NETWORK PROGRAMMING
CHAPTER 2: UNIX NETWORK PROGRAMMING

Compiled By:

Assoc. Prof. Madan Kadariya

HoD, Department of IT, NCIT

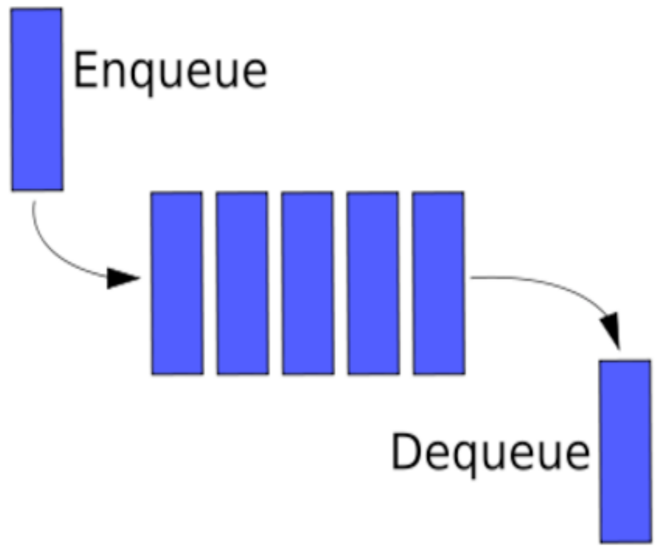
SOCKET ABSTRACTION AND INTERPROCESS COMMUNICATION

CHARACTERISTIC OF IPC

- Message passing between a pair of processes supported by two communication operations: **send** and **receive**
  - Defined in terms of destinations and messages.
- In order for one process A to communicate with another process B:
  - A sends a message (sequence of bytes) to a destination
  - Another process at the destination (B) receives the message.
- This activity involves the communication of data from the **sending** process to the **receiving** process and may involve the **synchronization** of the two processes

SENDING AND RECEIVING

- A queue is associated with each message destination.
- Sending processes cause messages to be added to remote queues.
- Receiving processes remove messages from local queues.
- Communication between the sending and receiving process may be either **Synchronous** or **Asynchronous**.



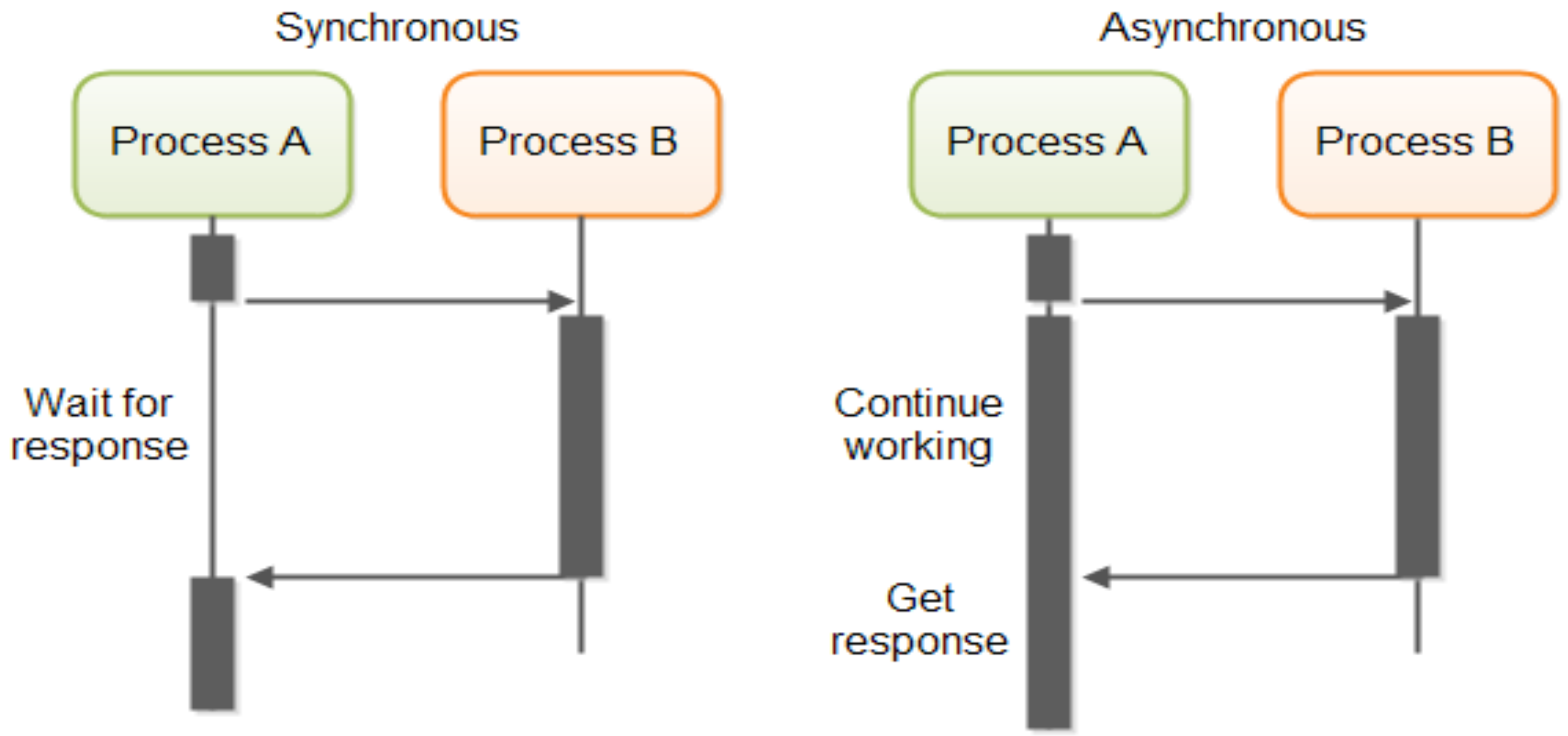
SYNCHRONOUS AND ASYNCHRONOUS COMMUNICATION

Synchronous Communication

- ➢The sending and receiving processes **synchronize** at every message.
- ➢In this case, both **send** and **receive** are **blocking** operations:
  - ➢whenever a **send** is issued the **sending process is blocked** until the corresponding **receive** is issued;
  - ➢whenever a **receive** is issued the **receiving process blocks** until a message arrives.

