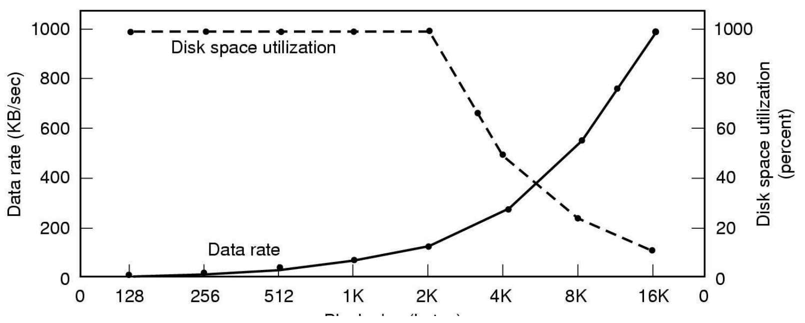


Figure 1: The solid curve (left hand scale) gives data rate of a disk. The dashed curve (right hand scale) gives disk space efficiency. All files 2KB (an approximate median file size).

0.1 File System Implementation

0.1.1 File System Layout

- Tradeoff in physical block size (see Fig. 1)
  - Sequential Access; The larger the block size, the fewer I/O operation required
  - Random Access
    - The larger the block size, the more unrelated data loaded.
    - Spatial locality of access improves the situation.
  - Choosing the an appropriate block size is a compromise

0.1.2 Implementing Files (see Fig. 2)

- The file system must keep track of
  - which blocks belong to which files.
  - in what order the blocks form the file
  - which blocks are free for allocation
- Given a logical region of a file, the file system must identify the corresponding block(s) on disk.
  - Stored in file allocation table (FAT) (see Fig. 2), directory, Inode
- Creating and writing files allocates blocks on disk

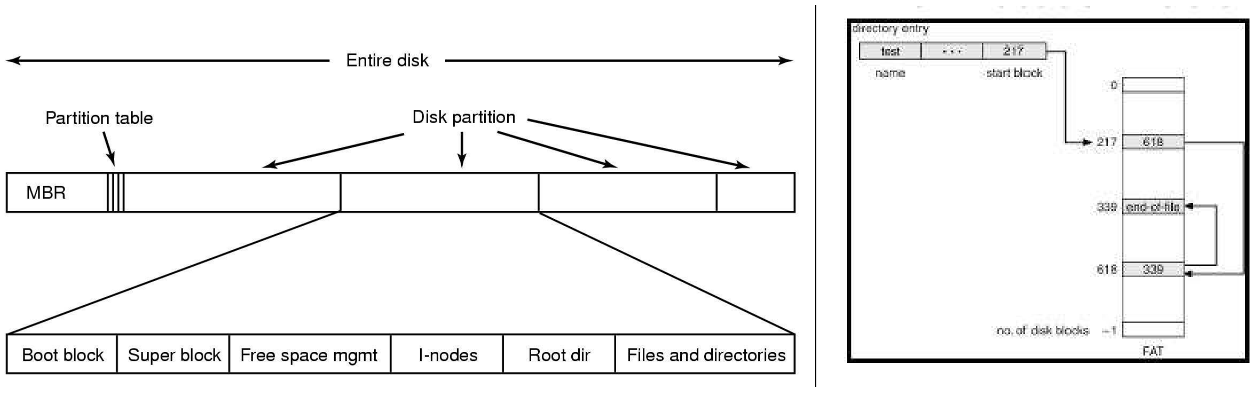


Figure 2: Left: A possible file system layout. Right: File Allocation Table.

