Implementing File System

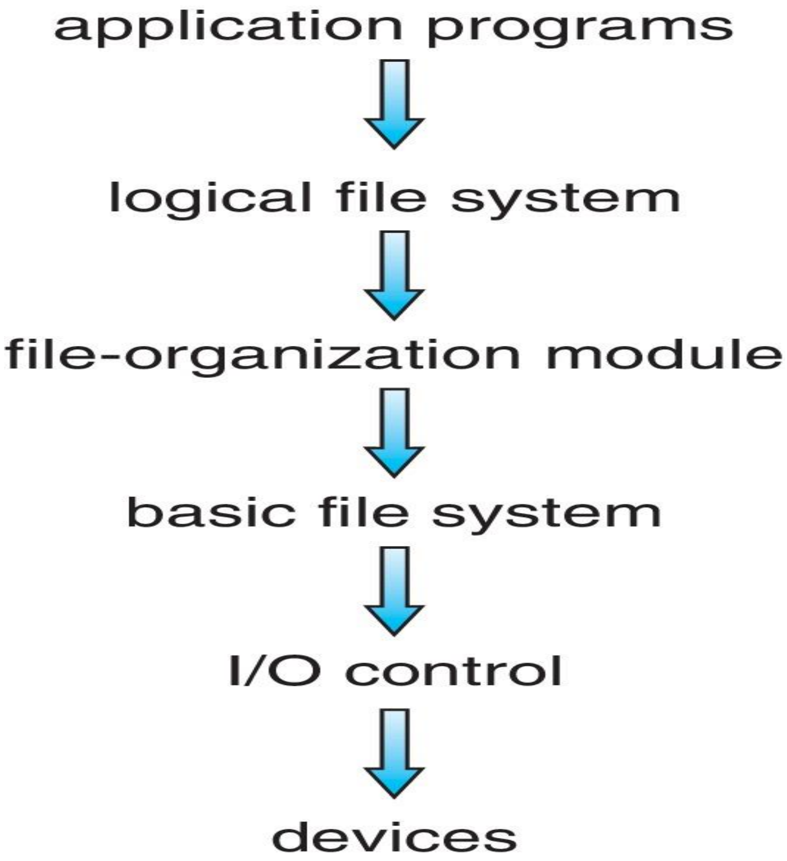
UNIT-IV

File system structure

- File systems provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily
- A file system poses two quite different design problems
  1. The first problem is defining how the file system should look to the user.
     - This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files
  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
- The structure shown in Figure in the next slide is an example of a layered design
- Each level in the design uses the features of lower levels to create new features for use by higher levels
- The I/O control level
  - consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system

Layered file system



application programs

logical file system

file-organization module

basic file system

I/O control

devices

- A device driver can be thought of as a translator
- Its input consists of high level commands, such as “retrieve block 123”
- Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system
- The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller which device location to act on and what actions to take

- The basic file system (called the “block I/O subsystem” in Linux)
  - needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device

  - issues commands to the drive based on logical block addresses
  - is also concerned with I/O request scheduling
  - This layer also manages the memory buffers and caches that hold various file system, directory and data blocks

- The file-organization module
  - knows about files and their logical blocks
  - Each file’s logical blocks are numbered from 0 (or 1) through N
  - also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested

- 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
  - manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name
  - maintains file structure via file-control blocks
  - A file control block (FCB) (an inode in UNIX file systems) contains information about the file, including ownership, permissions and location of the file contents
  - is also responsible for protection

- Merits
  - When a layered structure is used for file-system implementation, duplication of code is minimized
  - The I/O control and sometimes the basic file-system code can be used by multiple file systems
  - Each file system can then have its own logical file-system and file-organization modules
- Demerit
  - layering can introduce more operating-system overhead, which may result in decreased performance

- Many file systems are in use today, and most operating systems support more than one
- For example, most CD-ROMs are written in the ISO 9660 format, a standard format agreed on by CD-ROM manufacturers
- In addition to removable-media file systems, each operating system has one or more disk based file systems
  - UNIX uses the UNIX file system (UFS) which is based on the Berkeley Fast File System (FFS)

- Windows supports disk file system formats of
  - FAT, FAT32 and NTFS (or Windows NT File System)
  - CD-ROM and DVD file-system formats
- Although Linux supports over 130 different file systems, the standard Linux file system is known as the extended file system, with the most common versions being ext3 and ext4.

File system operations-Overview

- Several on-storage and in-memory structures are used to implement a file system
- These structures vary depending on the operating system and the file system, but some general principles apply
- On storage
  - the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files

- On storage structures used are:
- A **boot control block** (per volume)
  - can contain information needed by the system to boot an operating system from that volume
  - If the disk does not contain an operating system, this block can be empty
  - It is typically the first block of a 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 details, such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB 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
- A per-file FCB
  - Contains many details about the file

