Properties of the I/O Subsystem

- Anonymizes (unifies the interface to) hardware devices, both to users and to other parts of the kernel
- General device-driver code
- Hardware-specific device drivers

Three Main Kinds of I/O

- Filesystem
  - Actual files (although this is too restrictive)
  - Hierarchical name space, locking, quotas, attribute management, protection
  - More on this later (Ch. 8)
- Socket interface
  - Networking requests
Three Types of I/O (II)

- Character-device interface
  - Unstructured access to devices
  - Two types
    - Actual character devices (e.g. terminals)
    - “raw” access to block devices (e.g. disks, tapes)
      - I/O doesn’t use the filesystem or page cache
      - Requests are still in multiples of block size
      - Requests might need to be aligned

Kernel I/O Structure (pg. 216)

Internal I/O Structure

- Device drivers are modules of code dedicated to handling I/O for a particular piece or kind of hardware
- Drivers define a set of operations via well-known entry points (e.g., read, write, etc.)
- Character device interface is kept in cdevsw structure.
Character Device Switch Table

```c
struct cdevsw {
    int          d_maj;
    u_int        d_flags;
    const char   *d_name;
    d_open_t     *d_open;
    d_fdopen_t   *d_fdopen;
    d_close_t    *d_close;
    d_read_t     *d_read;
    d_write_t    *d_write;
    d_ioctl_t    *d_ioctl;
    d_poll_t     *d_poll;
    d_mmap_t     *d_mmap;
    d_strategy_t *d_strategy;
    dumper_t     *d_dump;
    d_kqfilter_t *d_kqfilter;
};
```

Device Numbers in 4.4BSD

- Historically used to select the device driver and actual device
- Device number was a (major, minor) pair
- Major number selected the device driver
- Minor number selects which (of possibly many) devices—interpreted by the driver

Device Numbers in FreeBSD 5.2

- Not used to select device driver
  - Present only for applications to examine
- Direct link in /dev file system to the device’s cdevsw entry
- On configuration:
  - New major device number assigned
  - Entry made in /dev
  - Minor device # still has meaning
Device Drivers

- Autoconfiguration & Initialization
  - Probes device, sets initial software state
  - Called once (system startup or device connection)
- Top Half
  - System calls or VM system
  - may call sleep()

Device Drivers (II)

- Bottom Half (Interrupt handlers)
  - no per-process state
  - does have kernel thread context, so can block (subject to locking restrictions)
  - Blocking in interrupt handlers probably undesirable (performance reasons)
- Error handling (optional)
  - Crash-dump routine, called by system when an unrecoverable error occurs.

I/O Queuing

- How data moves between the top and bottom halves of the kernel
- Typical top half sequence
  - receives request from user through system call
  - records request in data structure
  - places data structure in queue
  - starts device if necessary
Interrupt Handlers

- Device interrupts after completing the I/O operation
- Removes request from device queue
- Notifies requester that command has completed (how?)
- Starts next request from the queue

Top/Bottom Coordination

- What might happen if an interrupt occurred while a request was being enqueued in the top half?
- The queue is just like any other shared data structure
- To enforce synchronization, use mutexes.

Interrupt Mechanism

- Device interrupts
- Call glue routine through interrupt table (glue routine allows ISR to be machine independent)
- Glue routine saves volatile registers
- ... updates statistics
- ... calls ISR passing in device unit number
- ... restores volatile registers
- ... returns from interrupt
Character Devices

- Most peripherals (except networks) have character interfaces
  - For many devices, this is just a byte stream
    - /dev/lp0 (printer)
    - /dev/tty0 (terminal)
    - /dev/mem (memory)
    - /dev/null (sink/source of EOF)

Non-Byte-Stream Devices

- E.g., high-speed graphic interfaces
- Maintain own buffers -or-
- Do I/O directly to address space of user
  - User-level buffer must be wired during I/O

