Lecture 10

File-System Interface & File System Implementation I

Lecture Information

Ceng328 *Operating Systems* at May 04, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Dr. Cem Özdoğan

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).

Dr. Cem Özdoğan

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).
  - Some devices transfer a character or a block of characters at a time.

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).
  - Some devices transfer a character or a block of characters at a time.
  - Some can be accessed only sequentially, others randomly.

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).
  - Some devices transfer a character or a block of characters at a time.
  - Some can be accessed only sequentially, others randomly.
  - Some transfer data synchronously, others asynchronously.

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).
  - Some devices transfer a character or a block of characters at a time.
  - Some can be accessed only sequentially, others randomly.
  - Some transfer data synchronously, others asynchronously.
  - Some are dedicated, some shared.

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).
  - Some devices transfer a character or a block of characters at a time.
  - Some can be accessed only sequentially, others randomly.
  - Some transfer data synchronously, others asynchronously.
  - Some are dedicated, some shared.
  - They can be read-only or read-write.

File-System Interface I

- Since main memory is usually too small, the computer system must provide *secondary storage to back up main memory*.
- The file system provides the mechanism for storage of and (multiple) access to both data and programs residing on the disks.
- *File Management System:* Bridges the gap between low-level disk organization (an array of blocks) and the user's views (a stream or collection of records) (mapped).
  - Some devices transfer a character or a block of characters at a time.
  - Some can be accessed only sequentially, others randomly.
  - Some transfer data synchronously, others asynchronously.
  - Some are dedicated, some shared.
  - They can be read-only or read-write.
- Also includes tools outside the kernel; formatting, recovery, defrag, consistency, backup utilities (system administration).

File-System Interface II

- In many ways, they are also the slowest major component of the computer.

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;

File-System Interface II

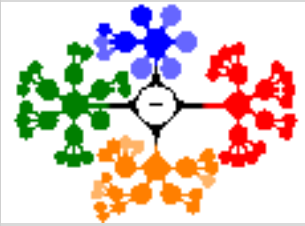
- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;
- Objectives for a file management system;

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;
- Objectives for a file management system;
  - Provide a convenient naming system for files.

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;
- Objectives for a file management system;
  - Provide a convenient naming system for files.
  - Provide a standardized set of I/O interface routines and provide access control for multiple users.



File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;
- Objectives for a file management system;
  - Provide a convenient naming system for files.
  - Provide a standardized set of I/O interface routines and provide access control for multiple users.
  - Guarantee that the data in the file are valid. Minimize or eliminate the potential for lost or destroyed data.

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;
- Objectives for a file management system;
  - Provide a convenient naming system for files.
  - Provide a standardized set of I/O interface routines and provide access control for multiple users.
  - Guarantee that the data in the file are valid. Minimize or eliminate the potential for lost or destroyed data.
  - Optimize performance.

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;
- Objectives for a file management system;
  - Provide a convenient naming system for files.
  - Provide a standardized set of I/O interface routines and provide access control for multiple users.
  - Guarantee that the data in the file are valid. Minimize or eliminate the potential for lost or destroyed data.
  - Optimize performance.
  - How do you keep one user from reading another's data?

File-System Interface II

- In many ways, they are also the slowest major component of the computer.
- A file is a collection of related information defined by its creator.
- The file system consists of two distinct parts:
  1. **a collection of files**;
  2. **a directory structure**;
- Objectives for a file management system;
  - Provide a convenient naming system for files.
  - Provide a standardized set of I/O interface routines and provide access control for multiple users.
  - Guarantee that the data in the file are valid. Minimize or eliminate the potential for lost or destroyed data.
  - Optimize performance.
  - How do you keep one user from reading another's data?
  - How do you know which blocks are free?

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.
  - Data files may be numeric, alphabetic, alphanumeric, or binary.

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.
  - Data files may be numeric, alphabetic, alphanumeric, or binary.
  - Files may be free form, such as text files, or may be formatted rigidly.

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.
  - Data files may be numeric, alphabetic, alphanumeric, or binary.
  - Files may be free form, such as text files, or may be formatted rigidly.
- A file has a certain defined structure, which depends on its **type**.

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.
  - Data files may be numeric, alphabetic, alphanumeric, or binary.
  - Files may be free form, such as text files, or may be formatted rigidly.
- A file has a certain defined structure, which depends on its **type**.
  - A *text file*; (sequence of characters)

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.
  - Data files may be numeric, alphabetic, alphanumeric, or binary.
  - Files may be free form, such as text files, or may be formatted rigidly.
- A file has a certain defined structure, which depends on its **type**.
  - A *text file*; (sequence of characters)
  - A *source file*; (sequence of subroutines and functions)

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.
  - Data files may be numeric, alphabetic, alphanumeric, or binary.
  - Files may be free form, such as text files, or may be formatted rigidly.
- A file has a certain defined structure, which depends on its **type**.
  - A *text file*; (sequence of characters)
  - A *source file*; (sequence of subroutines and functions)
  - An *object file* is a sequence of bytes organized into blocks understandable by the system's linker.

File Concept

- Think of a disk as a linear sequence of fixed-size blocks and supporting reading and writing of blocks.
- The OS abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**.
- A file is a named collection of related information that is recorded on secondary storage, with two views:
  - Logical (programmer) view,
  - Physical (OS) view,
- The information in a file is defined by its creator.
  - Data files may be numeric, alphabetic, alphanumeric, or binary.
  - Files may be free form, such as text files, or may be formatted rigidly.
- A file has a certain defined structure, which depends on its **type**.
  - A *text file*; (sequence of characters)
  - A *source file*; (sequence of subroutines and functions)
  - An *object file* is a sequence of bytes organized into blocks understandable by the system's linker.
  - An *executable file* is a series of code sections that the loader can bring into memory and execute.

File Attributes I

When a file is named, it becomes independent of the process, the user, and even the system that created it.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure: Some possible file attributes.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.
  - **Identifier**. This unique tag, usually a number, identifies the file within the file system (non-human-readable name)

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.
  - **Identifier**. This unique tag, usually a number, identifies the file within the file system (non-human-readable name)
  - **Type**.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.
  - **Identifier**. This unique tag, usually a number, identifies the file within the file system (non-human-readable name)
  - **Type**.
  - **Location**. This information is a pointer to a device and to the location of the file on that device.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.
  - **Identifier**. This unique tag, usually a number, identifies the file within the file system (non-human-readable name)
  - **Type**.
  - **Location**. This information is a pointer to a device and to the location of the file on that device.
  - **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.
  - **Identifier**. This unique tag, usually a number, identifies the file within the file system (non-human-readable name)
  - **Type**.
  - **Location**. This information is a pointer to a device and to the location of the file on that device.
  - **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size.
  - **Protection**. Access-control information determines who can do reading, writing, executing, and so on.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.
  - **Identifier**. This unique tag, usually a number, identifies the file within the file system (non-human-readable name)
  - **Type**.
  - **Location**. This information is a pointer to a device and to the location of the file on that device.
  - **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size.
  - **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
  - **Time, date, and user identification**. This information may be kept for creation, last modification, and last use.

File Attributes II

- The table of Fig. 1 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.
- A file's attributes vary from one OS to another but typically consist of these:
  - **Name**.
  - **Identifier**. This unique tag, usually a number, identifies the file within the file system (non-human-readable name)
  - **Type**.
  - **Location**. This information is a pointer to a device and to the location of the file on that device.
  - **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size.
  - **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
  - **Time, date, and user identification**. This information may be kept for creation, last modification, and last use.
- The information about all files is kept in the directory structure, which also resides on secondary storage.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.
    1. Space in the file system must be found for the file.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.
    1. Space in the file system must be found for the file.
    2. An entry for the new file must be made in the directory.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.
    1. Space in the file system must be found for the file.
    2. An entry for the new file must be made in the directory.
  - **Writing a file**. The write pointer must be updated whenever a write occurs.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.
    1. Space in the file system must be found for the file.
    2. An entry for the new file must be made in the directory.
  - **Writing a file**. The write pointer must be updated whenever a write occurs.
  - **Reading a file**. The system needs to keep a read pointer to the location in the file where the next read is to take place.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.
    1. Space in the file system must be found for the file.
    2. An entry for the new file must be made in the directory.
  - **Writing a file**. The write pointer must be updated whenever a write occurs.
  - **Reading a file**. The system needs to keep a read pointer to the location in the file where the next read is to take place.
    - Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-position pointer.

File Operations I

- A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files.
- Six basic file operations. The OS can provide system calls to create, write, read, reposition, delete, and truncate files.
  - **Creating a file**. Two steps are necessary to create a file.
    1. Space in the file system must be found for the file.
    2. An entry for the new file must be made in the directory.
  - **Writing a file**. The write pointer must be updated whenever a write occurs.
  - **Reading a file**. The system needs to keep a read pointer to the location in the file where the next read is to take place.
    - Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current-file-position pointer.
    - Both the read and write operations use this same pointer, saving space and reducing system complexity.

File Operations II

- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value (file seek).

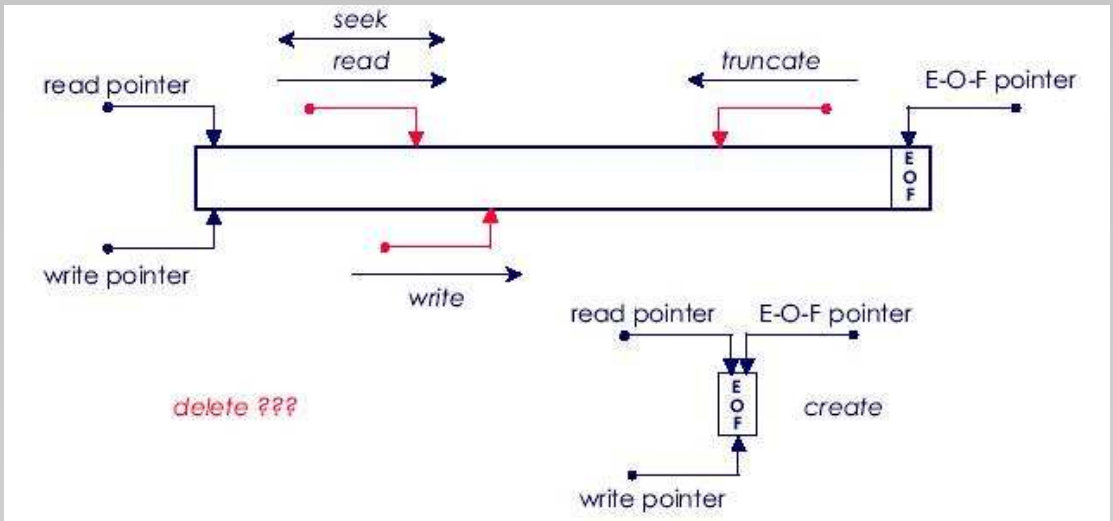


Figure: File operations.

