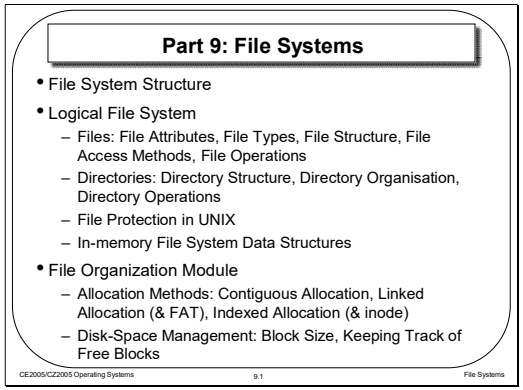
Slide 1

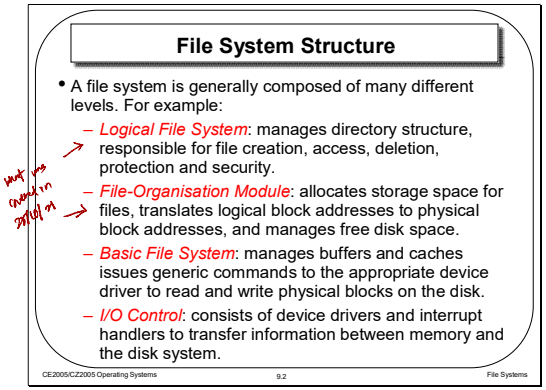



From the user's point of view, one of the most important parts of an operating system is the file system. The file system provides the resource abstractions typically associated with secondary storage. The file system permits users to create data collections, called files, with the following properties:

• **Long-term existence: Files are stored on disk or other secondary storage and** do not disappear when a user logs off.

• **Sharable between processes: Files have names and can have associated access** permissions that permit controlled sharing.

Slide 2




File system is typically organized in a layered structure.

At the top is the logical file system which is responsible for file creation, file access (like read & write), and file deletion. It manages metadata information. Metadata includes all of the information about a file except the actual data the file contains (as explained in the next slide). It also manages the directory structure to provide the file organization module with the information it needs.

File organization module allocates storage space files. It knows where each logical block file is located on the disk and about files and can translate logical block addresses to physical block addresses for the basic file system to perform the actual data transfer. The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Basic file system manages buffers and caches and issues generic commands to the appropriate device driver to read and write physical blocks on the disk.

I/O control consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.

[ Logical file system and file organization module are covered under this part of lectures (this week).
Basic file system and I/O control are covered under I/O management (next week) ]

## File Attributes

- A file may have the following attributes
  - *Name* - only information kept in human-readable form.
  - *Type* - needed for systems that support different types.
  - *Location* - pointer to file location on device.
  - *Size* - current file size
  - Protection - controls who can do reading, writing, executing.
  - *Time, Date, and User Identification* - data for protection, security, and usage monitoring.
- These information about files are kept in the directory structure, which is maintained on the disk.

CE2005/CZ2005 Operating Systems 9.3 File Systems

A file is a named collection of related information that is recorded on secondary storage. A file's attributes vary from one operating system to another but typically consist of Name, Type, Location, Size, Protection, Time, Date, and User Identification etc.

Collectively the file attributes define the meta data of a file (not the actual data the file contains). The information about all files is kept in the directory structure. Because directories, like files, must be nonvolatile, they must be stored on the disk and brought into memory as needed.

one entry for every file

Slide 4

## File Types

- A file has a contiguous logical address space which can store many different types of information:

| File Type | Usual extension | Function |
|---|---|---|
| Executable | exe, com, bin or none | ready-to-run machine-language program |
| Object | obj, o | complied, machine language, not linked |
| Source code | c, p, pas, 177, asm, a | source code in various languages |
| Batch | bat, sh | commands to the command interpreter |
| Text | txt, doc | textual data documents |
| Word processor | wp, tex, rrf, etc. | various word-processor formats |
| Library | lib, a | libraries of routines |
| Print or view | ps, dvi, gif | ASCII or binary file |
| Archive | arc, zip, tar | related files grouped into one file, sometimes compressed. |

CE2005/CZ2005 Operating Systems 9.4 File Systems

Many different types of information may be stored in a file, e.g., source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

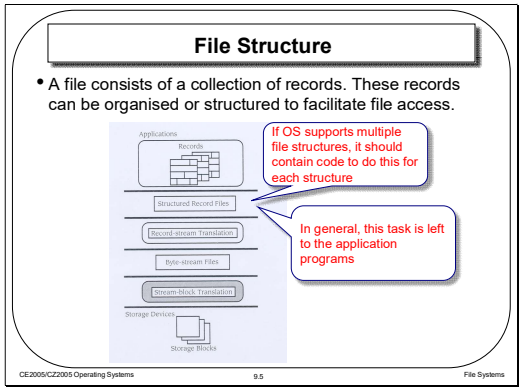
A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-a name and an extension. In this way, the user and the operating system can tell from the ~~name~~ extension alone what the type of a file is (for example, readme.txt, hello_world.exe). The extension is also used to indicate the type of operations that can be done on that file (e.g., text file can be read by a text editor and executable file can be executed by OS).

The file structure can be interpreted (and understood) by the corresponding application program (e.g., html file/web browser, ppt file/PowerPoint)

file may have its own internal structure

can only be opened in

Slide 5




The term file structure refers to the logical structuring of the records as determined by the way in which they are accessed (by application programs). However, the physical organization of the file on secondary storage depends on the blocking strategy and the file allocation strategy, issues dealt with later in this chapter.

A file has a certain defined structure which depends on its type. A file consists of a collection of records. These records can be organised or structured to facilitate file access. For example, a text file is a sequence of characters organized into lines. An object file is a sequence of bytes organized into blocks that can only be understood by the system's linker. A database file is organized as a sequence of records that can only be interpreted by the corresponding database system.

To store a file physically on a secondary storage device, then the following two operations need to be performed: record-stream translation (map logical file structure onto a contiguous logical address space, i.e., sequence of bytes) and stream-block translation (divide the sequence of bytes into blocks). To retrieve a file from the storage device, the reverse operations need to be performed.

This point brings us to one of the disadvantages of having the operating system support multiple file structures: it makes the operating system large and cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, it may be necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

In general, the file structure can be interpreted (and understood) by the corresponding application program.

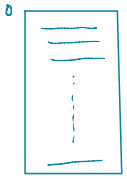
OS simply assume file is unstructured

Slide 6

## File Structure (Cont.)

- Unstructured: Sequence of Bytes.
  - A file is a stream of bytes. Each byte is individually addressable from the beginning of the file.
  - Used by UNIX and MSDOS (and **assumed in the following discussions**)






But, many OS (such as Unix) regards file as unstructured, that is, assuming logically a file contains a sequence of bytes.