Disk-like Character Devices

- For “block” devices, the character interface is called the “raw” interface
  - Direct access to the device
  - I/O directly to user space
  - User-level buffer must be wired during I/O
  - Not a byte-stream model; still have to talk in terms understood by underlying device
    - Size, alignment
  - Should only be used by programs that have an intimate knowledge of the underlying device and format; otherwise, chaos may ensue
Character Device Interface

- Open, close: just what you think
- Read, write: call physio() to do the transfer
- ioctl: the "swiss army knife" call; highly-dependent on device
- Poll: see whether the device is ready for reading/writing (cf. select() syscall)
- Stop: flow control on terminal devices
- Mmap: map device memory into address space
- Reset: re-initialize the device state

Disk Driver Entry Points

- Strategy: start read/write op and return immediately.
  - A buf structure is passed, containing parameters for the I/O request
  - If synchronous, caller sleeps on address of buf structure
- Dump: write physical memory to the device (used for controlled system crashes)

Disk Scheduling: dissort()

- Implements a variant of the elevator algorithm: which one is it?
- Keeps two sorted lists of requests:
  - The current one (the one we're servicing)
  - The other one (for the opposite direction)
  - Switch whenever we hit the end
- Modern disk controllers will allow multiple outstanding requests and will sort the requests themselves
Pseudocode for disksort()

Disksort(drive queue *dq, buffer *bp)
{
    if (active list is empty) {
        place the buffer at the front of the active list;
        return;
    }
    if (request lies before the first active request) {
        locate the beginning of the next-pass list;
        sort bp into the next-pass list;
    } else {
        sort bp into the active list;
    }
}

Disksort() and Modern Disks

- Most modern disks can handle several simultaneous requests
  - Sort them internally
- However, disks typically only buffer about 15 requests
  - During busy times, > 15 requests simultaneously
- Need to maintain longer list in kernel

Disk Labels

- The geometry of a disk is the number of cylinders/tracks/sectors on a disk, and the size of the sectors
- Disks can have different geometries imposed by a low-level format
- Disk labels are written in a well-known location on the disk, and contain the geometry, as well as information about partition usage.
Bootstrapping

- The first level bootstrap loader reads the first few disk sectors to find
  - The geometry of the disk
  - A second-level bootstrap routine
    - Think of lilo, multiboot, et al.
- See, for example, http://www.nomoa.com/bsd/dualboot.htm

File Descriptors

- System calls use file descriptors to refer to open files and devices
  - File entry contains file type and pointer to underlying object
    - Socket (IPC)
    - Vnode (device, file)
    - Virtual memory

Open File Entries

- "object-oriented" data structure
- Methods:
  - Read
  - Write
  - Poll
  - Ioctl
  - Stat
  - Check for kqueue events
  - Close/deallocate
- Really just a type and a table of function pointers
Open File Entries II

- Each file entry has a pointer to a data structure describing the file object
  - Reference passed to each function call operating on the file
  - All object state is stored in the data structure
  - Opaque to the routines that manipulate the file entry—only interpreted by code for underlying operations
- File offset determines position in file for next read or write
  - Not stored in per-object data structure. Why not?

File Descriptor Flags

- Protection flags (R, W, RW)
  - Checked before system call executed
  - System calls themselves don’t have to check
- O_NONBLOCK: return EAGAIN if the call would block the thread
- O_ASYNC: send SIGIO to the process when I/O becomes possible
- O_APPEND: set offset to EOF on each write
- O_FSYNC: Writes to file sync’ed with disk
- O_DIRECT: Ask kernel to skip copy
- Close on exec: stop sharing with children on exec (saved with desc.)
- Locking information (shared/exclusive)

File Entry Structure Counts

- Reference Counts
  - Count the number of times processes point to the file structure
    - A single process can have multiple references to (i.e., descriptors for) the file (dup, fcntl)
    - Processes inherit file entries on fork (shared offset)
- Message Counts
  - Descriptors can be sent in messages to local procs
    - Increment message count when descriptor sent, and decrement when received
File Structure Manipulation: 
fcntl()

