# Socket APIs

**Computer Network Programming**

## Outline

- Socket address structures
- Value-result arguments
- Byte ordering and manipulation functions
- Address conversion functions: *inet_aton*, *inet_addr*, *inet_ntoa*, *inet_pton*, *inet_ntop*, *sock_ntop*
- Stream socket I/O functions: *readn*, *writen*, *readline*
- File descriptor testing function: is fd type

## Socket Address Structures

- Most socket functions require a pointer to a socket address structure as an argument.
- Each supported protocol suite defines its own socket address structure.
- The names of these structures begin with sockaddr_ and end with a unique suffix for each protocol suite.

## IPv4 Socket Address Structure

- An IPv4 socket address structure, commonly called an "Internet socket address structure," is named sockaddr_in and is defined by including the <netinet/in.h> header.

# sockaddr_in

- **The Internet (IPv4) socket address structure:**

```
struct in_addr {
  in_addr_t   s_addr;            /* 32-bit IPv4 address */
                                 /* network byte ordered */
};

struct sockaddr_in {
  uint8_t         sin_len;       /* length of structure (16) */
  sa_family_t     sin_family;    /* AF_INET */
  in_port_t       sin_port;      /* 16-bit TCP or UDP port number */
                                 /* network byte ordered */
  struct in_addr  sin_addr;      /* 32-bit IPv4 address */
                                 /* network byte ordered */
  char            sin_zero[8];   /* unused */
};
```

## sockaddr_in

- There are several points we need to make about socket address structures in general using this example:
  - The length member, sin_len, was added with BSD-Reno, when support for the OSI protocols.
  - The first member was sin_family, which was historically an unsigned short.
  - The datatype that we show, uint8_t, is typical, and POSIX-compliant systems provide datatypes.

## POSIX specification requires

- The POSIX specification requires only three members in the structure:
  - sin_family,
  - sin_addr, and
  - sin_port.
- It is acceptable for a POSIX-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure.
- Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size

## POSIX data types

- The in_addr_t datatype must be an unsigned integer type of at least 32 bits,
- in_port_t must be an unsigned integer type of at least 16 bits,
- sa_family_t can be any unsigned integer type.
- The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported.

| Datatype | Description | Header |
|---|---|---|
| int8_t | signed 8-bit integer | <sys/types.h> |
| uint8_t | unsigned 8-bit integer | <sys/types.h> |
| int16_t | signed 16-bit integer | <sys/types.h> |
| uint16_t | unsigned 16-bit integer | <sys/types.h> |
| int32_t | signed 32-bit integer | <sys/types.h> |
| sa_family_t | address family of socket addr struct | <sys/types.h> |
| socklen_t | length of socket addr struct, uint32_t | <sys/types.h> |
| in_addr_t | IPv4 address, normally uint32_t | <sys/types.h> |
| in_port_t | TCP or UDP port, normally uint16_t | <sys/types.h> |

## IPv4 socket address structure

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in network byte order. We must be cognizant of this when using these member.
- The 32-bit IPv4 address can be accessed in two different ways.
  - For example, if serv is defined as an Internet socket address structure, then serv.sin_addr references the 32-bit IPv4 address as an in_addr structure, while serv.sin_addr.s_addr references the same 32-bit IPv4 address as an in_addr_t (typically an unsigned 32-bit integer).
  - The sin_zero member is unused, but we always set it to 0 when filling in one of these structures. By convention, we always set the entire structure to 0 before filling it in, not just the sin_zero member.

## Generic Socket Address Structure

- A socket address structures is always passed by reference when passed as an argument to any socket functions.
- But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.
- A problem arises in how to declare the type of pointer that is passed.
- In ANSI C, the solution is simple: void * is the generic pointer type.
- Define a generic socket address structure in the <sys/socket.h> header.

## sockaddr

- **The generic socket address structure:**

```
struct sockaddr {
  uint8_t      sa_len;
  sa_family_t  sa_family;     /* address family: AF_xxx value */
  char         sa_data[14];   /* protocol-specific address */
};
```

- The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the bind function:
  - int bind(int, struct sockaddr *, socklen_t);

