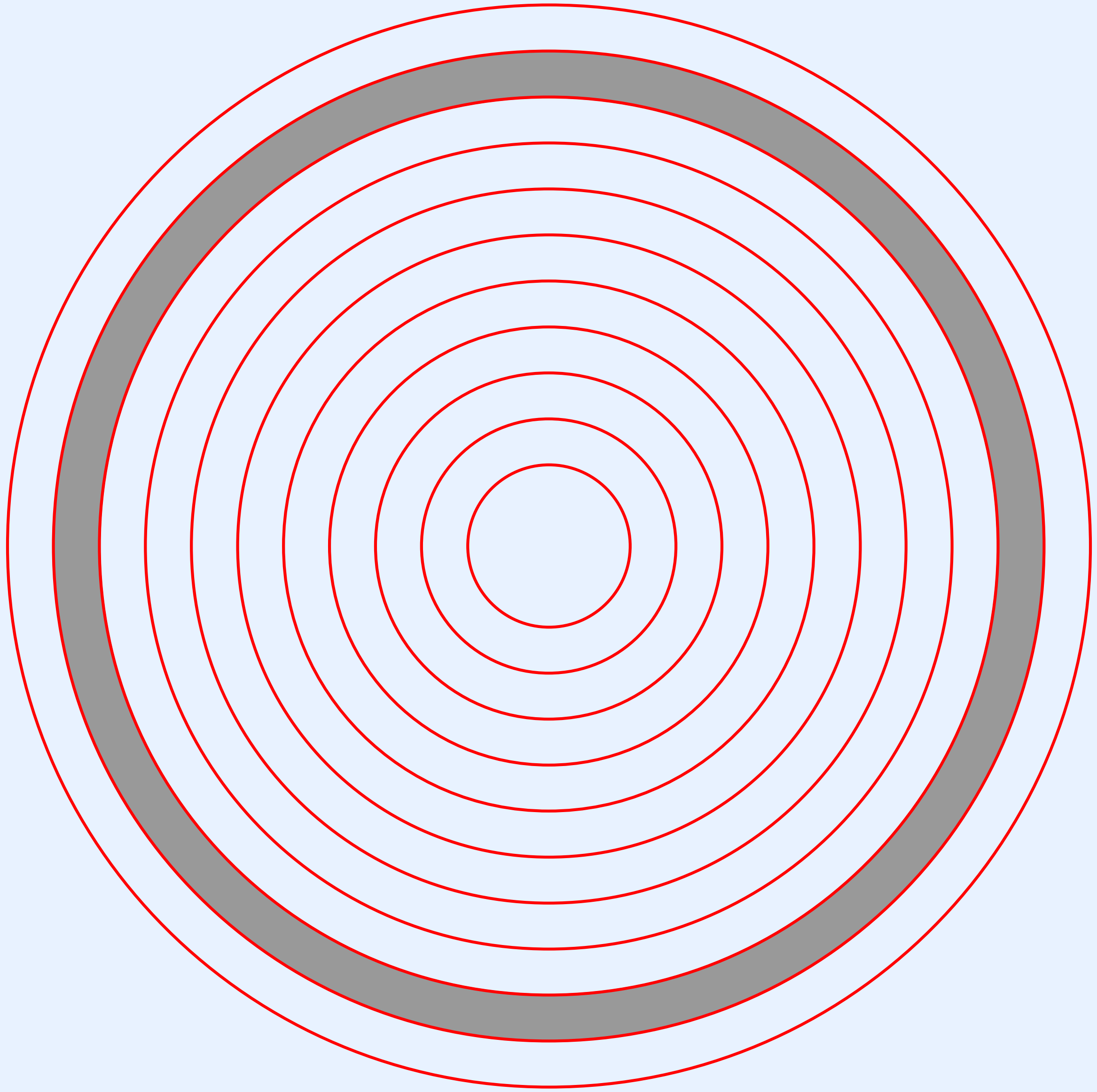
Module XII

File Systems

Location Of File Systems In The Hierarchy



Purpose Of A File System

- Manages data on nonvolatile storage
- Allows user to name and manipulate semi-permanent files
- Provides mechanisms used to organize files directories (aka folders)
- Stores metadata associated with a file
  - Size
  - Ownership
  - Access rights
  - Location on the storage system

Aspects Of A File System

- The relatively straightforward aspect
  - Allow applications to read and write data to files on local storage
- More difficult aspects
  - Control sharing on a multiuser system
  - Handle caching (important for efficiency)
  - Manage a distributed file system that allows applications on many computers to create, access, and change files

Sharing

- The most difficult aspects of file sharing revolve around the semantics of concurrent access
- An example: consider three applications that all have access to a given file
  - Application 1 opens the file, and is therefore positioned at byte 0
  - Before Application 1 reads or writes the file, Application 2 opens the file and reads 10 bytes
- At that point in time, Application 3 deletes the file
- Application 1 tries to read from the file
- What should happen?

