File Management

Chapter 12

Introduction (1)

- **All computer applications need to store and retrieve information**
  - **How about storing information within a process' address space?**
    - **The storage capacity is limited**
    - **After process termination, the information is lost**
    - **If we have information stored inside the address space of a single process, only that process can access it.**

Introduction (2)

- **Abstractions**
  - The concept of processors → the abstraction of processes
  - The concept of physical memory → the virtual address spaces
  - The concept of disks (or sometimes I/O devices) → the abstraction of files
- **Each file can be thought of as a kind of address space**
  - Files are used to model the disk instead of modeling the RAM

File System

- *For most users, the file system is the most visible aspect of an operating system.*
- *The file system consists of two distinct parts:*
  - *A collection of files: each storing related data*
    - *In the general view, a file is a container for a collection of information*
  - *A directory structure: organizes and provides information for all the files in the system*

Overview

What is a file?

- **What is a file?**
  - **A file is a named collection of related information that is recorded on secondary storage.**

- **Data cannot be written to secondary storage unless they are within a file.**

File Naming (1)
文件命名

- **Files are an abstract mechanism, providing a way to store/read information on/from the disk**
  - **When a process creates a file, it gives the file a name.**
  - **When the process terminates, the file continues to exist and can be accessed by other processes using its name**

File Naming (2)

- The exact rule for file naming vary from system to system
  - All current operating systems allow strings of one to eight letters as legal file names
  - Some file systems (UNIX) distinguish between upper and lower case letters, whereas others (MS-DOS) do not
  - Many operating systems support two-part file names, separated by a period. The part following the period is called the file extension
    - In some system (UNIX), file extensions are just conventions and convey no actual information to the computer
    - Windows is aware of the extensions and assigns meaning to them

Two Types of Files

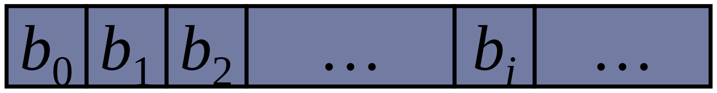
- ***Two types of files:***
  - *Low-Level, Byte-Stream Files* (低级字节流文件)
  - *Structured Files* (结构化文件)

Files
Low-Level Byte-Stream Files

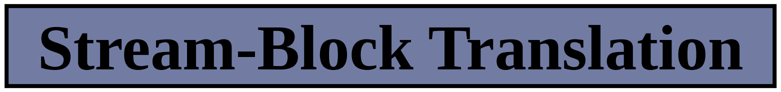
- **Byte-stream file (字节流文件) is a named sequence of bytes indexed by the nonnegative integers**
  - It can be thought of a byte stream as a large array of bytes, each with an index.
- **By convention, every process that opens a file uses a *file position* to reference a particular byte in the file**
  - When the file is opened, the file position references the first byte in the file
  - Each *k*-byte read or write operation advances the file position by a value of *k*.
  - At any given moment after the file has been opened, the file position references the next byte in the file to be read from or written to .

Files
Stream-Block Translation

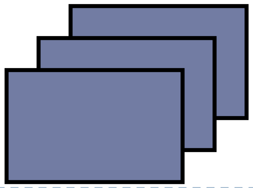
```
fid=open("filename",…);
...
read(fid,buf,buflen);
...
close(fid);
```



```
int open(...){...}
int close(...){...}
int read(...){...}
int write(...){...}
int seek(...){...}
```



Storage device response to commands



Files
Low-Level Byte-Stream Files

- **Typical operations on a byte-stream file**
  - `open(filename)`: This operation prepares the file for reading or writing.
  - `close(fileID)`: deallocates the internal descriptors created when the file is opened, along with any other resources used by the file system to manage the byte-stream I/O.
  - `read(fileID, buffer, length)`: copies a block of `length` (or fewer) bytes beginning at the current file position into the buffer for the specified file identifier, `fileID`.
  - `write(fileID, buffer, length)`: writes the `length` bytes of the information from the buffer to the current file position and then increments the file position by length
  - `seek(fileID, fileposition)`: changes the value of the file position to the value of the parameter.

Files
Structured Files 结构化文件

- **A classic example is ASCII character files (or text files)**
  - There are two assumptions about these text files: (1) The byte stream contains only "printable" ASCII characters; and (2) the characters are arranged into "lines" with each line terminated by the NEWLINE character
- **UNIX file manager does not distinguish text files from other byte streams.**
  - However, several commands (including UNIX system software and library routines) do make the distinction.

Files
Structured Files 结构化文件

- **A structured file can be used to hold any kind of information**
  - Source programs, word processing documents, graphic images, and audio/video streams, etc.
- **The UNIX character file example highlights the fact that**
  - If the file manager does not support data structure, then the applications must provide that capability.
- **Alternatively, the file manager might be designed so that it was able to translate storage blocks to/from application-oriented structured records.**

Files
Structured Files 结构化文件

- **Many applications need to store and access a set of records (record-oriented files)**
- **Terms used:**
  - Field: basic element of data; contains a single value; characterized by its length and data type;
  - Record: collection of related fields; treated as a unit, example: employee record;
  - File: collection of similar records; treated as a single entity; have unique file names; may restrict access
  - Database: collection of related data; relationships exist among elements

Structured Sequential Files

- **A structured sequential file is a named sequence of logical records, indexed by the nonnegative integers.**
  - **Access to the file is defined by a file position, which indexes records in the file instead of bytes.**

Typical Operations on Structured Sequential Files

- **Typical operations on a structured sequential file**
  - `open(filename)`: This operation prepares the file for reading or writing, and returns a file identifier used in the following operations
  - `close(fileID)`: Performs the same function as the `close()` for byte-stream files.
  - `getrecord(fileID, record)`: Returns the record addressed by file position.
  - `putrecord(fileID, record)`: Writes the designated `record` at the current position.
  - `seek(fileID, position)`: Moves the file position to point at the designated record.

These operations are equivalent to the operations for the byte-stream file, except the data are stored in records instead of bytes

File Organization and Access

This part is only related to record-oriented files

What does “file organization” mean?

- ***File Organization* refers to the *logical* structuring of the records as determined by the way in which they are *accessed*.**

- ***Physical organization* of the file on secondary storage depends on the *blocking strategy* (组块策略) and the *file allocation strategy* (文件分配策略)**
  - *To be discussed a little later*

Criteria for File Organization

- **Rapid access**
  - Needed when accessing a single record
  - Not needed for batch mode
- **Ease of update**
  - File on CD-ROM will not be updated, so this is not a concern

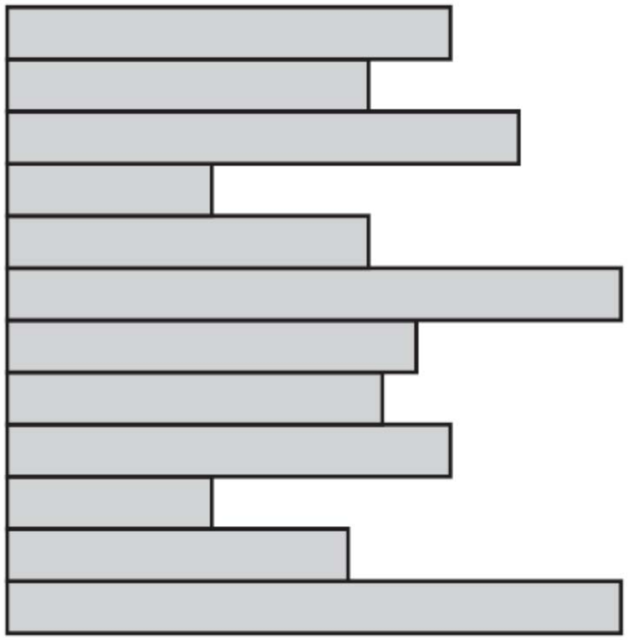
Five Fundamental Organizations

- The pile (堆文件)
- The sequential file (顺序文件)
- The indexed sequential file (索引顺序文件)
- The indexed file (索引文件)
- The direct, or hashed, file (直接或哈希文件)

File Organization - I
The Pile (1)

- Data are collected in the order they arrive
  - Purpose is to accumulate a mass of data and save it
- Records may have different fields, or similar fields in different orders
  - Thus, each field should be self-described, including a field name as well as a value
- There is no structure to the pile file
  - Thus, record access is by exhaustive search

File Organization - I
The Pile (2)


Variable-length records
Variable set of fields
Chronological order

(a) Pile File

When data are collected and stored prior to processing or when data are not easy to organize, we can use this type of files

- This type of files uses space well
- It is perfectly adequate for exhaustive searches
- It is easy to update

However, beyond these limited uses, it is unsuitable for most applications

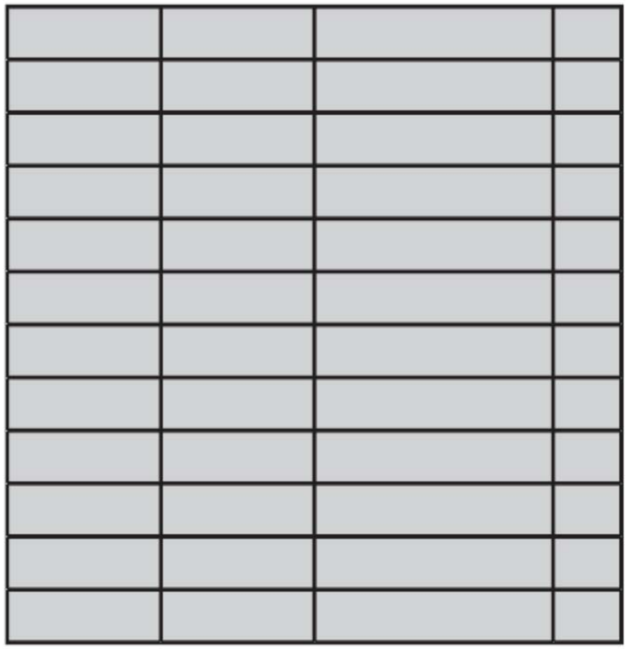
File Organization - II
The Sequential File (1)

- **In a sequential file, a fixed format is used for records**
  - All records are of the same length, consisting of the same number of fixed-length fields in a particular order
- **The field names and lengths are attributes of the file structure**
- **One field is the key field**
  - The key field uniquely identifies the record
  - Records are stored in key sequence:
    - alphabetical order for a text key, and numerical order for a numerical key

File Organization - II
The Sequential File (2)

- **Sequential files are typically used in batch applications**
  - They are generally optimum for such applications that need to process all the records
- **Sequential files provide *poor performance* for interactive applications that involve queries and/ or updates of individual records**
- **Additions to the file also present problems.**
  - How to solve this problem?

File Organization - II
The Sequential File (3)



Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

How to solve the problem of additions to a sequential file?

(1) New records are placed in a log file or transaction file

(2)Periodically, batch update is performed to merge the log file with the master file

File Organization - III
The Indexed Sequential File (1)

- **The indexed sequential file maintains the key characteristic of the sequential file:**
  - Records are organized in sequence based on a key field
- **Two features are added**
  - An index to the file to support random access
    - The index provides a lookup capability to reach quickly the vicinity of a desired record
  - An overflow file: similar to the log file used with a sequential file but is integrated so that records in the overflow file are located by following a pointer from their predecessor record

File Organization - III
The Indexed Sequential File (2)

- **Single-level indexing: the index is a simple sequential file**
  - Each record in the index file consists of two fields:
    - a key field that is the same as the key field in the main file
    - a pointer into the main file
- **To find a desired key value**
  - The index is searched to find highest key value that is equal or less than the desired key value
  - Search continues in the main file at the location indicated by the pointer

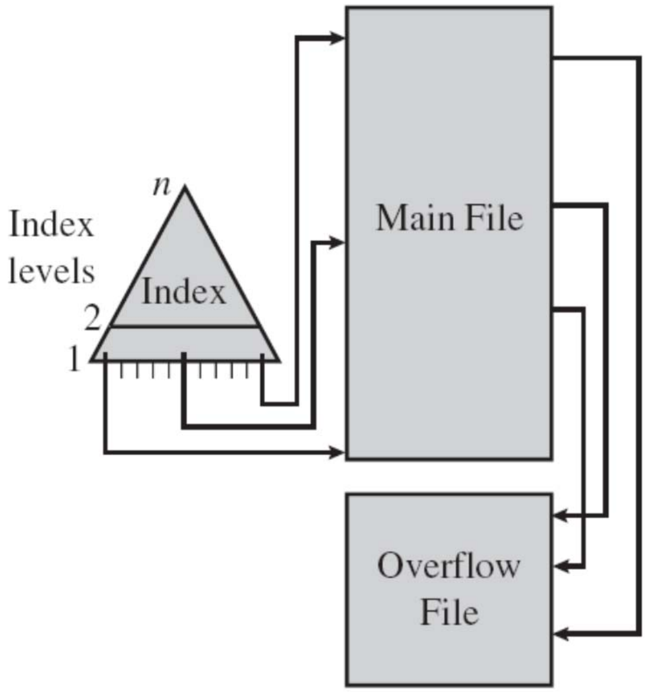
File Organization - III
The Indexed Sequential File (3)

- **Comparison of sequential and indexed sequential**
  - Example: a file contains 1 million records
  - On average 500,000 accesses are required to find a record in a sequential file
  - If an index contains 1000 entries, it will take on average 500 accesses to find the key, followed by 500 accesses in the main file. Now on average it is 1000 accesses

File Organization - III
The Indexed Sequential File (4)

- **Each record in the main file contains an additional field which is a pointer to the overflow file**
  - Invisible to the application
- **Addition to the indexed sequential file:**
  - The new record is added to an overflow file
  - The record in main file that immediately precedes the new record is updated to contain a pointer to the new record
  - If the immediately preceding record is itself in the overflow file, then the pointer in that record is updated.
- **The overflow is merged with the main file during a batch update**

File Organization - III
The Indexed Sequential File (5)



(c) Indexed Sequential File

To provide even greater efficiency in access, multiple levels of indexing can be used.

1000,000 records

10,000 entries in the lower-level index

100 entries in the upper-level index

How about the average length of search?

File Organization - IV
The Indexed File (1)

- **Uses multiple indexes for different fields**
  - One index for each field that may be the subject of a search
- **Records are accessed only through their indexes**
  - Any record should be referred to by a pointer in at least one index
- **There is no restriction on the placement of records**
- **Variable-length records can be employed.**

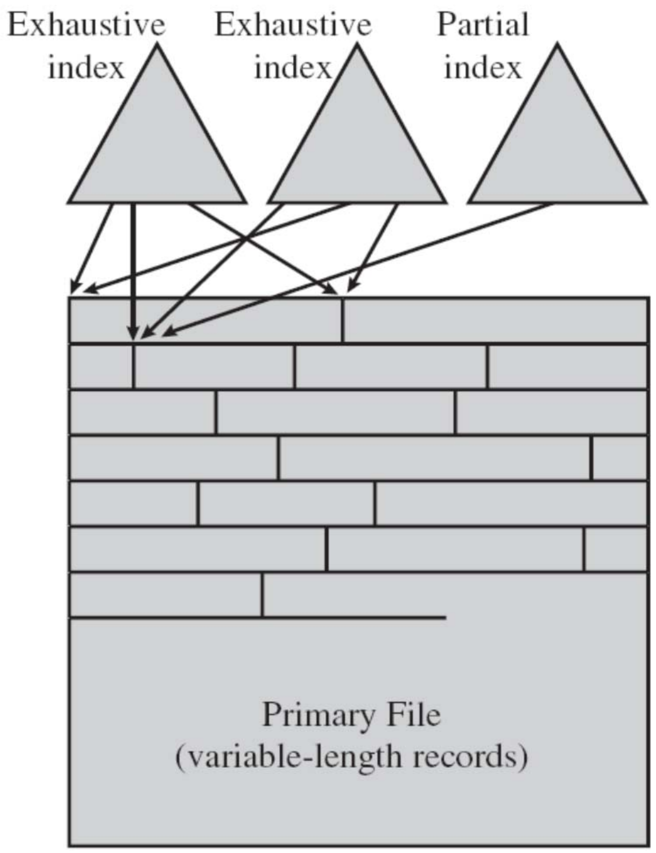
File Organization - IV
The Indexed File (2)

- **Two types of Indexes**
  - Exhaustive index: contains one entry for every record in the main file
  - Partial index: contains entries to records where the field of interests exists
- **When a new record is added to the main file, all of the index files must be updated**

File Organization - IV
The Indexed File (3)

Indexed files are used mostly in applications where timeliness of information is critical and where data are rarely processed exhaustively.