Slide 7

## File Access Methods

- Sequential Access: Information in a file is processed in order from the beginning of the file, one byte after the other.
- Direct Access: Bytes of a file can be read in any order (by referencing byte number).



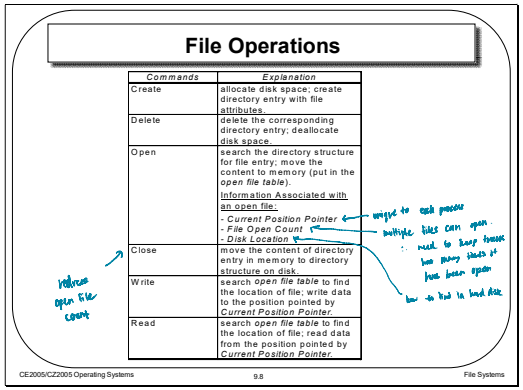
Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

The simplest access method is Sequential Access: Information in a file is processed in order from the beginning of the file, one byte after the other.

Another method is Direct Access: Bytes of a file can be read in any order (by referencing byte number). Direct access allows to access a specific location (i.e., byte) of a file directly.

* transferred through blocks

Slide 8




Any file system provides not only a means to store data organized as files, but a collection of functions that can be performed on files. Typical operations include the following:

• **Create:** Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file on next Thursday. Second, an entry for the new file must be made in the directory.

• **Delete:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space and erase the directory entry.

• **Open:** An existing file is declared to be “opened” by a process, allowing the process to perform functions on the file.

To avoid constant searching of directory, many systems require that an open () system call be made before a file is read or write. The operating system keeps a small table, called open file table, which caches meta information about all open files. The open() system call typically returns a pointer, called file descriptor, to the entry in the open-file table. This file descriptor, not the actual file name, is used in read/write operations.

There are several pieces of information associated with an open file:

Current position pointer: This is used to keep track the last read-write location of the file. This pointer is unique to each process operating on the file.

File open count: This is needed because multiple processes may have opened a file. An entry in the open file table is only removed when the file is closed by all the processed.

Disk location: This is used to locate data blocks of the file on the disk.

• **Close:** When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table.

• **Write:** Takes two parameters: the file descriptor and the data to be written to the file. Given the file descriptor the system searches the open file table to find the location of the file and write data to the position pointed by the current position pointer.

• **Read**: Also takes two parameters the file descriptor and amount of data to be read from the file. Given the file descriptor the system searches the open file table to find the location of the file and read data from the position pointed by the current position pointer.

OS provides all these operations through system calls.

Slide 9

## Directory Structure

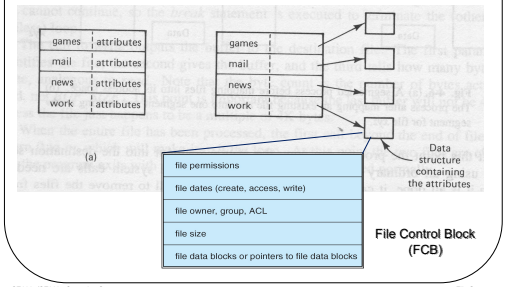
- A directory typically contains a number of entries, one per file. Both the directory and the files reside on disk.
- A directory can be structured in two ways:
  - (a) each entry contains a file name and other attributes; or
  - (b) each entry contains a file name and a pointer to another data structure where file attributes can be found.



A directory contains multiple entries, one for each file. Internally, a directory can be structured in two ways as illustrated in the next slide.

Slide 10

## Directory Structure (Cont.)






each entry contains entire meta data of file

A directory typically contains a number of entries, one per file. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

Diagram here shows how internally entries in a directory can be structured.

A directory can be structured in two ways:

(a) each entry contains a file name and other attributes; or

(b) each entry contains a file name and a pointer to another data structure where file attributes can be found.

Slide 11

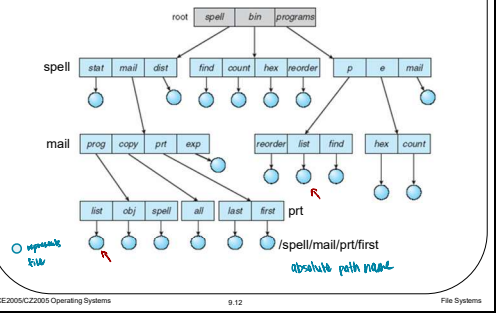
## Directory Structure (Cont.)

- A directory can be implemented using:
  - *Linear List*: Directory entries are arranged as linear list
    - simple to implement
    - time-consuming to execute: require linear search to find a particular entry
  - *Hash Table*: Directory entries are arranged as linear list with a hash table to facilitate file look-up.
    - decreases directory search time
    - collisions: situations where two file names hash to the same location (solution: each hash table entry contains a list of directory entries)



Slide 12

## Directory Organization






Files are grouped into directories and directories can be organized hierarchically to facilitate quick search of a file and convenience in naming files (different files can have the same name as long as they are under different directories).

A common way to organize directories is to use tree-structure as shown in the diagram here. The use of a tree-structured directory minimizes the difficulty in assigning unique names. Any file in the system can be located by following a path from the root or master directory down various branches until the file is reached. The series of directory names, culminating in the file name itself, constitutes a **pathname** for the file. Note that under the tree structure, it is perfectly acceptable to have several files with the same file name, as long as they have unique pathnames. For example, there are two files with name “list”, but they are under different directories and have different path names (“/spell/mail/ptr/list” and “/programs/list”).

Although the pathname facilitates the selection of file names, it would be awkward for a user to have to spell out the entire pathname every time a reference is made to a file. Typically, an interactive user or a process has associated with it a current directory, often referred to as the **working directory.** Files are then referenced relative to the working directory. For example, assuming the current working directory is “mail”, then the relative path name for the file “/spell/mail/ptr/list” will be “ptr/list”.