File Sharing In A Unix System

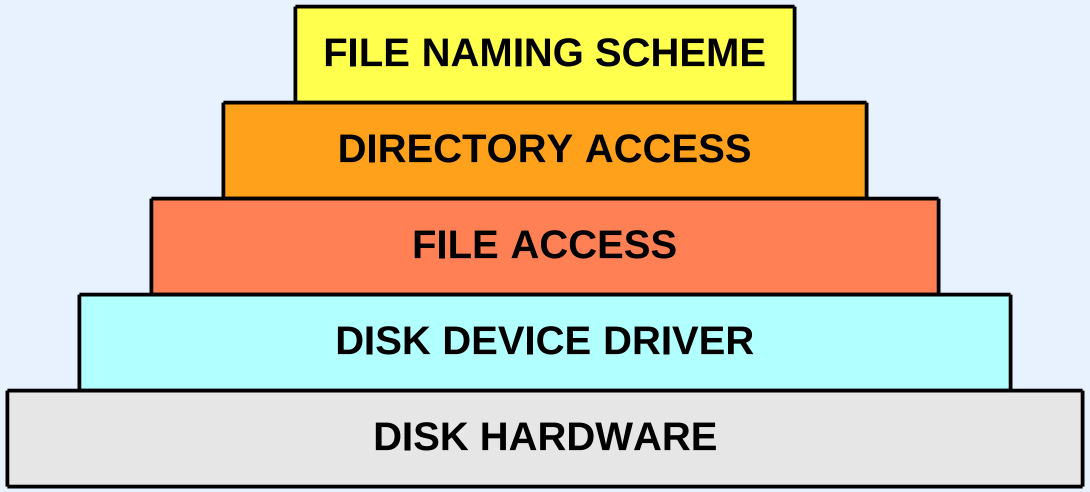
- What happens if
  - A file is deleted after it has been opened?
  - File permissions change *after* a file has been opened?
  - A file is moved to a new directory *after* it has been opened?
  - File ownership changes *after* a file has been opened?
- What happens to the file position in open files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

Sharing In A Unix System (Answers)

- Permissions are only checked when a file is opened
- Each process has its own position for a file; if two processes access the same file, chancing the position in one does not affect the position in the other
- In Unix, a file is separate from the directory entry for the file
  - Removing a file from a directory does not delete the file itself
  - When a file is removed, actual deletion is deferred until the last process that has opened the file closes it
  - Consequence: even if a file has been removed from the directory system, processes that have it open will be able to read/write it

File System Internals

The Conceptual Organization Of A File System



- Each level adds functionality
- An implementation may integrate multiple levels

The Function Of Each Level Of Software

- Naming level
  - Deals with name syntax
  - May determine the location of a file (e.g., whether file is local or remote)
- Directory access level
  - Maps a name to a file object
  - May be completely separate from naming or integrated
- File access level
  - Implements basic operations on files
  - Includes creation and deletion as well as reading and writing
- Disk driver level
  - Performs block I/O operations on a specific type of hardware

Two Fundamental Philosophies Have Been Used

- Typed files (MVS)
  - The operating system defines a set of types that specify file format / contents
  - A user chooses a type when creating file
  - The type determines operations that are allowed
- Untyped files (Unix)
  - A file is a “sequence of bytes”
  - The operating system does not understand contents, format, or structure
  - A small set of operations apply to all files

An Assessment Of Typed Files

- Pros
  - Types protect user from application / file mismatch
  - File access mechanisms can be optimized
  - A programmer can choose whichever file representation is best for a given need
- Cons
  - Extant types may not match new applications
  - It is extremely difficult to add a new file type
  - No “generic” commands can be written (e.g., *od*)

An Assessment Of Untyped Files

- Pros
  - Untyped files permit generic commands and tools to be used
  - The file system design is separate from the applications and the structure of data they use
  - There is no need to change the operating system when new applications need a different file format
- Cons
  - The operating system cannot prevent mismatch errors (e.g., *cat a.out* garbles the screen)
  - The file system may not be optimal for any particular application
  - The operating system owner does not know how files are being used

An Example Of Operations For Untyped Files

- The classic open-close-read-write interface is defined by Unix
- Conceptually, there are eight main functions

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 to a specified file position
control	– miscellaneous operations (e.g., change protection modes)

File Allocation Choices

- How should files be allocated?
- Static allocation
  - The historic approach
  - Space is allocated before the file is used
  - The file size cannot change
  - Easy to implement; difficult to use
- Dynamic allocation
  - Files grow as needed
  - Easy to use; more difficult to implement
  - Has the potential for starvation (one file takes all the space)

