Lecture 12

File System Implementation

Lecture Information

Ceng328 *Operating Systems* at May 11, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

File-System Structure I

- Disks provide the bulk of secondary storage on which a file system is maintained.
- They have two characteristics that make them a convenient medium for storing multiple files:
  1. A disk can be rewritten in place;
  2. A disk can access directly any given block of information it contains.
- Rather than transferring a byte at a time, I/O transfers between memory and disk are performed in units of blocks.
- A file system poses two quite different design problems.
  1. The first problem is defining how the file system should look to the user.
  2. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- The file system itself is generally composed of many different levels.

File-System Structure II

- The structure shown in Fig 1 is an example of a layered design.

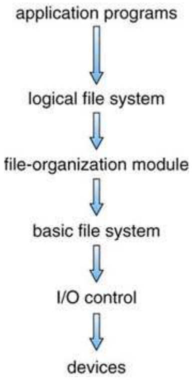


Figure: Layered file system.

- Each level in the design uses the features of lower levels to create new features for use by higher levels.

File-System Structure III

- The lowest level, the **I/O control**, consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- The **file-organization** module knows about files and their logical blocks, as well as physical blocks (mapping, free-space management).
- Finally, the **logical file system** manages metadata information.
    - Metadata includes all of the file-system structure except the actual data (or contents of the files).
    - It maintains file structure via file-control blocks (FCB). A FCB contains information about the file, including ownership, permissions, and location of the file contents.
- Many file systems are in use today; ISO 9660, UNIX file system (UFS), FAT, FAT32, NTFS, ext2, ext3, ext4.

Overview I

- On disk, the file system may contain information about
  - how to boot an OS stored there,
  - the total number of blocks,
  - the number and location of free blocks,
  - the directory structure,
  - individual files.
- Often the file system will contain some of the items shown in Fig. 2.

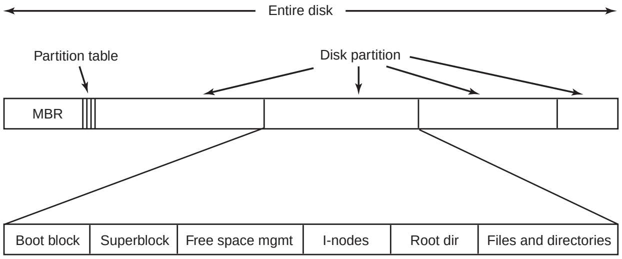


Figure: A possible file system layout.

Overview II

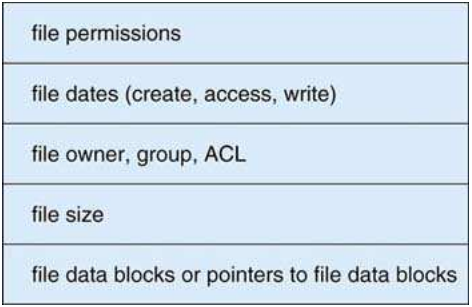
- Sector 0 of the disk is called the MBR (Master Boot Record) and is used to boot the computer. The end of the MBR contains the partition table.
- One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR.
- A **boot control block** (per volume) can contain information needed by the system to boot an OS from that volume.
  - In UFS, it is called the **boot block**;
  - In NTFS, it is the **partition boot sector**.
- A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, freeblock count and free-block pointers, and free FCB count and FCB pointers.
  - In UFS, this is called a **superblock**;
  - In NTFS, it is stored in the **master file table**.

Overview III

- Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers.
- A directory structure per file system is used to organize the files.
  - In UFS, this includes file names and associated **inode numbers**.
  - In NTFS it is stored in the **master file table**.
- After that might come the root directory, which contains the top of the file system tree.
- Finally, the remainder of the disk typically contains all the other directories and files.

Overview IV

- A per-file FCB contains many details about the file, including file permissions, ownership, size, and location of the data blocks.
  - In UFS, this is called the **inode**.
  - In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.
- A typical FCB is shown in Fig. 3.



file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure: A typical file-control block.

Overview V

- Some OSs, including UNIX, treat a directory exactly the same as a file-one with a type field indicating that it is a directory.
- The operating structures of a file-system implementation are summarized in Fig. 4.

