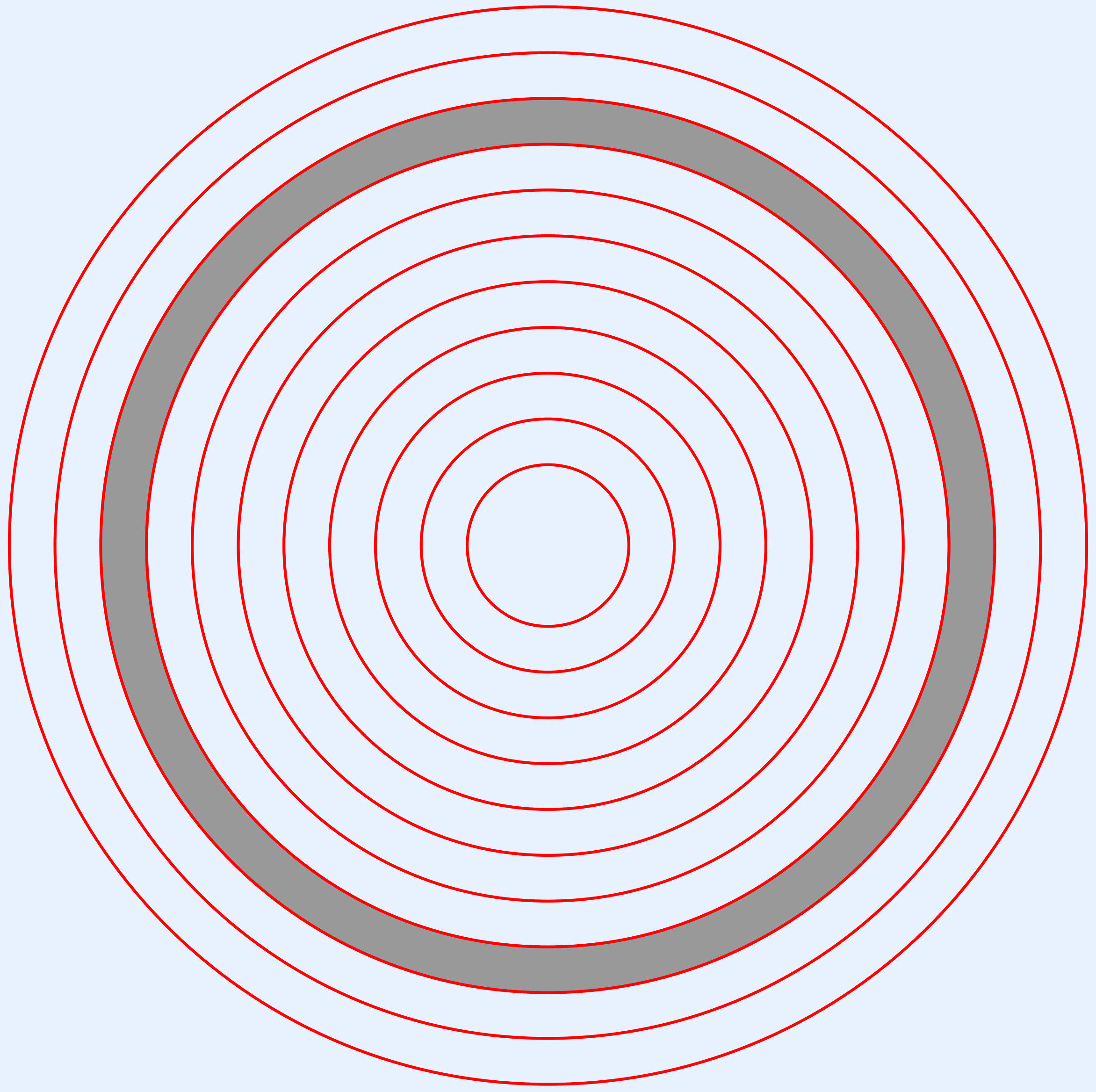
Module XI

Remote Disk And Remote File Access Mechanisms

Location Of Remote Disk And Remote File Access In The Hierarchy



Distributed Operating Systems

- Distributing OS functionality is extremely difficult
- The extent of sharing is determined by level of network communication in design hierarchy
- There have been many attempts to build a truly distributed operating system; the attempts have met with little success
- A few examples follow

Examples Of Distributed Systems

- Apollo Domain
  - The model: a shared 96-bit memory address space
  - Communication occurred at lowest level of the system
- Xerox Alto Environment
  - The model: shared files with a local process manager
  - Communication occurred between the process manager and the file system
- Unix with Internet protocols
  - The model: interconnected autonomous systems
  - The operating system supplies a communication service, but the operating system itself is not distributed

Examples Of Distributed Systems
(continued)

- Unix's Network File System (NFS)
  - The model: shared files and file names
  - Allows cross-mounting of directories
  - Builds on the Internet protocols (TCP/IP)
  - Only works among computers with identical userid values

Remote Storage

- Two aspects of distributed operating systems functionality have emerged as significant
  - Remote disk access
  - Remote file system access
- Industry uses the generic term *remote storage access* to encompass both
- Note: separation of storage from processors has become more popular with cloud computing

The History Of Remote Storage Access

- Remote disks in the 1980s
  - Diskless workstations appeared
  - A remote server provided storage
  - When disk I/O was needed, the operating system sent packets over the network to the remote disk server
- Remote file access in the 1980s
  - Remote files systems appeared
  - A remote server provided a file system
  - When an application performed file I/O, the operating system sent requests over the network to the remote file server

Which Is Better?

