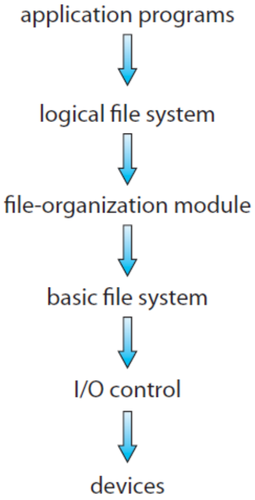
File System Implementation (Part II)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Reminder

File System Layers



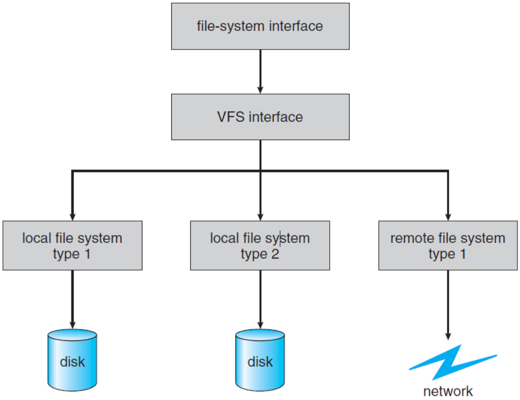
File-System Implementation

- Based on several on-disk and in-memory structures.

- On-disk
  - Boot control block (per volume)
  - Volume control block (per volume)
  - Directory structure (per file system)
  - File control block (per file)

- In-memory
  - Mount table
  - Directory structure
  - The open-file table (system-wide and per process)
  - Buffers of the file-system blocks

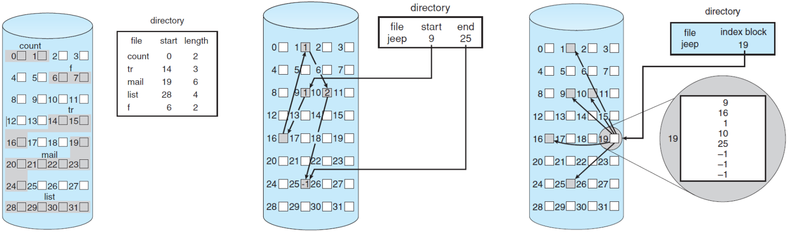
Virtual File Systems (VFS)



Directory Implementation

- Linear list
- Hash table

Allocation Methods



Free Space Management

Free-Space Management (1/2)

- File system maintains free-space list to track available blocks.

Free-Space Management (1/2)

- File system maintains free-space list to track available blocks.
- To create a file, OS searches the free-space list for the required amount of space and allocates that space to the new file.
  - This space is then removed from the free-space list.

Free-Space Management (1/2)

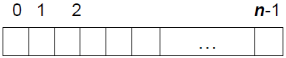
- File system maintains free-space list to track available blocks.
- To create a file, OS searches the free-space list for the required amount of space and allocates that space to the new file.
  - This space is then removed from the free-space list.
- When a file is deleted, its disk space is added to the free-space list.

Free-Space Management (2/2)

- Possible techniques:
  - Bit vector
  - Linked list
  - Grouping
  - Counting
  - Space maps

Bit Vector

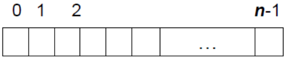
- Bit vector or bit map (n blocks)



- `bit[i] = 1`: block *i* is free
  `bit[i] = 0`: block *i* is occupied

Bit Vector

- Bit vector or bit map (n blocks)

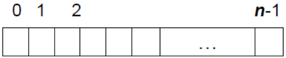


- `bit[i] = 1`: block i is free
  `bit[i] = 0`: block i is occupied

- Block number calculation to find the location of the first free block:
  (# of bits per word) × (# of 0-value words) + offset of first 1 bit

Bit Vector

- Bit vector or bit map (n blocks)



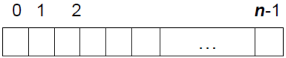
- `bit[i] = 1`: block i is free
  `bit[i] = 0`: block i is occupied

- Block number calculation to find the location of the first free block:
  (# of bits per word) × (# of 0-value words) + offset of first 1 bit

- Inefficient unless the entire vector is kept in main memory.

