Sockets Introduction

- 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: isfdtype

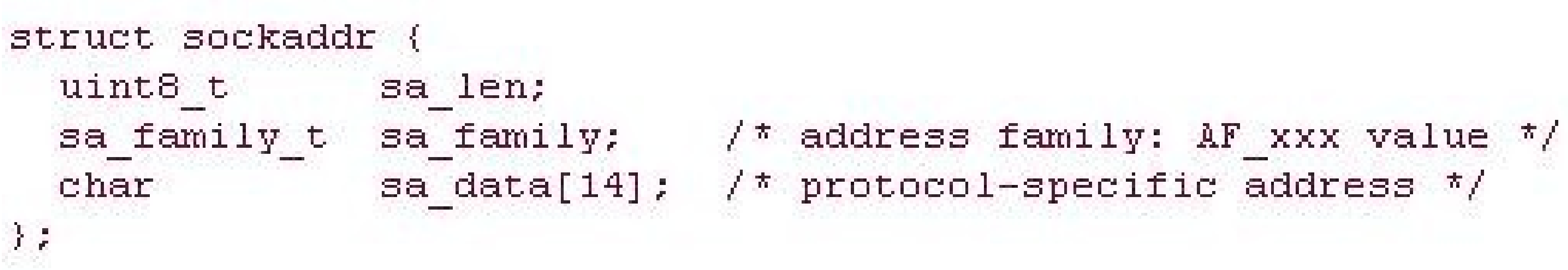
Socket Address Structures: IPv4, Generic, IPv6

```
ruct in_addr {
in_addr_t        s_addr;          /* 32-bit IPv4 address, network byte order */        };
ruct 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 */                                 };
ruct 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 */              };
ruct in6_addr {
unit8_t                 s6_addr[16];     /* 128-bit IPv6 address, network byte order */
ruct 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 */        };
```

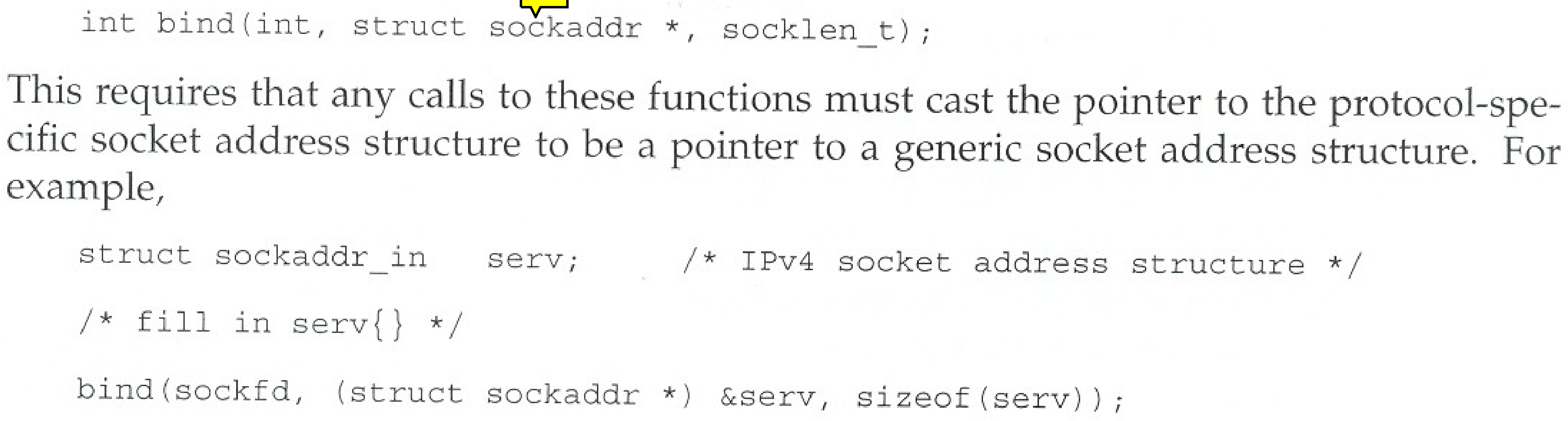
Datatypes Required by Posix.1g

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>

Figure 3.3 The generic socket address structure: sockaddr.