- Remote disk access
  - Advantage: the operating system can use whatever file system it chooses, guaranteeing that local files and remote files have the same semantics
  - Disadvantage: transferring entire disk blocks and metadata is inefficient compared to transferring data from files
- Remote file access
  - Advantage: Only file data needs to be transferred
  - Disadvantage: the server defines the naming scheme and the set of operations that are available

Remote Storage Access In The Modern World

- Remote storage access is alive and well, and is used in many data centers
- For various economic and technical reasons, data centers can choose to
  - Keep large storage facilities separate from the computational servers
  - Link server racks and storage with a high-speed network
  - Configure operating systems for remote access
- Note: data center customers who use cloud services may be unaware of the separation between servers and storage

The Cost Of Storage Access

- Bad news: I/O to storage is extremely slow
  - Even with a directly-attached solid state disk, a processor can execute thousands of instructions in the time it takes to perform one I/O operation
  - For remote accesses, the time required to access storage is even higher
- Consequences
  - Even for local storage, performance becomes unacceptable if storage I/O is not optimized
  - For remote storage, optimizations are essential or the system becomes unusable

Storage Access Pattern

- Storage accesses follow an important pattern that permit optimization:

Disk accesses are highly repetitive — once a given block of data is accessed, there is high probability that the same block will be access again and again in the near future.

- We use the term *temporal locality of reference* to describe the phenomenon, and say that disk accesses have *high temporal locality of reference*

Using Locality Of Reference

- To optimize storage I/O, an operating system makes heavy use of *caching*
- Two approaches have been used
  - The cache contains copies of disk blocks
  - The cache contains pieces of files
- The operating system
  - Allocates a set of buffers
  - Places an item in a buffer when the item is referenced
- Successive references extract data from the cached copy in the buffer without performing storage I/O

Output With A Storage Cache

- We described how a storage cache handles accesses
- A storage cache must also handle output (i.e., *write* operations)
- Consider caching disk blocks
  - When an application or the file system changes data in a disk block, the change is made to the copy in the cache
  - The cached copy is marked *dirty* to indicate that it has been modified since it was fetched from the storage system
- At some later time, dirty blocks must be written back to disk

Cache Management

- A storage cache is much more than a copy of items in memory because the operating system performs active *write-back*
- Using write-back
  - The OS maintains a request queue
  - When a disk block is changed, an entry is added to the queue of storage requests
  - In the background, the operating system continuously removes the next item from the queue and performs the request
- The result: although processes do not delay while data is written to remote storage, changes are propagated to the storage system without significant delay

Caching In A Distributed World

- A problem arises with shared storage items and multiple caches
- Example
  - Two operating systems are sharing a remote disk, *D*
  - Each operating system obtains a copy of disk block *B*
  - The applications on the two systems make changes
  - When the copies are written back to the storage server, one copy will be written first and the second copy will overwrite it
- Consequence: the changes made by one operating system will be lost

Handling Distributed Caches

- Sharing presents a *significant* problem for remote storage
- Specifically
  - On the one hand, each operating system *must* cache data or performance becomes unacceptable
  - On the other hand, unless the systems that are sharing items coordinate, the contents of their cached copies will differ
  - We say that the copies cached on the two systems can become *incoherent*
- The problem of keeping cached copies the same is known as the *cache coherence problem*

Two Examples Of Distributed OS Functionality From Xinu

- We will examine two abstractions from Xinu that deal with remote storage
  - A remote disk access system
  - A remote file access system
- Fortunately, Xinu avoids the unsolved problems associated with cache coherence by adding a restriction: a remote storage system can only be accessed by one operating system at a time

A Remote Disk System

Disk Hardware

- Conceptually, a disk appears to be an array of fixed-size *blocks*
- The de facto block size is 512 bytes (even a solid state disk provides a 512-byte block interface)
- Like an array in memory, the blocks on a disk are numbered 0, 1, 2, ...
- The hardware supports two operations
  - *Fetch* a copy of the i^{th} block into a 512-byte buffer in memory
  - *Store* data from a 512-byte buffer in memory to the i^{th} disk block
- The hardware always transfers a complete block between memory and disk

The Remote Disk Paradigm

- The idea: allow an operating system to fetch or store blocks to a remote disk
- We assume a computer that runs a *disk server* has one or more local disks attached
- The server is accessed by a *client* operating system over a network
- Each request sent to the server transfers a complete disk block
- A client can decide what to place on the disk (e.g., a file system)

A Disk Server In Practice

- A disk server is usually designed to be accessed by multiple clients
- The server can
  - Dedicate a real (i.e., physical) disk to the client
  - Use a virtualized disk
- Disk *virtualization*
  - Is popular
  - Always provides the illusion of a separate disk to each client (the client numbers blocks 0, 1, 2, ...)
  - Maps the client requests into local storage

