File handles

```
struct nfs_fh3 {
  opaque data<64>;
};
```

- **Server assigns an opaque file handle to each file**
  - Client obtains first file handle out-of-band (mount protocol)
  - File handle hard to guess – security enforced at mount time
  - Subsequent file handles obtained through lookups
- **File handle internally specifies file system / file**
  - Device number, i-number, *generation number*, ...
  - Generation number changes when inode recycled

File attributes

```
struct fattr3 {
  ftype3 type;
  uint32 mode;
  uint32 nlink;
  uint32 uid;
  uint32 gid;
  uint64 size;
  uint64 used;
  specdata3 rdev;
  uint64 fsid;
  uint64 fileid;
  nfstime3 atime;
  nfstime3 mtime;
  nfstime3 ctime;
};
```

- **Most operations can optionally return** `fattr3`
- **Attributes used for cache-consistency**

Lookup

```
struct diropargs3
  nfs_fh3 dir;
  filename3 name;
};
```

```
struct lookup3resok {
  nfs_fh3 object;
  post_op_attr obj_attributes;
  post_op_attr dir_attributes;
};
```

```
union lookup3res switch (nfsstat3 status) {
case NFS3_OK:
  lookup3resok resok;
default:
  post_op_attr resfail;
};
```

- **Maps** $\langle \text{directory}, \text{handle} \rangle \rightarrow \text{handle}$
  - Client walks hierarch one file at a time
  - No symlinks or file system boundaries crossed

Read

```
struct read3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
};
```

```
struct read3resok {
  post_op_attr file_attributes;
  uint32 count;
  bool eof;
  opaque data<>;
};
```

```
union read3res switch (nfsstat3 status) {
case NFS3_OK:
  read3resok resok;
default:
  post_op_attr resfail;
};
```

- **Offset explicitly specified (not implicit in handle)**
- **Client can cache result**

Data caching

- **Client can cache blocks of data read and written**
- **Consistency based on times in `fattr3`**
  - mtime: Time of last modification to file
  - ctime: Time of last change to inode
    (Changed by explicitly setting mtime, increasing size of file, changing permissions, etc.)
- **Algorithm: If mtime or ctime changed by another client, flush cached file blocks**

Write arguments

```
struct write3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
  stable_how stable;
  opaque data<>;
};
```

```
enum stable_how {
  UNSTABLE = 0,
  DATA_SYNC = 1,
  FILE_SYNC = 2
};
```

Write results

```
struct write3resok {
  wcc_data file_wcc;
  uint32 count;
  stable_how committed;
  writeverf3 verf;
};

union write3res switch (nfsstat3 status) {
case NFS3_OK:
  write3resok resok;
default:
  wcc_data resfail;
};
```

```
struct wcc_attr {
  uint64 size;
  nfstime3 mtime;
  nfstime3 ctime;
};
struct wcc_data {
  wcc_attr *before;
  post_op_attr after;
};
```

Data caching after a write

- **Write will change mtime/ctime of a file**
  - "`after`" will contain new times
  - Should cause cache to be flushed
- **"`before`" contains previous values**
  - If `before` matches cached values, no other client has changed file
  - Okay to update attributes without flushing data cache

Write stability

- **Server write must be at least as stable as requested**
- **If server returns write UNSTABLE**
  - Means permissions okay, enough free disk space, . . .
  - But data not on disk and might disappear (after crash)
- **If DATA_SYNC, data on disk, maybe not attributes**
- **If FILE_SYNC, operation complete and stable**

Commit operation

- **Client cannot discard any UNSTABLE write**
  - If server crashes, data will be lost
- **COMMIT RPC commits a range of a file to disk**
  - Invoked by client when client cleaning buffer cache
  - Invoked by client when user closes/flushes a file
- **How does client know if server crashed?**
  - Write and commit return `writeverf3`
  - Value changes after each server crash (may be boot time)
  - Client must resent all writes if verf value changes

Attribute caching

- **Close-to-open consistency**
  - It really sucks if writes not visible after a file close
    (Edit file, compile on another machine, get old version)
  - Nowadays, all NFS opens fetch attributes from server
- **Still, lots of other need for attributes (e.g., `ls -al`)**
- **Attributes cached between 5 and 60 seconds**
  - Files recently changed more likely to change again
  - Do weighted cache expiration based on age of file
- **Drawbacks:**
  - Must pay for round-trip to server on every file open
  - Can get stale info when statting a file

Alternatives caching strategy: Callbacks

- **Server maintains list of all clients caching info**
- **Calls back to each client when info changes**
- **Advantages**
  - Tight consistency
- **Disadvantages**
  - Server must maintain a lot of state
  - Updates potentially slow – must wait for n clients to acknowledge
  - When a client goes down, other clients will block

Leases

- **Hybrid mix of polling and callbacks**
  - Server agrees to notify client of changes for a limited period of time – the lease term
  - After the lease expires, client must poll for freshness
- **Avoids paying for a server round trip in many cases**
- **Server doesn't need to keep long-term track of callbacks**
  - E.g., lease time can be shorter than crash-reboot – no need to keep callbacks persistently
- **If client crashes, resume normal operation after lease expiration**

The different Unix contexts

- **User-level**
- **Kernel "top half"**
  - System call, page fault handler, kernel-only process, etc.
- **Software interrupt**
- **Device interrupt**
- **Timer interrupt (hardclock)**
- **Context switch code**

Transitions between contexts

- **User → top half: syscall, page fault**
- **User/top half → device/timer interrupt: hardware**
- **Top half → user/context switch: return**
- **Top half → context switch: sleep**
- **Context switch → user/top half**

Top/bottom half synchronization

- **Top half kernel procedures can mask interrupts**

```
int x = splhigh ();
/* ... */
splx (x);
```

- **splhigh disables all interrupts, but also splnet, splbio, splsoftnet, ...**
- **Masking interrupts in hardware can be expensive**
  - Optimistic implementation – set mask flag on splhigh, check interrupted flag on splx

Process scheduling

- **Goal: High throughput**
  - Minimize context switches to avoid wasting CPU, TLB misses, cache misses, even page faults.
- **Goal: Low latency**
  - People typing at editors want fast response
  - Network services can be latency-bound, not CPU-bound
- **BSD time quantum:** $1/10$ **sec (since** $\sim$**1980)**
  - Empirically longest tolerable latency
  - Computers now faster, but job queues also shorter

Multilevel feeedback queues (BSD)

- **Every runnable proc. on one of 32 run queues**
  - Kernel runs proc. on highest-priority non-empty queue
  - Round-robins among processes on same queue
- **Process priorities dynamically computed**
  - Processes moved between queues to reflect priority changes
  - If a proc. gets higher priority than running proc., run it
- **Idea: Favor interactive jobs that use less CPU**

Process priority

- `p_nice` – **user-settable weighting factor**
- `p_estcpu` – **per-process estimated CPU usage**
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second while process runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right) \texttt{p\_estcpu} + \texttt{p\_nice}$$

- **Run queue determined by** $\texttt{p\_usrpri}/4$

$$\texttt{p\_usrpri} \leftarrow 50 + \left(\frac{\texttt{p\_estcpu}}{4}\right) + 2 \cdot \texttt{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

- `p_estcpu` **not updated while asleep**
  - Instead `p_slptime` keeps count of sleep time
- **When process becomes runnable**

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right)^{\texttt{p\_slptime}} \times \texttt{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads

Discussion

- **10 people running vi have 1 sec latency?**
- **How do UNIX signals work?**
  - What if signal arrives while process in “top half”
- **Does UNIX kernel suffer from priority inversion?**