File Operations II

- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value (file seek).
- **Deleting a file**. 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.

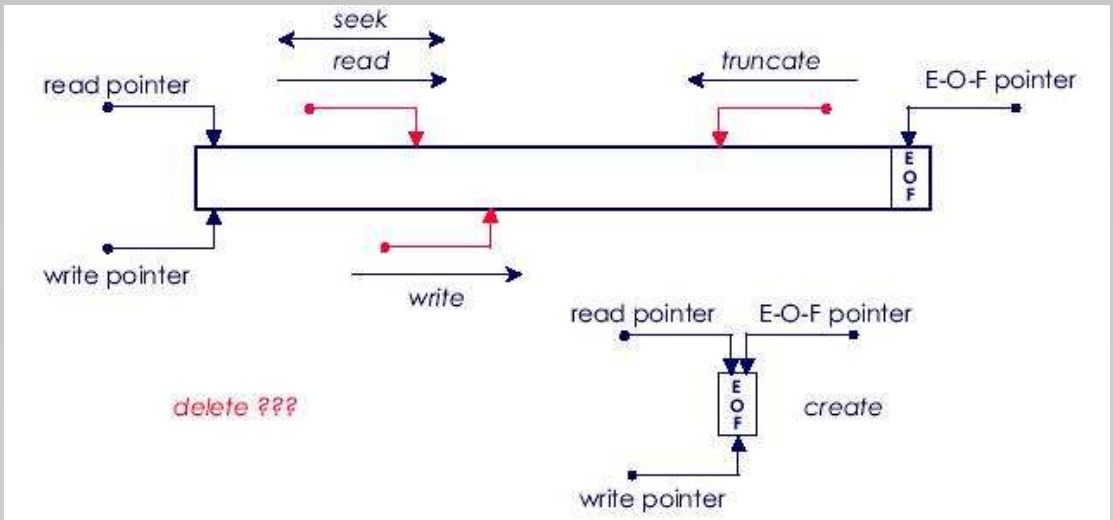
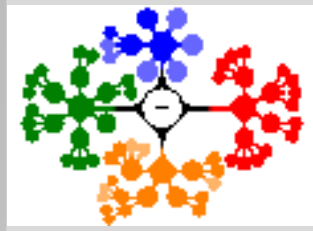


Figure: File operations.



File Operations II

- **Repositioning within a file**. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value (file seek).
- **Deleting a file**. 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.
- **Truncating a file**. The user may want to erase the contents of a file but keep its attributes. The file be reset to length zero and its file space released.

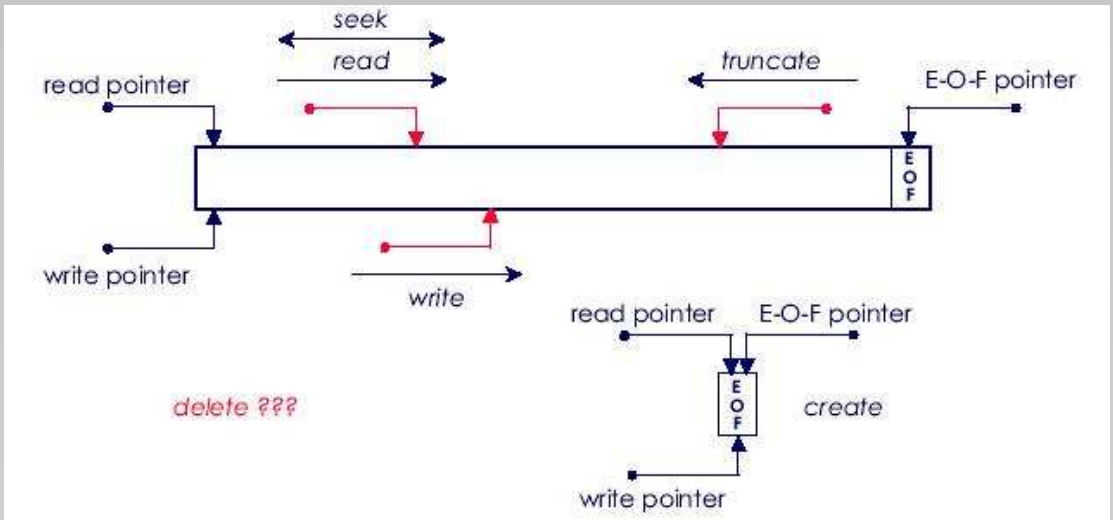
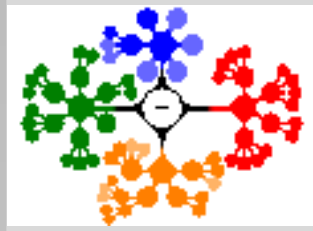


Figure: File operations.



File Operations III

- These primitive operations can then be combined to perform other file operations (i.e., copying).

File Operations III

- These primitive operations can then be combined to perform other file operations (i.e., copying).
- The OS keeps a small table, called the open-file table, containing information about all open files.

File Operations III

- These primitive operations can then be combined to perform other file operations (i.e., copying).
- The OS keeps a small table, called the open-file table, containing information about all open files.
  - When a file operation is requested, the file is specified via an index into this table, so no searching is required.

File Operations III

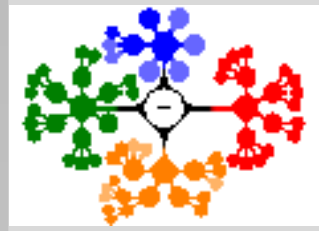
- These primitive operations can then be combined to perform other file operations (i.e., copying).
- The OS keeps a small table, called the open-file table, containing information about all open files.
  - When a file operation is requested, the file is specified via an index into this table, so no searching is required.
  - When the file is no longer being actively used, it is closed by the process, and the OS removes its entry from the open-file table.

File Operations III

- These primitive operations can then be combined to perform other file operations (i.e., copying).
- The OS keeps a small table, called the open-file table, containing information about all open files.
  - When a file operation is requested, the file is specified via an index into this table, so no searching is required.
  - When the file is no longer being actively used, it is closed by the process, and the OS removes its entry from the open-file table.
- Most systems require that the programmer open a file explicitly with the *open*() system call before that file can be used.

File Operations III

- These primitive operations can then be combined to perform other file operations (i.e., copying).
- The OS keeps a small table, called the open-file table, containing information about all open files.
  - When a file operation is requested, the file is specified via an index into this table, so no searching is required.
  - When the file is no longer being actively used, it is closed by the process, and the OS removes its entry from the open-file table.
- Most systems require that the programmer open a file explicitly with the *open*() system call before that file can be used.
- The implementation of the *open*() and *close*() operations is more complicated in an environment where several processes may open the file at the same time.



File Operations III

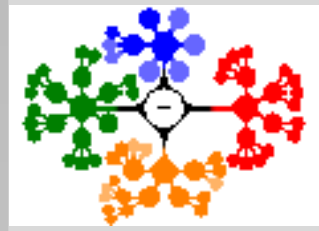
- These primitive operations can then be combined to perform other file operations (i.e., copying).
- The OS keeps a small table, called the open-file table, containing information about all open files.
  - When a file operation is requested, the file is specified via an index into this table, so no searching is required.
  - When the file is no longer being actively used, it is closed by the process, and the OS removes its entry from the open-file table.
- Most systems require that the programmer open a file explicitly with the *open*() system call before that file can be used.
- The implementation of the *open*() and *close*() operations is more complicated in an environment where several processes may open the file at the same time.
- This may occur in a system where several different applications open the same file at the same time.

File Operations IV

- Typically, the OS uses two levels of internal tables:

File Operations IV

- Typically, the OS uses two levels of internal tables:
  1. **A per-process table**. The per-process table tracks all files that a process has open. Access rights to the file and accounting information can also be included.



File Operations IV

- Typically, the OS uses two levels of internal tables:
  1. **A per-process table**. The per-process table tracks all files that a process has open. Access rights to the file and accounting information can also be included.
  2. **A system-wide table**. Each entry in the per-process table in turn points to a system-wide open-file table (also contains process-independent information, such as the location of the file on disk, access dates, and file size).

File Operations IV

- Typically, the OS uses two levels of internal tables:
  1. **A per-process table**. The per-process table tracks all files that a process has open. Access rights to the file and accounting information can also be included.
  2. **A system-wide table**. Each entry in the per-process table in turn points to a system-wide open-file table (also contains process-independent information, such as the location of the file on disk, access dates, and file size).
- Typically, the open-file table also has an open count associated with each file to indicate how many processes have the file open.

File Operations IV

- Typically, the OS uses two levels of internal tables:
  1. **A per-process table**. The per-process table tracks all files that a process has open. Access rights to the file and accounting information can also be included.
  2. **A system-wide table**. Each entry in the per-process table in turn points to a system-wide open-file table (also contains process-independent information, such as the location of the file on disk, access dates, and file size).
- Typically, the open-file table also has an open count associated with each file to indicate how many processes have the file open.
  - Each *close*() decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

File Operations V

- In summary, several pieces of information are associated with an open file.

File Operations V

- In summary, several pieces of information are associated with an open file.
  - **File pointer**.

File Operations V

- In summary, several pieces of information are associated with an open file.
  - **File pointer**.
  - **File-open count**.

File Operations V

- In summary, several pieces of information are associated with an open file.
  - **File pointer**.
  - **File-open count**.
  - **Disk location of the file**. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.

File Operations V

- In summary, several pieces of information are associated with an open file.
  - **File pointer**.
  - **File-open count**.
  - **Disk location of the file**. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
  - **Access rights**. Each process opens a file in an access mode. This information is stored on the per-process table so the OS can allow or deny subsequent I/O requests.

File Operations V

- In summary, several pieces of information are associated with an open file.
  - **File pointer**.
  - **File-open count**.
  - **Disk location of the file**. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
  - **Access rights**. Each process opens a file in an access mode. This information is stored on the per-process table so the OS can allow or deny subsequent I/O requests.
- Some OSs provide facilities for locking an open file (or sections of a file).

File Operations V

- In summary, several pieces of information are associated with an open file.
  - **File pointer**.
  - **File-open count**.
  - **Disk location of the file**. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
  - **Access rights**. Each process opens a file in an access mode. This information is stored on the per-process table so the OS can allow or deny subsequent I/O requests.
- Some OSs provide facilities for locking an open file (or sections of a file).
- File locks are useful for files that are shared by several processes.

An Example Program Using File System Calls I

- A simple UNIX program that copies one file from its source file to a destination file (see Fig. 3). The program has minimal functionality and even worse error reporting.

```
copyfile abc xyz
```


An Example Program Using File System Calls II

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

Figure: A simple program to copy a file.