Virtualized Disk Storage

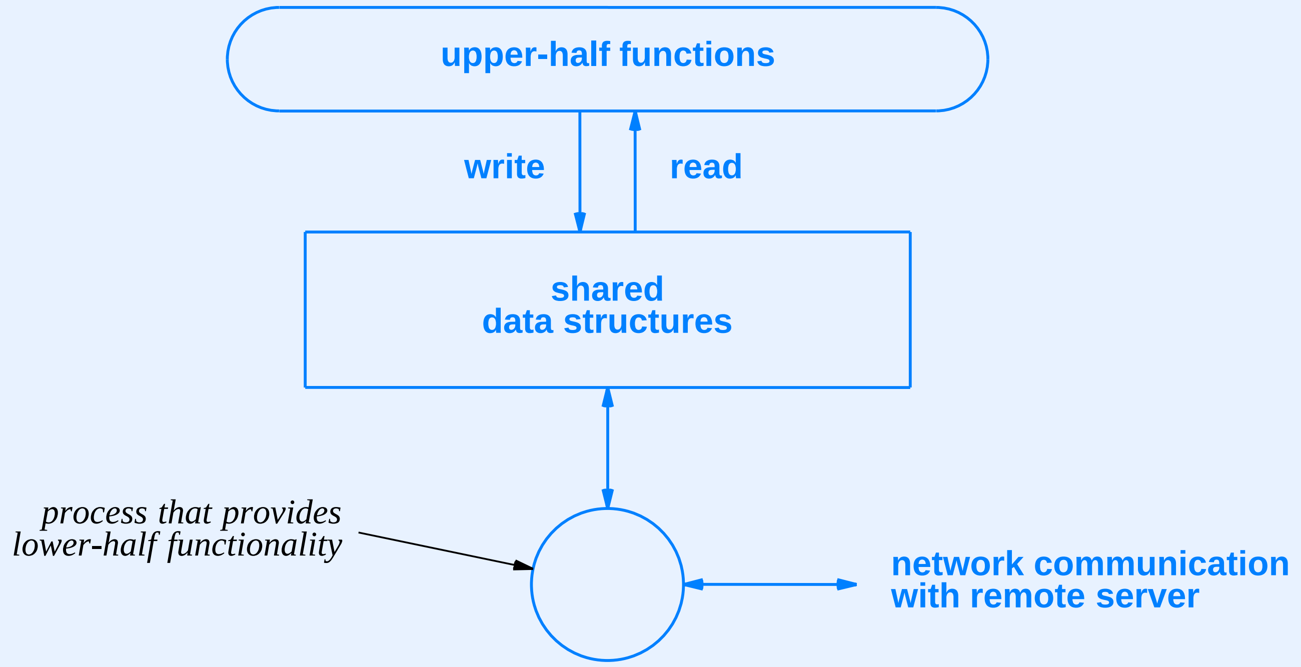
- Several mappings have been used for virtualized disks
- Partitioning
  - The server has a large physical disk
  - The disk is divided (Partitioned) into smaller regions, and each client is mapped onto one of the regions
- Files
  - The server has a local file system, and maps each client to a file
  - When a client accesses block *k*, the server accesses data in the file at byte offset $512 \times k$
- The point: in either case, a client remains competely unaware of virtualization at the server

The Xinu Remote Disk Interface

- Emulate a local disk
  - Configure a Xinu device that corresponds to the remote disk
  - Arrange driver software to support *read* and *write* operations
  - Hide all network communication
- Note: for all Xinu disks (local and remote), the “length” field of *read* and *write* calls is interpreted as a disk block number
- Example: if the remote disk has been given name *RDISK*, to read block 5 of the remote disk, a process calls *read* with 5 as the length argument:

```
read(RDISK, &buffer, 5);
```

The Structure Of Xinu Remote Disk Driver Software


upper-half functions
write
read
shared data structures
process that provides lower-half functionality
network communication with remote server

- A dedicated process handles all communication with the remote server
- The shared data structures include
  - A cache of recently-accessed disk blocks
  - A queue of pending requests

Disk Semantics

- Our remote disk emulates a local disk and offers the same semantics
- Input is synchronous: a *read* blocks the calling process until the data has been fetched from disk
- For output, last-write semantics are enforced: *read* from block *i* always returns the data most recently written to block *i*
- Optimizations that are used to increase performance make enforcement of last-write semantics more difficult
  - Caching
  - The queue of pending requests

The Queue Of Requests

- Operates exactly like the request queue used by a local disk driver
- Each item in the queue specifies
  - A block number
  - An operation (*read* or *write*)
  - A pointer to a buffer that contains data to be written (for *write*) or to be filled (for *read*)
- Items in a request queue are ordered
  - By block number for electro-mechanical disks
  - FIFO for solid-state disks
- Items in cache use same data structure with a block number and a pointer to a buffer