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



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

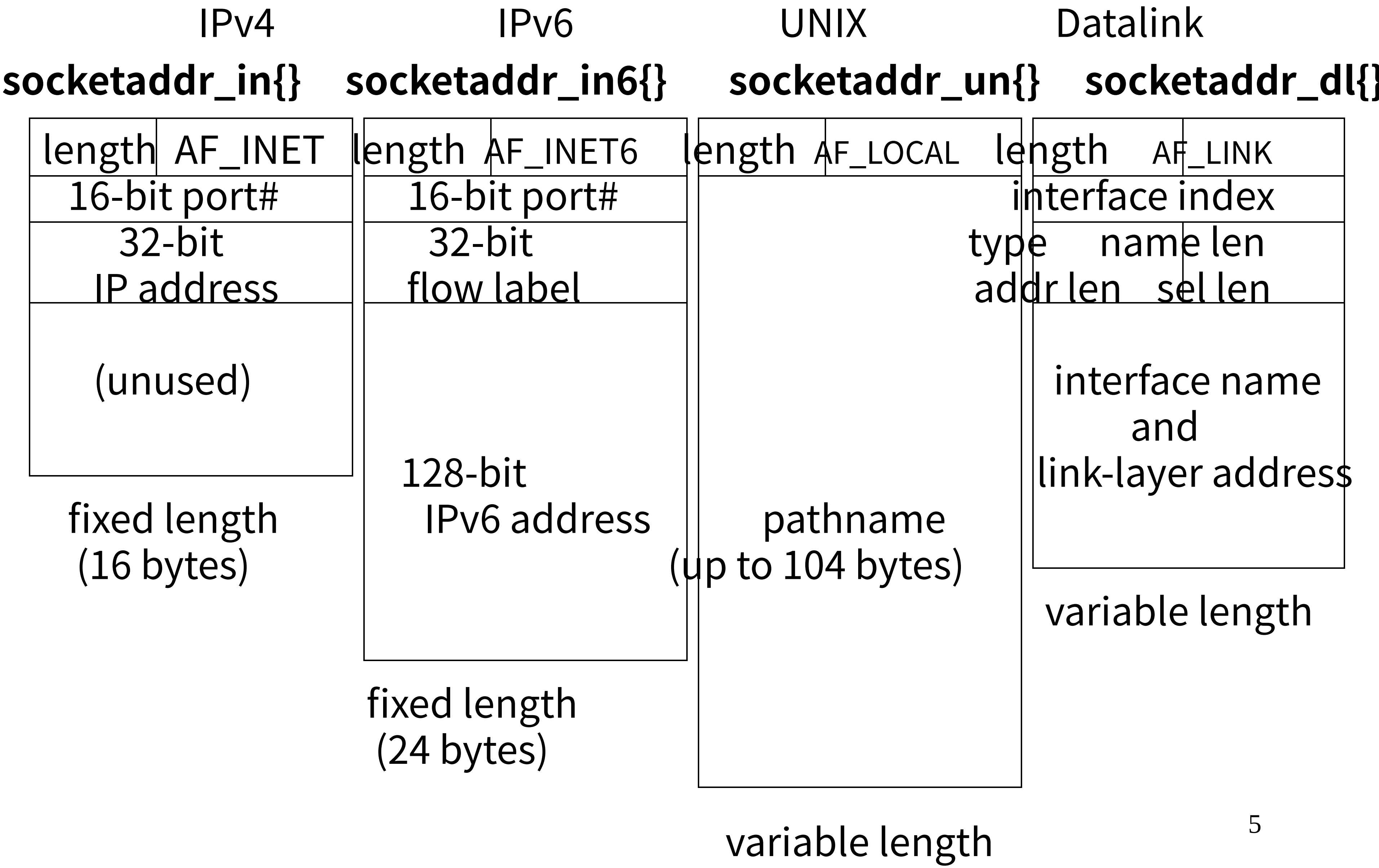
This requires that any calls to these functions must cast the pointer to the protocol-specific socket address structure to be a pointer to a generic socket address structure. For example,

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

/* fill in serv{} */

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

Comparison of Various Socket Address Structures



IPv4

socketaddr_in{}

length | AF_INET

16-bit port#

32-bit IP address

(unused)

fixed length (16 bytes)

IPv6

socketaddr_in6{}

length | AF_INET6

16-bit port#

32-bit flow label

128-bit IPv6 address

fixed length (24 bytes)

