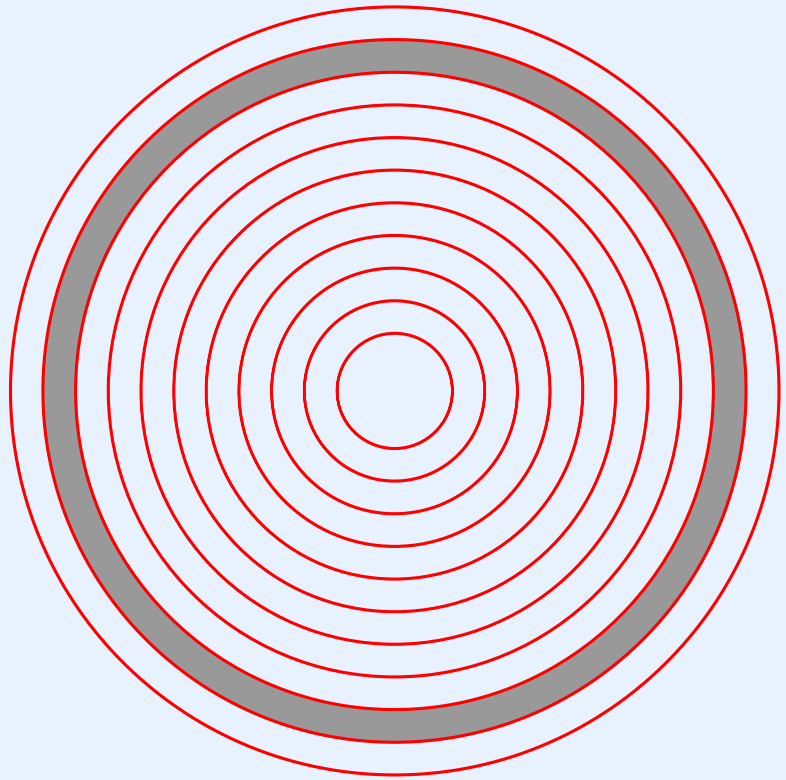
Module XII

File Systems

Location Of File Systems In The Hierarchy



Purpose Of A File System

- Manages data on (usually nonvolatile) storage
- Allows user to name and manipulate semi-permanent files
- Provides mechanisms used to organize files and store metadata

Aspects Of A File System

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

Sharing

- File system can be shared among
  - Multiple users
  - Multiple processes
- Concurrent access to multiple files is essential
- Design decisions
  - Locking granularity
  - Binding times (early or late)
  - Semantics of operations like truncation in a shared world

Caching

- Usually cache items in main memory
- Choice among 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?
  - Are items in multiple caches always coherent?

Distributed File Systems

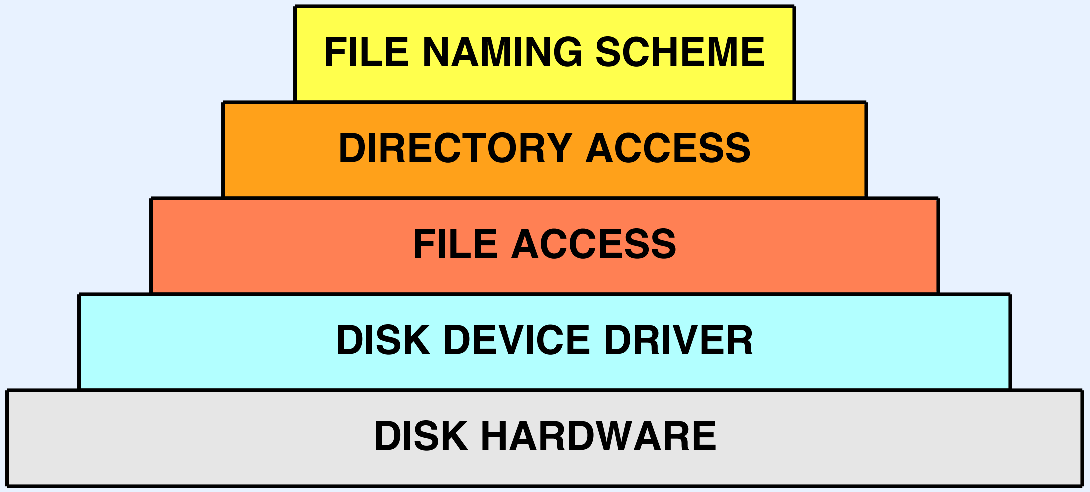
- Allows sharing among users across multiple computers
- Complexity arises from
  - Inherent delays
  - Global agreement on userids and authentication
  - Locking and caching across multiple computers