- **Allocation Strategies**
  - Preallocation
    - ∗ Need the maximum size for the file at the time of creation
    - ∗ Difficult to reliably estimate the maximum potential size of the file
    - ∗ Tend to overestimated file size so as not to run out of space
  - Dynamic Allocation; Allocated in portions as needed
- **Portion Size**
  - Extremes
    - ∗ Portion size = length of file (remember or contiguous)
    - ∗ Portion size = block size
  - Tradeoffs
    - ∗ Contiguity increases performance for sequential operations
    - ∗ Many small portion increase the size of the file allocation table
    - ∗ Fixed–sized portions simplify reallocation of space
    - ∗ Variable–sized portions minimize internal fragmentation losses
- **Methods of File Allocation**
  - The file system allocates disk space, when a file is created. With many files residing on the same disk, the main problem is how to allocate space for them. File allocation scheme has impact on the efficient use of disk space and file access time.
  - Common file allocation techniques are:

    - Contiguous
    - Chained (linked)
    - Indexed
  - All these techniques allocate disk space on a per block (smallest addressable disk unit) basis.

- **Methods of File Allocation; Contiguous allocation** (see Fig. 3a, b)

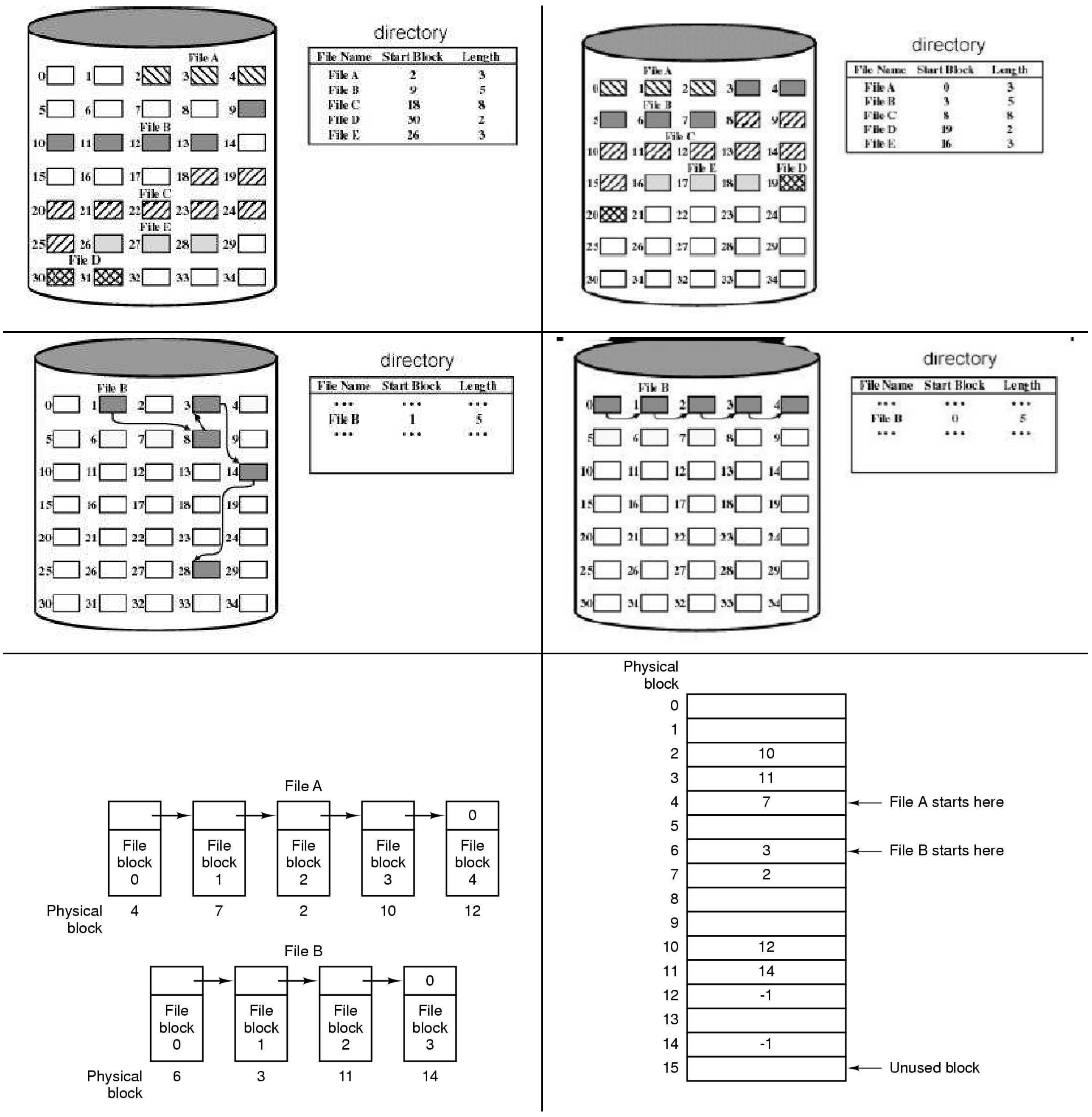
  - Allocate disk space like paged, segmented memory. Keep a free list of unused disk space.
  - Single set of blocks is allocated to a file at the time of creation
  - Advantages;
    - Only a single entry in the directory entry; Starting block and length of the file
    - easy access, both sequential and random
    - Simple, only starting location (block #) and length (number of blocks) to find all contents
    - few seeks
  - Disadvantages;
    - External fragmentation will occur
    - May not know the file size in advance
    - Eventually, we will need compaction to reclaim unusable disk space.
    - Files can't grow