UNIX

socketaddr_un{}

length | AF_LOCAL

pathname (up to 104 bytes)

variable length

Datalink

socketaddr_dl{}

length | AF_LINK

interface index

type | name len

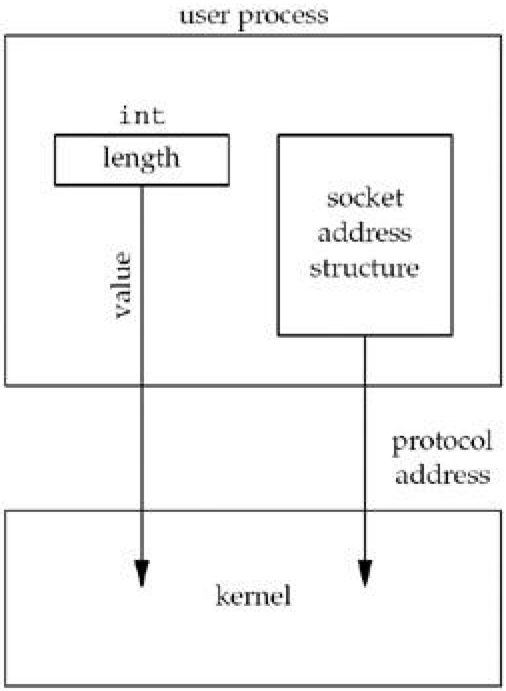
addr len | sel len

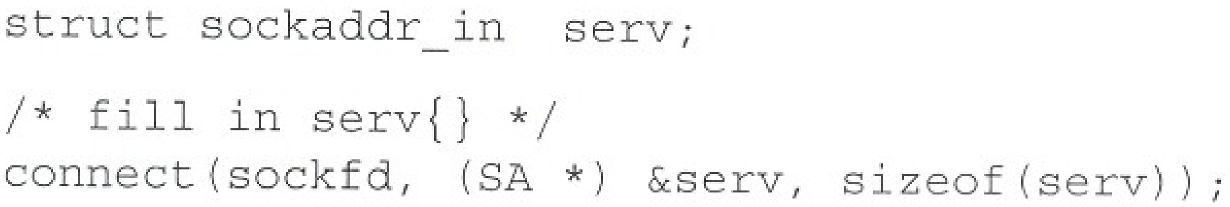
interface name and link-layer address

variable length

Value-Result Argument

Figure 3.7. Socket address structure passed from process to kernel.