File Types I

- A common technique for implementing file types is to include the type as part of the file name (see Fig. 4).

File Types I

- A common technique for implementing file types is to include the type as part of the file name (see Fig. 4).
  - The name is split into two parts -a name and an extension, usually separated by a period character.

File Types I

- A common technique for implementing file types is to include the type as part of the file name (see Fig. 4).
  - The name is split into two parts -a name and an extension, usually separated by a period character.
  - In this way, the user and the OS can tell from the name alone what the type of a file is.

File Types I

- A common technique for implementing file types is to include the type as part of the file name (see Fig. 4).
  - The name is split into two parts -a name and an extension, usually separated by a period character.
  - In this way, the user and the OS can tell from the name alone what the type of a file is.
- Many file systems support names as long as 255 characters. Some file systems distinguish between upper and lower case letters (Case (in)sensitivity).

File Types I

- A common technique for implementing file types is to include the type as part of the file name (see Fig. 4).
  - The name is split into two parts -a name and an extension, usually separated by a period character.
  - In this way, the user and the OS can tell from the name alone what the type of a file is.
- Many file systems support names as long as 255 characters. Some file systems distinguish between upper and lower case letters (Case (in)sensitivity).
- Windows 95 and Windows 98 both use the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed.

File Types I

- A common technique for implementing file types is to include the type as part of the file name (see Fig. 4).
  - The name is split into two parts -a name and an extension, usually separated by a period character.
  - In this way, the user and the OS can tell from the name alone what the type of a file is.
- Many file systems support names as long as 255 characters. Some file systems distinguish between upper and lower case letters (Case (in)sensitivity).
- Windows 95 and Windows 98 both use the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed.
- Windows NT and Windows 2000 support the MS-DOS file system and thus also inherit its properties. However, these OSs also have a native file system (NTFS) that has different properties.

File Types II

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, 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, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Figure: Common file types.

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools).

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools).
  - In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system (Interpreted by OS).

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools).
  - In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system (Interpreted by OS).
- UNIX also has *character* and *block special* files (Device Files).

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools).
  - In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system (Interpreted by OS).
- UNIX also has *character* and *block special* files (Device Files).
  - Character special files are related to input/output and used to model serial I/O devices such as terminals, printers, and networks.

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools).
  - In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system (Interpreted by OS).
- UNIX also has *character* and *block special* files (Device Files).
  - Character special files are related to input/output and used to model serial I/O devices such as terminals, printers, and networks.
  - Block special files are used to model disks.

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools).
  - In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system (Interpreted by OS).
- UNIX also has *character* and *block special* files (Device Files).
  - Character special files are related to input/output and used to model serial I/O devices such as terminals, printers, and networks.
  - Block special files are used to model disks.
- Other files are binary files, which just means that they are not ASCII files.

File Types III

- The UNIX system uses a crude **magic number** stored at the beginning of some files to indicate roughly the type of the file -executable program, batchfile (or shell script), PostScript file, and so on.
  - Not all files have magic numbers, so system features cannot be based solely on this information.
  - UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the OS (interpreted by tools).
  - In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system (Interpreted by OS).
- UNIX also has *character* and *block special* files (Device Files).
  - Character special files are related to input/output and used to model serial I/O devices such as terminals, printers, and networks.
  - Block special files are used to model disks.
- Other files are binary files, which just means that they are not ASCII files.
- Usually, they have some internal structure known to programs that use them (see Fig. 5).

File Types IV

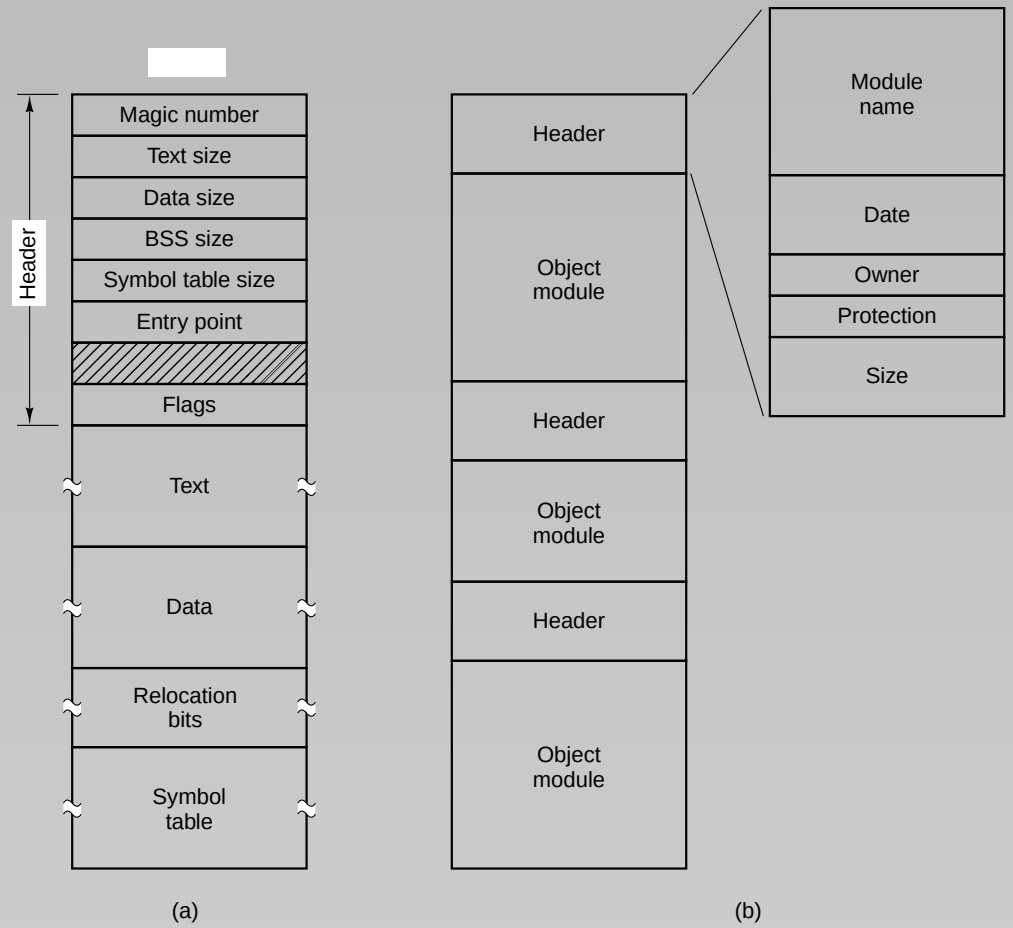
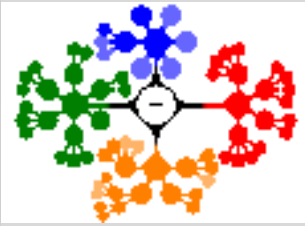


Figure: (a) An executable file. (b) An archive.

Every OS must recognize at least one file type; its own executable file.



File Types V

- A simple executable binary file taken from a version of UNIX is seen in Fig. 5a .

Internal File Structure I

- Three common possibilities for structuring are depicted in Fig. 6.

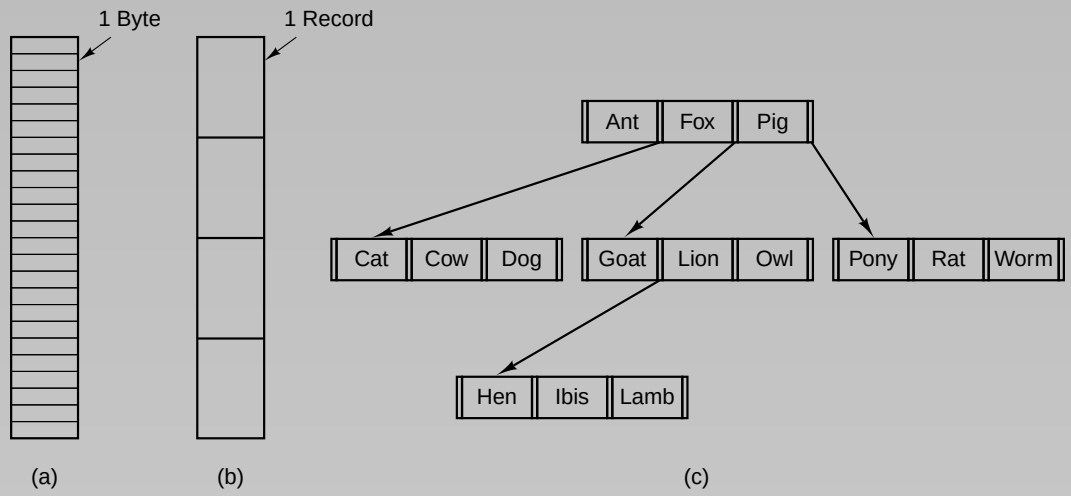


Figure: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Internal File Structure I

- Three common possibilities for structuring are depicted in Fig. 6.

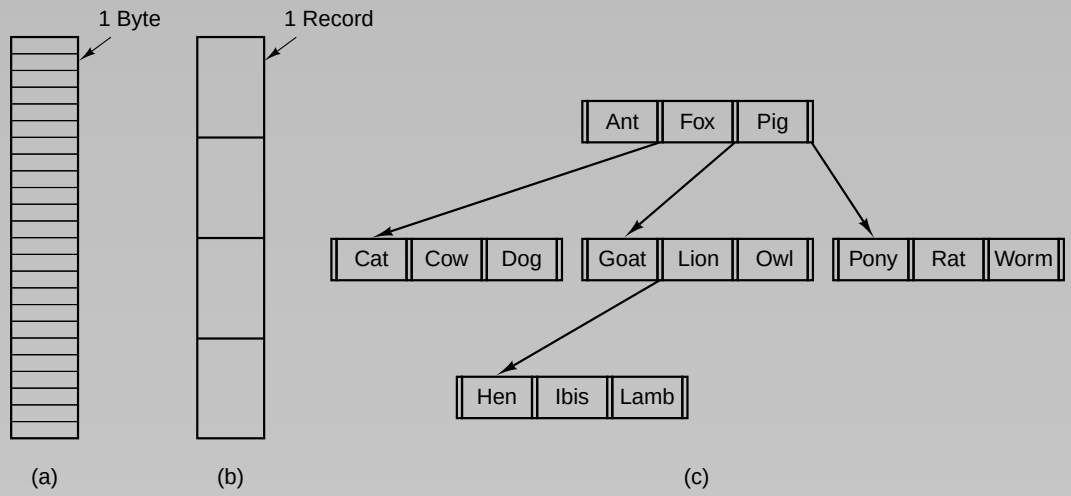


Figure: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

- **Stream of Bytes**. The file in Fig. 6a is an *unstructured* sequence of bytes. All it sees are bytes.