- **Methods of File Allocation; Chained (or linked list) allocation** (see Fig. 3c,d,e,f)

  - Space allocation is similar to page frame allocation. Mark allocated blocks as in-use
  - Each block contains a pointer to the next block in the chain
  - Only single entry in a directory entry; Starting block and length of file
  - No external fragmentation; Free-space manageable, no wasted space
  - Files can grow easily

  - Simple; need only starting address
  - Best for sequential files; Poor for random access
  - No accommodation of the principle of locality; Blocks end up scattered across the disk
  - To improve performance, we can run a defragmentation utility to consolidate files.
  - Storing a file as a linked list of disk blocks (see Fig. 3e)
  - Linked allocation with file allocation table (FAT) in RAM (see Fig. 3f)
    * Avoids disk accesses when searching for a block
    * Entire block is available for data
    * Table gets too large for large file systems; Can cache parts of it, but still can consume significant RAM or generate disk traffic
    * Used in MSDOS, OS/2

- **Methods of File Allocation; Indexed allocation** (see Fig. 4)
  - Allocate an array of pointers during file creation. Fill the array as new disk blocks are assigned
  - File allocation table contains a separate one level index for each file
  - The index has one entry for each portion allocated to the file
  - The file allocation table contains block number for the index
  - Supports both sequential and direct (easy) access to the file
  - Small internal fragmentation
  - Lots of seeks if the file is big
  - Maximum file size is limited to the size of a block
  - Portions
    * Block sized; Eliminates external fragmentation
    * Variable sized; Improves contiguity, Reduces index size
  - Most popular of the three forms of file allocation
  - Example: UNIX file system



directory

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

directory

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

directory

File Name	Start Block	Length
•••	•••	•••
File B	1	5
•••	•••	•••

directory

File Name	Start Block	Length
•••	•••	•••
File B	0	5
•••	•••	•••

Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

Figure 3: (a) Contiguous allocation (b) Contiguous allocation with compaction (c) Storing a file as a linked list of disk blocks (d) Storing a file as a linked list of disk blocks with defragmentation (e) Alternative representation of chained allocation (f) Linked list allocation using a file allocation table in main memory.

