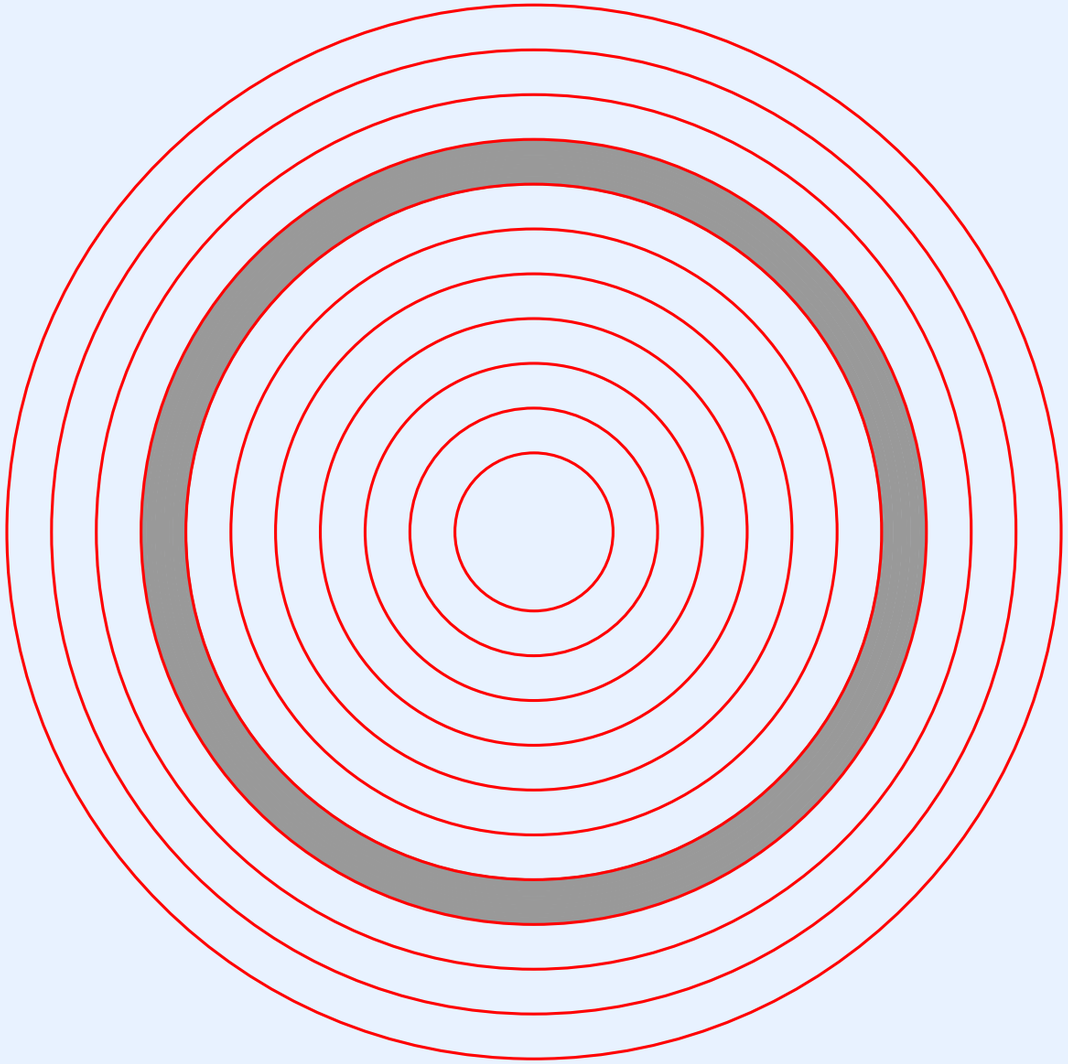
Module XVII

Networking And Protocol Implementation

Location Of Networking In The Hierarchy



Is The Hierarchical Level Correct?