Examples: airline reservation systems and inventory control systems

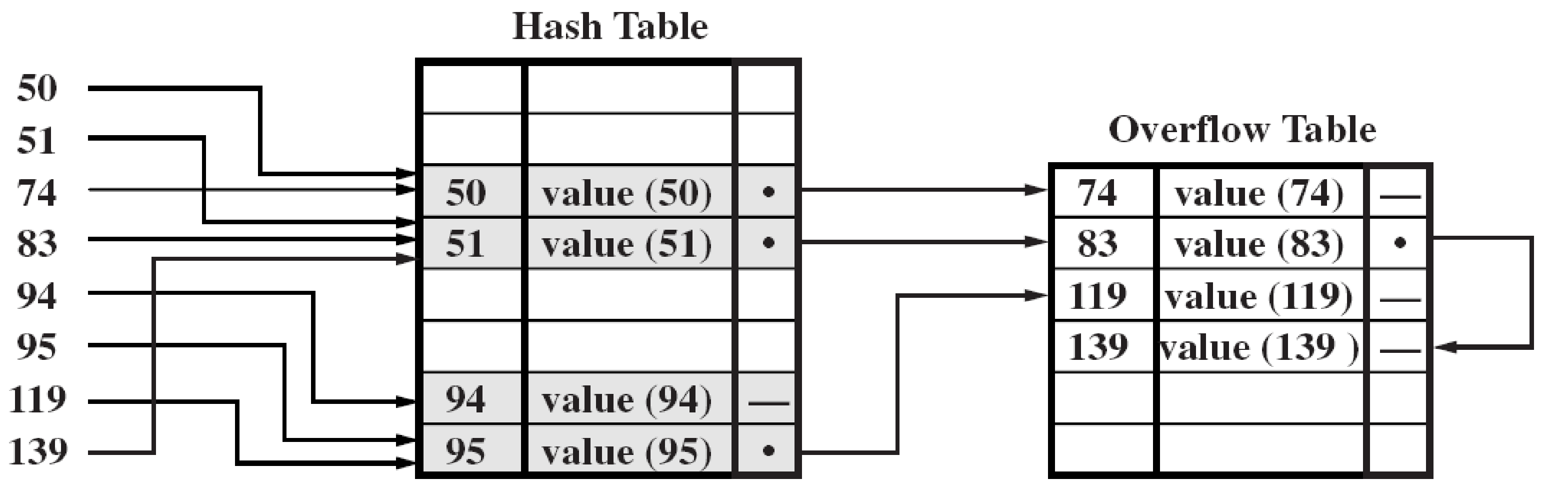


(d) Indexed File

File Organization - V
The Direct or Hashed File (1)

- The direct or hashed file exploits the capability found on disks to access directly any block at a known address
- A key field is required in each record (it is similar with sequential and indexed sequential files)
  - However, there is no concept of sequential ordering here.
- The direct file makes use of hashing on the key value.

File Organization - V
The Direct or Hashed File (2)



(b) Overflow with chaining

Direct files are often used where very rapid access is required, where fixed length records are used, and where records are always accessed one at a time.

- Examples: directories, pricing tables, schedules, and name lists

	Space		Update		Retrieval		
	Attributes		Record Size				
File Method	Variable	Fixed	Equal	Greater	Single record	Subset	Exhaustive
Pile	A	B	A	E	E	D	B
Sequential	F	A	D	F	F	D	A
Indexed sequential	F	B	B	D	B	D	B
Indexed	B	C	C	C	A	B	D
Hashed	F	B	B	F	B	F	E

A	=	Excellent, well suited to this purpose	$\approx O(r)$
B	=	Good	$\approx O(o \times r)$
C	=	Adequate	$\approx O(r \log n)$
D	=	Requires some extra effort	$\approx O(n)$
E	=	Possible with extreme effort	$\approx O(r \times n)$
F	=	Not reasonable for this purpose	$\approx O(n^{>1})$

where

r = size of the result
o = number of records that overflow
n = number of records in file

File Access
文件存取/访问

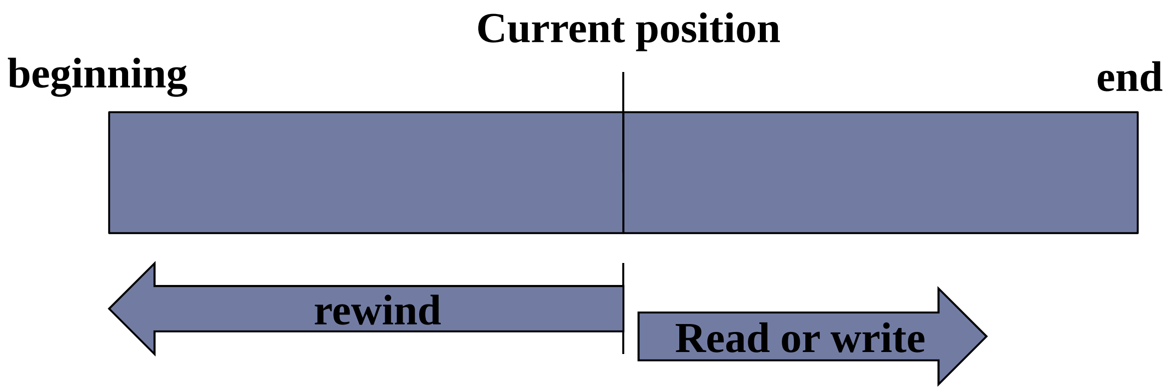
Two Kinds of File Access
Sequential Access and Direct Access

- *Sequential access* files: all the bytes or records in a file are read in order, starting at the beginning
  - Early OSes provided only sequential access
  - Sequential files were convenient when the storage medium was magnetic tape.
- *Random (or direct) access* files: read the bytes or records of a file in no particular order
  - With disks coming into use for storing files, it became possible

Sequential Access

- **The bulk of the operations on a file is reads and writes.**
  - **A read operation reads the next portion of the file and automatically advances a file pointer**
  - **A write operation appends to the end of the file and advances to the end of the newly written material (the new end of file).**
- **Such a file can be reset to the beginning**
  - **On some systems, a program may be able to skip forward or backward *n* records for some integer *n*.**

Sequential-Access File



Direct (or Relative) Access

- **A direct-access file allows arbitrary blocks to be read or written.**
  - The file is viewed as numbered sequence of blocks or records.
  - There is 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**
  - Databases are often of this type.

Direct (or Relative) Access

- For the direct-access method, the file operations must be modified to include the block number as a parameter
  - Thus, we have to `read n`, where `n` is the relative block number, rather than `read next`; and `write n` rather than `write next`.
  - Alternatively, to retain `read next` and `write next`, we can add an operation `position file to n`.
- A relative block number is an index relative to the beginning of the file.

Simulation of Sequential Access on a Direct-Access File

Sequential Access	Implementation for direct acess
`reset`	`cp = 0;`

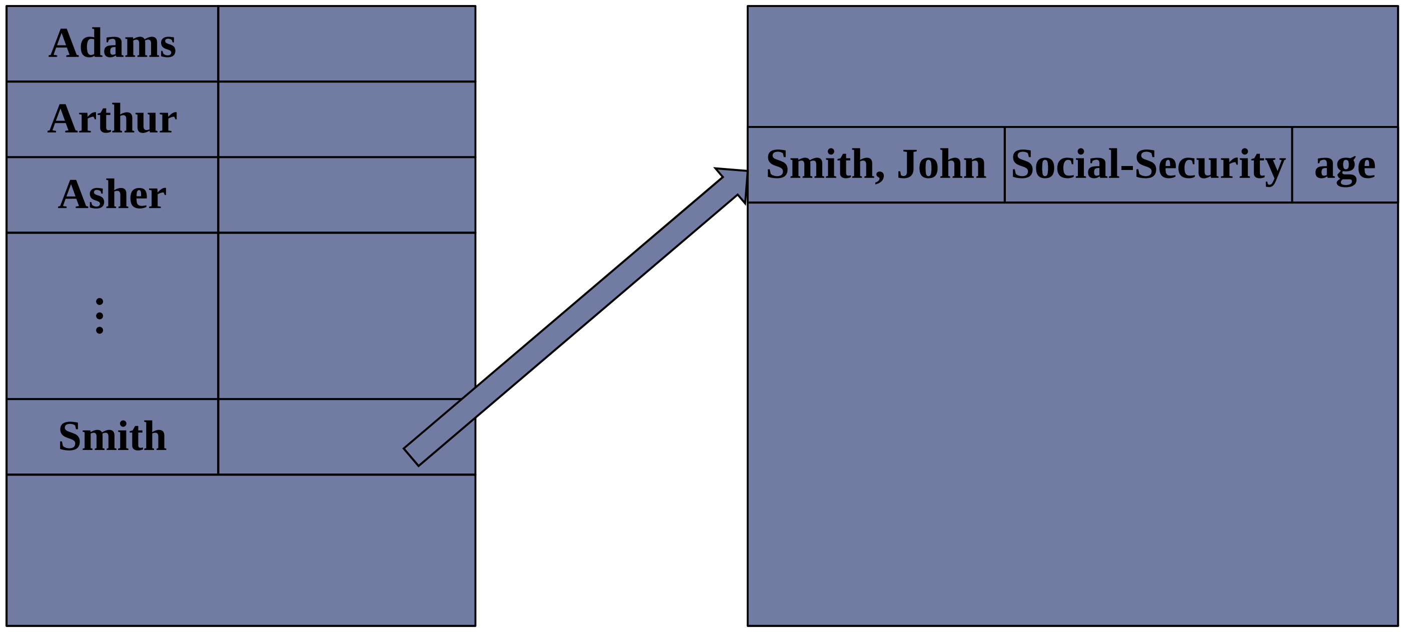
`read next`	`read cp;`
	`cp = cp +1;`

`write next`	`write cp;`
	`cp = cp +1;`

Other Access Methods

- **Other access methods can be built on top of a direct-access method.**
  - These methods generally involve the construction of an index for the file
  - The index contains pointers to the various blocks
  - To find a record in the file, we first search the index, and then use the pointer to access the file directly and to find the desired record.

Example of Index and Relative Files



File Directories
文件目录

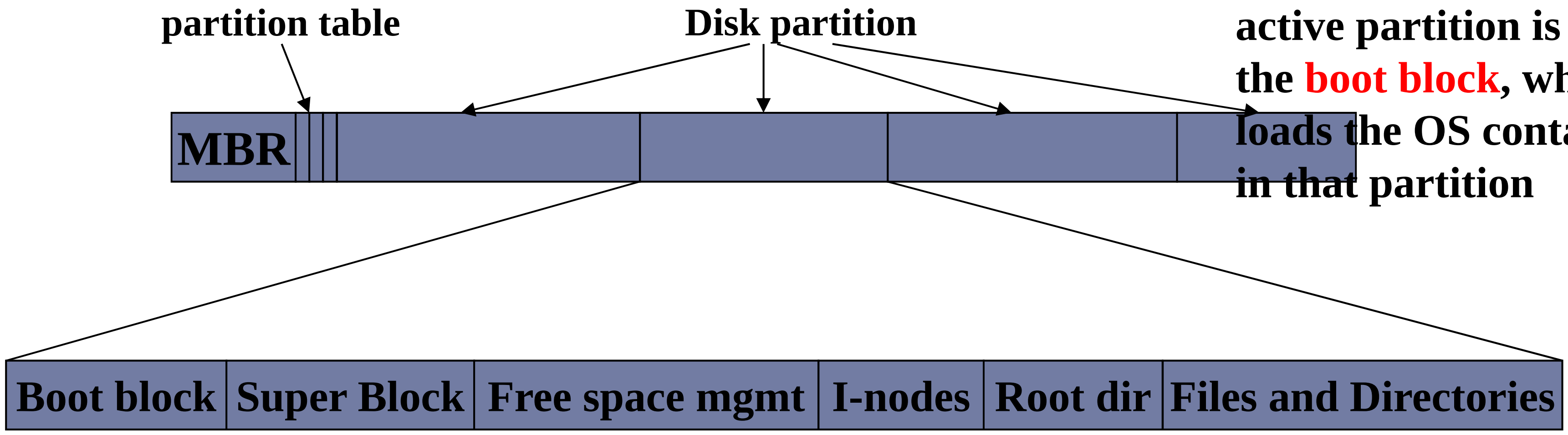
Partitions and Directories
分区与目录

- **To manage millions of files on disk, the organization is usually done in two parts**
  - *Disks are split into one or more partitions*
    - **Partitions are also known as minidisks in the IBM world or volumes (卷) in the PC and Macintosh arenas.**
  - *Each partition contains information about files within it.*
    - **This information is kept in entries in a device directory or volume table of contents.**

File System Layout

MBR (Master Boot Record): **The sector 0 of the disk; used to boot the computer**

Partition table: **contained at the end of the MBR; the table gives the starting and ending addresses of each partition; one of the partition in the table is marked as active.**



The first block of the active partition is called the boot block, which loads the OS contained in that partition

The superblock contains all the key parameters about the file system and is read into memory when the computer is booted.

Typical information includes the file system type, the number of blocks in the file system, and other key administrative information

Contents of File Directories

- The directory contains information about the files, including
  - Attributes
  - Location
  - Ownership
- Directories (in many systems) may be themselves files owned by the operating system
  - Users can not directly access the directory even in read-only mode
    - Some information in directories are provided indirectly to users by system routines
- Provides mapping between file names and the files themselves
  - Thus, directory can be viewed as a symbol table that translates file names into their directory entries.