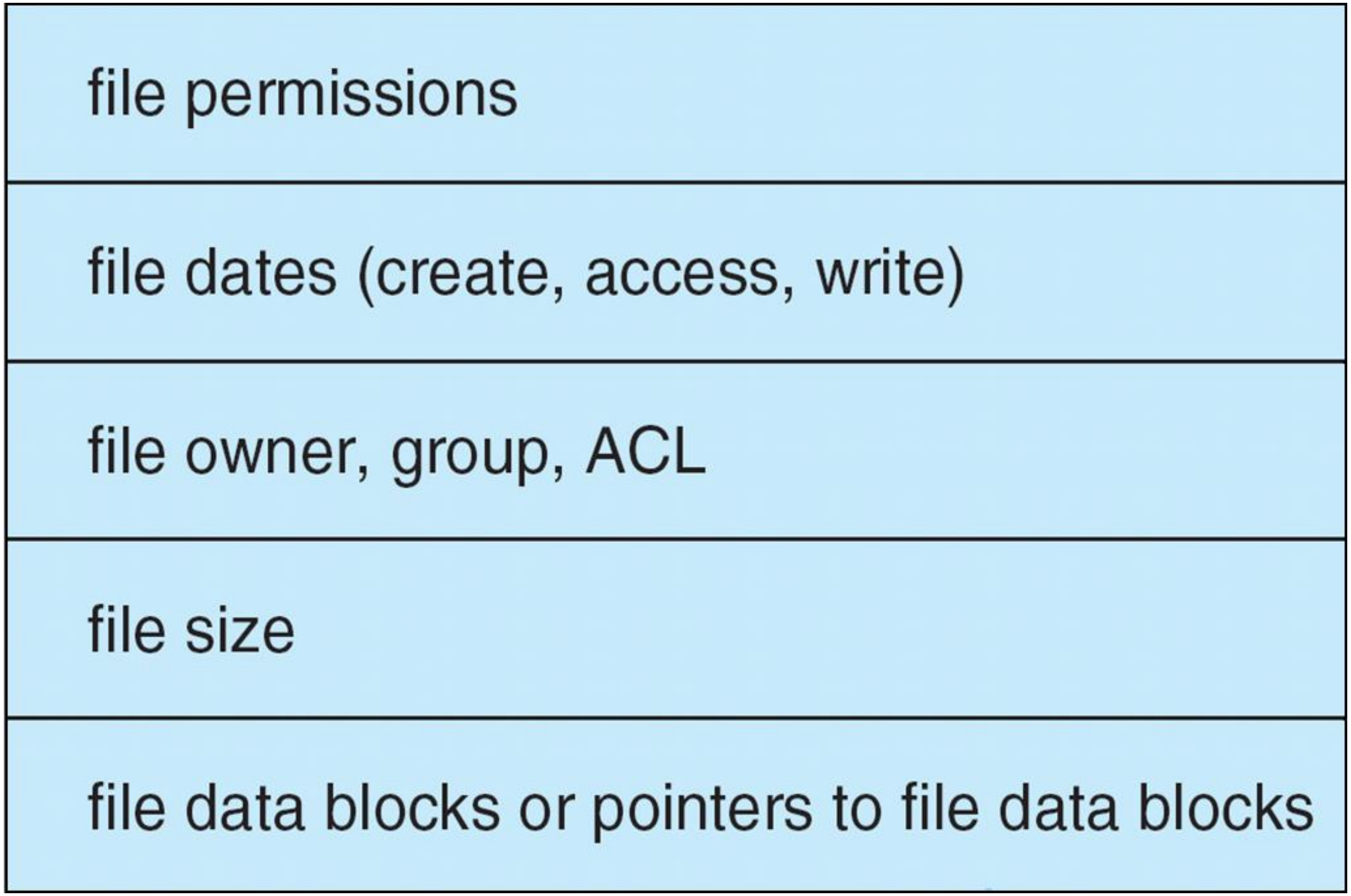
  - has a unique identifier number to allow association with a directory entry
  - In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file

- The in-memory information is used for both file-system management and performance improvement via caching
- The data are loaded at mount time, updated during file-system operations, and discarded at dismount
- Several types of structures may be included. They are:
  - An in-memory mount table contains information about each mounted volume

- An in-memory directory-structure cache holds the directory information of recently accessed directories
- The system-wide open-file table contains a copy of the FCB of each open file, as well as other information
- The per-process open-file table
  - contains pointers to the appropriate entries in the system-wide open-file table
  - as well as other information, for all files the process has open
- Buffers hold file-system blocks when they are being read from or written to a file system

- To create a new file, a process calls the logical file system
- The logical file system knows the format of the directory structures
- To create a new file, it allocates a new FCB
- The system then reads the appropriate directory into memory, updates it with the new file name and FCB and writes it back to the file system
- A typical FCB is shown in Figure in the next slide

File Control Block



file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

- Some operating systems, including UNIX, treat a directory exactly the same as a file
  - one with a “type” field indicating that it is a directory
- Other operating systems, including Windows, implement separate system calls for files and directories and treat directories as entities separate from files
- The logical file system can call the file-organization module
  - To map the directory I/O into storage block locations, which are passed on to the basic file system and I/O control system