## Generic socket address structure

- This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure.
- For example,

```
struct sockaddr_in  serv;      /* IPv4 socket address structure */

/* fill in serv{} */

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

## IPv6 Socket Address Structure

- The IPv6 socket address is defined by including the <netinet/in.h> header.

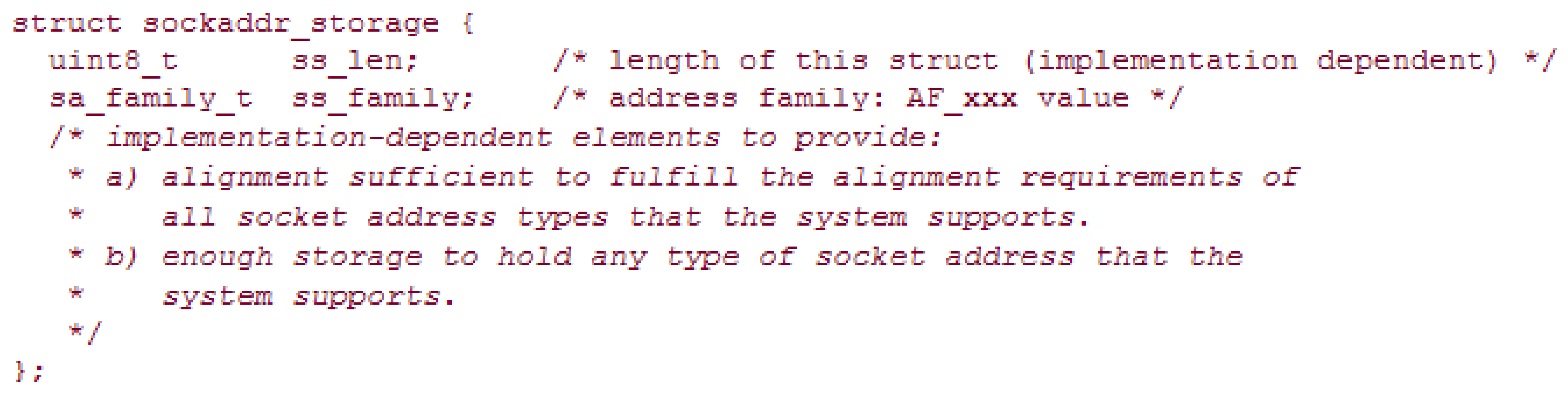
```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t         sin6_len;      /* length of this struct (28) */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* flow information, undefined */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
  uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};
```

## New Generic Socket Address Structure

- A new generic socket address structure was defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct sockaddr.
- Unlike the struct sockaddr, the new struct sockaddr_storage is large enough to hold any socket address type supported by the system.

## New Generic Socket Address Structure

- The sockaddr_storage structure is defined by including the <netinet/in.h> header.



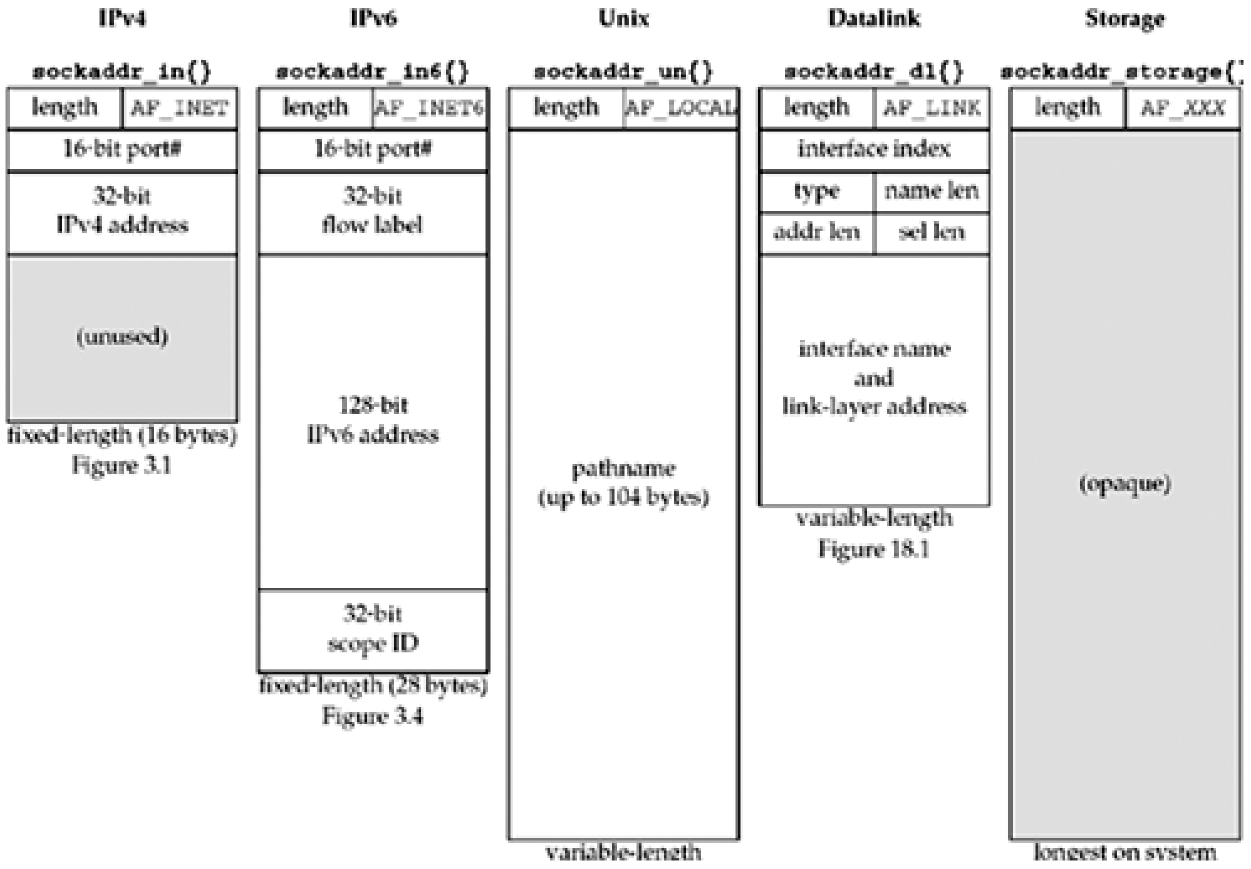
```
struct sockaddr_storage {
  uint8_t      ss_len;       /* length of this struct (implementation dependent) */
  sa_family_t  ss_family;    /* address family: AF_xxx value */
  /* implementation-dependent elements to provide:
   * a) alignment sufficient to fulfill the alignment requirements of
   *    all socket address types that the system supports.
   * b) enough storage to hold any type of socket address that the
   *    system supports.
   */
};
```

## Comparison of Various Socket Address Structures

- Here is a comparison of the five socket address structures that we will encounter in this text: IPv4, IPv6, Unix domain, datalink, and storage.

## Comparison of Various Socket Address Structures

In this figure, we assume that the socket address structures all contain a one-byte length field, that the family field also occupies one byte, and that any field that must be at least some number of bits is exactly that number of bits.



## Comparison of Various Socket Address Structures

- Two of the socket address structures are fixed-length, while the Unix domain structure and the datalink structure are variable-length.
- To handle variable-length structures, whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument.

## Comparison of Various Socket Address Structures

- The sock addr_un structure itself is not variable-length, but the amount of information the pathname within the structure is variable-length.
- When passing pointers to these structures, we must be careful how we handle the length field, both the length field in the socket address structure itself (if supported by the implementation) and the length to and from the kernel.

## Comparison of Various Socket Address Structures

- The length field was added to all the socket address structures with the BSD Reno release.
- Had the length field been present with the original release of sockets, there would be no need for the length argument to all the socket functions: the third argument to bind and connect, for example. Instead, the size of the structure could be contained in the length field of the structure.

## Comparison of Various Socket Address Structures

- The first four socket functions that pass a socket address structure from the process to the kernel, bind, connect, sendto, and sendmsg, all go through the sockargs function in a Berkeley-derived implementation (TCPv2).
- This function copies the socket address structure from the process and explicitly sets its sin_len member to the size of the structure that was passed as an argument to these four functions. The five socket functions that pass a socket address structure from the kernel to the process, accept, recvfrom, recvmsg, getpeername, and getsockname, all set the sin_len member before returning to the process.

# Socket Address Structures: IPv4, Generic, IPv6

```
struct in_addr {
  in_addr_t         s_addr;          /* 32-bit IPv4 address, network byte order */        };
struct sockaddr_in {
  uint8_t           sin_len;         /* length of structure */
  sa_family_t       sin_family;      /* AF_INET */
  in_port_t         sin_port;        /* 16-bit port#, network byte order */
  struct in_addr    sin_addr;        /* 32-bit IPv4 address, network byte order */
  char              sin_zero[8];     /* unused */                                          };
struct sockaddr {                    /* only used to cast pointers */
  uint8_t           sa_len;
  sa_family         sa_family;       /* address family: AF_xxx value */
  char              sa_data[14];     /* protocol-specific address */                       };
struct in6_addr {
  unit8_t           s6_addr[16];     /* 128-bit IPv6 address, network byte order */ };
struct sockaddr_in6 {
  uint8_t           sin6_len;        /* length of this struct [24] */
  sa_family         sin6_family;     /* AF_INET6 */
  in_port_t         sin6_port;       /* port#, network byte order */
  uint32_t          sin6_flowinfo;   /* flow label and priority */
  struct in6_addr   sin6_addr;       /* IPv6 adress, network byte order */    };
```

## Socket Address Structures: IPv4, Generic, IPv6

- Socket functions are defined to take a pointer to the generic socket address structure, e.g.,