Slide 13

## Directory Organization (Cont.)

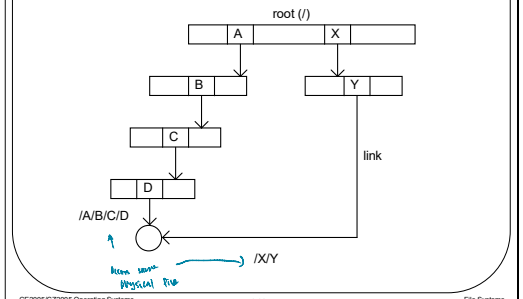
- Path Name
  - *Absolute Path Name*: begins at the root and follows a path down to the specific file, e.g., /spell/mail/prt/first
  - *Relative Path Name*: Defines a path from the current directory, e.g.,
    - current directory is: /spell/mail
    - relative path name for the above file is: prt/first
- Characteristics
  - *Efficient Searching*: File can be easily located according to the path name.
  - *Naming*: Files can have the same name under different directories.
  - *Grouping*: Files can be grouped logically according to their properties.



In summary, organizing directories into a tree, a file can be easily located according to the path name and files can have the same name as long as they are under different directories. A new directory can be always created according to the properties of files (e.g., files belonging to the same project).

Slide 14

## Acyclic-Graph Directory



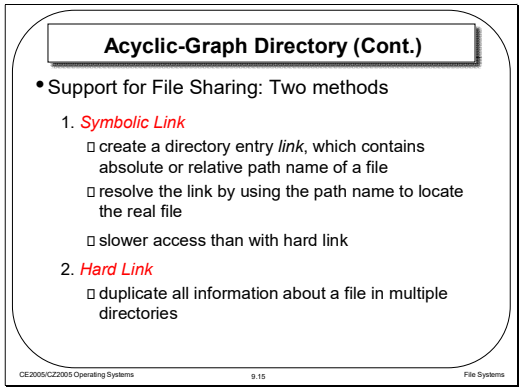


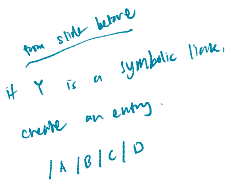

In a multiuser system, there is almost always a requirement for allowing files to be shared among a number of users. “Links” provide a convenient way for users to share the same physical file.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other user.

With links, a tree becomes an acyclic graph (that is, a graph without cycles). The acyclic graph is a natural generalization of the tree-structured directory scheme.

Slide 15

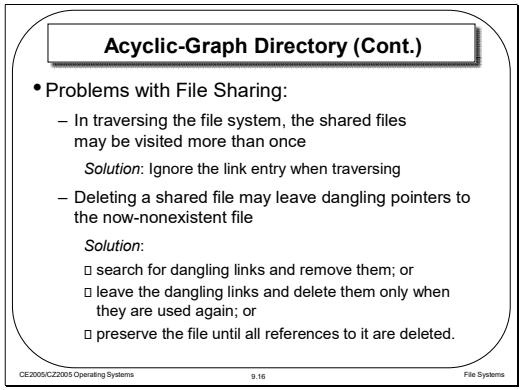



A link can be either a symbolic link or a hard link. A symbolic link is effectively a pointer to another file. It is implemented as a special directory entry contains either an absolute or a relative path name of the shared file. When a reference to the symbolic link is made, the link is resolved by following the path name to locate the directory entry of the shared file. In this case, the shared file is accessed indirectly. A symbolic link is effectively an indirect pointer.

A hard link is created by duplicating directory entry of a file. Thus, both entries are identical. Hence, unlike symbolic link, a shared file can be accessed directly and accessing a shared file takes less time than using symbolic link. However, a major problem with duplicate directory entries is maintaining consistency when a file is modified.

Slide 16




An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully.

A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. In traversing the file system, the shared files may be visited more than once because of the links. Solution to this problem is to ignore the link entry when traversing.

Deleting a shared file may leave dangling links to the file that is no longer existing. Solution to the second problem -> further discussion in tutorial.

Slide 17

## Directory Operations

| *Commands* | *Explanation* |
|---|---|
| Create | create a directory.<br>In UNIX, two entries "." and ".." are automatically added when a directory is created. "." refers to the *current directory*; and ".." refers to its *parent*. |
| Delete | delete a directory.<br>Only empty directory can be deleted (directory containing only "." and ".." is considered empty). |
| List | list all files (directories) and their contents of the directory entry in a directory |
| Search | search directory structure to find the entry for a particular file. |
| Traverse | access every directory and every file within a directory structure. |



•**Create**: Create a new directory. An entry is added to the parent directory. (Similar to create a file.)

•**Delete**: Delete a directory. An entry is deleted from the parent directory. (Similar to delete a file.)

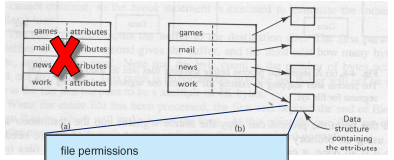
• **List directory:** All or a portion of the directory may be requested. Generally, this request is made by a user and results in a listing of all files owned by that user, plus some of the attributes of each file (e.g., type, access control information, usage information).

• **Search:** When a user or application references a file, the directory must be searched to find the entry corresponding to that file.

•**Traverse the file system**: We may wish to access every directory and every file within a directory structure (e.g., save the contents and structure of the entire file system)

Slide 18

## In-memory File System Data Structure



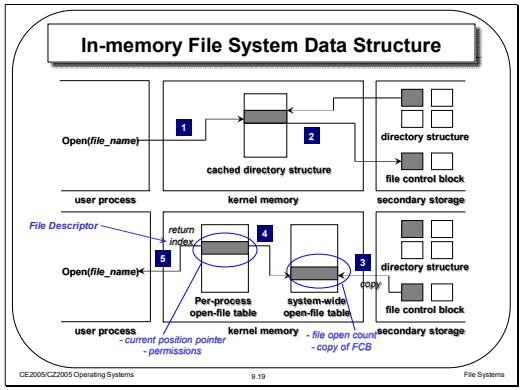

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

**File Control Block**



Unix inode, to be introduced later, can be considered as a file control block

Slide 19




get cache copy instead of going back to hard disk → more efficient

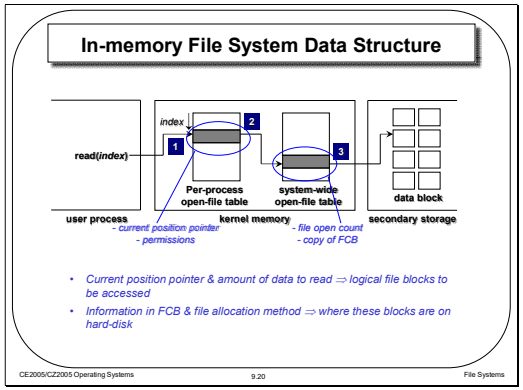
To avoid constant searching of directory, many systems require that an open () system call be made before a file is read or write. Given the file name, using cashed directory structure, OS will be able to locate the FCB of the file on the hard-disk.