Internal File Structure I

- Three common possibilities for structuring are depicted in Fig. 6.

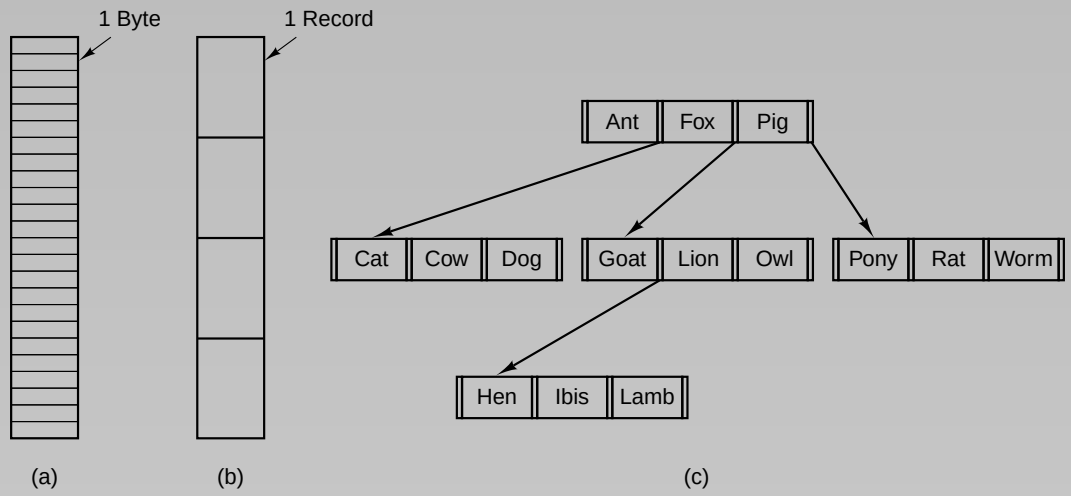


Figure: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

- **Stream of Bytes**. The file in Fig. 6a is an *unstructured* sequence of bytes. All it sees are bytes.
- **Records**. The first step up in structure is shown in Fig. 6b. A file is a sequence of *fixed-length records*, each with some internal structure.

Internal File Structure I

- Three common possibilities for structuring are depicted in Fig. 6.

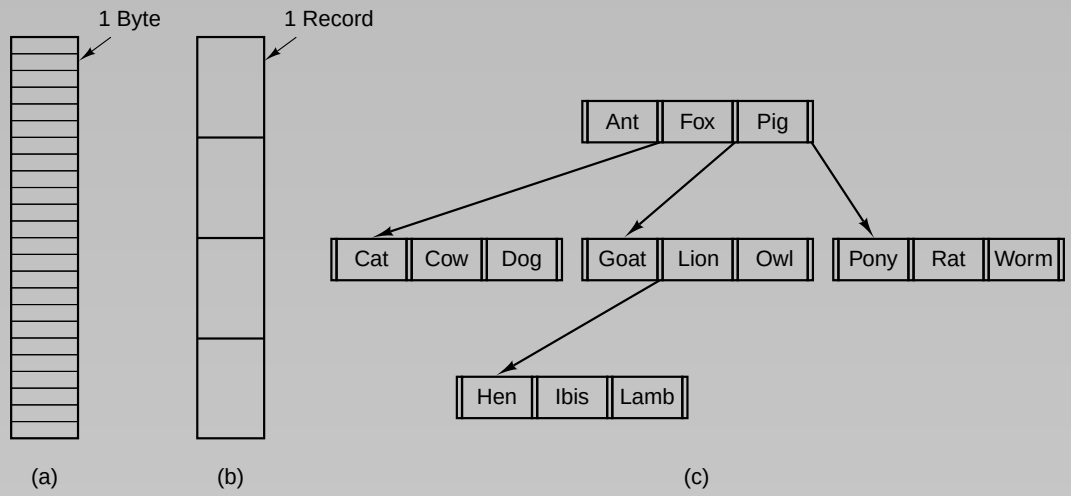


Figure: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

- **Stream of Bytes**. The file in Fig. 6a is an *unstructured* sequence of bytes. All it sees are bytes.
- **Records**. The first step up in structure is shown in Fig. 6b. A file is a sequence of *fixed-length records*, each with some internal structure.
- **Tree of Records**. The third kind of file structure is shown in Fig. 6c. In this organization, a file consists of a tree of records, not necessarily all the same length (a key field).

Internal File Structure II

- Internally, locating an offset within a file can be complicated for the OS.

Internal File Structure II

- Internally, locating an offset within a file can be complicated for the OS.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.

Internal File Structure II

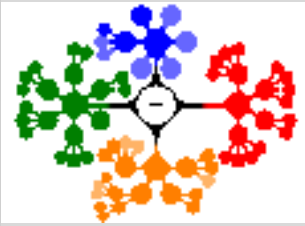
- Internally, locating an offset within a file can be complicated for the OS.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.

Internal File Structure II

- Internally, locating an offset within a file can be complicated for the OS.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Packing a number of logical records into physical blocks is a common solution to this problem.

Internal File Structure II

- Internally, locating an offset within a file can be complicated for the OS.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Packing a number of logical records into physical blocks is a common solution to this problem.
- The file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks.



Internal File Structure II

- Internally, locating an offset within a file can be complicated for the OS.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Packing a number of logical records into physical blocks is a common solution to this problem.
- The file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks.
- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted.

Internal File Structure II

- Internally, locating an offset within a file can be complicated for the OS.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Packing a number of logical records into physical blocks is a common solution to this problem.
- The file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks.
- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted.
- The waste incurred to keep everything in units of blocks (instead of bytes) is **internal fragmentation**.

Internal File Structure II

- Internally, locating an offset within a file can be complicated for the OS.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Packing a number of logical records into physical blocks is a common solution to this problem.
- The file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks.
- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted.
- The waste incurred to keep everything in units of blocks (instead of bytes) is **internal fragmentation**.
- All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

Sequential Access I

- The simplest access method is **sequential access**.

Sequential Access I

- The simplest access method is **sequential access**.
- Information in the file is processed in order, one record after the other.

Sequential Access I

- The simplest access method is **sequential access**.
- Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.

Sequential Access I

- The simplest access method is **sequential access**.
- Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
  - A read operation *read next* reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.

Sequential Access I

- The simplest access method is **sequential access**.
- Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
  - A read operation *read next* reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
  - Similarly, the write operation *write next* appends to the end of the file and advances to the end of the newly written material (the new end of file).

Sequential Access II

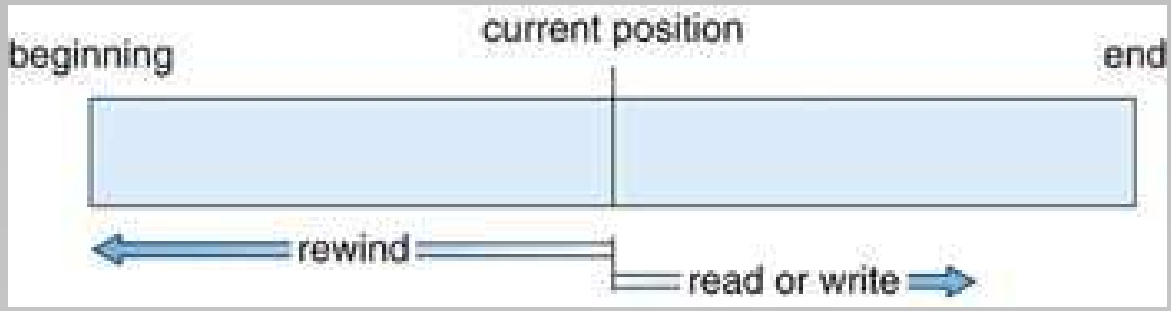
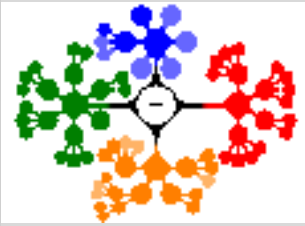


Figure: Sequential-access file.

Sequential access, which is depicted in Fig. 7, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.



Direct (Random) Access

- Another method is **direct access** (or **relative access**).

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- There are no restrictions on the order of reading or writing for a direct-access file.

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- There are no restrictions on the order of reading or writing for a direct-access file.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- There are no restrictions on the order of reading or writing for a direct-access file.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- Direct-access files are of great use for immediate access to large amounts of information (Databases).

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- There are no restrictions on the order of reading or writing for a direct-access file.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- Direct-access files are of great use for immediate access to large amounts of information (Databases).
- The block number provided by the user to the OS is normally a relative block number.

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- There are no restrictions on the order of reading or writing for a direct-access file.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- Direct-access files are of great use for immediate access to large amounts of information (Databases).
- The block number provided by the user to the OS is normally a relative block number.
  - A relative block number is an index relative to the beginning of the file.

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- There are no restrictions on the order of reading or writing for a direct-access file.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- Direct-access files are of great use for immediate access to large amounts of information (Databases).
- The block number provided by the user to the OS is normally a relative block number.
  - A relative block number is an index relative to the beginning of the file.
  - Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.

Direct (Random) Access

- Another method is **direct access** (or **relative access**).
- A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- There are no restrictions on the order of reading or writing for a direct-access file.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- Direct-access files are of great use for immediate access to large amounts of information (Databases).
- The block number provided by the user to the OS is normally a relative block number.
  - A relative block number is an index relative to the beginning of the file.
  - Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.
- Modern OSs have all their files are automatically random access.

Storage Structure I

- Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.

Storage Structure I

- Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions**, **slices**, or (in the IBM world) **minidisks**.

Storage Structure I

- Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions**, **slices**, or (in the IBM world) **minidisks**.
- A file system can be created on each of these parts of the disk. We simply refer to a chunk of storage that holds a file system as a **volume**.

Storage Structure I

- Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions**, **slices**, or (in the IBM world) **minidisks**.
- A file system can be created on each of these parts of the disk. We simply refer to a chunk of storage that holds a file system as a **volume**.
- Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**.

Storage Structure I

- Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions**, **slices**, or (in the IBM world) **minidisks**.
- A file system can be created on each of these parts of the disk. We simply refer to a chunk of storage that holds a file system as a **volume**.
- Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**.
- The device directory (more commonly known simply as a directory) records information-such as name, location, size, and type-for all files on that volume.

Storage Structure II

Figure 8 shows a typical file-system organization.