- Duplicate a descriptor (same as dup())
- Get/set flags enumerated on earlier slide
- Send signal to process when urgent data arises
- Set or get the PID or PGID to which to send the two signals
- Test or set status of a lock on a range of bytes in the underlying file

File Descriptor Locking

- Early Unix didn’t have file locking
  - So, it used an atomic operation on the file system to simulate it. How?
- Downsides of this approach:
  - Processes consumed CPU time looping
  - Locks left lying around have to be cleaned up by hand
  - Root processes would override the lock mechanism
- Real locks were added to 4.2 BSD

File Locking Options: 
Lock the Whole File

- Serializes access to the file
- Fast (can just record the PID/PGRP of the locking process and proceed)
- Inherited by children
- Released when last descriptor for a file is closed
- When does it work well? When poorly?
- 4.2 BSD and 4.3 BSD had this
File Locking Options:
Byte-Range Locks

- Can lock a piece of a file (down to a byte)
- Not powerful enough to do nested hierarchical locks (databases)
- Complex implementation
- Mandated by POSIX standard, so put into 4.4 BSD
- Released each time a close is done on a descriptor referencing the file

Problem with Byte-Range Locks (mandated by POSIX)

- Inadvertant release
  - Process locks file
  - Calls library routine
    - Open
    - Read
    - Close
    - Lock was just released!
- Must have file open for reading to exclusively lock

File Locking Options:
Advisory vs. Mandatory

- Mandatory:
  - Locks enforced for all processes
  - Cannot be bypassed (even by root)
- Advisory:
  - Locks enforced only for processes that request them
**File Locking Options:**

**Shared vs. Exclusive**

- Shared locks: multiple processes may access the file
- Exclusive locks: only one process
- Request a lock when another process has an exclusive lock => you block
- Request an exclusive lock when another process holds a lock => you block
- Think of the classic readers-writers problem

**FreeBSD Locks**

- Both whole-file and byte-range
  - Whole-file implemented as byte-range over the entire file, applied to descriptor
  - Byte-range lock applied to file
  - Closing behavior handled as special case check
- Advisory locks
- Both shared and exclusive may be requested
- May request lock when opening a file (avoid race condition)
- Done on per-filesystem basis
- Process may specify to return with error instead of blocking if lock is busy

**Doing I/O for Multiple Descriptors**

- Consider the X Server
  - Read from keyboard
  - Read from mouse
  - Write to frame buffer (video)
  - Read from multiple network connections
  - Write to multiple network connections
- If we try to read from a descriptor that isn’t ready, we’ll block
Historical Solution

- Use multiple processes communicating through shared facilities
- Example:
  - Have a “master” process fork off a separate process to handle each I/O device or client for the X Server
  - For N children, N+1 pipes:  
    - One for all children to talk to X Server
    - One per child for the X Server to talk back
- High overhead in context switches

FreeBSD’s Solution: Three I/O Options

- Polling I/O:
  - Repeatedly check a set of descriptors for available I/O; see discussion of Select
- Nonblocking I/O:
  - Complete immediately w/partial results
- Signal-driven I/O:
  - Notify process when I/O becomes possible

Possibilities to Avoid the Blocking Problem (1-2)

- Set all descriptors to nonblocking mode
  - This ends up being a different form of polling (busy waiting)
- Set all descriptors to send signals
  - Signals are expensive to catch; not good for a high I/O process like the X Server
Possibilities to Avoid the Blocking Problem (3-4)

- Have a query mechanism to find out which descriptors are ready
  - Block process if none are ready
  - Two system calls: one for check, one for I/O
- Have the process give the kernel a set of descriptors to read from, and be told on return which one provided the data
  - Can’t mix reads and writes

Which Possibilities are in FreeBSD?

