Department of Electronics & Communication Engg.

Course : Operating Systems -15EC553.   **Sem.: 5th (2018-19 ODD)**

Course Coordinator:

Prof. Nyamatulla M Patel

Operating Systems-15EC553

Module-04
File Systems & IOCS

Introduction

- Overview of File Processing
- Files and File Operations
- Fundamental File Organizations and Access Methods
- Directories
- Mounting of File Systems
- File Protection
- Allocation of Disk Space
- Interface Between File System and IOCS

Introduction (continued)

- File Processing
- File Sharing Semantics
- File System Reliability
- Journaling File Systems
- Virtual File System
- Case Studies of File Systems
- Performance of File Systems

Overview of File Processing

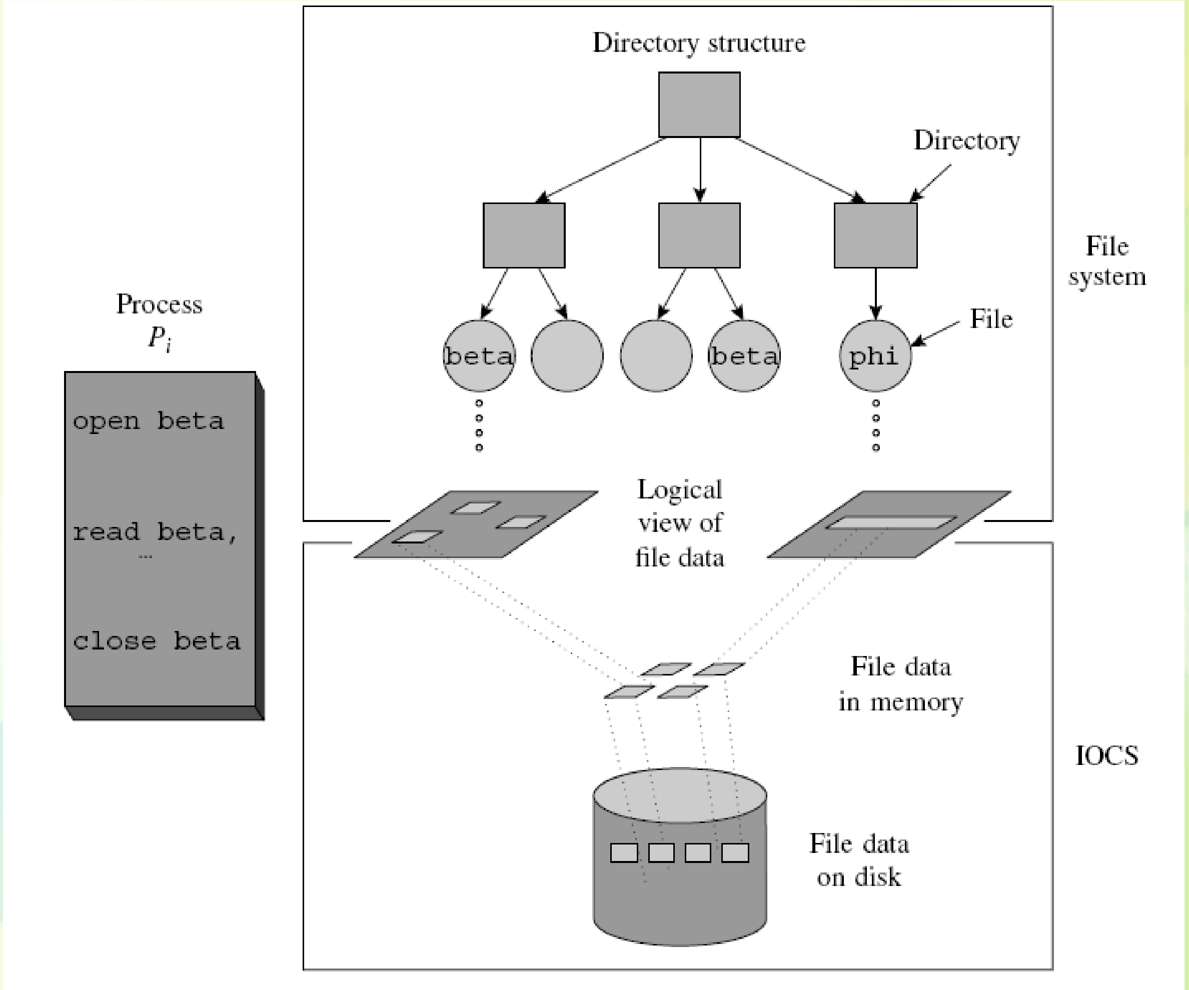


Figure 13.1 File system and the IOCS.

File System and the IOCS

- File system views a file as a collection of data that is *owned* by a user, *shared* by a set of authorized users, and *reliably stored* over an extended period
- IOCS views it as a repository of data that is *accessed speedily* and stored on I/O device that is *used efficiently*

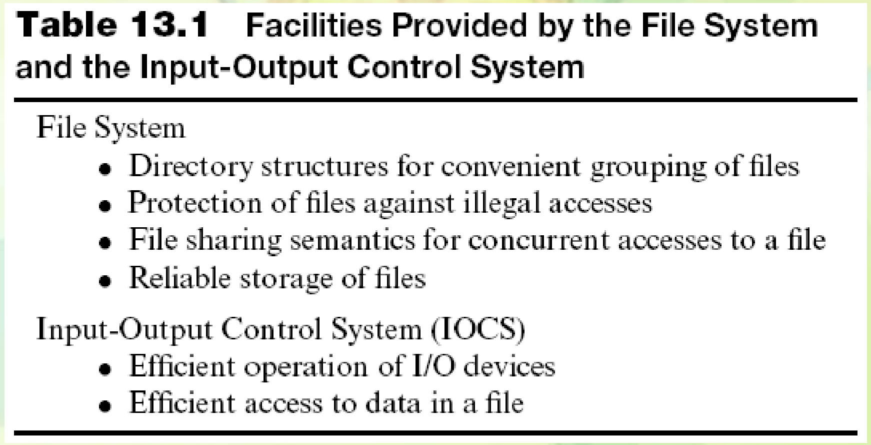


Table 13.1 Facilities Provided by the File System and the Input-Output Control System

File System
- Directory structures for convenient grouping of files
- Protection of files against illegal accesses
- File sharing semantics for concurrent accesses to a file
- Reliable storage of files

Input-Output Control System (IOCS)
- Efficient operation of I/O devices
- Efficient access to data in a file

- Two kinds of data: *file data* and *control data*

File Processing in a Program

- At programming language level:
  - File: object with *attributes* describing organization of its data and the method of accessing the data

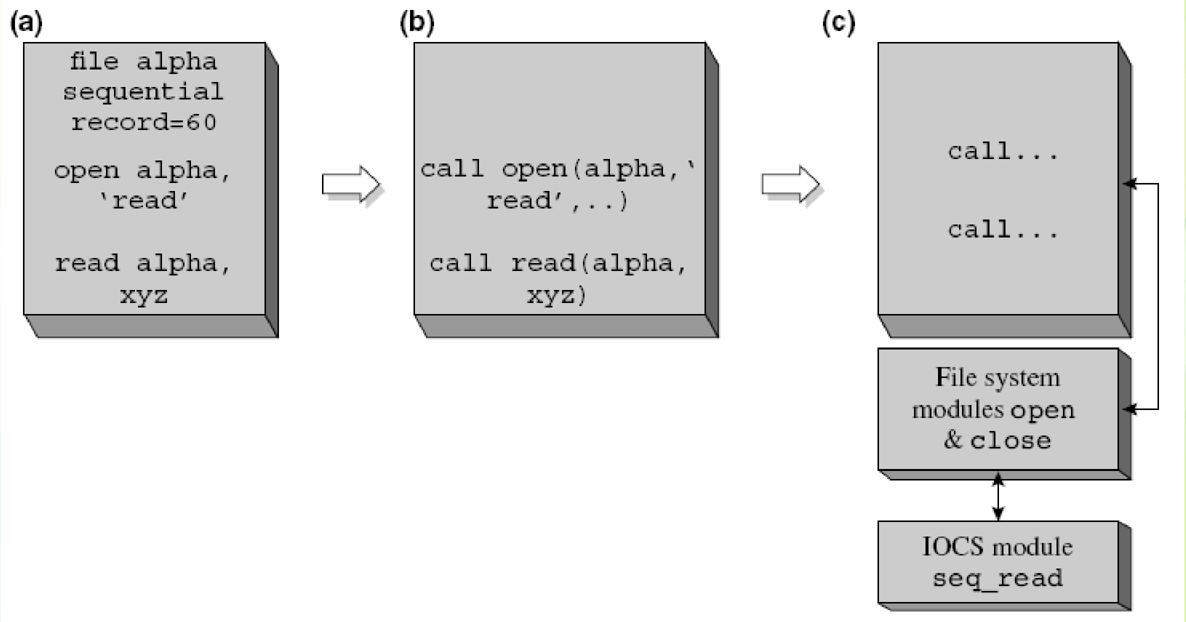


Figure 13.2 Implementing a file processing activity: (a) program containing file declaration statements; (b) compiled program showing calls on file system modules; (c) process invoking file system and IOCS modules during operation.

Files and File Operations

- File types can be grouped into two classes:
  - *Structured files:* Collection of records
    - *Record:* collection of fields
    - *Field:* contains a single data item
    - Each record is assumed to contain a unique *key* field
  - *Byte stream files:* "Flat"

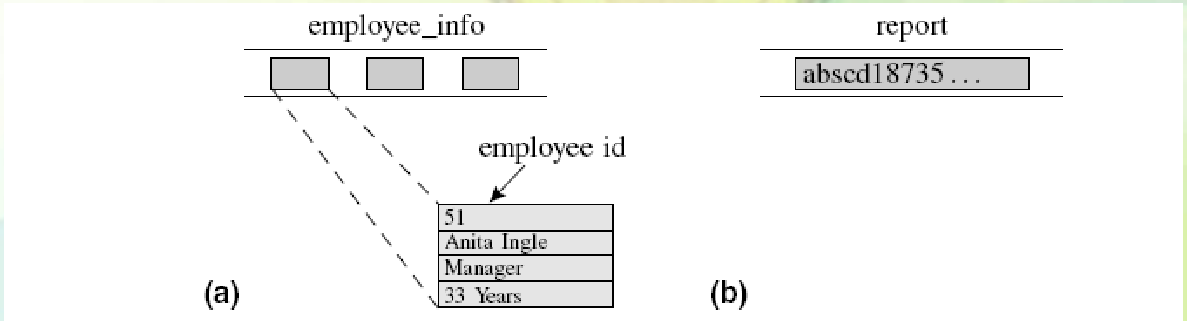


Figure 13.3 Logical views of (a) a structured file `employee_info`; (b) a byte stream file `report`.

- A file has attributes, stored in its directory entry

Fundamental File Organizations and Access Methods

- Fundamental record access patterns:
  - *Sequential access*
  - *Random access*
- *File organization* is a combination of two features:
  - Method of arranging records in a file
  - Procedure for accessing them
- Accesses to files governed by a specific file organization are implemented by IOCS module called *access method*

Sequential File Organization

- Records are stored in an ascending or descending sequence according to the key field
- Record access pattern of an application is expected to follow suit
- Two kinds of operations:
  - Read the next (or previous) record
  - Skip the next (or previous) record
- Uses:
  - When data can be conveniently presorted into an ascending or descending order
  - For byte stream files

Direct File Organization

- Provides convenience/efficiency of file processing when records are accessed in a random order
- Files are called *direct-access files*
- Read/write command indicates value in key field
  - Key value is used to generate address of record in storage medium
- Disadvantages:
  - Record address calculation consumes CPU time
  - Some recording capacity of disk is wasted
  - Dummy records exist for key values that are not in use

Example: Sequential and Direct Files

- Employees with the employee numbers 3, 5–9 and 11 have left the organization
  - Direct file has dummy records for them

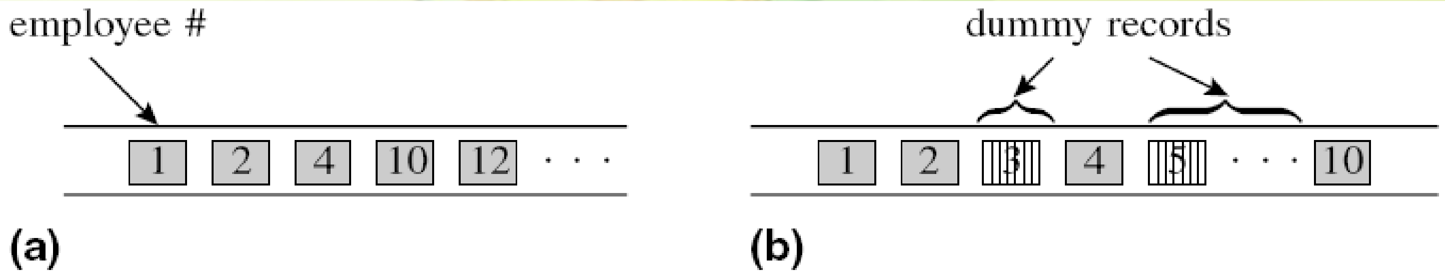
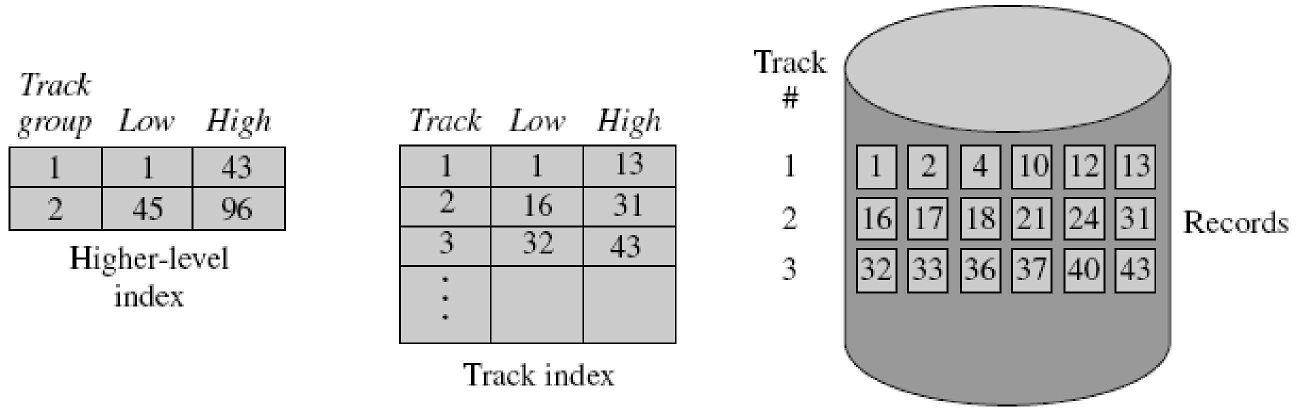


Figure 13.4 Records in (a) sequential file; (b) direct-access file.

