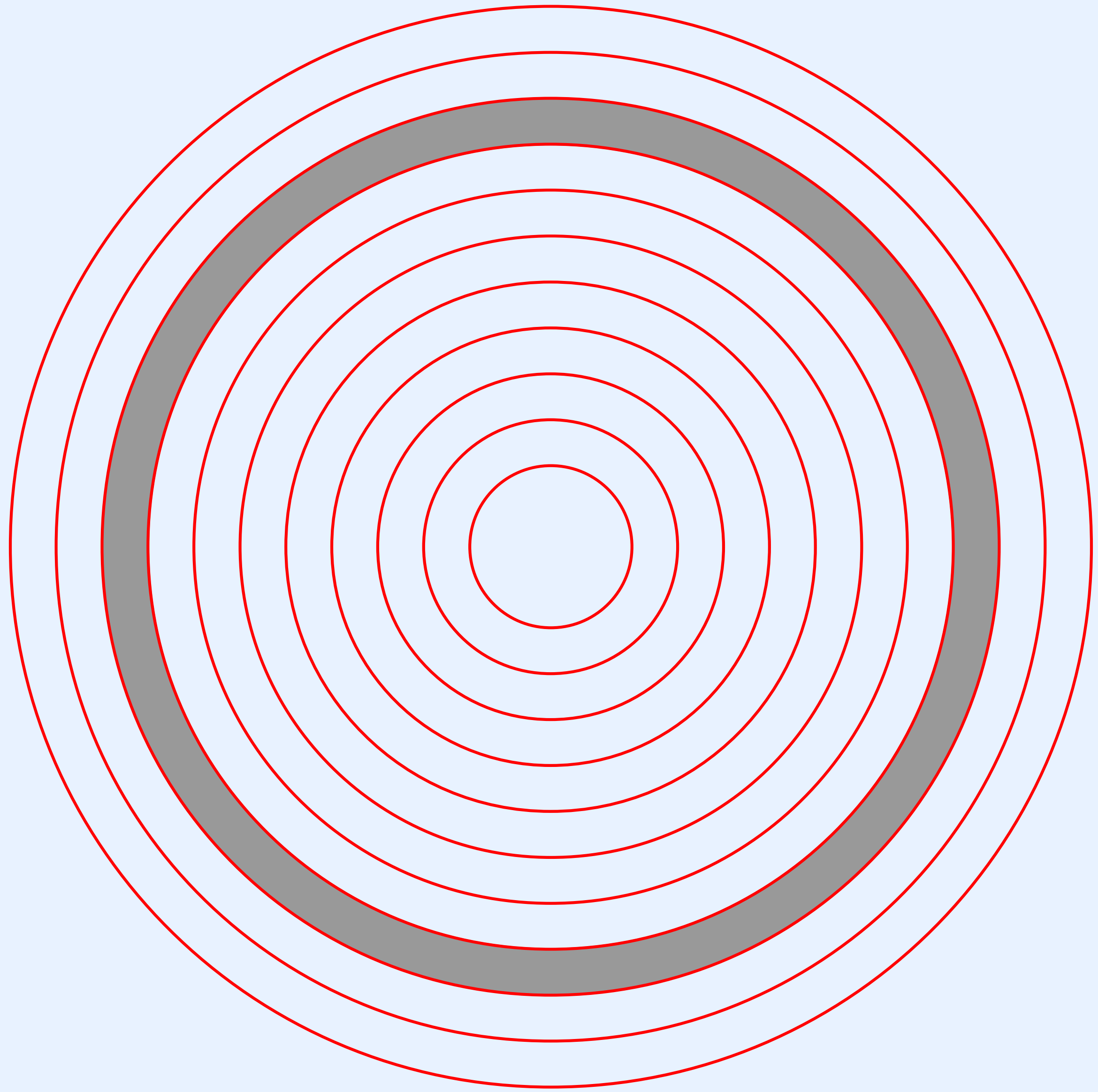
Module XVIII

Remote Access And A Remote Disk Driver

Location Of Remote Disk 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 the 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: computers on a network share a 96-bit address space
  - Communication is positioned at lowest level of the system
- Xerox Alto Environment
  - The model: each computer has a local process manager, and all share files
  - Communication is positioned 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 user IDs