- Possibility #1: nonblocking I/O
- Possibility #2: signal-driven I/O
- Possibility #3: select() or poll()
  - Two different interfaces to the same underlying function.
  - We’ll focus on select()
- Possibility #4: not in FreeBSD

Select()

- Process passes in three bitmaps of descriptors it’s interested in: read, write, and exceptions
- Kernel returns as soon as any of those are ready for I/O
- Process decides which descriptor to do I/O on (bitmap operations)
- Process can set timeout, and kernel will return with no bits set at that time (if it hasn’t already)
Device Polling: Low-level select() Code for FreeBSD

```c
struct selinfo {
    struct thread *si_thread; /* thread to be notified */
    short si_flags; /* SI_COLL - collision occurred */
}
```

Routine called with POLLIN, POLLOUT, or POLLRDBAND

```c
if (events & POLLIN) {
    if (nread > 0 || (tp->t_state & TS_CARR_ON) == 0) {
        revents |= POLLIN;
    } else {
        selrecord(curthread, &tp->t_rsel);
    }
}
```

Return success if operation is possible (success only means won’t block)

Record tid with the device

FreeBSD 5.2 select()

- Look in sys-generic.c at select() to see how FreeBSD 5.2 has implemented this

Awakening a Process

- Device/socket notices change in status
- selwakeup() is called pointing to the selinfo struct, and collision flag
- If the thread is sleeping on selwait, wake it up
- If thread has its selecting flag set, clear it (cf. select() code and retry)
- If collision occurred, wake up everyone who is sleeping on selwait
Kernel Data Movement: I/O Vectors (iovec)

- iovec is a base address and a length

```c
struct iovec {
    char    *iov_base; /* Base address. */
    size_t  iov_len;  /* Length. */
};
```

- Same as used in readv() and writev() system calls.

The uio Structure: Describing the I/O Operation

- A pointer to an array of iovec
- Number of elements in iovec array
- File offset for start of operation
- Flag showing whether source and destination are both in kernel, or between user and kernel
- Sum of the length of the I/O vectors
- Flag whether data is going from uio ⇒ kernel (UIO_WRITE) or kernel ⇒ uio (UIO_READ)
- Pointer to thread whose area is described by the uio structure (NULL for kernel area)
Use of iovec and uio

- All I/O in the kernel is described by uio and iovec structures.
- System calls that don’t take an iovec create a uio structure to hold data, and pass that down in the kernel.
- Lower levels of kernel use uiomove() to copy data between uio structures and kernel buffers.

The Virtual Filesystem Interface

- In older systems, file entries pointed directly to a filesystem index node (inode).
  - An index node describes the layout of a file on disk.
- However, this tied file entries to the particular filesystem (the Berkeley Fast File System, or FFS).
- Add a layer to sit between the file entries and the inodes: the virtual node (vnode).

Vnodes

- An object-oriented interface
- Vnodes for local filesystems map onto inodes
- Vnodes for remote filesystem map to network protocol control blocks describing how to find and access the file.
### Vnode Entries

- Flags for locking and generic attributes (e.g. filesystem root)
- Reference counts
- Pointer to mount structure for filesystem containing the vnode
- File read-ahead information
- A vm_object reference
- Reference to state of special devices, sockets, FIFOs

### Vnode Entries (II)

- Mutex to protect flags/counters
- Lock manager lock to protect portions of vnode changed by I/O operations
- Fields used by name cache
- Pointer to vnode operations
- Pointer to private information (inode, nsfnode)
- Type of underlying object
- Clean and dirty buffer pools
- Count of write operations in progress

### Vnode Operations

- At boot time, each filesystem registers the set of vnode operations it supports.
- Kernel builds a table listing the union of all operations on any filesystem, and an operations vector for each filesystem
- Supported operations point to routines; unsupported point to a default routine.
Filesystem Operations

- File systems provide two types of operations:
  - Name space operations
  - On-disk file storage
- 4.3 BSD had these tied together; 4.4BSD and FreeBSD 5.2 separate them

Pathname Translation

- The pathname is copied in from the user process (or extracted from network buffer for remote filesystem request)
- Starting point of the pathname is determined (root for absolute path names, current directory for relative names)—this is the lookup directory in the next step