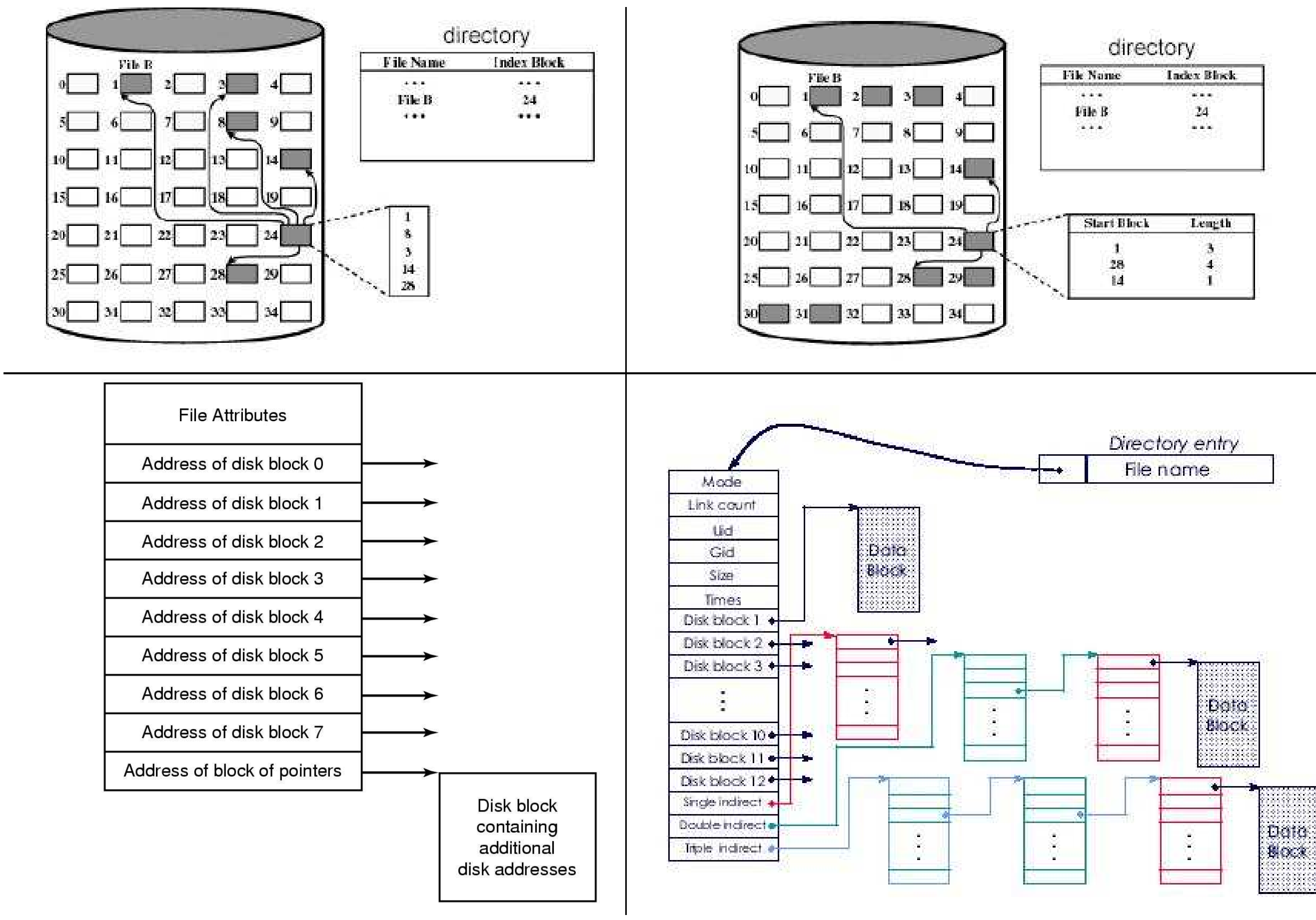


Figure 4: (a) Indexed allocation with block partitions (b) Indexed allocation with variable–length partitions (c) An example of i-node.

0.1.3 Implementing Directories

- (see Fig. 5 Left)
- A simple directory containing fixed–sized entries with the disk addresses and attributes in directory entry; **DOS/Windows**
- A directory in which each entry just refers to an inode; **UNIX**
- Fixed Size Directory Entries;
  - Either too small; Example: DOS 8+3 characters
  - Waste too much space; Example: 255 characters per file name