Asynchronous Communication

- ➢ The **send** operation is non-blocking:
  - ➢ the sending process is **allowed to proceed** as soon as the message has been copied to a local buffer;
  - ➢ The transmission of the message proceeds in **parallel with the sending process.**
- ➢ The **receive** operation can have **blocking and non-blocking** variants:
  - ➢ [**non-blocking**] the receiving process proceeds with its program after issuing a receive operation;
  - ➢ [**blocking**] receiving process blocks until a message arrives.


Synchronous
Process A
Process B
Wait for response
Asynchronous
Process A
Process B
Continue working
Get response

NETWORK PROGRAMMING INTRODUCTION

- Typical applications today consist of many cooperating processes either on the **same host** or on **different hosts**.
- For example, consider a client-server application. How to share (large amounts of) data?
- Share files? How to avoid **contention**? What kind of system support is available?
- We want a general mechanism that will work for processes irrespective of their location.

IPC USING SOCKETS

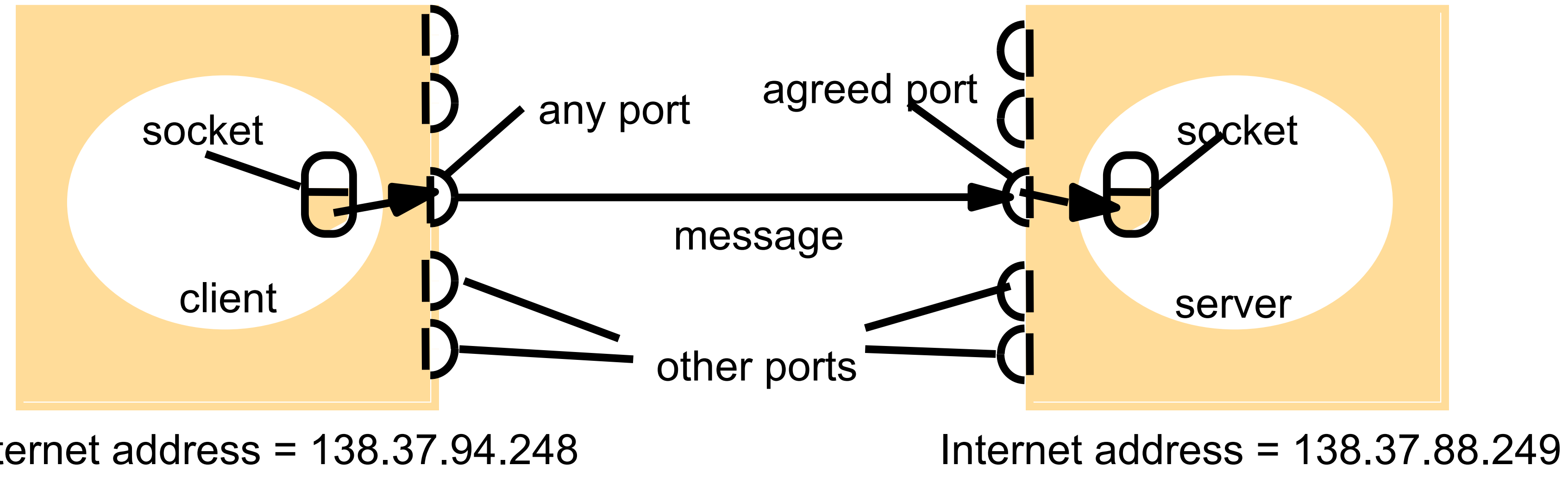
Purposes of IPC

- Data transfer
- Sharing data
- Event notification
- Process control
- What if wanted to communicate between processes that have no common ancestor? Ans: **sockets**
- IPC for processes that are not necessarily on the **same host**.
- Sockets use names to refer to one another: Means of network IO.

WHAT ARE SOCKETS?

- Socket is an **abstraction** for an **end point of communication** that can be manipulated with a **file descriptor**.
  - **File descriptor:** File descriptors are **nonnegative** integers that the kernel uses to identify the file being accessed by a particular process. Whenever the **kernel** opens an existing file or creates a new file it returns the file descriptor.
- A socket is an abstraction which provides an endpoint for communication between **processes**.
- A **socket address** is the combination of an **IP address** (the location of the computer) and a **port** (a specific service) into a **single identity**.
- **Interprocess communication** consists of transmitting a message between a **socket** in **one process** and a socket in **another process**.
- A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. Example: UNIX domain, internet domain.

SOCKETS AND PORTS



socket
client
any port
agreed port
message
other ports
socket
server
Internet address = 138.37.94.248
Internet address = 138.37.88.249

- Messages sent to a particular **Internet address** and **port number** can be received only by a process whose **socket** is associated with that **Internet address** and **port number**.
- Processes may use the same **socket** for **sending** and **receiving** messages.
- Any process may make use of **multiple ports** to receive messages, BUT a process cannot share ports with other processes on the same computer.
- Each socket is associated with a particular protocol, either UDP or TCP.

INTER PROCESS COMMUNICATION

- **IP address and port number**: In IPv4 about **2^{16}** ports are available for use by user processes.
- **Socket** is associated with a protocol.
- IPC is transmitting a message between a socket in **one process** to a socket in **another process**.
- Messages sent to particular IP and port# can be received by the process whose socket is associated with that IP and port#.
- Processes cannot share ports with other processes within the computer. Can receive messages on diff ports.