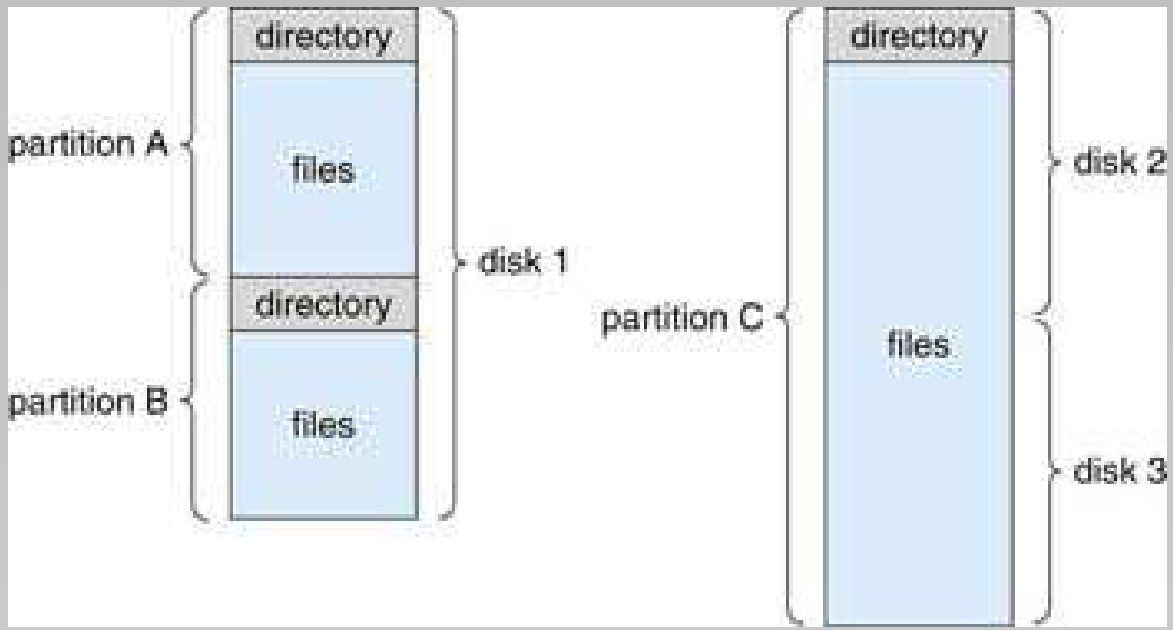
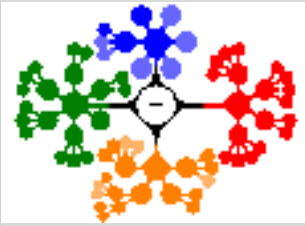


Figure: A typical file-system organization.



Directory Overview

- To keep track of files, file systems normally have directories or folders.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  - **Search for a file**.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  - **Search for a file**.
  - **Create a file**.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  - **Search for a file**.
  - **Create a file**.
  - **Delete a file**.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  - **Search for a file**.
  - **Create a file**.
  - **Delete a file**.
  - **List a directory**.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  - **Search for a file**.
  - **Create a file**.
  - **Delete a file**.
  - **List a directory**.
  - **Rename a file**.

Directory Overview

- To keep track of files, file systems normally have directories or folders.
- Usually, *a directory is itself a file*.
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- A typical directory entry contains information (attributes, location, ownership) about a file.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  - **Search for a file**.
  - **Create a file**.
  - **Delete a file**.
  - **List a directory**.
  - **Rename a file**.
  - **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals (backup copy).

Single-Level Directory

- The simplest directory structure is the single-level directory.

Single-Level Directory

- The simplest directory structure is the single-level directory.
- All files are contained in the same directory, which is easy to support and understand (see Fig. 9).

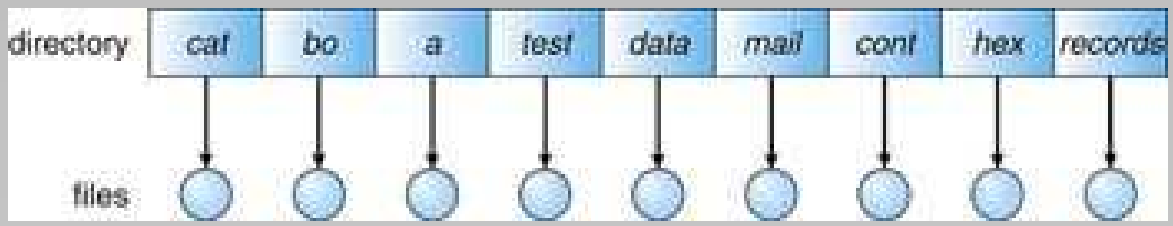


Figure: Single-level directory.

Single-Level Directory

- The simplest directory structure is the single-level directory.
- All files are contained in the same directory, which is easy to support and understand (see Fig. 9).

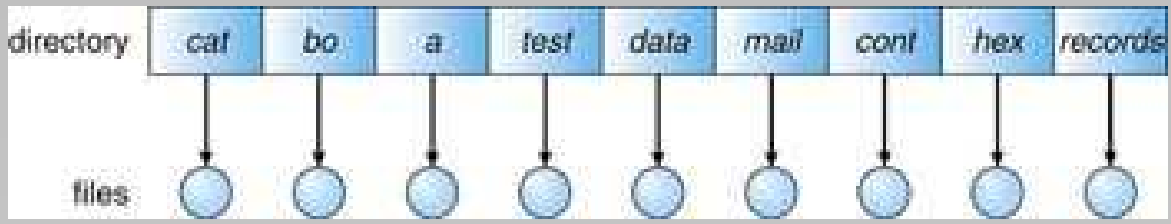


Figure: Single-level directory.

- A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.

Single-Level Directory

- The simplest directory structure is the single-level directory.
- All files are contained in the same directory, which is easy to support and understand (see Fig. 9).

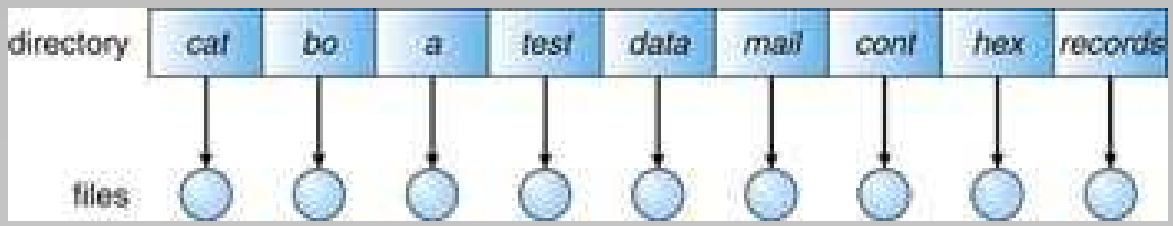


Figure: Single-level directory.

- A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names. If two users call their data file *test*, then the unique-name rule is violated.

Single-Level Directory

- The simplest directory structure is the single-level directory.
- All files are contained in the same directory, which is easy to support and understand (see Fig. 9).

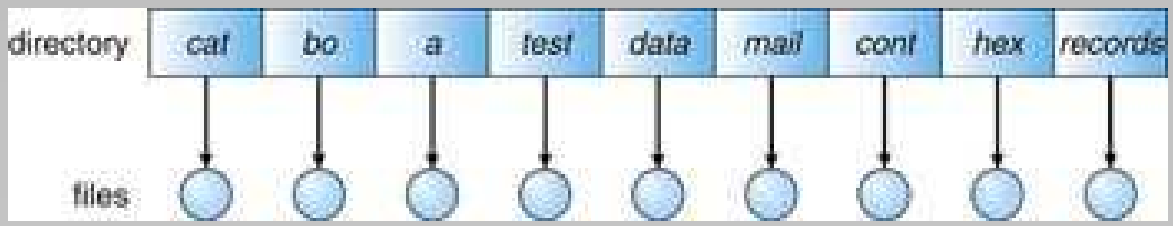


Figure: Single-level directory.

- A single-level directory has significant limitations, when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names. If two users call their data file *test*, then the unique-name rule is violated.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

Two-Level Directory I

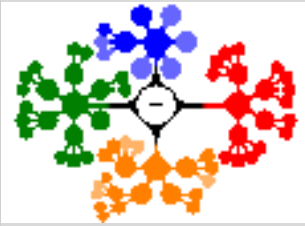
- The standard solution to limitations of single-level directory is to create a separate directory for each user.

Two-Level Directory I

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (UFD).

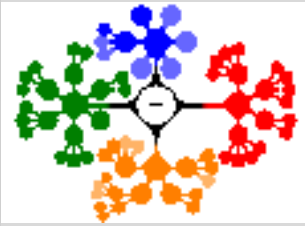
Two-Level Directory I

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (UFD).
- The UFDs have similar structures, but each lists only the files of a single user.



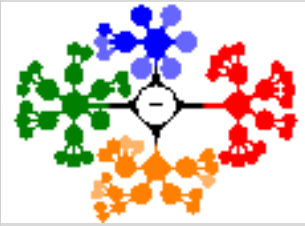
Two-Level Directory I

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (UFD).
- The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory** (MFD) is searched.



Two-Level Directory I

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (UFD).
- The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory** (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user (see Fig. 10).

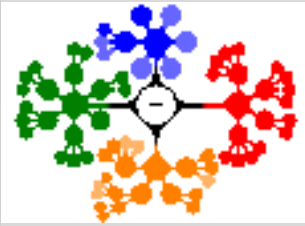


Two-Level Directory I

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (UFD).
- The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory** (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user (see Fig. 10).
- When a user refers to a particular file, only his own UFD is searched (create a file, delete a file?).

Two-Level Directory I

- The standard solution to limitations of single-level directory is to create a separate directory for each user.
- In the two-level directory structure, each user has his own **user file directory** (UFD).
- The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's **master file directory** (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user (see Fig. 10).
- When a user refers to a particular file, only his own UFD is searched (create a file, delete a file?).
- Although the two-level directory structure solves the name-collision problem, it still has disadvantages.



Two-Level Directory II

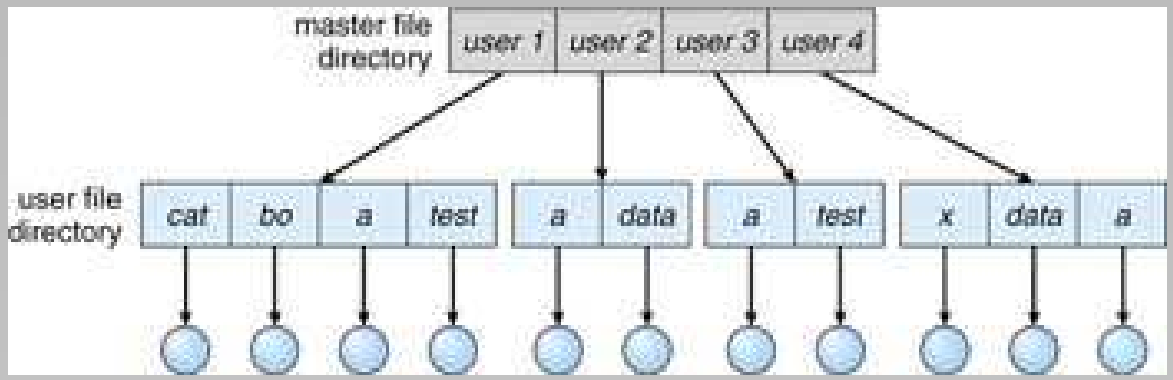


Figure: Two-level directory structure.