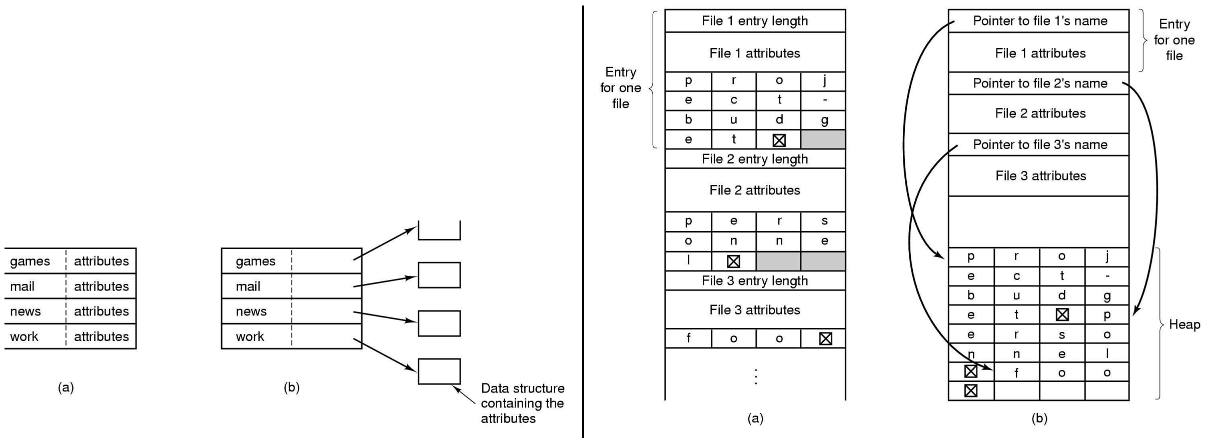


Figure 5: Left: (a) A simple directory containing fixed–sized entries with the disk addresses and attributes in directory entry (b) A directory in which each entry just refers to an inode. Right: Two ways of handling long file names in directory (a) Inline (b) In a heap.

- Free variable length entries can create external fragmentation in directory blocks; Can compact when block is in RAM

0.1.4 Shared Files (see Fig. 6)

- Copy entire directory entry (including file attributes)
  - Updates to shared file not seen by all parties
  - Not useful
- Keep attributes separate (in Inode) and create a new entry that points to the attributes (hard link)
  - Updates visible
  - If one link remove, the other remains (ownership is an issue)
- Create a special ”LINK” file that contains the pathname of the shared file (symbolic link)
  - File removal leaves dangling links
  - Not as efficient to access

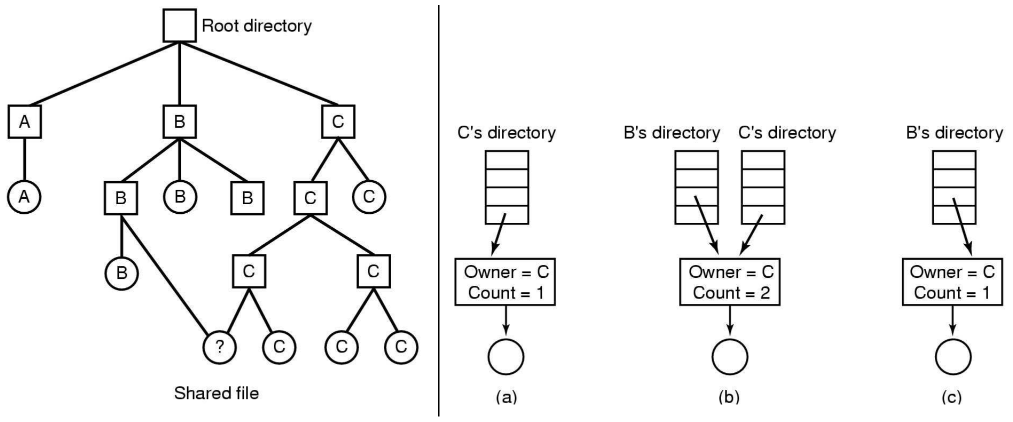


Figure 6: Left: File system containing a shared file. Right: (a) Situation prior to linking (b) After the link is created (c)After the original owner removes the file.