Bit Vector

- Bit vector or bit map (n blocks)



- `bit[i] = 1`: block i is free
  `bit[i] = 0`: block i is occupied

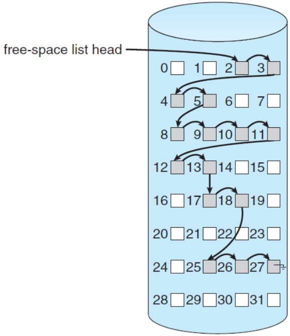
- Block number calculation to find the location of the first free block:
  (# of bits per word) $\times$ (# of 0-value words) + offset of first 1 bit

- Inefficient unless the entire vector is kept in main memory.

- Easy to get contiguous files.

Linked List

- Linked list (free-list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list



Grouping

- A modified version of the linked list approach

Grouping

- A modified version of the linked list approach
- It stores the addresses of n free blocks in the first free block.

Grouping

- A modified version of the linked list approach
- It stores the addresses of n free blocks in the first free block.
- The first $n - 1$ of these blocks are actually free.

Grouping

- A modified version of the linked list approach
- It stores the addresses of n free blocks in the first free block.
- The first $n - 1$ of these blocks are actually free.
- The last block contains the addresses of another n free blocks, and so on.

Grouping

- A modified version of the linked list approach
- It stores the addresses of n free blocks in the first free block.
- The first $n - 1$ of these blocks are actually free.
- The last block contains the addresses of another n free blocks, and so on.
- Easy to get contiguous files.

Counting

- Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering.

Counting

- Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering.
- Keep address of first free block and count of following free blocks.

Counting

- Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering.
- Keep address of first free block and count of following free blocks.
- Free space list then has entries containing addresses and counts.

Space Map

- Used in ZFS

Space Map

- Used in ZFS
- Consider meta-data I/O on very large file systems: full data structures like bit maps cannot fit in memory

Space Map

- Used in ZFS
- Consider meta-data I/O on very large file systems: full data structures like bit maps cannot fit in memory
- Divides device space into metaslab units and manages metaslabs.
  - A volume can contain hundreds of metaslabs.

Space Map

- Used in ZFS
- Consider meta-data I/O on very large file systems: full data structures like bit maps cannot fit in memory
- Divides device space into metaslab units and manages metaslabs.
  - A volume can contain hundreds of metaslabs.
- Each metaslab has associated space map: uses counting algorithm

Space Map

- Used in ZFS
- Consider meta-data I/O on very large file systems: full data structures like bit maps cannot fit in memory
- Divides device space into metaslab units and manages metaslabs.
  - A volume can contain hundreds of metaslabs.
- Each metaslab has associated space map: uses counting algorithm
- Rather than write counting structures to disk, it uses log-structured file-system techniques to record them.

Space Map

- Used in ZFS
- Consider meta-data I/O on very large file systems: full data structures like bit maps cannot fit in memory
- Divides device space into metaslab units and manages metaslabs.
  - A volume can contain hundreds of metaslabs.
- Each metaslab has associated space map: uses counting algorithm
- Rather than write counting structures to disk, it uses log-structured file-system techniques to record them.
- Metaslab activity → load space map into memory in balanced-tree structure

Efficiency and Performance

Efficiency and Performance

- Disks are the major bottleneck in system performance.
- A variety of techniques used to improve the efficiency and performance of secondary storage.

Efficiency (1/2)

- Efficiency dependent on disk allocation and directory algorithms.

Efficiency (1/2)

- Efficiency dependent on disk allocation and directory algorithms.
- Pre-allocation or as-needed allocation of metadata structures.