Information in a File Directory (1)

- **Basic Information**
  - File Name: Must be unique within a specific directory
  - File Type: for example, text, binary, load module, etc.
  - File Organization: for systems that support multiple organizations
- **Address Information**
  - Volume: indicates device on which file is stored
  - Starting address: starting physical address on secondary storage
  - Size used: current size of the file in bytes, words, or blocks
  - Size allocated: the maximum size of the file

Information in a File Directory (2)

- **Access control information**
  - Owner: user who is assigned control of this file
  - Access information: user's name and password for each authorized user
  - Permitted action: controls reading, writing, executing, transmitting over a network
- **Usage Information**
  - Date created, identity of creator, date last read access, identity of last reader, date last modified, identity of last modifier, date of last backup, current usage

Directory Structure
Operations Performed on Directory

- **When considering a particular directory structure, it is helpful to keep in mind the operations performed on a directory**
  - Search for a file
  - Create a file
  - Delete a file
  - List directory:
  - Update directory

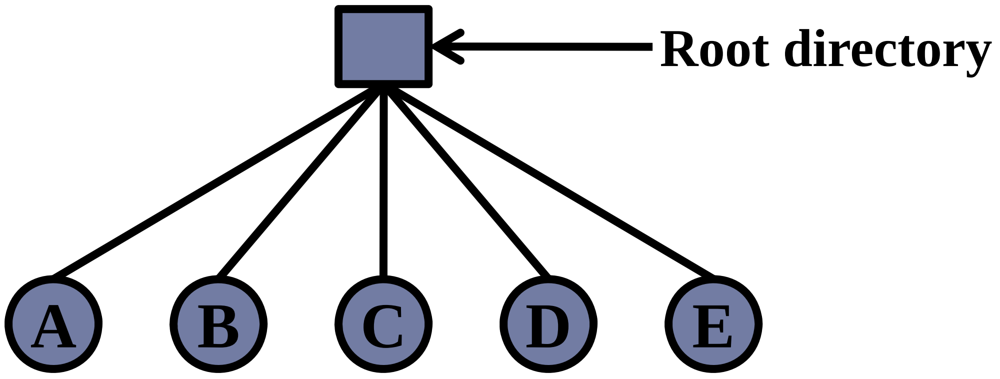
Logical Structure of a Directory

- **The most common schemes for the logical structure of a directory:**
  - Single-Level Directory
  - Two-Level Directory
  - Tree-Structured Directories
  - Acyclic-Graph Directories

Single-Level Directory - I

- **The simplest directory structure is a list of entries, one for each file**
  - This structure could be represented by a simple sequential file, with the file name serving as the key
- **Disadvantages:**
  - **Grouping problem:** Provides no help in organizing the files (the user may wish to organize the files into projects)
  - **Naming problem:** Forces user to be careful not to use the same name for two different files
    - Even worse on a multi-user shared system.

Single-Level Directory - II



A single-level directory system containing five files

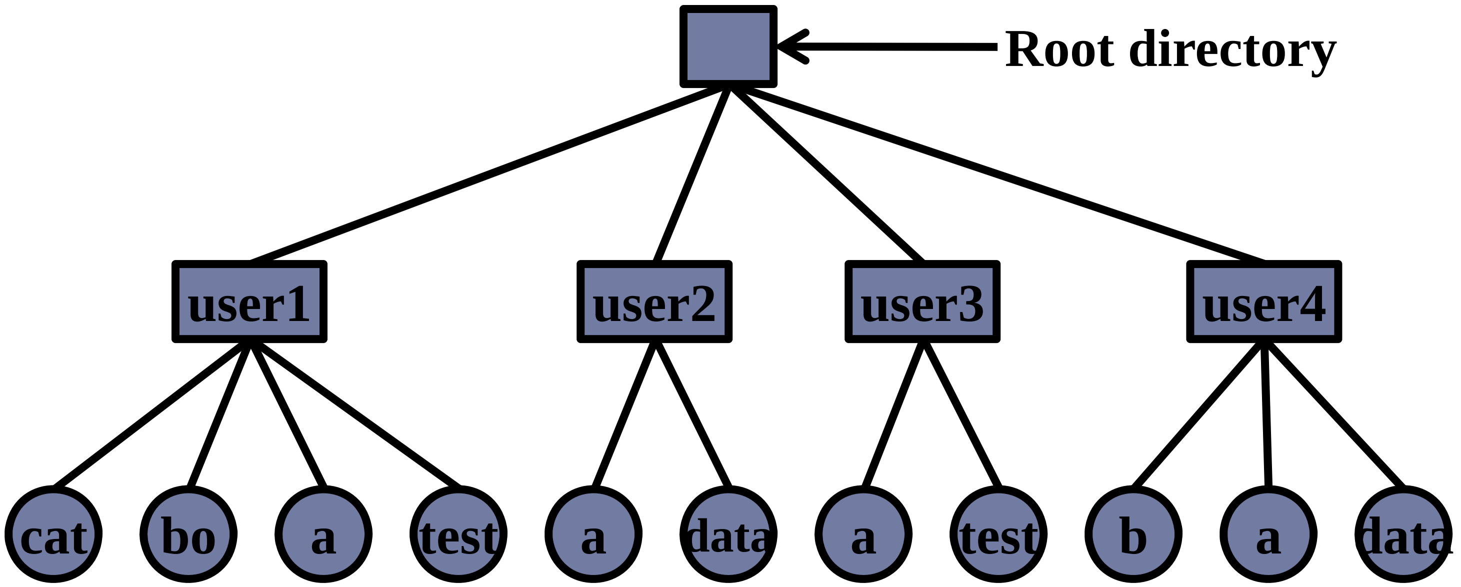
Two-level Directory - I

- In the two level scheme, there is *one separate directory for each user*, and *a master directory*
  - Master directory contains entry for each user, provides address and access control information
  - Each user directory is a simple list of files for that user
- Advantage
  - This structure effectively isolates one user from another, and thus solves the name-collision problem

Two-level Directory – II Disadvantages

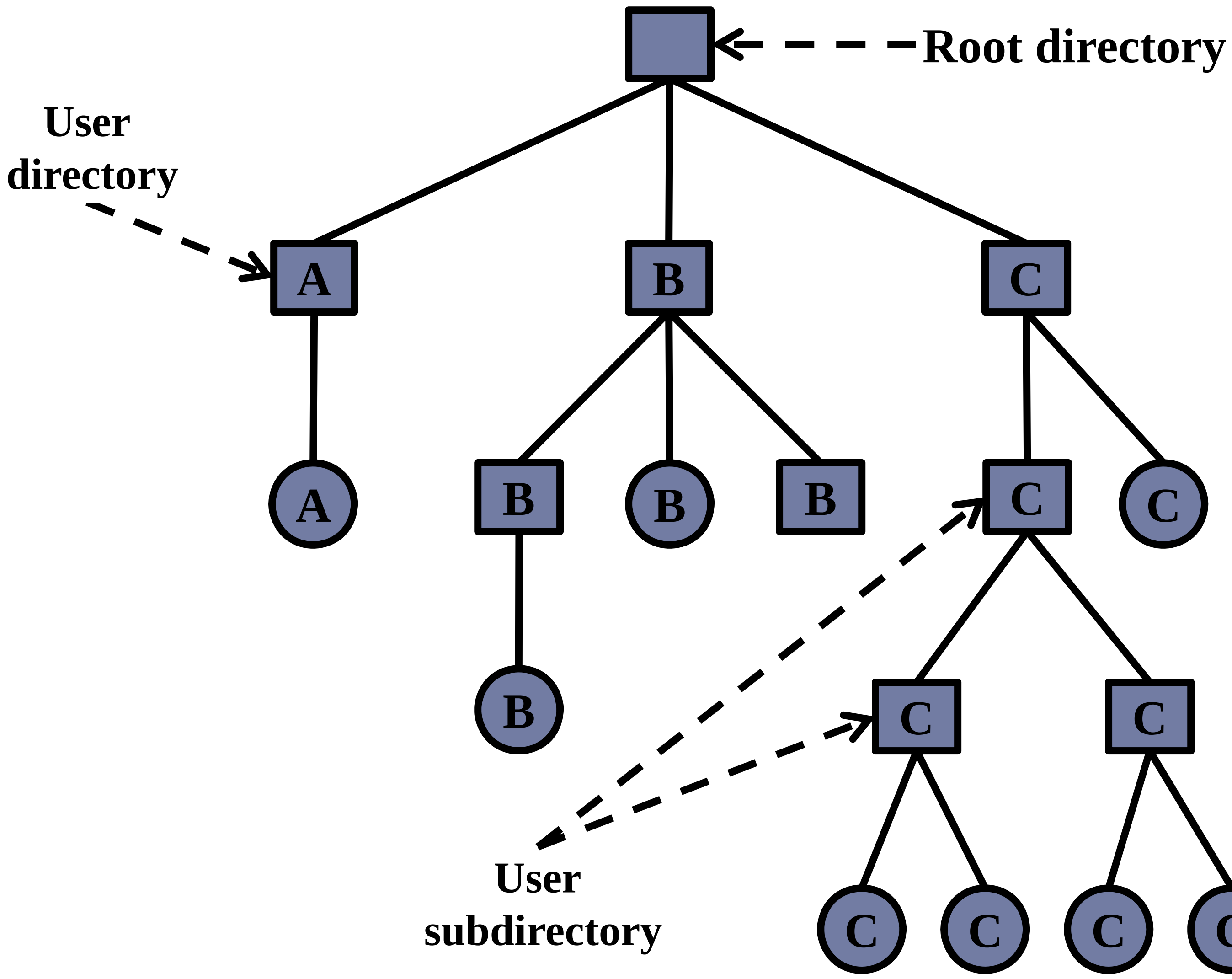
- **File names must be unique within the collection of files of a single user**
  - **A single user may own a large number of files.**
- **Still provides no help in structuring collections of files**
- **What if the users want to cooperate on some task and to access one another's files**
  - **Path name should be defined and used.**
  - **A special case of this situation occurs in regard to system files.**

Two-level Directory - III



Tree-Structured Directories - I

- **The directory structure can be extended to a tree of arbitrary height**
  - A two-level directory can be viewed as a two-level tree.
- **With tree-structured directory:**
  - A directory (or subdirectory) contains a set of files or subdirectories.
  - Each user can create an arbitrary number of subdirectories, which provides a powerful structuring tool for users to organize their files.


Root directory
User directory
User subdirectory
A
A
B
B
B
B
B
C
C
C
C
C
C
C
C
C

Tree-Structured Directory - II Naming

- **Files can be located by following a path from the root (or master) directory down various branches**
  - **This is the pathname for the file**
- **Can have several files with the same file name as long as they have unique path names**

Tree-Structured Directory - III
Path Names

- In normal case, each user has a *current directory or working directory* which should contain most of the files that are of current interest to the user
- Two types of path names:
  - An ***absolute path name*** begins at the root and follows a path down to the specified file, giving the directory names on the path
  - A ***relative path name*** defines a path from the current directory
    - Typically, an interactive user or a process has associated with a current directory, often referred to as working directory

Absolute Path Name

- **Absolute path names always start at the root directory and are unique.**
  - **An example:** */usr/ast/mailbox* means that the root directory contains a subdirectory *usr*, which in turn contains a subdirectory *ast*, which contains the file *mailbox*.
- **Different OSes may use different separators for the components of the path**
  - **UNIX:** /usr/ast/mailbox
  - **Windows:** \usr\ast\mailbox
  - **MULTICS:** >usr>ast>mailbox

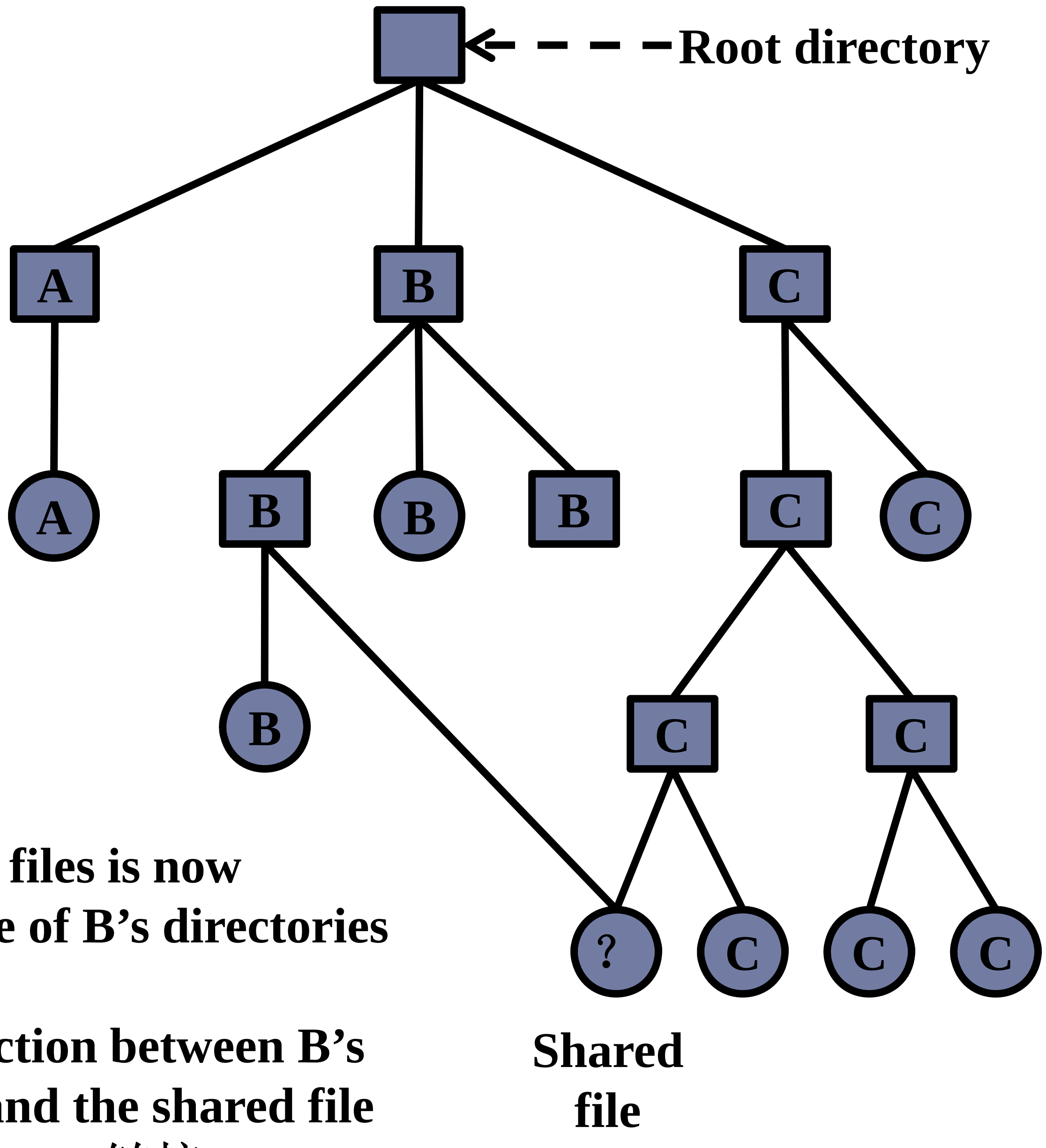
If the first character of the path name is the separator, then the path is absolute.