Index Sequential File Organization

- An *index* helps determine location of a record from its key value
  - Pure indexed organization: (key value, disk address)
  - *Index sequential* organization uses index to identify section of disk surface that may contain the record
    - Records in the section are then searched sequentially



Track group	*Low*	*High*
1	1	43
2	45	96

Higher-level index

Track	*Low*	*High*
1	1	13
2	16	31
3	32	43
⋮		

Track index

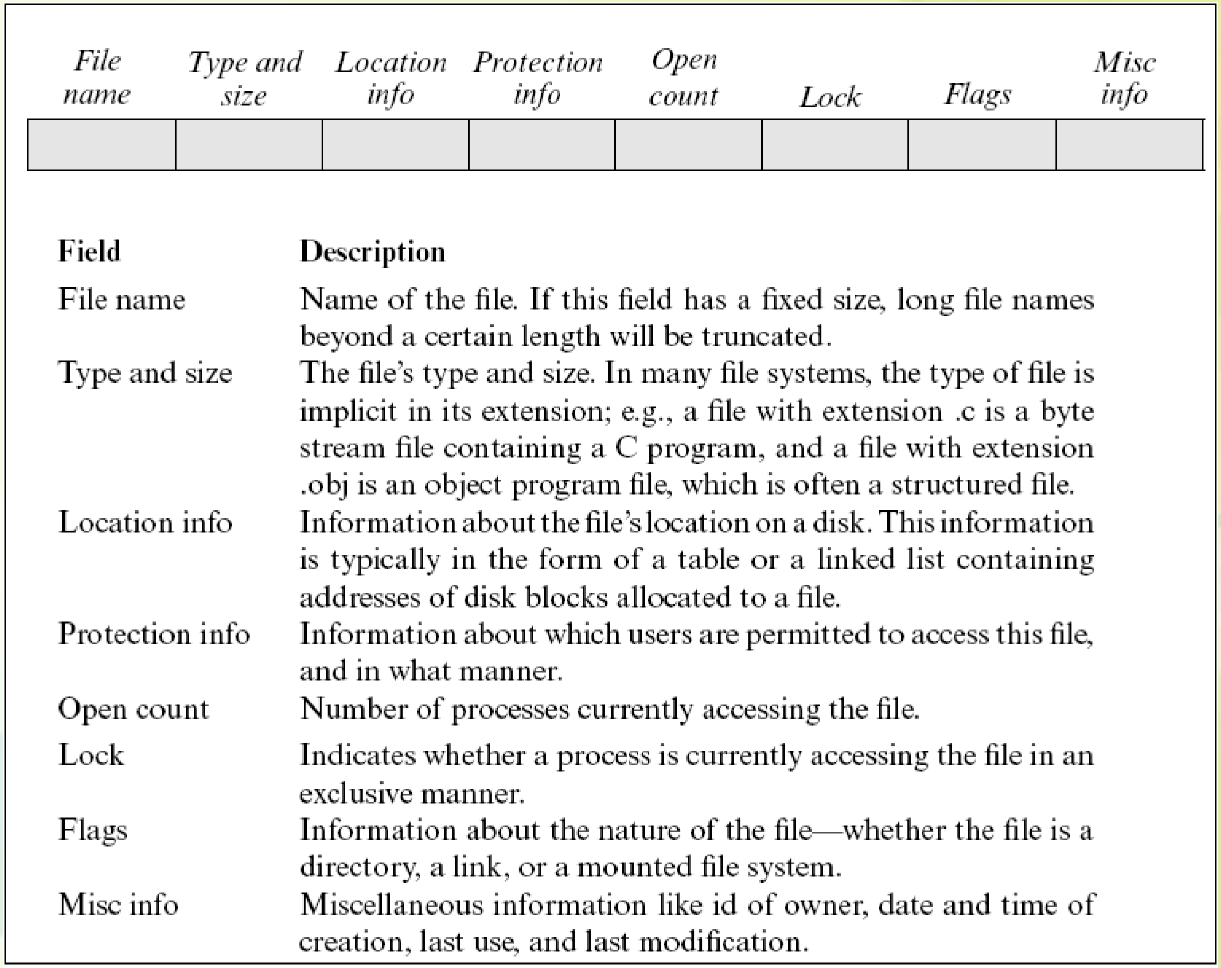
Track #						
1	1	2	4	10	12	13
2	16	17	18	21	24	31
3	32	33	36	37	40	43

Figure 13.5 Track index and higher-level index in an index sequential file.

Access Methods

- *Access method:* IOCS module that implements accesses to a class of files using a specific file organization
  - Procedure determined by file organization
  - Advanced I/O techniques are used for efficiency:
    - *Buffering* of records
      - Records of an input file are read ahead of the time when they are needed by a process
    - *Blocking* of records
      - A large block of data, whose size exceeds the size of a record in the file, is always read from, or written onto, the I/O medium

Directories



File name	*Type and size*	*Location info*	*Protection info*	*Open count*	*Lock*	*Flags*	*Misc info*

Field	**Description**
File name	Name of the file. If this field has a fixed size, long file names beyond a certain length will be truncated.
Type and size	The file's type and size. In many file systems, the type of file is implicit in its extension; e.g., a file with extension .c is a byte stream file containing a C program, and a file with extension .obj is an object program file, which is often a structured file.
Location info	Information about the file's location on a disk. This information is typically in the form of a table or a linked list containing addresses of disk blocks allocated to a file.
Protection info	Information about which users are permitted to access this file, and in what manner.
Open count	Number of processes currently accessing the file.
Lock	Indicates whether a process is currently accessing the file in an exclusive manner.
Flags	Information about the nature of the file—whether the file is a directory, a link, or a mounted file system.
Misc info	Miscellaneous information like id of owner, date and time of creation, last use, and last modification.

Figure 13.6 Fields in a typical directory entry.

Directories (continued)

- File system needs to grant users:
  - *File naming freedom*
  - *File sharing*
- File system creates several directories
  - Uses a *directory structure* to organize them
    - Provides file naming freedom and file sharing

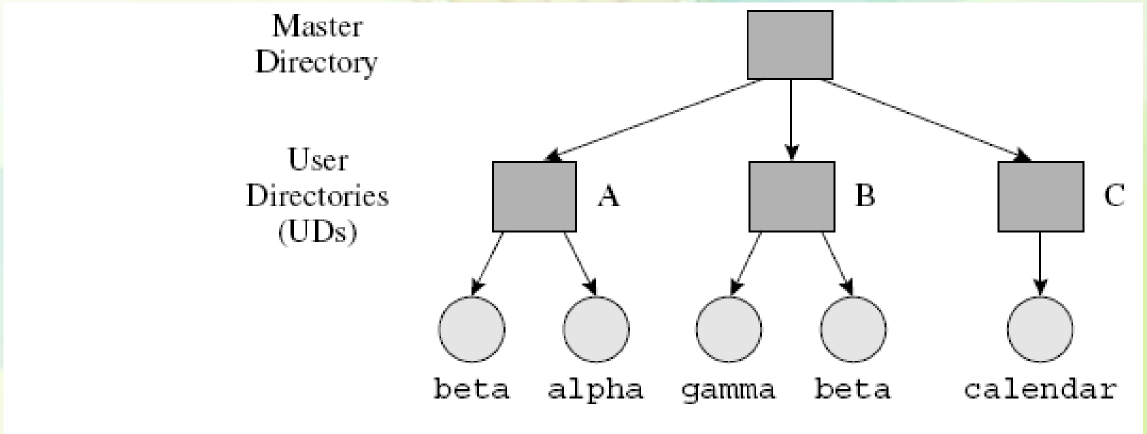


Figure 13.7 A directory structure composed of master and user directories.

Directory Trees

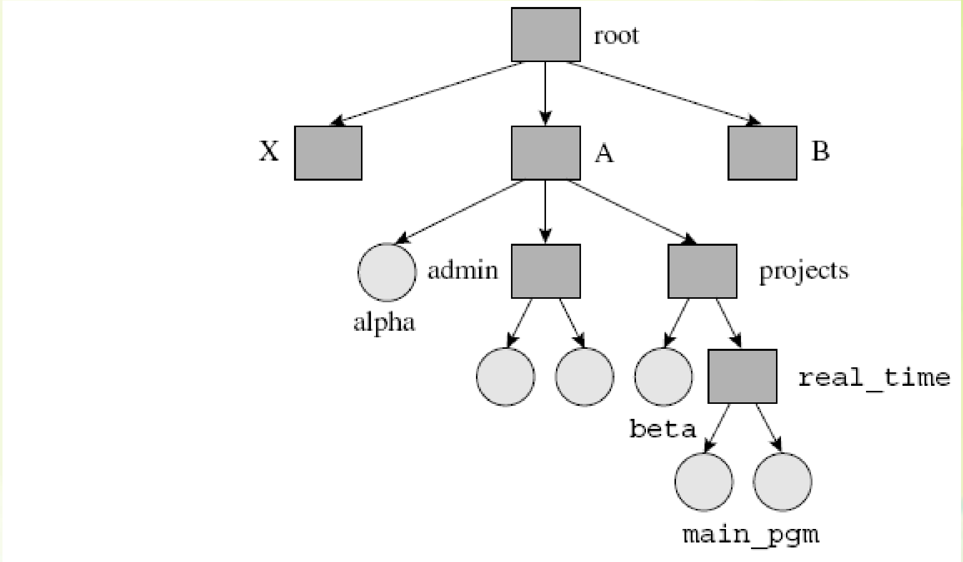


Figure 13.8 Directory trees of the file system and of user A.

- Some concepts: *home* directory, current *directory*
- *Path names* used to uniquely identify files
  - *Relative path name*
  - *Absolute path name*

Directory Graphs

- Tree structure leads to a fundamental asymmetry in the way different users can access a shared file
  - Solution: use *acyclic graph structure* for directories
    - A *link* is a directed connection between two existing files in the directory structure

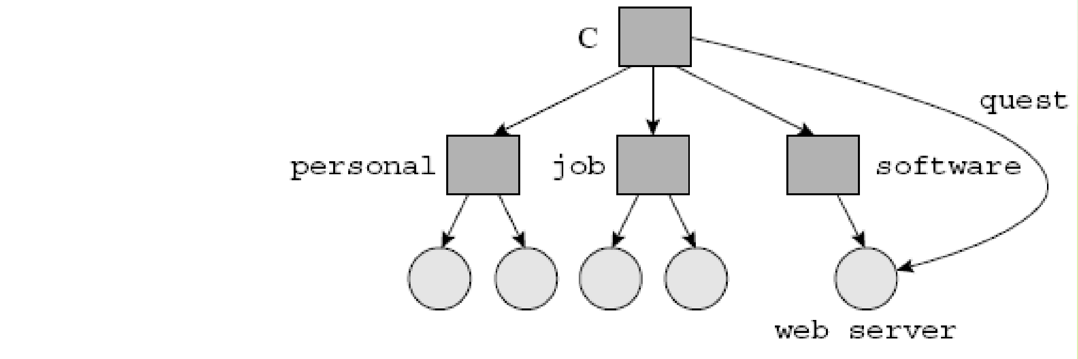


Figure 13.9 A link in the directory structure.