Efficiency (1/2)

- Efficiency dependent on disk allocation and directory algorithms.
- Pre-allocation or as-needed allocation of metadata structures.
  - E.g., Unix inodes are pre-allocated on a volume.

Efficiency (1/2)

- Efficiency dependent on disk allocation and directory algorithms.
- Pre-allocation or as-needed allocation of metadata structures.
  - E.g., Unix inodes are pre-allocated on a volume.
  - Even an empty disk has a percentage of its space lost to inodes.

Efficiency (1/2)

- Efficiency dependent on disk allocation and directory algorithms.

- Pre-allocation or as-needed allocation of metadata structures.
  - E.g., Unix inodes are pre-allocated on a volume.
  - Even an empty disk has a percentage of its space lost to inodes.
  - It improves the file system's performance, but consumes disk space.

Efficiency (2/2)

- Types of data kept in file's directory entry.

Efficiency (2/2)

- Types of data kept in file's directory entry.
  - E.g., "last write date" is recorded for some files.

Efficiency (2/2)

- Types of data kept in file's directory entry.
  - E.g., "last write date" is recorded for some files.
  - Updating this information is inefficient for frequently accessed files.

Efficiency (2/2)

- Types of data kept in file's directory entry.
  - E.g., "last write date" is recorded for some files.
  - Updating this information is inefficient for frequently accessed files.
  - Benefit against its performance cost?

Efficiency (2/2)

- Types of data kept in file's directory entry.
  - E.g., "last write date" is recorded for some files.
  - Updating this information is inefficient for frequently accessed files.
  - Benefit against its performance cost?

- Fixed-size or varying-size data structures.

Efficiency (2/2)

- Types of data kept in file's directory entry.
  - E.g., "last write date" is recorded for some files.
  - Updating this information is inefficient for frequently accessed files.
  - Benefit against its performance cost?

- Fixed-size or varying-size data structures.
  - E.g., Fix length process table: no more process after the process table becomes full

Efficiency (2/2)

- Types of data kept in file's directory entry.
  - E.g., "last write date" is recorded for some files.
  - Updating this information is inefficient for frequently accessed files.
  - Benefit against its performance cost?

- Fixed-size or varying-size data structures.
  - E.g., Fix length process table: no more process after the process table becomes full
  - Make them dynamic.

Performance

- Techniques to improve the file system performance:
  - Unified buffer cache
  - Optimizing sequential access
  - Synchronous and asynchronous writes

Unified Buffer Cache (1/4)

- Buffer cache
  - Separate section of main memory for frequently used blocks.

Unified Buffer Cache (1/4)

- Buffer cache
  - Separate section of main memory for frequently used blocks.
- Page cache

Unified Buffer Cache (1/4)

- Buffer cache
  - Separate section of main memory for frequently used blocks.

- Page cache
  - Cache file data as pages rather than as file-system blocks.

Unified Buffer Cache (1/4)

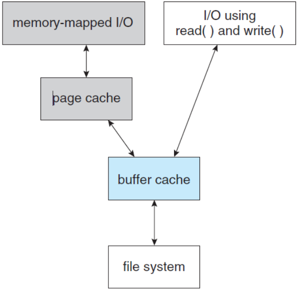
- Buffer cache
  - Separate section of main memory for frequently used blocks.
- Page cache
  - Cache file data as pages rather than as file-system blocks.
  - More efficient: accesses interface with virtual memory rather than the file system.