The Desired Cost Of File Operations

- Read / write
  - The most common operations performed
  - Provide sequential data transfer
  - The desired cost is *0(t)*, where *t* is size of transfer
- Move to an arbitrary position in the file
  - Needed for random access
  - Not often used
  - The desired cost is *0(log n)*, where *n* is file size

A Few Factors That Affect File System Design

- Many files are small; few are large
- Most access is sequential; random access is uncommon
- Overhead is important (e.g., the latency required to open a file and move to the first byte)
- Clever data structures are needed

The Underlying Hardware

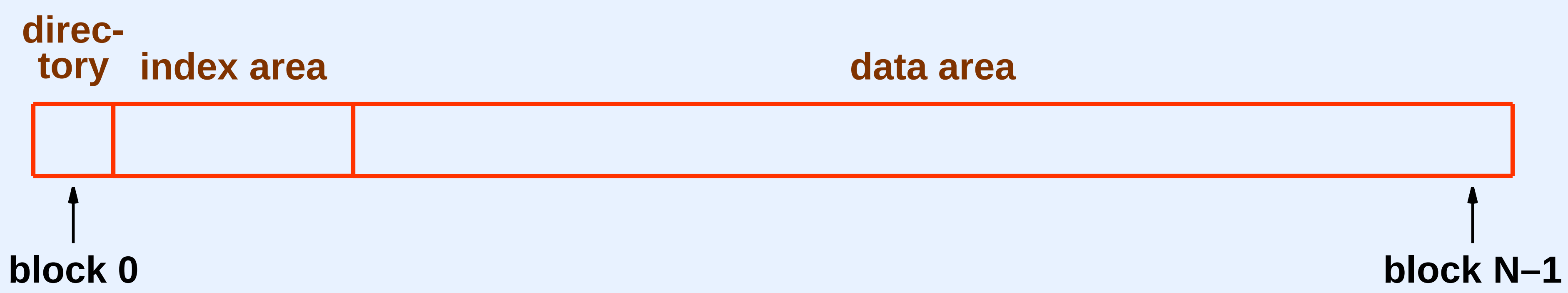
- Most files systems assume a traditional disk
  - The disk has fixed-size sectors that are numbered 0, 1, 2, ...
  - The standard sector size is 512 bytes, even for solid-state disks
- The disk interface
  - The hardware can only transfer (read or write) a complete block
  - The hardware provides random access by sector number
- An important point, especially for metadata

Disk hardware cannot perform partial-block transfers.

An Example File System

The Xinu File System

- Views the underlying disk as an array of disk blocks, where each block is 512 bytes
- Takes a simplistic approach by partitioning a disk into three areas
  - Directory area (one block)
  - File index area (a small number of blocks)
  - Data area (the rest of the disk)



The Data Area

- The file system treats the entire data area as an array of *data blocks*
  - Data blocks are numbered from *0* to $D-1$
  - Each data block is 512 bytes long, and occupies one physical disk sector
  - Blocks in the data area only store file contents
  - Currently unused data blocks are linked on a free list

The Index Area

- The file system treats the index area as an array of *index blocks* (*i-blocks*)
  - Index blocks are numbered from *0* to $I-1$
  - Because an i-block is smaller than 512 bytes, multiple blocks occupy a given disk sector
  - Each index block stores
    - Pointers to data blocks
    - The offset in file of first data byte indexed by the i-block
  - Currently unused index blocks are linked on free list