Operations on Directories

- Most frequent operation on directories: search
- Other operations are maintenance operations like:
  - Creating or deleting files
  - Updating file entries (upon a close operation)
  - Listing a directory
  - Deleting a directory
- Deletion becomes complicated when directory structure is a graph
  - A file may have multiple parents
  - File system maintains a link count with each file

Organization of Directories

- Flat file that is searched linearly inefficient
- Hash table directory efficient search
  - *Hash with open addressing* requires a single table
  - (Sometimes) at most two comparisons needed to locate a file
  - Cumbersome to change size, or to delete an entry
- B+ tree directory fast search, efficient add/delete
  - *m*-way search tree where $m \leq 2 \times d$ *(d: order* of tree)
  - Balanced tree: fast search
  - File information stored in leaf nodes
  - Nonleaf nodes of the tree contain *index entries*

Directory as a B+ tree

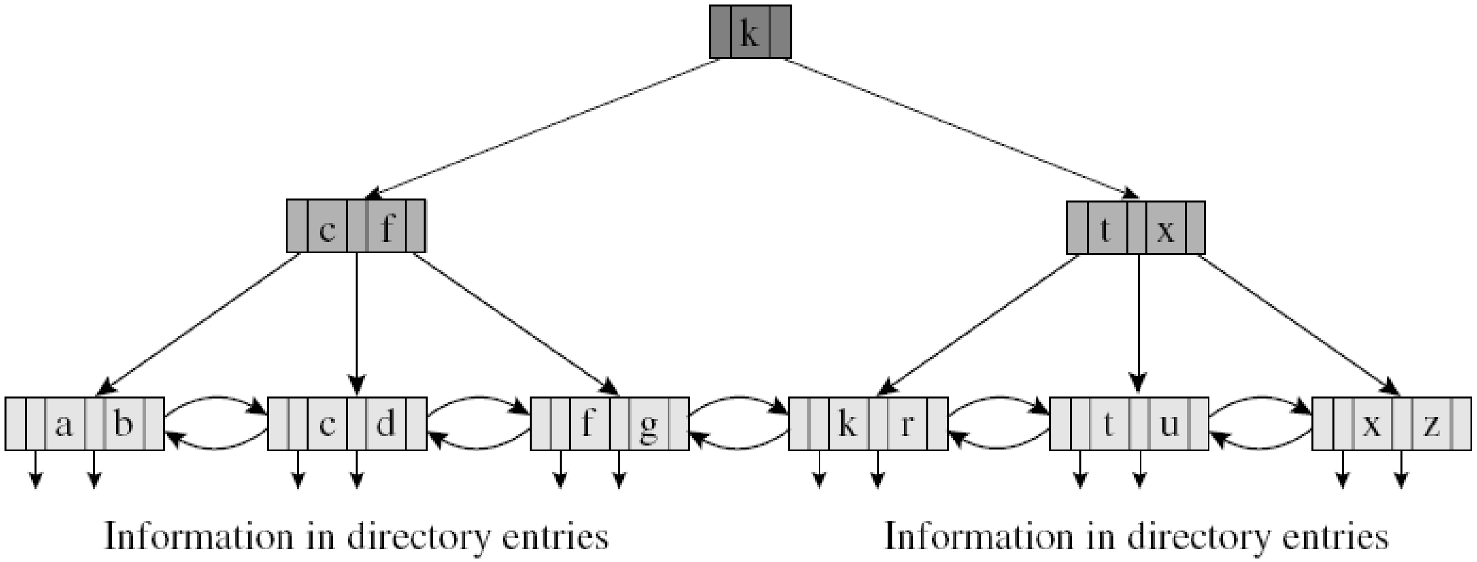

k
c
f
t
x
a
b
c
d
f
g
k
r
t
u
x
z
Information in directory entries
Information in directory entries

Figure 13.10 A directory organized as a B+ tree.

Mounting of File Systems

- There can be many file systems in an OS
- Each file system is constituted on a *logical disk*
  - i.e., on a partition of a disk
- Files can be accessed only when file system is *mounted*

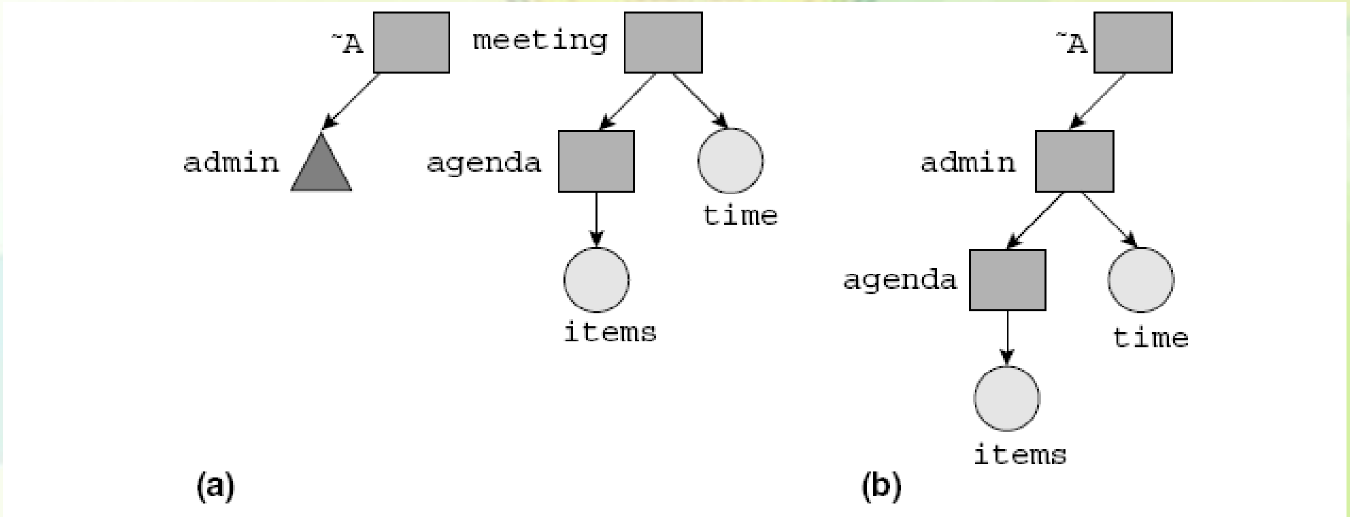


Figure 13.11 Directory structures (a) before a mount command; (b) after a mount command.

File Protection

- Users need *controlled sharing* of files
  - *Protection info* field of the file's directory entry used to control access to the file
- Usually, protection info. stored in *access control list*
  - List of (<*user_name*>,<*list_of_access_privileges*>)
    - User groups can be used to reduce size of list
- In most file systems, privileges are of three kinds:
  - *Read*
  - *Write*
  - *Execute*

Allocation of Disk Space

- Disk space allocation is performed by file system
- Before contiguous memory allocation model
  - Led to external fragmentation
- Now noncontiguous memory allocation model
  - Issues:
    - Managing free disk space
      - Use: *free list* or *disk status map* (DSM)
    - Avoiding excessive disk head movement
      - Use: *Extents (clusters*) or *cylinder groups*
    - Accessing file data
      - Depends on approach: *linked* or *indexed*

Allocation of Disk Space (continued)

- The DSM has one entry for each disk block
  - Entry indicates if block is free or allocated to a file
  - Information can be maintained in a single bit
    - DSM also called a *bit map*
- DSM is consulted every time a new disk block has to be allocated to a file

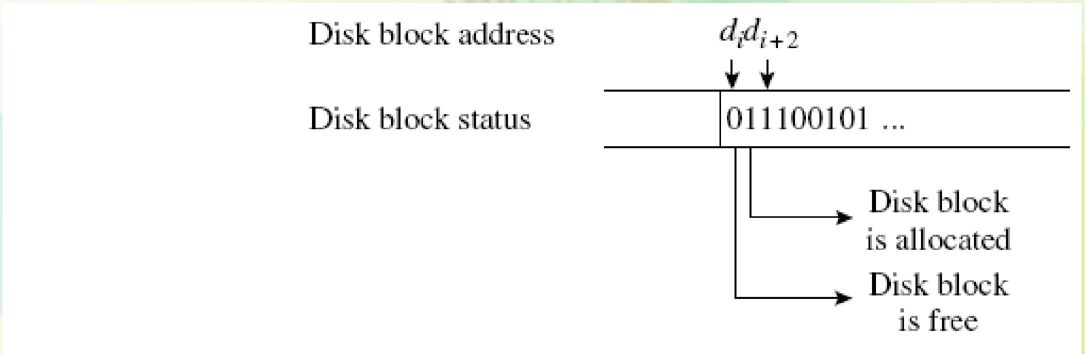


Figure 13.12 Disk status map (DSM).

Linked Allocation

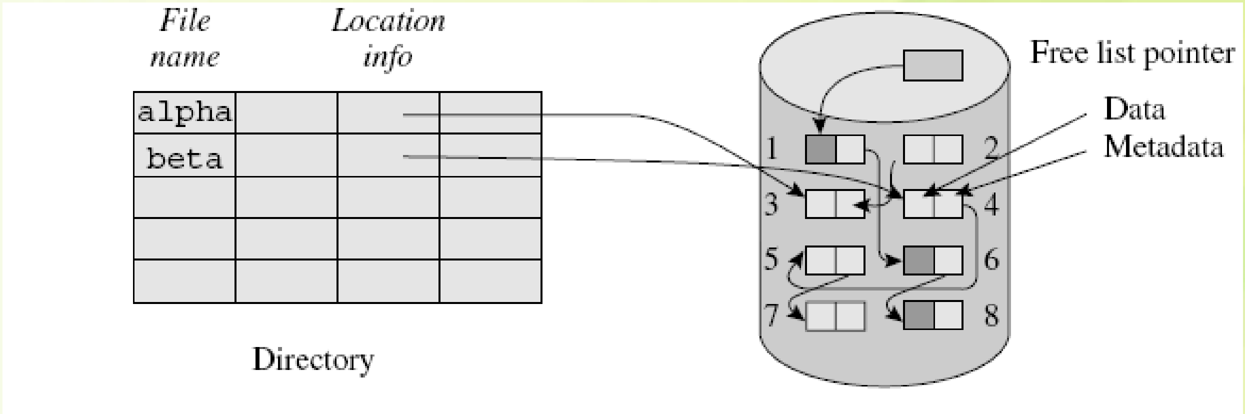


Figure 13.13 Linked allocation of disk space.

- Each disk block has data, address of next disk block
  - Simple to implement
  - Low allocation/deallocation overhead
- Supports sequential files quite efficiently
- Files with nonsequential organization cannot be accessed efficiently
- Reliability is poor (metadata corruption)

Linked Allocation (continued)

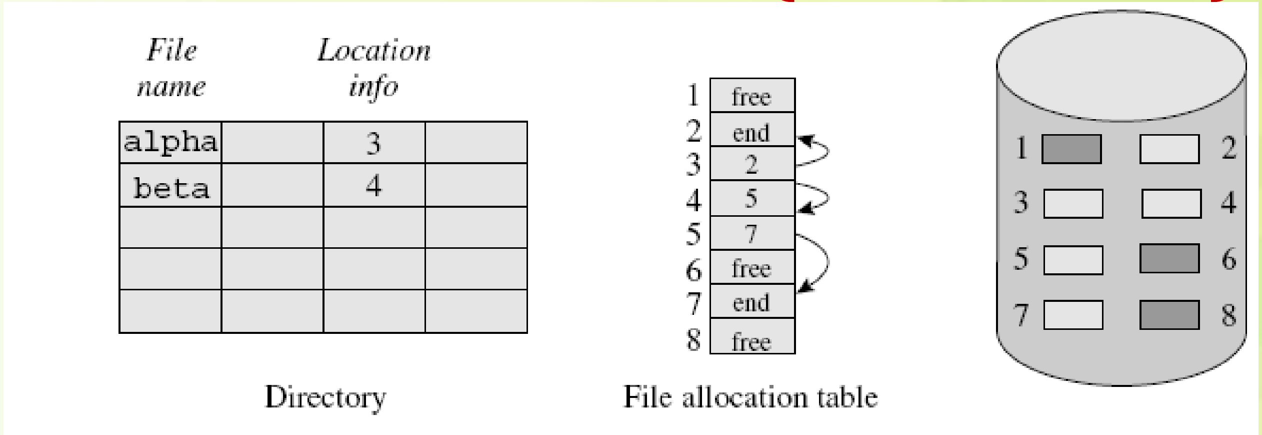


Figure 13.14 File Allocation Table (FAT).

- MS-DOS uses a variant of linked allocation that stores the metadata separately from the file data
- FAT has one element corresponding to every disk block in the disk
  - Penalty: FAT has to be accessed to obtain the address of the next disk block
    - Solution: FAT is held in memory during file processing

Indexed Allocation

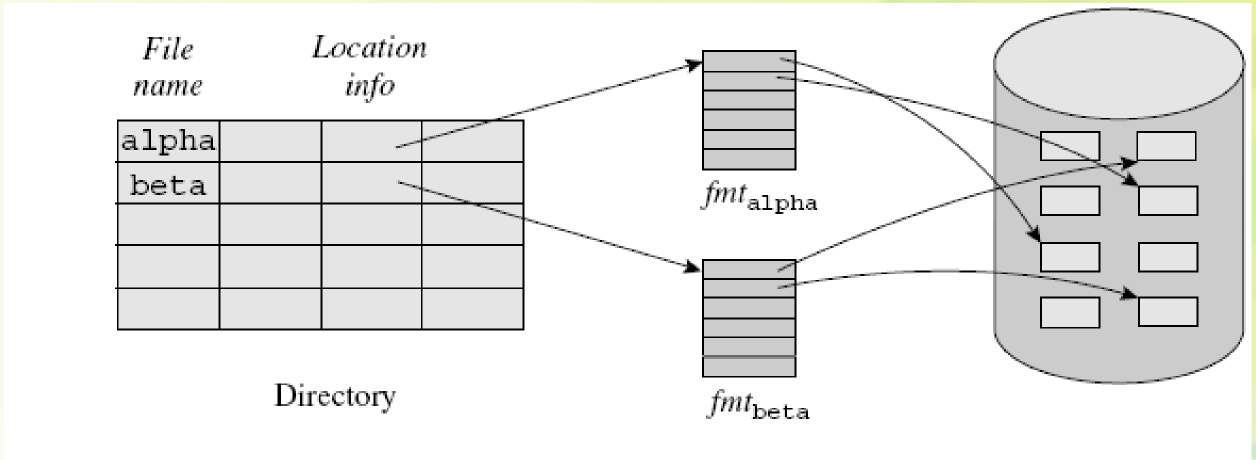


Figure 13.15 Indexed allocation of disk space.