- There are two possible approaches
  - Build a conventional operating system and add networking
  - Build networking code first and ensure all pieces of the operating system are distributed (e.g., a distributed process manager)
- Xinu places networking code at a high level of the hierarchy because most of the operating system is not distributed

A Fundamental Observation

One cannot undertake an operating system design without including network communication protocols, even in the embedded systems world.

Communication Systems

- A variety of network technologies have been devised
  - Wired (e.g., Ethernet)
  - Wireless (e.g., Wi-Fi and 5G)
- A computer can use
  - Local network communication: communicate directly over a network with other systems on the same network
  - Internet communication: communicate over a local network, but send packets through a *router* to an arbitrary computer on the Internet
- Internet communication has become the standard except for small, special-purpose embedded systems

Communication Protocols

- We use the term *communication protocols* to describe the standards that specify communication details such as
  - Message formats
  - Data representation (e.g., endianness)
  - Rules for message exchange
  - How to handle errors (e.g., lost packets)
- Protocols used in the Internet are known by the name *TCP/IP protocols*

Communication Protocols And This Course

- We will not discuss protocol details
- We will consider only a minimalistic subset of Internet protocols and focus on aspects pertinent to operating systems design
  - How applications use the communication system
  - The processes that are needed
  - The need for buffering
- To learn more
  - Read a leading text on TCP/IP
  - Take an internetworking course that uses an expert's text

Synchronous Interface For Network Hardware

- As in most operating systems, Xinu has a device driver for each network interface
- For example, Xinu defines an *ETHER* device for an Ethernet interface
- The device driver for the device provides
  - Synchronous *read* that blocks until a packet arrives and then returns the packet
  - Synchronous *write* that blocks until a buffer is available and then accepts an outgoing packet
- Our example code assumes all communication uses an Ethernet

DMA Device Drivers

- Ethernet device hardware uses *Direct Memory Access* (*DMA*)
- The operating system
  - Allocates a set of input buffers and a set of output buffers in memory and gives the device the addresses of the buffers
  - Marks the input buffers empty and starts device input
  - Places outgoing packets in the output buffers and starts device output
- The device hardware
  - Picks up outgoing packets directly from the output buffers
  - Delivers incoming packets directly to the input buffer
- See Chapter 16 in the text for explanation of how a DMA driver works

Network I/O

- Except for special-purpose embedded systems, application processes
  - Never *read* or *write* directly to a network device
  - Always invoke network protocol software functions to perform network communication
- Network protocol software in the operating system
  - Accepts requests from applications to contact a remote site, forms outgoing packets as needed, and sends them
  - Blocks applications that request network input until a message and/or data arrives
  - Uses a dedicated process to read incoming packets, process them, and deliver the results to waiting applications