The Structure of A Node On Request List Or Cache

```
/* excerpt from rdisksys.h */

/* Definition of a buffer with a header that allows the same node to be */
/*  used as a request on the request queue, an item in the cache, or a  */
/*  node on the free list of buffers                                    */

struct  rdbuff  {                       /* Request list node            */
        struct  rdbuff  *rd_next;       /* Ptr to next node on a list   */
        struct  rdbuff  *rd_prev;       /* Ptr to prev node on a list   */
        int32   rd_op;                  /* Operation - read/write/sync  */
        int32   rd_refcnt;              /* Reference count of processes */
                                        /*   reading the block          */
        uint32  rd_blknum;              /* Block number of this block   */
        int32   rd_status;              /* Is buffer currently valid?   */
        pid32   rd_pid;                 /* Process that initiated a     */
                                        /*   read request for the block */
        char    rd_block[RD_BLKSIZ];    /* Space to hold one disk block */
};
```

- Notice the reference count

Reference Counts And Requests

- We expect the remote disk facility to be used by a file system
- A file system
  - Stores metadata for files in disk blocks
  - May place metadata for multiple files in the same disk block
- Consequence: when multiple applications access files, multiple processes may issue a request to read a given disk block concurrently
- The consequence for a disk driver: a block must be left in cache until all processes that need the block have used it
- The reference count in struct *rdbuff* is used to prevent the buffer from being reclaimed too early

A Sync Operation

- We said that disk hardware only supports *fetch* and *store* operations
- However, three basic operations can appear on the request queue
  - *Read* (fetch a block from disk)
  - *Write* (store a block to disk)
  - *Sync* (synchronize requests)
- The *sync* operation is
  - Used by a file system to flush data to the disk
  - Especially important for metadata (e.g., to ensure that index blocks are stored permanently)
  - Invoked with a *control* function

Performing A Sync Operation

- The idea: block the calling process until all previously-changed blocks have been written back to disk
- The implementation is straightforward
  - A *sync* request is added to the end of the request queue
  - The calling process is blocked
  - When the *sync* request reaches head of queue, the calling process is made ready again
- Note: *sync* is handled locally — no message is sent to the remote server and no data is involved

Messages Exchanged With The Remote Disk Server

- The remote disk system in Xinu uses five message types when communicating between the local system and the remote disk server

Open	– Prepare the remote disk for use and specify a name
Close	– Discontinue use of the remote disk
Read	– Read a block from the remote disk
Write	– Write a block to the remote disk
Delete	– Remove the entire remote disk from the server

Names For Remote Disks

- The Xinu remote disk server
  - Maintains a set of virtual disks
  - Retains disk contents across server reboots
  - Can handle requests from multiple clients
- To prevent interference, each disk is given a unique name
- A disk name must be passed to the server in each request
- Examples
  - Students in a class each use their userid as a unique disk name
  - The IP address of a computer (converted to a text string) can be used as name

Message Formats

- The remote disk software in an operating system and the server software must agree on the format of messages and values used in the messages
- One possible approach
  - Write the definitions in a document
  - Have engineers who build pieces of the software follow the document
- A better approach
  - Place the definitions in an include (.h) file, and use the same file in both client and server software
  - Instead of defining individual hex values for each possible request and response, define a “response” bit and use it in the definition of message types
- Xinu uses the latter approach

Message Formats
(continued)

- The definitions include the message format for each possible message
  - Open request and reply
  - Read request and reply
  - Write request and reply
- Note that the format of a reply often differs from the format of a request
- Example
  - A *read request* merely specifies a block number
  - A *read reply* contains actual data in addition to the block number

Declarations For Message Types

```
/* excerpt from rdisksys.h */

/* Values for the type field in messages */

#define RD_MSG_RESPONSE 0x0100          /* Bit that indicates response  */

#define RD_MSG_RREQ     0x0010          /* Read request and response    */
#define RD_MSG_RRES     (RD_MSG_RREQ | RD_MSG_RESPONSE)

#define RD_MSG_WREQ     0x0020          /* Write request and response   */
#define RD_MSG_WRES     (RD_MSG_WREQ | RD_MSG_RESPONSE)

#define RD_MSG_OREQ     0x0030          /* Open request and response    */
#define RD_MSG_ORES     (RD_MSG_OREQ | RD_MSG_RESPONSE)

#define RD_MSG_CREQ     0x0040          /* Close request and response   */
#define RD_MSG_CRES     (RD_MSG_CREQ | RD_MSG_RESPONSE)

#define RD_MSG_DREQ     0x0050          /* Delete request and response  */
#define RD_MSG_DRES     (RD_MSG_DREQ | RD_MSG_RESPONSE)

#define RD_MIN_REQ      RD_MSG_RREQ     /* Minimum request type         */
#define RD_MAX_REQ      RD_MSG_DREQ     /* Maximum request type         */
```

Message Formats (Part 1)

```
/* excerpt from rdisksys.h */

/* Message header fields present in each message */

#define RD_MSG_HDR                          /* Common message fields        */\
        uint16  rd_type;                    /* Message type                 */\
        uint16  rd_status;                  /* 0 in req, status in response */\
        uint32  rd_seq;                     /* Message sequence number      */\
        char    rd_id[RD_IDLEN];            /* Null-terminated disk ID      */

/************************************************************************/
/*                              Open                                    */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_oreq     {                   /* Remote file open request     */
        RD_MSG_HDR                          /* Header fields                */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_ores     {                   /* Remote file open response    */
        RD_MSG_HDR                          /* Header fields                */
};
#pragma pack()
```