Note: we will see more later in the course

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 should happen 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 may integrate multiple levels

Function Of Each Part

- Naming
  - Deals with name syntax
  - May understand file location (e.g., whether file is local or remote)
- Directory access
  - Maps name to 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 (MVS)
  - System defines set of types that specify format / structure
  - User chooses type when creating file
  - Type determines operations that are allowed
- Untyped files (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 application / 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 Of Generic File 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 Choices

- 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 for 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

- A traditional disk
  - Fixed-size sectors (numbered)
  - Standard sector size 512 bytes
  - Some disks offer sectors of 4K bytes
- Disk interface
  - Random access using block number
  - Can only transfer a complete block

Disk hardware cannot perform partial-block transfers.

An Example File System

Xinu File System

- Views underlying disk as an array of sectors (disk blocks)
- Simplistic approach: partition disk into three areas
  - Directory area
  - File index area
  - Data area

Data Area

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

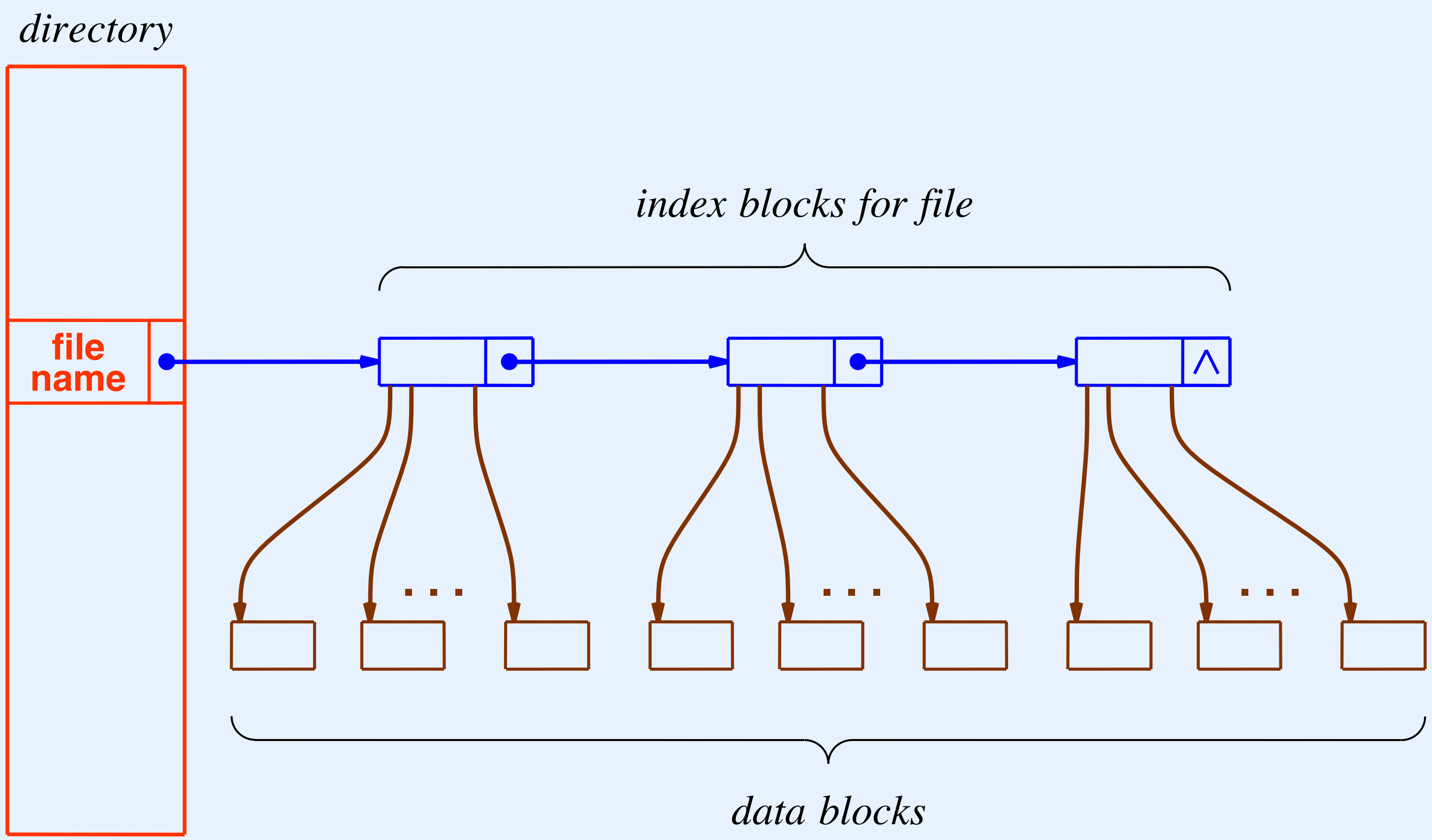
Xinu File System Index Area

- Abstraction is *index block* (*i-block*)
  - Multiple i-blocks per physical block
  - Each index block stores
    - Pointers to data blocks
    - The offset in file of first data byte indexed
  - Index blocks are numbered from *0* to *I*
  - Unused index blocks linked on free list

Xinu File System Directory

- Maps file name to index block number (first index block for the file)
- Conceptually, a directory is an array of pairs
  - File name
  - Number of first index block
- Xinu keeps the directory in the first physical disk block
  - Limited size, but sufficient for small embedded system

Xinu File System Data Structure



- Figure shows the index block list for one file
- Note: items 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
- The file access paradigm uses
  - A set of "file pseudo devices" defined when system configured
  - The driver for a pseudo device implements *read* and *write* operations
  - An application calls *open(LFILESYS, "filename", mode)*