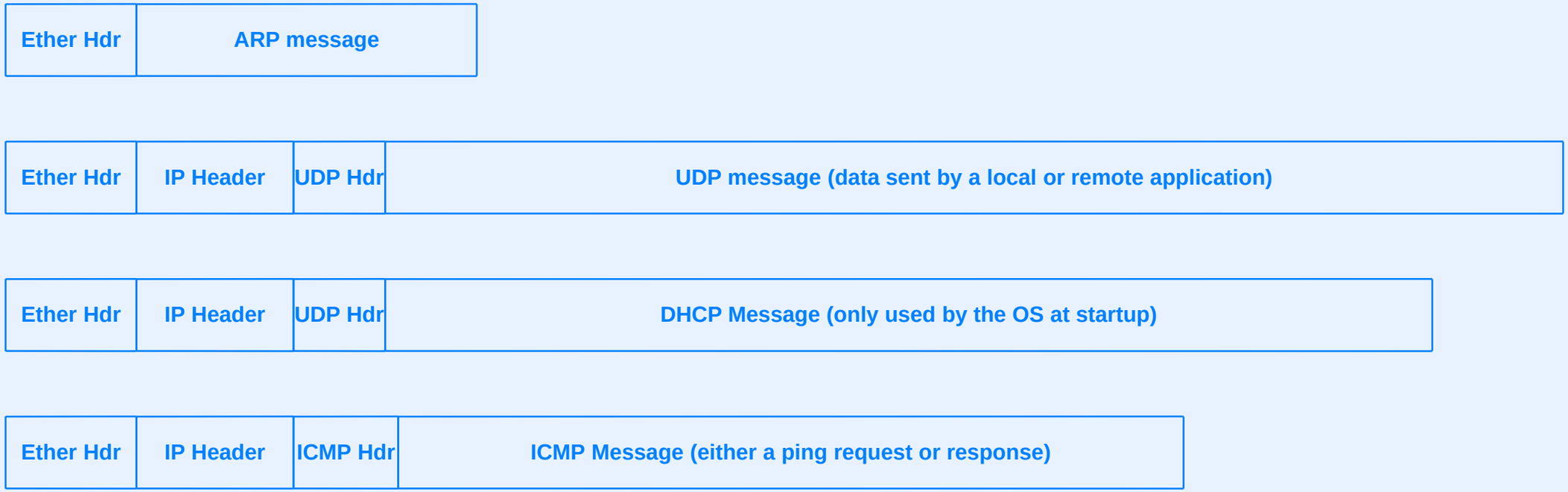
Protocols In Our Example

- You do not need to understand protocols, but you will see the following names

IP	Internet Protocol – defines an Internet Protocol address (IP address) for each computer on the global Internet plus the format of Internet packets
UDP	User Datagram Protocol – defines *protocol port numbers* used to identify individual applications on a given computer and a message format used when UDP messages travel across the Internet
ARP	Address Resolution Protocol – allows a computer to find the Ethernet address of a computer on a local network given its IP address
DHCP	Dynamic Host Configuration Protocol – used by a computer at startup to obtain its IP address and related information
ICMP	Internet Control Message Protocol – in our implementation, only used by the *ping* program to see if a computer is alive

Protocol Headers And Message Formats

- Each packet starts with a series of headers followed by data
- In our implementation, a packet being sent or received will have one of the following forms:



Implementing Concatenated Headers

- Most systems build a packet dynamically, adding headers one at a time as needed
- Xinu takes a shortcut: define two structures
  - One for an Ethernet header followed by an arp message
  - Another for the three cases of an Internet packet
    - Ethernet header, IP header, UDP header, UDP message
    - Ethernet header, IP header, UDP header, DHCP message
    - Ethernet header, IP header, ICMP header, ICMP message
- A further simplification: the only ICMP messages are *echo request* and *echo reply* (i.e., *ping* messages)
- The point is merely to illustrate protocols in an operating system

Packet Format Declarations

- The struct *arppacket* defines the format of an Ethernet packet carrying ARP messages
- The struct *netpacket* defines two cases of an Internet packet
- The netpacket struct starts with an Ethernet packet header followed by an IP header, and then has a union to define
  - A UDP packet encapsulated in the IP packet
  - An ICMP echo request or reply packet encapsulated in the IP packet
- A separate struct defines a DHCP message (which has many fields)

Network Definitions In net.h (Part 1)

```
/* net.h */

#define NETSTK          8192            /* Stack size for network setup */
#define NETPRIO         500             /* Network startup priority     */
#define NETBOOTFILE     128             /* Size of the netboot filename */

/* Constants used in the networking code */

#define ETH_ARP     0x0806              /* Ethernet type for ARP        */
#define ETH_IP      0x0800              /* Ethernet type for IP         */
#define ETH_IPv6    0x86DD              /* Ethernet type for IPv6       */

/* Format of an Ethernet packet carrying IPv4 and UDP */

#pragma pack(2)
struct  netpacket       {
        byte    net_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC address  */
        byte    net_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  net_ethtype;            /* Ethernet type field          */
        byte    net_ipvh;               /* IP version and hdr length    */
        byte    net_iptos;              /* IP type of service           */
        uint16  net_iplen;              /* IP total packet length       */
        uint16  net_ipid;               /* IP datagram ID               */
        uint16  net_ipfrag;             /* IP flags & fragment offset   */
        byte    net_ipttl;              /* IP time-to-live              */
        byte    net_ipproto;            /* IP protocol (actually type)  */
        uint16  net_ipcksum;            /* IP checksum                  */
        uint32  net_ipsrc;              /* IP source address            */
        uint32  net_ipdst;              /* IP destination address       */
```