Disk Hardware

- We use the term *disk* to refer to a solid-state disk (*SSD*) as well as to an older electro-mechanical disk
- Conceptually, a disk appears to consist of 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)
- The blocks on a disk are numbered 0, 1, 2, ...
- Disk hardware only 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} block on the disk
- The hardware always transfers a complete block between memory and disk
- Note: special-purpose hardware used in high-performance storage systems typically uses a larger block size (e.g., 4096 bytes)

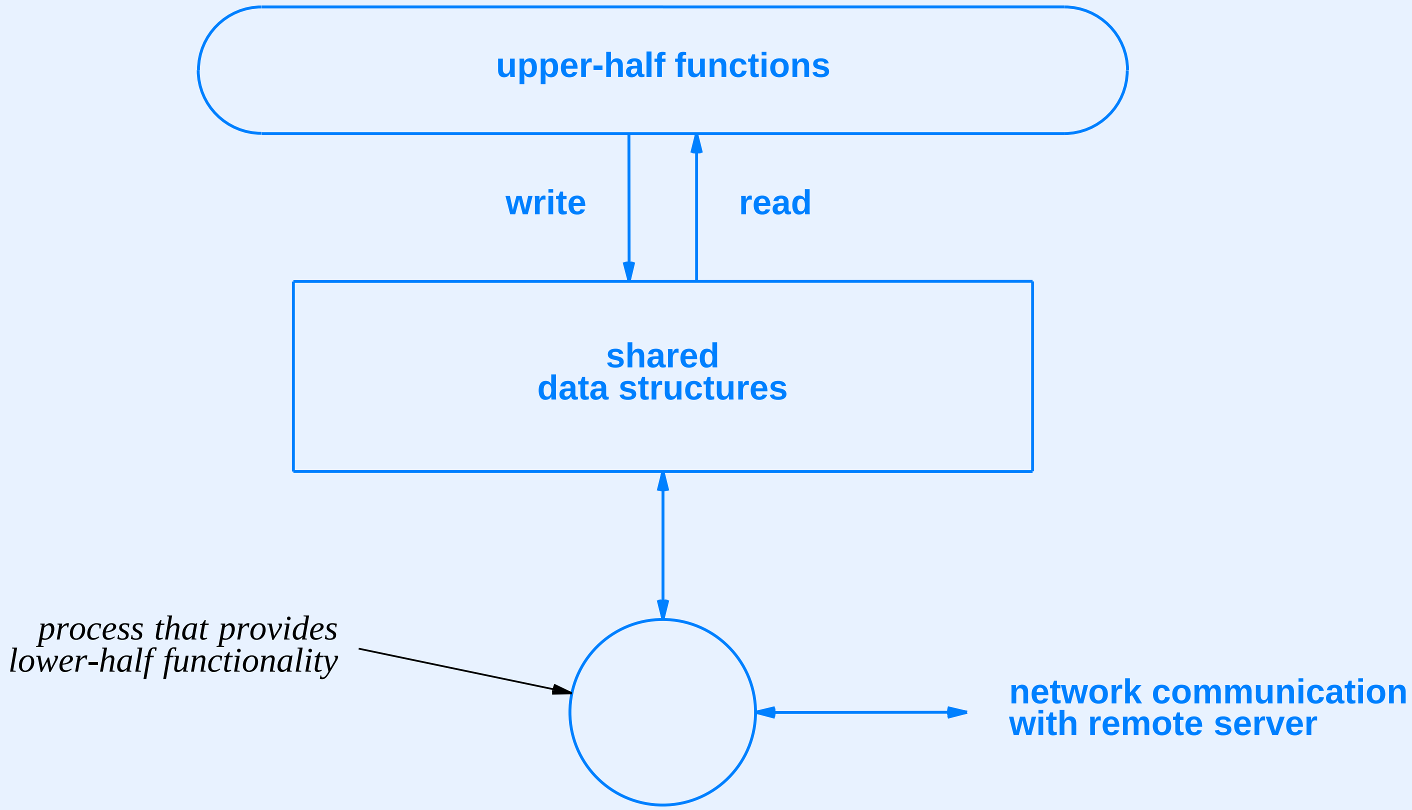
The Remote Disk Paradigm

- The idea: allow an operating system to fetch or store blocks to a remote disk
- In terms of hardware
  - A computer on the network runs *remote disk server* software
  - The computer has one or more physical disks attached
- In terms of software
  - A *client* operating system contains software that can send messages over a network to the remote disk server
  - Each request either contains a block to be written to the remote disk or a request to read a block from the remote disk
- Note: the client OS can store arbitrary data in each disk block (e.g., a boot block or pieces of a file system)