  - The call returns a device descriptor for one of the pseudo devices
  - When an application uses *read* and *write* with the descriptor, the pseudo device driver reads or writes the underlying file

Xinu File Access Paradigm

- When application opens a file, do the following
- If the directory is not in memory, obtain a copy from disk
- Search the directory to find the i-block number for the file
- Allocate a file pseudo-device for the file
- Set the file position to byte 0
- Obtain the data block for current position
  - Read the first i-block to find first d-block ID
  - Read the first d-block into a buffer
  - Set the byte pointer to first byte in the buffer

Xinu File Access Paradigm
(continued)

- When an application reads or writes
  - If the current file position lies outside of the current d-block, move to next d-block
  - Read or write data at current position in the d-block and increment the buffer position accordingly
- Note: the file system does not fetch the “next” data block until it is needed

Concurrent Access To A Shared File

- The chief design difficulty: shared file position
- Ambiguity arises when
  - A set of processes open a file for reading
  - Other processes open the same file for writing
  - Each process issues *read* and *write* calls without specifying the file position
  - The file position depends on when processes execute
- Xinu limits concurrent access
  - Only one active open on a given file at a given time
  - A programmer must choose how to share the file among processes

Index Block Access And Disk I/O

- Recall
  - The hardware always transfers a complete physical block
  - An index block is smaller than a physical block
- To store index block number *i*
  - Map *i* to a physical block, *p*
  - Read disk block *p*
  - Copy i-block *i* to the correct position in *p*
  - Write physical block *p* back to disk
- This is the same paradigm used by Unix i-nodes (discussed later)

Questions

- What should we cache?
  - Individual i-blocks
  - The disk block in which an i-block is contained
- How can the file system be extended?
  - Use a file to store the directory
  - Allow more sharing
  - Provide better caching
  - Cross multiple cores/machines

More Questions

More Questions

- What is the major disadvantage of the layout used by the Xinu file system?
  - Hint: think about scale

More Questions

- What is the major disadvantage of the layout used by the Xinu file system?
  - Hint: think about scale
- What design should be used on a solid-state disk that has a block size of 4K bytes?