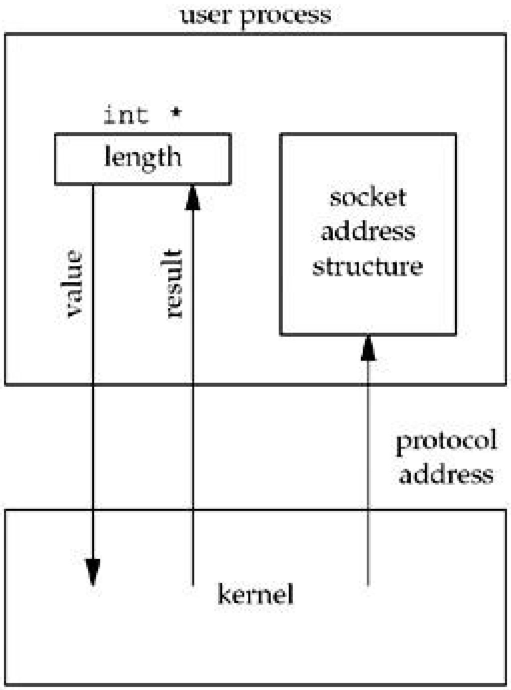


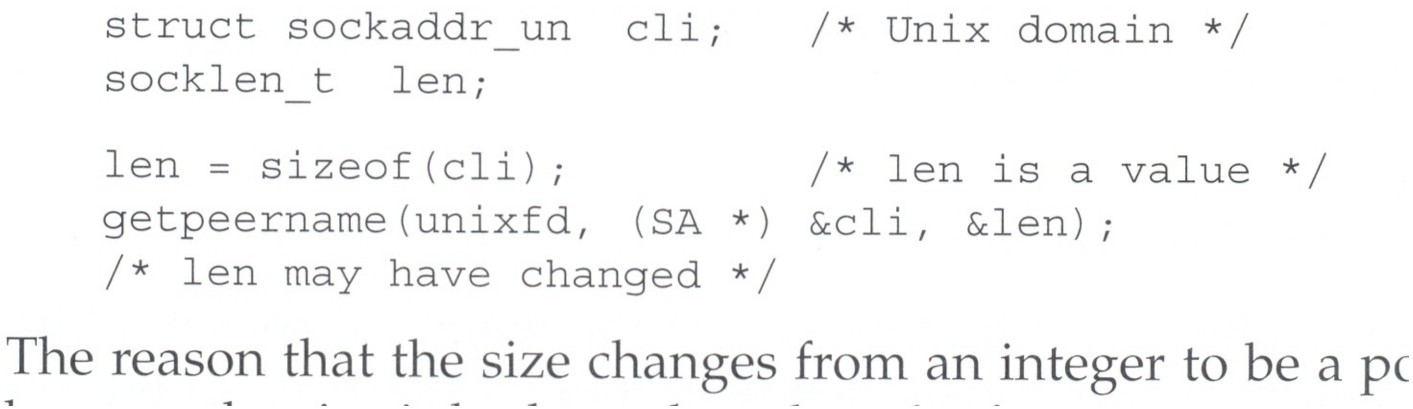


```
struct sockaddr_in  serv;

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

Figure 3.8. Socket address structure passed from kernel to process.





```
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 */
```

The reason that the size changes from an integer to be a pc

Byte Ordering Functions:

converting between the host byte order and the network byte order

address A+1 address A

little-endian byte order: high-order byte low-order byte

big-endian byte order: high-order byte low-order byte

(for a 16-bit integer) address A address A+1

Some machines use the little-endian host byte order while the others use the big-endian. The Internet protocols use the *big-endian* network byte order. Hence, conversion functions should be added in all cases.

#include <netinet/in.h>
uint16_t htons(uint16_t *host16bitvalue*); returns: value in network byte order
uint32_t htonl(uint32_t *host32bitvalue*); returns: value in network byte order
uint16_t ntohs(uint16_t *net16bitvalue*); returns: value in host byte order
uint32_t ntohl(uint32_t *net32bitvalue*); returns: value in host byte order

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
```

```
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
```