Network Definitions In net.h (Part 2)

```
        union {
         struct {
          uint16        net_udpsport;   /* UDP source protocol port     */
          uint16        net_udpdport;   /* UDP destination protocol port*/
          uint16        net_udplen;     /* UDP total length             */
          uint16        net_udpcksum;   /* UDP checksum                 */
          byte          net_udpdata[1500-28];/* UDP payload (1500-above)*/
         };
         struct {
          byte          net_ictype;     /* ICMP message type            */
          byte          net_iccode;     /* ICMP code field (0 for ping) */
          uint16        net_iccksum;    /* ICMP message checksum        */
          uint16        net_icident;    /* ICMP identifier              */
          uint16        net_icseq;      /* ICMP sequence number         */
          byte          net_icdata[1500-28];/* ICMP payload (1500-above)*/
         };
        };
};
#pragma pack()

#define PACKLEN sizeof(struct netpacket)

extern  bpid32  netbufpool;             /* ID of net packet buffer pool */
```

Network Definitions In net.h (Part 3)

```
struct  network {                       /* Network information          */
        uint32  ipucast;                /* Computer's IP unicast address*/
        uint32  ipbcast;                /* IP broadcast address         */
        uint32  ipmask;                 /* IP address mask              */
        uint32  ipprefix;               /* IP (network) prefix          */
        uint32  iprouter;               /* Default router address       */
        uint32  bootserver;             /* Boot server address          */
        uint32  dnsserver;              /* DNS server address           */
        uint32  ntpserver;              /* NTP (time) server address    */
        bool8   ipvalid;                /* Nonzero => above are valid   */
        byte    ethucast[ETH_ADDR_LEN]; /* Ethernet multicast address   */
        byte    ethbcast[ETH_ADDR_LEN]; /* Ethernet broadcast address   */
        char    bootfile[NETBOOTFILE];  /* Name of boot file            */
};

extern  struct  network NetData;        /* Local Network Interface info */
```

- Global variable *NetData* holds network information obtained at startup, including
  - The computer's IP address (needed for outgoing as well as incoming packets)
  - The address mask for the local network
  - The address of an Internet router to use (needed for outgoing packets)
  - The address of an NTP time server (used to obtain the time of day)

Services An Application Can Use

- In this version of Xinu, an application can either
  - Use UDP to exchange messages with another application running on a computer on the Internet
  - Use ICMP to send a *ping* packet and receive a reply from an arbitrary computer on the Internet
- The other protocols (ARP and DHCP) merely provide support; they are handled by the network code and invisible to an application

Identifying An Application

- UDP allows multiple applications on a given computer to communicate with other applications running on computers attached to the Internet
- To identify a remote application, a sending application must specify two items
  - The computer on which the remote application runs
  - An ID that identifies a specific application on the computer
- For the two items, UDP uses
  - The 32-bit IP address of the remote computer
  - A 16-bit integer called a *UDP protocol port number* that identifies an application
- For this course, you do not need to know how IP addresses and port numbers are obtained; just understand that two items are needed to identify each application

Features Of Networks Related To Operating Systems

- Three aspects of Internet software relate directly to the operating system
  - The interface that applications use to communicate over the Internet
  - The process structure used internally to implement protocols
  - The need for buffering
- We will consider all three

The Interface To Network Protocols

An Interface Used To Communicate Over The Internet

- Xinu follows the same approach as the well-known socket API
  - Before sending data, an application calls a function to register information about a remote destination (i.e., a specific application on a specific remote computer)
  - Network code in the operating system responds by allocating an internal data structure, placing the information in the data structure, and returning a small integer descriptor that the application uses for communication
  - Similar to other descriptors in Xinu, each descriptor used for network communication is an index into an array, and is informally called a *slot number*
  - The application uses the descriptor to *send* and *receive* data (there's no need to specify the remote application each time the application sends or receives data)
  - When finished, the application releases the descriptor