The Structure Of Xinu Remote Disk Driver Software

- Like a conventional device driver
  - Has upper-half *read* and *write* functions called by processes
  - Has shared data structures
- Unlike a conventional driver
  - Uses a dedicated, high-priority communication process in place of a lower-half
  - Does not use interrupts to trigger the lower-half, but arranges instead for the communication process to wait on a semaphore until a request arrives
- The communication process handles *all* communication with the remote server

Illustration Of Remote Disk Access



A Remote Disk Server In Practice

- Powerful remote disk server hardware provides disks for multiple clients
- The server may
  - Have *N* physical disks and dedicate each disk to one client
  - Dedicate a *virtualized* disk to each client
- *Virtualized disks*
  - Is a popular approach
  - Provides each client with the illusion of a separate physical disk (i.e., the client has its own set of blocks 0, 1, 2, ...)
  - Maps the client requests onto the set of local disks
  - Hides the details of the mapping from clients

Caching And its Importance

- The access pattern: file systems exhibit *temporal locality* in which a give disk block is accessed repeatedly
- To optimize performance, a disk driver maintains two data structures
  - A cache of recently-accessed disk blocks
  - A set of pending read or write requests
- Invariant: at any time, a copy of block *K* in the cache contains the latest data written to block *K*

Last-Write Semantics

- A disk driver receives a sequence of *read* and *write* requests where each request specifies a disk block number. the driver must guarantee that the block returned for a *read* request is the latest block that has been written to the disk
- To enforce last-write semantics
  - The driver inserts requests at the tail of a queue
  - The lower-half process continually removes and performs a request from the head of the queue
- Complication: the queue may contain *write* requests for the same block

An Interesting Synchronization Problem

- A semaphore will block a process that attempts to perform a *read* or *write* until space is available in the queue
- However...
  - The disk driver must guarantee that requests are handled in the order they occur
  - Multiple processes may pass the semaphore if multiple spaces are available
  - The scheduler may run a higher priority process first, which means that

Using a semaphore to access the queue of requests does not guarantee strict temporal order of requests.

Satisfying The Strict Ordering Requirement

- Introduce a *serialization mechanism*
- Have the mechanism guarantee that items will be processed in the order received
- Trick: use an extra *serial queue* that has enough slots so that every process has a slot
- Block a process until its request can be moved to the request queue

Serial Queue Insertion And Resumption Of the Communication Process

- We will see that the remote disk communication process suspends itself when the request queue becomes empty
- Consequence: when it inserts an entry in the serial queue, a process must perform two steps
  - Resume the remote disk communication process
  - Suspend itself (it will be resumed when its request is moved to the request queue)
- Unfortunately, a race condition exists because the communication process runs at high priority
- Once the communication process resumes, the calling process will not be able to suspend itself

Solving The Race Condition

- Create a function, *rdsars*, that atomically performs two tasks
  - Resumes the communication process
  - Suspends the calling process
- Implementation
  - Set the current process state to the desired state *PR_SUSP*
  - If the communication process is suspended, make it ready
  - If not, simply call resched

Code For Rdsars (Part 1)

```
/* rdsars.c - rdsars */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  rdsars - atomically resume a high-priority server process and suspend
 *              the current process
 *------------------------------------------------------------------------
 */
syscall rdsars(
          pid32         pid             /* ID of the process to resume  */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process's table entry */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }
```

Code For Rdsars (Part 2)

```
        /* Set current process state to suspended */
        proctab[currpid].prstate = PR_SUSP;

        /* If target process is suspended, resume it */

        prptr = &proctab[pid];
        if (prptr->prstate == PR_SUSP) {
                ready(pid);
        } else {
                resched();
        }
        restore(mask);
        return OK;
}
```

The Xinu Remote Disk Interface

- Works exactly like a local disk, by allowing a caller to
  - *Write* a specified block to the disk
  - *Read* a specified block from the disk
- Defines a Xinu device named *RDISK* that corresponds to the remote disk
- Arranges driver software for the RDISK device to support *read* and *write* operations
- Hides all network communication

