish freeing the file.

File Allocation in Progress

Allocating a file is committed when the first block is allocated. If we discover an entry that is marked allocation in progress, we finish it if the block is allocated, or discard it if the block is not.

File Free in Progress

If we discover a file which is marked free in progress, we queue an asynchronous operation to redo the free operation.

Hard Operations

Rename

Requires changing all the data blocks. Similar to remove.

Link

This is hard.

Small file space optimizations

Must split blocks. MESSY.