```
int bind(int, struct sockaddr *, socklen_t);
```

- Any calls to these functions must do casting
- For IPv4:

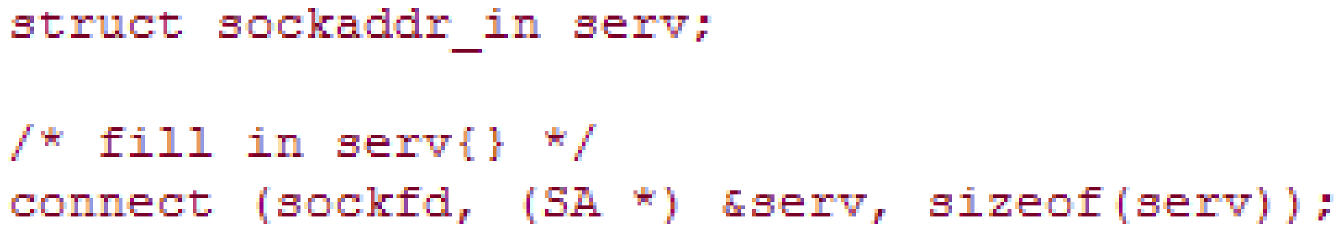
```
struct sockaddr_in  serv;  /* IPv4 socket address structure */
…
bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

$\underbrace{\texttt{(struct sockaddr *) \&serv}}_{\text{casting}}$

## VALUE-RESULT ARGUMENT

## Socket address structure passed from process to kernel

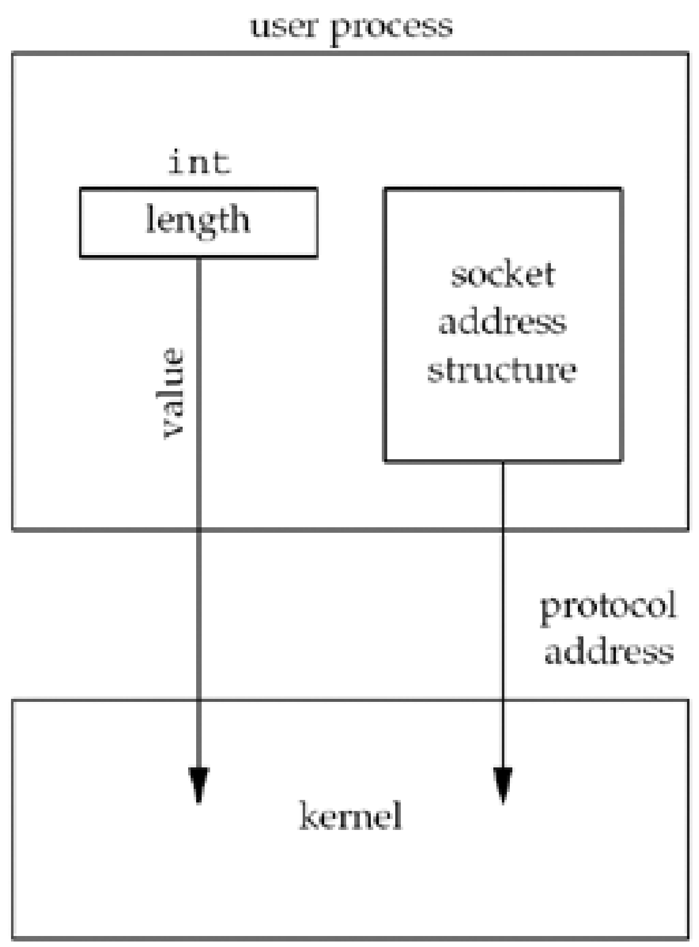
- Three functions, bind, connect, and sendto, pass a socket address structure from the process to the kernel.
- One argument to these three functions is the pointer to the socket address structure and another argument is the integer size of the structure, as in



```
struct sockaddr_in serv;

/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

## Socket address structure passed from process to kernel

- Since the kernel is passed both the pointer and the size of what the pointer points to, it knows exactly how much data to copy from the process into the kernel.



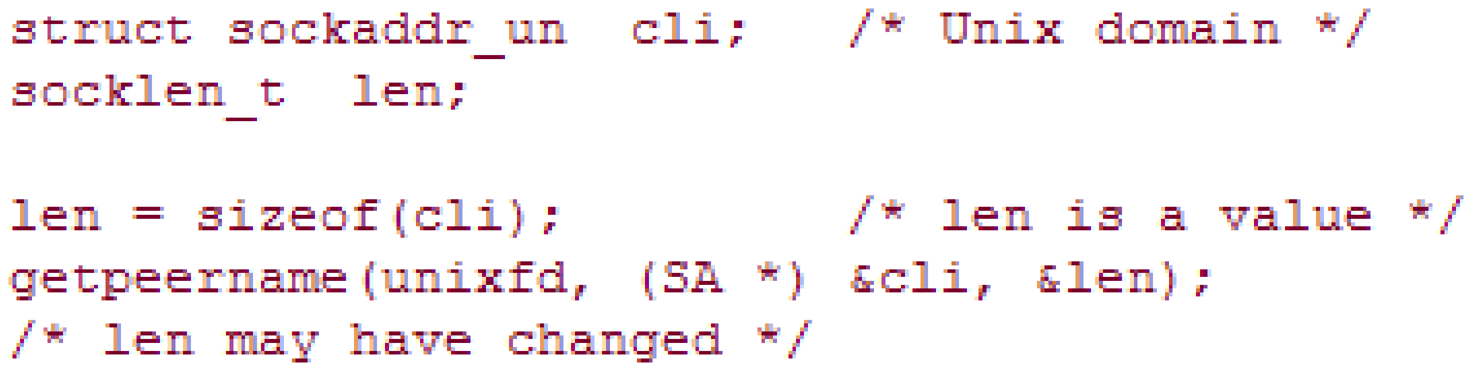
## Socket address structure passed from kernel to process

- Four functions,
  - accept,
  - recvfrom,
  - getsockname, and
  - getpeername,

  pass a socket address structure from the kernel to the process, the reverse direction from the previous scenario.

## Socket address structure passed from kernel to process

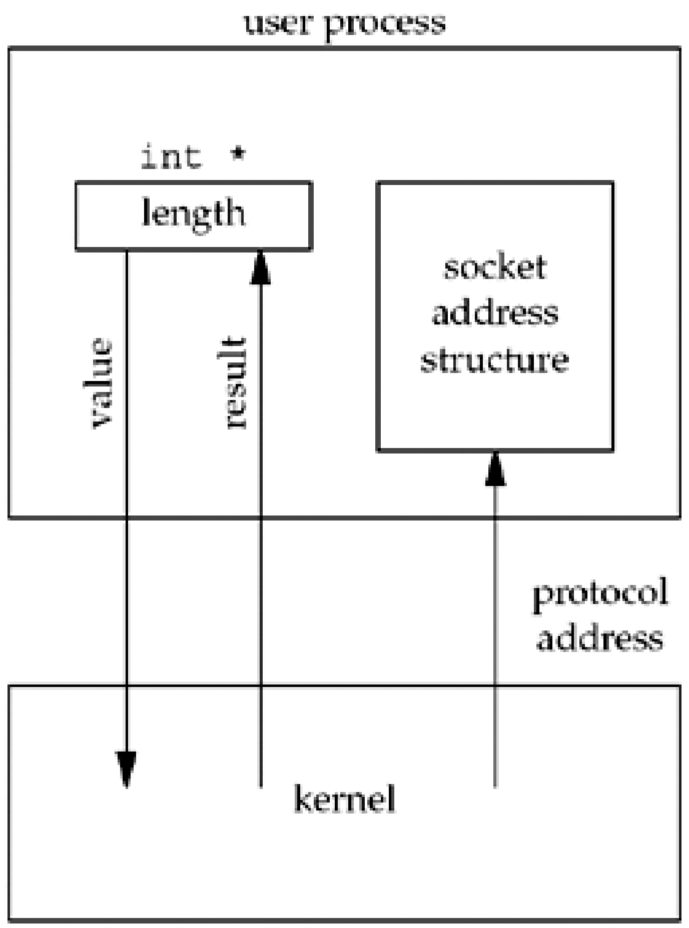
- Two of the arguments to these four functions are the pointer to the socket address structure along with a pointer to an integer containing the size of the structure.