Address Conversion Functions

between ASCII strings and network byte ordered binary values

For IPv4 only: ascii and numeric

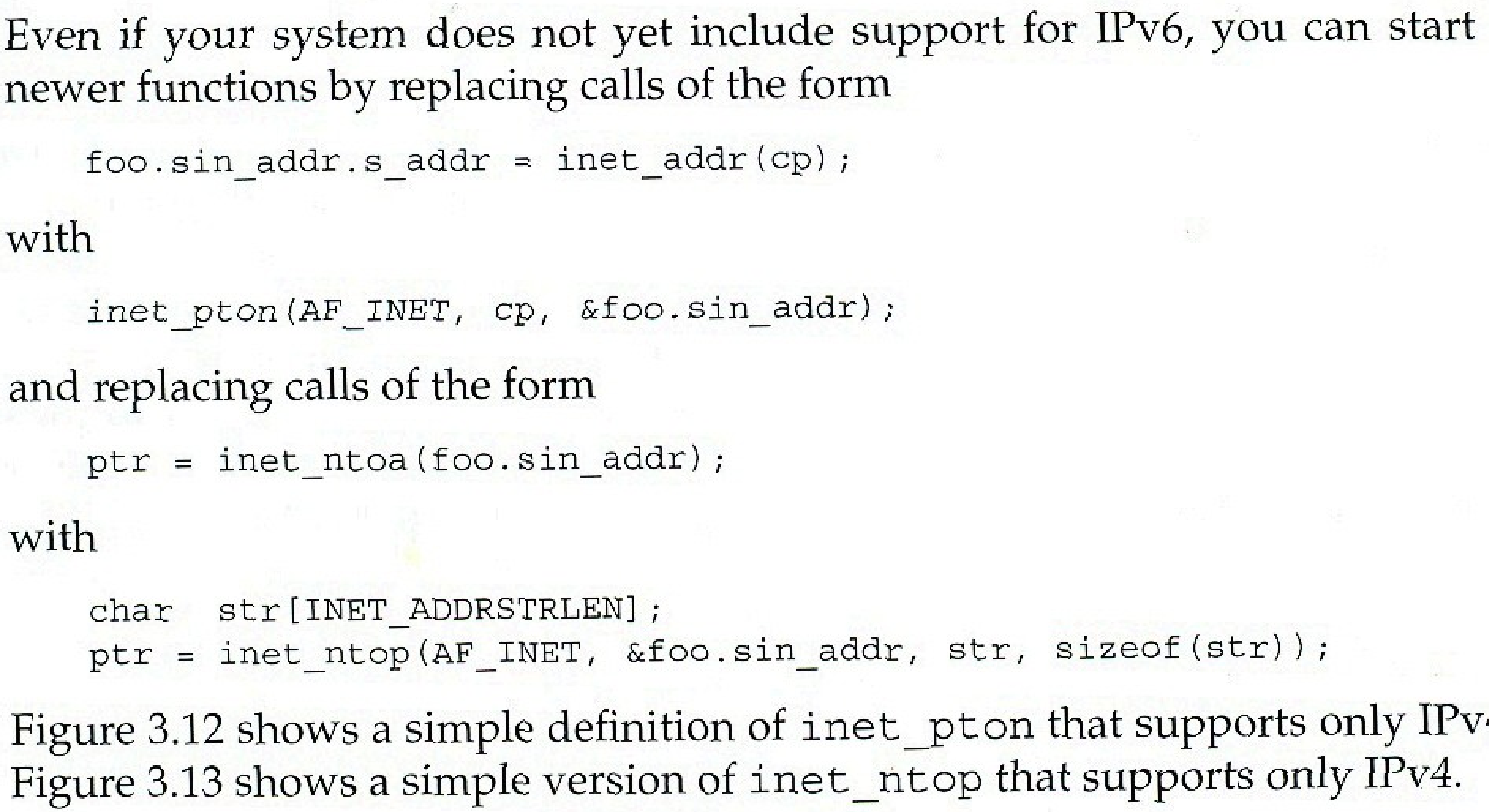
```
#include <arpa/inet.h>
int inet_aton (const char *strptr, struct in_addr *addrptr);
                              returns: 1 if string is valid, 0 on error
int_addr_t  inet_addr (const char *strptr);
                    returns: 32-bit binary IPv4 addr, INADDR_NONE if error
char * inet_ntoa (struct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

For IPv4 (AF_INET) and IPv6 (AF_INET6): presentation and numeric

```
#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), INET6_ADDRSTRLEN = 46 (for IPv6 hex string)

Example



Even if your system does not yet include support for IPv6, you can start
newer functions by replacing calls of the form

```
foo.sin_addr.s_addr = inet_addr(cp);
```

with

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

and replacing calls of the form

```
ptr = inet_ntoa(foo.sin_addr);
```

with

```
char  str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

Figure 3.12 shows a simple definition of `inet_pton` that supports only IPv
Figure 3.13 shows a simple version of `inet_ntop` that supports only IPv4.

Address Conversion Functions (cont.)

Protocol independent functions (not standard system functions):

#include "unp.h"
char *sock_ntop (const struct sockaddr **sockaddr*, socklen_t *addrlen*);
returns: nonnull pointer if OK, NULL on error

sock_bind_wild, sock_cmp_addr, sock_cmp_port, sock_get_port, sock_ntop_host, sock_set_addr, sock_set_port, sock_set_wild, etc.

Stream Socket I/O Functions:
preventing callers from having to handle a short count

read and *write* return, before reading or writing requested bytes, when the socket buffer limit is reached in the kernel.

When reading from or writing to a stream socket:
(need to handle EINTR signal)
#include “unp.h”
ssize_t readn (int *filedes*, void **buff*, size_t *nbytes*);
returns: #bytes read, -1 on error
ssize_t writen (int *filedes*, const void **buff*, size_t *nbytes*);
returns: #bytes written, -1 on error
ssize_t readline (int *filedes*, void **buff*, size_t *maxlen*);
returns: #bytes read, -1 on error

Figure 3.15 `readn` function: Read *n* bytes from a descriptor.

lib/readn.c