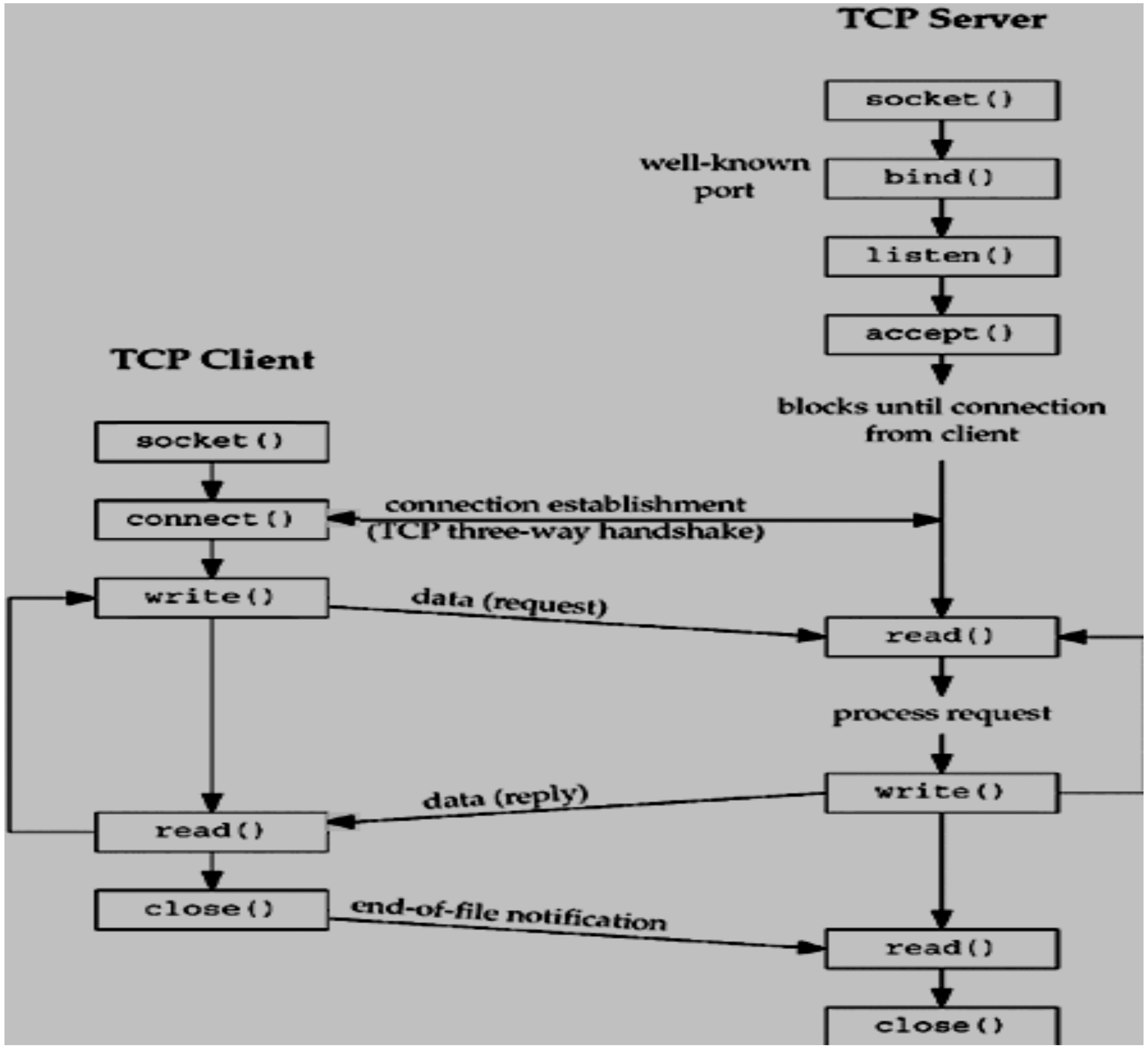
SOCKETS

- **Socket** is an **abstraction** that is provided to an application programmer to **send** or **receive** data to another process.
- Data can be **sent** to or **received** from another process running on the **same machine** or a **different machine.**
- **What is a socket?**
  - To the **kernel**, a **socket is an endpoint of communication.**
  - To an **application**, a socket is a **file descriptor that lets the application read/write from/to the network.**
    - Remember: All Unix I/O devices, including networks, are modeled as files.
- **Clients** and **servers** communicate with each by reading from and writing to socket descriptors.
- The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors.

SOCKET API

- A socket API is an **application-programming interface** (API), usually provided by the **operating system**, that allows application programs to control and use network sockets.
- Internet socket APIs are usually based on the **Berkeley sockets** standard.
- In the **Berkeley sockets** standard, sockets are a form of **file descriptor** (a file handle)
- In **inter-process communication**, each end will generally have its own socket, but these may use different APIs: they are abstracted by the network protocol.
- A socket address is the combination of an **IP address** and a **port number.**

SOCKET FUNCTIONS FOR TCP CLIENT/SERVER



SOCKET ADDRESS STRUCTURES

- Various structures are used in Unix Socket Programming to hold information about the address and port, and other information.
- Most socket functions require a **pointer to a socket address structure** as an argument.
- Generic socket address structure to hold socket information. It is passed in most of the socket function calls. Unix Socket Programming provides IP version specific socket address structures.

