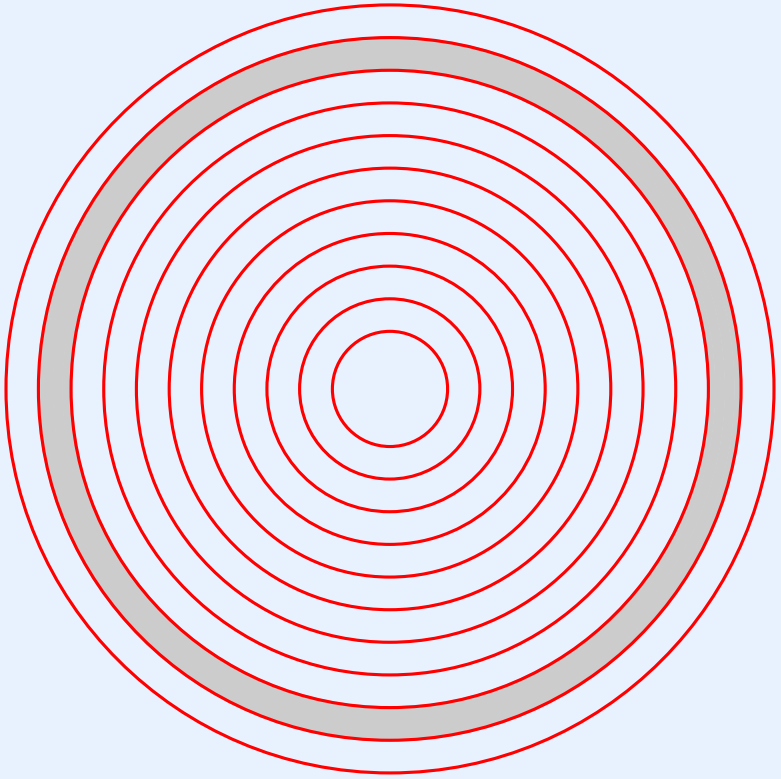
PART 12

File Systems

Location Of File Systems In The Hierarchy



Purpose Of A File System

- Manages data on permanent (nonvolatile) storage
- Allows user to manipulate named semi-permanent objects (files)
- Provides access to stored information

Aspects Of A File System

- Relatively straightforward
  - I/O to a file
- Difficult
  - Sharing
  - Caching

Sharing

- File system is shared among
  - Multiple users
  - Multiple processes/threads
- Concurrent access to multiple files is essential
- Design decisions
  - Locking granularity
  - Binding times (early or late)

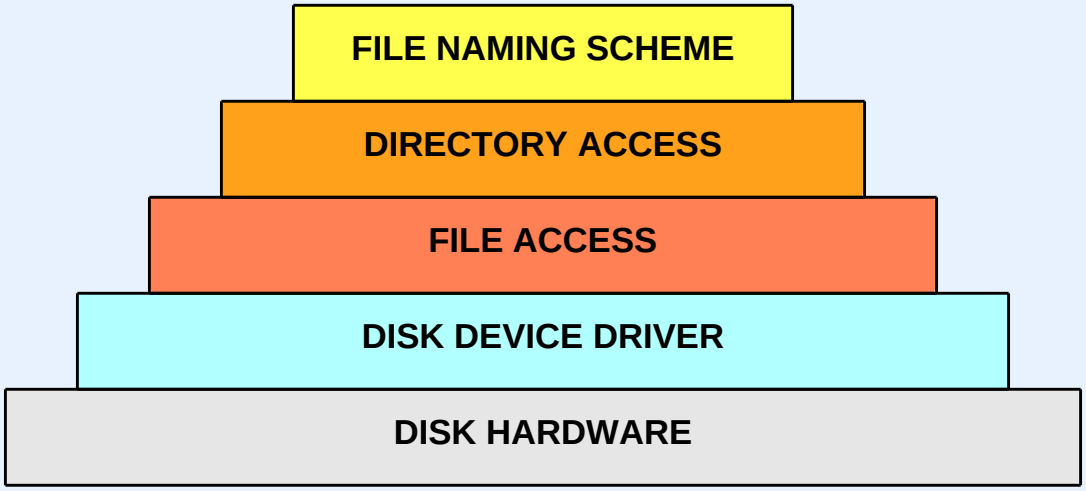
Caching

- Place items in main memory
- Items to be cached
  - Entire file contents or pieces?
  - Whole disk blocks?
  - File index blocks?
  - Directory or individual directory entries?
- Managing cached items
  - Least-recently used or least-concurrently used?
  - Exactly what can be shared?