The implementation of the open() and close() operations is more complicated in an environment where several processes may open the same file simultaneously. To deal with this, typically, the operating system uses two levels of open file tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. For instance, the current position pointer for each file is found here. Access rights to the file can also be included.

Each entry in the per-process table (open file) in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the cached copy of FCB (which includes location of the file on disk, access dates, and file size) and file open count. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an open() call on the same file, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. File open count keeps track how many processes have the file open. Each open() increases this count and each close() decreases this open count. When the file open count reaches zero, the file is no longer in use, and the file's entry is removed from the system-wide open-file table.

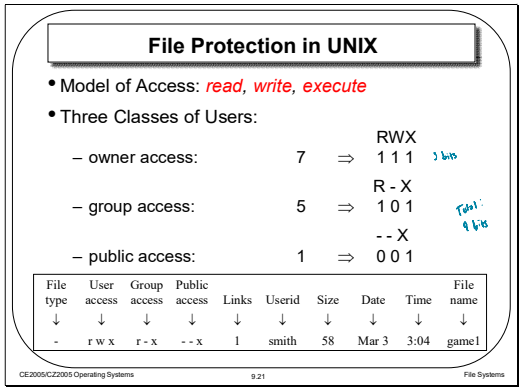
So, there is one entry for each file opened in the system-wide open table. Similarly, there is one entry for each file opened by a process in per-process open file table. open() system call returns the index of the entry in the per-process open-file table (an integer), which is called file descriptor.

Slide 20




open() system call returns an entry index of the per-process open file table. read() or write() system call takes this index as a parameter. The kernel will use it to locate the entry in the per-process open file table. From there, following the back-pointer, the kernel can then locate the FCB for the file in the system-wide open file table. The current position pointer in the entry of the per-process open file table together with the amount of data to read/write are used to determine the logical file blocks to be accessed. Information contained in the FCB together with the file allocation method used are used to determine where these file blocks are on hard-disk. (Question 3 of Tutorial 10 will explain this using an example.)

Slide 21




Other issues related to file sharing are access rights and the management of simultaneous access.

File owner/creator should be able to control: what can be done; and by whom

The file system should provide a number of options so that the way in which a particular file is accessed can be controlled. Typically, users or groups of users are granted certain access rights to a file.

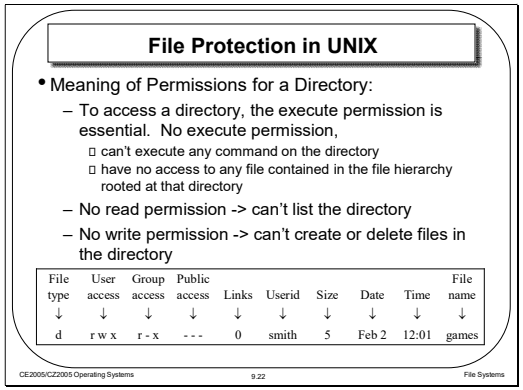
We illustrate the file protection using Unix as an example.

In Unix, each user has a primary group. When a file is created, it is initially associated with user's primary group. Groups are created by the system administrator.

There are three classes of users: owner, group, public.

For each class of user, there are three protection bits: read, write, and execute.

Slide 22



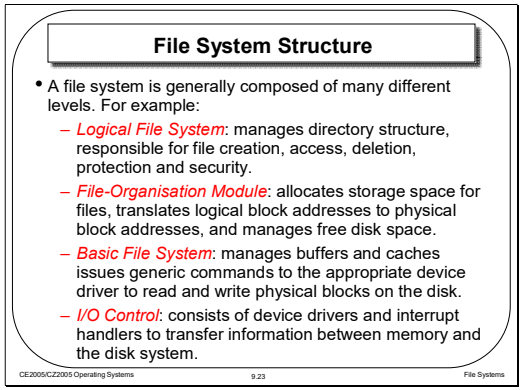

Like files, each directory file also has 9 protection bits.

Read permission means operation to list the content of a directory is permitted.

Write permission means operation to modify the content of a directory (e.g., delete a directory entry or insert an entry in the directory) is permitted.

However, to access a directory, the execute permission is essential. If there is no execute permission, read/write operation on the directory can't be performed even though read/write permission is set.

Slide 23

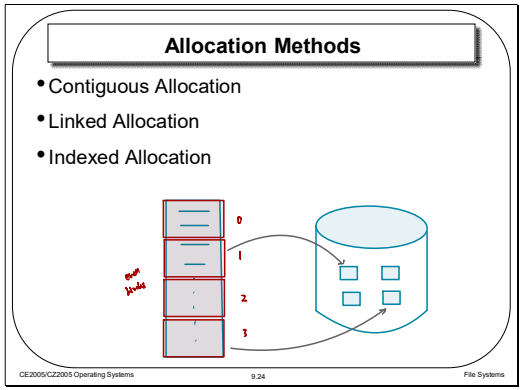



Typically, a secondary storage is a block-oriented device that consists of a collection of blocks. The operating system or file management system is responsible for allocating blocks to files. This raises two management issues. First, space on secondary storage must be allocated to files, and second, it is necessary to keep track of the space available for allocation.

[We will see that these two tasks are related; that is, the approach taken for file allocation may influence the approach taken for free space management. Further, we will see that there is an interaction between file structure and allocation policy.]

We begin this part of lecture by looking at alternatives for file allocation on a single disk. Then we look at the issue of free space management.

Slide 24

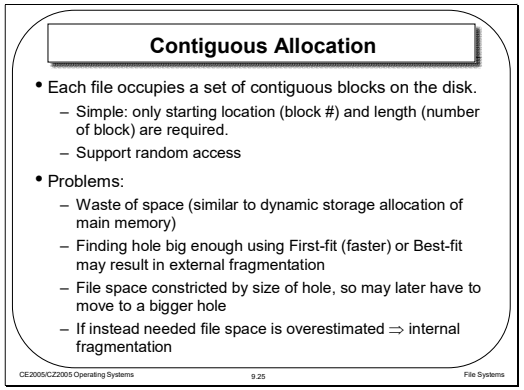



We assume a file is unstructured. A file can be viewed logically as a sequence of bytes of any length. However, a disk can only store data in fixed-size blocks, so the first step to store a file on a disk is to divide it into blocks. These blocks are called the logical blocks or data blocks. They are numbered with logical block numbers, starting from 0. Then, a file space allocation method is required to determine where these logical blocks can be stored on the disk.

Similar to memory allocation to process, disk space allocation to files can be either contiguous or non-contiguous.

We will cover three file allocation methods: contiguous allocation, linked allocation, and indexed allocation.