Unified Buffer Cache (2/4)

- Consider the two alternatives for opening and accessing a file:

Unified Buffer Cache (2/4)

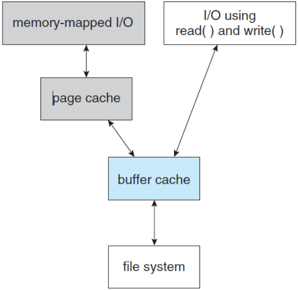
- Consider the two alternatives for opening and accessing a file:
  - Memory mapping
  - The standard system calls `read()` and `write()`



Unified Buffer Cache (2/4)

- Consider the two alternatives for opening and accessing a file:
  - Memory mapping
  - The standard system calls `read()` and `write()`

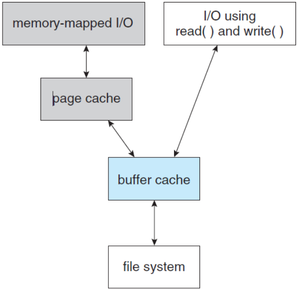
- Without a unified buffer cache:



Unified Buffer Cache (2/4)

- Consider the two alternatives for opening and accessing a file:
  - Memory mapping
  - The standard system calls `read()` and `write()`

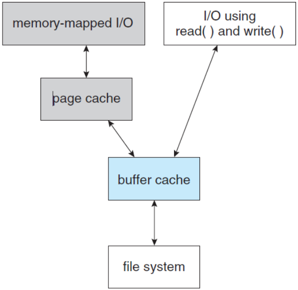
- Without a unified buffer cache:
  - The `read()` and `write()` go through the buffer cache.



Unified Buffer Cache (2/4)

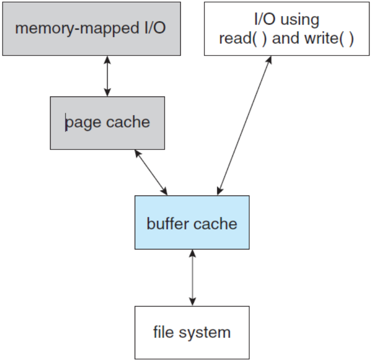
- Consider the two alternatives for opening and accessing a file:
  - Memory mapping
  - The standard system calls `read()` and `write()`

- Without a unified buffer cache:
  - The `read()` and `write()` go through the buffer cache.
  - The memory-mapping call, requires using two caches: the page cache and the buffer cache.



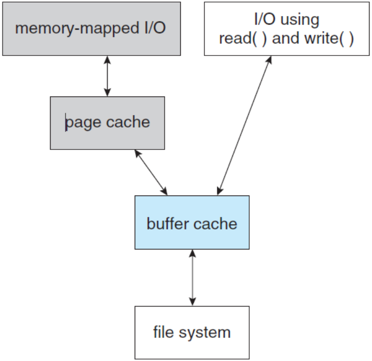
Unified Buffer Cache (3/4)

- Virtual memory does not interface with the buffer cache.



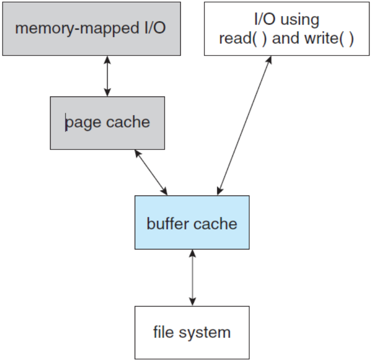
Unified Buffer Cache (3/4)

- Virtual memory does not interface with the buffer cache.
  - The contents of the file in the buffer cache must be copied into the page cache: double caching



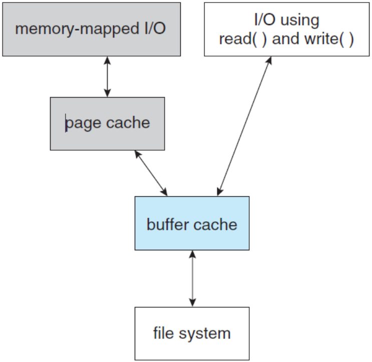
Unified Buffer Cache (3/4)

- Virtual memory does not interface with the buffer cache.
  - The contents of the file in the buffer cache must be copied into the page cache: double caching
  - Waste of memory, CPU and I/O cycles



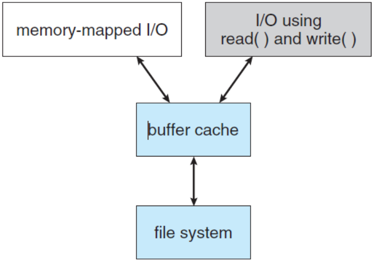
Unified Buffer Cache (3/4)

- Virtual memory does not interface with the buffer cache.
  - The contents of the file in the buffer cache must be copied into the page cache: double caching
  - Waste of memory, CPU and I/O cycles
  - Inconsistency



Unified Buffer Cache (4/4)

- With unified buffer cache.
- Both memory mapping and the `read()` and `write()` use the same page cache.
- LRU for block or page replacement.