- An index (*file map table* (FMT)) is maintained to note the addresses of disk blocks allocated to a file
  - Simplest form: FMT can be an array of disk block addresses

Indexed Allocation (continued)

- Other variations:
  - Two-level FMT organization: compact, but access to data blocks is slower
  - Hybrid FMT organization: small files of *n* or fewer data blocks continue to be accessible efficiently

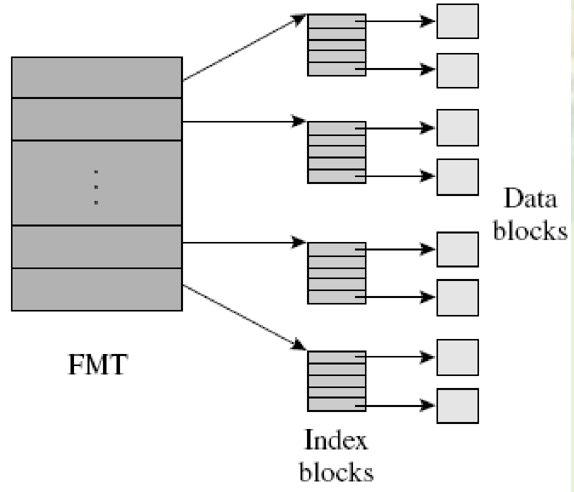


Figure 13.16 A two-level FMT organization.

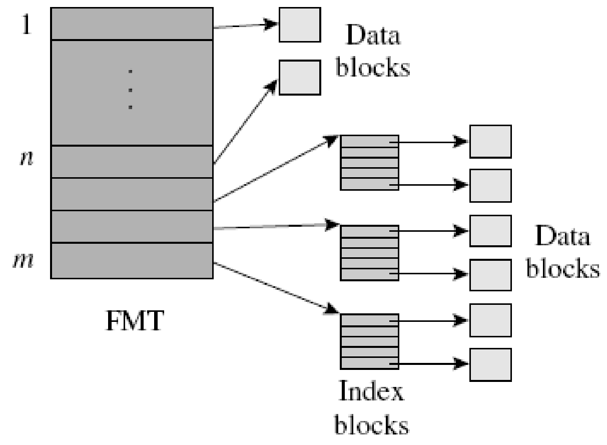


Figure 13.17 A hybrid organization of FMT.

Performance Issues

- Issues related to use of disk block as allocation unit
  - Size of the metadata
  - Efficiency of accessing file data
- Both addressed using a larger unit of allocation
  - Use the extent as a unit of disk space allocation
    - *Extent*: set of consecutive disk blocks
    - Large extents provide better access efficiency
      - Problem: more internal fragmentation
      - Solution: variable extent sizes
        - » Size is indicated in metadata

Interface Between File System and IOCS

- Interface between file system and IOCS consists of
  - *File map table* (FMT)
  - *Open files table* (OFT)
  - *File control block* (FCB)

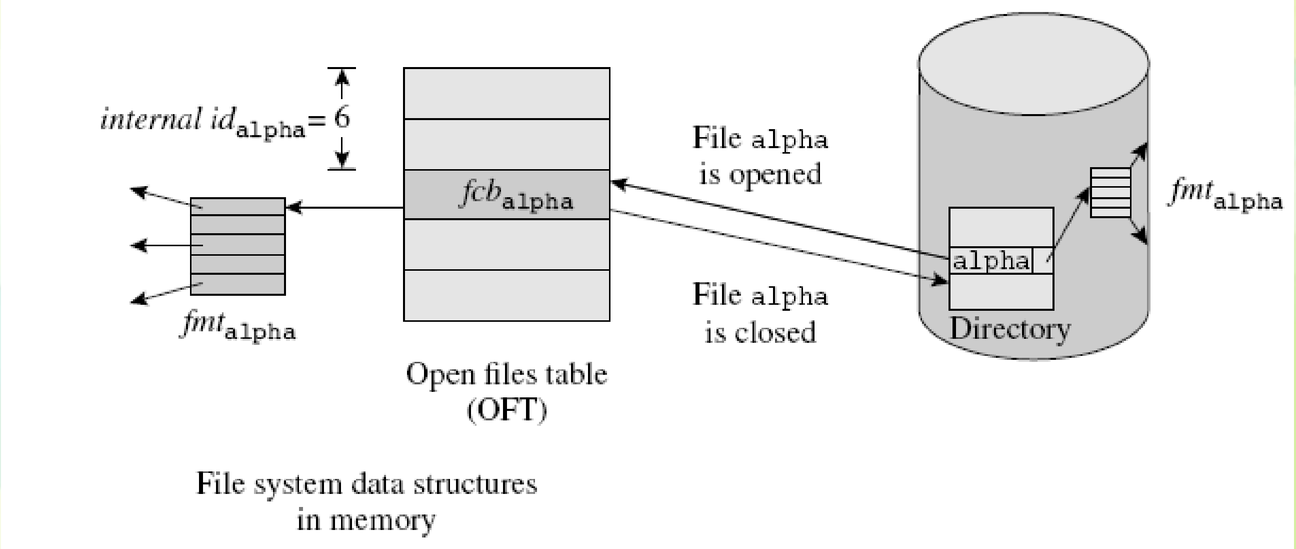


Figure 13.18 Interface between file system and IOCS—OFT, FCB and FMT.

Interface Between File System and IOCS (continued)

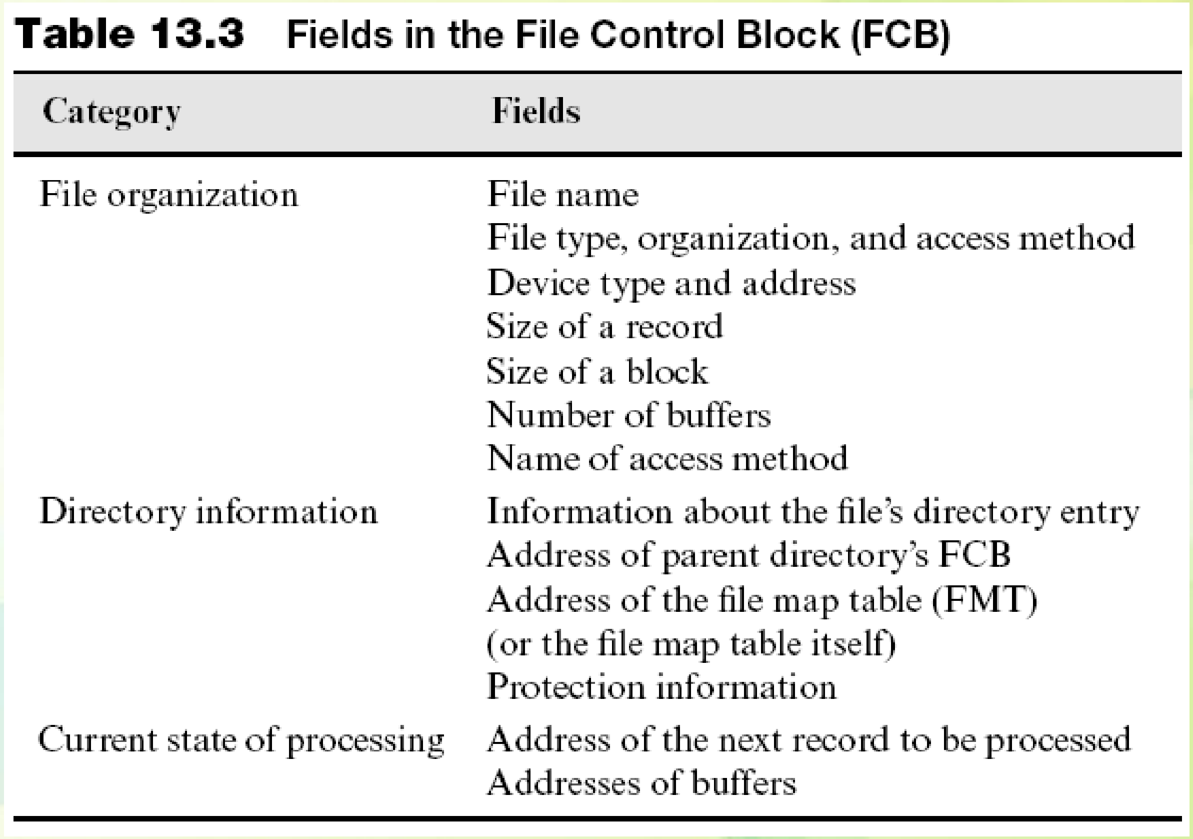


Table 13.3 Fields in the File Control Block (FCB)

Category	Fields
File organization	File name
	File type, organization, and access method
	Device type and address
	Size of a record
	Size of a block
	Number of buffers
	Name of access method
Directory information	Information about the file's directory entry
	Address of parent directory's FCB
	Address of the file map table (FMT)
	(or the file map table itself)
	Protection information
Current state of processing	Address of the next record to be processed
	Addresses of buffers

Interface Between File System and IOCS (continued)

When `alpha` is opened:

- File system copies $\text{FMT}_{\text{alpha}}$ in memory
- Creates fcb_{alpha} in the OFT
- Initializes fields appropriately
- Passes offset in OFT to process, as $internal\_id_{\text{alpha}}$

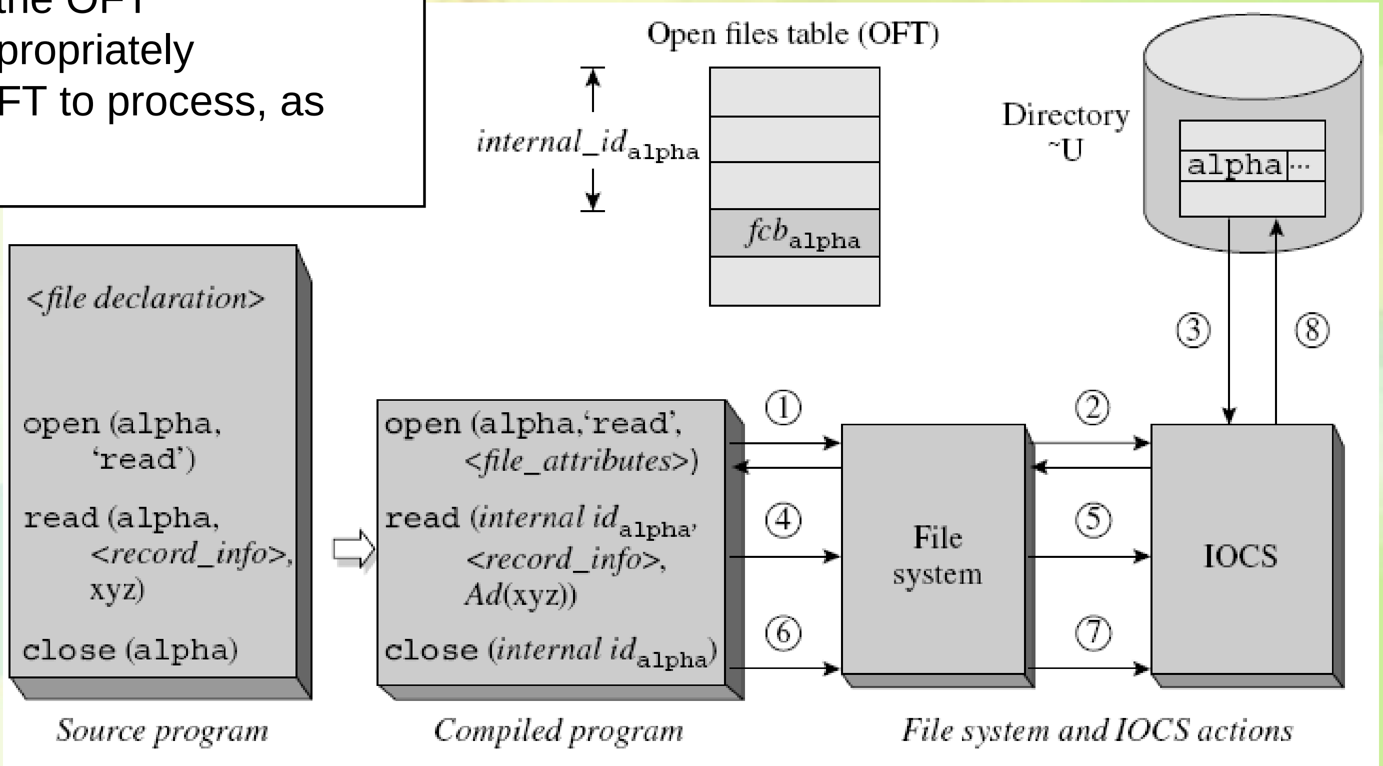


Figure 13.19 Overview of file processing.