Pathname Translation II

- Vnode layer calls filesystem-specific lookup() operation, passing in the remaining pathname and the lookup directory
- Filesystem code searches the lookup directory for the next component of the path, and returns either the vnode for that component or an error
Pathname Translation III: Possible Outcomes

- Error returned: propagate it
- Pathname not exhausted, but current vnode is not a directory: "not a directory" error
- Pathname exhausted, then the current vnode is the result; return it
- If vnode is the mount point for another file system, make the lookup directory the mounted filesystem; else make the lookup directory the returned vnode
  - In either case, iteratively call lookup() with the new lookup directory

Common Vnode (Exported) Filesystem Services

- Generic mount options
  - Noexec: don't execute any files found on this file system (useful in heterogeneous networked environments)
  - Nosuid: don't honor setuid or setgid flags for executables on this filesystem (security)
  - Nodev: don't allow any special devices on this filesystem to be opened (again, heterogeneous systems)
  - Noatime: don't update file access times on reads (performance improvement)
  - Sync: request I/O to the FS be done synchronously

Common Vnode Services II: Informational services

- statfs(): returns information about the filesystem resources (number of used and free disk blocks and inodes, filesystem mount point, and source of the mounted filesystem)
- getfsstat(): returns information about all mounted filesystems
Filesystem-Independent Services: Inactive()

- Called by the vnode layer when last reference to a file is closed, usage count goes to 0
- Filesystem can then write dirty blocks, etc.
- Vnode is placed on system-wide free list
- Floating free list allows vnodes to migrate to the file system where they’re needed

Filesystem-Independent Services: reclaim() & vgone()

- reclaim() disassociates the vnode with the underlying file system
- vgone() uses reclaim and then associates the vnode with the dead filesystem
  - All calls except close() generate an error
  - Used to disassociate a terminal when the session leader exits, or when unmounting a filesystem where processes have open files

Name Cache

- Associate a name with its vnode to save lookups in the file system
  - Add name and vnode to cache
  - Look up vnode by name
  - Delete name from the cache
  - Invalidate all names that reference a vnode
    - Vnode has list of names in the name cache
    - Directory vnode also has list of cache entries contained within it
Negative Caching

- Names of files that aren’t there are looked up in the cache, miss, then also not found in the filesystem—expensive!
  - Pathname searching in the shell
  - Alternative: put an entry in the cache with that name, and a null vnode pointer.
  - This entry will be found on next lookup, short-circuiting the process
  - If name added to directory, lookup in name cache and purge negative entry if present

Buffer Management: the Buffer Cache

- In older BSD (pre-mmap), there were two caches of memory backed on disk
  - Disk blocks for explicit file access
  - Pages for VM
- What if a file is opened by process A (traditional), while process B has it mmaped?
  - Two caches means a big, hairy mess
  - A should see changes B makes

Merging the Caches

- Buffer cache and VM cache are now a single page cache
- VM: named files (backed by file) identified by vnode and logical block #
- Old buffer-caching interface reimplemented as emulation layer over VM cache
Buffer Cache Emulation

- Same interface as old routines
- Looks up requested file pages in VM cache
  - If not in memory, VM system reads it in
  - Map into kernel VM space (so there is reference to it)

The Buffer Cache

- Remember that the buffer cache speeds up access to disk blocks
  - 85% of transfers can be skipped
- Buffer format
  - Header
  - Content
  - Stored separately (allows buffers to be on page boundaries, for efficient transfer and manipulation)
  - Buffer data can be from 512 up to 64k bytes

Buffer Format

- hash link
- free-list link
- flags
- vnode pointer
- file offset
- byte count
- buffer size
- buffer pointer
- buffer header
- buffer contents
- (64 Kbyte) MAXBSIZE
Buffer Management III