SOCKET ADDRESS STRUCTURES

Generic socket address:

- For address arguments to `connect`, `bind`, and `accept`.

```
struct sockaddr {
  unsigned short  sa_family;    /* protocol family */
  char            sa_data[14];  /* address data.   */
};
```

Internet-specific socket address (IPv4 socket address structure) :

- Must cast (`sockaddr_in *`) to (`sockaddr *`) for `connect`, `bind`, and `accept`.

```
struct sockaddr_in  {
  unsigned short  sin_family;  /* address family (always AF_INET) */
  unsigned short  sin_port;    /* port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
};
```

```
struct in_addr {
    unsigned long s_addr;              // this holds IPv4 address
};
```

SOCKET ADDRESS STRUCTURES

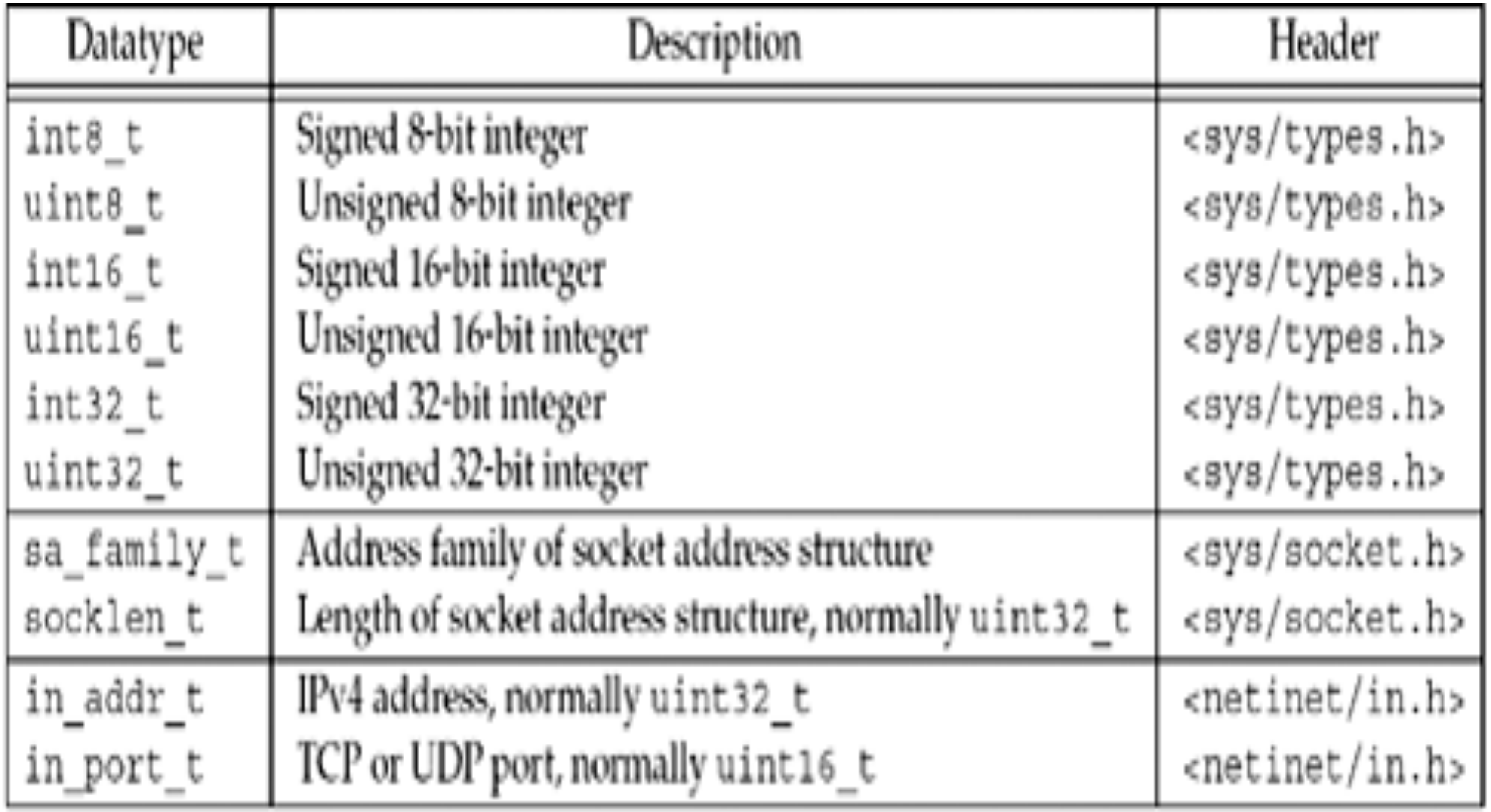
- **POSIX** requires only three members in the structure: **sin_family**, **sin_addr**, and **sin_port**. Almost all implementations add the **sin_zero** member so that all socket address structures are at least 16 bytes in size.
- The **in_addr_t** datatype must be an **unsigned integer** type of at least **32 bits**, **in_port_t** must be an unsigned integer type of at least **16 bits**, and **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.
- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- The **sin_zero** member is unused. By convention, we always set the entire structure to 0 before filling it in.
- Socket address structures are used only on a given host: The structure itself is not communicated between different hosts

IPV6 SOCKET ADDRESS STRUCTURE

```
struct sockaddr_in6 {
    u_int16_t           sin6_family;     // AF_INET6
    u_int16_t           sin6_port;        // this holds port number
    u_int32_t           sin6_flowinfo;  // this holds IPv6 flow information
    struct in6_addr     sin6_addr;    // used to hold IPv6 address
    u_int32_t           sin6_scope_id;// scope id
};

strcut in6_addr {
              unsigned char s6_addr[16]; // this holds IPv6 address
};
```

- The **IPv6** family is **AF_INET6**, whereas the **IPv4** family is **AF_INET**
- The members in this structure are ordered so that if the **sockaddr_in6** structure is 64-bit aligned, so is the 128-bit **sin6_addr** member.
- The **sin6_flowinfo** member is divided into two fields:
  - The low-order 20 bits are the flow label
  - The high-order 12 bits are the reserved
- The **sin6_scope_id** identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address
-

DATATYPES REQUIRED BY POSIX



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>`
`uint32_t`	Unsigned 32-bit integer	`<sys/types.h>`
`sa_family_t`	Address family of socket address structure	`<sys/socket.h>`
`socklen_t`	Length of socket address structure, normally `uint32_t`	`<sys/socket.h>`
`in_addr_t`	IPv4 address, normally `uint32_t`	`<netinet/in.h>`
`in_port_t`	TCP or UDP port, normally `uint16_t`	`<netinet/in.h>`

VALUE-RESULT ARGUMENTS

➢ When a **socket address structure** is passed to any socket function, it is always passed by **reference**. That is, a **pointer to the structure** is passed.