Unix File System

Unix File Access Paradigm

- Open file table
  - Internal table used by the OS to record all open files
  - Uses a reference count for concurrent access
- File descriptor
  - Integer index into a per-process descriptor table that is returned by *open*
  - Meaningless outside the process
  - Think of it as a *capability* a process uses for file operations
- Entry in the descriptor table
  - Points to an entry in the open file table
  - Maintains a current location in the file

Generalization Of Descriptors

- Unix descriptors are generalized beyond local files
- Descriptor can also refer to
  - A remote file
  - An I/O device
  - A Network socket
  - A memory region
- A single paradigm is used for all access

Inheritance, Sharing, And Reference Counts

- Recall: a reference count is kept for each entry in open file table
  - Initialized to *1* when the file is first opened
  - Incremented when a descriptor is copied (e.g., during *fork*)
  - Decremented when a file is closed
  - The entry is removed when the count reaches *0*
- Note: Unix closes all open descriptors automatically when a process exits

Unix File System

Unix File System

- Allows small or large files

Unix File System

- Allows small or large files
- Highly tuned access
- Logarithmic overhead (asymptotic)

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

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

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 inside the operating system files are known by their index rather than by name

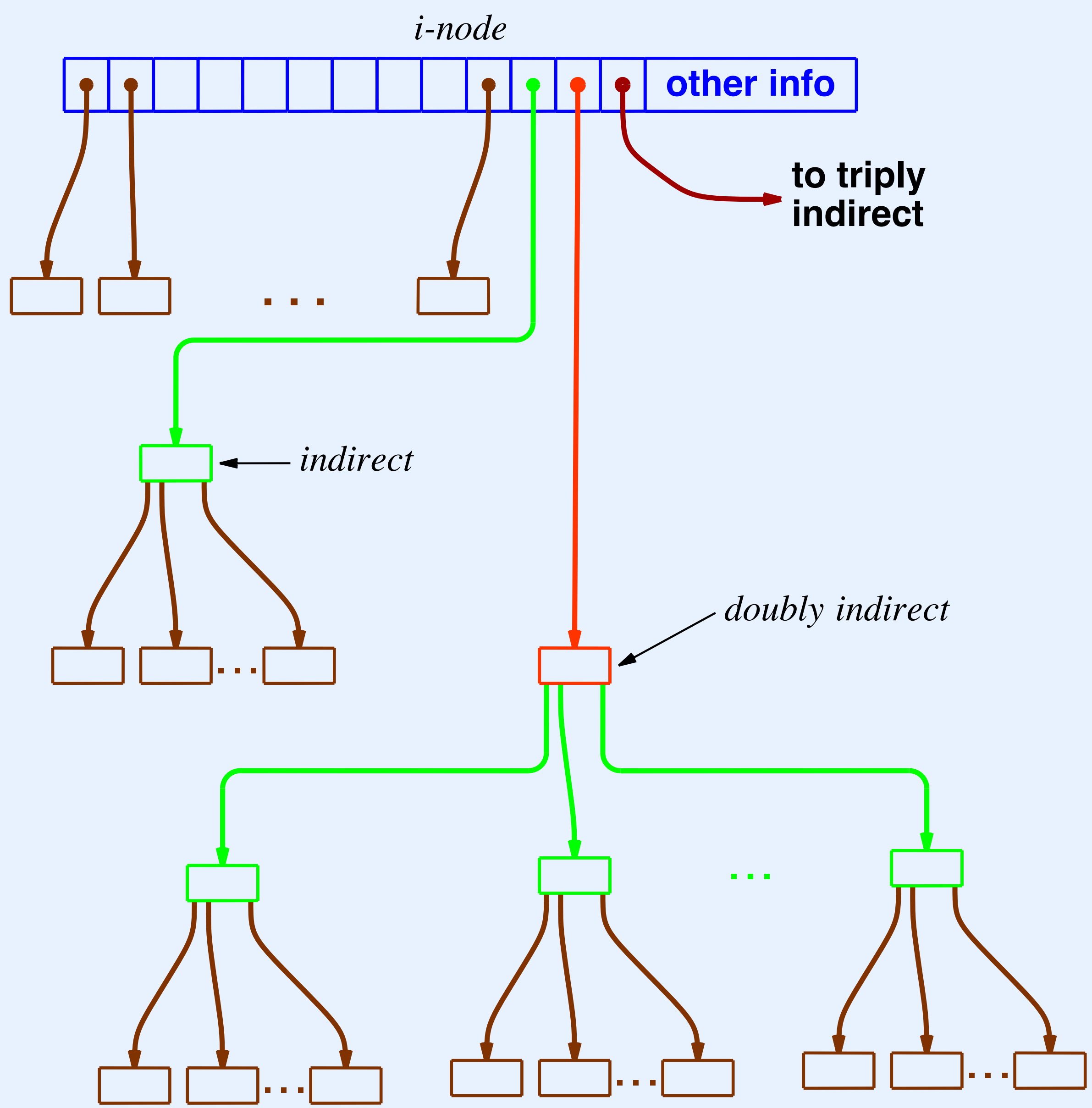
Contents Of A Unix I-node

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

The 13 Pointers In An I-node

- Ten *direct* pointers to data blocks
- One *indirect* pointer to a block of *128* pointers to data blocks
- One *doubly indirect* pointer to a block of *128* indirect pointers
- One *triply indirect* pointer to a 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 A Unix I-node



Unix File Sizes

- 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; FreeBSD increased sizes to use 64-bit pointers, making the maximum size 8ZB.

Unix Hierarchical Directories

- A scheme for organizing file names
- Derived from *MULTICS*
- Implements a hierarchy of *directories* (aka *folders*)
- A given directory can contain
  - Files
  - Subdirectories
- The top directory is called the *root*

Unix File Name

- A text string
- A name corresponds to a specific file
- The name gives a *path* through the hierarchy
- Example
  - / u / u5 / dec / junk
- Two special names are found in each directory
  - The current directory is named “**.**”
  - The parent directory is named “**. .**”

Unix Hierarchical Directory Implementation