Why Sharing Is Difficult (Unix™ Examples)

- What happens if
  - File permissions change *after* a file has been opened?
  - A file is moved to a new directory *after* it has been opened?
  - File ownership changes *after* a file has been opened?
- What happens to the file position in open files after a *fork()*?
- What happens if two processes open a file and concurrently write data
  - To different locations?
  - To the same location?

Conceptual Organization Of A File System



- Each level adds functionality
- Implementation is often integrated

Function Of Each Piece

- Naming
  - Deals with name syntax
  - May understand file location (e.g., whether file is local or remote
- Directory access
  - Maps name to (usually local) file object
  - May be completely separate from naming
- File access
  - Implements operations on files
    - Includes creation and deletion as well as reading and writing

Two Fundamental Philosophies

- Typed files (e.g., MVS)
  - System defines set of types that specify format / structure
  - User chooses type when creating file
  - Type determines operations that are allowed
- Untyped files (e.g., Unix)
  - File is a “sequence of bytes”
  - System does not understand contents, format, or structure
  - Small set of operations apply to all files

Assessment Of Typed Files

- Pros
  - Protect user from program / file mismatch
  - Allow file access mechanisms to be optimized
  - Programmer can choose file representation that is best for given need
- Cons
  - Extant types may not match new applications
  - Extremely difficult to add new file types
  - No “generic” commands (e.g., *od*)

Assessment Of Untyped Files

- Pros
  - Permit generic commands and tools
  - Separate file system design from set of applications and types of data being used
  - No need to change system when new applications arise
- Cons
  - Cannot prevent mismatch errors (e.g., *cat a.out*)
  - File system not optimal for any particular application
  - System cannot control allocation easily

Example Generic Operations

create – start a fresh file object

destroy – remove existing file

open – provide access path to file

close – remove access path

read – transfer data from file to application

write – transfer data from application to file

seek – move current {file position}

control – miscellaneous operations (e.g., change protection modes)

File Allocation

- Static allocation
  - Allocate space before use
  - Fixed file size (can be contiguous)
  - Easy to implement; difficult to use
- Dynamic allocation
  - Files grow as needed
  - Easy to use; difficult to implement
  - Potential starvation

Desired Cost Of File Operations

- Read / write
  - Sequential data transfer
  - Most common operations
  - Desired cost *0(t)*, where *t* is size of transfer
- Seek
  - Random access
  - Seldom used
  - Desired cost *0(log n)* or better, where *n* is file size

Factors To Consider

- Many files are small; few are large
- Most access is sequential; random access is uncommon
- Constants are important
- A clever data structure needed

Underlying Hardware

- Typically, a hard disk
  - Fixed-size sectors (numbered)
- Interface to disk
  - Provides random access to blocks
  - Always transfers a complete block at a time

Disk hardware cannot perform partial-block transfers.

An Example File System (Xinu)

- Underlying disk is an array of blocks
- File system has three conceptual pieces
  - Directory
  - File index
  - Data
- Simplistic approach: partition disk into three areas

Xinu File System Data Area

- Abstraction is *data block*
  - Stores file contents
  - One data block per physical disk block
  - Numbered from *0* to *N*
  - Unused data blocks linked on free list