```
struct sockaddr_un  cli;   /* Unix domain */
socklen_t  len;

len = sizeof(cli);         /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```

# Socket address structure passed from kernel to process

- The reason that the size changes from an integer to be a pointer to an integer is because the size is both a value when the function is called and a result when the function.
- This type of argument is called a value-result argument.



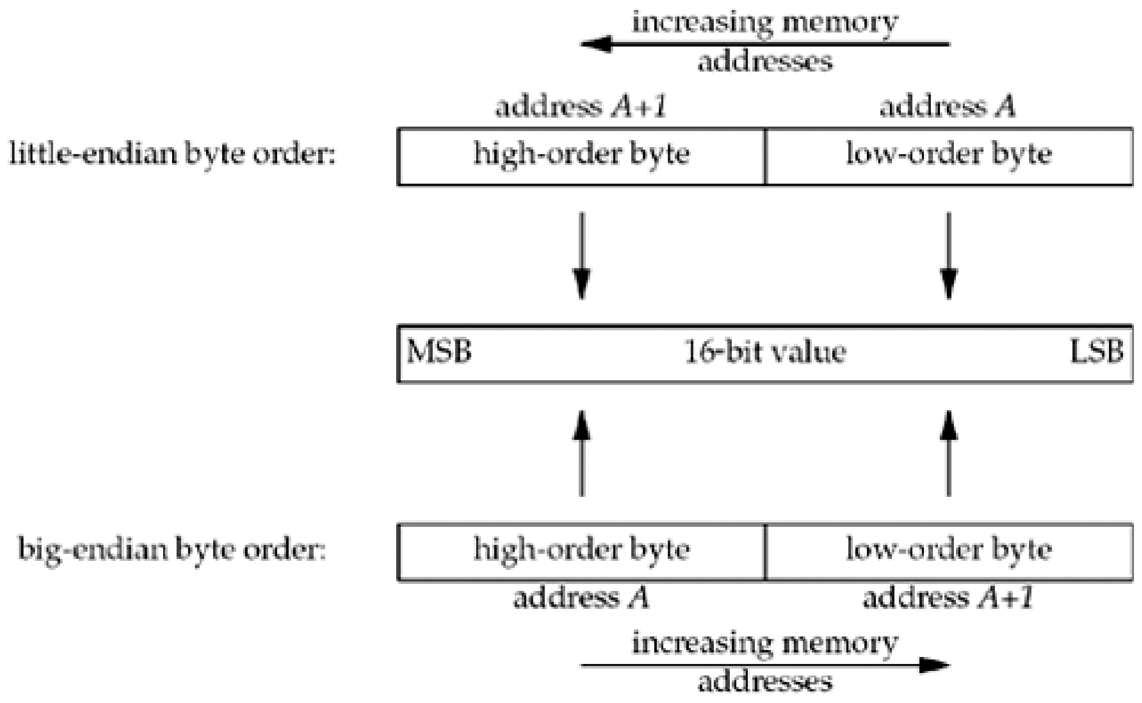
## BYTE ORDERING FUNCTIONS

## Byte Ordering Functions

- Consider a 16-bit integer that is made up of 2 bytes.
- There are two ways to store the two bytes in memory:
  - with the low-order byte at the starting address, known as little-endian byte order, or
  - with the high-order byte at the starting address, known as big-endian byte order.
- The terms “little-endian 低位元組的放在低位址處” and “big-endian高位元組的放在低位址處" indicate which end of the multibyte value, the little end or the big end, is stored at the starting address of the value.

## Byte Ordering Functions

- This figure shows increasing memory addresses going from right to left in the top, and from left to right in the bottom.
- It also show the most significant bit (MSB) as the leftmost bit of the 16-bit value and the least significant bit (LSB) as the rightmost bit.



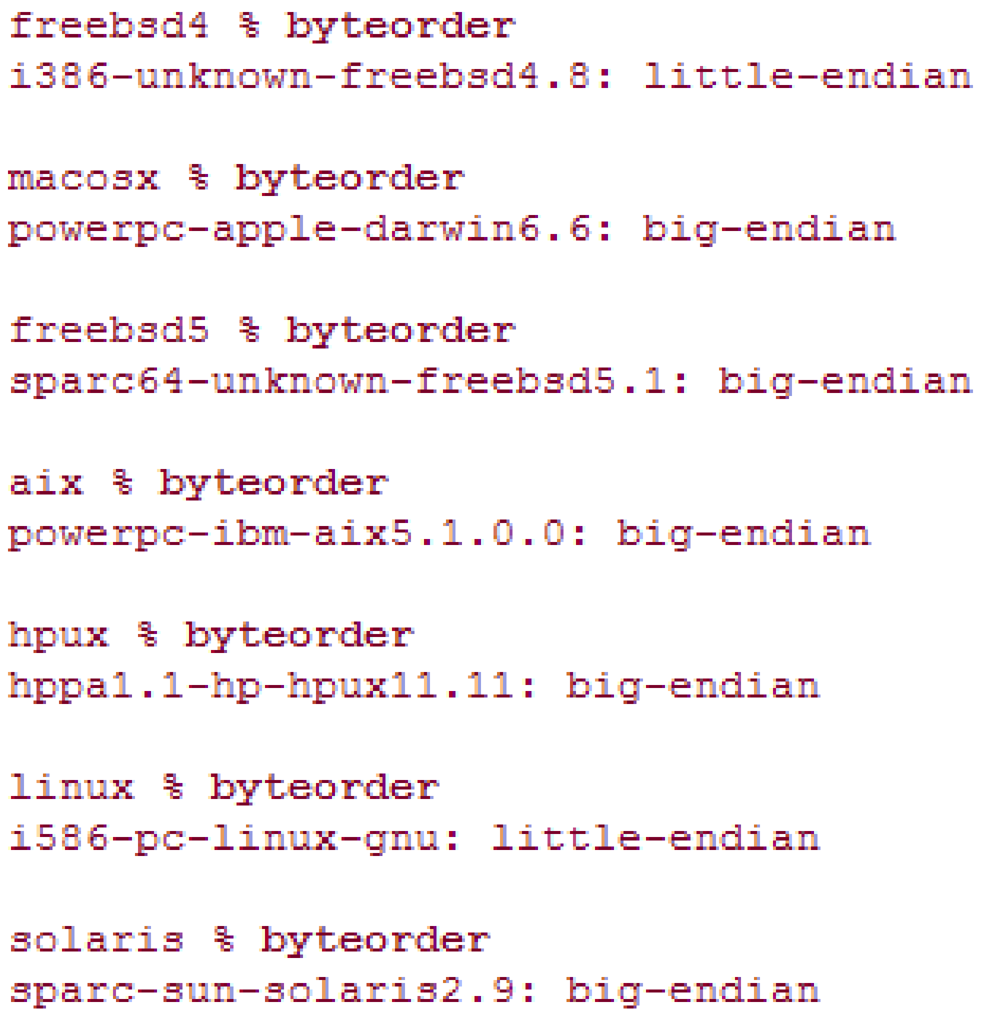
# intro/byteorder.c

- We refer to the byte ordering used by a given system as the host byte order.

```c
#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);
}
```

