Network Programming

Dr. Thaier Hayajneh

Computer Engineering Department

Sockets 2

Necessary Background Information: POSIX data types

`int8_t`	signed 8bit int
`uint8_t`	unsigned 8 bit int
`int16_t`	signed 16 bit int
`uint16_t`	unsigned 16 bit int
`int32_t`	signed 32 bit int
`uint32_t`	unsigned 32 bit int

More POSIX data types

`sa_family_t`

address family of socket address structure

`socklen_t`

length of socket address structure, (uint32_t)

`in_addr_t`

IPv4 address, normally (uint32_t)

`in_port_t`

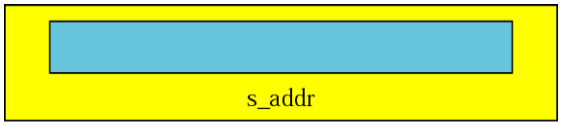
TCP or UDP port number, normally (uint36_t)

Defined by including <sys/types.h> header

Internet address structure

- IPv4 address is defined as a structure in C.
- in_addr contains only one field called s_addr
- The structure holds an IP address as a 32-bit binary number

in_addr just provides a name for the 'C' type associated with IP addresses.



in_addr

```
struct      in_addr
{
            in_addr_t   s_addr ;
} ;
```

SOCKETS

struct sockaddr_in (IPv4)

- struct sockaddr_in {
  uint8_t sin_len; /*unsigned 8 bit integer*/
  sa_family_t sin_family; /*AF_INET*/
  in_port_t sin_ port ; /* 16 bit TCP or UDP port number */
  struct in_addr sin_addr; /* 32 bit IPv4 address */
  char sin _zero[8]; /*unused*/
  }

A special kind of sockaddr structure

Defined by including <netinet/in.h> header

Generic Socket Address Structure

•*A socket address structure always passed by reference when passed as an argument to any socket function*

•How to declare the pointer that is passed?

•Define a generic socket address structure

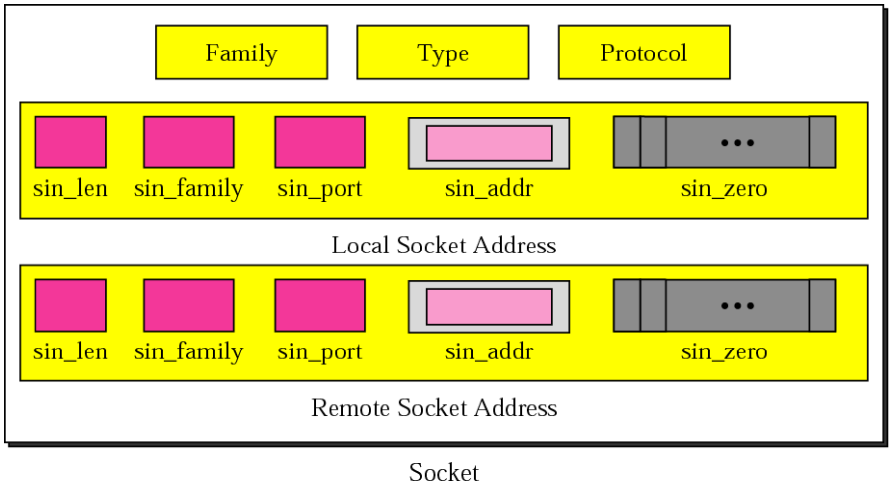
```
struct sockaddr {
    uint8_t      sa_len;   /*unsigned 8 bit integer*/
    sa_family_t  sa_family; /*AF_INET*/
    char         sa_data[14] ; /* protocol specific address*/
}
```

sa_family specifies the address type.
sa_data specifies the address value.

Prototype for bind

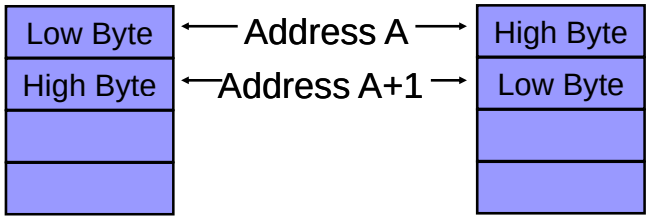
```
int bind (int, struct sockaddr * socklen_t)
struct sockaddr_in serv;
bind (sockfd, (struct sockaddr *) &serv,sizeof(serv));
```

Socket structure



Byte Ordering

- Different computer architectures use different byte ordering to represent multibyte values.
- 16 bit integer:



Byte Ordering Functions

•Two ways to store 2 bytes (16-bit integer) in memory

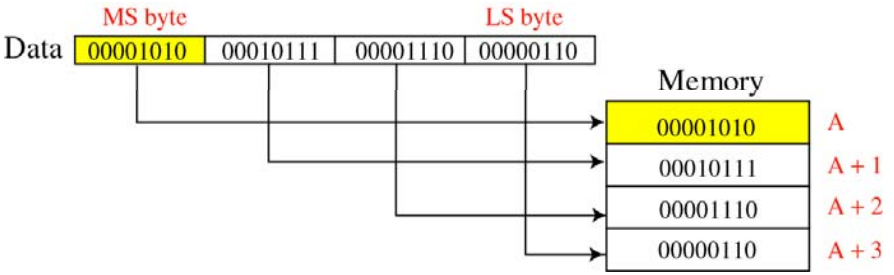
- ➢Low-order byte at starting address → little-endian byte order
- ➢High-order byte at starting address → big-endian byte order