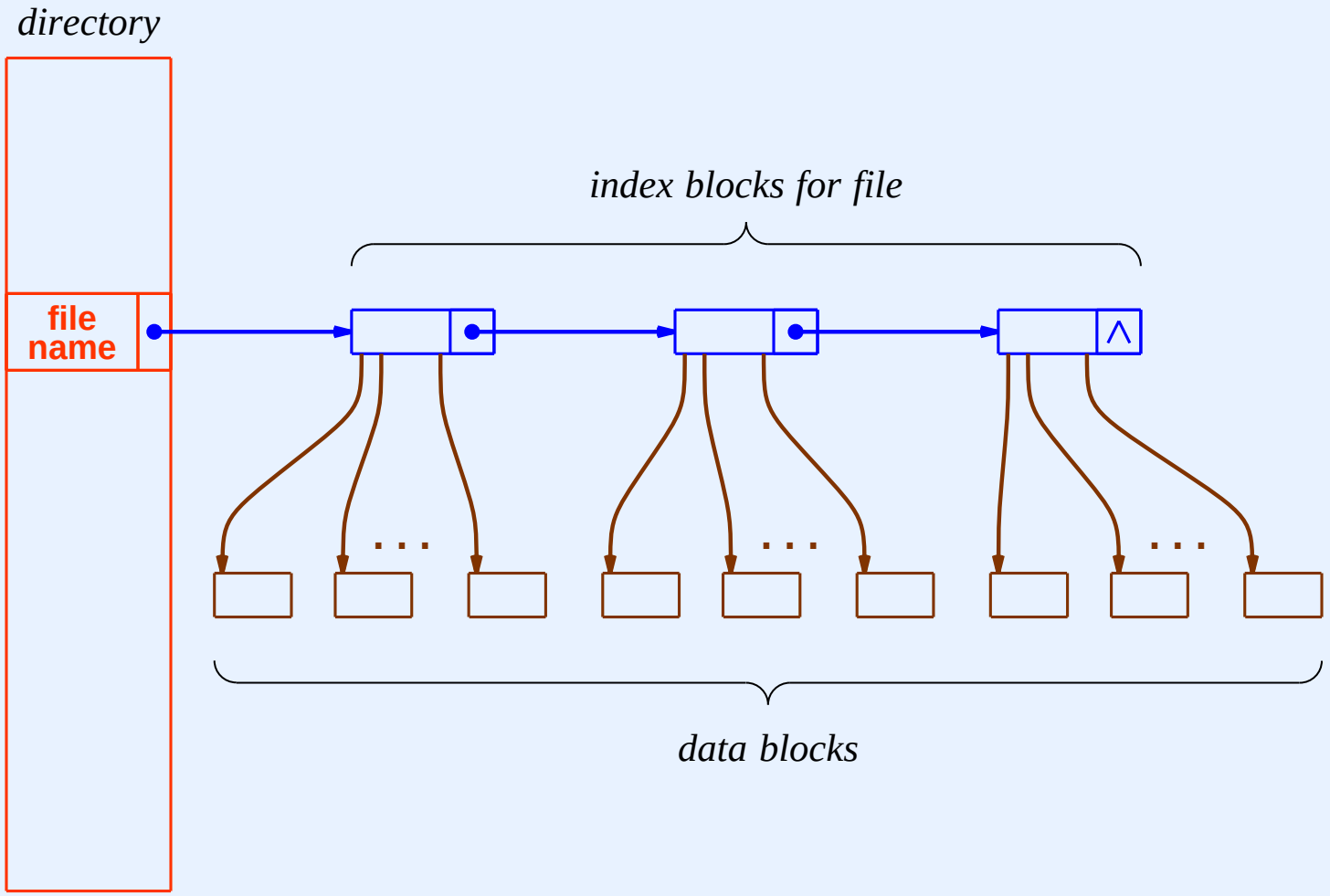
Xinu File System Index Area

- Abstraction is *index block* (*i-block*)
  - Stores
    - Pointers to data blocks
    - Offset in file of first data byte indexed
  - Multiple i-blocks per physical block
  - Numbered from *0* to *I*
  - Unused index blocks linked on free list

Xinu File System Directory

- Maps file name to index block (first index block for the file)
- Conceptually, array of pairs
  - File name
  - Number of first index block
- Kept in first disk block

Xinu File System Data Structures



Note: index and data blocks not drawn to scale

Important Concept

Within the operating system, a file is referenced by the i-block number of the first index block, not by name.

(File names are for humans.)

File Access In Xinu

- In Xinu, everything is a device
- File access paradigm
  - Set of “file pseudo devices” defined when system configured
  - Driver for pseudo device implements *read* and *write* operations
  - Application calls *open(DISK, "filename", mode)*
  - Call returns a device descriptor for one of the pseudo devices
  - When application uses *read* and *write* with the descriptor, the pseudo device driver reads or writes the underlying file

Access Paradigm

- When application opens a file
  - If directory not in memory, read a copy
  - Search directory to find i-block
  - Read i-block to find first d-block
  - Read d-block into buffer and lock
- When application reads or writes
  - If current position lies outside of d-block, move to next d-block
  - Access data at current position in d-block and update position