## Examples

- Some examples here in the output from this program when run on the various systems.



```
freebsd4 % byteorder
i386-unknown-freebsd4.8: little-endian

macosx % byteorder
powerpc-apple-darwin6.6: big-endian

freebsd5 % byteorder
sparc64-unknown-freebsd5.1: big-endian

aix % byteorder
powerpc-ibm-aix5.1.0.0: big-endian

hpux % byteorder
hppa1.1-hp-hpux11.11: big-endian

linux % byteorder
i586-pc-linux-gnu: little-endian

solaris % byteorder
sparc-sun-solaris2.9: big-endian
```

## The host byte order and network byte order

- Therefore converting between host byte order and network byte order.
- Four functions are used to convert between these two byte orders.

```
#include <netinet/in.h>
    uint16_t htons(uint16_t host16bitvalue) ;
    uint32_t htonl(uint32_t host32bitvalue) ;
```

- Both return: value in network byte order

```
    uint16_t ntohs(uint16_t net16bitvalue) ;
    uint32_t ntohl(uint32_t net32bitvalue) ;
```

- Both return: value in host byte order

## The bit ordering in Internet Standard

- Another important convention in Internet standards is bit ordering.
- In many Internet standards, you will see "pictures" of packets that look similar to the following (this is the first 32 bits of the IPv4 header from RFC 791):

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

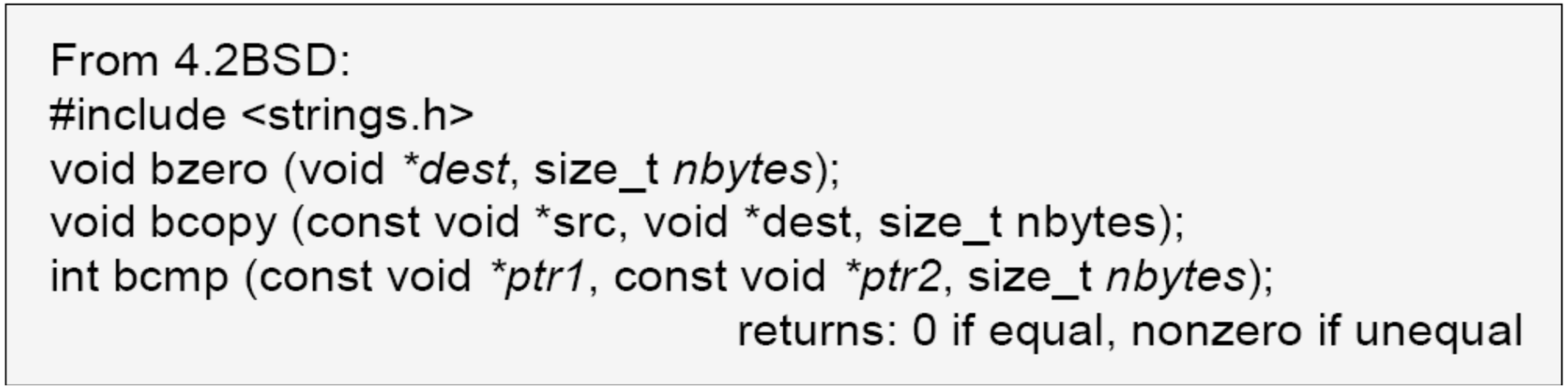
- This represents four bytes in the order in which they appear on the wire; the leftmost bit is the most significant.
- However, the numbering starts with zero assigned to the most significant bit. This is a notation that you should become familiar with to make it easier to read protocol definitions in RFCs.

## BYTE MANIPULATION FUNCTIONS

## BYTE MANIPULATION FUNCTIONS

- There are two groups of functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string.

## Byte Manipulation Functions: operating on multibyte fields



```
From 4.2BSD:
#include <strings.h>
void bzero (void *dest, size_t nbytes);
void bcopy (const void *src, void *dest, size_t nbytes);
int bcmp (const void *ptr1, const void *ptr2, size_t nbytes);
                                        returns: 0 if equal, nonzero if unequal
```

- bzero sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0.
- bcopy moves the specified number of bytes from the source to the destination.
- bcmp compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero.

## Byte Manipulation Functions: operating on multibyte fields

```
From ANSI C:
#include <string.h>
void *memset (void *dest, int c, size_t len);
void *memcpy (void *dest, const void *src, size_t nbytes);
int memcmp (const void *ptr1, const void *ptr2, size_t nbytes);
                                        returns: 0 if equal, nonzero if unequal
```

- memset sets the specified number of bytes to the value c in the destination.
- memcpy is similar to bcopy, but the order of the two pointer arguments is swapped.
- bcopy correctly handles overlapping fields, while the behavior of memcpy is undefined if the source and destination overlap.
- The ANSI C memmove function must be used when the fields overlap.

## INET_ATON, INET_ADDR, AND INET_NTOA FUNCTIONS

## Address Conversion Functions for Both IPv4 and IPv6

- Two groups of address conversion functions in this and following sections, they convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).

## Address Conversion Functions for IPv4 Only

```
#include <arpa/inet.h>
int inet_aton (const char *strptr, struct in_addr *addrptr);
        returns: 1 if string is valid, 0 on error
in_addr_t inet_addr (const char *strptr);
        returns: 32-bit binary IPv4 addr, INADDR_NONE if error
char *inet_ntoa (struct in_addr inaddr);
        returns: pointer to dotted-decimal string
```

- inet_aton, inet_ntoa, and inet_addr convert an IPv4 address from a dotted-decimal string (e.g., "206.168.112.96") to its 32-bit network byte ordered binary value.
- You will probably encounter these functions in lots of existing code.

