

Announcements

- Project #5 is due next Wed

Project #5 Notes

- Uid

- First process has uid of 0
- Spawned processes
 - Inherit uid of parent
 - Unless setuid bit is set on program to run, then the uid of the owner of that file is used

- ACLs

- First ACL entry is owner
- Others are for other users
 - Can delete these entire with `setACL(file, uid, 0)`
- Uid 0 can open any file regardless of ACLs

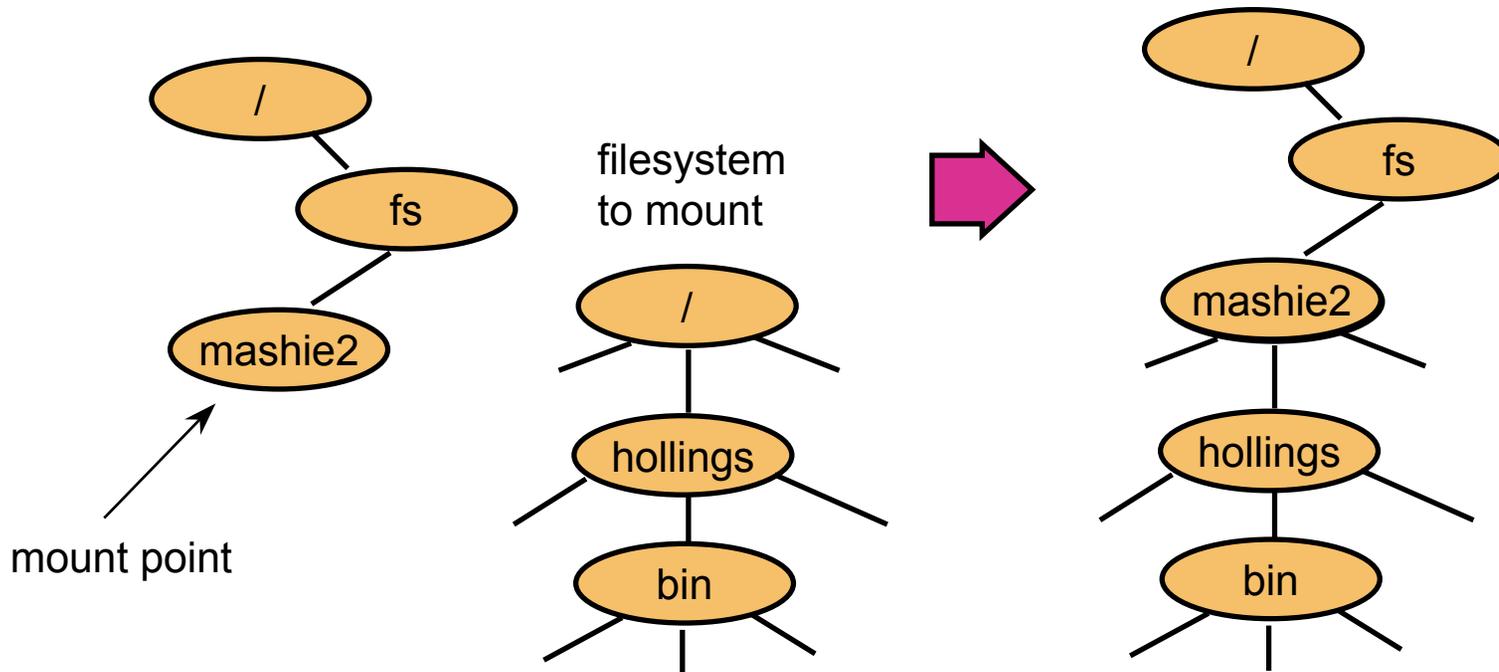
File Server State

- Does the fileserver maintain information between requests?
- Stateless
 - example: NFS
 - each request contains a request to read/write a specific part of a file
 - requests must be *idempotent*
 - the same request can be applied several times
 - makes recovery of failed clients/servers easier
- Stateful
 - example: AFS
 - servers maintain connections for clients
 - improves performance
 - required for server based cache management