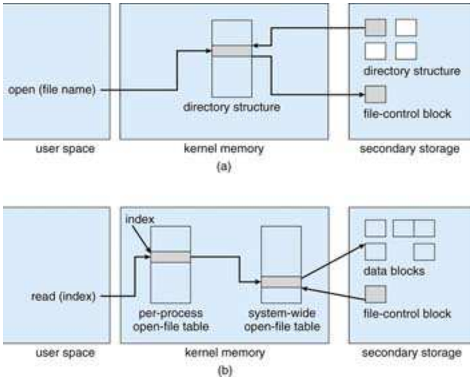


Figure: In-memory file-system structures. (a) File open. (b) File read.

Partitions and Mounting

- A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks (RAID).
- Each partition can be either “raw”, containing no file system (UNIX swap space can use a raw partition), or “cooked”, containing a file system.
- The **root partition**, which contains the OS kernel and sometimes other system files, is mounted at boot time.
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the OS.
- Microsoft Windows-based systems mount each volume in a separate name space, denoted by a letter and a colon (F:).
- On UNIX, file systems can be mounted at any directory.
  - Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory.
  - The flag indicates that the directory is a mount point.
- The mount table entry contains a pointer to the superblock of the file system on that device.

Virtual File Systems I

- How does an OS allow multiple types of file systems to be integrated into a directory structure?
- An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type.
- Instead, most OSs, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation.
- The use of these methods allows very dissimilar file-system types to be implemented within the same structure.
- The file-system implementation consists of three major layers, as depicted schematically in Fig. 5.

1. The first layer is the file-system interface, based on the *open*(), *read*(), *write*(), and *close*() calls and on file descriptors.

Virtual File Systems II

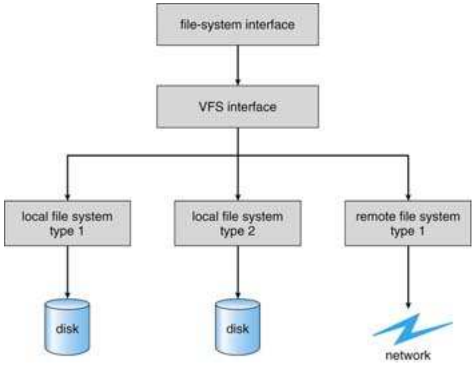


Figure: Schematic view of a virtual file system.

Virtual File Systems III

2 The second layer is called the virtual file system (VFS) layer; it serves two important functions:
    - It separates file-system-generic operations from their implementation by defining a clean VFS interface.
    - The VFS provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a *vnode*, that contains a numerical designator for a network-wide unique file.

- The VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

Allocation Methods

- The direct-access nature of disks allows us flexibility in the implementation of files.
- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- Three major methods of allocating disk space are in wide use:
  1. **contiguous**,
  2. **linked**,
  3. **indexed**.

Contiguous Allocation I

- **Contiguous allocation** requires that each file occupy a set of contiguous blocks on the disk (see Fig. 6).

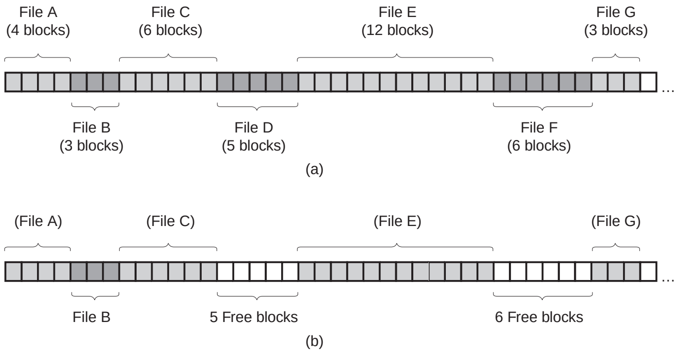


Figure: (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files D and F have been removed.

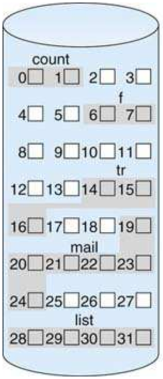
- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.