```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 */
}
```

Figure 3.16 **writen** function: Write *n* bytes to a descriptor.

lib/writen.c

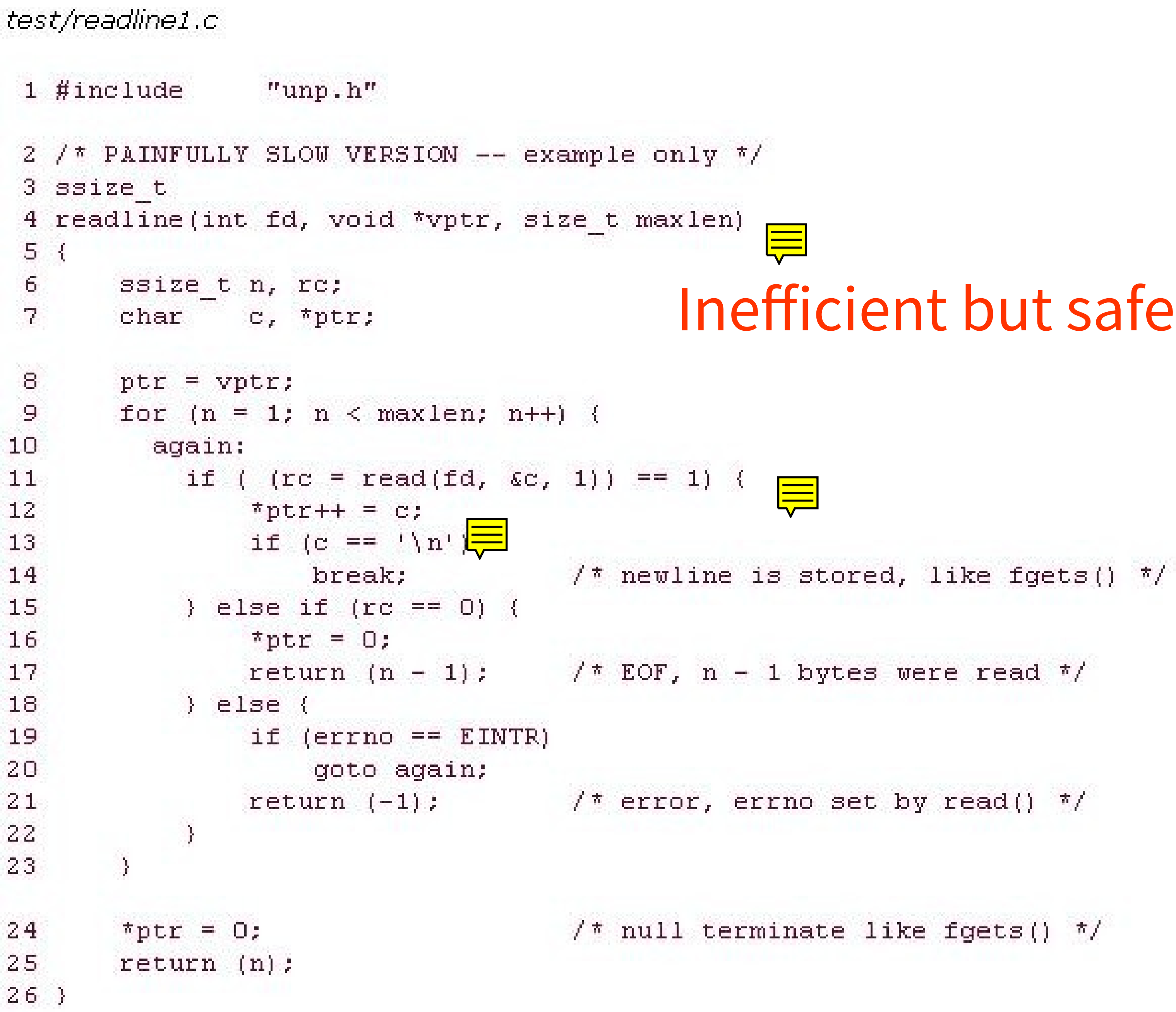
```
#include    "unp.h"

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);
}
```

Figure 3.17 `readline` function: Read a text line from a descriptor, one byte at a time.



test/readline1.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);
}
```

Inefficient but safe

Figure 3.18 Better version of `readline` function.