Read And Write Operations On A Disk With Xinu

- Each read or write operation on a disk requires a block number, but *read* and *write* do not have an extra argument
- Trick: the length of a disk block is fixed, so interpret the “length” field in a *read* and *write* call as a disk block number
- Example: if the remote disk device is named *RDISK*, to read block 5 of the remote disk, a process calls *read* with 5 as the length argument:

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

The Request Queue

- Operates exactly like the request queue used by a local disk driver
- Each item in the request queue specifies
  - A disk block number
  - An operation (*read* or *write*)
  - A pointer to a buffer that either contains data to be written (for *write*) or to be filled (for *read*)
  - The ID of a process waiting for the request to be fulfilled
- Items in a request queue are ordered in FIFO order
- Each item in the cache contains a block number and buffer that holds the data for one disk block

Implementation Of The Lower-half Software

- Consists of a communication process named *rdsprocess* that handles all communication with remote server
- Operation
  - Repeatedly extract and handle the next request (i.e., build a request message and send to the remote server)
  - Updates the cache when a block is changed (written or read)
- On each iteration, the rdsprocess moves items from the serial queue to the request queue if space is available

Implementation Of The Lower-half Software
(continued)

- Rdsprocess satisfies requests locally, if possible
- Obvious optimization: when processing a *read* request, search the cache to see if the block is in the cache
- Another optimization: search the request queue backward to find the last pending *write* request for the block

The Use Of Rdscomm

- Rdsprocess calls function *rdscomm* to
  - Format a message
  - Add a unique sequence number to each outgoing message
  - Send the message to the server and wait for a reply
  - Retransmit the message, if the reply does not arrive promptly
- Separating rdsprocess from rdscomm allows rdscomm to be used when opening, closing, or deleting a remote disk

The Sync Operation

- Disk hardware only supports *fetch* and *store* operations
- However, the request queue supports three operations
  - *Read* (fetch a block from disk)
  - *Write* (store a block to disk)
  - *Sync* (synchronize requests)
- The *sync* operation
  - Blocks the caller until all previously-written blocks have been stored on disk
  - Is used by a file system to guarantee metadata is saved
  - Is invoked with a *control* function
  - May be used by individual processes as well as a file system

How Sync Works

- A process invokes the “sync” control function
- The device driver
  - Adds a sync request for the process to the request queue
  - Suspends the calling process
- Once the *sync* request reaches head of queue (i.e., all previous requests have been satisfied), the communication process
  - Resumes the process that made the sync request
- Note that *sync* is handled locally — no message is sent to the remote server and no data is transferred

Structure Of A Request Queue Node (from rdisksys.h)

```
/* excerpt frpm rdiisksys.h */

/* Operations for request queue */

#define RD_OP_READ      1                /* Read operation on req. list  */
#define RD_OP_WRITE     2                /* Write operation on req. list */
#define RD_OP_SYNC      3                /* Sync operation on req. list  */

/* Definition of a serial queue node */

struct  rdsent {                         /* Entry in the serial queue    */
        int32   rd_op;                   /* Operation - read/write/sync  */
        uint32  rd_blknum;               /* Disk block number to use     */
        char    *rd_callbuf;             /* Address of caller's buffer   */
        pid32   rd_pid;                  /* Process that initiated the   */
};                                       /*   request                    */

/* Definition of a request queue node */

struct  rdqnode {                        /* Node in the request queue    */
        struct  rdqnode *rd_next;        /* Pointer to next node         */
        struct  rdqnode *rd_prev;        /* Pointer to previous node     */
        int32   rd_op;                   /* Operation - read/write/sync  */
        uint32  rd_blknum;               /* Disk block number to use     */
        char    *rd_callbuf;             /* Address of caller's buffer   */
        pid32   rd_pid;                  /* Process making the request   */
        char    rd_wbuf[RD_BLKSIZ];      /* Data for a write operation   */
};
```

Structure Of A Cache Node (from rdisksys.h)

```
/* Definition of a node in the cache */

struct  rdcnode {                       /* Node in the cache             */
        struct  rdcnode *rd_next;       /* Pointer to next node          */
        struct  rdcnode *rd_prev;       /* Pointer to previous node      */
        uint32  rd_blknum;              /* Number of this disk block     */
        byte    rd_data[RD_BLKSIZ];     /* Data for the disk block       */
};
```