Contiguous Allocation II

- Disk addresses define a linear ordering on the disk. With this ordering,
  - accessing block $b + 1$ after block b normally requires no head movement.
  - When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next.
  - Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.
- Contiguous allocation is widely used on CD-ROMs.
- Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system.

Contiguous Allocation III

- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (see Fig. 7).



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Figure: Contiguous allocation of disk space.

Contiguous Allocation IV

- As files are allocated and deleted, the free disk space is broken into little pieces.
- **External fragmentation** exists whenever free space is broken into chunks.
    - It becomes a problem when the largest contiguous chunk is insufficient for a request;
    - Storage is fragmented into a number of holes, no one of which is large enough to store the data.
- Compacting all free space into one contiguous space, solves the fragmentation problem.
    - The cost of this compaction is time (could be severe).
- Another problem with contiguous allocation is determining **how much space is needed** for a file.
- When the file is created, the total amount of space it will need must be found and allocated.

Contiguous Allocation V

- How does the creator (program or person) know the size of the file to be created?
- If we allocate too little space to a file, we may find that the file cannot be **extended**.
- Two possibilities then exist.
  - First, the user program can be terminated with an appropriate error message. The user must then allocate more space and run the program again.
  - The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space.
- Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient.

Linked Allocation I

- The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 8.

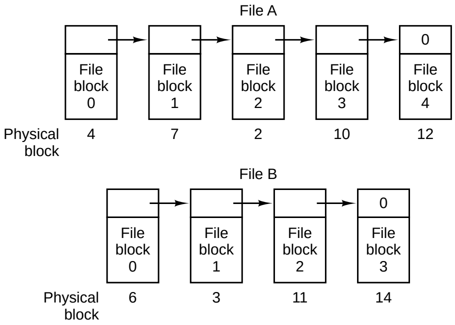


Figure: Storing a file as a linked list of disk blocks.

- **Linked allocation** solves all problems of contiguous allocation.
  - The disk blocks may be scattered anywhere on the disk.
  - The directory contains a pointer to the first and last blocks of the file.

Linked Allocation II

For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (see Fig. 9).

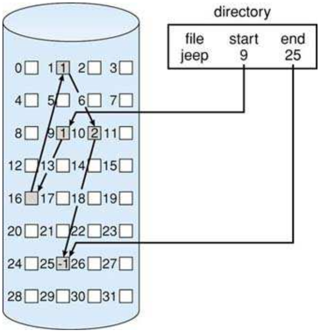


Figure: Linked allocation of disk space.

Linked Allocation III

- To create a new file, we simply create a new entry in the directory.
    - The pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file.
    - The size field is also set to O.
- A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
- To read a file, we simply read blocks by following the pointers from block to block.
- No space is lost to disk fragmentation (except for internal fragmentation in the last block).
- The size of a file need not be declared when that file is created. A file can continue to grow as long as free blocks are available.
- Consequently, it is never necessary to compact disk space.

Linked Allocation IV

- The major problem is that it can be used effectively only for sequential-access files.
  - To find the i^{th} block of a file, we must start at the beginning of that file and follow the pointers until we get to the i^{th} block.
  - Each access to a pointer requires a disk read, and some require a disk seek.
- Consequently, it is inefficient to support a direct-access capability for linked-allocation files.
- Another disadvantage is the space required for the pointers.
  - If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.
- The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks.
- Yet another problem of linked allocation is reliability.
  - What would happen if a pointer were lost or damaged?
  - A bug in the OS software or a disk hardware failure might result in picking up the wrong pointer.

Linked Allocation V

- An important variation on linked allocation is the use of a file-allocation table (FAT) (MS-DOS and OS/2 OSs).
- An illustrative example is the FAT structure shown in Fig. 10 for a file consisting of disk blocks 217, 618, and 339.

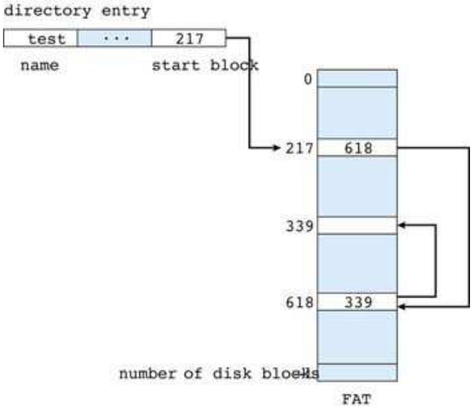


Figure: File allocation table.