Simplifying Concurrent Access

- Chief design difficulty: optimizing concurrent access without allowing interference
- Complexity can arises when
  - Some processes open a file for reading
  - Other processes open the same file for writing
- To control complexity
  - Assume that a given file can only be opened once
  - Possible in Xinu because device descriptors are global, making sharing easy

Index Block Access

- Recall
  - Hardware transfers complete physical block
  - Index block is smaller than physical block
- To store index block number *i*
  - Map *i* to physical block, *p*
  - Read disk block *p*
  - Copy i-block to correct position in *p*
  - Write physical block *p*

Xinu I-Block Definition

```
/* excerpt from localfile.h */

#define IB_NULL         -1              /* null pointer to an i-block   */
#define IB_LEN          14              /* d-block ptrs per i-block     */
#define IB_AREA         1               /* first sector of i-blocks     */

/* Structure of an i-block on disk */

struct  iblk            {               /* format of index block        */
        uint32          ib_offset;      /* first data byte in file      */
                                        /*  indexed by this i-block     */
        dbid32          ib_dba[IB_LEN]; /* ptrs to data blocks indexed  */
        ibid16          ib_next;        /* address of next index block  */
        uint16          ib_padding;     /* unused: padding to 64 bytes  */
};

/* Conversion between i-block number and disk sector plus offset within */
/*      a disk sector (assumes 512-byte sector and 64-byte i-block)     */

#define ib2sect(ib)     (((ib)>>3)+IB_AREA)     /* iblock to disk block */
#define ib2disp(ib)     (((ib)&07)*sizeof(struct iblk)) /* displancment */
```

Xinu Function To Read An I-Block

```
/* excerpt from ibget.c */

/* ibget  --  get an iblock from disk given its number (assumes i-block */
/*              mutex is already held)                                   */

void    ibget(
          did32         diskdev,        /* ID of disk device            */
          ibid16        inum,           /* number of i-block to fetch   */
          struct iblk   *buffer         /* buffer to hold i-block       */
        )
{
        char    *from, *to;             /* pointers used in copying     */
        int32   i;                      /* loop index used during copy  */
        char    dbuff[LF_SECTOR_SIZ];   /* buffer to hold disk block    */

        read(diskdev, dbuff, ib2sect(inum));

        /* extract i-block from and copy to specified buffer */

        from = dbuff + ib2disp(inum);
        to = (char *)buffer;
        for (i=0 ; i<sizeof(struct iblk) ; i++)
                *to++ = *from++;
        return;
}
```

Xinu Function To Write An I-Block (part 1)

```
/* excerpt from ibput.c */

/* ibput  --  write an iblock to a specified disk given the i-block index*/

status  ibput(
          did32         diskdev,        /* ID of disk device            */
          ibid16        inum,           /* number of i-block to fetch   */
          struct iblk   *ibuff          /* buffer that holds an i-block */
        )
{
        dbid32  diskblock;              /* ID of sector on disk         */
        char    *from, *to;             /* pointers used in copying     */
        int32   i;                      /* loop index used during copy  */
        char    dbuff[LF_SECTOR_SIZ];   /* temp. buffer to hold d-block */

        /* compute disk block number and location i-block will go */

        diskblock = ib2sect(inum);
        to = dbuff + ib2disp(inum);
        from = (char *)ibuff;
```

Xinu Function To Write An I-Block (part 2)

```
        /* exclude for access, read disk block, replace the specified   */
        /* iblock, and write the disk block back to the disk            */

        wait(lfmdata.lfmmutex);
        if (read(diskdev, dbuff, diskblock) == SYSERR) {
                return SYSERR;
        }
        for (i=0 ; i<sizeof(struct iblk) ; i++)
                *to++ = *from++;
        write(diskdev, dbuff, diskblock);
        signal(lfmdata.lfmmutex);

        return OK;
}
```

Open File Table

- Internal to OS
- Records all open files
- Purpose: optimize access
  - Permission checked once (at *open*)
  - Not checked per transfer

File Descriptor

- File access abstraction
- Usually a small integer
- Returned by *open*
- Used for all file operations
- Allows “close on exit”
- Form of *capability*

Generalized Descriptors

- Concept of descriptor can be generalized
- Descriptor can refer to
  - Local file
  - Remote file
  - I/O device
  - Network socket
  - Memory region
- Single paradigm for all access

Scope Of Descriptors

- Global
  - Completely shared
- Per-Process (most common)
  - Inherited from creator
  - Shared by threads within the Process
- Per-thread
  - No sharing