Relative Path Name

- **The relative path name is used in conjuction with the concept of working directory**
  - **For example: if the current working directory is */usr/ast*, then the file with absolute path name */usr/ast/mailbox* can be referenced simply as *mailbox*.**
- **Most OSes that support a hierarchical directory system have two special entries in every directory, "." and ".."**
  - **Dot refers to the current directory; dotdot refers to its parent (except in the root directory, where it refers to itself)**

Acyclic-Graph Directories (1)
File Sharing

- **When several users are working together on a project, it is often convenient for a shared file/directory to *appear simultaneously* in *different directories belonging to different users***
- **An acyclic-graph allows directories to have shared subdirectories and files.**
  - The same file or subdirectory may be in two (or more) different directories.
  - Two techniques: hard linking and symbolic linking (or soft linking)



One of C’s files is now present one of B’s directories as well.
The connection between B’s directory and the shared file is called a link (链接)

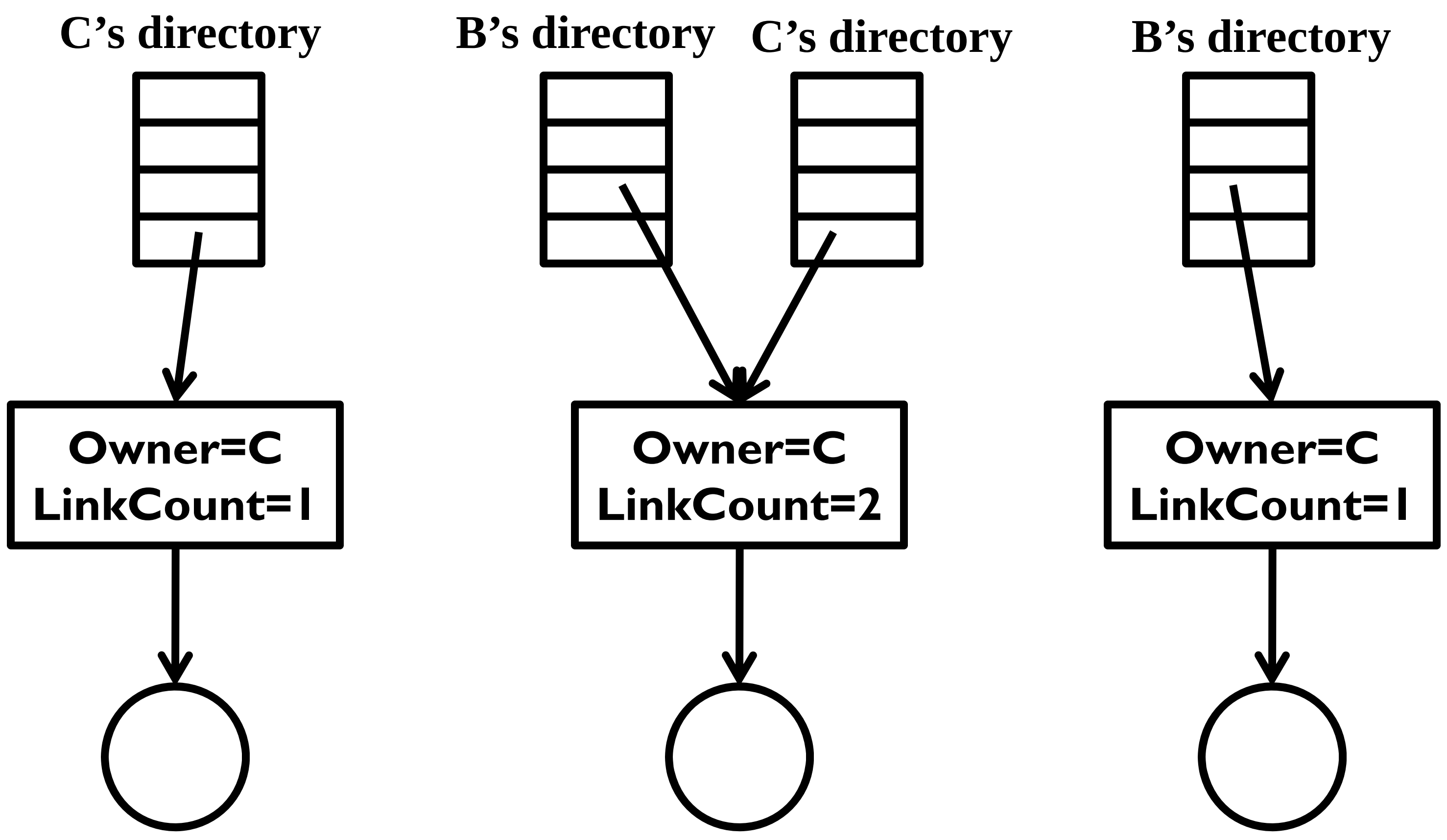
Acyclic-Graph Directories - II
Hard Linking

- **To make user B share one of the user C's files, we can copy the related directory entry of the file into the directory of B.**
  - **Problem 1: what if the directory entry contains the addresses of the data blocks of the file**
  - **Problem 2: what if the user C will delete the file some later?**
- **One solution (adopted by UNIX):**
  - **Disk blocks are not listed in directories, but in a little data structure associated with the file itself**
    - **This little data structure is the i-node**
    - **The directories would then point just to the little data structure.**
  - **The i-node contain a *link-count* field.**

Hard Linking
Should Directories contain Disk Addresses of Files

- **For hard linking, the disk addresses of files should not be present in directory entries**
  - **Otherwise, a copy of the disk addresses will have to be made in B's directory when the file is linked**
    - **If either B or C subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append. This change will be invisible to the other user**
- **Instead, disk blocks are listed in a little data structure associated with the file itself.**
  - **The directories would then point just to the little data structure (UNIX adopts this approach)**

An Example of Hard Linking and Its problem



Situation prior to linking

After the link is created

After the original owner removes the file

Acyclic-Graph Directories - II Symbolic Linking

- **User B links to one of user C’s files by having the system create a new file of type LINK, and entering that file in B’s directory**
  - The new file contains just the path name of the file to which it is linked.
  - When user B reads from the linked file, the OS sees that the file being read from is of type LINK, looks up the name of the file, and reads that file.
- **This approach is called symbolic linking, to contrast it with traditional (hard) linking**

Acyclic-Graph Directories - II
Symbolic Linking

- **With symbolic links, only the true owner has a pointer to the i-node. Users who have linked to the file just have path names, not i-node pointers**
  - When the owner removes the file, it is destroyed. Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file.
  - Removing a symbolic link does not affect the file at all.

Problem with Symbolic Links

- Extra overhead is required
  - The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached.
    - All these activity may require a considerable number of extra disk accesses
  - Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path
    - If the path name is short, the system could store it in the i-node itself, as a kind of optimization

TIME

SPACE

Advantage with Symbolic Links

- They can be used to link to files on machines anywhere in the world,
  - by simply providing the network address of the machine where the file resides in addition to its path on that machine

Directory Implementation (1)

- **The selection of directory-allocation and directory management algorithms has a large effect on the efficiency, performance, and reliability of the file system**
- **Here, we consider two methods of directory implementation**
  - **Linear list**
  - **Hash table**

Directory Implementation (2)
Linear List (线性列表) - I

- The simplest method is to use a linear list of file names with pointers to the data blocks.
  - A linear list of directory entries requires a linear search to find a particular entry.
- It is time-consuming to execute
  - To create a new file: we must first search the directory to be sure that no existing file has the same name
  - To delete a file: we search the directory for the named file, then release the space allocated to it

Directory Implementation (3)
Linear List (线性列表) - II

- **How to reuse a directory entry whose corresponding file has been just deleted?**
  - One way: we can mark the entry as unused (by assigning it a special name, or with a used-unused bit in each entry)
  - Another way: we can attach it to a list of free directory entries
  - A third alternative: to copy the last entry in the directory into the freed location, and to decrease the length of the directory

Directory Implementation (4)
Hash Table

- ***A linear list*** **stores the directory entries, but** ***a hash data structure*** **is also used**
  - **The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list**
  - **It can greatly decrease the directory search time**
- ***The major difficulties*** **are its** ***fixed size*** **and the** ***dependence of the hash function*** **on the size.**

File Protection

Protection

- **Protection: keep files safe from improper access**
  - NOTE: Reliability is to keep files from physical damage
- **Protection of files is a direct result of the ability to share files**
  - Systems that do not permit access to the files of other users do not need protection
  - What we need is controlled access (受控存取)

Access Rights (1)
存取权限

- **None**
  - User may not know of the existence of the file; User is not allowed to read the user directory that includes the file
- **Knowledge**
  - User can only determine that the file exists and who its owner is
- **Execution**
  - The user can load and execute a program but cannot copy it
- **Reading**
  - The user can read the file for any purpose, including copying and execution

Access Rights (2)

- **Appending:**
  - The user can add data to the file but cannot modify or delete any of the file's contents
- **Updating**
  - The user can modify, delete, and add to the file's data. This includes creating the file, rewriting it, and removing all or part of the data
- **Changing protection**
  - User can change access rights granted to other users
  - This right is typically held only by the owner of the file
- **Deletion**
  - User can delete the file from the system

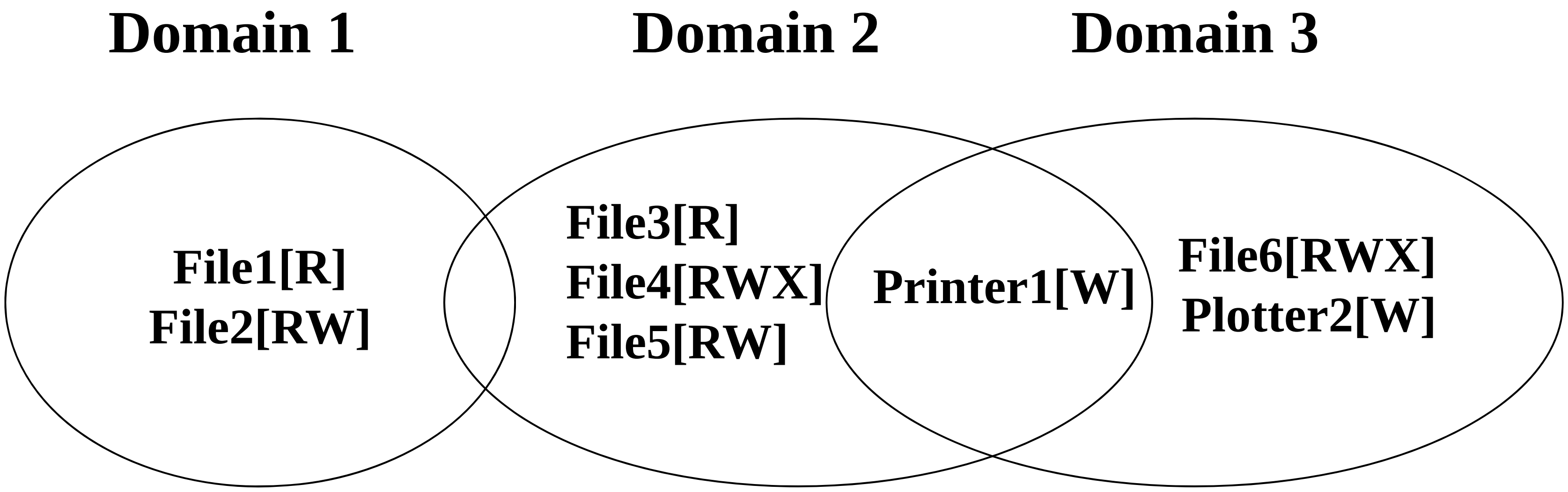
Access Rights (4)

- **All the rights above can be considered to constitute a hierarchy**
  - Each right implies those precede it.
- **If a user is granted the updating right**
  - Then that user is also granted the following rights: knowledge, executing, reading, and appending

Domain

- A computer system contains many "objects"
  - Hardware: CPUs, memory segments, disk drives, or printers
  - Software: processes, files, databases, or semaphores
- Each object has a finite set of operations that can be carried out on it
  - The `read` and `write` operations are appropriate to a file
  - The `wait` and `signal` make sense on a semaphore
- A domain is a set of (object, rights) pairs.
  - Each pair specifies an object and some subset of the operations that can be performed on it.

An Example of Three Protection Domains



Domains in UNIX

- **In UNIX, the domain of a process is defined by its UID and GID.**
  - Given any (UID, GID) combination, it is possible to make a complete list of all objects that can be accessed and whether they can be accessed for reading, writing, or executing
- **Domain Switch**
  - Each process has two halves: the user part and the kernel part. The kernel part has access to a different set of objects from the user part. A system call causes a domain switch
  - When a process does an `exec` on a file with the SETUID or SETGID bit on, it acquires a new effective UID or GID. Running a program with SETUID or SETGID is also a domain switch.

A Protection Matrix

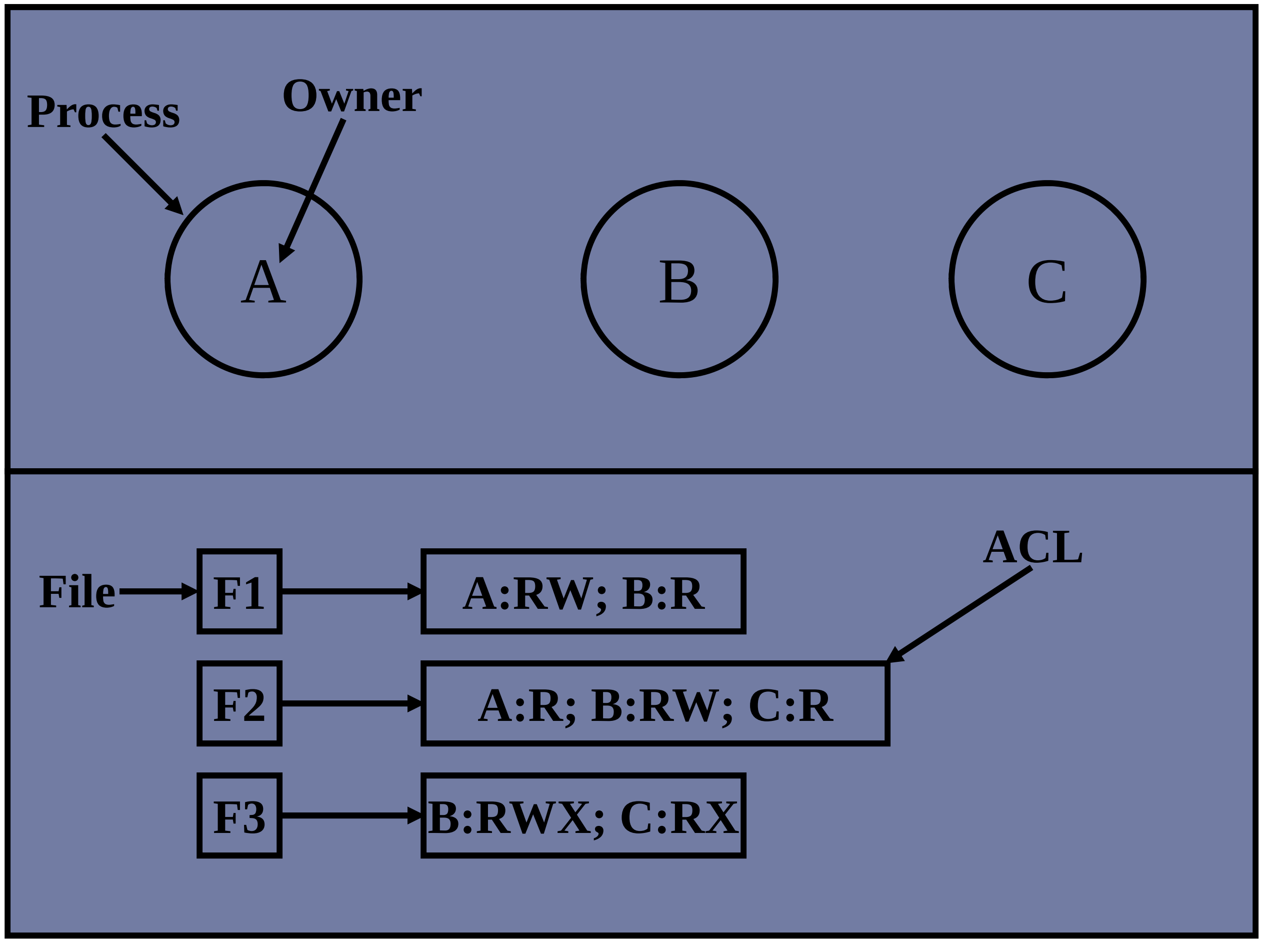
	File1	File2	File3	File4	File5	File6	Printer1	Plotter 2
Domain 1	Read	Read Writ e						
Domain 2			Read	Read Write Execute	Read Writ e		Write	
Domain 3						Read Write Execute	Write	Write