File Processing

- File System Actions at **open**
  - Sets up the arrangement involving FCB and OFT
- File System Actions during a File Operation
  - Performs disk space allocation if necessary
- File System Actions at **close**
  - Updates directories if necessary

File system actions at open

- Perform *path name resolution*
  - For each component in the path name, locate the correct directory or file
  - Handle path names passing through mount points
    - A file should be allocated disk space in its own file system
  - Build FCB for the file
- Retain sufficient information to perform a *close* operation on the file
  - Close may have to update the file’s entry in the parent directory
  - It may cause changes in the parent directory’s entry in ancestor directories

File System Actions at open

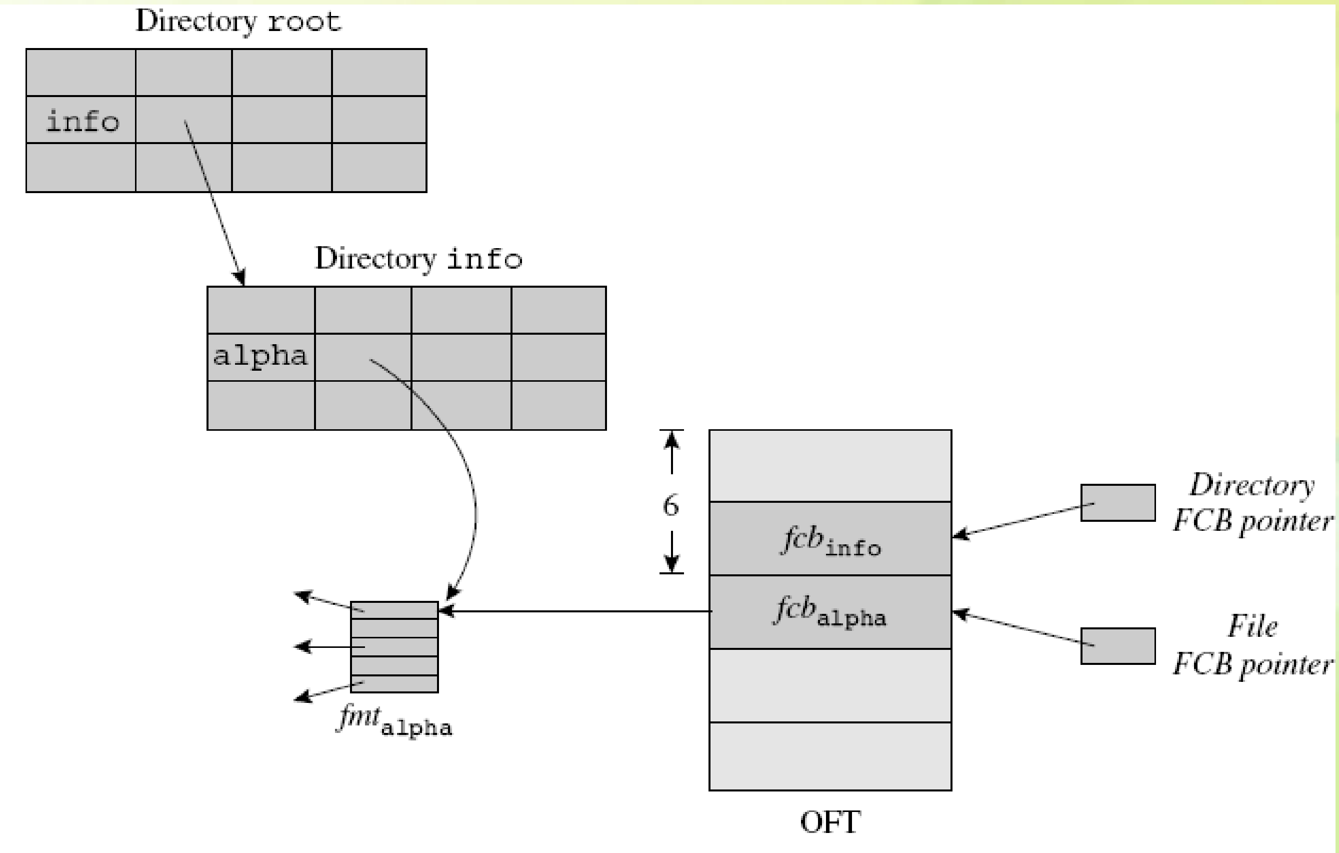


Figure 13.20 File system actions at open.

File System Actions during a File Operation

- Each file operation is translated into a call:
  - *< opn > (internal_id, record_id,< IO_areaaddr >)*;
    - *Internal_id* is the internal id of *<file_name>* returned by the open call
    - *Record_id* is absent for sequential-access files
      - Operation is performed on the *next* record
- Disk block address obtained from *record_id*

File System Actions at close

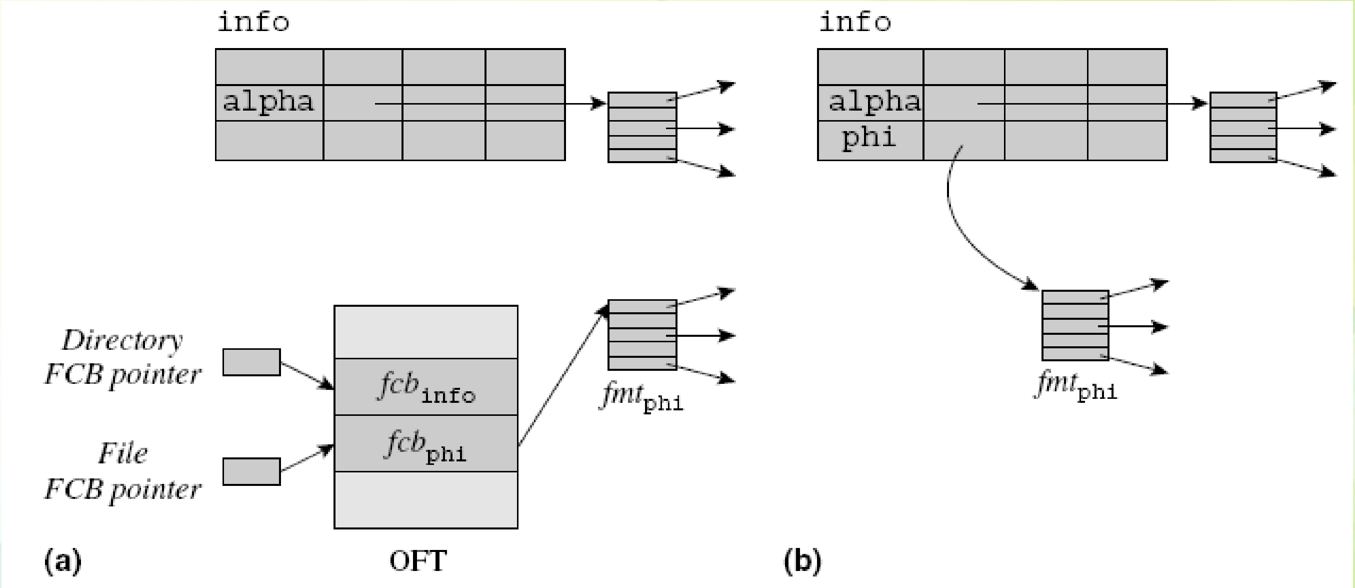


Figure 13.21 File system data structures (a) before; (b) after `close`.

File Sharing Semantics

- File system provides two methods of file sharing for processes to choose from:
  - Sequential sharing
    - Only one process accesses a file at a time
    - Implemented through *lock* field in file's directory entry
  - Concurrent sharing
    - System creates a separate FCB for each process
    - Three sharing modes exist (see Table 13.4)
- *File sharing semantics:*
  - Determine how results of file manipulations performed by concurrent processes are visible

File Sharing Semantics (continued)

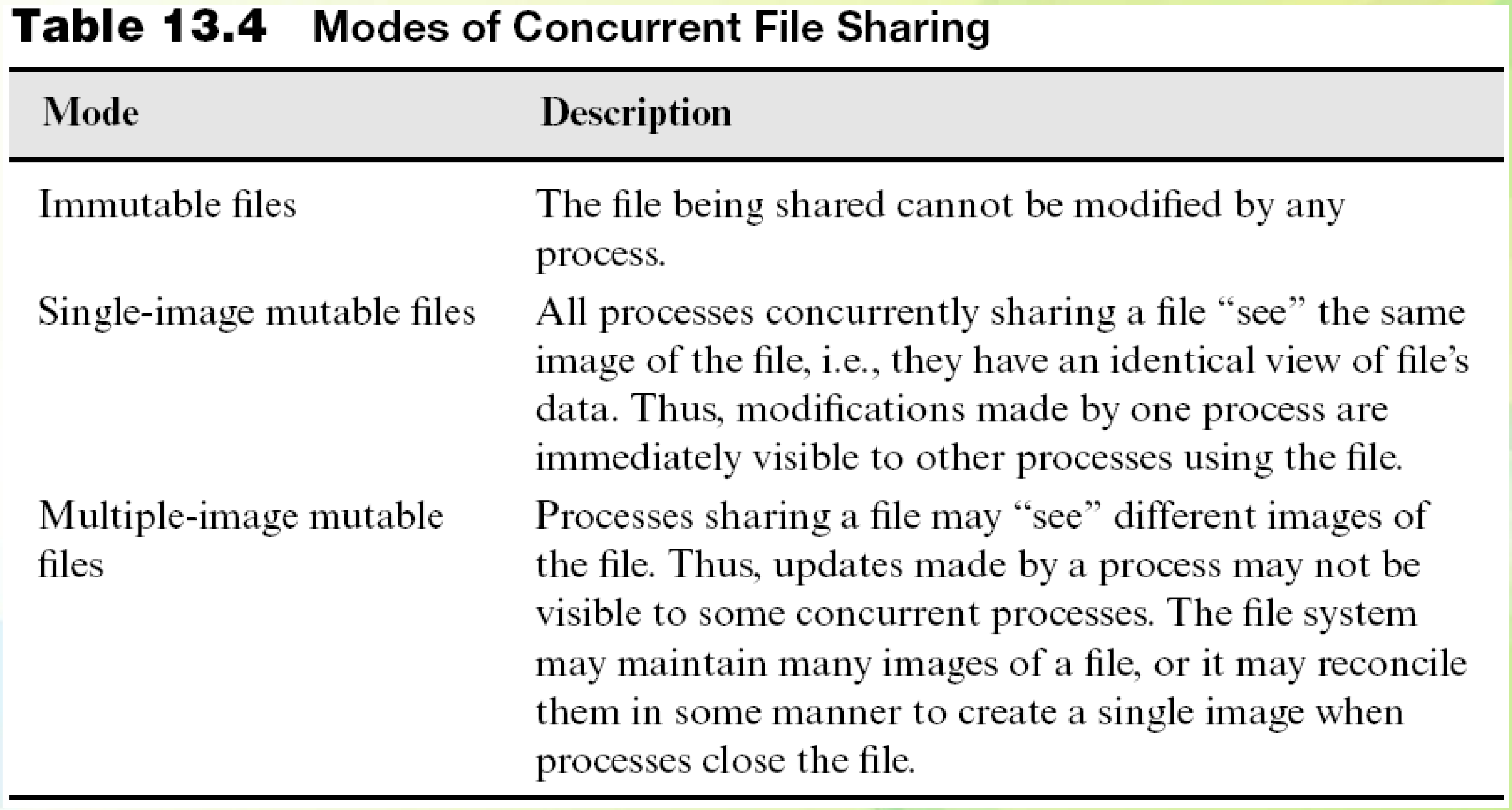


Table 13.4 Modes of Concurrent File Sharing

Mode	Description
Immutable files	The file being shared cannot be modified by any process.
Single-image mutable files	All processes concurrently sharing a file "see" the same image of the file, i.e., they have an identical view of file's data. Thus, modifications made by one process are immediately visible to other processes using the file.
Multiple-image mutable files	Processes sharing a file may "see" different images of the file. Thus, updates made by a process may not be visible to some concurrent processes. The file system may maintain many images of a file, or it may reconcile them in some manner to create a single image when processes close the file.

Single-image Mutable Files

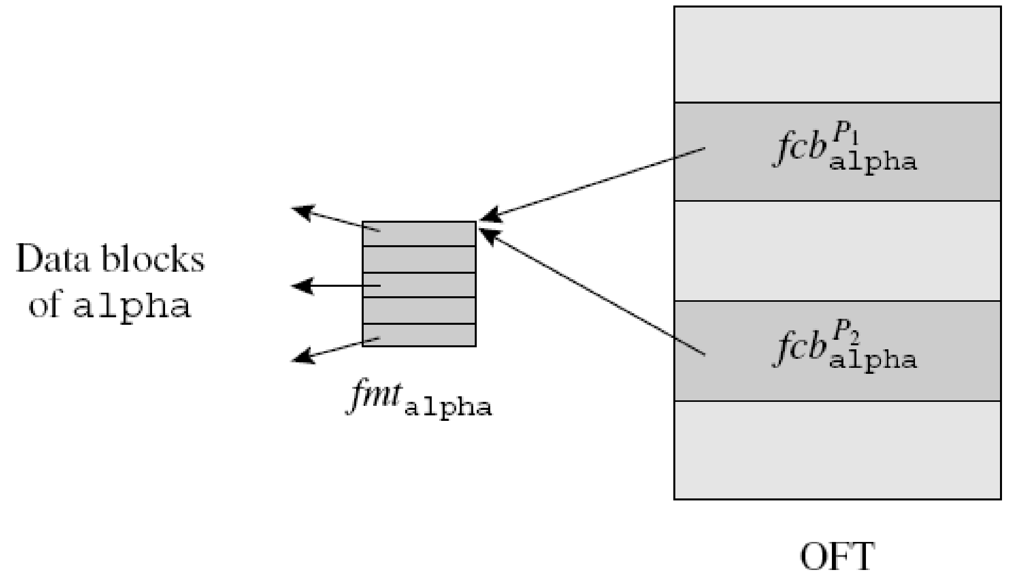


Figure 13.22 Concurrent sharing of a single-image mutable file by processes P_1 and P_2.