Usage

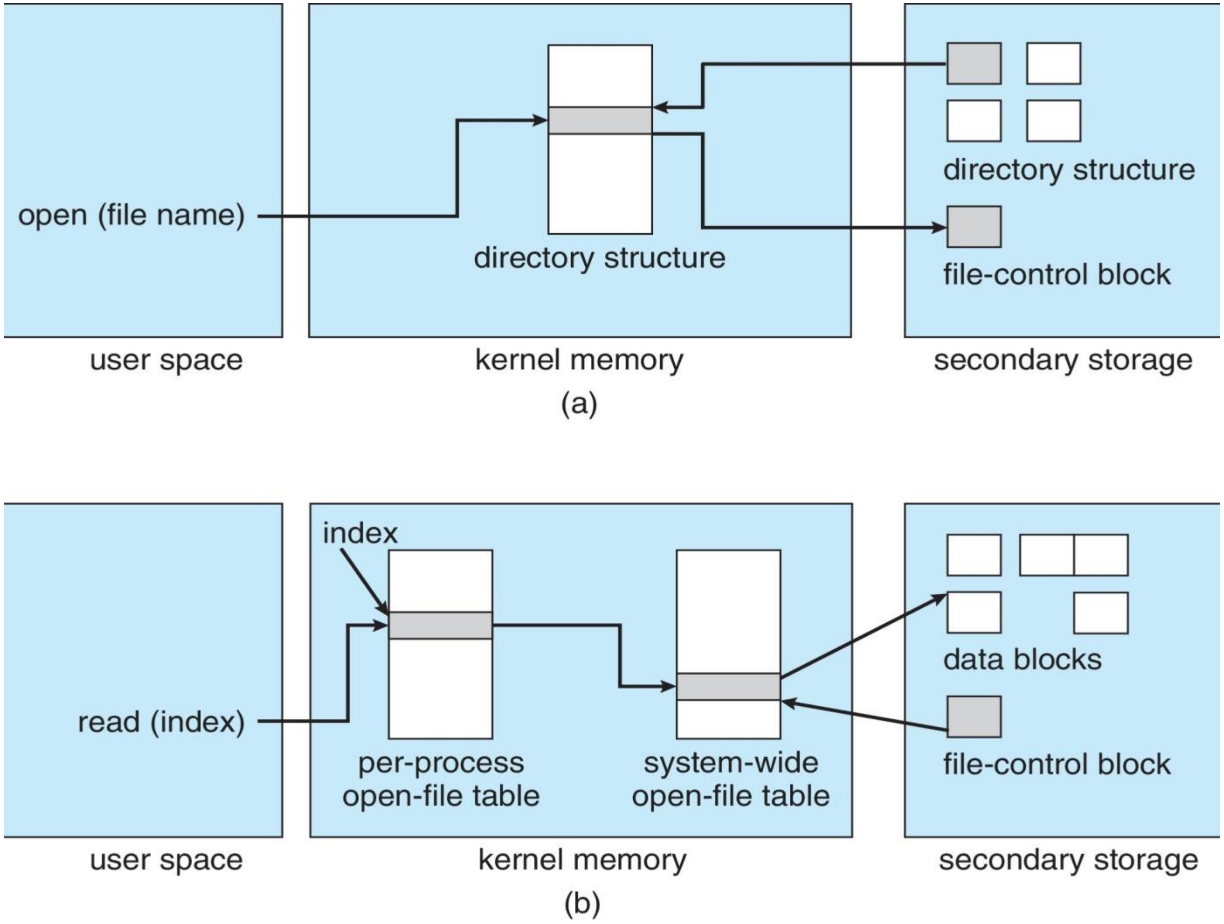
- Now that a file has been created, it can be used for I/O
- First, though, it must be opened
- The open() call passes a file name to the logical file system
- The open() system call first searches the system-wide open-file table to see if the file is already in use by another process
- If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table
- If the file is not already open, the directory structure is searched for the given file name

- Once the file is found, the FCB is copied into a system-wide open-file table in memory
- This table not only stores the FCB but also tracks the number of processes that have the file open
- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields
- The open() call returns a pointer to the appropriate entry in the per-process file-system table

- All file operations are then performed via this pointer
- The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk
- The name given to the entry varies
  - UNIX systems refer to it as a file descriptor
  - Windows refers to it as a file handle
- When a process closes the file, the per-process table entry is removed and the system-wide entry’s open count is decremented

- When all users that have opened the file close it
  - any updated metadata are copied back to the disk-based directory structure
  - the system-wide open-file table entry is removed
- The operating structures of a file-system implementation are summarized in Figure in the next slide

In-Memory File System Structures



Directory Implementation

- The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system
- Two ways to implement directory are:
  - Linear List
  - Hash Table

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks
- This method is simple to program but time-consuming to execute
- To create a new file
  - First search the directory to be sure that no existing file has the same name
  - Then, add a new entry at the end of the directory

- To delete a file
  - Search the directory for the named file
  - Then release the space allocated to it
- To reuse the directory entry
  - Mark the entry as unused by assigning it a special name, such as an all-blank name, assigning it an invalid inode number (such as 0), or by including a used–unused bit in each entry
  - Attach it to a list of free directory entries
  - Copy the last entry in the directory into the freed location and to decrease the length of the directory

- A linked list can also be used to decrease the time required to delete a file
- The real disadvantage of a linear list of directory entries is that finding a file requires a linear search
  - Directory information is used frequently and users will notice if access to it is slow
- In fact, many operating systems implement a software cache to store the most recently used directory information

Hash Table

- Another data structure used for a file directory is a hash table
- Here, a linear list stores the directory entries, but a hash data structure is also used
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list
- Therefore, it can greatly decrease the directory search time
- Insertion and deletion are also fairly straight forward

- The major difficulties with a hash table are
  - Its generally fixed size and
  - the dependence of the hash function on that size
- For example
  - Assume that we make a linear-probing hash table that holds 64 entries
  - The hash function converts file names into integers from 0 to 63 by using the remainder of a division by 64
  - If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries

  - As a result, we need a new hash function that must map file names to the range 0 to 127
  - we must reorganize the existing directory entries to reflect their new hash-function values
- Alternatively, we can use a chained-overflow hash table
  - Each hash entry can be a linked list instead of an individual value
  - We can resolve collisions by adding the new entry to the linked list
  - This method is likely to be much faster than a linear search through the entire directory