Linked Allocation VI

- The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.
- The primary disadvantage of this method is that the entire table must be in memory all the time to make it work.
  - With a 20-GB disk and a 1-KB block size, the table needs 20 million entries.
  - Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes.
  - Thus the table will take up 60 MB or 80 MB of main memory all the time.

Indexed Allocation I

- In the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.
- A data structure called an **i-node** (index-node), which lists the attributes and disk addresses of the files blocks (see Fig. 11).

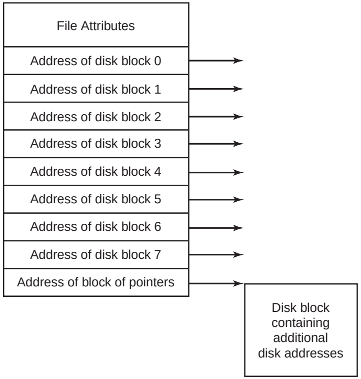


Figure: An example i-node.

Indexed Allocation II

- **Indexed allocation** solves this problem by bringing all the pointers together into one location: the index block.
- Each file has its own index block, which is an array of disk-block addresses.
- The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block (see Fig. 12).

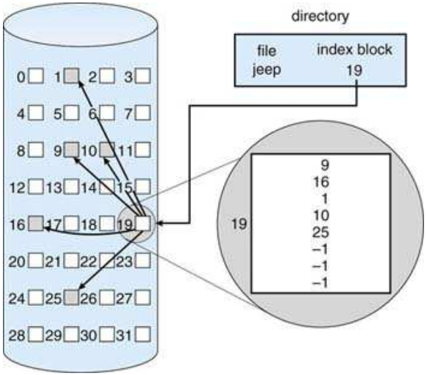
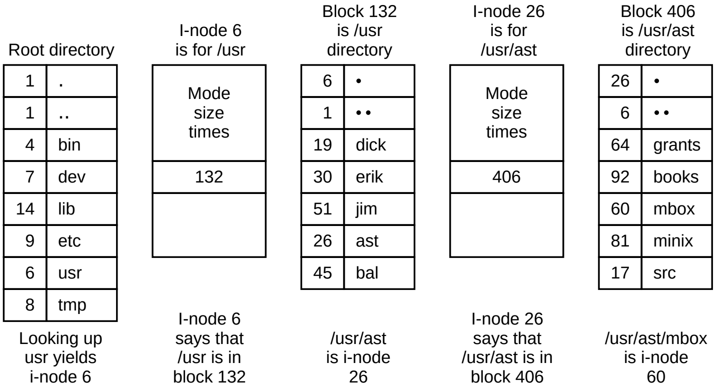


Figure: Indexed allocation of disk space.

Indexed Allocation III

- To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry (paging scheme).
- Given the i-node, it is then possible to find all the blocks of the file.
- The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open.
- When the file is created, all pointers in the index block are set to *nil*.
- When a file is opened, the file system must take the file name supplied and locate its disk blocks.
- Let us consider how the path name */usr/ast/mbox* is looked up. The lookup process is illustrated in Fig. 13.

Indexed Allocation IV


Root directory
I-node 6 is for /usr
Block 132 is /usr directory
I-node 26 is for /usr/ast
Block 406 is /usr/ast directory
Mode size times
132
Mode size times
406
Looking up usr yields i-node 6
I-node 6 says that /usr is in block 132
/usr/ast is i-node 26
I-node 26 says that /usr/ast is in block 406
/usr/ast/mbox is i-node 60

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Block 132 is /usr directory

6	•
1	• •
19	dick
30	erik
51	jim
26	ast
45	bal

Block 406 is /usr/ast directory

26	•
6	• •
64	grants
92	books
60	mbox
81	minix
17	src

Figure: The steps in looking up */usr/ast/mbox*.