Multiple-image Mutable Files

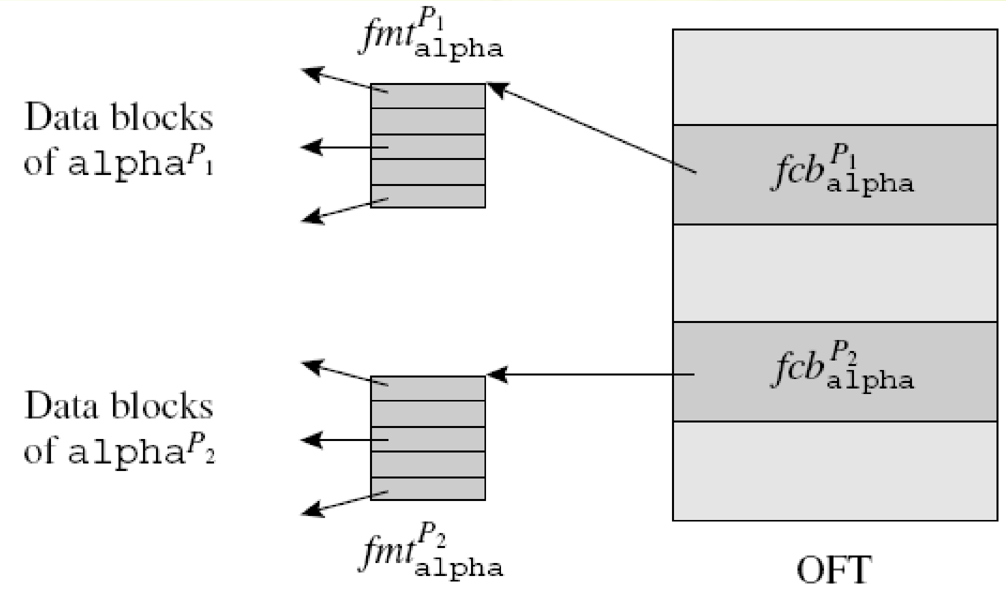

$fmt^{P_1}_{\texttt{alpha}}$
Data blocks of $\texttt{alpha}^{P_1}$
$fcb^{P_1}_{\texttt{alpha}}$
$fcb^{P_2}_{\texttt{alpha}}$
Data blocks of $\texttt{alpha}^{P_2}$
$fmt^{P_2}_{\texttt{alpha}}$
OFT

Figure 13.23 Concurrent sharing of a multiple-image mutable file by processes P_1 and P_2.

File System Reliability

- Degree to which a file system will function correctly even when faults occur
  - E.g., data corruption in disk blocks, system crashes due to power interruptions
- Two principal aspects are:
  - Ensuring correctness of file creation, deletion, and updates
  - Preventing loss of data in files
- Fault: defect in some part of the system
  - Occurrence of a fault causes a failure
- Failure: system behavior that is erroneous
  - Or that differs from its expected behavior

Loss of File System Consistency

- *File system consistency* implies correctness of metadata and correct operation of the file system
- A fault may cause following failures:
  - Some data from an open file may be lost
  - Part of an open file may become inaccessible
  - Contents of two files may get mixed up
- For example, consider addition of a disk block to a file and a fault during step 3:

```
1. $d_j$.next := $d_1$.next;
2. $d_1$.next := address ($d_j$);
3. Write $d_1$ to disk.
4. Write $d_j$ to disk.
```

Loss of File System Consistency (continued)

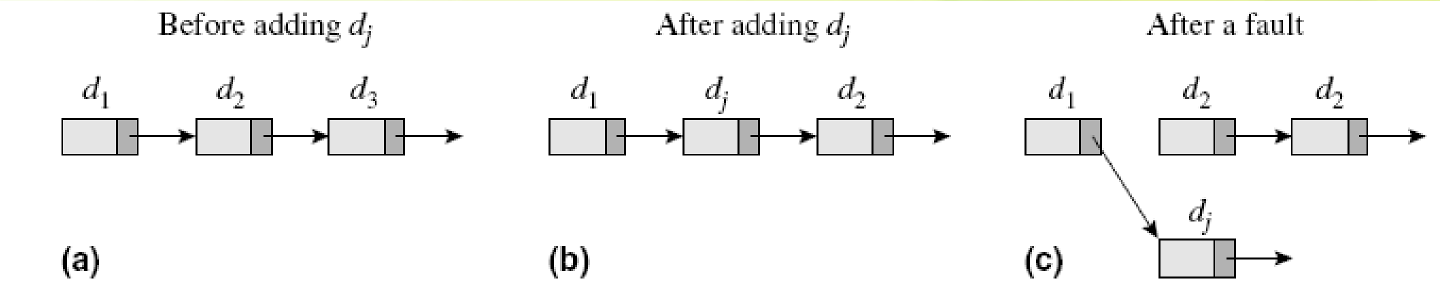


Figure 13.24 Inconsistencies in metadata due to faults: (a)–(b) before and after adding d_j during normal operation; (c) after a fault.

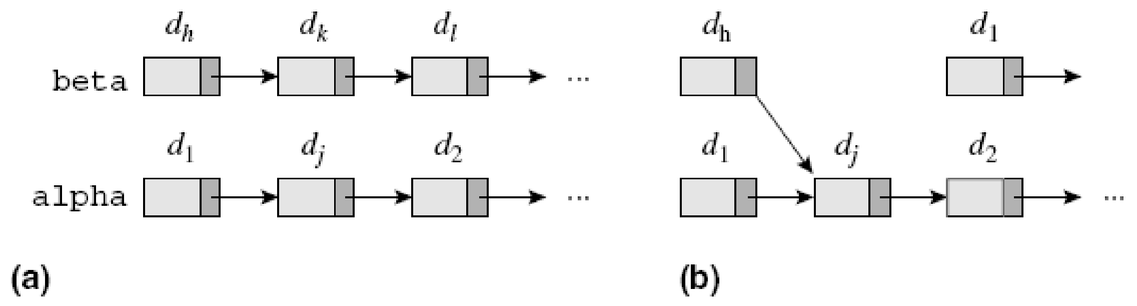


Figure 13.25 Files `alpha` and `beta`: (a) after adding d_j during normal operation; (b) if $d_j = d_k$, `alpha` is closed and a power outage occurs.

Approaches to File System Reliability

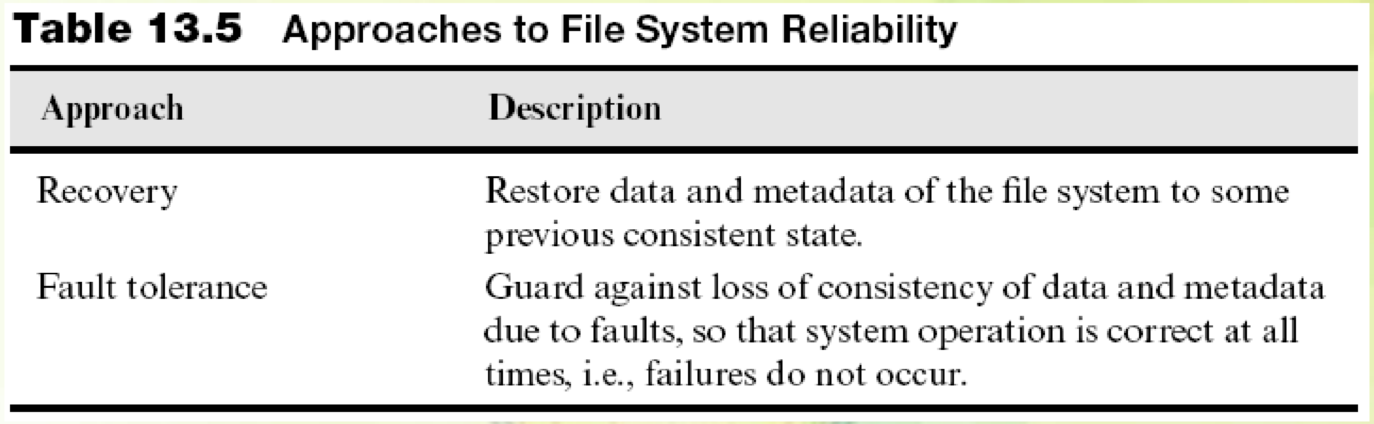


Table 13.5 Approaches to File System Reliability

Approach	Description
Recovery	Restore data and metadata of the file system to some previous consistent state.
Fault tolerance	Guard against loss of consistency of data and metadata due to faults, so that system operation is correct at all times, i.e., failures do not occur.

- *Recovery* is a classic approach that is activated when a failure is noticed
- *Fault tolerance* provides correct operation of file system at all times

Recovery Techniques

- A *backup* is a recording of the *file system state*
  - Overhead of creating backups
    - When indexed allocation of disk space is used, it is possible to create an on-disk backup of a file cheaply with technique that resembles *copy-on-write* of virtual memory
  - Overhead of reprocessing
    - Operations performed after lash backup have to be reprocessed
  - Solution: Use a combination of backups and incremental backups

Recovery Techniques (continued)

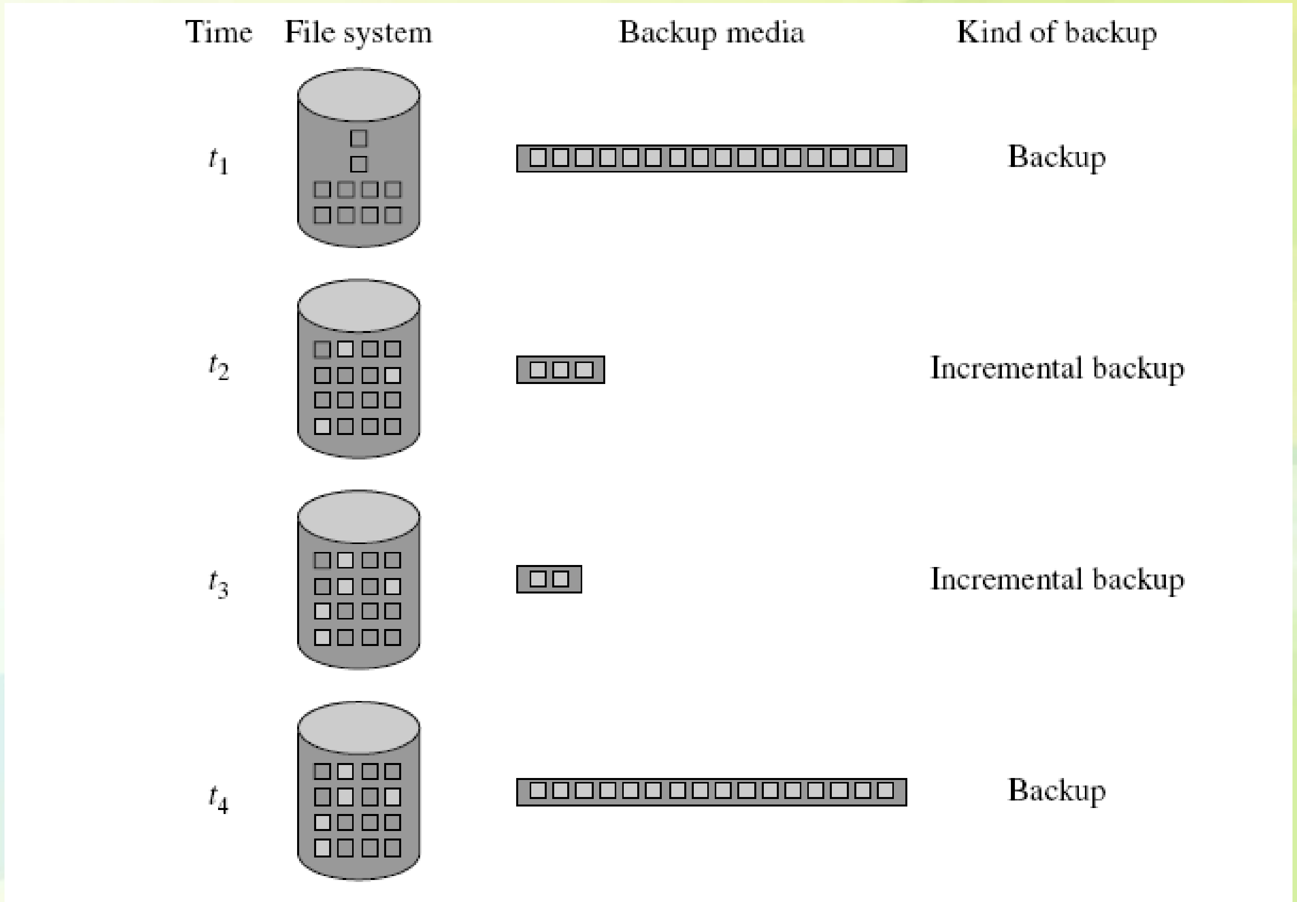


Figure 13.26 Backups and incremental backups in a file system.