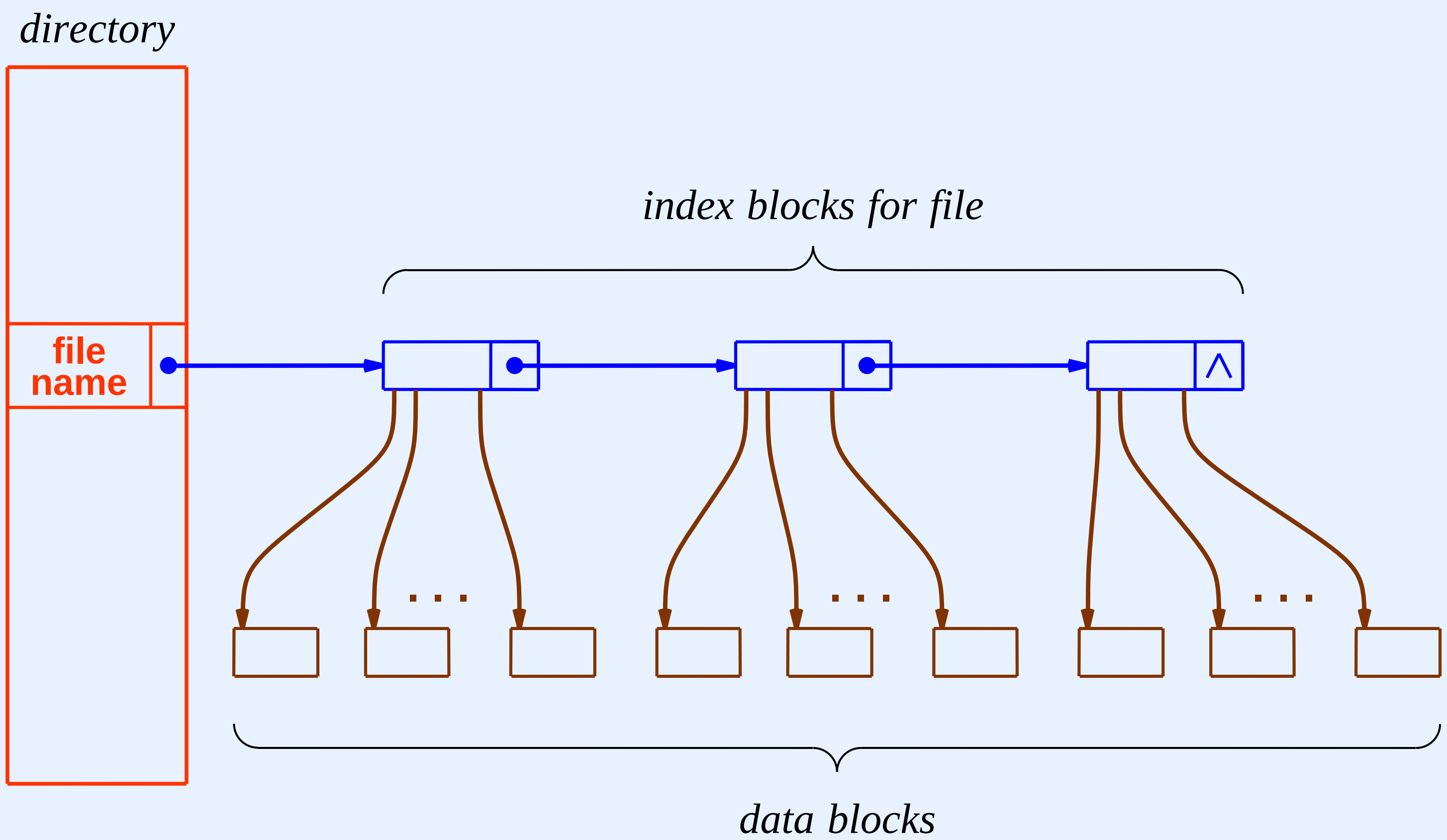
The Directory Area

- The file system treats the directory as an array of pairs:

(file name, first index block for the file)

- Conceptually, a directory entry provides a mapping from a name to the actual file
- The entire directory occupies the first physical sector on the disk
- The directory is limited, but has sufficient size for a small embedded system

The Xinu File System Data Structure



- Index blocks for a file are linked together, and each index block points to a set of data blocks
- The figure is not drawn to scale (a data block is actually larger than an index block)

Important Concept

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

(A name is only needed when opening a file.)

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
  - A single pseudo device, *LFILESYS*, is used to open files, and a set of *K* additional pseudo devices are used for data transfer
  - The device driver for a data transfer pseudo device implements *read* and *write* operations
  - The device driver for the *LFILESYS* pseudo device implements *open*

Using The Xinu Local File System

- To open a file, an application calls

  desc = open(LFILESYS, name, mode);

- The call sets *desc* to the device descriptor of one of the data transfer pseudo devices, and associates the pseudo device with the named file
- The application calls *read*, *write*, and possibly *seek*, passing *desc* as the device descriptor
- The device driver for the data transfer pseudo device performs *read* and *write* operations on the file that has been opened
- When it finishes using the file, the application calls *close*

The Xinu File Access Paradigm

- When an application opens a file, the code takes the following steps
  - Obtain a copy of the directory from disk if it is not already in memory
  - Search the directory to find the i-block number for the file
  - Allocate a data transfer pseudo-device for the application to use
  - Set the initial file position to zero
  - Obtain the data block that contains byte zero of the file
    - Read the first i-block to find first data block ID
    - Read the first data block into a buffer
    - Set the byte pointer to first byte in the buffer

The Xinu File Access Paradigm
(continued)

- When the application reads or writes data
  - If the current file position has moved outside the current data block, fetch the data block for the current position
  - Read or write data from/to the current data block buffer, incrementing the buffer position for each byte
- Note: even if all data in a buffer is consumed, the file system does not fetch the “next” data block until it is needed