## inet_aton

- int inet_aton (const char **strptr*, struct in_addr **addrptr*);
  - 輸入參數string包含ASCII表示的IP地址。
  - 輸出參數addr是將要用新的IP位址更新的結構。
  - 返回值：
    - 如果這個函數成功，函數的返回值非零，如果輸入位址不正確則會返回零。使用這個函數並沒有錯誤碼存放在errno中，所以它的值會被忽略。

## inet_addr

- in_addr_t inet_addr(const char* strptr);
  - 返回：若字串有效則將字串轉換為32位元二進位網路位元組序的IPV4地址，否則為INADDR_NONE

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

## inet_ntoa

- char *inet_ntoa (struct in_addr *inaddr*);
  - 將一個IP轉換成一個互聯網標準點分格式的字串。
- 返回值：
  - 如果正確，返回一個字元指標，指向一塊存儲著點分格式IP位址的靜態緩衝區（同一執行緒內共用此記憶體）；錯誤，返回NULL

## Address Conversion Functions for Both IPv4 and IPv6

#include <arpa/inet.h>
int inet_pton (int *family*, const char **strptr*, void **addrptr*);
    returns: 1 if OK, 0 if invalid presentation, -1 on error

const char *inet_ntop (int *family*, const void **addrptr*, char **strptr*, size_t *len*);
    returns: pointer to result if OK, NULL on error

- INET_ADDRSTRLEN = 16 (for IPv4 dotted-decimal),
- INET6_ADDRSTRLEN = 46 (for IPv6 hex string)

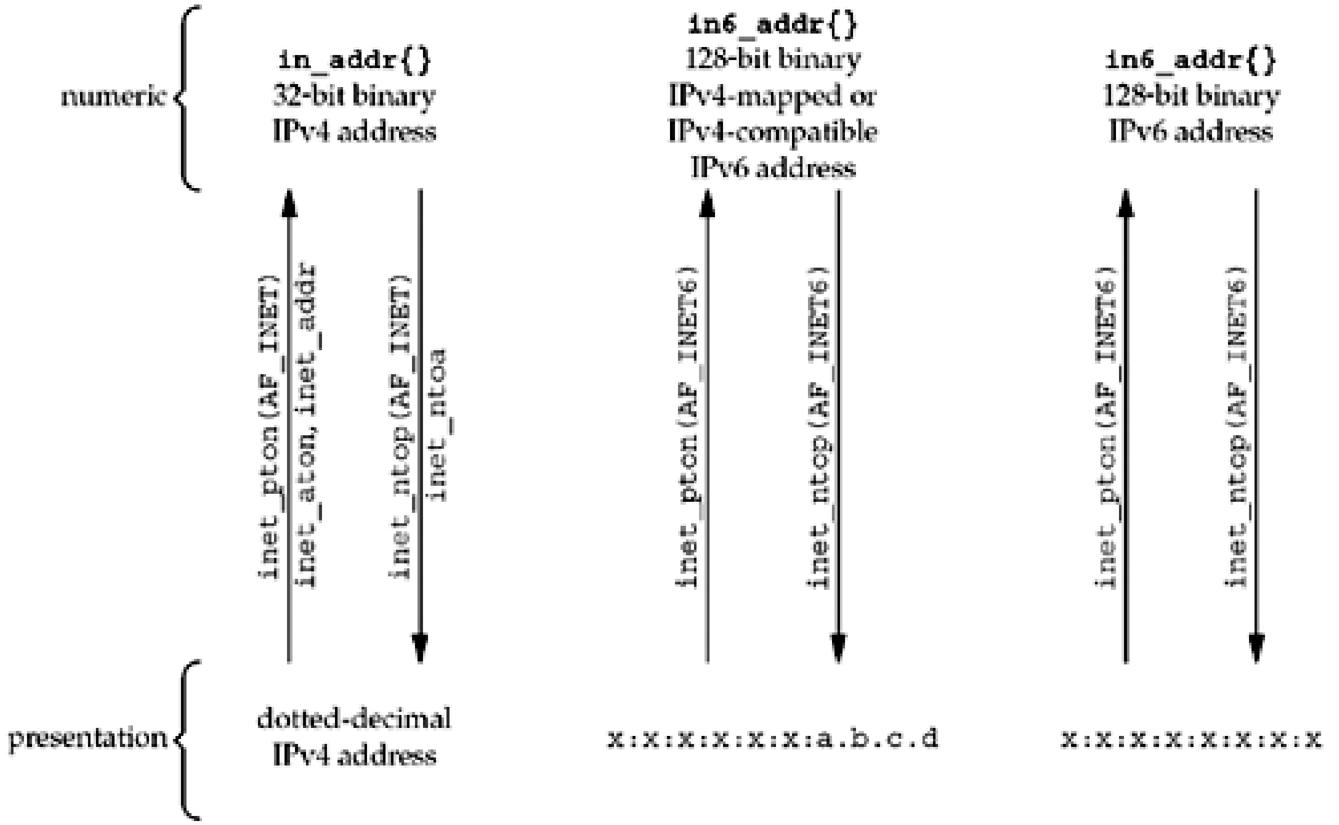
## inet_pton function

- inet_pton function tries to convert the string pointed to by strptr, storing the binary result through the pointer addrptr. If successful, the return value is 1. If the input string is not a valid presentation format for the specified family, 0 is returned.
  - 將 “十進位” －> “二進位整數”
  - int inet_pton(int af, const char *src, void *dst);
  - 轉換字串到網路位址，第一個參數af是地址，第二個參數*src是來源地址，第三個參數* dst接收轉換後的資料。

## inet_ntop function

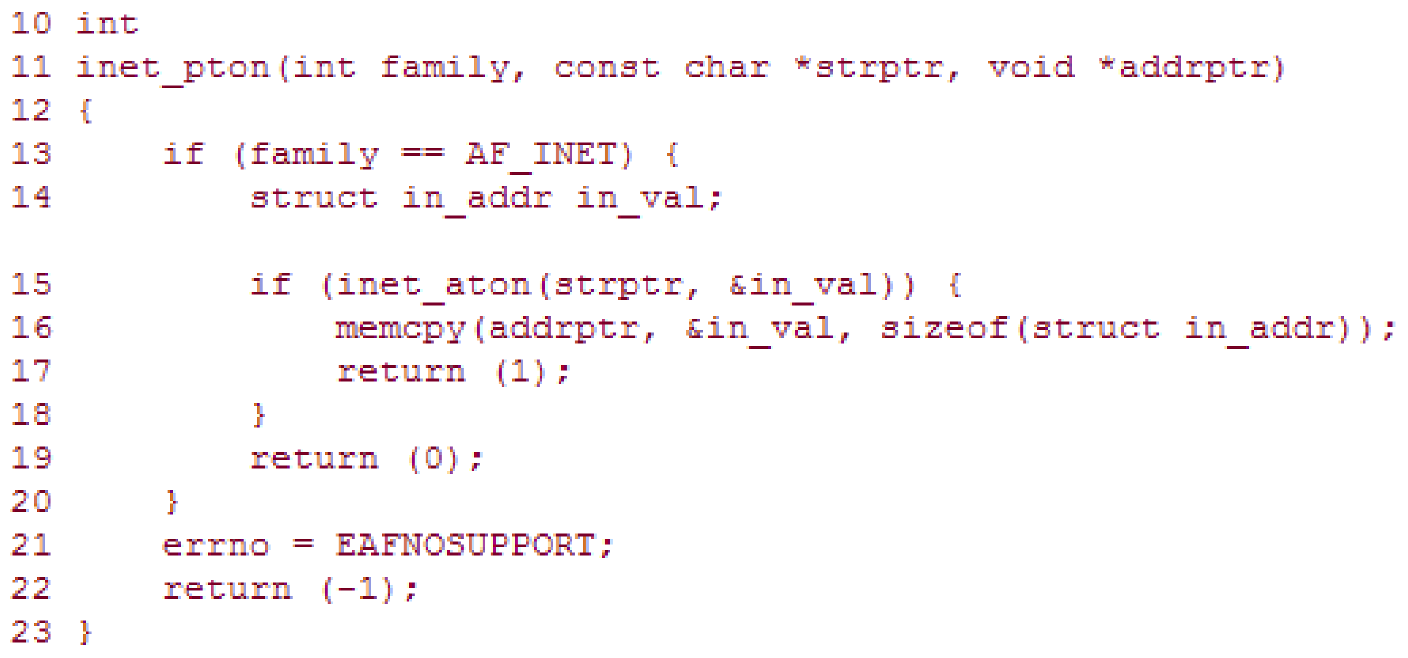
- inet_ntop does the reverse conversion, from numeric (addrptr) to presentation (strptr). The len argument is the size of the destination, to prevent the function from overflowing the caller's buffer. To help specify this size, the following two definitions are defined by including the <netinet/in.h> header:
- 這個函數轉換網路二進位結構到ASCII類型的地址，參數的作用和inet_pton 相同，只是多了一個參數socklen_t cnt,他是所指向緩存區dst的大小，如果緩存區太小無法存儲位址的值，則返回一個null point，並將errno置為ENOSPC。