Because this matrix is large and sparse, practical methods usually store only the nonempty elements by rows or by columns

Access Control Lists
存取控制列表

- **Access control lists store the protection matrix by column**
  - **Each object is associated with an (ordered) list containing all the domains that may access the object and how.**
  - **ACL specifies the user name and the access rights allowed for each user.**

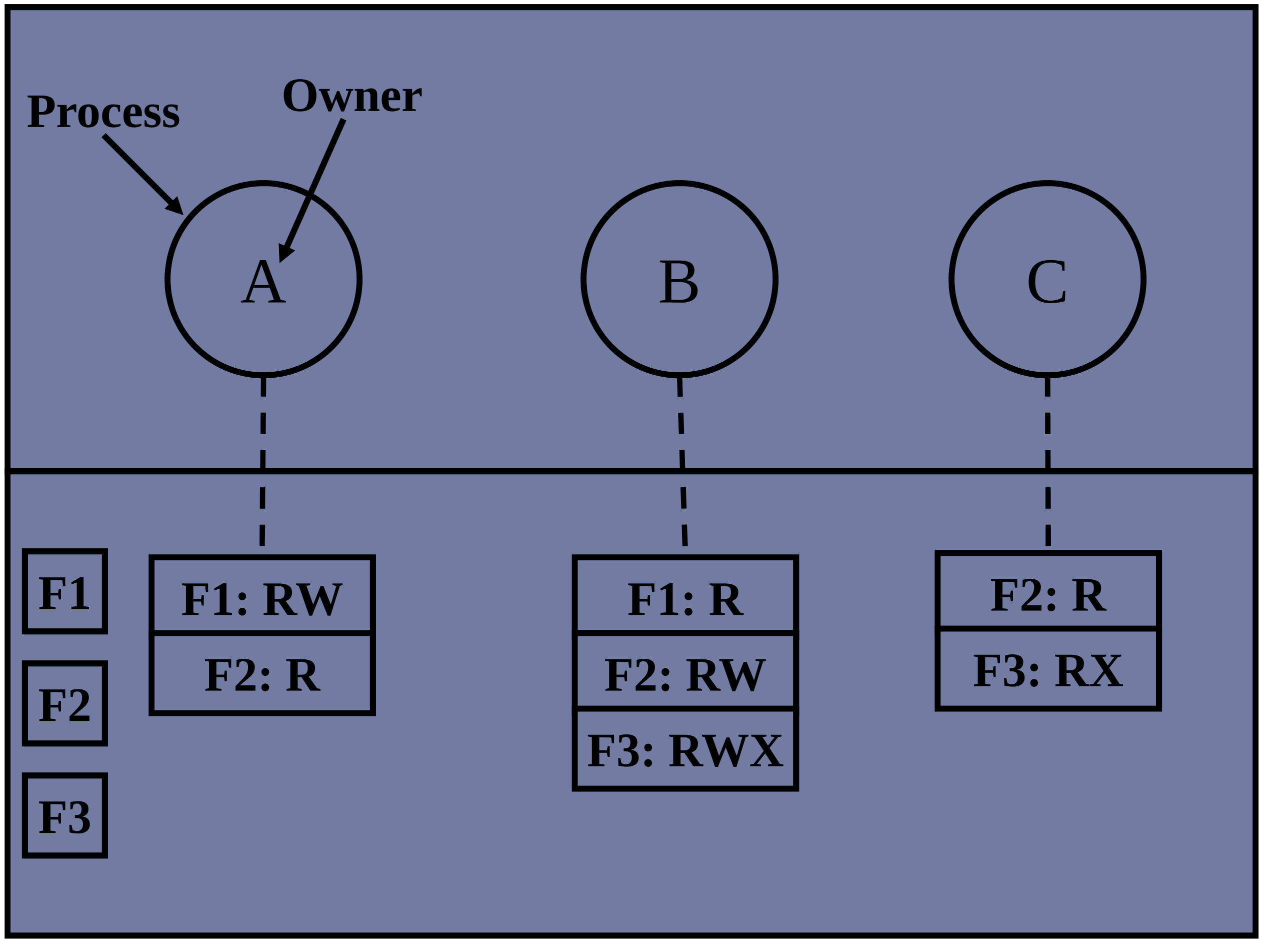
For simplicity, we assume that each domain corresponds to exactly one user.



File F1 has two entries in its ACL. The first entry says that any process owned by user A may read and write the file. The second entry says that any process owned by user B may read the file.
All other accesses by these users and all accesses by other users are forbidden.

Capabilities
权能

- Each process is associated with a list of objects that may be accessed, along with an indication of which operations are permitted on each (or in other words, its domain)
  - This list is called a capability list or C-list and individual items on it are call capabilities



Each process has a capability list

Storing ACLs for Files

- **The main problem with access lists is their length**
  - **Constructing such a list may be a tedious and unrewarded task**
  - **The directory entry now needs to be of variable size, resulting in more complicated space management**

Access Control Lists(2)

- **To condense the length of the access control lists, many systems recognize three classification of users in connection with each file:**
  - **Owner (擁有者): the user who created the file**
  - **Group or work group (工作組): A set of users who are sharing the file and need similar access**
  - **Universe (全域): all other users in the system constitute the universe**

Access Control Lists(3)

- **The most common recent approach is to combine *access control lists* with the more general (and easier to implement) *owner, group, and universe access control scheme*.**
  - **For example: Solaris 2.6 and beyond uses the three categories of access by default, but allow access control lists to be added to specific files and directories when more fine-grained access control is desired**

Access Control (3)
存取控制

- **The owner has all rights previously listed**
- **May grant rights to others using the following classes of users**
  - Specific user
  - User groups
  - All users (This file is then public file)

Simultaneous Access

- **A brute-force approach is to allow user to lock the entire file when it is to be updated**
- **A finer grain of control is to lock individual records during update**
- **Essentially, it is the readers/writers problem**
  - **Mutual exclusion and deadlock are issues for shared access**

Record Blocking
记录组块

Records and Blocks

- It is unlikely that the physical block size will exactly match the length of the desired logical record
  - Records are the logical units of access of a file; Logical records may even vary in length
  - Blocks are the unit of I/O with secondary storage; All blocks are of the same size
- For I/O to be performed, records must be organized as blocks

Issues to Consider

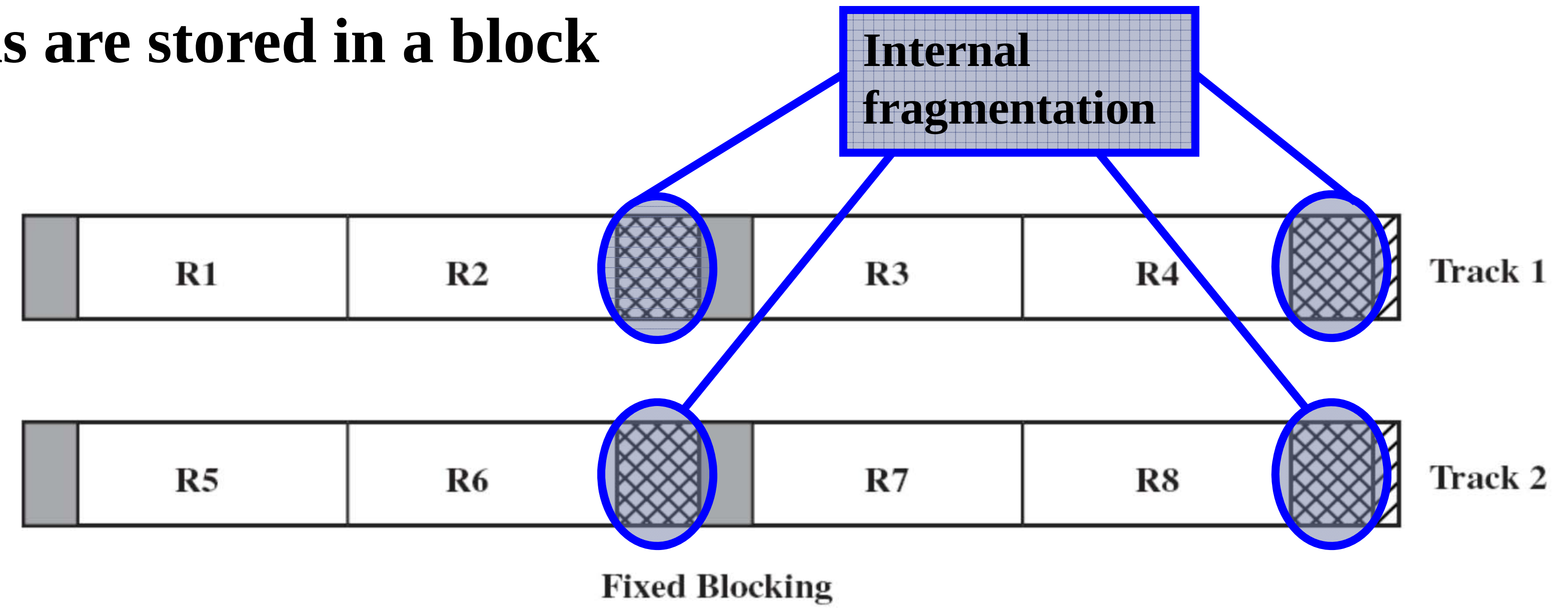
- What should the relative size of a block be compared to the average record size?
  - The larger the block, the more records that are passed in one I/O operation.
    - If a file is being processed sequentially, this is an advantage, because the number of I/O operations is reduced by using larger blocks
    - On the other hand, if records are being access randomly and no particular locality of reference is observed, then larger blocks result in the unnecessary transfer of unused records

Methods of Blocking
组块方法

- **Given the size of block, there are three methods of blocking**
  - Fixed blocking (固定组块)
  - Variable-length spanned blocking (可变长度跨越式组块)
  - Variable-length unspanned blocking (可变长度非跨越式组块)

Fixed Blocking

Fixed-length records are used, and an integral number of records are stored in a block

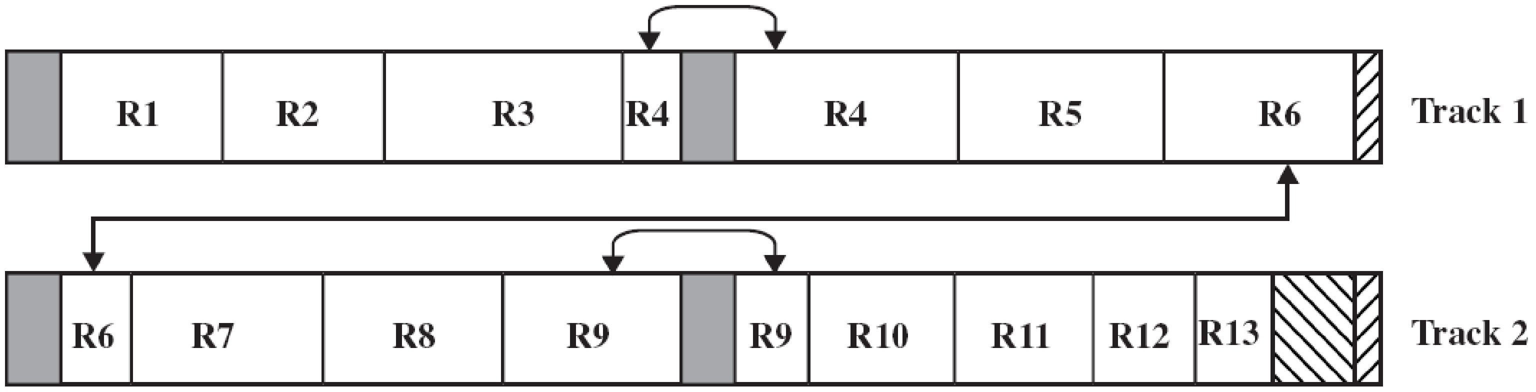


☐	Data	▧	Waste due to record fit to block size
■	Gaps due to hardware design	▩	Waste due to block size constraint from fixed record size
▨	Waste due to block fit to track size		

Variable Blocking: Spanned

Variable-length records are used, and are packed into blocks with no unused space.

Some records must span two blocks, with the continuation indicated by a pointer to the successor block

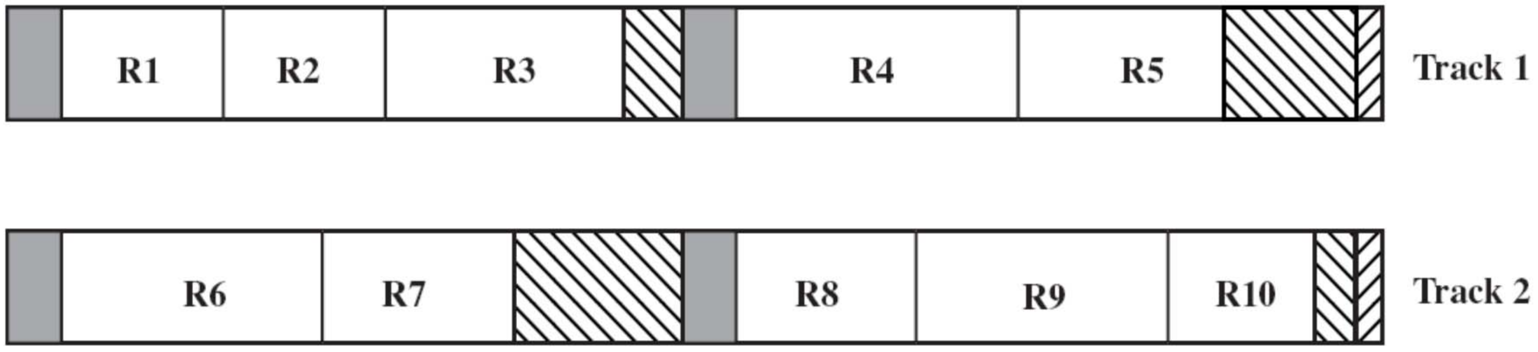


Variable Blocking: Spanned

Variable Blocking Unspanned

Variable-length records are used, but spanning is not employed.

There is wasted space in most blocks because the remainder of a block is not enough to hold the next record.