- This structure effectively isolates one user from another.

Dr. Cem Özdoğan

Two-Level Directory II

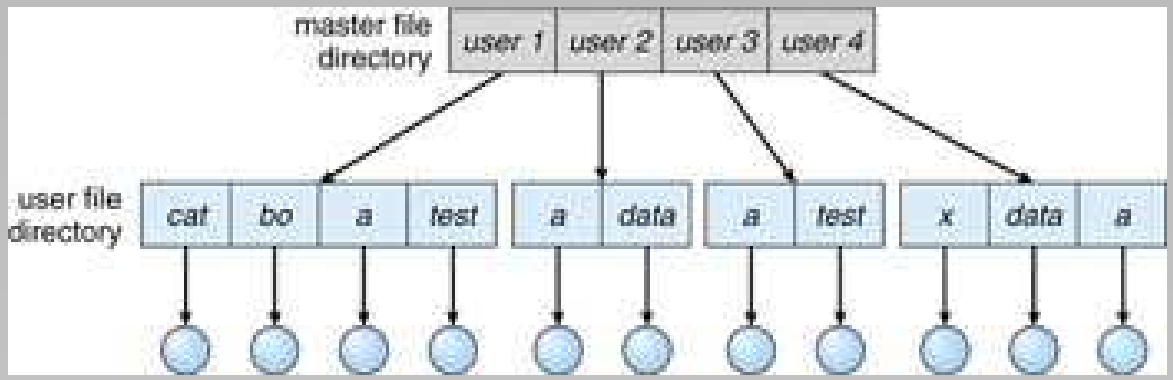


Figure: Two-level directory structure.

- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.

Two-Level Directory II

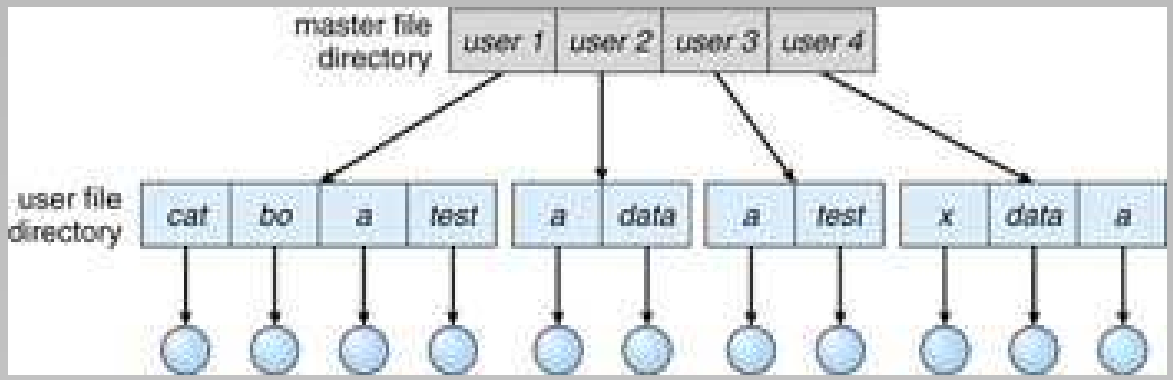


Figure: Two-level directory structure.

- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.
- A two-level directory can be thought of as a tree, or an inverted tree, of height 2.

Two-Level Directory II

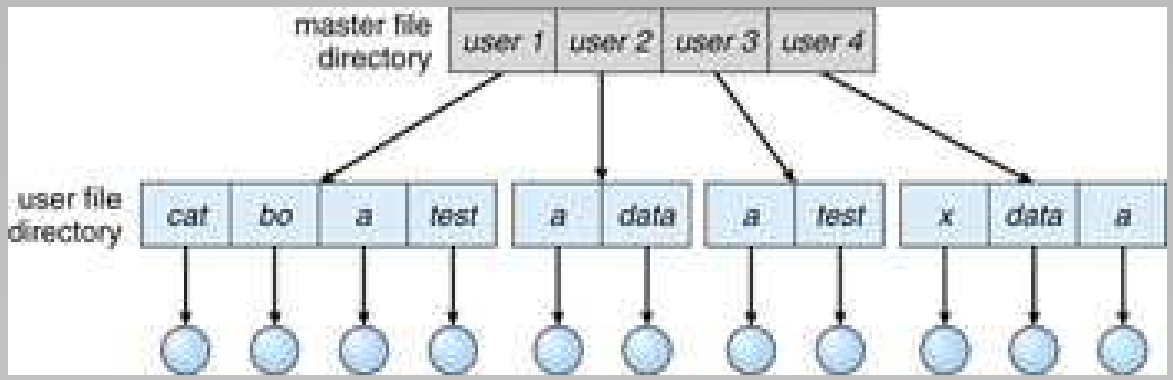


Figure: Two-level directory structure.

- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.
- A two-level directory can be thought of as a tree, or an inverted tree, of height 2.
  - The root of the tree is the MFD.

Two-Level Directory II

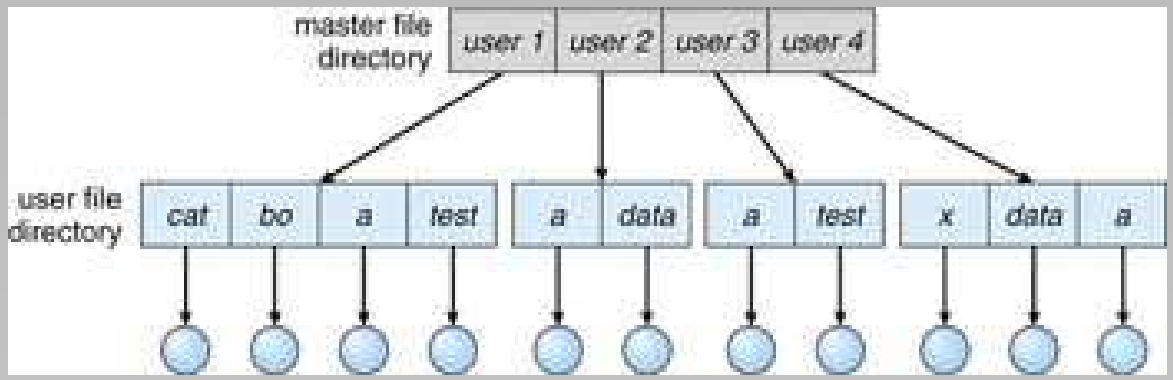


Figure: Two-level directory structure.

- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.
- A two-level directory can be thought of as a tree, or an inverted tree, of height 2.
  - The root of the tree is the MFD.
  - Its direct descendants are the UFDs.

Two-Level Directory II

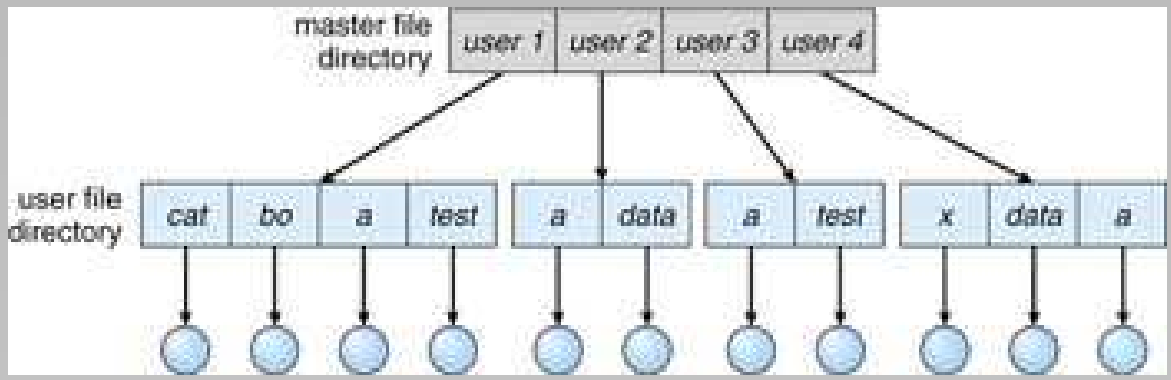


Figure: Two-level directory structure.

- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files.
- A two-level directory can be thought of as a tree, or an inverted tree, of height 2.
  - The root of the tree is the MFD.
  - Its direct descendants are the UFDs.
  - The descendants of the UFDs are the files themselves. The files are the leaves of the tree.

Two-Level Directory III

- Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file).

Tree-Structured Directories I

- Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (see Fig. 11).

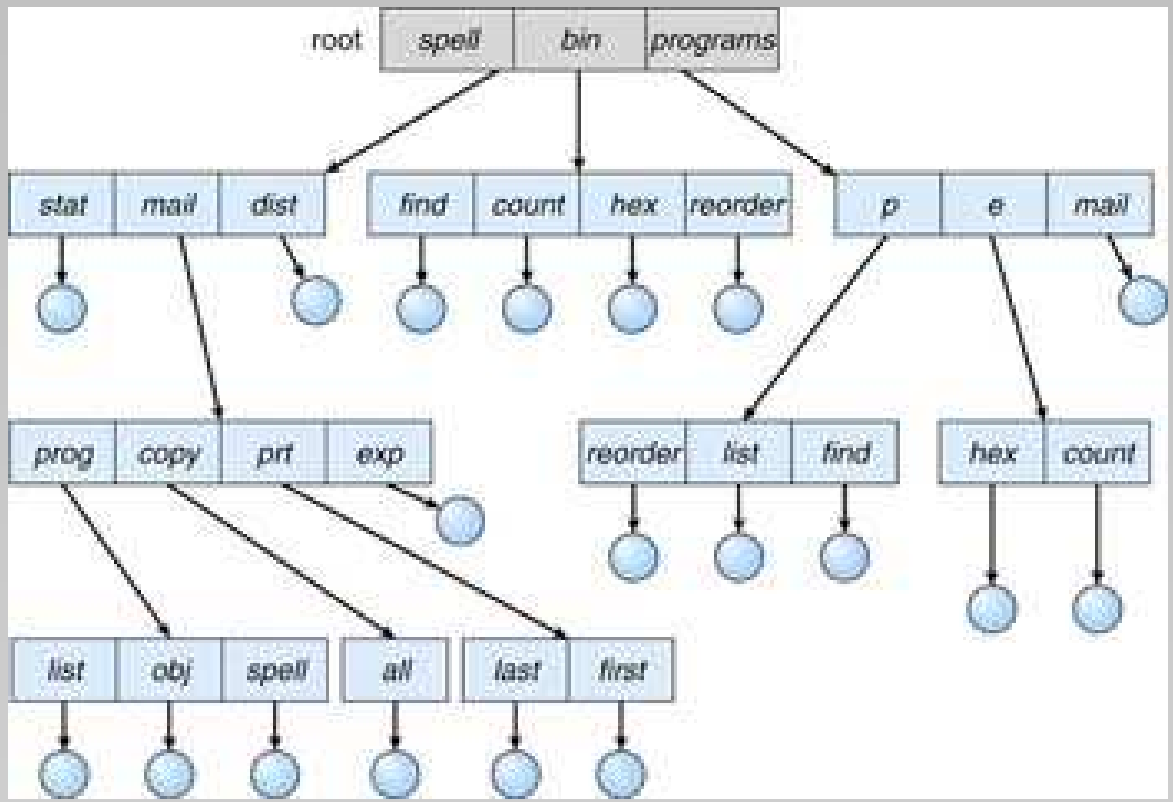


Figure: Tree-structured directory structure.