Implementation Of Descriptors

- Maintained by operating system
- Internal table keeps information
- Descriptor
  - Integer index into table
  - Given to application
  - Used in I/O operations

Per-Process Descriptor Scope (UNIX)

- Descriptor table, *D*
  - One per Process
  - Each entry points to open file
  - Contains current position
- Descriptor value
  - Index into *D*
  - Meaningless outside the Process

Inheritance And Sharing

- Reference count used
  - Associated with entry in underlying table (e.g. open file table)
  - Initialized to *1* when file opened
  - Incremented when descriptor copied (e.g., *fork*)
  - Decremented when file closed
  - Entry removed when count reaches *0*

Unix File System

- Allows small or large files
- Highly tuned access
- Logarithmic overhead (asymptotic)
- Hierarchical directory from *MULTICS*
- Uses index nodes (*i-nodes*) and data blocks
- Embeds directories in files

Note: embedding directory in a file is possible because files are known by their index rather than by name

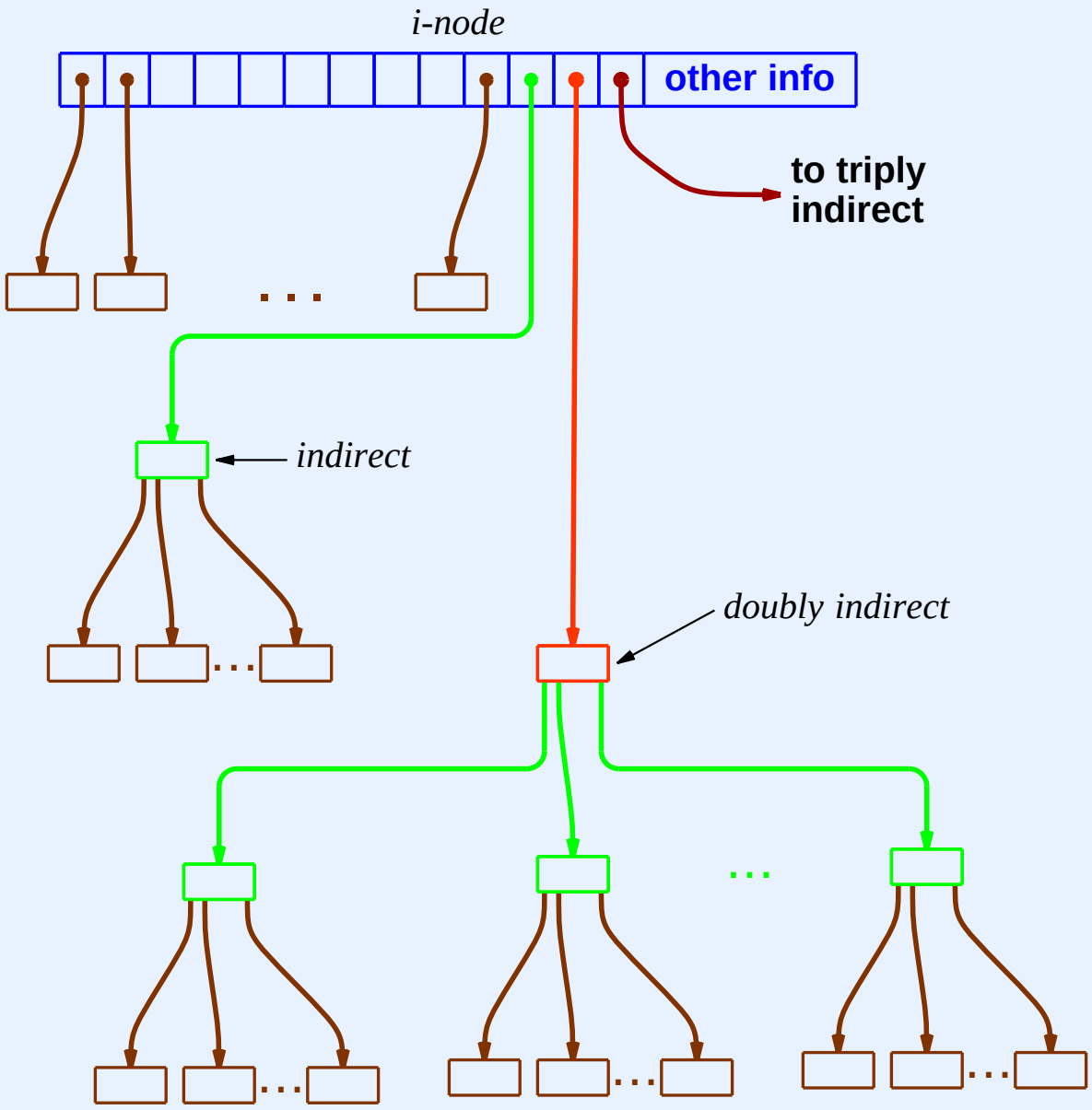
Contents Of UNIX I-Node

- File owner
- Current size
- Number of links
- Read / write / execute protection bits
- Access / create / update timestamps
- Pointers to data

13 Pointers In I-Node

- Ten *direct* pointers to data blocks
- One *indirect* pointer to block of *128* pointers to data blocks
- One *doubly indirect* pointer to block of *128* indirect pointers
- One *triply indirect* pointer to block of *128* doubly indirect pointers
- Accommodates
  - Rapid access to small files
  - Fairly rapid access to intermediate files
  - Reasonable access to large files

Illustration Of Unix I-Node



Unix File Size

- Accessible via direct pointers
  - 5,120 bytes
- Accessible via indirect pointer
  - 70,656 bytes
- Accessible via doubly indirect pointer
  - 8,459,264 bytes
- Accessible via triply indirect pointer
  - 1,082,201,088 bytes
- Note: maximum size file seemed immense when Unix was designed

Unix Hierarchical Directories

- Scheme for organizing file names
- Derived from *MULTICS*
- Hierarchy of *directories* (aka *folders*)
- Given directory can contain
  - Files
  - Subdirectories
- Top directory called *root*

Unix File Name

- Text string
- Names specific file
- Gives *path* through hierarchy
- Example
  - /u/u5/dec/junk
- Special names in each directory
  - Current directory named “**.**”
  - Parent directory named “**..**”

Unix Implementation
Hierarchical Directories

- Directory implemented as file
  - New file type (*directory*)
  - Contains triples

    (type, file name, i-node number)
- *Root directory* at i-node *2*
- Path resolved one component at a time