Allocation Methods

- The direct-access nature of secondary storage gives us flexibility in the implementation of files
- In almost every case, many files are stored on the same device
- The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly

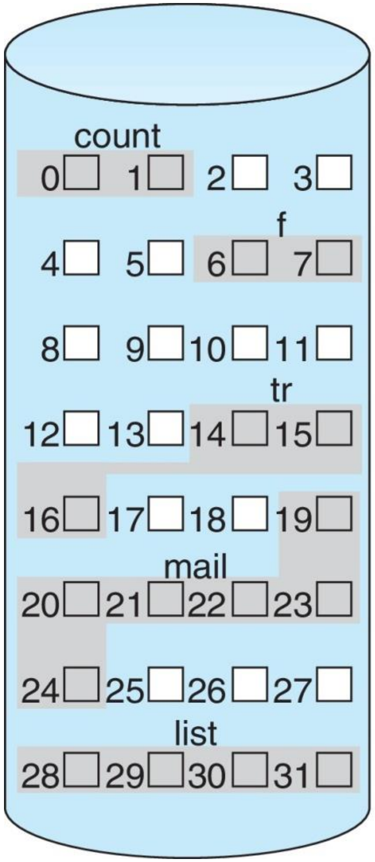
- Three major methods of allocating secondary storage space are in wide use:
  1. Contiguous allocation
  2. Linked allocation
  3. Indexed allocation
- Each method has advantages and disadvantages
- Although some systems support all three, it is more common for a system to use one method for all files within a file-system type

Contiguous Allocation

- Contiguous allocation requires that each file occupy a set of contiguous blocks on the device
- Device addresses define a linear ordering on the device
- With this ordering, assuming that only one job is accessing the device, 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

  - A cylinder is basically the set of all tracks that all the heads are currently located at
- Thus, for HDDs, the number of disk seeks required for accessing contiguously allocated files is minimal
  - Assuming blocks with close logical addresses are close physically
- Seek time is also minimal
- Contiguous allocation of a file is defined by the address of the first block and length (in block units) of the file
- If the file is n blocks long and starts at location b, then it occupies blocks b, b+1, b+2,..., b+n−1

- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file



directory

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

- Contiguous allocation is easy to implement but has limitations, and is therefore not used in modern file systems
- Accessing a file that has been allocated contiguously is easy
- For sequential access
  - the file system remembers the address of the last block referenced and when necessary, reads the next block
- For direct access
  - to block i of a file that starts at block b, we can immediately access block b + i
- Thus, both sequential and direct access can be supported by contiguous allocation

- Contiguous allocation suffers from the problem of external fragmentation
  - As files are allocated and deleted, the free storage 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, none of which is large enough to store the data

- One strategy for preventing loss of significant amounts of storage space to external fragmentation is to copy an entire file system onto another device
- The original device is then freed completely, creating one large contiguous free space
- We then copy the files back onto the original device by allocating contiguous space from this one large hole
- This scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem

- The cost of this compaction is time, however, and the cost can be particularly high for large storage devices
- 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
    - In general, however, the size of an output file may be difficult to estimate
    - Even if the total amount of space needed for a file is known in advance, prior allocation may be inefficient

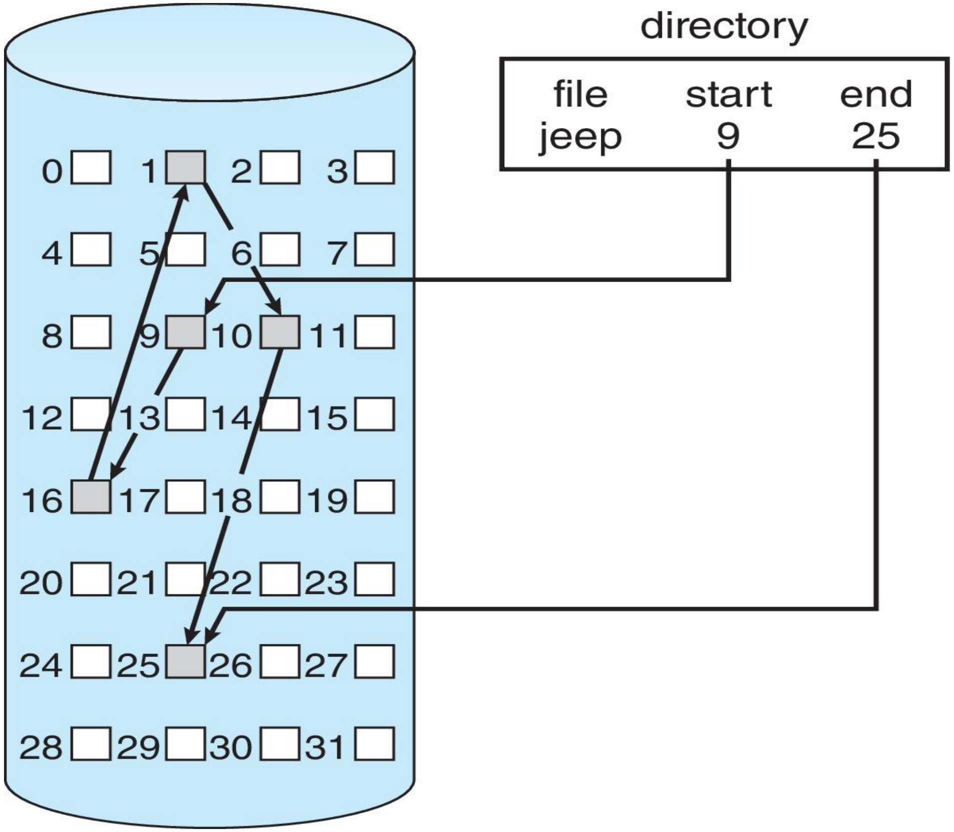
- To minimize these drawbacks, an operating system can use a modified contiguous-allocation scheme
  - Here, a contiguous chunk of space is allocated initially
  - Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an extent, is added
  - The location of a file’s blocks is then recorded as a location and a block count, plus a link to the first block of the next extent
  - On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect

- Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can become a problem as extents of varying sizes are allocated and deallocated

Linked Allocation

- Linked allocation solves all problems of contiguous allocation
- Here, each file is a linked list of storage blocks; the blocks may be scattered anywhere on the device
- The directory contains a pointer to the first and last blocks of the file
- 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

Linked Allocation



directory

file	start	end
jeep	9	25

- Each block contains a pointer to the next block
- These pointers are not made available to the user
- Thus, if each block is 512 bytes in size, and a block address(the pointer) requires 4 bytes, then the user sees blocks of 508 bytes
- To create a new file, simply create a new entry in the directory
- Here, each directory entry has a pointer to the first block of the file
- This pointer is initialized to null (the end-of-list pointer value) to signify an empty file

- The size field is also set to 0
- 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, simply read blocks by following the pointers from block to block
- There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request

- The size of a file need not be declared when the 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 does have disadvantages
  - The major problem is that it can be used effectively only for sequential-access files
  - To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block
  - Each access to a pointer requires a storage device read, and some require an HDD seek

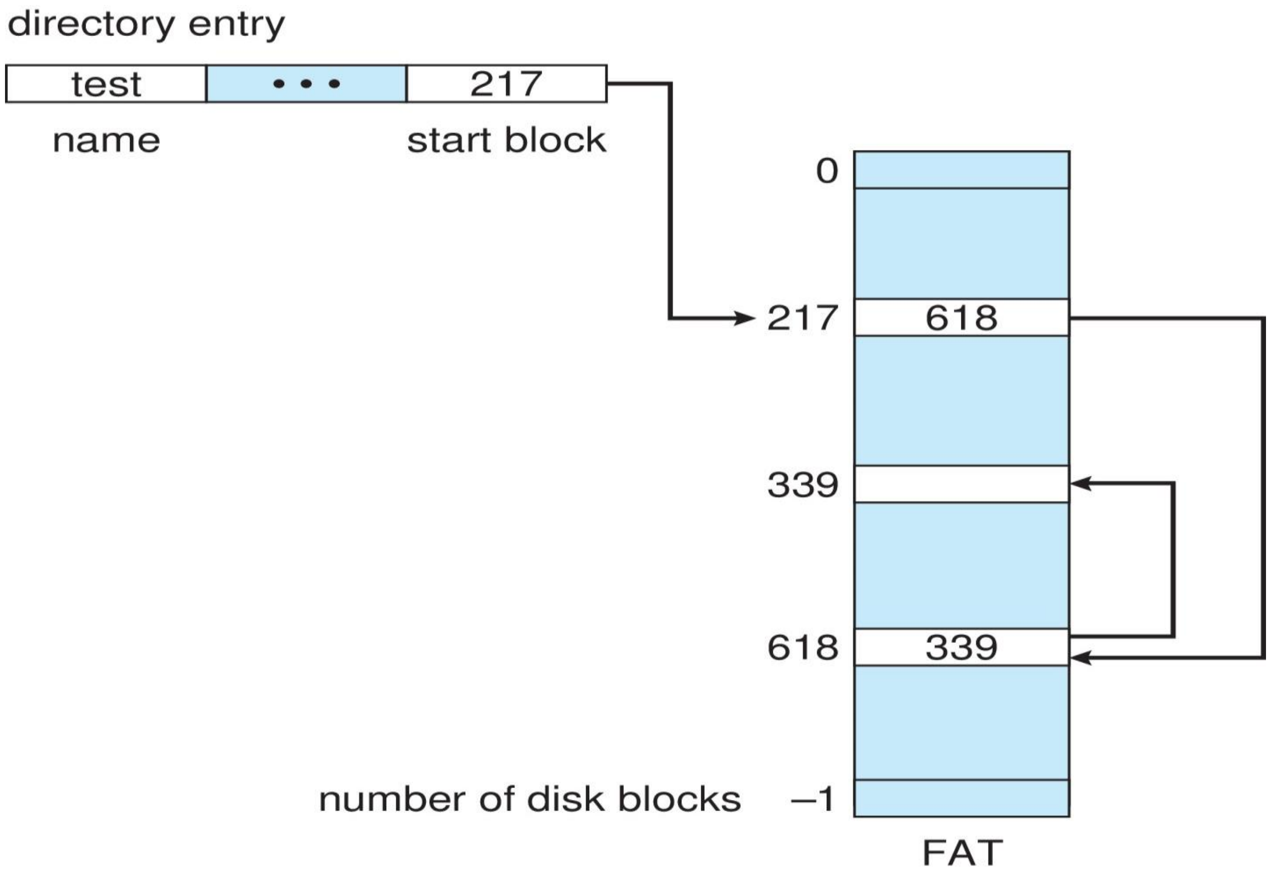
- Consequently, it is inefficient to support a direct-access capability for linked-allocation files
- Another disadvantage is the space required for the pointers
  - If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information
  - Each file requires slightly more space than it would otherwise
- 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
  - the files are linked together by pointers scattered all over the device, and consider what would happen if a pointer was lost or damaged