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.

Tree-Structured Directories II

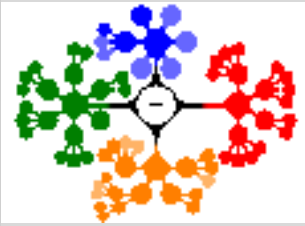
- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.



Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry
  - as a file (0),

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry
  - as a file (0),
  - as a subdirectory (1).

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry
  - as a file (0),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry
  - as a file (0),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**
- With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry
  - as a file (0),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**
- With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.
  - For example, user *B* can access a file of user *A* by specifying its path names.

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry
  - as a file (0),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**
- With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.
  - For example, user *B* can access a file of user *A* by specifying its path names.
  - User *B* can specify either an absolute or a relative path name.

Tree-Structured Directories II

- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory is simply another file, but it is treated in a special way.
- One bit in each directory entry defines the entry
  - as a file (0),
  - as a subdirectory (1).
- Path names can be of two types: **absolute** and **relative**
- With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.
  - For example, user *B* can access a file of user *A* by specifying its path names.
  - User *B* can specify either an absolute or a relative path name.
  - Alternatively, user *B* can change her current directory to be user *A*'s directory and access the file by its file names.

Acyclic-Graph Directories I

- The acyclic graph is a natural generalization of the tree-structured directory scheme.

Acyclic-Graph Directories I

- The acyclic graph is a natural generalization of the tree-structured directory scheme.
- The common subdirectory should be shared.

Acyclic-Graph Directories I

- The acyclic graph is a natural generalization of the tree-structured directory scheme.
- The common subdirectory should be shared.
- A tree structure prohibits the sharing of files or directories.

Acyclic-Graph Directories I

- The acyclic graph is a natural generalization of the tree-structured directory scheme.
- The common subdirectory should be shared.
- A tree structure prohibits the sharing of files or directories.
- An **acyclic graph** (a graph with no cycles) allows directories to share subdirectories and files (see Fig. 12).

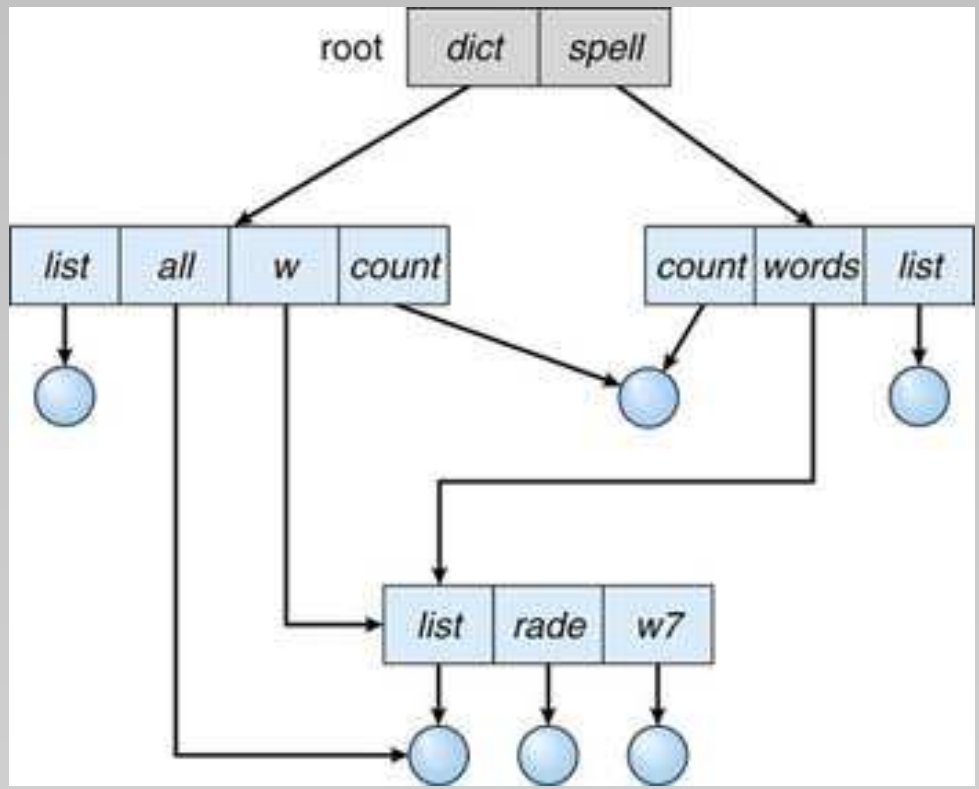


Figure: Acyclic-graph directory structure.

Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.

Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- It is important to note that a shared file (or directory) is not the same as two copies of the file.

Acyclic-Graph Directories II

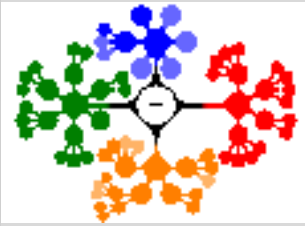
- The same file or subdirectory may be in two different directories.
- It is important to note that a shared file (or directory) is not the same as two copies of the file.
  - With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy.

Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- 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.

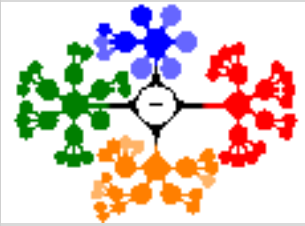
Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- 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.
- A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.



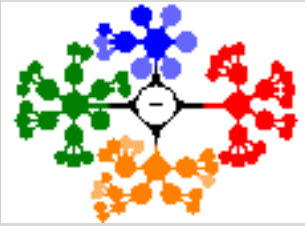
Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- 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.
- A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.
- A link is effectively a pointer to another file or subdirectory.



Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- 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.
- A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.
- A link is effectively a pointer to another file or subdirectory.
- We resolve the link by using that path name to locate the real file.



Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- 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.
- A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.
- A link is effectively a pointer to another file or subdirectory.
- We resolve the link by using that path name to locate the real file.
- Links are easily identified by their format in the directory entry and are effectively named indirect pointers.

Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- 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.
- A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.
- A link is effectively a pointer to another file or subdirectory.
- We resolve the link by using that path name to locate the real file.
- Links are easily identified by their format in the directory entry and are effectively named indirect pointers.
- Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories.

Acyclic-Graph Directories II

- The same file or subdirectory may be in two different directories.
- 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.
- A common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.
- A link is effectively a pointer to another file or subdirectory.
- We resolve the link by using that path name to locate the real file.
- Links are easily identified by their format in the directory entry and are effectively named indirect pointers.
- Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories.
- Thus, both entries are identical and equal.

Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.

Acyclic-Graph Directories III

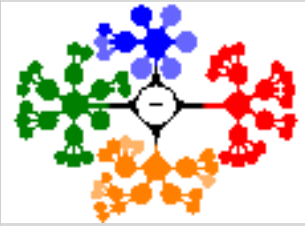
- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.

Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.

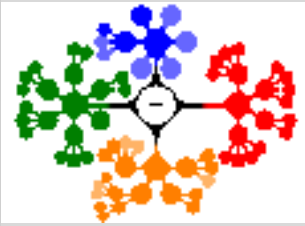
Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.
  - Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?



Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.
  - Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?
    - One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now nonexistent file.

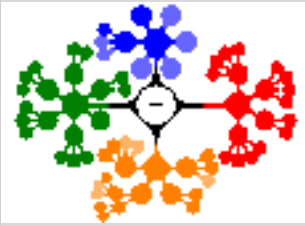


Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.
  - Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?
    - One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now nonexistent file.
    - Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

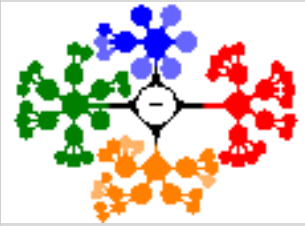
Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.
  - Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?
    - One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now nonexistent file.
    - Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.
- In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle.



Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.
  - Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?
    - One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now nonexistent file.
    - Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.
- In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle.
  - The deletion of a link need not affect the original file; only the link is removed.



Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.
  - Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?
    - One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now nonexistent file.
    - Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.
- In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle.
  - The deletion of a link need not affect the original file; only the link is removed.
  - If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling.

Acyclic-Graph Directories III

- A link is clearly different from the original directory entry; thus, the two are not equal.
- Several problems must be considered carefully for an acyclic-graph directory structure.
  - A file may now have multiple absolute path names.
  - Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused?
    - One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now nonexistent file.
    - Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.
- In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle.
  - The deletion of a link need not affect the original file; only the link is removed.
  - If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling.
- In the case of UNIX, symbolic links are left when a file is deleted. Microsoft Windows (all flavours) uses the same approach.

Acyclic-Graph Directories IV

- Another approach to deletion is to preserve the file until all references to it are deleted.

Acyclic-Graph Directories IV

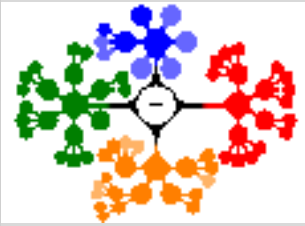
- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.

Acyclic-Graph Directories IV

- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.
- The trouble with this approach is the variable and potentially large size of the file-reference list.

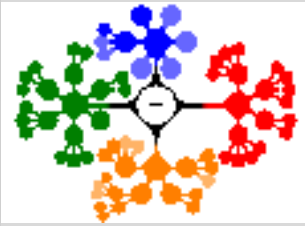
Acyclic-Graph Directories IV

- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.
- The trouble with this approach is the variable and potentially large size of the file-reference list.
- However, we really do not need to keep the entire list -we need to keep only a count of the number of references.