Variable Blocking: Unspanned

Comments

- Fixed blocking is the common mode for sequential files with fixed-length records
- Variable-length spanned blocking
  - Advantages: efficient of storage and does not limit the size of records.
  - Disadvantages: difficult to implement.
    - Records that span two blocks requires two I/O operations
    - Files are difficult to update
- Variable-length unspanned blocking
  - Disadvantages: (1) wasted space, (2) record size is limited to the size of a block

Secondary Storage Management
二级存储管理

Secondary Storage Management

- ***On secondary storage, a file consists of a collection of blocks (块)***
- ***To allocate blocks to files, the operating system must fulfill two tasks:***
  - *Space on secondary storage must be allocated to files* (**File Allocation**)
  - *It is necessary to keep track of the space available for allocation* (**Free Space Management**)

File Allocation Issues

- ***When a new file is created, is the maximum space required for the file allocated at once?***
  - **Preallocation versus Dynamic Allocation**
- ***What size of portion should be used for file allocation***
  - **Portion size (区段大小)**

译成“分区”不合适，与**partition**混淆

- ***What sort of data structure or table is used to keep track of the portions assigned to a file?***
  - **File allocation methods**

File Allocation
Preallocation versus Dynamic Allocation

- **Preallocation requires that the maximum size for a file be declared at the time of the file creation request**
  - Difficult to reliably estimate the maximum potential size of the file
  - Tend to overestimated file size so as not to run out of space
- **Dynamic allocation dynamically allocates space to a file in portions as needed.**

File Allocation
Portion Size - I

- **Two extremes**
  - One extreme: a portion large enough to hold the entire file is allocated
  - The other extreme: space on the disk is allocated one block at a time
- **Tradeoff between efficiency of a single file versus system efficiency**

File Allocation Portion Size - II

- **What to consider in the tradeoff?**
  - Contiguity of space increases performance
  - Having a large number of small portions increase the size of tables needed to manage the allocation information
  - Having fixed-size portion simplifies the allocation of space
  - Having variable-size or small fixed-size portions minimizes waste of unused storage due to over allocation

File Allocation Portion Size - III

- Two major alternatives
  - Variable, large contiguous portions:
    - better performance
    - The variable size avoids waste
    - The file allocation tables are small
    - However, space is hard to reuse
  - Blocks
    - Small fixed portions provides greater flexibility
    - They may require large tables or complex structures for their allocation.
    - Contiguity has been abandoned; blocks are allocated as needed

File Allocation
Allocation Methods

- Three methods are in common use
  - Contiguous allocation (连续分配)
  - Chained allocation (链式分配)
  - Indexed allocation (索引分配)

File Allocation
Contiguous Allocation - I

- ***With contiguous allocation, a single contiguous set of blocks is allocated to a file at the time of file creation***
  - If the file is *n* blocks long and starts at location b, then it occupies blocks *b*, *b*+1, *b*+2, …, *b*+*n*–1.
  - Only a *single entry* in the file allocation table
    - ***Starting block and length of the file***
- **This is a preallocation strategy, using variable-size portions**

File Allocation
Contiguous Allocation - II

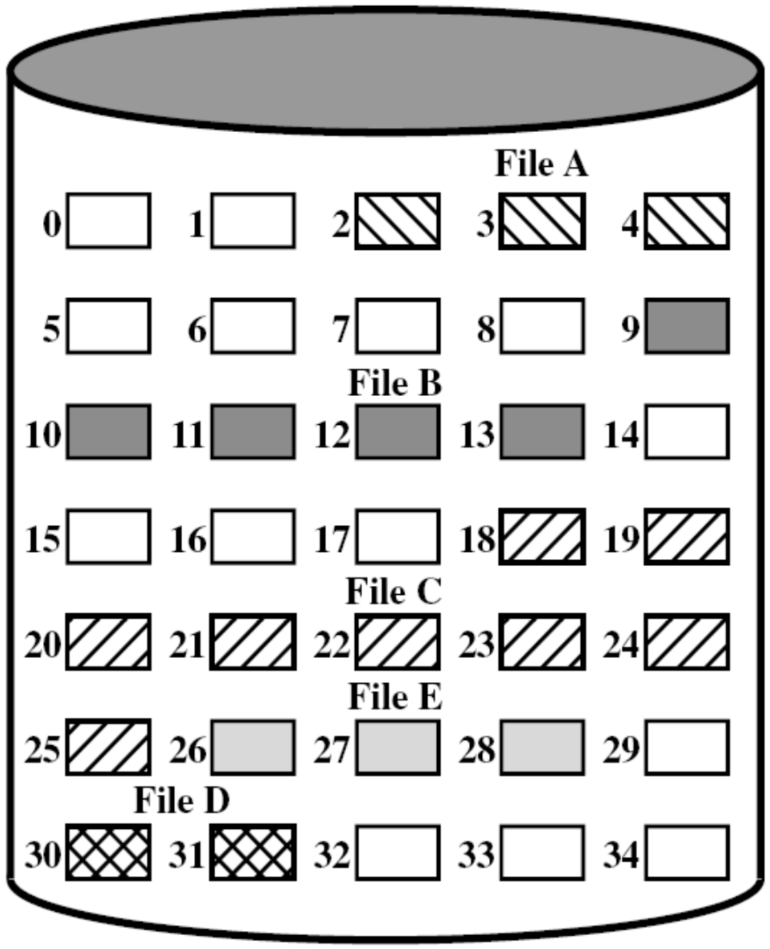
- **The main advantage is:** ***Accessing a file that has been allocated contiguously is easy for both sequential access and direct access.***
  - Multiple blocks can be brought in at a time to improve performance for sequential processing (only one seek 仅需一次磁盘寻道)
  - It is easy to retrieve a single block (direct access)
    - **For direct access to block *i* of a file that starts at block *b*, we can immediately access block *b+i*.**

File Allocation
Contiguous Allocation - III

- Contiguous allocation presents some problems
  - External fragmentation problem:
    - Necessary to perform a compaction (紧缩/紧致) algorithm to free up additional space on the disk
  - Size declaration problem
    - With preallocation, it is necessary to declare the size of the file at the time of creation
- Even if the total amount of space needed for a file in known in advance, contiguous preallocation may be inefficient
  - A file that grows slowly over a long period must be allocated enough space for its final size (large amount of internal fragementation)

File Allocation
Contiguous Allocation - IV

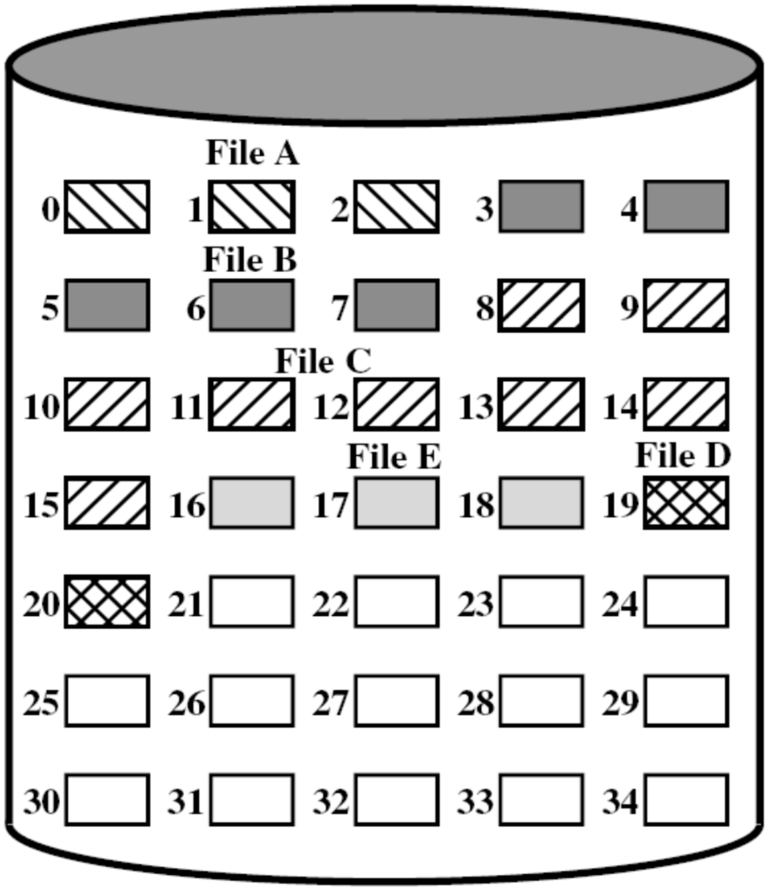
- **Contiguous allocation is feasible and in fact widely used in one situation**
  - **On CD-ROM**



File Allocation Table

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

Figure 12.7 Contiguous File Allocation



File Allocation Table

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

Figure 12.8 Contiguous File Allocation (After Compaction)

File Allocation Methods
Chained Allocation (链式分配) - I

- **With chained allocation, allocation is on an individual block basis**
  - **Each block contains a pointer to the next block in the chain**
    - **If each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes**
  - **Only single entry in the file allocation table**
    - **Showing the starting block and the length of file**
- **No external fragmentation, and no size declaration problem**
  - **This method is best suited for sequential files that are to be processed sequentially**

File Allocation Methods
Chained Allocation - II

- **Disadvantage 1: It can be used effectively only for sequential-access files.**
  - To find the *i*th block of a file, we must start at the beginning of the file, and follow the pointers until we get to the *i*th block.
  - Each access to a pointer requires a disk read, and sometimes a disk seek.
  - Thus, it is inefficient to support a direct access capability.

File Allocation Methods
Chained Allocation - III

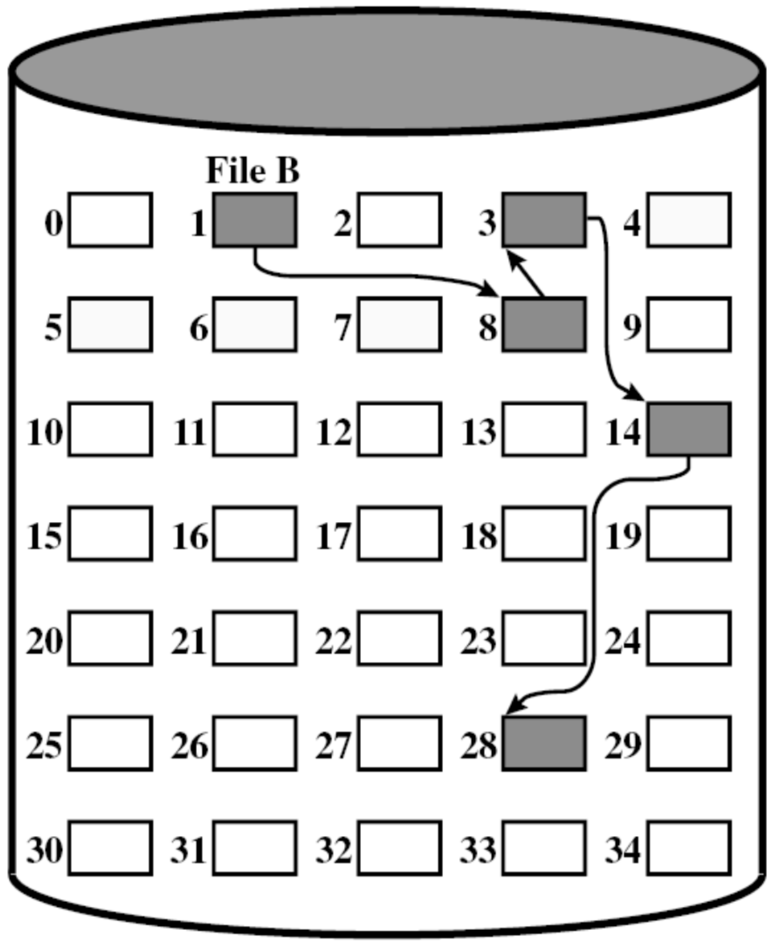
- **Disadvantage 2: Consolidation is required to improve the performance of sequential access**
  - The blocks of a file is scattered all over the disk
    - If several blocks of a file is to be brought in, it is required to access different parts of the disk (multiple disk seeks)
    - To overcome this problem, the system periodically consolidates(合并) files

File Allocation Methods
Chained Allocation - IV

- **Disadvantage 3: The pointers require some disk space.**
  - Each file requires slightly more space than it would.
  - The usual solution to this problem: to collect blocks into multiples, called clusters (簇), and to allocate clusters rather than blocks.
    - The logical-to-physical block mapping remain simple
    - The disk throughput is improved (fewer disk head seeks)
    - The space needed for block allocation and free-list management is reduced
    - However, internal fragmentation is increased

File Allocation Methods
Chained Allocation - V

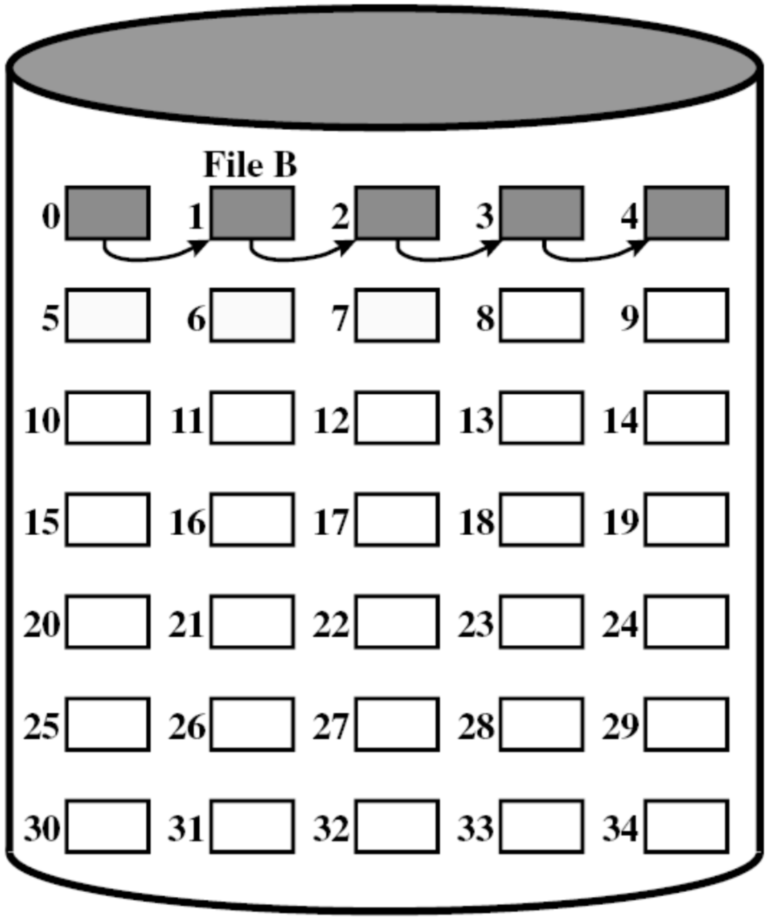
- **Disadvantage 4: Reliability**
  - The blocks of a file are linked together by pointers scattered all over the disk. What happen if a pointer were lost or damaged?
  - Partial solutions are to use doubly linked list or to store the filename and relative block number in each block
    - However, these schemes require even more overhead for each file



