Recap 1

• Devices
  – Block devices
  – Character devices
• Device Controllers
• DMA vs Programmed I/O
• I/O Software Goals
  – Abstract away from device specifics
  – Uniform naming
  – Handle errors robustly
Recap 2

• Device Drivers
  – Autoconf & Init
  – Top Half (I/O Requests)
  – Bottom Half (Interrupt Service Routines)
• Entry Points for Block Devices
• Entry Points for Character Devices
The diagram shows the system call interface to the kernel and various layers of the operating system architecture, including:

- **The hardware**
- **Block device driver**
- **Character device driver**
- **Buffer cache**
- **Virtual Memory (VM)**
- **Network interface drivers**
  - **NFS**
  - **Local naming (UFS)**
  - **Special devs**
  - **FFS**
  - **IFS**
  - **MFS**
- **System call interface to the kernel**
  - **Cooked disk**
  - **Raw Disk And TTY**
  - **TTY**
  - **Swap Space mgmt**
  - **Line discipline**

The diagram illustrates the flow of data and control through these layers, emphasizing the interaction between the kernel and various hardware components.
Disks

• Disk Performance
  – Seek Time
  – Rotational Latency
  – Time to Transfer the Actual Data
• Example (Cheetah 73LP 73GB Disk)
  – 10K RPM; 2.99 msec avg. latency
  – 5.1 msec/5.5 msec avg seek time read/write
  – Internal transfer rate 399 to 671 Mbits/sec
  – 4 discs/8 heads; 29,549 cylinders
  – Seek errors: 10 per 10^8; Read errors: 10 per 10^12
Disk Arm Scheduling

• FCFS - simple but has lousy performance
• Pick - like FCFS, but if we pass something that we want, we service it
• Shortest Seek Time First - greedy algorithm; it tends to stay in the middle of the disk and starve outer requests
• Scan - arm moves one way filling requests, at the end (last request) it reverses direction and does the same going the other way. No starvation. Also called elevator.
• Circular Scan - like scan, but only fills request in one direction. Then it goes to the farthest request in the other direction and begins again.
RAID

- Redundant Array of Independent Disks
- Mirroring - two disks contain the same data. Allows for fault tolerance and possibly multiple readers.
- Striping - spread the contents of one disk on several, so consecutive blocks are on different disks. This makes large reads faster.
- N striped disks with a parity or error correction disk. Common number for N is 4.
RAID Issues

• Writing small amounts of data can be very expensive
  – read old data
  – compute the change
  – write new data
  – compute new parity
  – write new parity

• For larger amounts of data, we only have the increased time to write and compute the parity
Virtual File System Interface

- File entries reference inodes
- Inodes are unique per file system
- What do we do when we network mount other FS
- Add a new layer - the vnode
- Change all of the interfaces in the system from inode to vnode
- vnode for a local file system refers to an inode
- vnode for a remote file system refers to a protocol control block with location and naming info
vnode

- Flags for locking and attributes like root of FS
- Various ref counts
- A pointer to the mount structure of the FS the vnode is in
- NFS info
- A pointer to the set of vnode operations defined
- Pointer to either the inode or nfsnode
- etc.
Pathname Translation

• We used to do namei, but that doesn’t work
• Vnode pathname translation
  – determine starting point as either root or cwd
  – vnode calls FS specific lookup() and passes path and current lookup dir
  – lookup returns a vnode and we repeat
• Key point - the underlying FS doesn’t know what is being used as a mount point
• Crossing a mount point means we switch FS
File Systems

- Standard Unix FS
- Memory FS (MFS)
- Union Mount FS
- Portal FS
- Network File Systems
  - RFS
  - NFS
  - AFS
Union Mount

Union Mount

```
/usr  /foo
  |
  src
  |
  x  bar  baz  x
    |   |
    a  b  c

```

```
  src
  |
  x
  |
  x  bar  baz
    |
    a  b  c
```
Network FS History

• Objective: share files across machines
  – Easiest model is to allow remote machines to mount FS as if they were local

• 3 Important Concerns
  – Semantics
  – Coherence
  – Performance

• Remote Disk
  – a scheme that allowed a machine to talk to another’s disk
Remote Disk vs NFS

Machine A

Remote Disk

NFS, AFS, RFS

Machine B
NFS Background

• Client/Server Protocol
• Client
  – imports the FS
• Server
  – exports local FS to other machines
• Protocol spec in the public domain
NFS Protocol