- Each buffer initially gets 4096 bytes (multiple pages)
  - May give up part of its physical memory later
- Buffers identified by logical block # in file
  - Used to be done by physical block #, but
    - Didn't always have access to physical device
    - Slow lookup through indirect blocks in filesystem to find physical number
  - To avoid aliasing, kernel prohibits mounting a partition while its block device is open (and vice versa)

Buffer Calls

- bread(): takes a vnode, logical block number, and size, and returns pointer to locked buffer
  - Further accesses to this buffer by other processes cause them to sleep
  - Four ways for a node to be unlocked
    - brelse(), bdwrite(), bawrite(), bwrite()

Buffer Calls II

- brelse(): release an unmodified buffer; awaken any threads sleeping on the buffer
- bdwrite(): mark the buffer as dirty and put it back in the free list, and awaken processes waiting for it
- bawrite(): schedule a write on a full buffer, and return (asynchronous; implicit brelse())
- bwrite(): ensure write is complete before returning (implicit brelse())
- bqrelse(): used by filesystem to indicate buffer will be used again soon; don't dissolve
Buffer Lists

- Bufhash: quick lookup for buffers with valid contents (exactly one)
- Four "free" lists:
  - LOCKED: buffers can’t be flushed (used to be superblock, now only used to hold buffers being written out in background)
  - DIRTY: an LRU cache of used buffers, waiting to be written to disk
  - CLEAN: unmodified buffers FS will want soon
  - EMPTY: buffers with no physical memory

Buffer Pool

Buffer Management Implementation

- bread() called with request for a datablock of specified size for a specified vnode
- breadn() gets block and starts readahead
Buffer Management Implementation II

- `getblk()` finds out if the data block is already in a buffer
  - If in a buffer, call `bremfree()` to take it off of whatever free list it's on and mark it busy
  - If not in buffer, call `getnewbuf()`, then `allocbuf()`
- `getnewbuf()` allocates a new buffer
- `allocbuf()` makes sure buffer has physical memory
- `bread()` then passes the buffer to the strategy routine for the underlying filesystem.

Buffer Allocation

- `allocbuf()` ensures that the buffer has enough physical memory
- If there is excess physical memory, take some of it and give it to a buffer on the EMPTY list (move it to CLEAN)—keep if none there
- If not enough, call `getnewbuf()` and then steal its memory (putting it on the EMPTY list if you steal all of its memory)
- Keeps each disk block mapped into at most one buffer at any time

Overlapping Buffers

- The kernel must ensure that disk blocks appear in at most one buffer
- If multiple dirty buffers held portions of the same block, the kernel wouldn't know which one was more recent
- Kernel purges buffers when files are shortened or removed
Vnode Evolution

- Vnodes were originally only used to provide an OO-interface to the underlying filesystem
- Concept then generalized to allow vnodes to point to other vnodes (see http://citeseer.nj.nec.com/rosenthal90evolving.html and http://citeseer.nj.nec.com/heidemann94file.html)

Stackable Filesystem

- With vnodes being able to point to other vnodes, we can treat filesystems as layers, and stack them
- So, we can add features (such as transparent encryption) to an underlying filesystem without changing that filesystem at all

Mounting Layers

- Mount introduces an alias for one filesystem in another
- Traditional mount: map device into a filesystem, overlaying the old directory with a file system
- With layers, mount pushes a new layer on a vnode stack
  - Can overlay old vnode (use same name in file space)
  - Can appear separately (use different name)
Layered Filesystems

- File access is dependent upon which name is used for the file
  - New name: use new layer
  - Old name: use old layer
  - This assumes new mounts are at different points from old ones

Vnode Options on File Access

- Perform requested operation and return result
- Pass operation unchanged to next-lower vnode; might (or might not) modify result
- Modify operands and pass on to next-lower vnode; might (or might not) modify result
- Operation passed to the bottom of the stack without action generates an error

Old vs. New Vnodes

- Old vnodes
  - Vnode operations implemented as indirect function calls
- New vnodes
  - Intermediate layers pass operation to lower layers
  - New operations added into the system at boot time
  - Pass operations that may not have been defined when the filesystem was written
  - Pass parameters of unknown type and number
Solution