Xinu Interface Functions That Applications Use for UDP

- *udp_register* – called by an application to register endpoint information, a remote computer (IP address), remote UDP port, and a local UDP port
- *udp_send* – called by an application to send a UDP packet to a previously-registered endpoint
- *udp_recv* – called by an application to receive a UDP packet from a previously-registered remote endpoint
- *udp_recvaddr* – called by a server application to receive a UDP packet and record the sender's address (allows an application to receive messages from an arbitrary application)
- *udp_release* – called by an application to release a previously-registered endpoint
- Note: the descriptor returned by udp_register must be passed to the other functions

Processing An Incoming UDP Packet

- When a packet arrives, the network code calls internal function *udp_in*
- *Udp_in* searches the table of registered endpoints
  - If the incoming packet matches a registered endpoint, the packet is enqueued on the entry, and the semaphore for the entry is signaled to allow a waiting process (if any) to become ready and read the message
  - Else no match is found in the table, and the incoming packet is ignored (silently dropped)

Timeout And Retransmission

- Retransmission of a packet is fundamental in networking
- Retransmission handles packet loss by sending a second copy if the original is lost
- The idea: repeat the following *K* times
  - Send a request
  - Wait up to *N* milliseconds for a reply
- If a reply arrives, process the reply immediately
- If no reply arrives after *K* times declare failure
- Typically, (*K* is a small number, such as 3)

Xinu Network Functions And Timeout

- The network interface functions allow an application to specify a maximum time to wait for a reply
- Example
  - When calling udp_recv, an application specifies a maximum time to wait
  - The call either returns a message that was received or *TIMEOUT*
- The ICMP (ping) interface operates the same way as the UDP interface
- The network code for UDP and ICMP implements the timeout

Implementation Of Timeout

- Functions that read an incoming message uses *recvtime* to implement timeout (recall that *recvtime* was covered previously)
- When it calls *udp_recv*, a process specifies a maximum wait time
- The code in *udp_recv* performs the following steps
  - If no packet has arrived in the slot, the code
    * Places the current process ID in the data structure for the slot
    * Calls *recvtime* to block the calling process (note:when a packet arrives for the slot, the network code uses *send* to send a message to the waiting process)
    * If a TIMEOUT occurs, returns TIMEOUT to the caller
  - Copies the contents of the packet to the callers buffer

An Example Of Using UDP: Time Of Day

- The first time a process requests the time of day, the *gettime* function
  - Calls *getutime* to contact an NTP time server and obtains the current time of day (seconds since January 1, 1900)
  - Converts the time to Xinu time (seconds since January 1, 1970)
  - Computes and stores the time of day when the system booted (i.e., subtracts *clktime* from the current time of day)
- Once the time of day at which the system booted has been stored, Xinu never needs to contact a time server again
- Instead, *getutime* merely adds *clktime* to the time of day at which the system booted

Obtaining The Time Of Day From A Server

- Communication with an NTP server uses UDP
- To send an NTP message, a sender must know
  - The Internet address of a computer running an NTP server
  - The UDP port protocol number that the NTP server uses
  - A local UDP protocol port number that can be used
- Either DHCP returns the IP address for an NTP server or the code uses the address given by constant *TIMESERVER*
- The local and remote protocol port numbers to use are given by constants
  - Constant *TIMELPORT* defines a local UDP protocol port number to use
  - Constant *TIMERPORT* define the protocol port number for an NTP server (123)

Obtaining The Time Of Day From A Server (continued)

- Steps taken to obtain the current time
  - Call *udp_register* to obtain a slot number
  - Form an NTP request message and use *udp_send* to send the request to the server
  - Call *udp_recv*, specifying a maximum wait time of TIMETIMEOUT milliseconds
  - Call *udp_release* to release the slot
  - If a timeout occurred, return SYSERR; otherwise, store the time the system booted and return the current time
- Note:
  - Function *getutime* always returns *OK* or *SYSERR*
  - An argument specifies where to store the time of day if successful