Slide 25



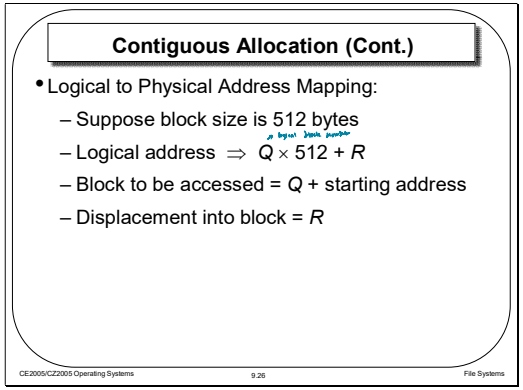

Contiguous allocation requires that each file occupies a set of contiguous blocks on the disk. If a file is n blocks long and starts at disk block b, then it occupies disk blocks b, b+1, b+2, …, b+n-1.

Contiguous allocation is the best from the point of view of sequential file. It is also easy to retrieve a specific block. For example, if a file starts at disk block *b*, and the logical block *i* of the file is wanted, its location on the disk is simply *b* + *i*.

Dynamic allocation policies, such as first fit, best fit and worst fit, can be used to find the required space (that is, a large enough hole) for the file on the disk. However, external fragmentation will occur. As files are created and deleted, the disk may become fragmented.

Another problem with contiguous allocation is to determine how much space is needed for a file. We may find that there is little space on both side of the file on the disk and hence the file cannot grow in its current place. To extend the file, we may have to move it to a larger hole.

On the other hand, if the space required by a file is over-estimated, that is, we allocate more space than what the file actually needs, this will result in part of the space assigned to the file is not really in use, resulting in internal fragmentation.




e.g.
file pointer : 4 bytes
file size: $2^{32}$ bytes

Recall, file position pointer specifies the byte position in a file to be accessed. Given a 4-byte file position pointer, it can be used to address a file consisting of 2^32 bytes.

To access a specific position (in terms of byte position) of a file for read or write, the following two steps need to be performed:

1. Determine the logical block where the position is.

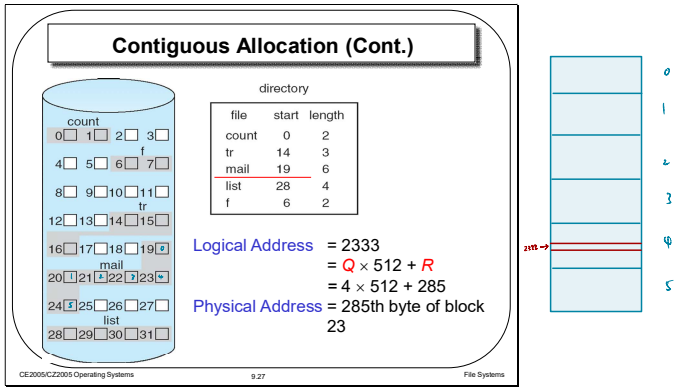
2. Determine the disk block where the logical block is on the disk.

Slide here shows given a position in byte (that is, logical address), how the logical block number and disk block number (that is, physical address) are determined.

The logical block number is determined by dividing logical address by the block size. Quotient is the logical block number.

Disk block number is determined by adding the logical block number to the starting disk block number for the file.

Slide 27




Now let's look at an example.

The diagram shows the current state of a disk. Shaded blocks are occupied.

The directory entry for each file indicates the address of the starting disk block and the length of the file.

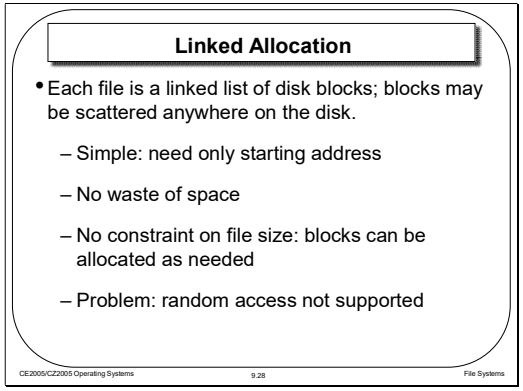
The example in the slide is for file "mail", the starting disk block number is 19 and there are 6 blocks in the file. So, logical block 0 of file "mail" is in disk block 19.

Given block size 512, logical address 2333 is in logical block 4. Since the starting disk block number is 19, the block that needs to be accessed is disk block 19+4=23.

↑ quotient

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For example, assume the last disk block accessed for file "mail" is block 20, for sequential access, the next disk block to be accessed will be block 21.

For direct access to logical block i of a file that starts at disk block b on the disk, we can immediately determine that the required logical block is at disk block b+i. As we just illustrated now, the logical block 4 of file "mail" can be directly located at disk block 19+4=23.

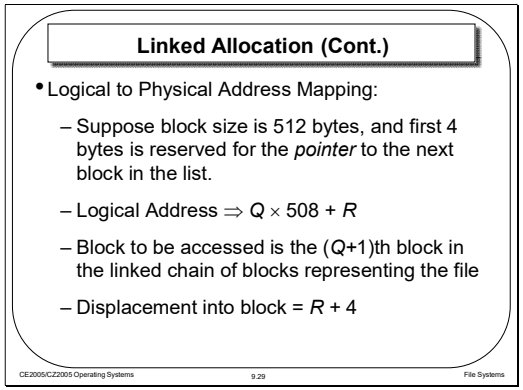



With linked allocation, all disk blocks allocated to a file are chained using a link list. Each block contains a pointer to the next block in the chain.

The selection of blocks to assign to a file is now a simple matter: Any free block can be used. So, there is no external fragmentation to worry about. There is no constraint on file size. Additional blocks can be added easily. However, the disk blocks allocated to a file may be scattered anywhere on the disk.

To access a specific logical block of a file requires tracing through the chain to the desired block. So, this allocation method is not suitable for random access.

Slide 29

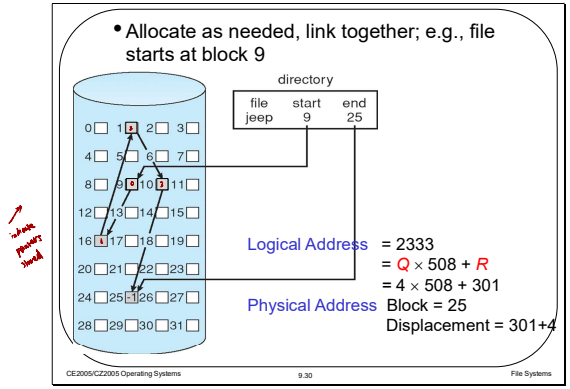



Similar to the contiguous allocation, to access a specific position (in terms of byte position) of a file for read or write, we need to first determine this position is in which logical block and then where the logical block is on the disk.

