Module XX

A Remote File System

A Remote File System

Remote File Access

- Involves two software components that
  - Operate on two separate computers
  - Communicate over a network or the Internet
- The *remote file server* component
  - Runs on a computer that has a local file system
  - Accepts requests to perform file operations
- The remote file *client* component
  - Is part of an operating system
  - Sends requests to a server and obtains replies

Operations A Remote File System Supports

- A remote file system usually supports typical file operations
  - Open or close a file
  - Read data from an open file
  - Write data to an open file
  - Move to an arbitrary position in an open file
  - Create, delete, or rename files
  - Change a file’s metadata, such as the ownership and access privileges

Design Questions

- Can multiple clients access a given server?
- Can a client access files on more than one server at the same time?
- Must user IDs on the client computer agree with user IDs on the server?
- Exactly what file semantics does a remote file system support?
  - Precisely the same file operations and semantics as a local file system?
  - A subset of the operations and semantics supported by the local file system?
  - A superset of the operations and semantics supported by the local file system?

The Xinu Remote File Paradigm

- Clients on multiple Xinu machines are allowed to send read and write requests to a given server concurrently
- Allowing multiple clients to access a server introduces the possibility of interference (e.g., two clients may attempt to write to the same byte of a file at the same time)
- The Xinu solution
  - A server serializes all incoming requests (i.e., enqueues them and handles one at a time)
  - A subsequent *read* always returns the last value written, independent of which client wrote it
  - If additional coordination is needed among applications using a file, it is the programmer’s responsibility

Server Operation

- A server maintains a set of currently open files
- When a client sends an *open* request, the server
  - Checks to see if the file is already open, and does nothing if it is
  - Otherwise opens the file and records it in the set of open files
- Read and write operations from all clients refer to the same open file (a client does not have its own open file on the server)
- When all clients close a file, the server closes the file

The Remote File Interface On A Xinu Client System

- Follow pattern to defining a master device (*RFILESYS*) and a set of pseudo-devices
- To open a remote file, a process calls *open* on the master device

```
d = open(RFILESYS, "file", mode);
```

- The *open* call
  - Allocates one of the file pseudo devices
  - Returns, d, the descriptor of the open device
- The caller uses descriptor d to *read* or *write* data to the file
- When it finishes using the file, the process calls *close* on descriptor d
- Note: the RFILESYS device is also used for control operations (e.g., delete a file)

The Structure Of The Remote File System Code

- The remote file system client code differs from the remote disk client code
- Unlike a remote disk client, a remote file client does not maintain a queue of requests, and does not need a communication process
- Instead
  - Remote server access uses a synchronous approach
  - Each operation causes a request-response exchange with the remote server
  - Only one request can be outstanding at a time
- An upper-half function
  - Forms a request message and calls function *rfscomm* to send the request to the server and obtain a response
  - Waits for a response, and returns the response to its caller

The Cost Of Remote File Operations

- The Xinu design 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 communication overhead is highest when only a small amount of data is transferred per request
- Example: sending 1000 bytes of data in a single request instead of one byte reduces the number of packets transferred by a factor of 1000!
- Consequence: programmers are discouraged from using *putc* or *getc* to access a remote file because using *read* and *write* to transfer large blocks of data incurs much less overhead

The Question Of File System Semantics

- The Xinu remote file server runs on a Unix system (Linux) that 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 an exact equivalent
- There are two possibilities
  - Arrange the remote file server to emulate (when possible) the Xinu file semantics
  - Allow applications running on Xinu to use the Linux file system functionality and semantics

Our Design

- Our remote file server implements Xinu file system semantics whenever doing so is both feasible and efficient
  - Example: to emulate Xinu "o" and "n" modes, the remote file 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 Xinu process to
  - Create or remove a directory
  - Truncate a file
  - Obtain the current size of a file (which allows a Xinu application to move to the end and *append* new data)

Operations For The Xinu Remote File System

- *Open* – open a file
- *Close* – terminate use of file
- *Read* – obtain data from a file
- *Write* – 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)
- Note *getc* and *putc* are provided, but their use is discouraged

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

- Each call to open is assigned a new pseudo device
- The pseudo-device maintains a file position
- Consequence: two processes can open the same file and maintain their own file position
- Because Xinu stores the position at the client, every request sent to the server must specify a position
- 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 */

#ifndef Nrfl
#define Nrfl    10
#endif

/* Control block for a remote file pseudo-device */

#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
#define RF_SERVER       "example.com"
#endif

#ifndef RF_SERVER_PORT
#define RF_SERVER_PORT  53224
#endif

#ifndef RF_LOC_PORT
#define RF_LOC_PORT     53224
#endif
```

Definitions For The Remote File System (Part 2)

```
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      */
        int32   rf_udp_slot;            /* UDP slot to use              */
        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   */
```

Message Formats Used With The Remote File System

- Use the same approach as the remote disk system
- Define a request and reply message for each operation
- Note
  - File rfilesys.h is shared between client and server software
  - Key concept: the message formats and constants are only defined once

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 message
  - 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                     */
        char    err[128];               /* Error message buffer         */
```

Remote File Communication (Part 2)

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

if ( ! Rf_data.rf_registered ) {

    /* Convert the server name to an IP address */

    if (dnslookup(RF_SERVER, &Rf_data.rf_ser_ip) == SYSERR) {
        sprintf(err, "rfs server %s is invalid", RF_SERVER);
        panic("err");
    }

    if ( (slot = 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 = slot;
    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 and calls *rfscomm* to send the message
  - *Rfscomm* blocks the calling process to await a reply
  - When the reply arrives, *rfscomm*
    - Extracts the data from the message and copies it to the caller’s buffer
    - Updates the file position in the control block for the pseudo device
    - Returns to the caller, allowing the *read* to complete
- 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<ntohl(resp.rf_len); i++) {
                *buff++ = resp.rf_data[i];
        }
        rfptr->rfpos += ntohl(resp.rf_len);

        signal(Rf_data.rf_mutex);
        if (ntohl(resp.rf_len) == 0) {
                return EOF;
        }
        return ntohl(resp.rf_len);
}
```

The Remote File Server

- Is launched in a directory on a Unix system
- Accepts requests from client(s)
- Opens each file in the directory where the server is running
- Prevents clients from accessing files in higher-level directories
- Examples: the server forbids access to files with names

```
/users/xxx/y/z
./../../../bbb/qq
```

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?