Message Formats (Part 2)
(continued)

```
/* excerpt from rdisksys.h */

/************************************************************************/
/*                              Read                                    */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_rreq     {               /* Remote file read request     */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number to read         */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_rres     {               /* Remote file read reply       */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number that was read   */
        char    rd_data[RD_BLKSIZ];     /* Array containing one block   */
};
#pragma pack()
```

Message Formats (Part 3)
(continued)

```
/* excerpt from rdisksys.h */

/************************************************************************/
/*                              Write                                   */
/************************************************************************/
#pragma pack(2)
struct  rd_msg_wreq     {               /* Remote file write request    */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number to write        */
        char    rd_data[RD_BLKSIZ];     /* Array containing one block   */
};
#pragma pack()

#pragma pack(2)
struct  rd_msg_wres     {               /* Remote file write response   */
        RD_MSG_HDR                      /* Header fields                */
        uint32  rd_blk;                 /* Block number that was written*/
};
#pragma pack()
```

A Review Of Disk Semantics And The Remote Disk Implementation

- The remote disk implements the same semantics as a local disk
- *Write*
  - The upper-half software allocates a buffer, copies the user’s data into the buffer, and links the buffer into the request queue as a *write* request
  - The application continues executing
- *Read*
  - The upper-half software generates a request and deposits it in the request queue
  - The calling application is blocked to wait until the request has been satisfied
  - Later, when the data actually arrives, the upper-half copies the data into the application buffer and then allows the application to run again
- Note: a conventional disk driver uses an interrupt to move to the next request

The Structure Of The Lower-half Process

- The process, which is named *rdsprocess*, repeatedly
  - Extracts next item from request queue
  - Builds a message for the server
  - Calls function *rdscomm* to send message to the server and wait for a reply
- Function *rdscomm*
  - Performs all communication with remote server
  - Adds a sequence number to each outgoing message
  - Handles the details of timeout and retransmission, if needed
- Using a separate function for communication allows extra messages to be sent that are not in the request queue (e.g., a *control* message to delete a disk)

Caching Blocks

- Remote communication is extremely expensive
- We know that the disk access pattern means repeated access to a given block
- Conclusion: caching is an essential optimization for a remote disk
- An item is kept in cache until the buffer it occupies is needed to satisfy another request
- Note: most disk drivers (including drivers for local disks) apply the same approach of leaving an item in the cache as long as possible

Satisfying Requests

- The presence of a request queue and cache means that when an application makes a *read* or *write* request, it may be possible to handle the request from data present in memory
- *Write*
  - If the request queue already contains a *write* operation for the same block, use the new data to replace the contents of the buffer that exists on the request queue and return to the caller
  - If the block specified in the write operation is already present in the cache, the data in the cached copy is now stale, so unlink the block from the cache
  - Form a *write* request and add it to the request queue

Satisfying Requests
(continued)

- *Read*
  - If the specified block is already present in the cache, copy the contents to the caller's buffer and return
  - If request list contains *write* request for the specified block, copy the contents to the caller's buffer and return
  - Form a *read* request, deposit on the request queue, and block the current process until an answer arrives. Also check the cache, and if it contains an item for same block, unlink it from cache
- Note: when a *read* completes, *rdsprocess* moves the node from the request list to the cache and unblocks the process that is waiting

A Remote File System

Remote File Access

- A remote file *server* runs on a computer that has a file system
- A *client* operating system accesses the remote file system by sending requests to the server and obtaining replies
- A client must be able to perform a set of file operations
  – Read data from a file
  – Write data to a file
  – Create, delete, or rename a file
  – Change the file’s ownership or privileges

The Xinu Remote File Paradigm

- In principle, a server can be accessed by multiple clients
- In practice, coordination of requests from multiple clients requires considerable communication overhead and complex software
- Solution: restrict access to a single client (or have the programmer arrange to coordinate among multiple clients so that only one client makes changes to a file at any time)

The Xinu Remote File System Interface

- Uses the Xinu device mechanism
- A set of “pseudo devices” are defined that can be used to interact with files
- A master pseudo device (*RFILESYS*) corresponds to the file system itself
  - It can be used to *open* a file
  - It can also be used for *control* operations, such as file deletion
- A set of *N* pseudo-devices are defined for individual file access
  - An *open* of *RFILESYS* returns the descriptor of one of file pseudo devices
  - The application calls *read* and *write* on the descriptor
  - When finished, the application *close*s the descriptor

The Structure Of The Remote File System Code

- The structure differs from the one used for remote disk access
- The software differs because file access uses a synchronous approach
  - Each operation causes a request-response exchange with the remote server
  - There is no request queue, so there is no need for a communication process
  - Only one request can be outstanding at a time
  - There is no need for a cache
- Each upper-half function
  - Forms a request message
  - Calls *rfscomm* to send a request to server and obtain a response
  - Waits for a response, and returns to its caller