•Byte order used by a given system known as *host byte order*

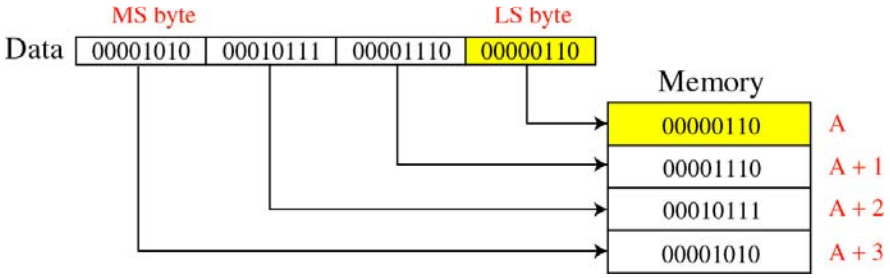
•Network programmers use *network byte order*

•Internet protocol uses big-endian byte ordering for integers (port number and IP address)

Big-endian byte order



Little-endian byte order



Byte Ordering

Little-Endian

Low Byte	High Byte
Addr A	**Addr A+1**

IBM 80x86
DEC VAX
DEC PDP-11

- Big-Endian

High Byte	Low Byte
Addr A	**Addr A+1**

- IBM 370
- Motorola 68000
- Sun

Byte Order and Networking

- Suppose a Big Endian machine sends a 16 bit integer with the value 2:

0000000000000010

- A Little Endian machine will think it got the number 512:

0000001000000000

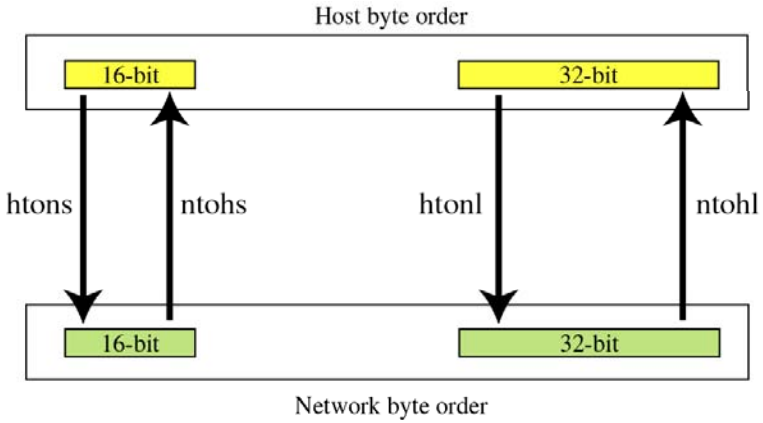
Network Byte Order

- Conversion of application-level data is left up to the presentation layer.
- But hold on !!! How do lower level layers communicate if they all represent values differently ? (data length fields in headers)
- A fixed byte order is used (called *network byte order*) for all control data.

Network Byte Order

- All values stored in a **`sockaddr_in`** must be in network byte order.
  - **`sin port`** a TCP/IP port number.
  - **`sin_addr`** an IP address.

Bite-order transformation



Network Byte Order Functions

'**h**' : host byte order '**n**' : network byte order

'**s**' : short (16bit) '**l**' : long (32bit)

```
uint16_t htons(uint16_t);
uint16_t ntohs(uint_16_t);

uint32_t htonl(uint32_t);
uint32_t ntohl(uint32_t);
```


Byte Ordering Functions

```
#include     "unp.h"
int main(int argc, char **argv)
{
        union {
          short  s;
          char   c[sizeof(short)];
        } un;

        un.s = 0x0102;
        printf("%s: ", CPU_VENDOR_OS);
        if (sizeof(short) == 2) {
                if (un.c[0] == 1 && un.c[1] == 2)
                        printf("big-endian\n");
                else if (un.c[0] == 2 && un.c[1] == 1)
                        printf("little-endian\n");
                else
                        printf("unknown\n");
        } else
                printf("sizeof(short) = %d\n", sizeof(short));

        exit(0);
}
```

•Sample program to figure out little-endian or big-endian machine

Byte Manipulation Functions

```
#include <strings.h>
void bzero (void *dest, size_t nbytes);
```

// sets specified number of bytes to 0 in the destination

We may use it to initialize the socket address structure to zero

```
void bcopy (const void *src, void * dest, size_t nbytes);
```

// moves specified number of bytes from source to destination

```
void bcmp (const void *ptr1, const void *ptr2,size_t nbytes)
```

//compares two arbitrary byte strings, return value is zero if two byte strings are identical, otherwise, nonzero