- Kernel places vnode operation name and arguments into a struct
- That struct passed as single parameter to vnode operation (uniform interface)
- If the operation is supported, the vnode knows how to decode the argument
- If it’s not supported, then it’s passed to the next lower layer w/same argument

Structure (for access() call)

```
struct vop_access_args {
    struct vnodeop_desc *a_desc;
    struct vnode *a_vp;
    int a_mode;
    struct u_cred *a_cred;
    struct thread *a_td;
};
```

- The first element of every argument structure is a vnodeop_desc structure.

The Null File System: nullfs

- This would be better called the "identity file system"
- Just passes all requests and responses without changing them at all
- Can make a filesystem appear within itself
- Useful as starting point for implementing other file systems
User Mapping File System: umapfs

- umapfs mounts the same file system in two places (/local into /export) on the file server
- /export is exported to external domains that use different UID/GID.
- Mapping is automatically done by the umapfs layer
- Request handed to /local

Benefits of umapfs

- No cost to local clients (they don't go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously

Outside administrative exports

- NFS server
- Passport
- UFS
- /local
- umapfs
- root
- /export
- NFS server
- /local
- umapfs
- root
- /export
- Operation not supported

Benefits of umapfs

- No cost to local clients (they don’t go through the umapfs)
- No changes required to NFS or local filesystem code
- We can expand this idea to give each outside domain its own mapping, accessed through its own mount point

Union Mount Filesystem: unionfs

- Traditional mount hides any files in the directory before the mount
- Union filesystem does a transparent overlay (you can still see the old)
- Both can be accessed simultaneously
More unionfs

- Start with an empty top-level directory
- As the underlying fs is traversed, entries are made in the top level
- Lower levels are read-only
- Any names appearing in top fs obscure lower fs

```
/usr/src  /tmp/src  /usr/src
src--------src--------src
lib          lib

Makefile  sh.h  sh.c  sh
```

Still More unionfs

- Two conflicting requirements:
  - Lower layers are read-only
  - User should think this is a regular file system
- We already copy files up on writes, but how do we handle deletions?
  - Deletions of unmodified files (lower layer)
  - Deletions of previously modified files (top layer)
- Use "whiteout" entries in the top level: inode 1
- On hitting a whiteout entry, kernel returns error.

Tricky Issues With Whiteout: Directories

- What happens if I create a file with the same name as a whiteout file?
- What about a directory? Is there any difference?
- Compare what happens in the "normal" situation where the same directory appears in multiple layers of the filesystem stack
Uses of the unionfs

- Heterogeneous architectures can build from a common source base
  - NFS mount source tree
  - Union mount over the top
  - Builds stay local
- Can compile sources “on” read-only media (e.g., CD-ROM)
- Private source directories from common code bases

Other Common Filesystems: Portal

- Mount a process into the filesystem
- Pass remainder of path name to process as input
- Process returns file descriptor
  - Could be socket to process
  - Could be to file
- Can be used, e.g., to provide filesystem access to network services (opening file creates TCP connection to server)

Other Common Filesystems: procfs

- procfs provides information on running processes, through the file system (viz ps)
- An ls of /proc shows a directory for each process, which contains:
  - ctl: file allowing control of the process
  - file: the executable (compiled code)
  - mem: virtual memory for the process
  - regs: registers for the process
  - status: text file with information (e.g. state, ppid, …)
Small Group Discussion

Consider the design of, and explain how you would build, a vnode layer to automatically encrypt files for a user. Briefly address the following issues:

- Encryption should be invisible to the users, but all files should be encrypted.
- Files for multiple users are stored on the same underlying file system.
- How will you ensure that only the qualified user can see the unencrypted files?
- How/when will a user authenticate itself?
- How can users share files?
- What happens if a user program does a seek on a file in the top layer (are there issues with file offsets)?