The Cost Of Remote File Operations

- The synchronous approach has a downside: high latency
- Except for *seek*, each upper-half function performs an exchange with server
- Sending a request over a network and obtaining a response introduces significant latency
- The overhead is highest when only a small amount of data is transferred
- Consequence: programmers are advised that using *putc* or *getc* to access a remote file is extremely expensive and is discouraged

The Question Of File System Semantics

- The Xinu remote file server runs on a Unix system (Linux), and has
  - Hierarchical directories
  - File modes and timestamps
  - Hard and symbolic links
- Further, Xinu defines an "o" mode used when opening a file (the file must exist), but Linux does not have any equivalent
- There are two possibilities
  - Arrange the remote file server to emulate (when possible) the Xinu file semantics
  - Provide a way for applications running on Xinu to use the Linux file system

Our Design

- Our remote file server uses Xinu semantics when feasible (and efficient)
  - Example: to simulate Xinu "o" and "n" modes, the server checks whether the file exists before opening it
- The system provides Xinu applications with access to additional Linux file system functionality via the *control* function, allowing a program to
  - Create or remove a directory
  - Truncate a file
  - Obtain the file current size (which allows a Xinu application to *append* to a file)

Operations For The Xinu Remote File System

- *Open* – open a file
- *Close* – terminate use of file
- *Read* or *getc* – obtain data from a file
- *Write* or *putc* – deposit data in a file
- *Size* – obtain the current file size
- *Delete* – remove a file
- *Truncate* – discard any existing contents
- *Mkdir* – make a directory
- *Rmdir* – remove a directory
- *Seek* – move to specified position (handled locally; no message sent to server)

File Position Information

- Operations like *read* and *write* assume the file system maintains a current position for each open file
- For example, when an application calls *read*, the application receives byes starting at the current file position (and the position is updated)
- Question: should the file position be maintained at the server or the client?
- Note the location where the position is stored affects sharing
  - If the position is kept at the server and multiple clients share an open file, the position changes whenever any of the clients *read* or *write*
  - Keeping the position information at the client allows multiple clients to each maintain their own file position (and works as long as they coordinate to avoid overwriting parts of the file that others are reading)

Xinu File Position And Seek

- Xinu stores the position at the client
  - Every request sent to the server specifies a position explicitly
  - If multiple clients access a file simultaneously, they do not interfere with each other’s position information
- The *Seek* operation allows an application to move to a specific byte offset within the file
- The Xinu design means *seek* is extremely efficient because the operation can be performed locally; no exchange with the server is needed

Definitions For The Remote File System (Part 1)

```
/* rfilesys.h - definitions for remote file system pseudo-devices */

/* Constants for the remote file system */

#define RF_NAMLEN       128             /* Maximum length of file name  */
#define RF_DATALEN      1024            /* Maximum data in read or write*/
#define RF_MODE_R       F_MODE_R        /* Bit to grant read access     */
#define RF_MODE_W       F_MODE_W        /* Bit to grant write access    */
#define RF_MODE_RW      F_MODE_RW       /* Mask for read and write bits */
#define RF_MODE_N       F_MODE_N        /* Bit for "new" mode           */
#define RF_MODE_O       F_MODE_O        /* Bit for "old" mode           */
#define RF_MODE_NO      F_MODE_NO       /* Mask for "n" and "o" bits    */

/* Global data for the remote server */

#ifndef RF_SERVER_IP
#define RF_SERVER_IP    "255.255.255.255"
#endif

#ifndef RF_SERVER_PORT
#define RF_SERVER_PORT  33123
#endif

#ifndef RF_LOC_PORT
#define RF_LOC_PORT     33123
#endif
```

Definitions For The Remote File System (Part 2)

```
/* Global data for remote file server access */

struct  rfdata  {
        int32   rf_seq;                 /* Next sequence number to use  */
        uint32  rf_ser_ip;              /* Server IP address            */
        uint16  rf_ser_port;            /* Server UDP port              */
        uint16  rf_loc_port;            /* Local (client) UPD port      */
        sid32   rf_mutex;               /* Mutual exclusion for access  */
        bool8   rf_registered;          /* Has UDP port been registered?*/
};

extern  struct  rfdata  Rf_data;

/* Definition of the control block for a remote file pseudo-device      */

#define RF_FREE 0                       /* Entry is currently unused    */
#define RF_USED 1                       /* Entry is currently in use    */

struct  rflcblk {
        int32   rfstate;                /* Entry is free or used        */
        int32   rfdev;                  /* Device number of this dev.   */
        char    rfname[RF_NAMLEN];      /* Name of the file             */
        uint32  rfpos;                  /* Current file position        */
        uint32  rfmode;                 /* Mode: read access, write     */
                                        /*      access or both          */
};

extern  struct  rflcblk rfltab[];       /* Remote file control blocks   */
```