➢ The length of the structure is also passed as an argument. But the way in which the length is passed depends on which direction the structure is being passed:

- From the process to the kernel
- From the kernel to the process

From process to kernel

bind(), *connect()*, and *sendto()* functions pass a **socket address structure** from the process to the kernel

Arumgents to these functions:

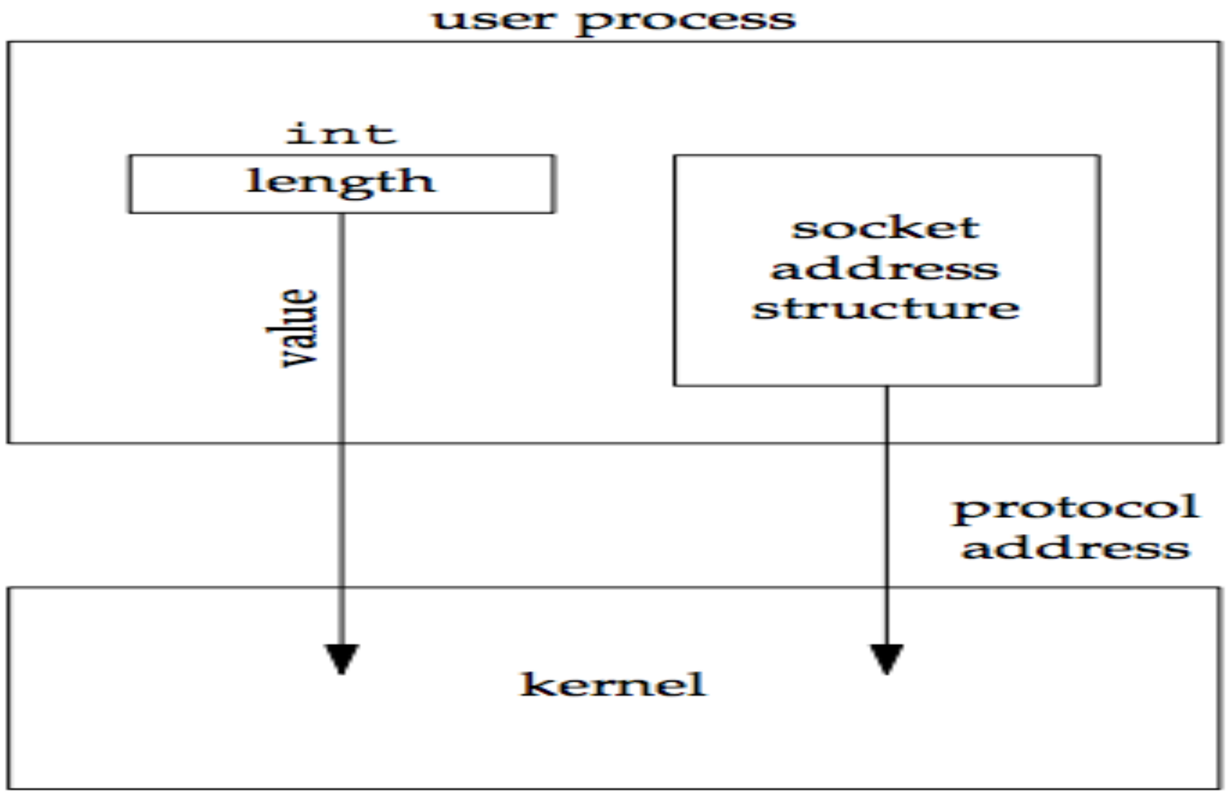
The pointer to the socket address structure

The integer size of the structure

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

The datatype for the size of a socket address structure is actually *socklen_t* and not *int*, but the POSIX specification recommends that *socklen_t* be defined as *uint32_t*

VALUE-RESULT ARGUMENTS



VALUE-RESULT ARGUMENTS

Kernel to Process

accept(), *recvfrom()*, *getsockname()*, and *getpeername()* functions pass a **socket address structure** from the kernel to the process.

Arguments to these functions:

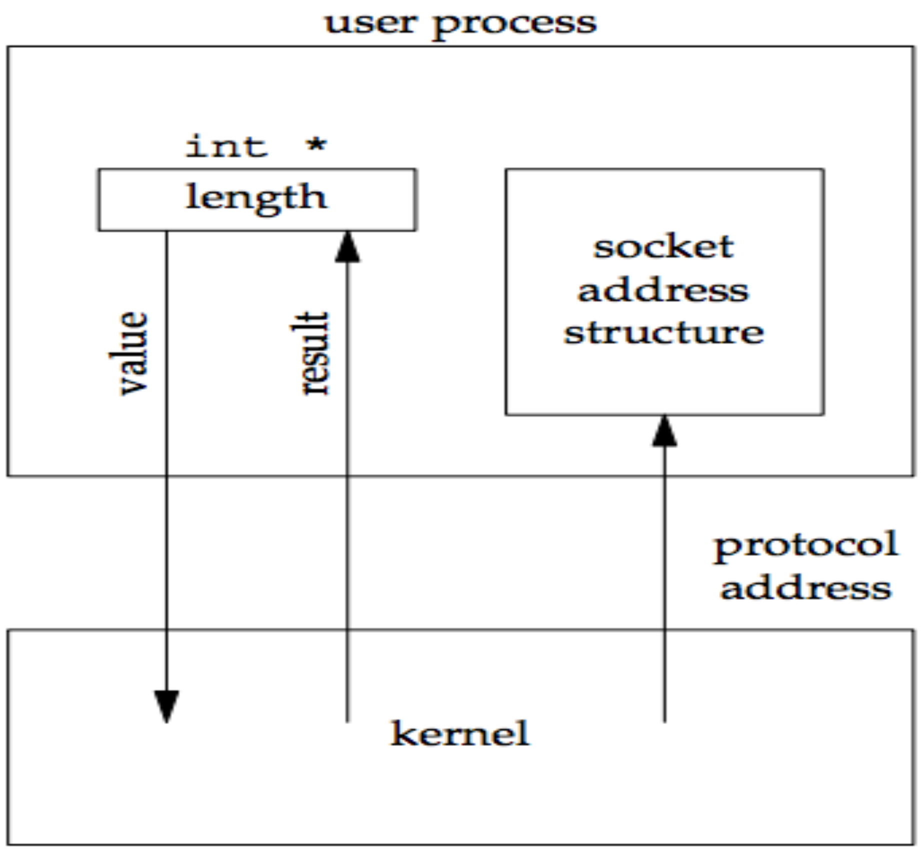
The pointer to the socket address structure