# Address Conversion Summary



# Simple version of inet_pton that supports only IPv4

- libfree/inet_pton_ipv4.c



```c
int
inet_pton(int family, const char *strptr, void *addrptr)
{
    if (family == AF_INET) {
        struct in_addr in_val;

        if (inet_aton(strptr, &in_val)) {
            memcpy(addrptr, &in_val, sizeof(struct in_addr));
            return (1);
        }
        return (0);
    }
    errno = EAFNOSUPPORT;
    return (-1);
}
```

# Simple version of inet_ntop that supports only IPv4

- libfree/inet_ntop_ipv4.c

```c
const char *
inet_ntop(int family, const void *addrptr, char *strptr, size_t len)
{
    const u_char *p = (const u_char *) addrptr;

    if (family == AF_INET) {
        char    temp[INET_ADDRSTRLEN];

        snprintf(temp, sizeof(temp), "%d.%d.%d.%d", p[0], p[1], p[2], p[3]);
        if (strlen(temp) >= len) {
            errno = ENOSPC;
            return (NULL);
        }
        strcpy(strptr, temp);
        return (strptr);
    }
    errno = EAFNOSUPPORT;
    return (NULL);
}
```

## SOCK_NTOP AND RELATED FUNCTIONS

## A problem for inet_ntop

- A basic problem with inet_ntop is that it requires the caller to pass a pointer to a binary address.
- This address is normally contained in a socket address structure, requiring the caller to know the format of the structure and the address family.
- That is, to use it, we must write code of the form

  struct sockaddr_in addr; inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));

  – for IPv4, or

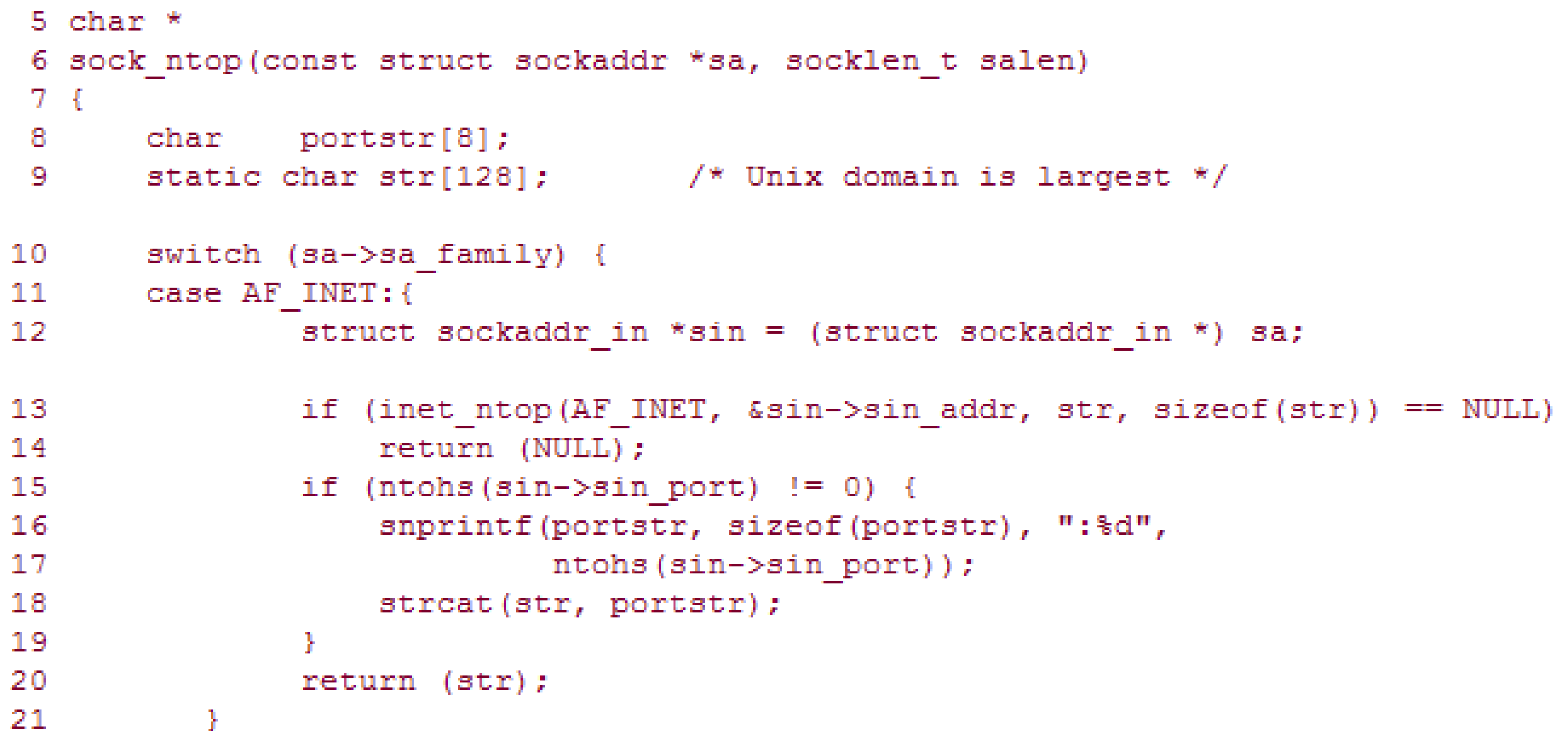
  struct sockaddr_in6 addr6; inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str)); for IPv6.
- This makes our code protocol-dependent.

## A problem for inet_ntop

- To solve this, we will write our own function named sock_ntop that takes a pointer to a socket address structure, looks inside the structure, and calls the appropriate function to return the presentation format of the address.
- #include "unp.h"
- char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
  - Returns: non-null pointer if OK, NULL on error

# sock_ntop function

- lib/sock_ntop.c



```c
char *
sock_ntop(const struct sockaddr *sa, socklen_t salen)
{
    char        portstr[8];
    static char str[128];       /* Unix domain is largest */

    switch (sa->sa_family) {
    case AF_INET:{
            struct sockaddr_in *sin = (struct sockaddr_in *) sa;

            if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
                return (NULL);
            if (ntohs(sin->sin_port) != 0) {
                snprintf(portstr, sizeof(portstr), ":%d",
                         ntohs(sin->sin_port));
                strcat(str, portstr);
            }
            return (str);
        }
```

## READN, WRITEN, AND READLINE FUNCTIONS

## READN, WRITEN, AND READLINE FUNCTIONS

- Stream sockets (e.g., TCP sockets) exhibit a behavior with the read and write functions that differs from normal file I/O.
- A read or write on a stream socket might input or output fewer bytes than requested, but this is not an error condition.
- The reason is that buffer limits might be reached for the socket in the kernel.
- All that is required to input or output the remaining bytes is for the caller to invoke the read or write function again.