The File System Pseudo-device Control Block

```
/* excerpt from lfilesys.h */

struct  lflcblk {                       /* Local file control block     */
        byte    lfstate;                /* Is entry free or used        */
        did32   lfdev;                  /* device ID of this device     */
        sid32   lfmutex;                /* Mutex for this file          */
        struct  ldentry *lfdirptr;      /* Ptr to file's entry in the   */
                                        /*  in-memory directory         */
        int32   lfmode;                 /* mode (read/write/both)       */
        uint32  lfpos;                  /* Byte position of next byte   */
                                        /*   to read or write           */
        char    lfname[LF_NAME_LEN];    /* Name of the file             */
        ibid32  lfinum;                 /* ID of current index block in */
                                        /*   lfiblock or LF_INULL       */
        struct  lfiblk  lfiblock;       /* In-mem copy of current index */
                                        /*   block                      */
        dbid32  lfdnum;                 /* Number of current data block */
                                        /*   in lfdblock or LF_DNULL    */
        char    lfdblock[LF_BLKSIZ];    /* in-mem copy of current data  */
                                        /*   block                      */
        char    *lfbyte;                /* Ptr to byte in lfdblock if   */
                                        /*  pos is inside current block */
        bool8   lfibdirty;              /* Has lfiblock changed?        */
        bool8   lfdbdirty;              /* Has lfdblock changed?        */
};
```

Example File Access: lflgetc.c (Part 1)

```
/* lflgetc.c - lfgetc */

#include <xinu.h>

/*------------------------------------------------------------------------
 * lflgetc  -  Read the next byte from an open local file
 *------------------------------------------------------------------------
 */
devcall lflgetc (
          struct dentry *devptr         /* Entry in device switch table */
        )
{
        struct  lflcblk *lfptr;         /* Ptr to open file table entry */
        struct  ldentry *ldptr;         /* Ptr to file's entry in the   */
                                        /*   in-memory directory        */
        int32   onebyte;                /* Next data byte in the file   */

        /* Obtain exclusive use of the file */

        lfptr = &lfltab[devptr->dvminor];
        wait(lfptr->lfmutex);

        /* If file is not open, return an error */

        if (lfptr->lfstate != LF_USED) {
                signal(lfptr->lfmutex);
                return SYSERR;
        }
```

Example File Access: lflgetc.c (Part 2)

```
        /* Return EOF for any attempt to read beyond the end-of-file */

        ldptr = lfptr->lfdirptr;
        if (lfptr->lfpos >= ldptr->ld_size) {
                signal(lfptr->lfmutex);
                return EOF;
        }

        /* If byte pointer is beyond the current data block, set up     */
        /*      a new data block                                        */

        if (lfptr->lfbyte >= &lfptr->lfdblock[LF_BLKSIZ]) {
                lfsetup(lfptr);
        }

        /* Extract the next byte from block, update file position, and  */
        /*      return the byte to the caller                           */

        onebyte = 0xff & *lfptr->lfbyte++;
        lfptr->lfpos++;
        signal(lfptr->lfmutex);
        return onebyte;
}
```