        - This error could in turn result in linking into the free-space list or into another file
    - One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block

- An important variation on linked allocation is the use of a file-allocation table (FAT)
    - This simple but efficient method of disk-space allocation was used by the MS-DOS operating system
    - A section of storage at the beginning of each volume is set aside to contain the table
    - The table has one entry for each block and is indexed by block number

- The directory entry contains the block number of the first block of the file
- The table entry indexed by that block number contains the block number of the next block in the file
- This chain continues until it reaches the last block, which has a special end-of-file value as the table entry
- An unused block is indicated by a table value of 0
- Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end of-file value with the address of the new block
- The 0 is then replaced with the end-of-file value

- An illustrative example is the FAT structure shown in Figure below for a file consisting of disk blocks 217, 618, and 339



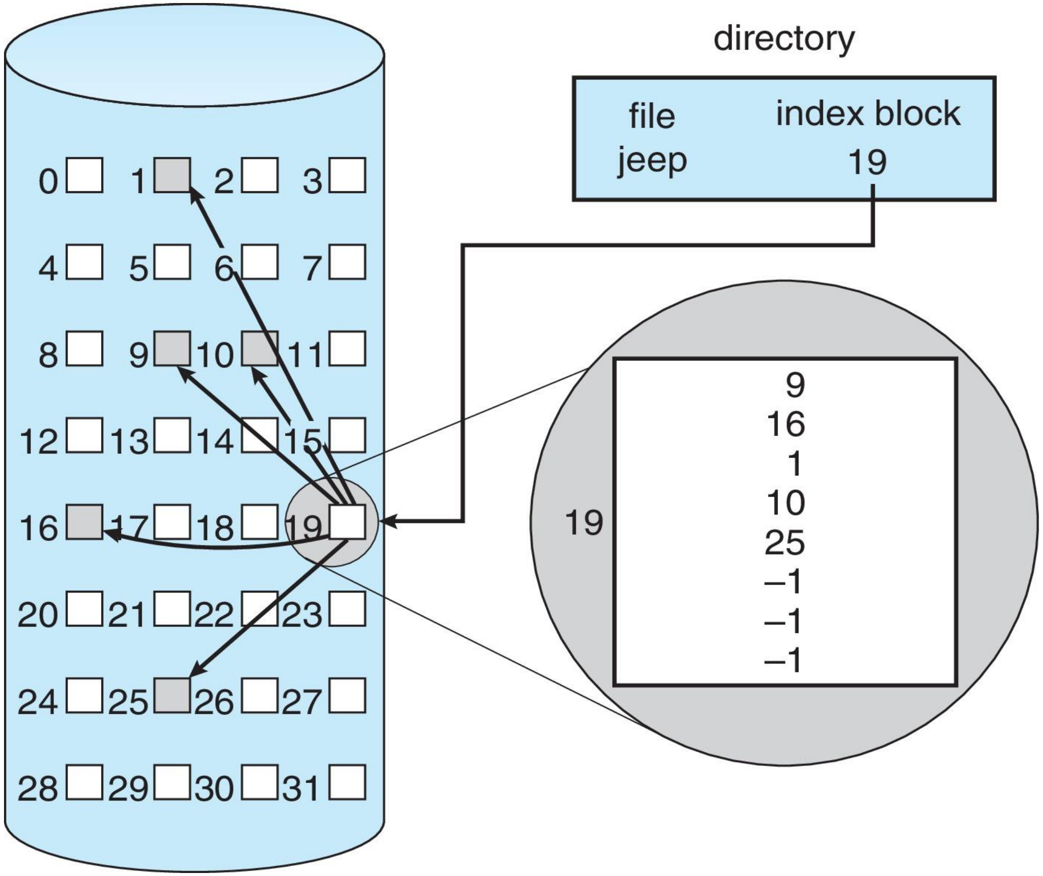
- The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached

Indexed Allocation

- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation
- However, 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
- 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 storage-block addresses
- The ith entry in the index block points to the ith block of the file
- The directory contains the address of the index block
- Figure in the next slide illustrates the concept
- To find and read the ith block, we use the pointer in the ith index-block entry

Example of indexed allocation



- When the file is created, all pointers in the index block are set to null
- When the ith block is first written, a block is obtained from the free-space manager, and its addressis put in the ith index-block entry
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the storage device can satisfy a request for more space

- 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 indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null
- 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,
  - a mechanism will have to be available to deal with this issue

- Mechanisms for this purpose include the following:
- Linked scheme
  - An index block is normally one storage block
  - Thus, it can be read and written directly by itself
  - 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 null (for a small file) or is a pointer to another index block (for a large file)