Indexed Allocation V

- Indexed allocation supports direct access, without suffering from external fragmentation.
- Indexed allocation does suffer from wasted space, however.
- Consider a common case in which we have a file of only one or two blocks.
  - With linked allocation, we lose the space of only one pointer per block.
  - With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be *non-nil*.
- This point raises the question of how large the index block should be.
  - Every file must have an index block, so we want the index block to be as small as possible.
  - If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue.
- Mechanisms for this purpose include the followings.

Indexed Allocation VI

- **Linked scheme**. An index block is normally one disk block.To allow for large files, we can link together several index blocks.
  - For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses.
  - The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).
- **Multilevel index**. A variant of the linked representation is to use a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.
  - To access a block, the OS uses the first-level index to find a second-level index block and then uses that block to find the desired data block.
  - This approach could be continued to a third or fourth level, depending on the desired maximum file size.
  - With 4096-byte blocks, we could store 1024 4-byte pointers in an index block.
  - Two levels of indexes allow 1048576 data blocks and a file size of up to 4 GB.

Indexed Allocation VII

- **Combined scheme.** Another alternative, used in the UFS (UNIX File System), is to keep the first, say, 15 pointers of the index block in the file's inode.
  - The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file.
  - If the block size is 4 KB, then up to 48 KB of data can be accessed directly.
  - The next three pointers point to indirect blocks.
    - The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.
    - The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.
    - The last pointer contains the address of a triple indirect block.
  - Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many OSs (32-bit file pointer: 4 GB).
  - Many UNIX implementations now support up to 64-bit file pointers (terabytes).

Indexed Allocation VIII

A UNIX inode is shown in Fig. 14.

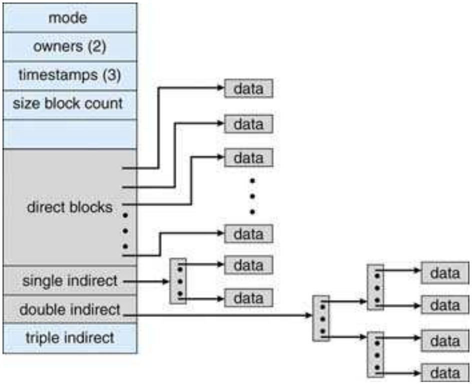


Figure: The UNIX inode.

Bit Vector I

- Frequently, the free-space list is implemented as a **bit map** or **bit vector**.
- Each block is represented by 1 bit.
  - If the block is free, the bit is 1;
  - If the block is allocated, the bit is O.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18,25,26, and 27 are free and the rest of the blocks are allocated.
- The free-space bit map would be

  `001111001111110001100000011100000 ...`
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk.
- Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to disk occasionally for recovery needs).

Bit Vector II

- Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.
- A 1.3-GB disk with 512-byte blocks would need a bit map of over 332 KB to track its free blocks, although clustering the blocks in groups of four reduces this number to over 83 KB per disk
  $(1.3 * 1020 * 1024 * 1024/512/8/1.024 = 332.8\ KB)$.
- A 40-GB disk with 1-KB blocks requires over 5 MB to store its bit map
  $(40 * 1020 * 1024 * 1024/1024/8/1.024 = 5.12\ MB)$.
- A 500-GB disk with a 1-KB block and a 32-bit (4 bytes) disk block number, we need 488 million bits for the map, which requires just under 60000 1-KB blocks to store $((500x10^9/1KB)/1024/8)$.

Linked List I

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on. In our earlier example (Section 1);

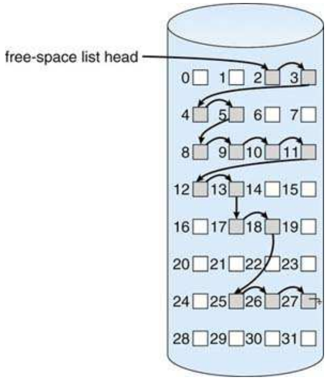


Figure: Linked free-space list on disk.

- We would keep a pointer to block 2 as the first free block.
- Block 2 would contain a pointer to block 3,
- which would point to block 4,
- which would point to block 5,
- which would point to block 8, and so on (see Fig. 15).