The 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 */
```

VALUE-RESULT ARGUMENTS

- Value-only: ***bind, connect, sendto*** (from process to kernel)
- Value-Result: ***accept, recvfrom, getsockname, getpeername*** (from kernel to process, pass a pointer to an integer containing size)
  - ➢*Tells process how much information kernel actually stored*



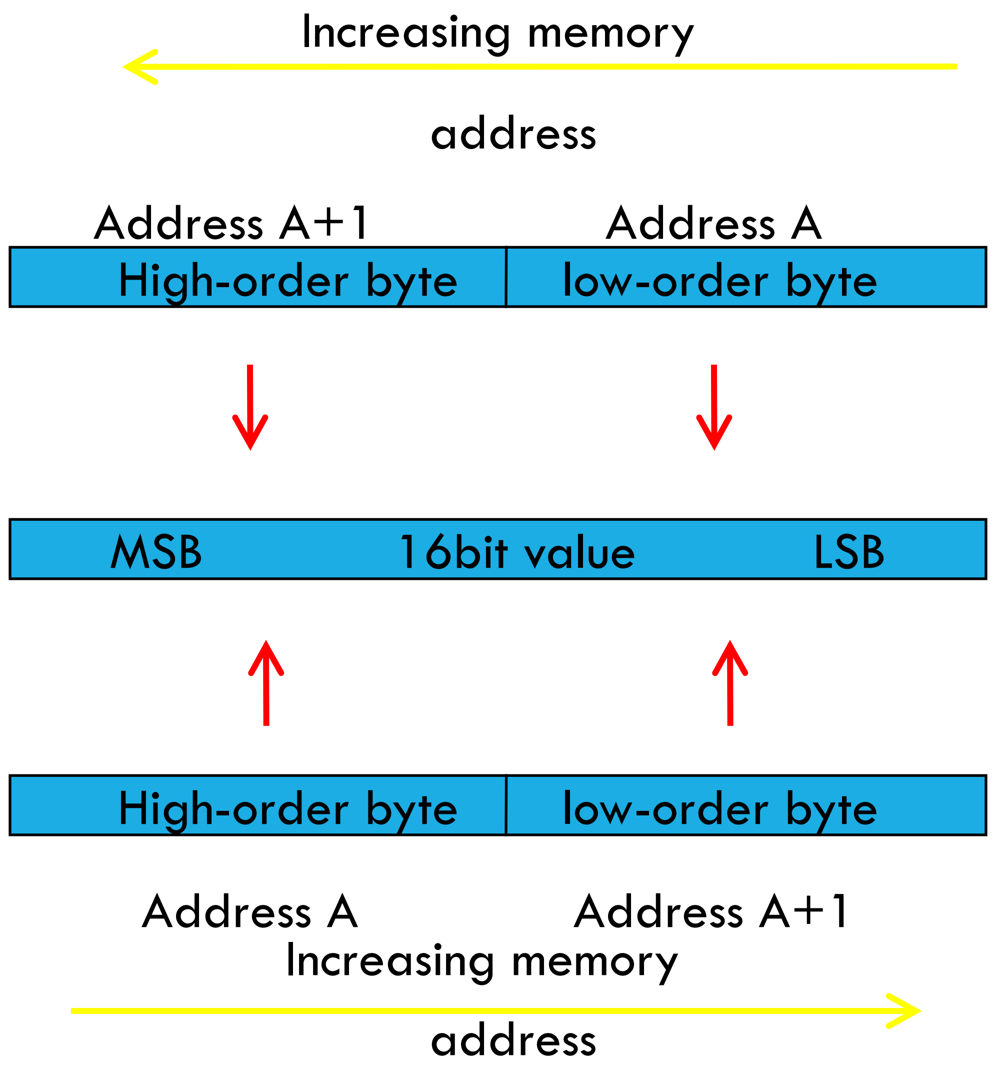
BYTE ORDERING FUNCTIONS

- 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
- *in a big-endian computer* → store 4F52
  - ➢Stored as 4F52 → 4F is stored at storage address 1000, 52 will be at address 1001, for example
- *In a little-endian system* → store 4F52
  - ➢it would be stored as 524F (52 at address 1000, 4F at 1001)
- 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)

BYTE ORDERING FUNCTIONS

Little-endian byte order:

big-endian byte order:



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
- Source code in byteorder.c

BYTE ORDERING FUNCTIONS

Converts between **host byte order** and **network byte order**

- 'h' = host byte order
- 'n' = network byte order
- 'l' = long (4 bytes), converts IP addresses
- 's' = short (2 bytes), converts port numbers

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int
hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int
netshort);
```

BYTE ORDERING FUNCTIONS 4/4

- To convert between byte orders
  - ➢Return value in network byte order
    - ✓htons (s for short word 2 bytes)
    - ✓htonl (l for long word 4 bytes)
  - ➢Return value in host byte order
    - ✓ntohs
    - ✓ntohl
- Must call appropriate function to convert between host and network byte order
- On systems that have the same ordering as the Internet protocols, four functions usually defined as null macros

```
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(13);
```

DEALING WITH IP ADDRESSES

IP Addresses are commonly written as strings (“128.2.35.50”), but programs deal with IP addresses as integers.