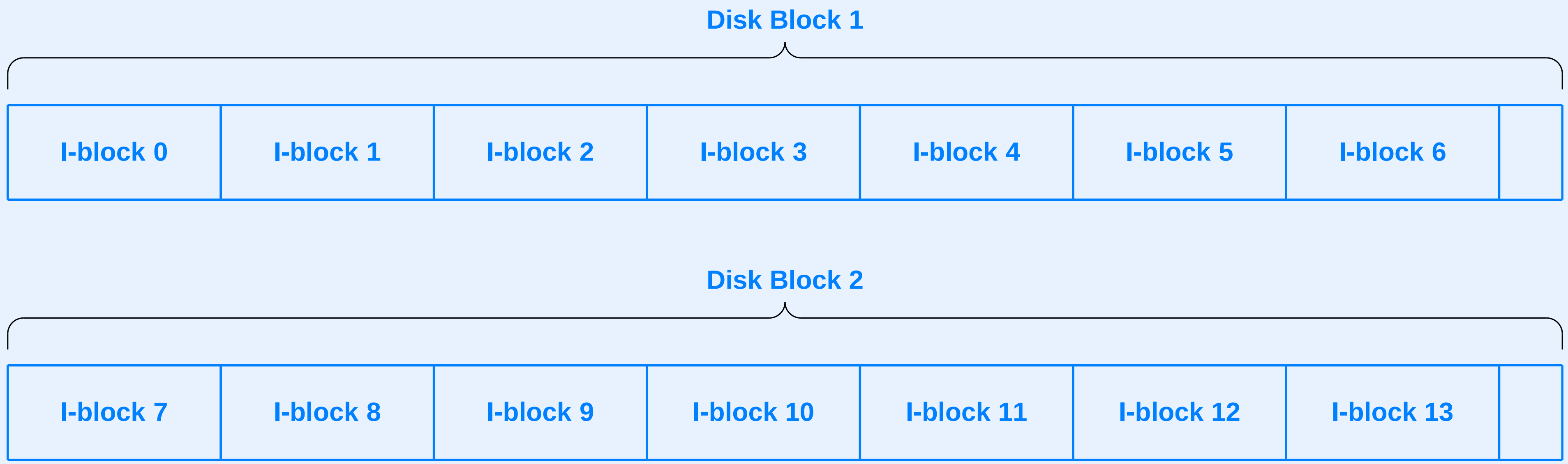
Concurrent Access To A Shared File

- The chief design difficulty: shared file position
- Ambiguity can 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 a file position
  - The file position depends on when processes execute
- To avoid the problem, Xinu prohibits concurrent access
  - Only one active open can exist on a given file at a given time
  - A programmer must choose how to share a 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
- Unix i-nodes use the same paradigm (discussed later)

Illustration Of Index Blocks In A Disk Block



- Xinu stores seven I-blocks in each disk block
- To find the disk block number in which an I-block resides, divide the I-block number by 7 (integer arithmetic) and add 1
- To find the byte position within a disk block, calculate *r*, the remainder of dividing the I-block number by 7, and multiply *r* times the size of an I-block

Xinu I-block Definition

```
/* excerpt from lfilesys.h */
#define LF_AREA_IB      1               /* First sector of i-blocks     */
#define LF_INULL        (ibid32) -1     /* Index block null pointer     */
#define LF_IBLEN        16              /* Data block ptrs per i-block  */
#define LF_IMASK        0x00001fff      /* Mask for the data indexed by */
                                        /*   one index block (i.e.,     */
                                        /*   bytes 0 through 8191).     */
#define LF_IDATA        8192            /* Bytes of data indexed by a   */
                                        /*   single index block         */
/* Structure of an index block on disk */

struct  lfiblk          {               /* Format of index block        */
        ibid32          ib_next;        /* Address of next index block  */
        uint32          ib_offset;      /* First data byte of the file  */
                                        /*  Indexed by this i-block     */
        dbid32          ib_dba[LF_IBLEN];/* Ptrs to data blocks indexed */
};
/* Conversion between index block number and disk sector number */
#define ib2sect(ib)     (((ib)/7)+LF_AREA_IB)
/* Conversion between index block number and the relative offset within */
/*      a disk sector                                                    */
#define ib2disp(ib)     (((ib)%7)*sizeof(struct lfiblk))
```

Xinu Function To Read An I-block

```
/* excerpt from lfibget.c */

/* lfibget  --  get an index block from disk given its number */

void    lfibget(
          did32         diskdev,        /* Device ID of disk to use     */
          ibid32        inum,           /* ID of index block to fetch   */
          struct lfiblk *ibuff          /* Buffer to hold index block   */
        )
{
        char    *from, *to;             /* Pointers used in copying     */
        int32   i;                      /* Loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* Buffer to hold disk block    */

        /* Read disk block that contains the specified index block */

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

        /* Copy specified index block to caller’s ibuff */

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

Xinu Function To Write An I-block (Part 1)

```
/* lfibput.c - lfibput */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  lfibput  -  Write an index block to disk given its ID (assumes
 *                      mutex is held)
 *------------------------------------------------------------------------
 */
status  lfibput(
          did32         diskdev,        /* ID of disk device            */
          ibid32        inum,           /* ID of index block to write   */
          struct lfiblk *ibuff          /* Buffer holding the index blk */
        )
{
        dbid32  diskblock;              /* ID of disk sector (block)    */
        char    *from, *to;             /* Pointers used in copying     */
        int32   i;                      /* Loop index used during copy  */
        char    dbuff[LF_BLKSIZ];       /* Temp. buffer to hold d-block */

        /* Compute disk block number and offset of index block */

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

Xinu Function To Write An I-block (Part 2)

```
        /* Read disk block */

        if (read(diskdev, dbuff, diskblock) == SYSERR) {
                return SYSERR;
        }

        /* Copy index block into place */

        for (i=0 ; i<sizeof(struct lfiblk) ; i++) {
                *to++ = *from++;
        }

        /* Write the block back to disk */

        write(diskdev, dbuff, diskblock);
        return OK;
}
```

Questions

- What should be cached?
  - Individual index blocks?
  - The disk block in which an index block is contained?
- How can the Xinu file system be extended to
  - Allow concurrent file access?
  - Use a file to store the directory?
  - Provide better caching?