Linked List II

Keeping it in main memory;

- With a 1-KB block and a 32-bit (4 bytes) disk block number, each block on the free list holds the numbers of 255 free blocks. (1KB/32-bit=256; one slot is needed for the pointer to the next block. The number of blocks that could be addressed:$2^{32} \simeq 4.3x10^{9}$).
- A 500-GB disk, which has about 488 million ($500x10^{9}/1KB$) disk blocks . To store all these addresses at 255 per block requires about 1.9 million blocks ($500x10^{9}/1KB/255$). Generally, free blocks are used to hold the free list, so the storage is essentially free.
- It is not surprising that the bitmap requires less space (60000 blocks), since it uses 1 bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked list scheme require fewer blocks than the bitmap.

Linked List III

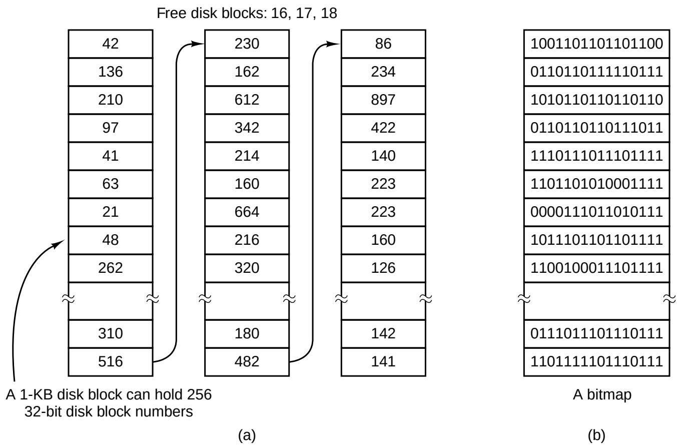


Figure: (a) Storing the free list on a linked list. (b) A bitmap.

Log-Structured File Systems I

- The one parameter that is not improving and bounds is *disk seek time*.
- The idea that drove the LFS (the Log-structured File System) design is that disk caches are increasing rapidly.
- Consequently, it is now possible to satisfy a very substantial fraction of all read requests directly from the file system cache, with <u>no disk access needed</u>.
- Most disk accesses will be writes. In most file systems, writes are done in very small chunks.
- Small writes are highly inefficient, since a 50-μsec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay.
- With these parameters, disk efficiency drops to a fraction of 1 percent.
- While the writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done.

Log-Structured File Systems II

- From this reasoning, the LFS designers decided to re-implement the UNIX file system in such a way as to achieve the full bandwidth of the disk.
- The basic idea is to structure the entire disk as a *log*.
- The logging algorithms have been also applied successfully to the problem of consistency checking.
- The resulting implementations are known as **log-based transaction-oriented** (or **journaling**) file systems.
- Such file systems are actually in use (NTFS, ext3, ReiserFS).
- Recall that a system crash can cause inconsistencies among on-disk file system data structures, such as directory structures, free-block pointers, and free FCB pointers.

Log-Structured File Systems III

- A typical operation, such as file create, can involve many structural changes within the file system on the disk.
  - Directory structures are modified,
  - FCBs are allocated,
  - Data blocks are allocated,
  - The free counts for all of these blocks are decreased.
- These changes can be interrupted by a crash, and inconsistencies among the structures can result.
- For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB.
- The consistency check may not be able to recover the structures, resulting in loss of files and even entire directories.
- The solution to this problem is to apply log-based recovery techniques to file-system metadata updates.
- Both NTFS and the Veritas (improved UFS) file system use this method, and it is an optional addition to UFS on Solaris 7 and beyond.

Log-Structured File Systems IV

- Fundamentally, all metadata changes are written sequentially to a log. Each set of operations for performing a specific task is a **transaction**.
- The log may be in a separate section of the file system or even on a separate disk.
- If the system crashes, the log file will contain zero or more transactions.
  - Any transactions it contains were not completed to the file system, even though they were committed by the OS, so they must now be completed.
  - The transactions can be executed from the pointer until the work is complete so that the file-system structures remain consistent.
- The only problem occurs when a transaction was aborted -that is, was not committed before the system crashed.
  - Any changes from such a transaction that were applied to the file system must be undone, again preserving the consistency of the file system.
  - This recovery is all that is needed after a crash, eliminating any problems with consistency checking.