File Allocation Table

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

Figure 12.9 Chained Allocation



File Allocation Table

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

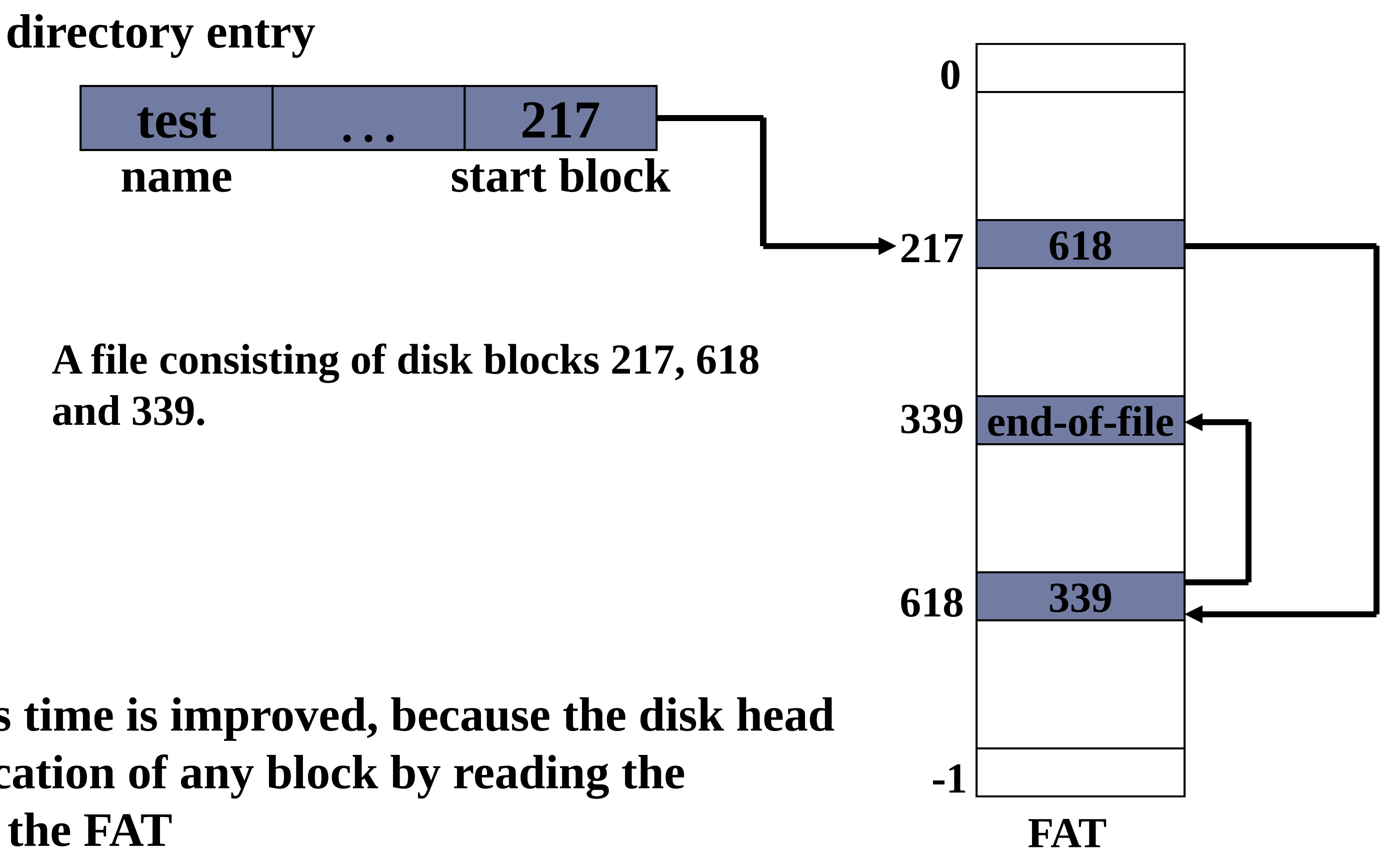
Figure 12.10 Chained Allocation (After Consolidation)

FAT: File Allocation Table (1)

An Important Variation on The Chained Allocation

- **FAT is a simple but efficient method of disk-space allocation used by the *MS-DOS and OS/2***
- **A section of disk *at the beginning of each partition* is set aside to contain the table**
  - **The table has one entry for each disk block, and is indexed by block number.**
  - **The directory entry contains the block number of the first block of the file.**
  - **The table entry indexed by that block number then contains the block number of the next block in the file**
  - **This chain continues until the last block, which has a special end-of-file value as the table entry.**
  - **The unused blocks are indicated by a 0 table value**

FAT: File Allocation Table (2)



A file consisting of disk blocks 217, 618 and 339.

Random access time is improved, because the disk head can find the location of any block by reading the information in the FAT

FAT: File Allocation Table (3)

- **Allocating a new block to a file is simple**
  - Finding the first 0-valued table entry
  - Replacing the previous end-of-file value with the address of the new block
  - The 0 is then replaced with the end-of-file value

FAT: File Allocation Table (4)

- **If the FAT is not cached, the FAT allocation scheme can result in a significant number of disk head seeks**
  - The disk head must move to the start of the partition to read the FAT and find the location of the block in question
  - Then move to the location of the block itself.
- **However, the benefit is that random access time is improved**
  - Because the disk head can find the location of any block by reading the information in the FAT

File Allocation Methods
Indexed Allocation 索引分配 - I

- **Chained allocation solves the external fragmentation and size-declaration problems of contiguous allcoation**
  - **However, it can not support efficient direct access.**
- **Indexed allocation solves this problem by bringing all the pointers together into one location: the index block**

File Allocation Methods
Indexed Allocation - II

- **With indexed allocation, file allocation table contains a *separate one-level index* for each file**
  - The index has *one entry for each portion* allocated to the file
  - *The file index is kept in a separate block*, and the entry in the file allocation table points to that block
- **Allocation may be on the basis of either *fixed-size blocks* or *variable-size portions***
- **Index allocation supports both *sequential* and *direct* access to the file**
  - It is *the most popular* form of file allocation

File Allocation Methods
Indexed Allocation - III

- **Indexed allocation does suffer from wasted space**
  - **The pointer overhead of the index block is generally greater than the pointer overhead of chained allocation**
    - **Consider a common case where we have a file of only one or two blocks: with indexed allocation, an *entire index block* must be allocated.**

File Allocation Methods
Indexed Allocation - IV

- **A new question is raised: How large should be the index block?**
  - Chained Scheme: several index blocks are chained together to allow for large files
  - Multilevel index: Using a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.
  - Combined Scheme: shown later in “UNIX file management”

Allocation by blocks eliminates external fragmentation

Whereas allocation by variable-size portions improves locality

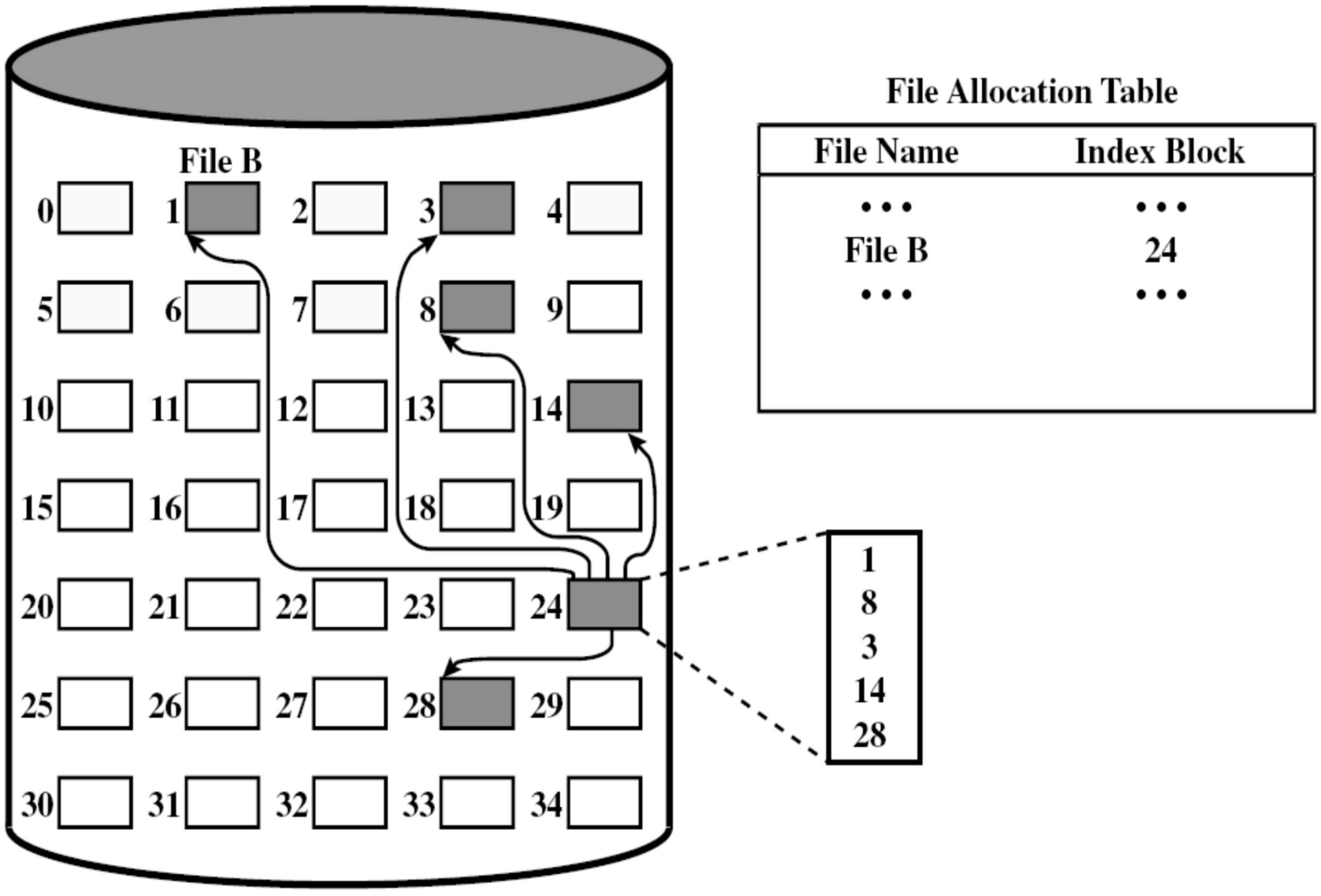
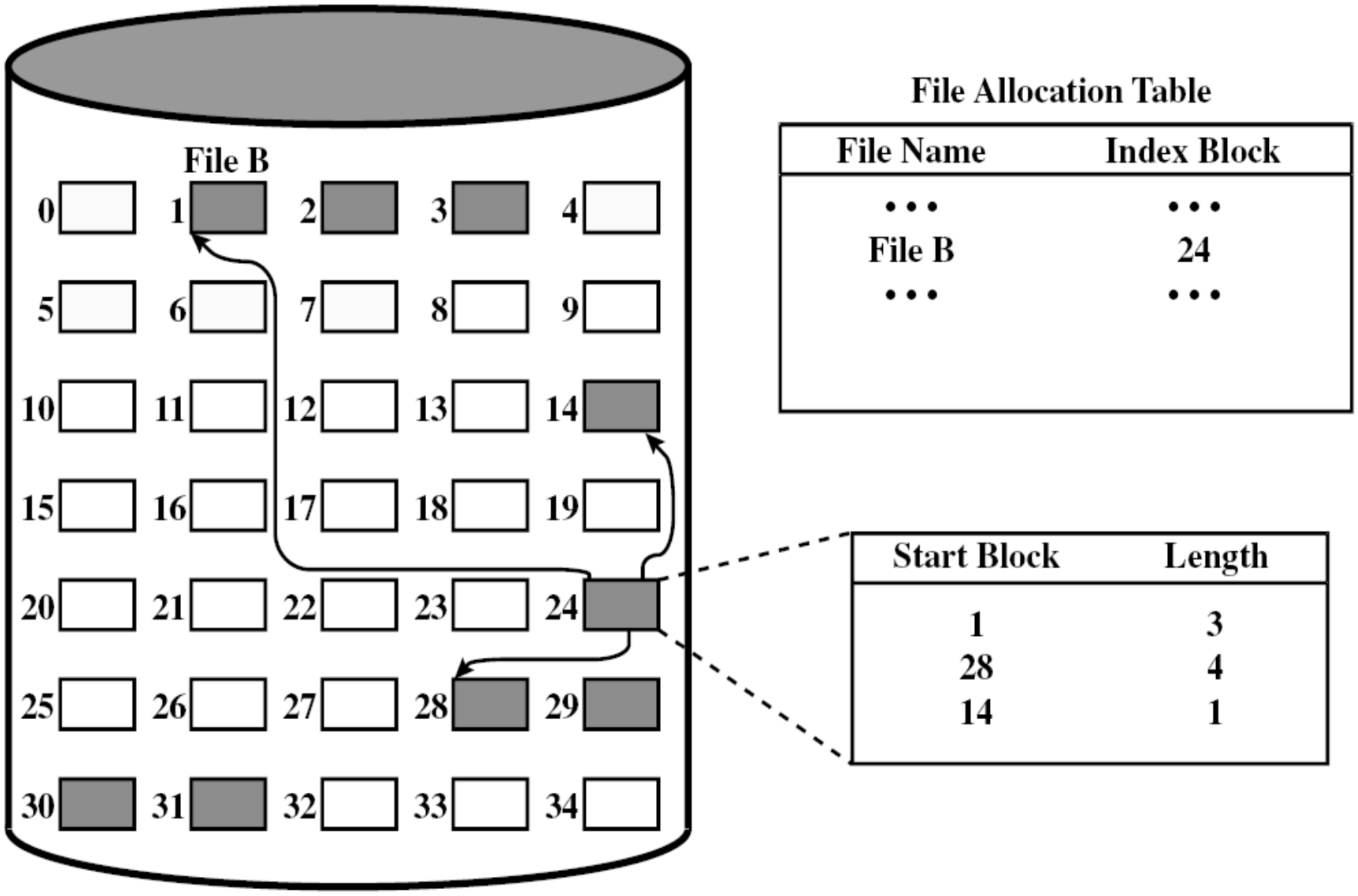


Figure 12.11 Indexed Allocation with Block Portions

File consolidation may be done from time to time to improve locality.
File consolidation reduces the size of the index in the case of variable-size portions, but not in the case of block allocation.