The Unix File Access Paradigm

- The operating system maintains an *open file table*
  - Internal to the operating system
  - One entry for each open file
  - Uses a reference count for concurrent access
- Each process has a *file descriptor table*
  - An array where each entry points to an entry in the open file table
  - Each entry contains a position in the file for the process
- A file descriptor
  - Is a small integer returned by *open*
  - Provides an index into the process's file descriptor table
  - Is meaningless outside the process

The Generalization Of Unix File Descriptors

- Unix file descriptors provide access to mechanisms other than local files
- A descriptor can refer to
  - An I/O device
  - A network socket
  - A remote file
- The open-read-write-close paradigm is used for all descriptors

Inheritance, Sharing, And Reference Counts

- Recall: a reference count is kept for each entry in the open file table
- The reference count is initialized to *1* when a file is first opened
- When a process uses *fork* to create a new process
  - The new process gains a copy of each descriptor
  - The reference count in the open file table is incremented
- When a process calls *close*, the reference count in the open file table is decremented, and the entry in the process's file descriptor table is released for reuse
- When a reference count in open file table reaches zero, the entry is released
- Unix closes all open descriptors automatically when a process exits, so the above steps are followed whether a process explicitly closes a file or merely exits

Unix File System Properties

- The design accommodates both small and large files
- It has highly tuned access mechanisms
- The overhead is logarithmic in the size of allocated files
- It provides a hierarchical directory system (like *MULTICS*)
- The data structure uses index nodes (*i-nodes*) and data blocks
- An interesting twist: directories are actually files!

Embedding directory in a file is possible because inside the operating system, files are known by their index rather than by name