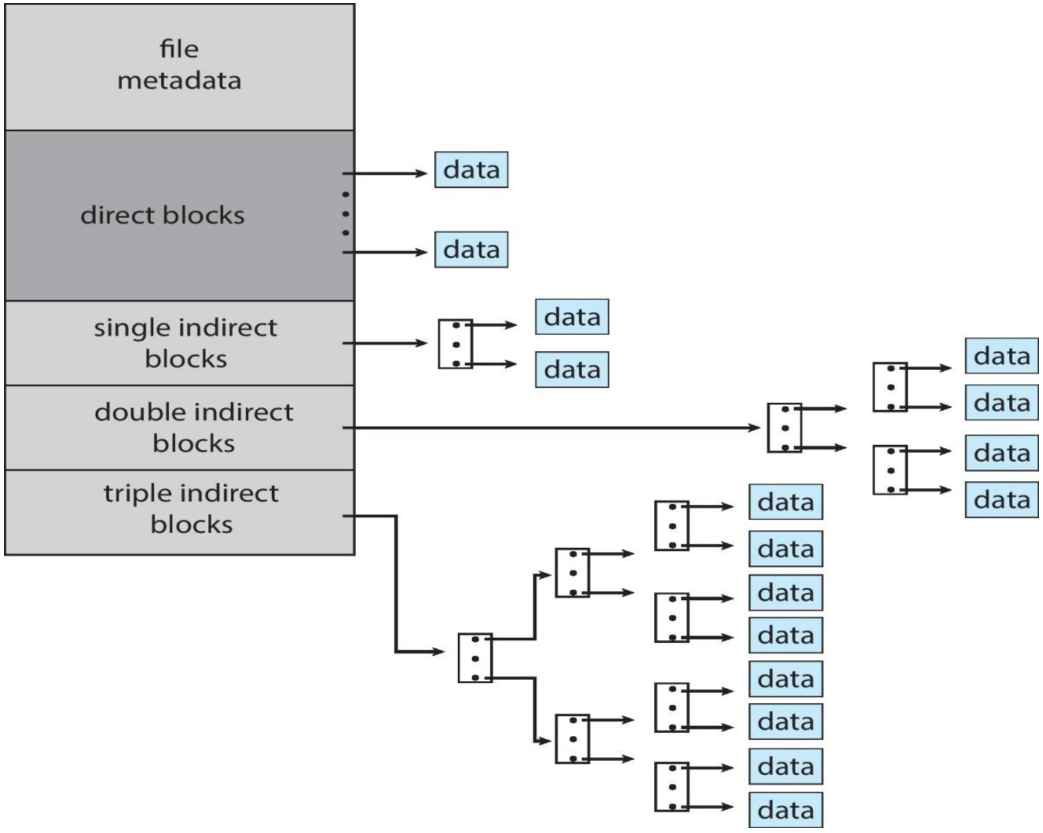
- Multilevel index
  - A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks
  - To access a block, the operating system uses the first-level index to find a second-level index block and then uses that 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

- Combined Scheme
  - Another alternative, used in UNIX-based file systems, 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
  - Thus, the data for small files (of no more than 12 blocks) do not need a separate index block
  - If the block size is 4KB, then up to 48KB 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
- UNIX inode is shown in the figure in the next slide

- A 32-bit file pointer reaches only 2^{32} bytes, or 4 GB
- Many UNIX and Linux implementations now support 64-bit file pointers

Combined Scheme: UNIX UFS



Performance

- The allocation methods that we have discussed vary in their storage efficiency and data-block access times
- Both are important criteria in selecting the proper method or methods for an operating system to implement
- Before selecting an allocation method, we need to determine how the systems will be used
- A system with mostly sequential access should not use the same method as a system with mostly random access

- For any type of access, contiguous allocation requires only one access to get a block
- Since we can easily keep the initial address of the file in memory, we can calculate immediately the address of the ith block (or the next block) and read it directly
- For linked allocation, we can also keep the address of the next block in memory and read it directly
- This method is fine for sequential access

- For direct access, however, an access to the ith block might require i block reads
- This problem indicates why linked allocation should not be used for an application requiring direct access
- As a result, some systems support direct-access files by using contiguous allocation and sequential-access files by using linked allocation
- For these systems, the type of access to be made must be declared when the file is created

- A file created for sequential access will be linked and cannot be used for direct access
- A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created
- In this case, the operating system must have appropriate data structures and algorithms to support both allocation methods

- Indexed allocation is more complex
    - If the index block is already in memory, then the access can be made directly
    - However, keeping the index block in memory requires considerable space
    - If this memory space is not available, then we may have to read first the index block and then the desired data block
    - For a two-level index, two index-block reads might be necessary
    - For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks before the needed data block finally could be read

- Thus, the performance of indexed allocation depends on the index structure, on the size of the file, and on the position of the block desired
- Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks) and automatically switching to an indexed allocation if the file grows large
- For NVM devices, there are no disk head seeks, so different algorithms and optimizations are needed

- Using an old algorithm that spends many CPU cycles trying to avoid an on existent head movement would be very inefficient
- Existing file systems are being modified and new ones being created to attain maximum performance from NVM storage devices

Free space management

- Since storage space is limited, we need to reuse the space from deleted files for new files, if possible
- To keep track of free disk space, the system maintains a free-space list
- The free-space list records all free device blocks i.e., those not allocated to some file or directory
- To create a file, we search the free-space list for the required amount of space and allocate that space to the new file

- This space is then removed from the free-space list
- When a file is deleted, its space is added to the free-space list
- The free-space list, despite its name, is not necessarily implemented as a list
- Different approaches for free space list implementation are:
  - Bit vector
  - Linked List
  - Grouping
  - Counting
  - Space Maps
  - TRIMing unused blocks

Bit Vector

- Frequently, the free-space list is implemented as a bitmap 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 0
- 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 bitmap would be

  001111001111110001100000011100000 ...