Getutime: A Function That Uses UDP (Part 1)

```
/* getutime.c - getutime */

#include <xinu.h>
#include <stdio.h>

/*------------------------------------------------------------------------
 * getutime  -  Obtain time in seconds past Jan 1, 1970, UCT (GMT)
 *------------------------------------------------------------------------
 */
status  getutime(
          uint32  *timvar               /* Location to store the result */
        )
{
        uint32  now;                    /* Current time in xinu format  */
        int32   retval;                 /* Return value from call       */
        uid32   slot;                   /* Slot in UDP table            */
        struct  ntp {                   /* Format of an NTP message     */
                byte    livn;           /* LI:2 VN:3 and mode:3 fields  */
                byte    strat;          /* Stratum                      */
                byte    poll;           /* Poll interval                */
                byte    precision;      /* Precision                    */
                uint32  rootdelay;      /* Root delay                   */
                uint32  rootdisp;       /* Root dispersion              */
                uint32  refid;          /* Reference identifier         */
                uint32  reftimestamp[2];/* Reference timestamp          */
```

Getutime: A Function That Uses UDP (Part 2)

```
        uint32  oritimestamp[2];/* Originate timestamp          */
        uint32  rectimestamp[2];/* Receive timestamp            */
        uint32  trntimestamp[2];/* Transmit timestamp           */
} ntpmsg;

if (Date.dt_bootvalid) {        /* Return time from local info  */
        *timvar = Date.dt_boot + clktime;
        return OK;
}

/* Verify that we have obtained an IP address */

if (getlocalip() == SYSERR) {
        return SYSERR;
}

/* If the DHCP response did not contain an NTP server address   */
/*      use the default server                                  */

if (NetData.ntpserver == 0) {
        if (dnslookup(TIMESERVER, &NetData.ntpserver) == SYSERR) {
                return SYSERR;
        }
}
```

Getutime: A Function That Uses UDP (Part 3)

```
/* Contact the time server to get the date and time */

slot = udp_register(NetData.ntpserver, TIMERPORT, TIMELPORT);
if (slot == SYSERR) {
        fprintf(stderr,"getutime: cannot register a udp port %d\n",
                                TIMERPORT);
        return SYSERR;
}

/* Send a request message to the NTP server */

memset((char *)&ntpmsg, 0x00, sizeof(ntpmsg));
ntpmsg.livn = 0x1b;     /* Client request, protocol version 3 */
retval = udp_send(slot, (char *)&ntpmsg, sizeof(ntpmsg));
if (retval == SYSERR) {
        fprintf(stderr,"getutime: cannot send to the server\n");
        udp_release(slot);
        return SYSERR;
}

/* Read the response from the NTP server */

retval = udp_recv(slot, (char *) &ntpmsg, sizeof(ntpmsg), TIMETIMEOUT);
```

Getutime: A Function That Uses UDP (Part 4)

```
        if ( (retval == SYSERR) || (retval == TIMEOUT) ) {
                udp_release(slot);
                return SYSERR;
        }
        udp_release(slot);

        /* Extract the seconds since Jan 1900 and convert */

        now = ntim2xtim( ntohl(ntpmsg.trntimestamp[0]) );
        Date.dt_boot = now - clktime;
        Date.dt_bootvalid = TRUE;
        *timvar = now;
        return OK;
}
```

- Notes
  - Only a few lines of code call the network functions
  - The Internet protocols send integers in *network byte order*, and function *ntohl* converts from **n**etwork byte order **to h**ost byte order for a **l**ong integer

The ICMP Interface (For Ping)

- Follows the same approach as UDP
- An application
  - Calls *icmp_register* to register the remote address and receive a descriptor
  - Generates an ICMP request packet and calls *icmp_send* to send the packet
  - Calls *icmp_recv* to receive a reply, specifying a timeout
  - Handles the reply, if a valid reply was received
  - Calls *icmp_release* to release the registered endpoint
  - Either reports success or an error, if the request timed out