Assuming the block size is 512 bytes, since the first 4 bytes of each block is reserved for the *pointer* to the next block assigned to the file, the effective block size is 508. block size - pointer size = 512 - 4 = 508

The logical block number is determined by dividing logical address (that is, position in byte) by the effective block size. Quotient is the logical block number.

Slide 30



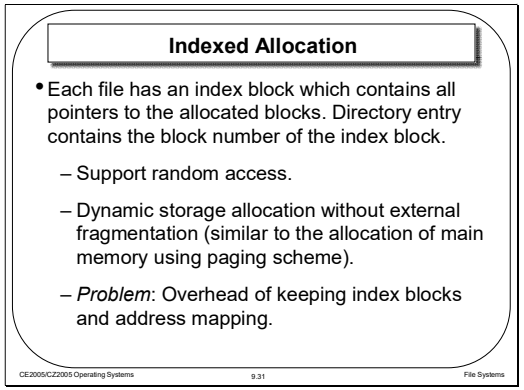

Using linked allocation, the directory entry contains a pointer to the first and last blocks of the file.

To access byte 2333 of file "jeep", we can determine that it is in logical block 4 (2333 divided by 508, quotient is 4).

To access logical block 4 of file "jeep", we need to locate where logical block 0 is on the disk from the directory entry and follow the links to locate the next logical block and so on until logical block 4 is reached. In this case, logical block 4 is in disk block 25.

In general, to find logical block i of a file, we must start at the beginning of that file and follow the links until we get to the required block. Each access to a link requires a disk read. So, locating logical block i, i disk I/Os are needed.

Slide 31




Unlike the linked allocation, **indexed allocation** puts all the pointers together into an **index block**.

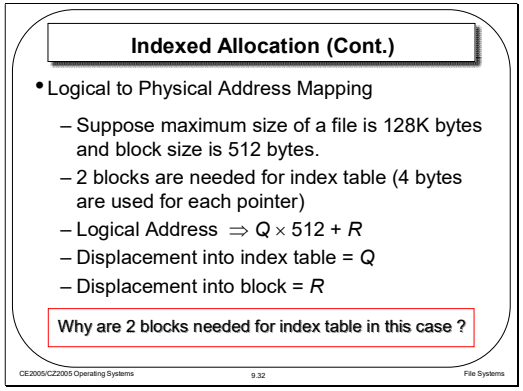
Each file has its own index block, which is an array of disk-block addresses (i.e., block pointers), one for each block of the file. The entry i in the index block points to the disk location of logical block i of the file.

[When the file is created, all pointers in the index block are set to null (that is "-1"). When logical block i is first written, a free disk block is obtained and its address is put in entry i in the index block. ]

This scheme supports random access. The disk location of a logical block can be easily determined by checking the corresponding entry in the index block.

Similar to the linked allocation scheme, any free disk block can be used. There is no external fragmentation. However, one consequence of this is that there is no accommodation of the principle of locality. Disk blocks assigned to a file can be anywhere on the disk. Thus, to access a sequence of blocks of a file, as in sequential processing, a series of accesses to different parts of the disk are required. In contrast, contiguous allocation observes a good locality since disk blocks assigned to a file are contiguous on the disk.

Another problem of the index allocation is the overhead of keeping index blocks which take up disk space.



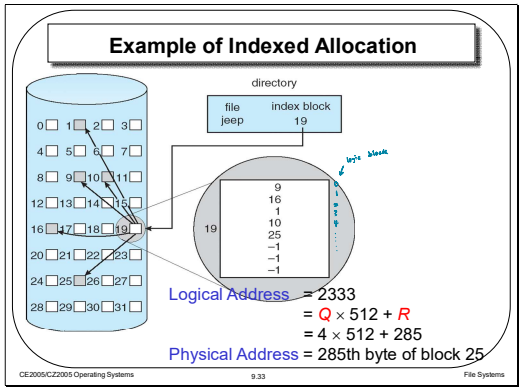

Given the maximum file size is 128K bytes, block size is 512 bytes, and each block pointer occupies 4 bytes, in this case two blocks are required to accommodate the index table. (number of blocks: 128K/512 = 2^8; size of the index table: 2^8 x 4 = 2^10)

Similarly, to access a specific position (in terms of byte position) of a file for read or write, we need to determine the logical block where the position is by dividing the position (that is, the logical address) by the block size.

---

4 byes block pointer -> 32 bits -> 2^32 blocks -> size of the disk 2^32 x 2^9 = 2^41 bytes

Slide 33




Using index allocation, the directory contains the address (i.e., disk block number) of the index block.

Assume that we need to allocate disk space for file “jeep” that requires five blocks. Disk block 19 is the index block of file “jeep”. The first five entries in the index block contain the disk block addresses where the five blocks of file “jeep” are on the disk.

So, logical block 0 of file “jeep” is at disk block 9, logical block 1 is at disk block 16, logical block 2 is at disk block 11, logical block 3 is at disk block 10, and logical block 4 is at disk block 25.

To find where logical block i of a file on the disk, we use the block pointer (that is, disk block number) in the entry i of the index block. For example, to access byte 2333 of the file “jeep”, we can determine that it is in logical block 4. From entry 4 of the index block, we know the logical block 4 is in disk block 25.

Slide 34

## Indexed Allocation: UNIX inode

- For each file or directory, there is an *inode* (index block).
- The inode contains:
  - *file attributes*
  - 12 pointers point to *direct blocks* (data blocks)
  - 3 pointers point to *indirect blocks* (index blocks). They are:
    - *single indirect*;
    - *double indirect*; and
    - *triple indirect*

CE2005/CZ2005 Operating Systems 9.34 File Systems

A variation of index allocation is the inode scheme used by Unix.

An inode can be considered as a file control block (FCB). There is one inode per file and a directory entry contains the file name and its corresponding inode number.

An inode contains file attributes and pointers to where the logical blocks are on the disk. There are 12 direct pointers and 3 indirect pointers. The first 12 pointers point directly to where the first 12 logical blocks are on the disk.

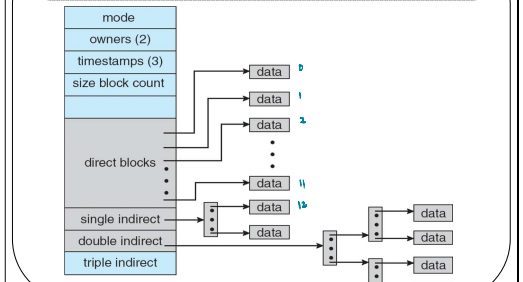
The single indirect pointer points to an indirect block (i.e., an index block), which contains the addresses (or pointers) of logical blocks on the disk.