- A directory is implemented as file
  - Uses a new file type (*directory*)
  - The directory contains triples

(type, file name, i-node number)

- The *root directory* is at i-node *2*
- A path is resolved one component at a time
- The directory system is general enough for an arbitrary graph; restrictions are 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 can grow as needed
- Directories can grow as needed
- Economy of mechanism is achieved because directories are embedded in files

Disadvantages Of Unix File System

- No type mechanism
- Access granularity restricted to three sets *owner, group, other*
- The single access mechanism not optimized for any particular purpose
- Data structures can be corrupted during system crash
- Integrated directory / file system is 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, the file system must cache data items in memory
- To guarantee mutual exclusion, cached items must be locked
- What granularity of locking is best?
  - An entire directory?
  - One i-node?
  - One physical disk block?
- Does it make sense to lock a disk block that contains i-nodes from multiple files?
- Can locking at the level of disk blocks lead to a deadlock?

Caching, Locking Granularity, And Efficiency
(continued)

- A file system cannot afford to write every change to disk immediately
- When should updates be made?
  - Periodically?
  - After a significant change?
- How can a file system 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 the most recently used file objects in memory
- It is possible to have separate caches for
  - Data blocks
  - Index blocks
  - Directory blocks

Importance Of Caching

- An i-node cache eliminates the need to reread the index
- A disk block cache keeps directories near the root (because they are searched often)
- Caching provides dramatic performance improvements

Memory-mapped Files

- Feasible with large memories (especially with a 64-bit address space)
- Idea
  - Map a file into part of the virtual address space
  - Allow applications to manipulate the entire file as an array of bytes in memory
- Can use conventional paging hardware to read and write blocks of the file
- A high-speed copy to a disk file is also possible

Summary

- A file system manages permanent storage
- The functionality includes
  - A naming mechanism
  - A directory manager
  - Individual file access
- Files can be typed or untyped

Summary
(continued)

- The example file system contains files and directories
- Files are implemented with index blocks that point to data blocks
- Directories can be embedded in files (ala Unix)
- Caching is essential for high performance
- Memory-mapped files are feasible with a large address space

Questions?