- Stateless
- Supports Unix FS Semantics but allows FS with less rich semantics, as well
- Access follows Unix, UID, group model but allows FS with weaker models, as well
- Protocol is transport independent
  - Designed for UDP
  - Ported to TCP and many other protocols
NFS Design Limitations

• Assumes clients & server are on a fast local net
  – Works badly on slow networks
  – Works badly for disconnected machines
• Caching model assumes files won’t be shared
  – If they are, the performance is lousy
• Stateless model loses some Unix FS semantics
  – flock
NFS Operations

- NFS Server is stateless
  - all requests are self contained
  - requests may be sent multiple times if things are slow
  - operations can be idempotent or not
  - for those that aren’t, the server keeps a cache of recent operations so as to avoid having problems with non idempotent ops
# NFS Operation List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETATTR</td>
<td>Get file attributes</td>
<td>yes</td>
</tr>
<tr>
<td>SETATTR</td>
<td>Set file attributes</td>
<td>yes</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>Look up file name</td>
<td>yes</td>
</tr>
<tr>
<td>READLINK</td>
<td>Read from a sym link</td>
<td>yes</td>
</tr>
<tr>
<td>READ</td>
<td>Read from a file</td>
<td>yes</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write to file</td>
<td>yes</td>
</tr>
<tr>
<td>CREATE</td>
<td>Create file</td>
<td>yes</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Remove file</td>
<td>no</td>
</tr>
<tr>
<td>RENAME</td>
<td>Rename file</td>
<td>no</td>
</tr>
<tr>
<td>LINK</td>
<td>Create link to a file</td>
<td>no</td>
</tr>
<tr>
<td>SYMLINK</td>
<td>Create a symbolic link</td>
<td>yes</td>
</tr>
<tr>
<td>MKDIR</td>
<td>Create directory</td>
<td>no</td>
</tr>
<tr>
<td>RMDIR</td>
<td>Remove directory</td>
<td>no</td>
</tr>
<tr>
<td>READDR</td>
<td>Read from a directory</td>
<td>yes</td>
</tr>
<tr>
<td>STATFS</td>
<td>Get FS attributes</td>
<td>yes</td>
</tr>
</tbody>
</table>
NFS Server

- Files are globally identified by a file handle
  - this is a token by which a client identifies a file to the server
  - handle is returned by lookup
  - handle is file system id, inode number, generation number of the inode
  - generation number insures that the inode still refers to the same file
  - file handle allows the server to find the file being referred to
NFS Protocol (more)

- Server is stateless
  - doesn’t maintain any info about which clients there are or which files they are accessing
  - RPC message has all info necessary to satisfy a request
  - In fact, the server will maintain a cache of recently accessed files so it isn’t really stateless
Stateless Server

• Can simply startup
  – no state recovery necessary
  – no need to worry about which clients are around
  – network partitions or failures followed by reconnects don’t cause problems
Drawbacks to Stateless Servers

• Local FS have state
  – When a file is unlinked, it is accessible until the last reference goes away
  – NFS can’t do this; it has to delete on the last unlink
  – Similar problems with FS advisory locking (flocks)

• Synchronous writes
  – Writes must be committed to stable store
Typical NFS Server

- `nfsd` forks children that reenter the kernel using `nfssvc` system call
  - they remain kernel resident
  - each `nfsd` receives messages, verifies the message and passes the request to a local FS
  - when the local FS finishes the request, it is returned to the `nfsd` which returns it to the client
Typical NFS Client

- nfsiod daemon forks children that reenter the kernel using nfssvc system call
  - they remain kernel resident
  - they do asynchronous read aheads and write behinds
  - nfsiod can buffer requests
- You don’t need this, but then all requests are synchronous
NFS Client Server Interaction

1. Client does a write() system call
2. Data copied to a kernel buffer on the client and write() returns
3. An nfsiod picks up the dirty buffer and sends the buffer to a server
4. The incoming request is given to an nfsd; the nfsd writes the data to local disk and waits for the i/o to complete
5. The nfsd sends an ack back to the client; the nfsiod gets the ack and marks the buffer as clean
NFS Complications

• Local FS are unaffected by network problems
  – they tend always to be available
• NFS Clients must be able to deal with the possibility that the remote file system is unavailable
• 3 kinds of NFS mounts
  – Hard mount - try forever and hang
  – Soft mount - try some number of times then return an error (most apps die at this point)
  – Interruptible mount - allow the operation to be interrupted