Recovery Techniques (continued)

- To reduce overhead of creating backups (when indexed allocation is used) only the FMT and disk block whose contents are updated after the backup is created would be copied
  - Conserves both disk space and time

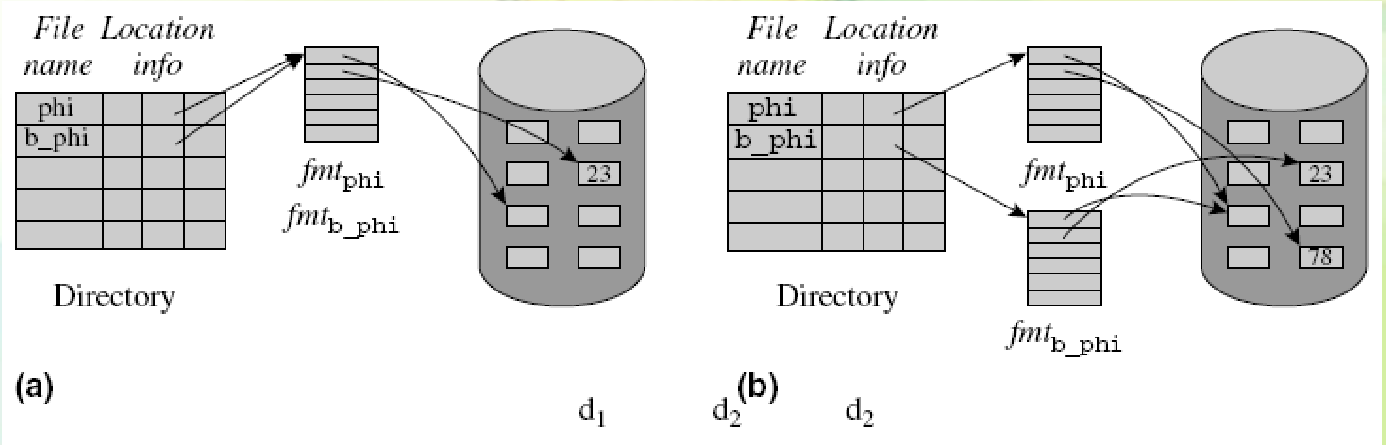


Figure 13.27 Creating a backup: (a) after backing up file `phi`; (b) when `phi` is modified.

Fault Tolerance Techniques

- File system reliability can be improved by taking two precautions:
  - Preventing loss of data or metadata due to I/O device malfunction
    - Approach: use stable storage
  - Preventing inconsistency of metadata due to faults
    - Approach: use atomic actions

Stable Storage

- Maintain two copies of data
  - Can tolerate one fault in recording of a data item
  - Incurs high space and time overhead
  - Can't indicate if copy that survived is old or new

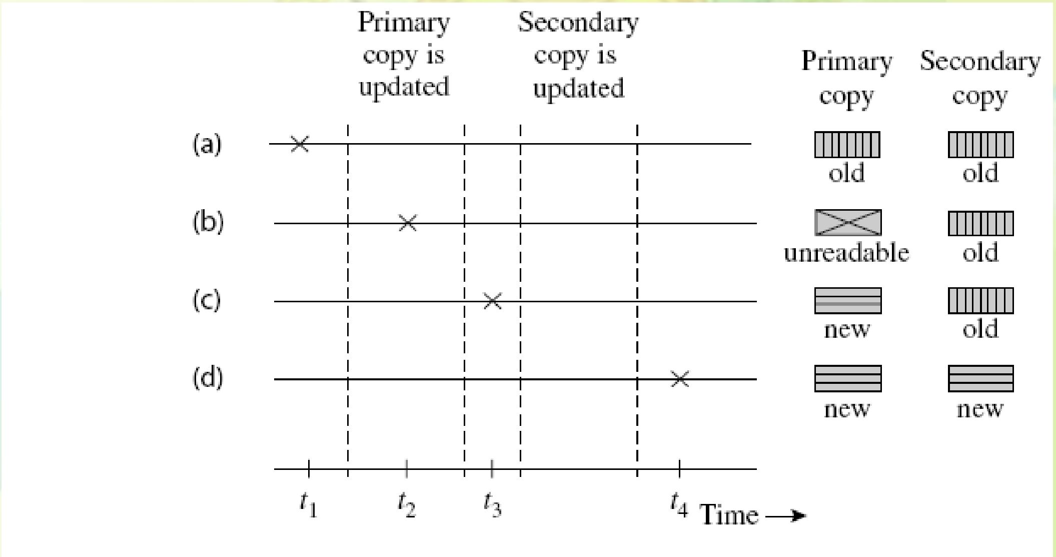


Figure 13.28 Fault tolerance using the stable storage technique.

Atomic Actions

Definition 13.1 Atomic Action An action that consists of a set of subactions and whose execution has the property that either

1. The effects of all of its subactions are realized, or
2. The effects of none of its subactions are realized.

```
begin atomic action add_a_block;
    $d_j$.next := $d_1$.next;
    $d_1$.next := address($d_j$);
    write $d_1$;
    write $d_j$;
end atomic action add_a_block;
```

Figure 13.29 Atomic action *add_a_block*.

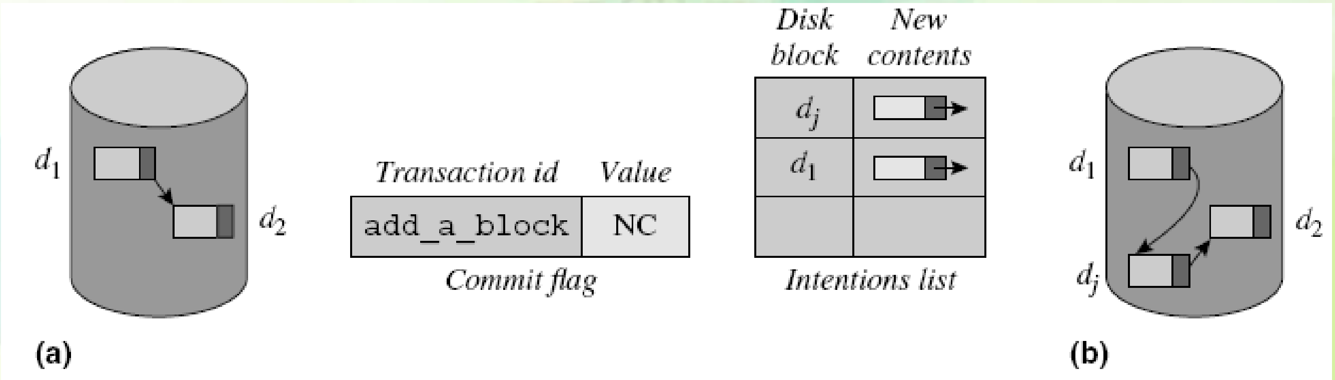
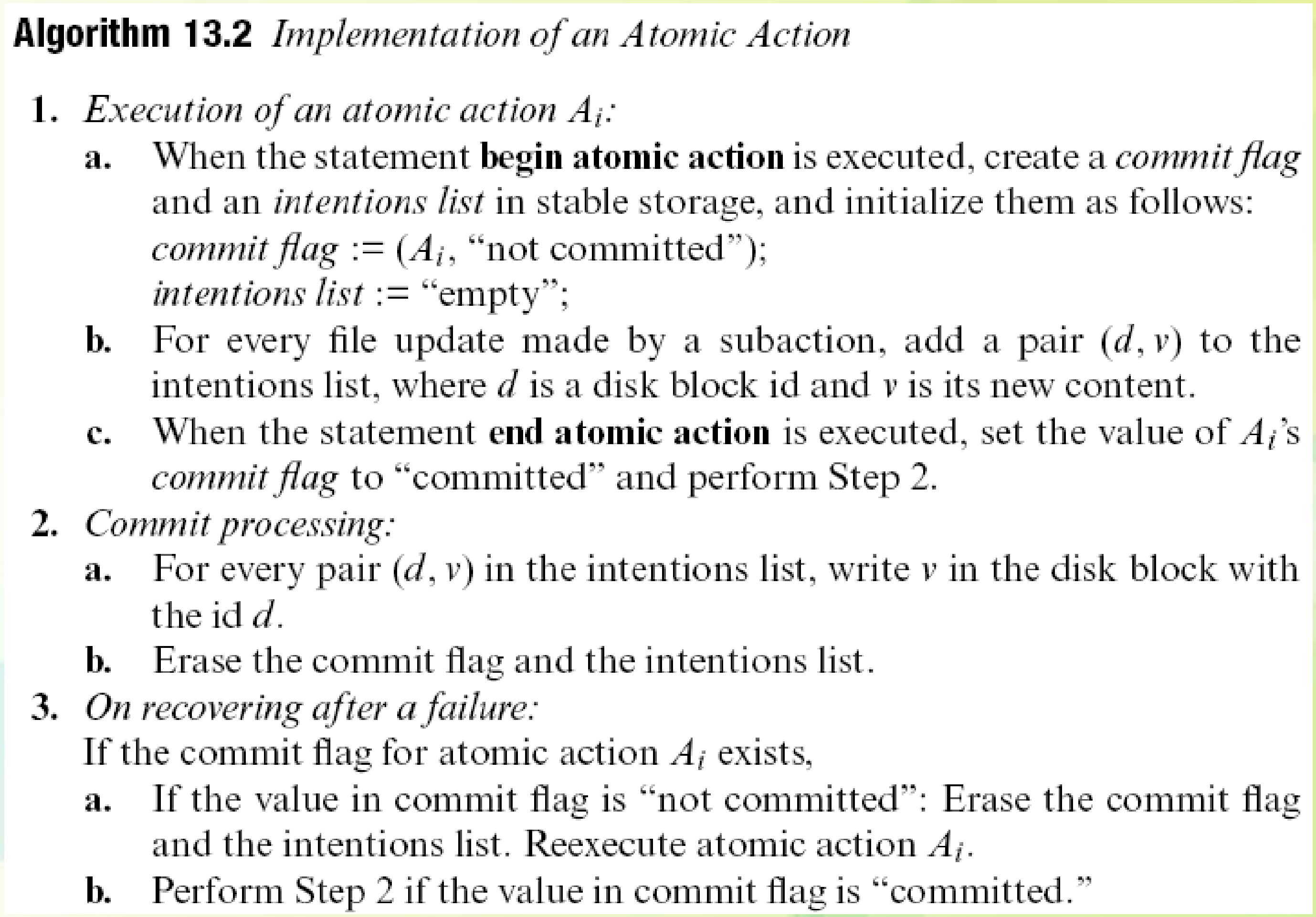


Figure 13.30 (a) Before and (b) after commit processing. (*Note:* NC means *not committed*.)

Atomic Actions (continued)



Algorithm 13.2 *Implementation of an Atomic Action*

1. *Execution of an atomic action A_i:*
   - **a.** When the statement **begin atomic action** is executed, create a *commit flag* and an *intentions list* in stable storage, and initialize them as follows:
     commit flag := (A_i, “not committed”);
     intentions list := “empty”;
   - **b.** For every file update made by a subaction, add a pair (d, v) to the intentions list, where d is a disk block id and v is its new content.
   - **c.** When the statement **end atomic action** is executed, set the value of A_i’s *commit flag* to “committed” and perform Step 2.
2. *Commit processing:*
   - **a.** For every pair (d, v) in the intentions list, write v in the disk block with the id d.
   - **b.** Erase the commit flag and the intentions list.
3. *On recovering after a failure:*
   If the commit flag for atomic action A_i exists,
   - **a.** If the value in commit flag is “not committed”: Erase the commit flag and the intentions list. Reexecute atomic action A_i.
   - **b.** Perform Step 2 if the value in commit flag is “committed.”

Journaling File Systems

- An *unclean* shutdown results in loss of data
  - Traditional approach: recovery techniques
  - Modern approach: use fault tolerance techniques so system can resume operation quickly after shutdown
    - A *journaling file system* implements fault tolerance by maintaining a *journal*

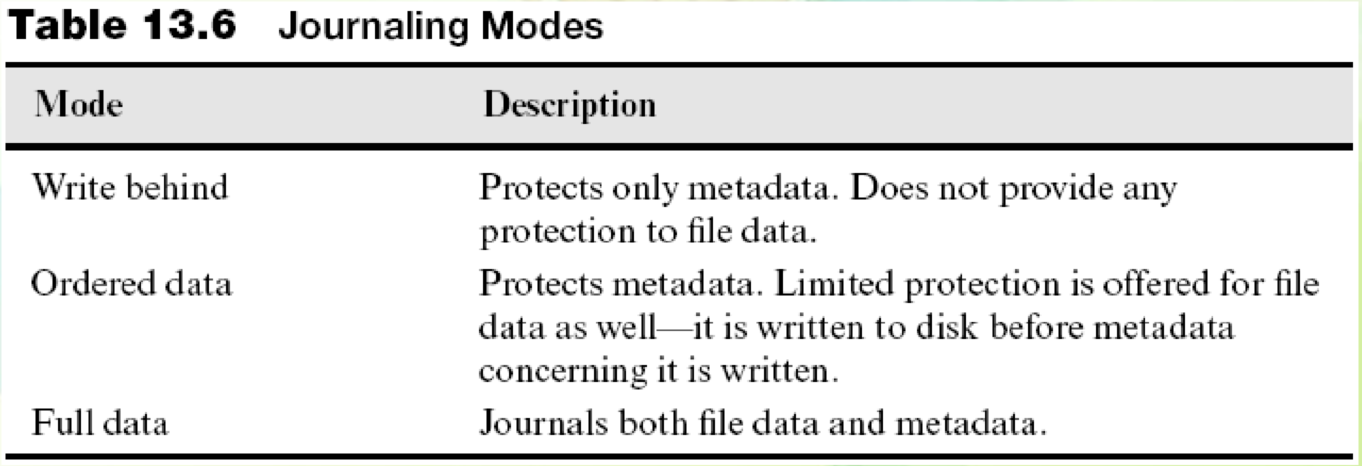


Table 13.6 Journaling Modes

Mode	Description
Write behind	Protects only metadata. Does not provide any protection to file data.
Ordered data	Protects metadata. Limited protection is offered for file data as well—it is written to disk before metadata concerning it is written.
Full data	Journals both file data and metadata.