The Process Model
For Network Code

DHCP, Network Processes, And Delayed Use

- A computer uses DHCP at startup to obtain an IP address and related information
  - DHCP is only used once (i.e., it is only run during startup)
  - A DHCP message is sent using UDP (i.e., DHCP uses the UDP interface)
  - Sending a UDP message normally requires the sender to know its IP address
- How can a computer send a DHCP message *before* the computer has an IP address?
- Answer: the computer
  - Uses an all-0s IP address as the sender's address (0.0.0.0 in dotted decimal)
  - Sends its initial DHCP request to a special all-1s IP broadcast address (255.255.255.255 in dotted decimal)
- The resulting packet is broadcast across the local network and a DHCP server responds without needing the computer's IP address

DHCP, Network Processes, And Delayed Use (continued)

- An interesting process coordination problem arises with DHCP
  - To use DHCP, network processes must be running (explained later)
  - Network processes are not started until late in the bootstrap sequence
- Our solution: delay using DHCP until an application needs to use the Internet
  - Start the network processes at the end during system initialization
  - Wait until the first time an application calls *getlocalip* to obtain the local IP address, use the call to trigger sending a DHCP request, obtain the reply and store the IP address locally
  - Note: successive calls to *getlocalip* obtain the stored value locally
- In essence, DHCP runs as a side effect of requesting the local IP address

The Need For Network Processes

- Most operating system functions are merely called by application processes
- Network code requires independent processes to ensure that an incoming packet is handled, even if no application is waiting for the packet
- Examples
  - When a ping request arrives from another computer, the receiver must generate and send a reply even if no application is running
  - When an ARP request arrives from another computer, the receiver must send a reply before Internet packets can arrive from the computer
- The device driver for a network hardware device allows processes to *read* incoming packets and *write* outgoing packets, but does not interpret the packets or send replies
- Consequence: a process must always be waiting to read and handle incoming packets

Xinu's Network Input Process

- To handle asynchronous packet arrivals, Xinu keeps a *network input process* running at all times
- The network input process repeatedly
  - Calls *read* on the ETHER device to block and wait for the next incoming packet
  - Handles the packet (e.g., if the packet contains UDP, the network input process calls *udp_in*)
- Sending an ARP replay is trivial — the network input process calls a function that forms a reply and *writes* it to the *ETHER* device
- Unfortunately, sending a ping reply causes a problem

The Problem With Ping Replies

- Ping replies travel in an IP packet
- Sending an outgoing IP packet *may* require an ARP exchange with another computer
- The steps are
  - Start with an outgoing IP packet
  - While holding the outgoing packet, send an ARP request to find the receiver's Ethernet address
  - Receive an ARP reply
  - Add the information in the ARP reply to the original IP packet and send it
- The problem: if the network input process blocks to wait until the needed information arrives, a deadlock will result because no process will be running to read the ARP reply packet from the ETHER device

Avoiding Deadlock

- To avoid deadlock

The network input process must never call a function that blocks to wait for a reply.