## READN, WRITEN, AND READLINE FUNCTIONS

- Some versions of Unix also exhibit this behavior when writing more than 4,096 bytes to a pipe.
- This scenario is always a possibility on a stream socket with read, but is normally seen with write only if the socket is nonblocking.
- Nevertheless, we always call our writen function instead of write, in case the implementation returns a short count.

## READN, WRITEN, AND READLINE FUNCTIONS

- We provide the following three functions that we use whenever we read from or write to a stream socket:

```
#include "unp.h"
ssize_t readn(int filedes, void *buff, size_t nbytes);
ssize_t writen(int filedes, const void *buff, size_t nbytes);
ssize_t readline(int filedes, void *buff, size_t maxlen);
```

All return: number of bytes read or written, –1 on error

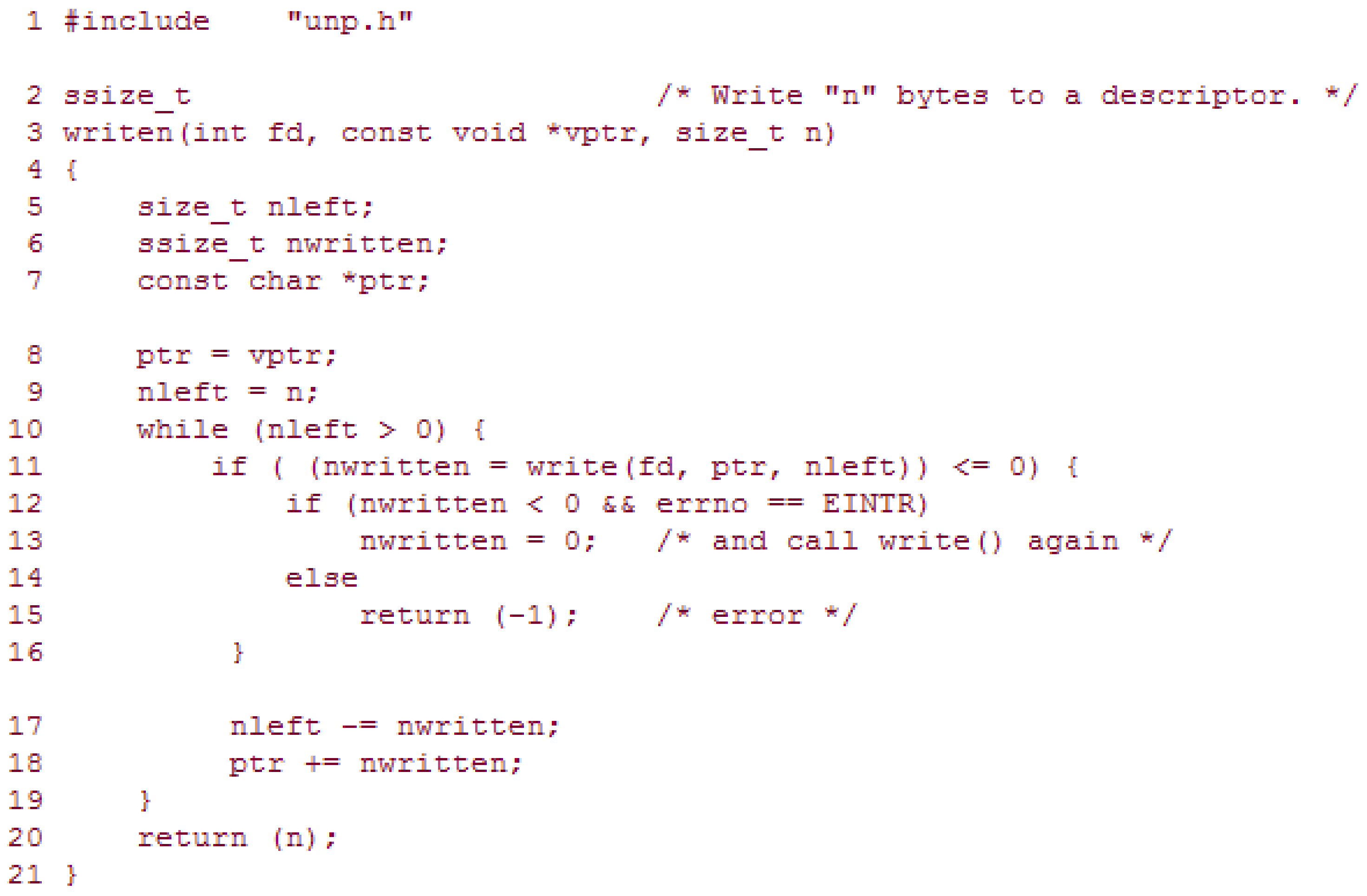
## Read n bytes from a descriptor

- lib/readn.c

```
#include     "unp.h"

ssize_t                              /* Read "n" bytes from a descriptor. */
readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char   *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;      /* and call read() again */
            else
                return (-1);
        } else if (nread == 0)
            break;              /* EOF */

        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

# Write n bytes to a descriptor

- lib/writen.c



```c
#include    "unp.h"

ssize_t                             /* Write "n" bytes to a descriptor. */
writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
                nwritten = 0;   /* and call write() again */
            else
                return (-1);    /* error */
         }

         nleft -= nwritten;
         ptr += nwritten;
    }
    return (n);
}
```

# Read a text line from a descriptor, one byte at a time

- test/readline1.c

```c
#include     "unp.h"

/* PAINFULLY SLOW VERSION -- example only */
ssize_t
readline(int fd, void *vptr, size_t maxlen)
{
    ssize_t n, rc;
    char    c, *ptr;

    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
      again:
        if ( (rc = read(fd, &c, 1)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;          /* newline is stored, like fgets() */
        } else if (rc == 0) {
            *ptr = 0;
            return (n - 1);     /* EOF, n - 1 bytes were read */
        } else {
            if (errno == EINTR)
                goto again;
            return (-1);        /* error, errno set by read() */
        }
    }

    *ptr = 0;                   /* null terminate like fgets() */
    return (n);
}
```

## The Performance of readline Function

- The readline function calls the system's read function once for every byte of data. This is very inefficient, and why we've commented the code to state it is "PAINFULLY SLOW.“
- A **read** or **write** on a stream socket might input or output fewer bytes than requested => short count
  - Because buffer limit might be reached for the socket in the kernel
  - The caller needs to invoke the **read** or **write** function again for the remaining bytes

## Better version of readline function

- **lib/readline.c**
- 2–21 The internal function my_read reads up to MAXLINE characters at a time and then returns them, one at a time.
- 29 The only change to the readline function itself is to call my_read instead of read.
- 42–48 A new function, readlinebuf, exposes the internal buffer state so that callers can check and see if more data was received beyond a single line.