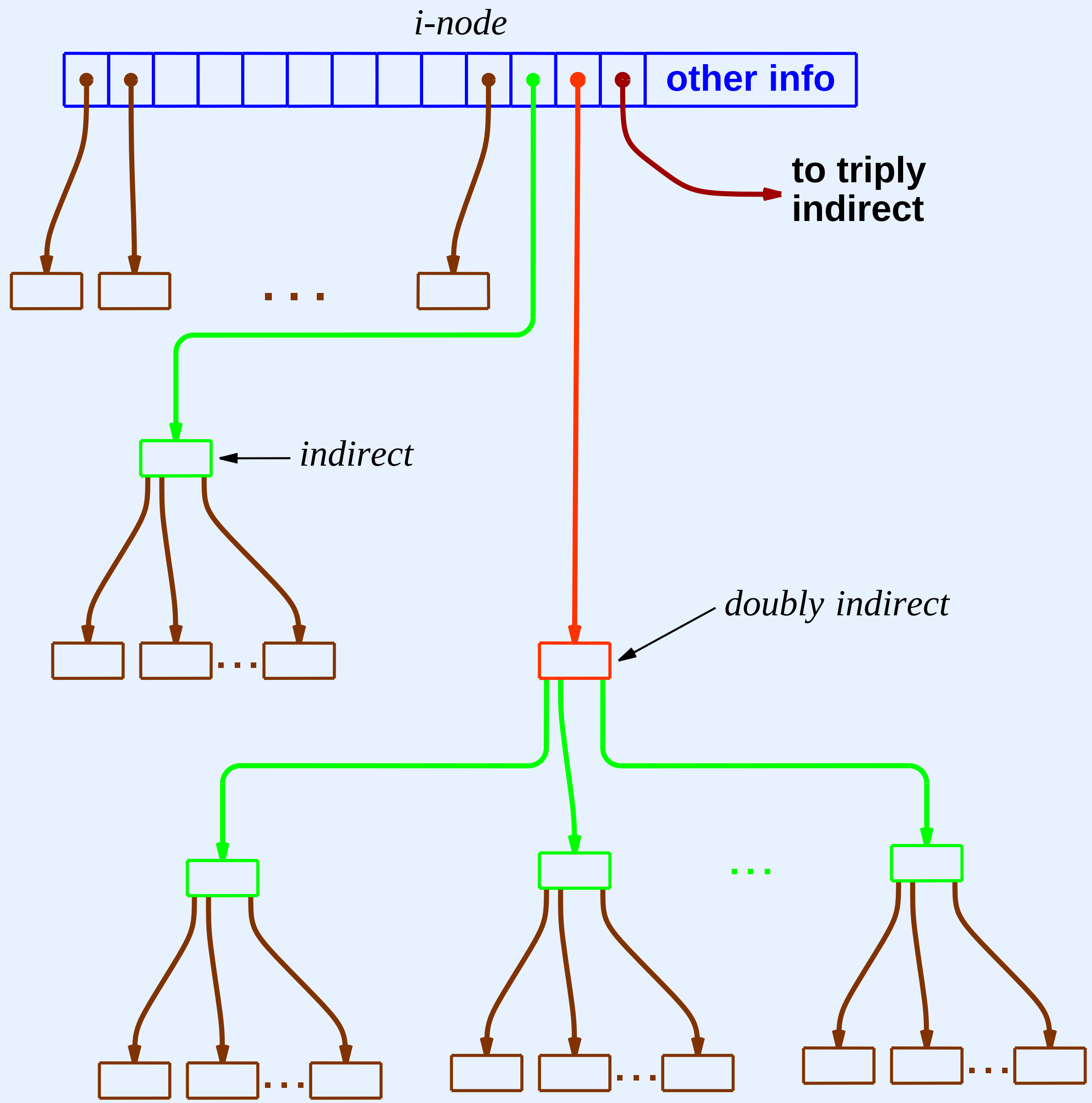
The Contents Of A Unix I-node

- The owner's user ID
- A group ID
- The current file size
- The number of links (how many directory entries point to the file)
- Permissions (i.e., read, write, and execute protection bits)
- Timestamps for creation, last access, and last update
- A set of 13 pointers that lead to the data blocks of the file

The 13 Pointers In An I-node

- Ten *direct* pointers each point to a data block
- One *indirect* pointer points to a block of *128* pointers to data blocks
- One *doubly indirect* pointer points to a block of *128* indirect pointers
- One *triply indirect* pointer points to a block of *128* doubly indirect pointers
- The scheme accommodates
  – Rapid access to small files
  – Fairly rapid access to intermediate files
  – Reasonable access to large files

Illustration of Pointers In A Unix I-node



Unix File Sizes

- The data accessible using direct pointers
  - Up to 5,120 bytes
- The data accessible via the indirect pointer
  - Up to 70,656 bytes
- The data accessible via the doubly indirect pointer
  - Up to 8,459,264 bytes
- The data accessible via the 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 Directory Mechanism

- Provides the scheme used to organize file names
- Was derived from the *MULTICS* system
- Allows a hierarchy of *directories* (aka *folders*)
- A given directory can contain
  - Files
  - Subdirectories
- The top-level directory is called the *root*

A Unix File Name

- A name is a text string
- Each name corresponds to a specific file
- The name specifies a *path* through the hierarchy
- Example
  - / u / u5 / dec / stuff
- 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 a file
  - Files that contain directories have a special file type (*directory*)
  - Each directory contains a set of triples

(type, file name, i-node number)

- The *root directory* is always at i-node *2*
- A path is resolved one component at a time, starting with i-node *2*
- The directory system is general enough for an arbitrary graph; restrictions are added to simplify administration

Advantages Of Unix File System

- Imposes very little overhead for sequential access
- Allows random access to specified position
  - Especially fast search in a short file
  - Logarithmic search in a 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

- The protections are restricted to three sets: *owner*, *group*, and *other*
- The single access mechanism may not be optimized for any particular purpose
- The data structures can be corrupted during system crash
- The integrated directory / file system is not easily distributed

Caching

- Recall that

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 Questions

- To be efficient, a file system must cache data items in memory
- To guarantee mutual exclusion, cached items must be locked
- What granularity of locking works best?
  - Should an entire directory be locked?
  - Should individual i-nodes be locked?
  - Should individual disk blocks be locked?
- 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 Questions (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 on disk?
  - In which order should indirect blocks be written to disk?

The Importance Of Caching

- An i-node cache eliminates the need to reread the index
- A disk block cache tends to keep the directories near the root in memory because they are searched often
- Caching provides dramatic performance improvements

Memory-mapped Files

- The idea
  - Map a file into part of a process's virtual address space
  - Allow the process to manipulate the entire file as an array of bytes in memory
  - Use the virtual memory paging system to fetch pages of the file from disk when they are needed
- The approach works best with a large virtual address space (e.g., a 64-bit address space)

File System Partitions

- The idea: divide a physical disk into multiple areas, and place a separate file system in each area
- Glue all partitions together by using *mount* to link all partitions into a single, unified directory hierarchy
- Motivation
  - Higher reliability: fewer files tend to be lost in a crash
  - Higher performance: keeps i-nodes closer to data blocks, which speeds up performance on an electromechanical disk
  - Lower maintenance cost: a smaller file system is much faster to check or repair

Summary

- A file system manages data on non-volatile storage
- The functionality includes
  - A naming mechanism
  - A directory system
  - Individual file access
- The Xinu file system contains files and a directory
- Files are implemented with index blocks that point to data blocks
- Unix embeds directories in files, a technique that is possible because files are identified by i-node numbers
- Caching is essential for high performance
- Memory-mapped files are feasible, especially with a large virtual address space

Questions?