Acyclic-Graph Directories IV

- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.
- The trouble with this approach is the variable and potentially large size of the file-reference list.
- However, we really do not need to keep the entire list -we need to keep only a count of the number of references.
  - Adding a new link or directory entry increments the reference count;



Acyclic-Graph Directories IV

- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.
- The trouble with this approach is the variable and potentially large size of the file-reference list.
- However, we really do not need to keep the entire list -we need to keep only a count of the number of references.
  - Adding a new link or directory entry increments the reference count;
  - Deleting a link or entry decrements the count.

Acyclic-Graph Directories IV

- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.
- The trouble with this approach is the variable and potentially large size of the file-reference list.
- However, we really do not need to keep the entire list -we need to keep only a count of the number of references.
  - Adding a new link or directory entry increments the reference count;
  - Deleting a link or entry decrements the count.
  - When the count is 0, the file can be deleted; there are no remaining references to it.

Dr. Cem Özdoğan

Acyclic-Graph Directories IV

- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.
- The trouble with this approach is the variable and potentially large size of the file-reference list.
- However, we really do not need to keep the entire list -we need to keep only a count of the number of references.
  - Adding a new link or directory entry increments the reference count;
  - Deleting a link or entry decrements the count.
  - When the count is 0, the file can be deleted; there are no remaining references to it.
- The UNIX OS uses this approach for non-symbolic links (or **hard links**), keeping a reference count in the file information block (or **inode**).

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point**.

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point**.
- Typically, a mount point is an empty directory.

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point**.
- Typically, a mount point is an empty directory.
- Next, the OS verifies that the device contains a valid file system.

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point**.
- Typically, a mount point is an empty directory.
- Next, the OS verifies that the device contains a valid file system.
- Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point.

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point**.
- Typically, a mount point is an empty directory.
- Next, the OS verifies that the device contains a valid file system.
- Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point.
- To illustrate file mounting, consider the file system depicted in Fig. 13.

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point**.
- Typically, a mount point is an empty directory.
- Next, the OS verifies that the device contains a valid file system.
- Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point.
- To illustrate file mounting, consider the file system depicted in Fig. 13.
- At this point, only the files on the existing file system can be accessed.

File-System Mounting I

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The OS is given the name of the device and the **mount point**.
- Typically, a mount point is an empty directory.
- Next, the OS verifies that the device contains a valid file system.
- Finally, the OS notes in its directory structure that a file system is mounted at the specified mount point.
- To illustrate file mounting, consider the file system depicted in Fig. 13.
- At this point, only the files on the existing file system can be accessed.
- Figure 14 shows the effects of mounting the volume residing on */dev/disk* over */users*.

File-System Mounting II

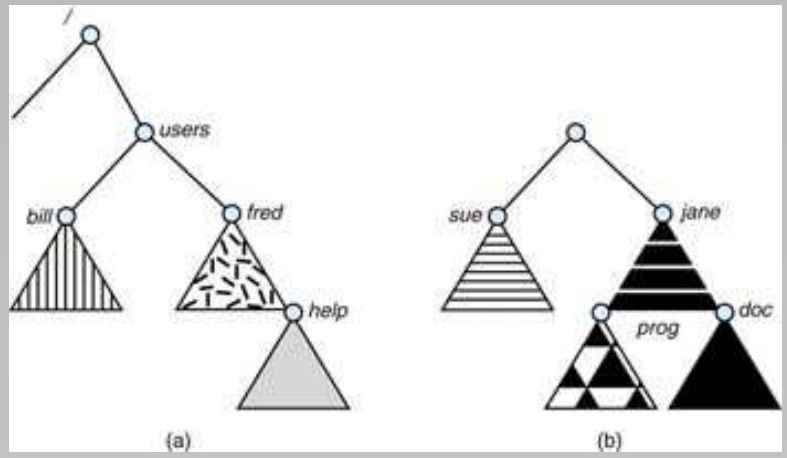


Figure: File system. (a) Existing system. (b) Unmounted volume.

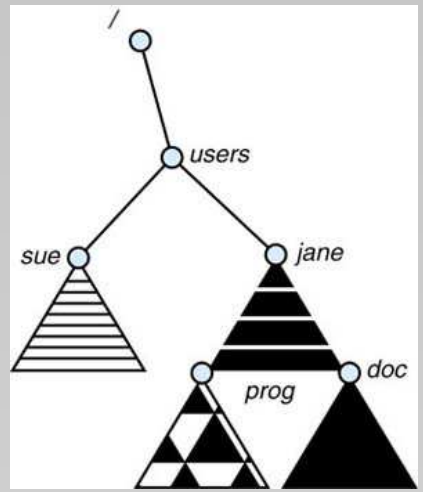
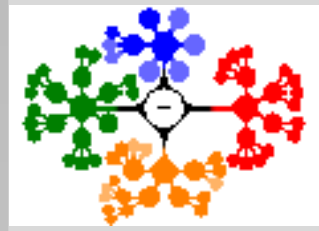


Figure: Mount point.



Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.
  - The owner is the user who can change attributes and grant access and who has the most control over the file.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.
  - The owner is the user who can change attributes and grant access and who has the most control over the file.
  - The group attribute defines a subset of users who can share access to the file.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.
  - The owner is the user who can change attributes and grant access and who has the most control over the file.
  - The group attribute defines a subset of users who can share access to the file.
- For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.
  - The owner is the user who can change attributes and grant access and who has the most control over the file.
  - The group attribute defines a subset of users who can share access to the file.
- For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations.
- The owner and group IDs of a given file (or directory) are stored with the other file attributes.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.
  - The owner is the user who can change attributes and grant access and who has the most control over the file.
  - The group attribute defines a subset of users who can share access to the file.
- For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations.
- The owner and group IDs of a given file (or directory) are stored with the other file attributes.
- When a user requests an operation on a file, the *user ID can be compared with the owner attribute* to determine if the requesting user is the owner of the file.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.
  - The owner is the user who can change attributes and grant access and who has the most control over the file.
  - The group attribute defines a subset of users who can share access to the file.
- For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations.
- The owner and group IDs of a given file (or directory) are stored with the other file attributes.
- When a user requests an operation on a file, the *user ID can be compared with the owner attribute* to determine if the requesting user is the owner of the file.
- Likewise, the group IDs can be compared.

Multiple Users

- When an OS accommodates multiple users, the issues of file sharing, file naming, and file protection become important.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user*) and *group*.
  - The owner is the user who can change attributes and grant access and who has the most control over the file.
  - The group attribute defines a subset of users who can share access to the file.
- For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations.
- The owner and group IDs of a given file (or directory) are stored with the other file attributes.
- When a user requests an operation on a file, the *user ID can be compared with the owner attribute* to determine if the requesting user is the owner of the file.
- Likewise, the group IDs can be compared.
- The result indicates which permissions are applicable.

Protection

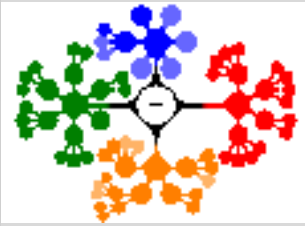
- When information is stored in a computer system, we want to keep it safe from **physical damage (reliability)** and **improper access (protection)**.

Protection

- When information is stored in a computer system, we want to keep it safe from **physical damage (reliability)** and **improper access (protection)**.
- Reliability is generally provided by duplicate copies of files (copy disk files to tape).

Protection

- When information is stored in a computer system, we want to keep it safe from **physical damage (reliability)** and **improper access (protection)**.
- Reliability is generally provided by duplicate copies of files (copy disk files to tape).
- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism.



Protection

- When information is stored in a computer system, we want to keep it safe from **physical damage (reliability)** and **improper access (protection)**.
- Reliability is generally provided by duplicate copies of files (copy disk files to tape).
- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism.
- Files may be deleted accidentally.

Protection

- When information is stored in a computer system, we want to keep it safe from **physical damage (reliability)** and **improper access (protection)**.
- Reliability is generally provided by duplicate copies of files (copy disk files to tape).
- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism.
- Files may be deleted accidentally.
- Bugs in the file-system software can also cause file contents to be lost.

Types of Access

- The need to protect files is a direct result of the ability to access files.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.
  - **Execute**. Load the file into memory and execute it.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.
  - **Execute**. Load the file into memory and execute it.
  - **Append**. Write new information at the end of the file.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.
  - **Execute**. Load the file into memory and execute it.
  - **Append**. Write new information at the end of the file.
  - **Delete**.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.
  - **Execute**. Load the file into memory and execute it.
  - **Append**. Write new information at the end of the file.
  - **Delete**.
  - **List**. List the name and attributes of the file.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.
  - **Execute**. Load the file into memory and execute it.
  - **Append**. Write new information at the end of the file.
  - **Delete**.
  - **List**. List the name and attributes of the file.
- Other operations, such as renaming, copying, and editing the file, may also be controlled.

Types of Access

- The need to protect files is a direct result of the ability to access files.
  - Systems that do not permit access to the files of other users do not need protection.
  - Alternatively, we could provide free access with no protection.
- Both approaches are too extreme for general use. What is needed is **controlled access**.
- Several different types of operations may be controlled:
  - **Read**.
  - **Write**.
  - **Execute**. Load the file into memory and execute it.
  - **Append**. Write new information at the end of the file.
  - **Delete**.
  - **List**. List the name and attributes of the file.
- Other operations, such as renaming, copying, and editing the file, may also be controlled.
- These higher-level functions may be implemented by a system program that makes lower-level system calls.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
- These problems can be resolved by use of a condensed version of the access list.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
- These problems can be resolved by use of a condensed version of the access list.
- To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
- These problems can be resolved by use of a condensed version of the access list.
- To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:
  - **Owner**. The user who created the file is the owner.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
- These problems can be resolved by use of a condensed version of the access list.
- To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:
  - **Owner**. The user who created the file is the owner.
  - **Group**. A set of users who are sharing the file and need similar access is a group, or work group.

Access Control I

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
- These problems can be resolved by use of a condensed version of the access list.
- To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:
  - **Owner**. The user who created the file is the owner.
  - **Group**. A set of users who are sharing the file and need similar access is a group, or work group.
  - **Universe**. All other users in the system constitute the universe.

Access Control II

- With this more limited protection classification, only three fields are needed to define protection.

Access Control II

- With this more limited protection classification, only three fields are needed to define protection.
- Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it.

Access Control II

- With this more limited protection classification, only three fields are needed to define protection.
- Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it.
- For example, the UNIX system defines three fields of 3 bits each—*rwx*, where *r* controls read access, *w* controls write access, and *x* controls execution.

Access Control II

- With this more limited protection classification, only three fields are needed to define protection.
- Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it.
- For example, the UNIX system defines three fields of 3 bits each—*rwx*, where *r* controls read access, *w* controls write access, and *x* controls execution.
- In this scheme, nine bits per file are needed to record protection information.