Optimizing Sequential Access

- Optimizing page replacement in page cache for a file being read or written sequentially.

Optimizing Sequential Access

- Optimizing page replacement in page cache for a file being read or written sequentially.
  - The most recently used page will be used last, or perhaps never again.

Optimizing Sequential Access

- Optimizing page replacement in page cache for a file being read or written sequentially.
  - The most recently used page will be used last, or perhaps never again.

- Free-behind: removes a page from the buffer as soon as the next page is requested.

Optimizing Sequential Access

- Optimizing page replacement in page cache for a file being read or written sequentially.
  - The most recently used page will be used last, or perhaps never again.

- Free-behind: removes a page from the buffer as soon as the next page is requested.

- Read-ahead: a requested page and several subsequent pages are read and cached.
  - Retrieving these data from the disk in one transfer and caching them saves time.

Synchronous and Asynchronous Writes

- Synchronous writes sometimes requested by applications or needed by OS.
    - No buffering/caching: writes must hit disk before acknowledgement.

- Asynchronous writes more common, buffer-able, faster.

Recovery

Recovery

- A system crash can cause inconsistencies among on-disk file-system data structures.
    - E.g., directory structures, free-block pointers, and free FCB pointers.

Recovery

- A system crash can cause inconsistencies among on-disk file-system data structures.
  - E.g., directory structures, free-block pointers, and free FCB pointers.
  - For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB.

Recovery

- A system crash can cause inconsistencies among on-disk file-system data structures.
  - E.g., directory structures, free-block pointers, and free FCB pointers.
  - For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB.

- Methods to deal with corruption:
  - Consistency checking
  - Log-structured file systems
  - Backup and restore

Consistency Checking

- To detect a problem, OS scans of all the metadata on each file system to confirm or deny the consistency of the system.

Consistency Checking

- To detect a problem, OS scans of all the metadata on each file system to confirm or deny the consistency of the system.

- Consistency checking: compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.

Consistency Checking

- To detect a problem, OS scans of all the metadata on each file system to confirm or deny the consistency of the system.

- Consistency checking: compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.

- Can be slow and sometimes fails.

Log Structured File Systems (1/2)

- Transaction: a set of operations for performing a specific task.

Log Structured File Systems (1/2)

- Transaction: a set of operations for performing a specific task.
- Log structured (or journaling) file systems record each metadata update to the file system as a transaction.

Log Structured File Systems (1/2)

- Transaction: a set of operations for performing a specific task.
- Log structured (or journaling) file systems record each metadata update to the file system as a transaction.
- All transactions are written to a log.

Log Structured File Systems (1/2)

- Transaction: a set of operations for performing a specific task.
- Log structured (or journaling) file systems record each metadata update to the file system as a transaction.
- All transactions are written to a log.
  - A transaction is considered committed, once it is written to the log.

Log Structured File Systems (1/2)

- Transaction: a set of operations for performing a specific task.
- Log structured (or journaling) file systems record each metadata update to the file system as a transaction.
- All transactions are written to a log.
  - A transaction is considered committed, once it is written to the log.
  - Sometimes to a separate device or section of disk.

Log Structured File Systems (2/2)

- The transactions in the log are asynchronously written to the file system structures.

Log Structured File Systems (2/2)

- The transactions in the log are asynchronously written to the file system structures.
  - When the file system structures are modified, the transaction is removed from the log.