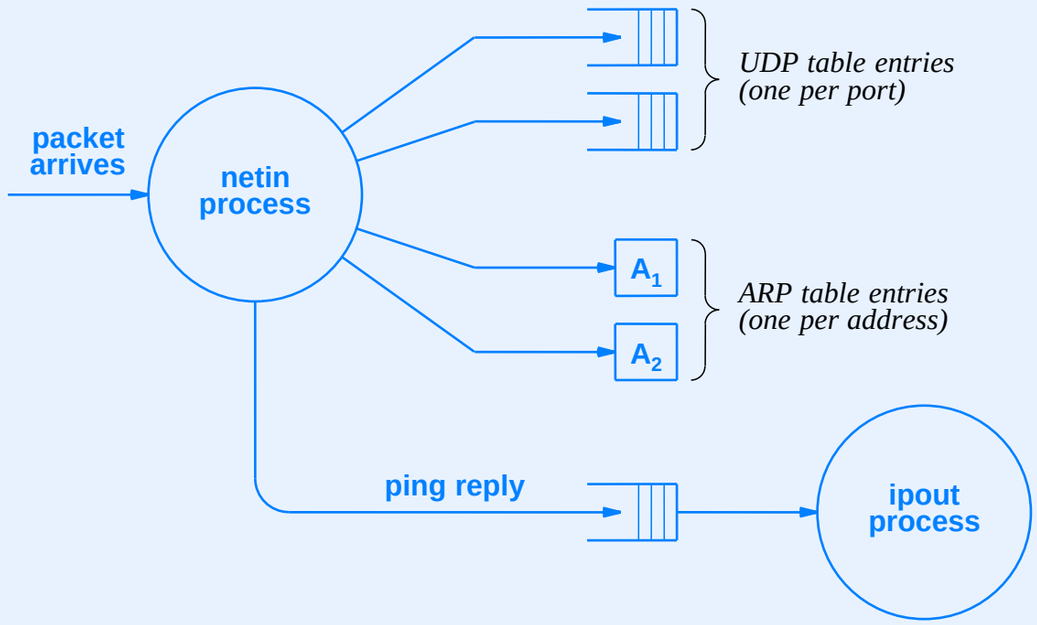
- To prevent the network input process from blocking, Xinu uses a separate IP output process, and arranges for the network input process to deposit outgoing IP packets on a queue for the output process to handle
- The IP output process can block waiting for an ARP reply because the network input process remains running

Communication Between The Network Input And IP Output Processes

- Uses a queue of packets and a semaphore for coordination
- The output process repeatedly
  - Waits on the semaphore until an outgoing IP packet is placed in the queue
  - Performs the ARP exchange if necessary, possibly blocking to wait until a reply has been received and the information extracted
  - Uses the information to send the IP packet
- Note: it is not important that you understand the protocol details, but it important that you realize that protocols dictate the process structure that is needed

Simplified Illustration Of The Xinu Network Process Model

- *Netin* handles incoming UDP and ARP packets



- *Netin* enqueues ping replies for *ipout*, thereby preventing *netin* from blocking

Buffering Incoming Packets

The Need For Packet Queues

- The *netin* process has a high priority
- An application process may have a low priority
- Consequences
  - An application that is waiting for a packet may not execute immediately after the packet arrives
  - A second packet may arrive for a given application before the first packet has been handled
- To accommodate delayed processing, Xinu uses packet queues to absorb a small burst of packets without discarding any
- Note: the above only applies to UDP and ICMP because ARP packets are processed immediately by the netin process

The ARP Cache And Cache Timeout Processing

- The ARP protocol specifies that the network code must keep a cache of recent address bindings
- Entries in the cache should be removed after 10 minutes
- Is an additional process needed to implement ARP cache timeout?
- Using an additional process has disadvantages
  - More context switching overhead
  - Uses system resources, such as stack space, with little real value

The Xinu Approach To Cache Timeout

- To avoid having an extra process handle cache timeout, Xinu uses a trick
- When storing an entry in the cache, Xinu stores the current time in a timestamp field in the entry
- Whenever searching the cache, the code examines the timestamp field in each entry, and removes the entry if the time has expired
- The approach works well for an ARP cache because the cache is only expected to contain a few entries, and the search proceeds sequentially

Choosing Process Priorities

- The easy part: network processes should run at higher priority than user processes
- The hard part: deciding whether the input process or output process should have higher priority
- The choice is not clear
  - If the input process has higher priority, output may become a bottleneck
  - If the output process has a higher priority, incoming packets may not be handled quickly

Summary

- Networking is an essential part of any operating system, and three aspects are important to OS designers
- The interface to protocols (Xinu's is similar to the socket interface)
  - Register to specify a remote endpoint and obtain a descriptor to use
  - Use the descriptor to send and receive data
  - Release the descriptor
- The process structure for network processes
  - Depends on protocols
  - An input and output process are needed
- Packet buffers (queues to hold packets)
  - Needed because packets arrive in bursts (typically, a small queue suffices)

Questions?