- The main advantage of this approach is
  - its relative simplicity
  - its efficiency in finding the first free block or n consecutive free blocks on the disk
- Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose

- One technique for finding the first free block on a system that uses a bit vector to allocate space is
  - to sequentially check each word in the bitmap to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks
  - The first non-0 word is scanned for the first 1 bit, which is the location of the first free block
  - The calculation of the block number is
  - (number of bits per word)×(number of 0-value words) + offset of first 1 bit

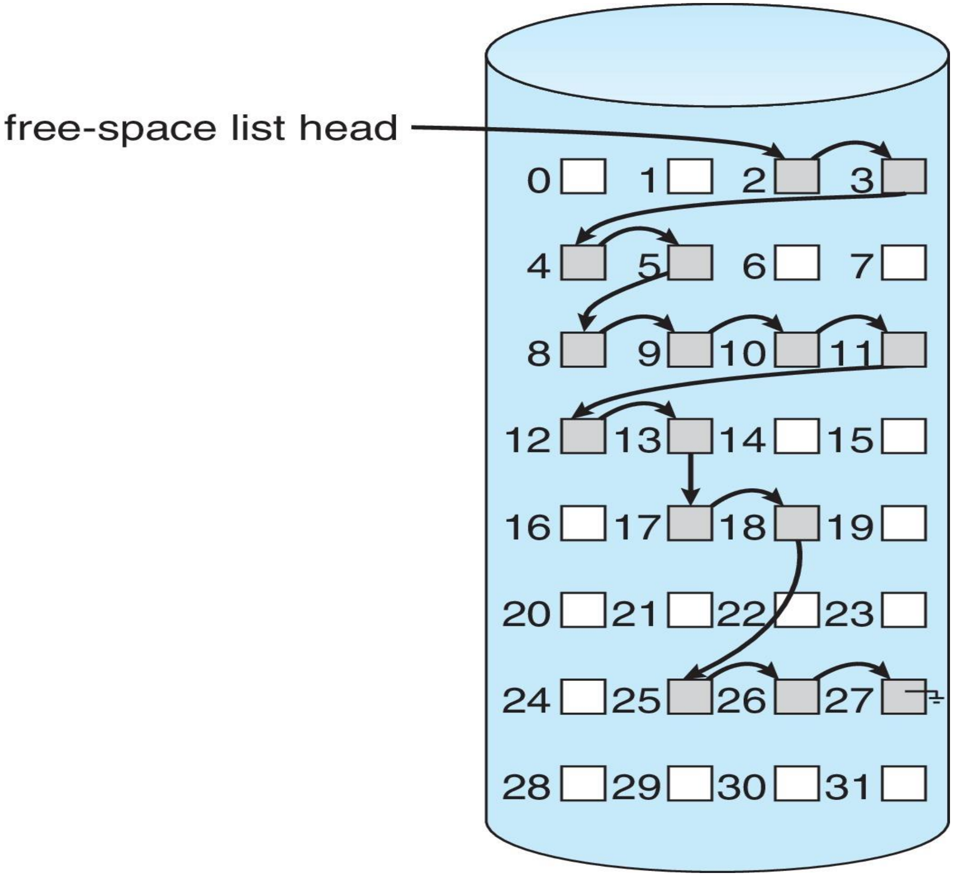
- Bit vectors are inefficient unless the entire vector is kept in main memory
- Keeping it in main memory is possible for smaller devices but not necessarily for larger ones

Linked List

- This approach links together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory
- This first block contains a pointer to the next free block and so on
- For example, again 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

- In this situation, 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 (Figure in the next slide)
- This scheme is not efficient
  - to traverse the list, we must read each block, which requires substantial I/O time on HDDs
- Fortunately, however, traversing the free list is not a frequent action

Linked free space list on disk



- Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used
- The FAT method incorporates free-block accounting into the allocation data structure

Grouping

- A modification of the free-list approach 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
- The addresses of a large number of free blocks can now be found quickly
    - Unlike the situation when the standard linked-list approach is used

Counting

- Is another approach which takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously
  - particularly when space is allocated with the contiguous-allocation algorithm or through clustering
- Thus, rather than keeping a list of n free block addresses, we can keep
  - the address of the first free block and the number(n) of free contiguous blocks that follow the first block

- Each entry in the free-space list then consists of a device address and a count
- These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion

Space Maps

- Oracle’s ZFS file system was designed to encompass huge numbers of files, directories, and even file systems
- ZFS file system is found in Solaris and some other operating systems
- In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures
- First, ZFS creates metaslabs to divide the space on the device into chunks of manageable size

- A given volume may contain hundreds of metaslabs
- Each metaslab has an associated space map
- ZFS uses the counting algorithm to store information about free blocks
- The space map is a log of all block activity (allocating and freeing), in time order, in counting format
- When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure indexed by offset, and replays the log into that structure

- The in-memory space map is then an accurate representation of the allocated and free space in the metaslab
- ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry
- Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS

TRIMing Unused Blocks

- HDDs and other storage media that allow blocks to be overwritten for updates need only the free list for managing free space
- Blocks do not need to be treated specially when freed
- A freed block typically keeps its data until the data are overwritten when the block is next allocated
- Storage devices that do not allow overwrite, such as NVM flash-based storage devices, suffer badly when these same algorithms are applied

- A new mechanism is needed to allow the file system to inform the storage device that a page is free and can be considered for erasure once the block containing the page is entirely free
- That mechanism varies based on the storage controller
  - For ATA -attached drives, it is TRIM ,
  - For NVM e-based storage, it is the *unallocate* command
- Whatever the specific controller command, this mechanism keeps storage space available for writing
-

- Without such a capability, the storage device gets full and needs garbage collection and block erasure, leading to decreases in storage I/O write performance
- With the TRIM mechanism and similar capabilities,
  - the garbage collection and
  - erase steps can occur before the device is nearly full, allowing the device to provide more consistent performance