Mounting a filesystem

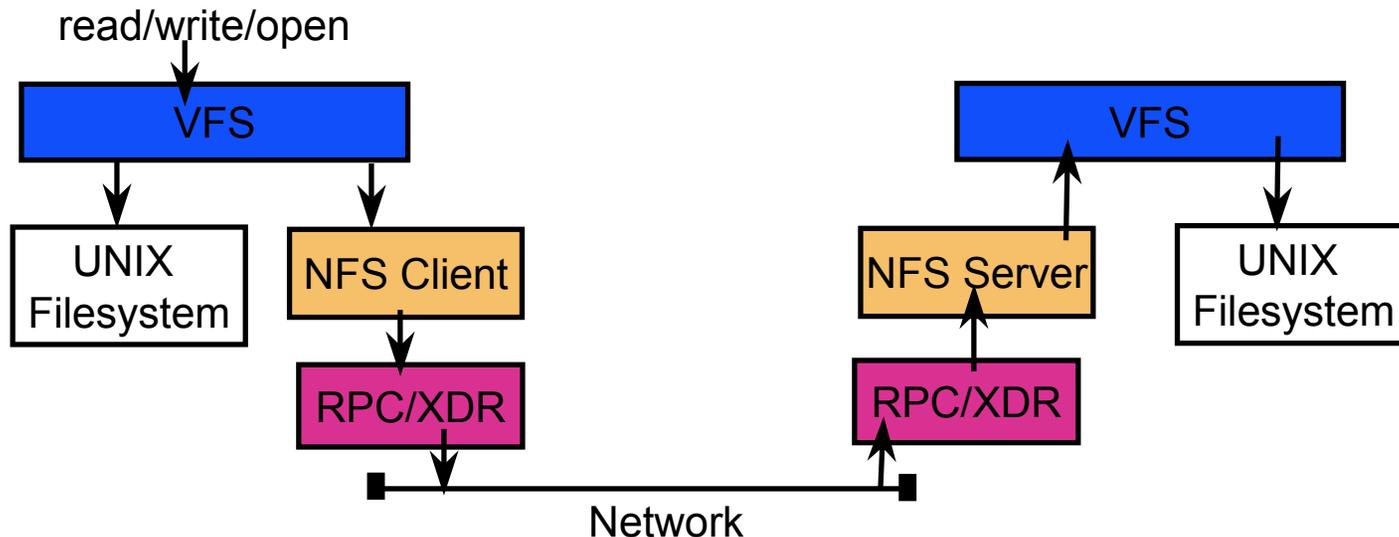
- Mount attaches a filesystem to a directory
 - can be used for local or remote (NFS) filesystems

Before Mount



NFS

- Provides a way to mount remote filesystems
 - can be done explicitly
 - can be done automatically (called an automounter)
 - clients are provided “file handle” by the server for future use
- Uses VFS: extended UNIX filesystem
 - inodes are replaced by vnodes
 - network wide unique inodes
 - can refer to local or remote files



NFS (cont.)

- Requests
 - are sent via RPC to the server
 - include read/write
 - query: lookup this directory info
 - must be done one step (directory) at a time
 - change meta data: file permissions, etc.
- Popular due to free implementations
- Provides no coherency

AFS

- Designed to scale to 5,000 or more workstations
- Location independent naming
 - within a single cell
- volumes
 - basic unit of management
 - can vary in size
 - can be migrated among servers
- names are mapped to “fids”
 - 96 bit unique id’s for a file
 - three parts: volume, vnode, and unique identifier
 - location information is stored in a volume to location DB
 - replicated on every server

AFS (cont.)

- **File Access**

- open: file is transferred from server to client
 - very large files may only be partially transferred
- read/write: performed on the client
- close: file (if dirty) is written back to server
 - can fail if the disk is full

- **Consistency**

- clients have callbacks
- sever informs client when another client writes data
- only applies to open operation
- only requires communication when:
 - more than one client wants to write
 - one client wants to write and others to read

Display and Window Management

- The screen is a resource in a workstation system
 - multiple processes desire to access the device and control it
 - OS needs to provide abstractions to permit the interaction
- Services
 - protection
 - windows
 - multiplex keyboard and mouse
 - configuration and placement
- Issues
 - how to get good performance and remain device independent
 - how much policy to dictate to users

My Research Interests

- **Parallel Computing**

- There are limits to how fast one processor can run
- solution: use more than one processor

- **Issues in parallel computing design**

- do the processors share memory?
 - is the memory “uniform”?
 - how do processors cache memory?
- if not how do they communicate?
 - message passing
 - what is the latency of message passing

Parallel Processing

- What happens in parallel?
- Several different processing steps
 - pipeline
 - simple example: `grep foo | sort > out`
 - called: *multiple instruction multiple data* (MIMD)
- The same operation
 - every processor runs the same instruction (or no-instruction)
 - called: *single instruction multiple data* (SIMD)
 - good for image processing
- The same program
 - every processor runs the same program, but not “lock step”
 - called: *single program multiple data* (SPMD)
 - most common model

Issues in effective Parallel Computation

- **Load balancing**
 - every processor should to have some work to do.
- **Latency hiding/avoidance**
 - getting data from other processors (or other disks) is slow
 - need to either:
 - hide the latency
 - processes can “pre-fetch” data before they need it
 - block and do something else while waiting
 - avoid the latency
 - use local memory (or cache)
 - use local disk (of file buffer cache)
- **Limit communication bandwidth**
 - use local data
 - use “near” data (i.e. neighbors)

My Research:

- Given a parallel program and a machine
- Try to answer performance related questions
 - Why is the programming running so slowly?
 - How do I fix it?
- Issues:
 - how to measure a program without changing it?
 - how do you find (and then present) the performance problem, not tons of statistics?
- Techniques:
 - dynamic data collection
 - automated search
 - analysis of process interactions