Log Structured File Systems (2/2)

- The transactions in the log are asynchronously written to the file system structures.
    - When the file system structures are modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed.

Log Structured File Systems (2/2)

- The transactions in the log are asynchronously written to the file system structures.
  - When the file system structures are modified, the transaction is removed from the log.

- If the file system crashes, all remaining transactions in the log must still be performed.

- Faster recovery from crash, removes chance of inconsistency of metadata.

Backup and Restore (1/2)

- Back up data from disk to another storage device, such as a magnetic tape or other hard disk.

- Recovery from the loss of an individual file, or of an entire disk, may then be a matter of restoring the data from backup.

Backup and Restore (2/2)

- A typical backup schedule:
  - Day 1. full backup: copy all files from the disk to a backup medium.
  - Day 2. incremental backup: copy all files changed since day 1 to another medium.
  - Day 3. incremental backup: copy all files changed since day 2 to another medium.
  - ...
  - Day N. incremental backup: copy all files changed since day N-1 to another medium. Then go back to day 1.

NFS

The Network File System (1/2)

- An implementation and a specification of a software system for accessing remote files across LANs or WANs.

The Network File System (1/2)

- An implementation and a specification of a software system for accessing remote files across LANs or WANs.

- NFS views a set of interconnected workstations as a set of independent machines with independent file systems.

The Network File System (1/2)

- An implementation and a specification of a software system for accessing remote files across LANs or WANs.
- NFS views a set of interconnected workstations as a set of independent machines with independent file systems.
- The goal is to allow some degree of sharing among these file systems in a transparent manner.

The Network File System (1/2)

- An implementation and a specification of a software system for accessing remote files across LANs or WANs.
- NFS views a set of interconnected workstations as a set of independent machines with independent file systems.
- The goal is to allow some degree of sharing among these file systems in a transparent manner.
- Sharing is based on a client-server relationship: either TCP or UDP/IP.

The Network File System (2/2)

- A remote directory is mounted over a local file system directory.

The Network File System (2/2)

- A remote directory is mounted over a local file system directory.
  - The mounted directory looks like an integral subtree of the local file system.

The Network File System (2/2)

- A remote directory is mounted over a local file system directory.
  - The mounted directory looks like an integral subtree of the local file system.
  - It replaces the subtree descending from the local directory.

The Network File System (2/2)

- A remote directory is mounted over a local file system directory.
  - The mounted directory looks like an integral subtree of the local file system.
  - It replaces the subtree descending from the local directory.

- Specification of the remote directory for the mount operation is non-transparent.

The Network File System (2/2)

- A remote directory is mounted over a local file system directory.
  - The mounted directory looks like an integral subtree of the local file system.
  - It replaces the subtree descending from the local directory.

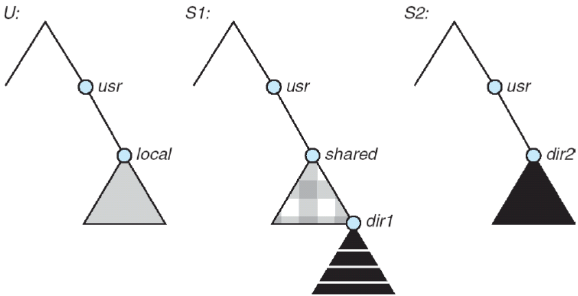
- Specification of the remote directory for the mount operation is non-transparent.
  - The host name of the remote directory has to be provided.

The Network File System (2/2)

- A remote directory is mounted over a local file system directory.
  - The mounted directory looks like an integral subtree of the local file system.
  - It replaces the subtree descending from the local directory.

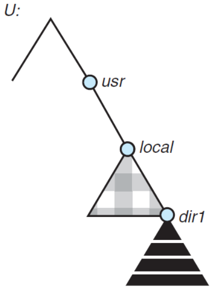
- Specification of the remote directory for the mount operation is non-transparent.
  - The host name of the remote directory has to be provided.
  - Files in the remote directory can then be accessed in a transparent manner.