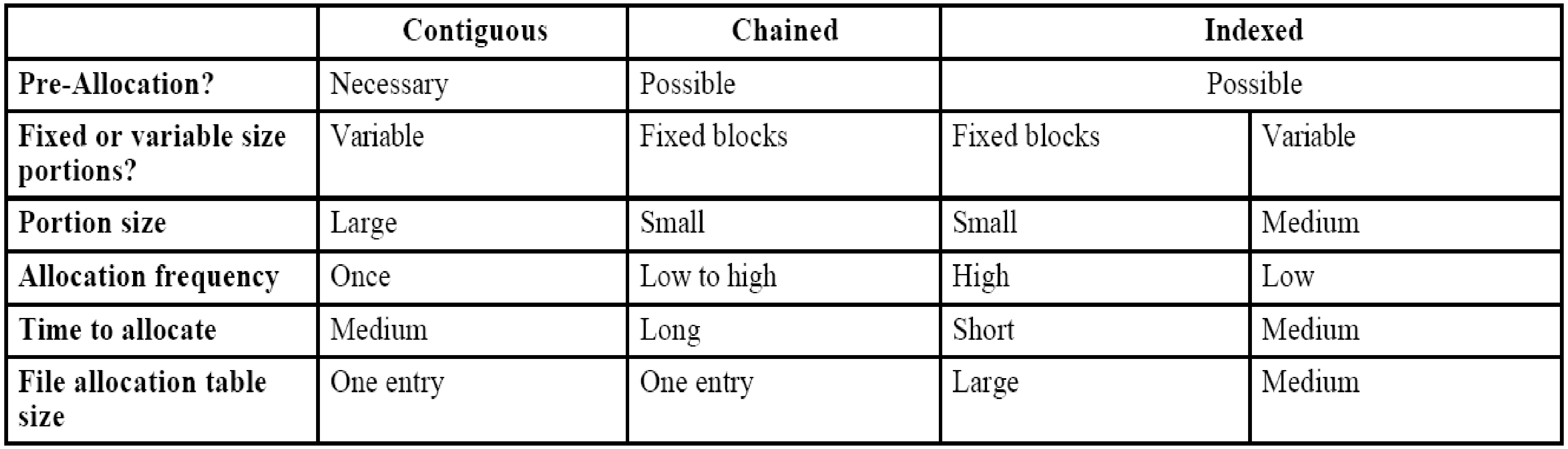
File Allocation Table

File Name	Index Block
• • •	• • •
File B	24
• • •	• • •

Start Block	Length
1	3
28	4
14	1

Figure 12.12 Indexed Allocation with Variable-Length Portions

Summary on File Allocation Methods



	Contiguous	Chained	Indexed	
Pre-Allocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

Free Space Management
空闲空间管理

- To perform file allocation, it is necessary to know what blocks on the disk are available
  - We need a disk allocation table in addition to a file allocation table
- Techniques for free space management
  - Bit tables (Bitmap, or Bit Vector)
  - Linked List
  - Free Block List
  - Grouping
  - Chained free portions

Free Space Management
Bit Tables (位表) - I

- **This method uses a vector containing one bit for each block on the disk**
  - Each entry of 0 corresponds to a free block
  - Each 1 corresponds to a block in use

Free Space Management Bit Tables (位表) - II

- **Advantages**
  - **It is relatively easy to find one or a contiguous group of free blocks**
    - **It works well with any of the file allocation methods.**
  - **It is very small**
    - **(disk size in bytes)/(8*file system block size)**
    - **For a 16Gbyte disk with 512-byte blocks, the bit table occupies about 4Mbytes**

Free Space Management Bit Tables (位表) - III

- **Where should we put the bit table?**
  - In main memory or on disk?
  - If on disk, we cannot afford to search that amount of disk space every time a block is needed
  - Even in main memory, an exhaustive search of the table can slow file system performance to an unacceptable degree
- **Solution: maintain auxiliary data structures that summarize the contents of subranges of the bit table**
  - For example: The table could be divided into a number of equal size subranges, the number of free blocks and the maximum-sized contiguous blocks

Free Space Management
Linked List

- **Each block is assigned a number sequentially**
  - **Depending on the disk size, either 24 or 32 bits is needed to store a single block number**

Free Space Management
Linked List

- **Link together all the free disk blocks**
  - **Keep a pointer to the first free block in a special location on the disk**
    - **Cache the pointer in memory**
  - **This first block contains a pointer to the next free block, and so on.**
- **This scheme is not efficient**

Free Space Management
Free Block List

- **To store the free block numbers in a reserved region on disk**
  - The space for the free block list is less than 1% of the total disk space
  - If a 32-bit block number is used, then the space penalty is 4 bytes for every 512-byte block

Free Space Management Grouping

- **We could store the addresses of *n* free blocks in the first free block**
  - The first *n*−1 of these blocks are actually free.
  - The last block contains the addresses of another *n* free blocks, and so on.

Free Space Management
Chained Free Portions 链式空闲区 - I

- **The free portions may be chained together by using *a pointer and length value* in each free portion**
  - We only need a pointer to the beginning of the chain and the length of the first portion

Free Space Management
Chained Free Portions - II

- **This method is suited to all of the file allocation methods**
  - If allocation is a block at a time, simply choose the free block at the head of the chain and adjust the first pointer or length value
  - If allocation is by variable-length portion, a first-fit algorithm may be used

Free Space Management
Chained Free Portions - III

- **Disadvantages:**
  - After some use, the disk will become quite fragmented and many portions will be a single block long
  - If all portions are one block long, every time you allocate a block, you need to read the block first to recover the pointer to the new first free block before writing data to that block

File System Consistency

- **Many file systems read blocks, modify them, and write them out later**
  - What if the system crashes before all modified block have been written out?
  - The file system can be left in an inconsistent state
  - This problem is especially critical if the blocks that are not written out includes file allocation table, disk allocation table, etc.
- **Two kinds of consistency checks: block consistency and file consistency**

Block Consistency (1)

- **To check for block consistency, two tables are constructed:**
  - **Each table contains a counter for each block, initially set to 0.**
  - **The counters in the first table keep track of how many times each block is present in a file**
  - **The counters in the second table record how often each block is present in the free list (or the bitmap of free blocks)**

Block Consistency (2)
Consistent

- **If the file system is consistent, each block will have a 1 either in the first table or in the second table.**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

Block Consistency (3)
Missing block

- **A block does not occur in either table**
  - It is reported as a missing block
  - Missing blocks do no real harm, but they do waste space
  - Solution: the file system checker just adds them to the free list

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

Block Consistency (4)
Duplicate block in free list

- **A block occurs twice or more times in the free list**
  - **Duplicate can occur only if the free list is really a list; with a bitmap it is impossible**
  - **Solution: rebuild the free list**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	Free blocks

Block Consistency (5) Duplicate data block

- **The worst thing that can happen is that the same data block is present in two or more files.**
  - If either of these files is removed, the block will be put on the free list, leading to a situation where the same block is both in use and free at the same time.
  - If both files are removed, the block will be put onto the free list twice
  - Solution: allocate a free block, copy the contents of the data block into it, and insert the copy into one of the files

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

File Consistency (1)

- It starts from the root directory and recursively descends the tree, inspecting each directory in the file system. For every i-node in every directory, it increments a counter for that file's usage count.

File Consistency (2)

- **If the link count is higher than the number of directory entries**
  - Even if all the files are removed from the directories, the count will still be nonzero and the i-node will not be removed.
  - This error is not serious, but wastes space on the disk with files that are not in any directory.
- **If the link count is lower than the number of directory entries**
  - When an i-node count goes to zero, the file system marks it as unused and releases all its blocks
  - However, one of the directory entries is still pointing to this unused i-node, whose blocks may soon be assigned to other files

UNIX File Management
UNIX文件管理

The UNIX File System

- ***The UNIX kernel treats all files as streams of bytes.***
  - *Any internal logical structure is application specific*
  - *UNIX is concerned with the physical structure of files*

File Types

- **Four types of files are distinguished:**
  - Ordinary: *Files that contain information entered in them by a user, an user program, or a system utility program*
  - Directory: *Contains a list of file names plus pointers to associated i-nodes (index nodes).*
    - *Directories are actually ordinary files with special write protection privileges so that only the file system can write into them, while read access is available to user programs*
  - Special: *Used to access peripheral devices. Each I/O device is associated with a special file.*
  - Named: *Name pipes, discussed in Section 6.7*

Inodes

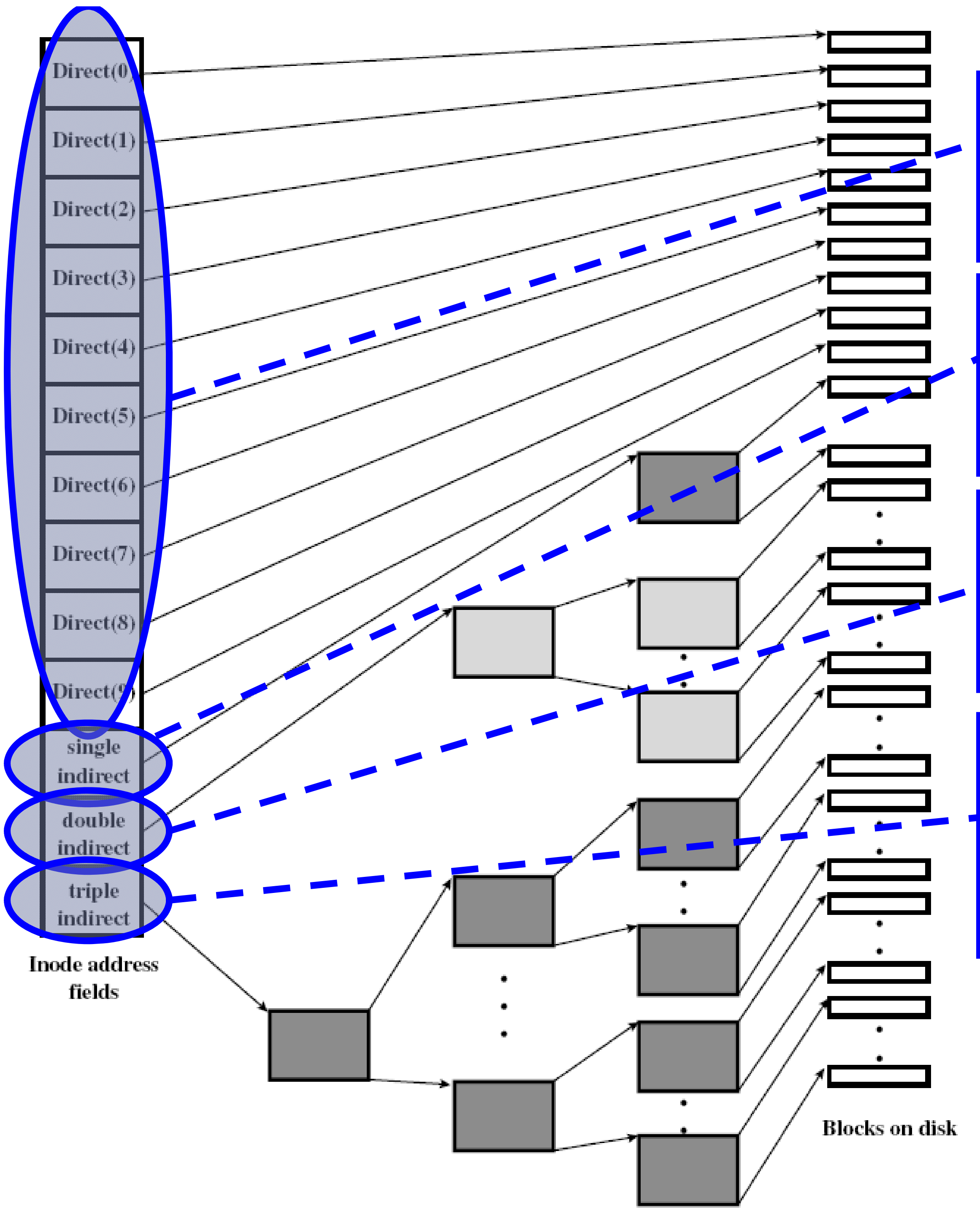
- **All types of UNIX files are administrated by the OS by means of inodes. An inode contains the key information needed by the OS for a particular file:**
  - **File Mode: 16-bit flag storing access and executing permissions**
  - **Link Count: number of directory references to this inode**
  - **Owner ID: individual owner of file**
  - **Group ID: group owner associated with this file**
  - **File Size: number of bytes in file**
  - **File Addresses: 39 bytes of address information**
  - **Last Accessed: time of last file access**
  - **Last Modified: time of last file modification**
  - **Inode Modified: time of last inode modification**

Several file names may be associated with a single inode

But an active inode is associated with exactly one file.

File Allocation

- **File allocation is done on a block basis.**
  - Allocation is dynamic, rather than preallocation
  - An indexed method is used to keep track of each file, with part of the index stored in the inode for the file
  - The inode includes thirteen 3-byte addresses



The first 10 addresses point to the first 10 blocks of the file

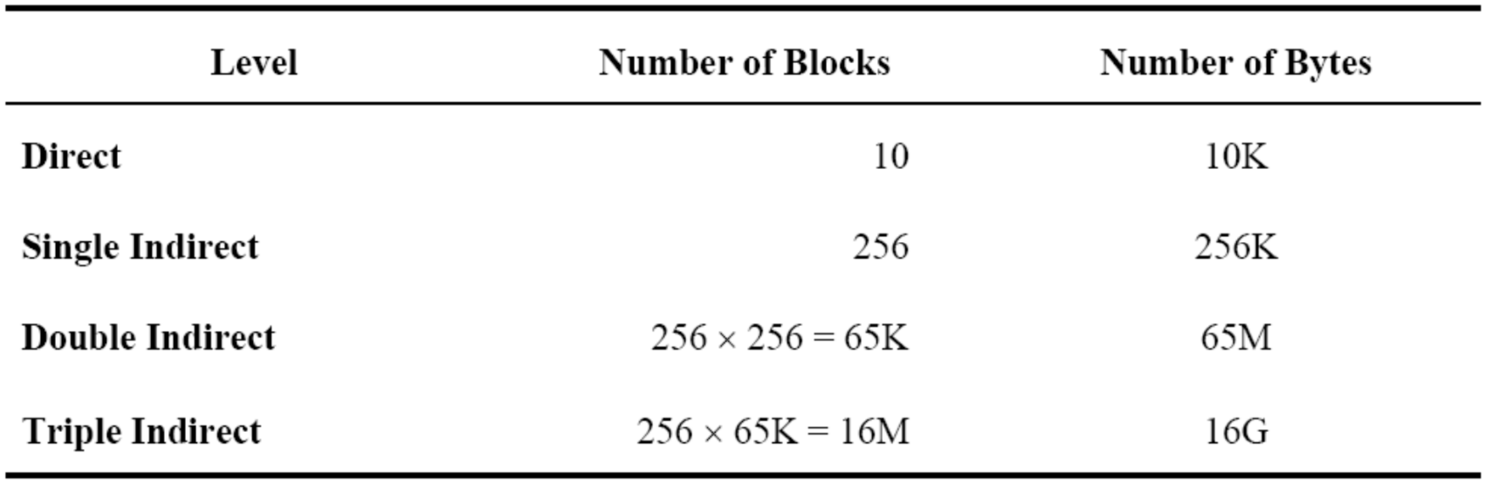
The 11th address points to a block that contain the pointers to succeeding blocks in the file (single indirect block)

The 12th address points to a double indirect block. This block contains a list of addresses of additional single indirect blocks

The 13th address points to a triple indirect block that is a third level of indexing. This block points to additional double indirect blocks

Capacity of a UNIX file

The length of a block is 1 Kbyte in UNIX system, and each block can hold a total of 256 block addresses.



Level	Number of Blocks	Number of Bytes
Direct	10	10K
Single Indirect	256	256K
Double Indirect	256 × 256 = 65K	65M
Triple Indirect	256 × 65K = 16M	16G

File Open in UNIX (1)

- How to open the file of path name */usr/ast/mbox* in UNIX?
  - Firstly, the file system locates the root directory whose i-node is located at a fixed place on the disk
    - From the i-node, it locates the root directory which can be anywhere on disk
  - Then, the file system reads the root directory and looks up the first component of the path, *usr*, to find the i-node number of the file */usr*.
    - From this i-node, the system locates the directory for */usr*, and looks up the next component, *ast*, in it.
  - When it has find the entry for *ast*, it has the i-node for the directory */usr/ast*.
    - From this i-node it can find the directory itself and look up mbox. The i-node for this file is then read into memory and kept there until the file is closed

File Open in UNIX (2)

Root directory

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

Looking up *usr* yields i-node 6

I-node 6 is for /usr

mode size times
132

I-node 6 says that /usr is in block 132

Block 132 is /usr directory

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

/usr/ast is i-node 26

I-node 26 is for /usr/ast

mode size times
406

I-node 26 says that /usr/ast is in block 406

Block 406 is /usr/ast directory

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

/usr/ast/mbox is i-node 60