- General enough for arbitrary graph; restrictions added to simplify administration

Advantages Of Unix File System

- Little overhead for sequential access
- Random access to specified position
  - Fast search in short file
  - Logarithmic search in large files
- Files grow as needed
- Directories grow as needed
- Economy of mechanism because directories are embedded in files

Disadvantages Of Unix File System

- No type mechanism
- Restricted access granularity *owner*, *group*, *other*
- Single access mechanism not optimized for any particular purpose
- Data structures can be corrupted during system crash
- Integrated directory / file system not easily distributed

An Important Idea

The most difficult aspects of file system design arise from the tension between efficient concurrent access, caching, and the need to guarantee consistency on disk.

Caching, Locking Granularity, And Efficiency

- To be efficient, must cache file system data items in memory
- To guarantee mutual exclusion must lock cached copies
- What granularity of locking is best?
  - Entire directory?
  - One i-node?
  - One disk block?
- Does it make sense to lock a disk block that contains multiple i-nodes?
- Can locking at the level of disk blocks lead to a deadlock?

Caching, Locking Granularity, And Efficiency (continued)

- Cannot afford to write every change to disk immediately
- When should updates be made?
  - Periodically?
  - After a significant change?
- How can we maintain consistency on disk?
  - Must an i-node be written first?
  - When should the i-node free list be updated
  - In which order should indirect blocks be written?

File System Caching

- Essential to high-speed file access
- Technique: keep most recently used objects in memory
- Possible to have separate caches for
  - Data blocks
  - Index blocks
  - Directory blocks

Importance Of Caching

- I-node cache eliminates need to reread index
- Disk block cache keeps directories near root
- Dramatic performance improvements

Memory-Mapped Files

- Feasible with large (i.e., gigabyte) memories
- Arrange to map file into part of virtual address space
- Allow application to manipulate entire file as an array of bytes in memory
- Can use conventional paging hardware to read and write blocks of the file

Summary

- File system manages permanent storage
- Functionality
  - Naming
  - Directories
  - File access
- Files can be typed or untyped

Summary
(continued)

- File system contains files and directories
- Files implemented with index blocks that point to data blocks
- Directory can be embedded in files
- Caching essential for performance
- Memory-mapped files are feasible with large memories