The double indirect pointer points to an indirect block, which contains the addresses of blocks that in turn contain pointers to the logical blocks on the disk. In this case, indirection is two-level.

For trip indirect pointer, indirection is three-level.

Slide 35

## inode (Cont.)

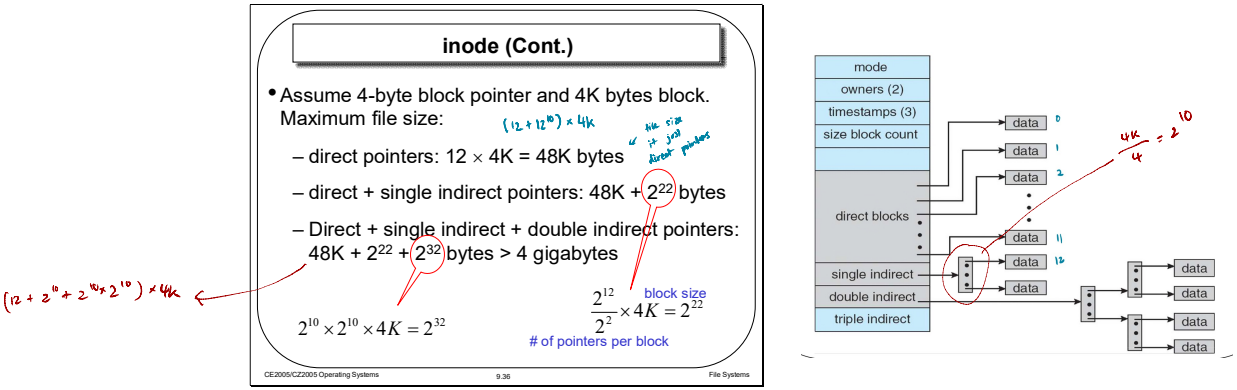



CE2005/CZ2005 Operating Systems 9.35 File Systems

Diagram here graphically shows how an inode looks like.

So, for small files which has less than 12 blocks, indirect blocks need not to be used. All logical blocks can be located using direct pointers in this case. In other words, to access logical block 0 to logical block 11 of a file, one of the direct pointers will be used. For large files, which pointer needs to be used depends on which logical block needs to be accessed. For example, to locate logical block 12, the single indirect pointer needs to used.

Slide 36



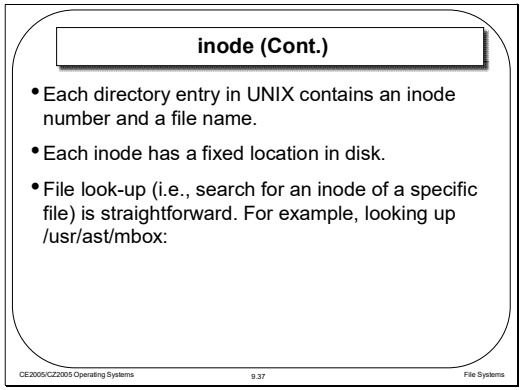




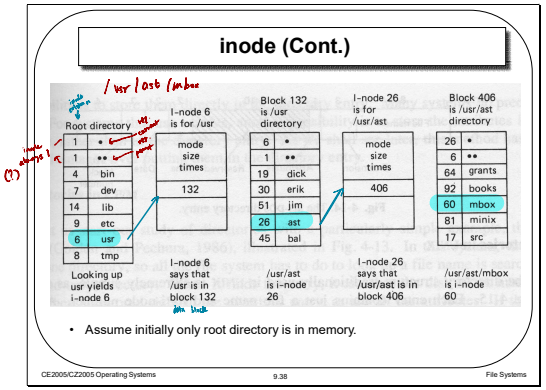
It's common nowadays to have a 64-bit file pointers. Pointers of this size allow files and file systems to be terabytes in size.

Now, let's figure out what is the maximum file size the inode structure can support.

Slide 37



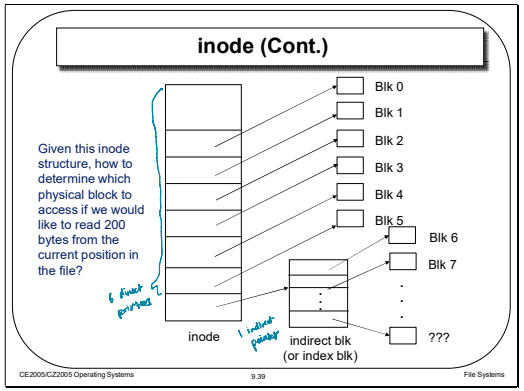

Slide 38



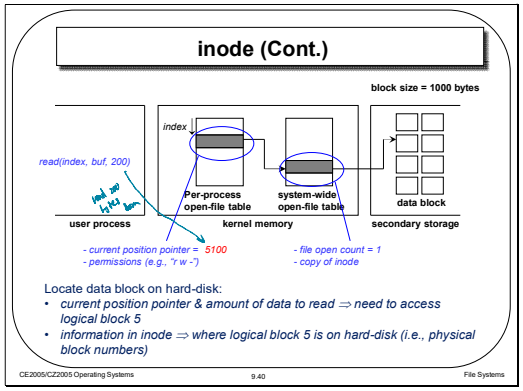

User process refers to a file using a file name, recall the open() system call.

OS maps the file name to an inode number through file-system directories, and the corresponding inode contains the space-allocation information of the file.

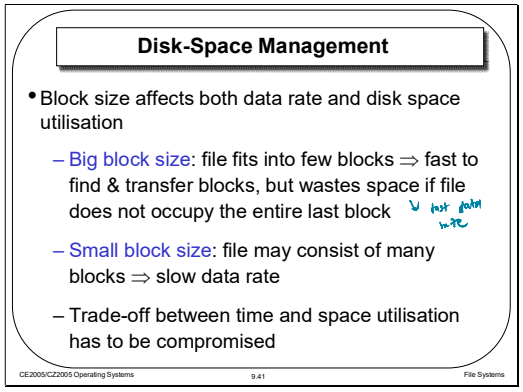
Slide 39



Slide 40

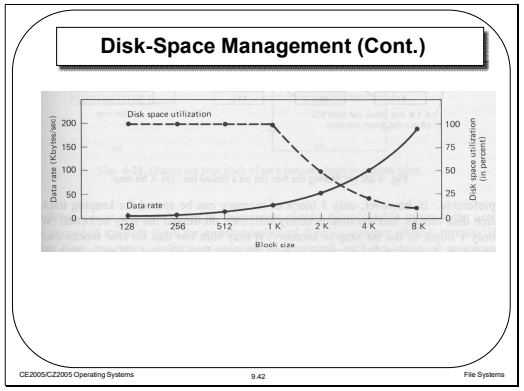



Slide 41




One important issue is to determine the block size. Block size affects both data rate and disk space utilisation. There is a trade-off data rate and space utilisation.