0.1.5 Disk Space Management (Free space management)

- Since the amount of disk space is limited (posing a management problem similar to that of physical memory), it is necessary to reuse the space released by deleted files
- In general, file systems keep a list of free disk blocks (initially, all the blocks are free) and manage this list by one of the following techniques
- **Bit tables** (see Fig. 7b)
  - Individual bits in a bit vector flags used/free blocks
  - 16GB disk with 1KB blocks; for each block 1 bit is used $(16Gb/1KB) \approx 2^{24}/8/1KB = 2048$ blocks; 2MB table
  - 16GB disk with 512byte blocks 4MB table
  - May be too large to hold in main memory
  - Expensive to search; But may use a two level table
- **Chained free portions** (see Fig. 7a)
  - Free portions are linked
  - Fragmentation if using variable allocation $\Rightarrow$ many small portions
  - Required read before write to a block
- **Free block list**

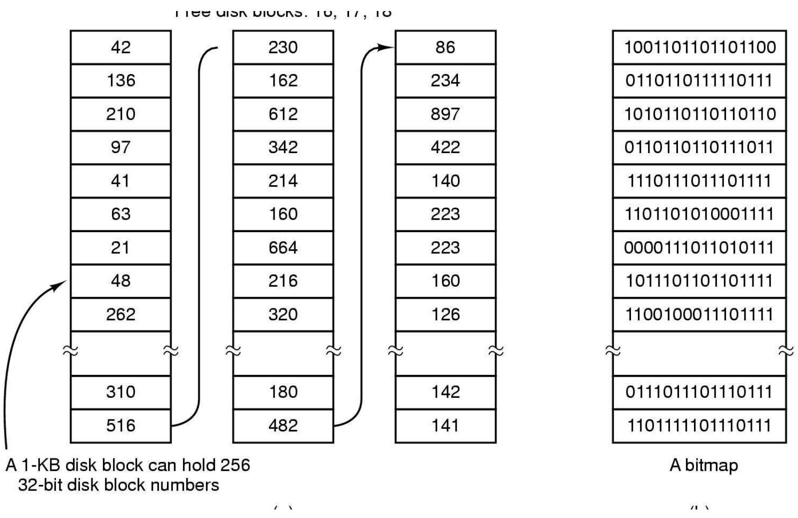


Figure 7: (a) Storing the free list on a linked list (b) A bit map.

- Single list of a set of free block lists (unallocated blocks)
- Manage as LIFO or FIFO on disk with ends in main memory
- Background jobs can reorder list for better contiguity

0.1.6 Other file system implementation issues

- Recovery, Reliability
  - Consistency checking; compares data in directory with data blocks on disk, tries to fix inconsistencies (e.g., UNIX **fsck**)
  - Use system programs to back up data from disk to another device (optical media, magnetic tape) (disaster scenarios)
  - Recover lost file or disk by restoring from backup
- Efficiency and Performance
  - Efficiency dependent on:
    - disk allocation and directory algorithms
    - types of data kept in file's directory entry
  - Performance: disk and page caches
    - disk cache: frequently used blocks in main memory
    - free-behind, read-ahead: optimize sequential access

    - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.

- Log Structured File Systems
  - Log structured (or journaling) file systems record each update to file system as transaction.
  - All transactions written to log. Transaction considered committed once written to log file system may not yet be updated.
  - Transactions in log asynchronously written to file system. When file system modified, transaction removed from log.
  - If file system crashes, all remaining logged transactions must still be performed
- Sun Network File System (NFS)
  - Implementation and specification of software system for accessing remote files across LANs (or WANs)
  - Implemented as part of Solaris, SunOS on Suns: uses unreliable datagram protocol (UDP/IP and Ethernet)
  - Widely used in UNIX (including free kinds)
  - Interconnected machines seen as independent with independent file systems allowing transparent sharing
    - remote directory mounted over local file system directory; mounted directory looks like integral subtree of local file system, replacing subtree descending from local directory
    - specification of remote directory for mount operation non-transparent; host name of remote directory provided: files in remote directory can then be accessed transparently
    - subject to access-rights, potentially any file system (or directory within file system), can be mounted remotely on top of any local directory
  - NFS designed to operate in heterogeneous environment: different machines, OSes, network architectures; NFS specifications independent of these details