lib/readline.c

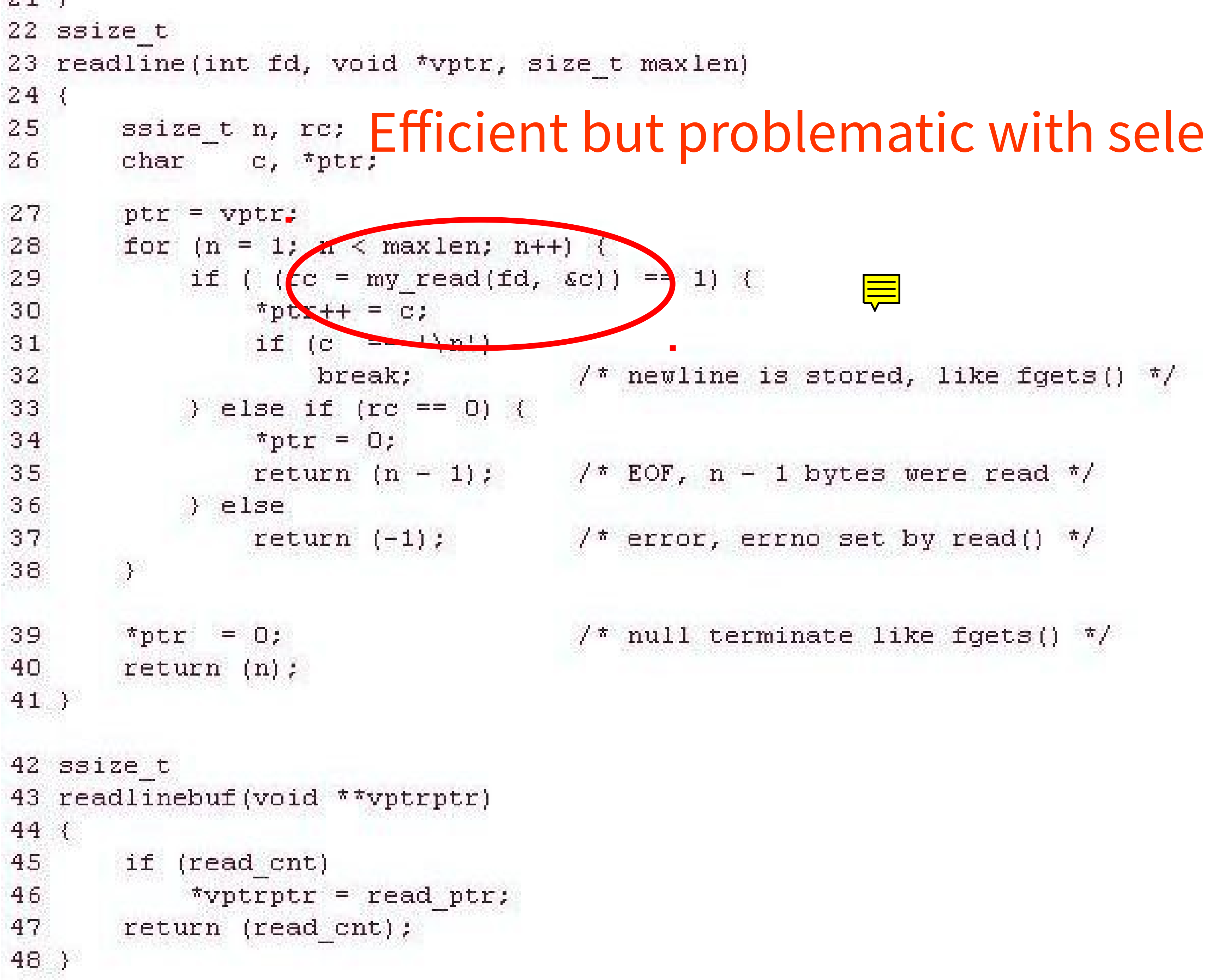
```
#include    "unp.h"

static int read_cnt;
static char *read_ptr;
static char read_buf[MAXLINE];

static ssize_t
my_read(int fd, char *ptr)
{

    if (read_cnt <= 0) {
      again:
        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR)
                goto again;
            return (-1);
        } else if (read_cnt == 0)
            return (0);
        read_ptr = read_buf;
    }

    read_cnt--;
    *ptr = *read_ptr++;
    return (1);
}
ssize_t
readline(int fd, void *vptr, size_t maxlen)
```



Efficient but problematic with select(

```
}
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++) {
        if ( (rc = my_read(fd, &c)) == 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
            return (-1);        /* error, errno set by read() */
    }

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

ssize_t
readlinebuf(void **vptrptr)
{
    if (read_cnt)
        *vptrptr = read_ptr;
    return (read_cnt);
}
```