NFS Mount (1/3)



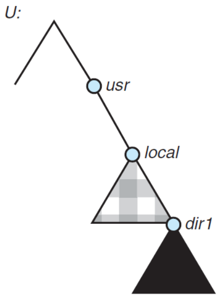
- Three independent file systems of machines named U, S1, and S2.

NFS Mount (2/3)



- Mounting S1:/usr/shared over U:/usr/local.
  `mount -t nfs S1:/usr/shared /usr/local`
- U can access any file within the dir1 using /usr/local/dir1.

NFS Mount (3/3)



- Cascading: mount a file system over another file system that is remotely mounted.
- Mounting `S2:/usr/dir2` over `U:/usr/local/dir1`, which is already remotely mounted from `S1`.
  `mount -t nfs S2:/usr/dir2 /usr/local/dir1`

NFS Mount Protocol (1/2)

- Establishes initial logical connection between server and client.

NFS Mount Protocol (1/2)

- Establishes initial logical connection between server and client.

- Mount operation includes name of remote directory to be mounted and name of server machine storing it.

NFS Mount Protocol (1/2)

- Establishes initial logical connection between server and client.
- Mount operation includes name of remote directory to be mounted and name of server machine storing it.
- Mount request is mapped to corresponding RPC and forwarded to mount server.

NFS Mount Protocol (1/2)

- Establishes initial logical connection between server and client.
- Mount operation includes name of remote directory to be mounted and name of server machine storing it.
- Mount request is mapped to corresponding RPC and forwarded to mount server.
- Export list: specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them.

NFS Mount Protocol (2/2)

- Following a mount request that conforms to its export list, the server returns a file handle (a key for further accesses).

NFS Mount Protocol (2/2)

- Following a mount request that conforms to its export list, the server returns a file handle (a key for further accesses).

- File handle: a file-system identifier and an inode number to identify the mounted directory within the exported file system.

NFS Mount Protocol (2/2)

- Following a mount request that conforms to its export list, the server returns a file handle (a key for further accesses).

- File handle: a file-system identifier and an inode number to identify the mounted directory within the exported file system.

- The mount operation changes only the user's view and does not affect the server side.

NFS Protocol (1/2)

- Provides a set of RPCs for remote file operations.
- The procedures support the following operations:
  - Searching for a file within a directory
  - Reading a set of directory entries
  - Manipulating links and directories
  - Accessing file attributes
  - Reading and writing files

NFS Protocol (2/2)

- NFS servers are stateless; each request has to provide a full set of arguments.
  - NFS V4 is just coming available: very different, stateful

NFS Protocol (2/2)

- NFS servers are stateless; each request has to provide a full set of arguments.
  - NFS V4 is just coming available: very different, stateful

- Modified data must be committed to the server's disk before results are returned to the client.

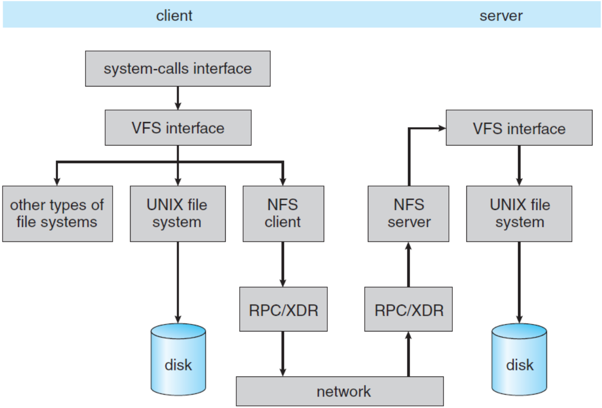
NFS Protocol (2/2)

- NFS servers are stateless; each request has to provide a full set of arguments.
  - NFS V4 is just coming available: very different, stateful

- Modified data must be committed to the server's disk before results are returned to the client.