Definitions For The Remote File System (Part 3)

```
/* Definitions of parameters used when accessing a remote server        */

#define RF_RETRIES      3               /* time to retry sending a msg  */
#define RF_TIMEOUT      1000            /* wait one second for a reply  */

/* Control functions for a remote file pseudo device */

#define RFS_CTL_DEL     F_CTL_DEL       /* Delete a file                */
#define RFS_CTL_TRUNC   F_CTL_TRUNC     /* Truncate a file              */
#define RFS_CTL_MKDIR   F_CTL_MKDIR     /* make a directory             */
#define RFS_CTL_RMDIR   F_CTL_RMDIR     /* remove a directory           */
#define RFS_CTL_SIZE    F_CTL_SIZE      /* Obtain the size of a file    */

/************************************************************************/
/*                                                                      */
/*      Definition of messages exchanged with the remote server         */
/*                                                                      */
/************************************************************************/

/* Values for the type field in messages */

#define RF_MSG_RESPONSE 0x0100          /* Bit that indicates response  */

#define RF_MSG_RREQ     0x0001          /* Read Request and response    */
#define RF_MSG_RRES     (RF_MSG_RREQ | RF_MSG_RESPONSE)

#define RF_MSG_WREQ     0x0002          /* Write Request and response   */
#define RF_MSG_WRES     (RF_MSG_WREQ | RF_MSG_RESPONSE)
```

Definitions For The Remote File System (Part 4)

```
/* Message header fields present in each message */

#define RF_MSG_HDR                          /* Common message fields        */\
        uint16  rf_type;                    /* Message type                 */\
        uint16  rf_status;                  /* 0 in req, status in response */\
        uint32  rf_seq;                     /* Message sequence number      */\
        char    rf_name[RF_NAMLEN];         /* Null-terminated file name    */

/* The standard header present in all messages with no extra fields     */

/************************************************************************/
/*                                                                      */
/*                              Header                                  */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_hdr {                        /* Header fields present in each*/
        RF_MSG_HDR                          /*   remote file system message */
};
#pragma pack()
```

Definitions For The Remote File System (Part 5)

```
/************************************************************************/
/*                                                                      */
/*                             Read                                     */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_rreq     {               /* Remote file read request     */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Position in file to read     */
        uint32  rf_len;                 /* Number of bytes to read      */
                                        /*   (between 1 and 1024)       */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_rres     {               /* Remote file read reply       */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Position in file             */
        uint32  rf_len;                 /* Number of bytes that follow  */
                                        /*   (0 for EOF)                */
        char    rf_data[RF_DATALEN];    /* Array containing data from   */
                                        /*   the file                   */
};
#pragma pack()
```

Definitions For The Remote File System (Part 6)

```
/************************************************************************/
/*                                                                      */
/*                              Write                                   */
/*                                                                      */
/************************************************************************/

#pragma pack(2)
struct  rf_msg_wreq     {               /* Remote file write request    */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Position in file             */
        uint32  rf_len;                 /* Number of valid bytes in     */
                                        /*   array that follows         */
        char    rf_data[RF_DATALEN];    /* Array containing data to be  */
                                        /*   written to the file        */
};
#pragma pack()

#pragma pack(2)
struct  rf_msg_wres     {               /* Remote file write response   */
        RF_MSG_HDR                      /* Header fields                */
        uint32  rf_pos;                 /* Original position in file    */
        uint32  rf_len;                 /* Number of bytes written      */
};
#pragma pack()
```

Communication With The Remote File Server

- All communication goes through a single function, *rfscomm*
- Each upper-half function (except *seek*) uses *rfscomm*
- *Rfscomm* handles
  - Registering a UDP port
  - The assignment of a sequence number to each outgoing messages
  - The transmission of a request
  - Timeout and retry
  - The reception of a reply
  - Validation of reply (to ensure it matches the request)

Remote File Communication (Part 1)

```
/* rfscomm.c - rfscomm */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rfscomm  -  Handle communication with RFS server (send request and
 *              receive a reply, including sequencing and retries)
 *------------------------------------------------------------------------
 */
int32   rfscomm (
         struct rf_msg_hdr *msg,        /* Message to send              */
         int32  mlen,                   /* Message length               */
         struct rf_msg_hdr *reply,      /* Buffer for reply             */
         int32  rlen                    /* Size of reply buffer         */
        )
{
        int32   i;                      /* Counts retries               */
        int32   retval;                 /* Return value                 */
        int32   seq;                    /* Sequence for this exchange   */
        int16   rtype;                  /* Reply type in host byte order*/
        int32   slot;                   /* UDP slot                     */
```

Remote File Communication (Part 2)

```
/* For the first time after reboot, register the server port */

if ( ! Rf_data.rf_registered ) {
        if ( (retval = udp_register(Rf_data.rf_ser_ip,
                        Rf_data.rf_ser_port,
                        Rf_data.rf_loc_port)) == SYSERR) {
                return SYSERR;
        }
        Rf_data.rf_udp_slot = retval;
        Rf_data.rf_registered = TRUE;
}

/* Assign message next sequence number */

seq = Rf_data.rf_seq++;
msg->rf_seq = htonl(seq);

/* Repeat RF_RETRIES times: send message and receive reply */

for (i=0; i<RF_RETRIES; i++) {
```