Virtual File System

- A *virtual file system* (VFS) facilitates simultaneous operation of several file systems
  - It provides generic open, close, read and write
  - Invokes operations of a specific file system

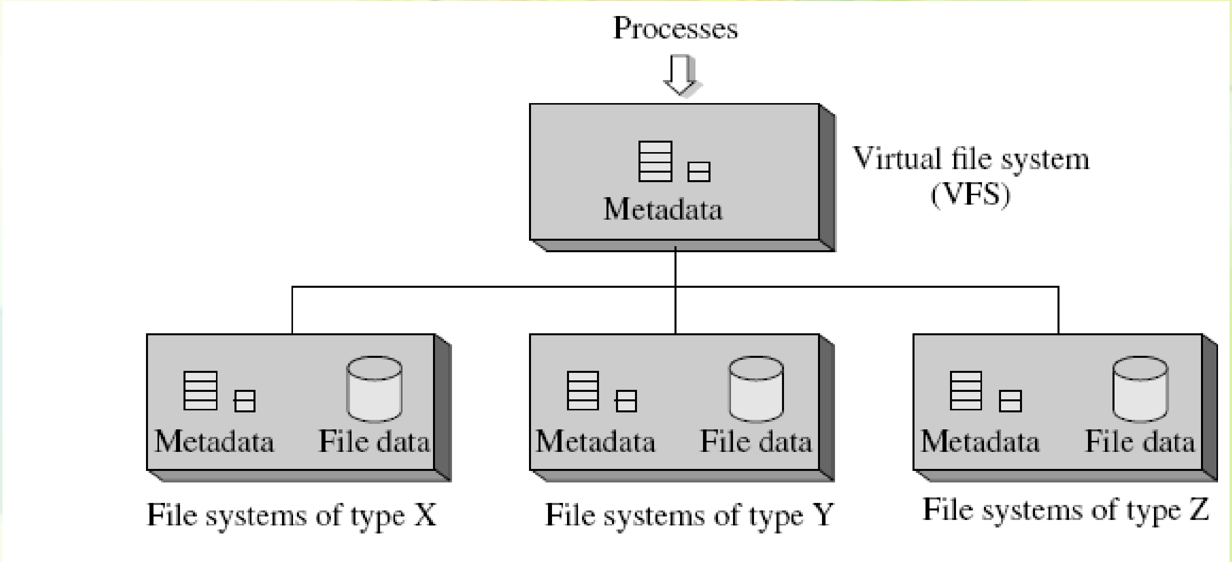


Figure 13.31 Virtual file system.

Case Studies of File Systems

- Unix File System
  - Berkeley Fast File System
- Linux File System
- Solaris File System
- Windows File System

Unix file system

- File system data structures
  - A directory entry contains only the file name
  - *Inode* of a file contains file size, owner id, access permissions and disk block allocation information
  - A file structure contains information about an open file
    - It contains current position in file, and pointer to its inode
  - A file descriptor points to a file structure
  - Indexed disk space allocation uses 3 levels of indirection
- Unix file sharing semantics
  - Result of a write performed by a process is immediately visible to all other processes currently accessing the file

Unix File System

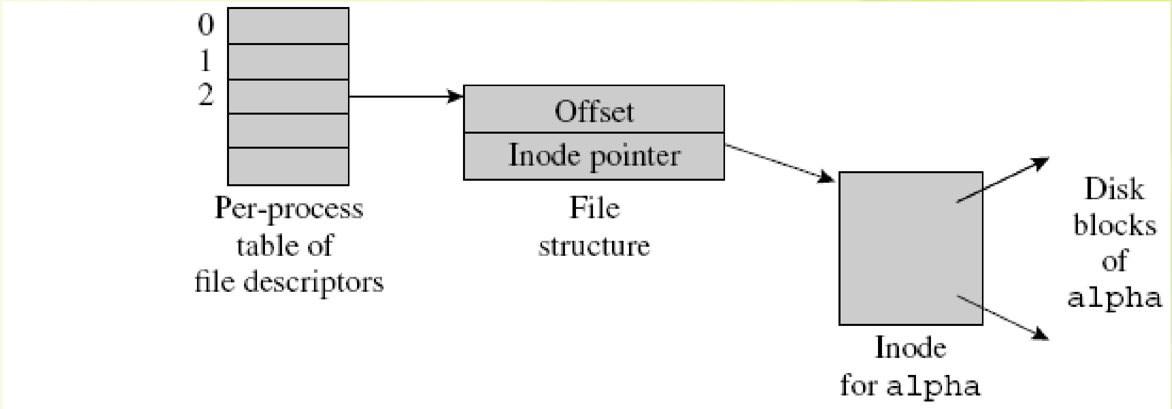


Figure 13.32 Unix file system data structures.

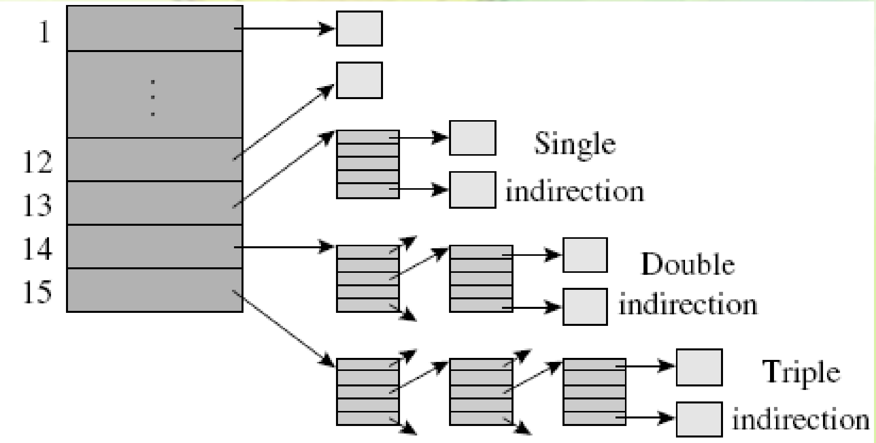


Figure 13.33 Unix file allocation table.

Berkeley Fast File System

- FFS was developed to address the limitations of the file system s5fs
- Supports some enhancements like long file names and use of symbolic links
- Includes several innovations concerning disk block allocation and disk access:
  - Permits use of large disk blocks (up to 8KB)
  - Uses *cylinder groups* to reduce disk head movement
  - Tries to minimize rotational latency when reading sequential files

Linux File System

- Linux provides a virtual file system (VFS)
  - Supports a common file model that resembles the Unix file model
- Standard file system is ext2
  - Variety of file locks for process synchronization
    - *Advisory locks, mandatory locks, leases*
  - Uses notion of a *block group*
- ext3 incorporates journaling

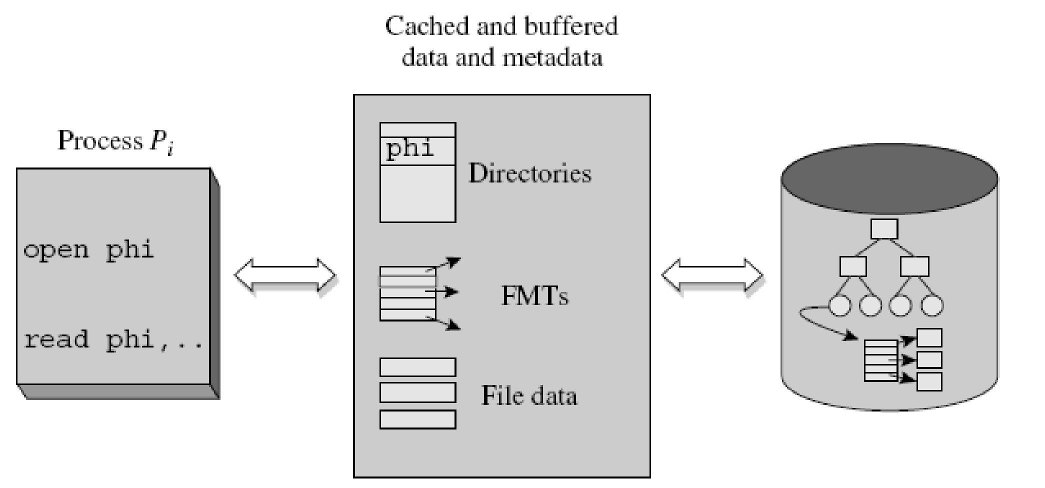
Solaris File System

- Unix-like file access permissions
  - Three access control pairs in each access control list
- Convenience and flexibility in file processing, through a virtual file system
- Record-level locking provided to implement fine-grained synchronization between processes
  - *Nonblocked I/O* mode to avoid indefinite waits
- *Asynchronous I/O* mode: a process is not blocked for its I/O operation to complete
- Provides file integrity

Windows File System

- NTFS is designed for servers and workstations
  - Key feature: recoverability of the file system
- Notion of *partition* and *volumes (*single *and spanned);* volumes have a *master file table* (MFT)
- Directory organized as a B+ tree
- Hard links and symbolic links (called *junctions*)
- Special techniques for sparse files and data compression
- Metadata modifications are atomic transactions
- *Write behind* capabilities of journaling file systems
- Vista has many new features for recovery

Performance of File Systems



Operation	Techniques employed for speedup
Directory access	Directory cache
Directory search	Hashtables, B+ trees
Accessing file map table	File map table cache in memory
Accessing a disk block	Disk block allocation in extents and cylinder groups, Disk block cache in memory, disk scheduling, disk block cache in I/O device
Accessing data	Buffering and blocking of data, or use of a file cache

Figure 13.34 Techniques employed to provide high file access performance.

Log-Structured File System

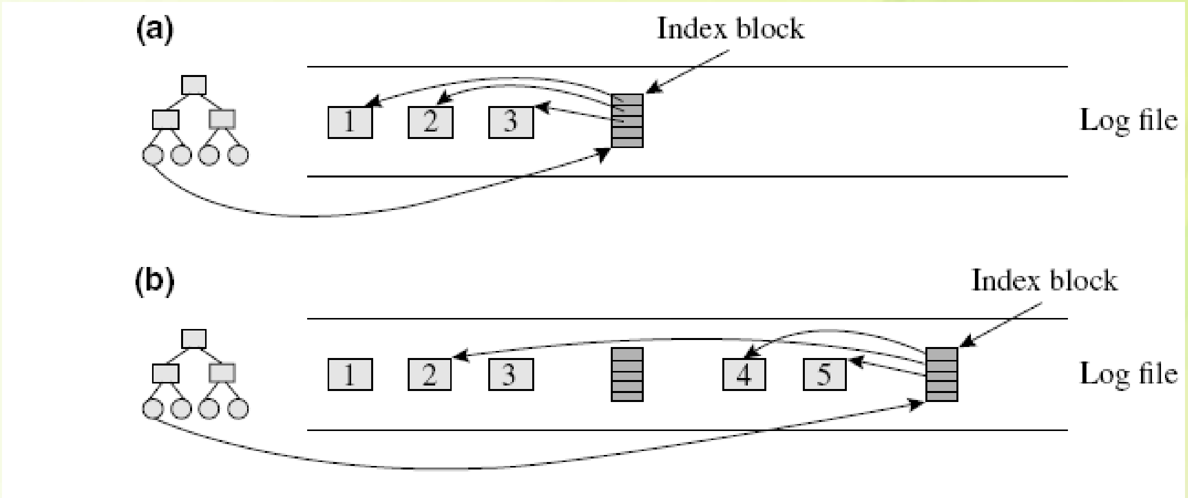


Figure 13.35 File update in a log-structured file system.

- Caching reduces disk head movement during reads
- *Log-structured file systems* reduce head movement through a radically different file organization
  - Writes file data of *all* files in a single sequential structure that resembles a journal (*log file)*
    - Little head movement during write operations

Summary

- Files are structured or unstructured *(byte stream)*
- File system provides:
  - *File organizations* (*sequential*, *direct*, *indexed*)
  - Directories for grouping of related files logically
  - *Sharing* and *protection* of files
  - Disk space allocation, typically indexed
    - File map table (FMT) stores allocation information
- File control block (FCB) stores information about a file's processing
- *Atomic actions* can be used for fault tolerance
- *Journaling file systems* provide reliability modes
- VFS permits several file systems to be in operation

Queries ....?