- A given block only appears once in the cache (the latest copy)
- A block is not placed in the cache until it has been written to disk (i.e., a block number is not duplicated in both the request queue and cache)

Constants For The Remote Disk (from rdisksys.h)

```
/* Constants for remote disk device control block */

#define RD_IDLEN        64              /* Size of a remote disk ID     */
#define RD_STACK        16384           /* Stack size for comm. process */
#define RD_PRIO         600             /* Priorty of comm. process     */
                                        /*  (Must be higher than any    */
                                        /*   process that reads/writes  */

/* Constants for state of the device */

#define RD_CLOSED       0               /* Device is not in use         */
#define RD_OPEN         1               /* Device is open               */
#define RD_PEND         2               /* Device is being opened       */
#define RD_DELETING     3               /* Device is being deleted      */
```

Structure Of A Remote Disk Control Block (from rdisksys.h)

```
/* Device control block for a remote disk */

struct  rdscblk {                       /* Remote disk control block    */
        int32   rd_state;               /* State of device              */
        char    rd_id[RD_IDLEN];        /* Disk ID currently being used */
        int32   rd_seq;                 /* Next sequence number to use  */
        struct  rdcnode *rd_chead;      /* Head of cache                */
        struct  rdcnode *rd_ctail;      /* Tail of cache                */
        struct  rdcnode *rd_cfree;      /* Free list of cache nodes     */
        struct  rdqnode *rd_qhead;      /* Head of request queue        */
        struct  rdqnode *rd_qtail;      /* Tail of request queue        */
        struct  rdqnode *rd_qfree;      /* Free list of request nodes   */
        struct  rdsent  rd_sq[RD_SSIZE];/* Serial queue circular buffer */
        int32   rdshead;                /* Head of the serial queue     */
        int32   rdstail;                /* Tail of the serial queue     */
        int32   rdscount;               /* Count serial queue items     */
        pid32   rd_comproc;             /* Process ID of comm. process  */
        uint32  rd_ser_ip;              /* Server IP address            */
        uint16  rd_ser_port;            /* Server UDP port              */
        uint16  rd_loc_port;            /* Local (client) UPD port      */
        bool8   rd_registered;          /* Has UDP port been registered?*/
        int32   rd_udpslot;             /* Registered UDP slot          */
};

extern  struct  rdscblk rdstab[];       /* Remote disk control block    */
```

Messages Exchanged With The Remote Disk Server

- The remote disk system uses five message types when communicating between the local operating 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 remote server

Names For Remote Disks

- A remote disk server
  - Retains disk contents across server reboots
  - Maintains multiple virtual disks
  - 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
- Possibilities
  - Students in a class could each use their login ID as a unique disk name
  - The IP address of a Xinu back-end computer (converted to a text string) could be used as a disk 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 software 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)

- For each operation, two message formats must be defined, such as
  - *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 the block number to fetch
  - A *read reply* contains actual data in addition to the block number

Declarations For Message Types (from rdisksys.h)

```
/************************************************************************/
/*      Definition of messages exchanged with the remote disk server    */
/************************************************************************/
/* 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 (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      */

/*********************************************************************/
/*                              Header                              */
/*********************************************************************/
/* The standard header present in all messages with no extra fields */
#pragma pack(2)
struct  rd_msg_hdr {                    /* Header fields present in each*/
        RD_MSG_HDR                      /*   remote disk system message */
};
#pragma pack()
```

Message Formats (from rdisksys.h)

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

#pragma pack(2)
struct  rd_msg_rres     {                   /* Remote disk 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 (from rdisksys.h)

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

The Importance Of Disk Block Caching

- Disk I/O, even for a local disk, is much slower than memory accesses
- Communication to a remote disk server makes disk access *extremely* slow
- Reminder: disk accesses exhibit temporal locality in which a given block is accessed repeatedly
- Keeping a disk block in a memory cache speeds up access times substantially
- Result: all disk drivers (even for SSDs) rely on a cache to achieve reasonable performance

Next Steps

- Look through the files for the remote disk driver (in directory device/rds or online) to see how
  - A call to rdsread works
  - A call to rdswrite works
  - What happens when a process calls

```
control(RDISK, RD_CTL_SYNC, 0);
```

- Either ask questions now or come to the next class with questions

Note: be sure to look at the latest Xinu code.

Questions?