Remote File Communication (Part 3)

```
/* Send a copy of the message */

retval = udp_send(Rf_data.rf_udp_slot, (char *)msg,
        mlen);
if (retval == SYSERR) {
        kprintf("Cannot send to remote file server\n");
        return SYSERR;
}

/* Receive a reply */

retval = udp_recv(Rf_data.rf_udp_slot, (char *)reply,
        rlen, RF_TIMEOUT);

if (retval == TIMEOUT) {
        continue;
} else if (retval == SYSERR) {
        kprintf("Error reading remote file reply\n");
        return SYSERR;
}
```

Remote File Communication (Part 4)

```
                /* Verify that sequence in reply matches request */

                if (ntohl(reply->rf_seq) != seq) {
                        continue;
                }

                /* Verify the type in the reply matches the request */

                rtype = ntohs(reply->rf_type);
                if (rtype != ( ntohs(msg->rf_type) | RF_MSG_RESPONSE) ) {
                        continue;
                }

                return retval;          /* Return length to caller */
        }

        /* Retries exhausted without success */

        kprintf("Timeout on exchange with remote file server\n");
        return TIMEOUT;
}
```

An Example Remote File Operation (read)

- An application calls *read*, and control is passed to the upper-half read function, *rflread*
- *Rflread*
  - Forms a *read request* message
  - Calls *rfscomm* to send the message
  - *Rflcomm* blocks while awaiting a reply
  - Extracts data from the message and copies it to the caller's buffer
  - Updates the file position
- Note: each open file has a mutual exclusion semaphore to prevent other processes from using the file while an operation proceeds

Remote File Read (Part 1)

```
/* rflread.c - rflread */

#include <xinu.h>

/*------------------------------------------------------------------------
 * rflread  -  Read data from a remote file
 *------------------------------------------------------------------------
 */
devcall rflread (
          struct dentry *devptr,        /* Entry in device switch table */
          char  *buff,                  /* Buffer of bytes              */
          int32 count                   /* Count of bytes to read       */
        )
{
        struct  rflcblk *rfptr;         /* Pointer to control block     */
        int32   retval;                 /* Return value                 */
        struct  rf_msg_rreq  msg;       /* Request message to send      */
        struct  rf_msg_rres resp;       /* Buffer for response          */
        int32   i;                      /* Counts bytes copied          */
        char    *from, *to;             /* Used during name copy        */
        int32   len;                    /* Length of name               */

        /* Wait for exclusive access */

        wait(Rf_data.rf_mutex);
```

Remote File Read (Part 2)

```
/* Verify count is legitimate */

if ( (count <= 0) || (count > RF_DATALEN) ) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Verify pseudo-device is in use */

rfptr = &rfltab[devptr->dvminor];

/* If device not currently in use, report an error */

if (rfptr->rfstate == RF_FREE) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}

/* Verify pseudo-device allows reading */

if ((rfptr->rfmode & RF_MODE_R) == 0) {
        signal(Rf_data.rf_mutex);
        return SYSERR;
}
```

Remote File Read (Part 3)

```
/* Form read request */

msg.rf_type = htons(RF_MSG_RREQ);
msg.rf_status = htons(0);
msg.rf_seq = 0;                 /* Rfscomm will set sequence    */
from = rfptr->rfname;
to = msg.rf_name;
memset(to, NULLCH, RF_NAMLEN);  /* Start name as all zero bytes */
len = 0;
while ( (*to++ = *from++) ) {   /* Copy name to request         */
        if (++len >= RF_NAMLEN) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }
}
msg.rf_pos = htonl(rfptr->rfpos);/* Set file position           */
msg.rf_len = htonl(count);      /* Set count of bytes to read   */

/* Send message and receive response */

retval = rfscomm((struct rf_msg_hdr *)&msg,
                                sizeof(struct rf_msg_rreq),
                 (struct rf_msg_hdr *)&resp,
                                sizeof(struct rf_msg_rres) );
```

Remote File Read (Part 4)

```
        /* Check response */

        if (retval == SYSERR) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (retval == TIMEOUT) {
                kprintf("Timeout during remote file read\n");
                signal(Rf_data.rf_mutex);
                return SYSERR;
        } else if (ntohs(resp.rf_status) != 0) {
                signal(Rf_data.rf_mutex);
                return SYSERR;
        }

        /* Copy data to application buffer and update file position */

        for (i=0; i<htonl(resp.rf_len); i++) {
                *buff++ = resp.rf_data[i];
        }
        rfptr->rfpos += htonl(resp.rf_len);

        signal(Rf_data.rf_mutex);
        return htonl(resp.rf_len);
}
```

Summary

- Remote storage access mechanisms are a popular part of many operating systems
- In the Xinu remote disk subsystem, a device corresponds to a remote disk
- The remote disk device driver relies on a process to perform lower-half functions
- Caching is an important optimization for disk systems
- The Xinu remote disk system maintains a cache of recently-used disk blocks as well as a queue of requests
- The Xinu remote file access mechanism uses a synchronous approach that requires a message exchange for each operation, which means the remote file system access code does not need a separate process
- The Xinu remote file system implements Xinu file semantics whenever possible, and uses *control* to allow applications to access Linux file operations that are not normally available

Questions?