- The NFS protocol does not provide concurrency-control mechanisms.

Schematic View of NFS Architecture



- NFS is integrated into the OS via a VFS.

Three Major Layers of NFS Architecture

- Unix file-system interface
  - Based on the open, read, write, and close calls, and file descriptors.

Three Major Layers of NFS Architecture

- Unix file-system interface
  - Based on the open, read, write, and close calls, and file descriptors.

- Virtual File System (VFS) layer
  - Distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
  - Calls the NFS protocol procedures for remote requests.

Three Major Layers of NFS Architecture

- Unix file-system interface
  - Based on the open, read, write, and close calls, and file descriptors.

- Virtual File System (VFS) layer
  - Distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
  - Calls the NFS protocol procedures for remote requests.

- NFS service layer
  - Implements the NFS protocol.

NFS Path-Name Translation

- It involves the parsing of a path name into separate components. E.g., `/usr/local/dir1/file.txt` into `usr`, `local`, and `dir1`.

NFS Path-Name Translation

- It involves the parsing of a path name into separate components. E.g., `/usr/local/dir1/file.txt` into `usr`, `local`, and `dir1`.
- Performs a separate NFS lookup call for every pair of component name and directory vnode.

NFS Path-Name Translation

- It involves the parsing of a path name into separate components. E.g., `/usr/local/dir1/file.txt` into `usr`, `local`, and `dir1`.
- Performs a separate NFS lookup call for every pair of component name and directory vnode.
- To make lookup faster, a directory-name-lookup cache on the client's side holds the vnodes for remote directory names.

NFS Remote Operations (1/2)

- Nearly one-to-one correspondence between regular Unix system calls and the NFS protocol RPCs.
  - Except opening and closing files.

NFS Remote Operations (1/2)

- Nearly one-to-one correspondence between regular Unix system calls and the NFS protocol RPCs.
  - Except opening and closing files.

- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance.

NFS Remote Operations (2/2)

- File-attribute cache: the attribute cache is updated whenever new attributes arrive from the server.
  - When a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes.
  - By default, discarded after 60 seconds.

NFS Remote Operations (2/2)

- File-attribute cache: the attribute cache is updated whenever new attributes arrive from the server.
  - When a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes.
  - By default, discarded after 60 seconds.

- File-blocks cache
  - Cached file blocks are used only if the corresponding cached attributes are up to date.

NFS Remote Operations (2/2)

- File-attribute cache: the attribute cache is updated whenever new attributes arrive from the server.
  - When a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes.
  - By default, discarded after 60 seconds.

- File-blocks cache
  - Cached file blocks are used only if the corresponding cached attributes are up to date.

- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk

Summary

Summary

- Free space management: bit vector, linked list, grouping, counting, space maps

Summary

- Free space management: bit vector, linked list, grouping, counting, space maps
- Efficiency: pre-allocated vs. as-needed allocation structures, types of data, fixed-size vs. varying size structures

Summary

- Free space management: bit vector, linked list, grouping, counting, space maps
- Efficiency: pre-allocated vs. as-needed allocation structures, types of data, fixed-size vs. varying size structures
- Performance: unified buffer cache, optimizing sequential access, synchronous and asynchronous writes

Summary

- Free space management: bit vector, linked list, grouping, counting, space maps
- Efficiency: pre-allocated vs. as-needed allocation structures, types of data, fixed-size vs. varying size structures
- Performance: unified buffer cache, optimizing sequential access, synchronous and asynchronous writes
- Recovery: consistency checking, log-structured FS, backup and restore

Summary

- Free space management: bit vector, linked list, grouping, counting, space maps
- Efficiency: pre-allocated vs. as-needed allocation structures, types of data, fixed-size vs. varying size structures
- Performance: unified buffer cache, optimizing sequential access, synchronous and asynchronous writes
- Recovery: consistency checking, log-structured FS, backup and restore
- NFS: mount protocol, path-name translation, remote operation

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.