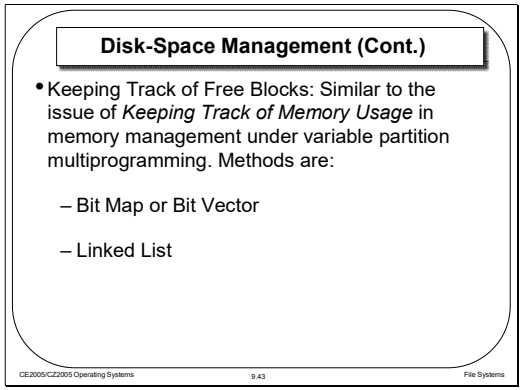
Slide 42




If block size is large, a file fits into few blocks. To read the file, less number of I/O will be required. This translates to fast data rate. However, a file may not occupy the entire last block. So, disk space utilization is low.

On the other hand, if block size is small, a file may need many blocks. To read the file, more number of I/O will be required. Data rate is low. However, the wastage in the last block is small.

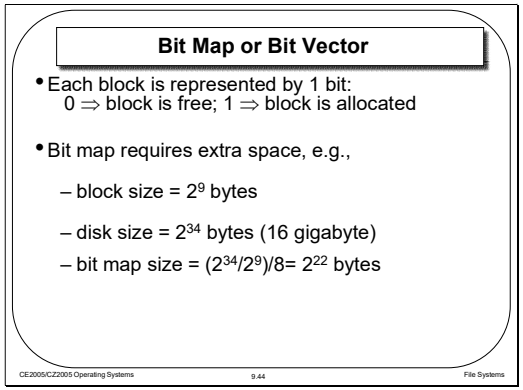
Slide 43




Just as the space that is allocated to files must be managed, so the space that is not currently allocated to any file must also be managed. To perform any of the file allocation techniques described previously, it is necessary to know what blocks on the disk are available.

There are two methods: bit map and linked list.

Slide 44

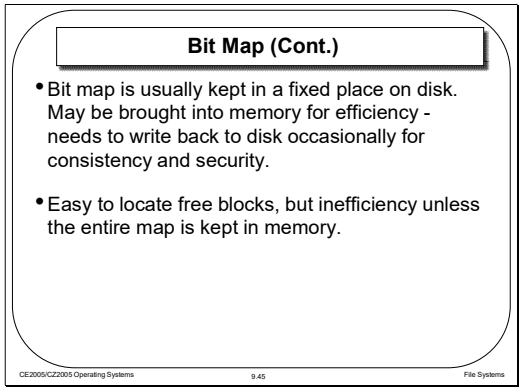



This method is simple, however, the bit map can be sizable. The amount of memory (in bytes) required for a bitmap is

(disk size in bytes/file system block size in bytes)/8

Thus, for a 16-Gbyte disk with 512-byte blocks, the bit table occupies about 4 Mbytes.

Slide 45



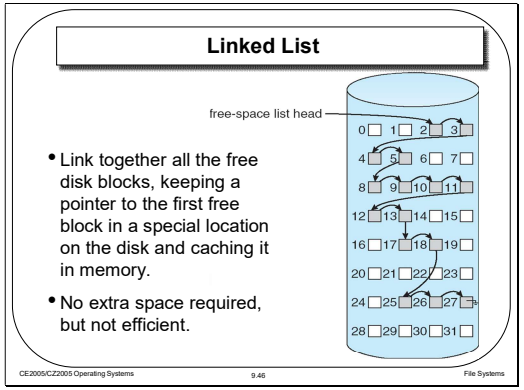

In the example discussed in the last slide, the size of bit map is 4 Mbytes. Can we spare 4 Mbytes of main memory for the bit map? If so, then the bit map can be searched without the need for disk access. But even with today's memory sizes, 4 Mbytes is still a hefty chunk of main memory.

Even when the bit map is in main memory, an exhaustive search of the map can introduce considerable overhead. This is especially true when the disk is nearly full and there are few free blocks remaining. Many operating systems maintain some additional data structures to minimize this searching operation.

---

For example, the map could be divided logically into a number of equal-size subranges. A summary table could include, for each subrange, the number of free blocks and the maximum-sized contiguous number of free blocks. When the file system needs a number of contiguous blocks, it can scan the summary table to find an appropriate subrange and then search that subrange.

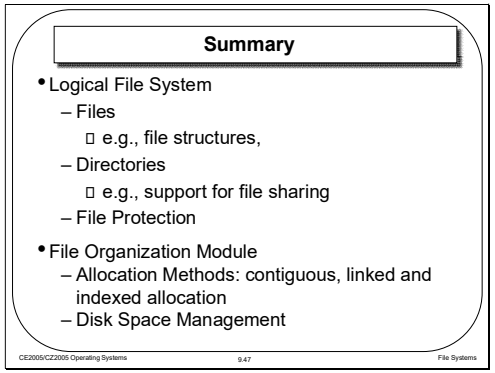
Slide 46




Using linked list approach, the free disk blocks are chained together by using a linked list. This method has negligible space overhead.

If allocation is a block at a time (e.g., linked or indexed allocation), simply choose the free block at the head of the chain and adjust the first pointer or length value. If allocation is contiguous, a first-fit algorithm may be used to search the chain to locate required number of contiguous free blocks. Again, pointer and length values are adjusted and link list needs to be modified.

However, this method has its own problems. Every time you allocate a block, you need to read the block first to recover the pointer to the next free block before writing data to that block. If many individual blocks need to be allocated at one time for a file operation, this greatly slows file opetration. Similarly, deleting a file whose blocks are at various locations on the disk is very time consuming because freed up blocks need to be added into the linked list in order.



# True or False

Accessing a file using symbolic link takes longer time than using hard link. True

Open file table is used to temporarily cache data blocks of a file to improve efficiency. False

Assuming the protection bits of a file are "r - x r - x - - x", only the owner of this file is able to read the file. false

All file allocation methods suffer from internal fragmentation. true