Converting strings to numerical address:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr(“128.2.35.50”);
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
      fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Converting a numerical address to a string:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
      fprintf(stderr, “inet_ntoa failed!\n”); exit(1);
}
```

BYTE MANIPULATION FUNCTIONS

```
#include <strings.h>
void bzero (void *dest, size_t nbytes);
// sets specified number of bytes to 0 in the destination

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
```

ADDRESS CONVERSION FUNCTIONS 1/2

Convert an IPv4 address from a dotted-decimal string “206.168.112.96” to a 32-bit network byte order binary value

#include <arpa/inet.h>
int inet_aton (const char* strptr, struct in_addr *addrptr);
// return 1 if string was valid, 0 on error. Address stored in *addrptr

in_addr_t inet_addr (const char * strptr);
// returns 32 bit binary network byte order IPv4 address, currently deprecated

char * inet_nota (struct in_addr inaddr);
//returns pointer to dotted-decimal string

ADDRESS CONVERSION FUNCTIONS 2/2

To handle both IPv4 and IPv6 addresses

#include <arpa/inet.h>

int inet_pton (int family, const char* strptr, void *addrptr);

// return 1 if OK, 0 on error. 0 if not a valid presentation, -1 on error, Address stored in *addrptr

[Here p:presentation, n: Numeric]

Const char * inet_ntop (int family, const void* addrptr, char *strptr, size_t len);

// return pointer to result if OK, NULL on error

```
if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
                    err_quit("inet_pton error for %s", argv[1]);

ptr = inet_ntop (AF_INET,&addr.sin_addr,str,sizeof(str));
```

The family argument for both functions is either AF_INET or AF_INET6. If family is not supported, both functions return an error with errno set to EAFNOSUPPORT.

FORK AND EXEC

FORK AND EXEC FUNCTIONS

- **fork()** is called **once** but it returns **twice**.
- The creation of a new process is done using the **fork()** system call.
- A new program is run using the **exec(l,lp,le,v,vp)** family of **system calls**.
- These are two separate functions which may be used independently.

- A call to **fork()** will create a completely separate **sub-process** which will be exactly the same as the parent.
- The process that initiates the call to **fork** is called the **parent process**.
- The new process created by **fork** is called the **child process**.
- The child gets a copy of the **parent's text** and **memory space**.
- They do not share the **same memory** .

FORK RETURN VALUES

- **fork()** system call returns an integer to both the parent and child processes:
- -**1** this indicates an **error** with no child process created.
- A value of **zero** indicates that the child process code is being executed.
- Positive integers represent the **child's process identifier** (PID) and the code being executed is in the parent's process.

```
if ( (pid = fork()) == 0)

        printf("I am the child\n");

else

        printf("I am the parent\n");
```

THE EXEC() SYSTEM CALL

- Calling one of the **exec()** family will terminate the currently running program and starts executing a new one which is specified in the parameters of exec in the context of the existing process. The process id is not changed.

EXEC Family of Functions

int **execl**(const char *path, const char *arg, ...);

int **execle**(const char *path, const char *arg , ..., char * const envp[]);

int **execv**(const char *path, char *const argv[]);

int **execv**(const char *path, char *const argv[], char * const envp[]);

int **execlp**(const char *file, const char *arg, ...);

int **execvp**(const char *file, char *const argv[]);

- The first difference in these functions is that the first four take a pathname argument while the last two take a filename argument. When a filename argument is specified:
  - if filename contains a slash, it is taken as a pathname.
  - otherwise the executable file is searched for in the directories specified by the PATH environment variable.

SIMPLE EXECLP EXAMPLE

```
#include <sys/type.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0){ /* error occurred */
                fprintf(stderr, “Fork Failed”);
                exit(-1);
        }
        else if (pid == 0){ /* child process */
                execlp(“/bin/ls”,”ls”,NULL);
        }
        else { /* parent process */
                /* parent will wait for child to complete */
                wait(NULL);
                printf(“Child Complete”);
                exit(0);
        }
}
```

FORK() AND EXEC()

- When a program wants to have another program running in parallel, it will typically first use **fork**, then the child process will use **exec** to actually run the desired program.
- The **fork-and-exec** mechanism switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, and environment variables.

Purpose of exec() functions

- When a process calls one of the **exec** functions that process is completely replaced by the new program.
- The new program starts execution from **main** function.
- The process does not change across an exec because a new process is not created.
- But this function replaces the current process with new program from disk.

FORK() AND EXEC()

Problems of fork() function

- **Fork** is expensive. Because memory and all descriptors are duplicated in the child.
- **Inter process communication** is required to pass information between the **parent** and the **child** after the fork.
- A **descriptor** in the child process can affect a subsequent read or write by the parent.
- This **descriptor** copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.

WAIT() AND WAITPID()

Wait() / Waitpid() :

- Either of **wait** or **waitpid** can be used to remove zombies.
- **wait** (and **waitpid** in it's blocking form) temporarily suspends the execution of a parent process while a child process is running.
- Once the **child** has finished, the **waiting parent** is restarted.

Declarations:

```
#include <sys/types.h>
#include <sys/wait.h>
    pid_t wait(int *statloc);          /* returns process ID if OK, or -1 on error */
    pid_t waitpid(pid_t pid, int *statloc, int options);   /* returns process ID : if OK,
          0        : if non-blocking option && no zombies around
          -1       : on error
```

- The **statloc** argument can be one of two values:
- **NULL pointer**: the argument is simply ignored
- **pointer to an integer**: when **wait** returns, the integer this describes will contain status information of the terminated process.

WAIT() AND WAITPID()

Wait()	**Waitpid()**
wait blocks the caller until a child process terminates	**waitpid** can be either **blocking** or **non-blocking**: • If **options** is 0, then it is **blocking** • If **options** is **WNOHANG**, then is it **non-blocking**
if more than one **child** is running then **wait()** returns the first time one of the parent's offspring exits	**waitpid** is more flexible: • If **pid == -1**, it waits for any child process. In this respect, **waitpid** is equivalent to wait • If **pid > 0**, it waits for the child whose process ID equals **pid** • If **pid == 0**, it waits for any child whose process group ID equals that of the calling process • If **pid < -1**, it waits for any child whose process group ID equals that absolute value of **pid**

PROCESS RELATED FUNCTION

Functions those who are return a **process ID** are:

- ➢ **getpid ()** : returns the current **process ID.**
- ➢ **getppid ()** : returns the parent **process ID** of the **calling process**.
- ➢ **getuid()** : returns the real **user ID** of the **calling process**.
- ➢ **geteuid()**: returns the effective **user ID** of the calling process.
  - ➢ effective user ID gives the process additional permissions during execution of "set-user-ID" mode processes
- ➢ **getgid ()**: returns the **real group ID** of the calling process
- ➢ **getegid()**: returns the **effective group ID** of the calling process.

Process group

- ➢ A process group is a collection of one or more processes.
- ➢ Each process group has a unique **process ID**.
- ➢ A function **getpgrp()** returns the process **group id** of the calling process.
- ➢ Each process group have a **leader**.
- ➢ The leader is identified by having its **process group ID equal its process ID**.
- ➢ A process joins an existing process group or creates a new process group by calling **setpgid()**.

CONCURRENT SERVER

- When a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. The server that handles multiple clients simultaneously is a concurrent server.
- Outline of typical Concurrent Server

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket(…);
/* fill in sockaddr_in{} with server’s well-know port */
Bind(listenfd, …);
Listen(listenfd, LISTENQ);
for(;;) {
              connfd = Accept(listenfd, …); // probably blocks
              if( (pid = Fork()) ==0) {
                            Close(listenfd); // child closes listening socket
                            doit(connfd); // process the request
                            Close(connfd); // done with this client
                            exit(0); //child terminates
        }
   Close(connfd);
}
```

- When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket).
- The parent closes the connected socket since the child handles the new client. The function do it does whatever is required to service the client.
- Calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. However, the close of connfd by the parent (in the outline) doesn't terminate its connection with the client. This is because every file or socket has a reference count.
- The reference count is maintained in the file table entry. This is a count of the number of descriptors that are currently open that refer to this file or socket. In the outline, after socket returns, the file table entry associated with listenfd has a reference count of 1.
- After accept returns, the file table entry associated with connfd has a reference count of 1. But, after fork returns, both descriptors are shared between the parent and child, so the file table entries associated with both sockets now have a reference count of 2.
- Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all. The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0. This will occur at some time later when the child closes connfd.

CONCURRENT SERVER

Visualization of sockets and connection that occur in the outline code

1. The server is blocked in the call to accept and the connection request arrives from the client.
2. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.
3. The fork is called. After fork returns, listenfd and connfd are shared between the parent and child.
4. The parent closes the connected socket and the child closes the listening socket.

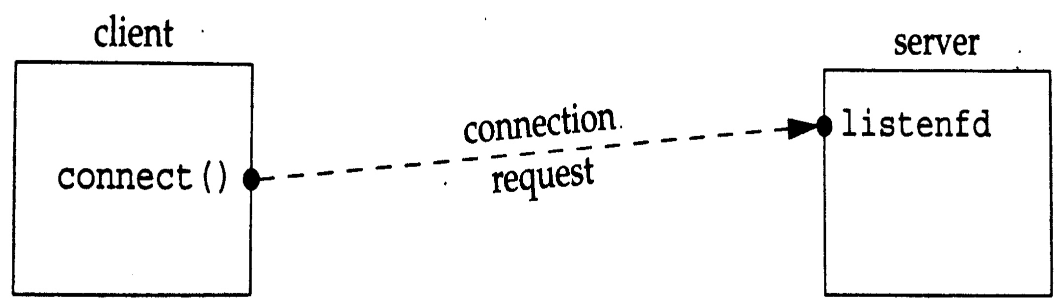


Figure 4.14 Status of client/server before call to accept returns.

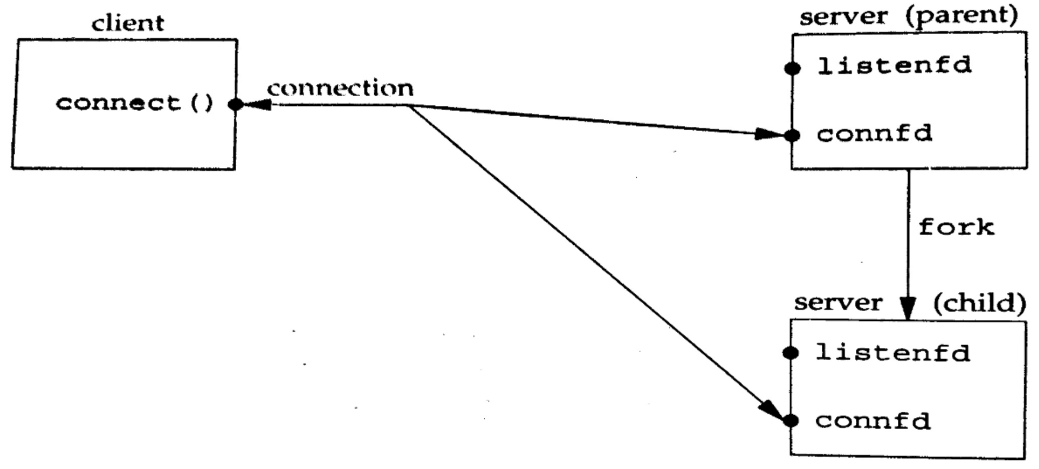


Figure 4.16 Status of client/server after fork returns.

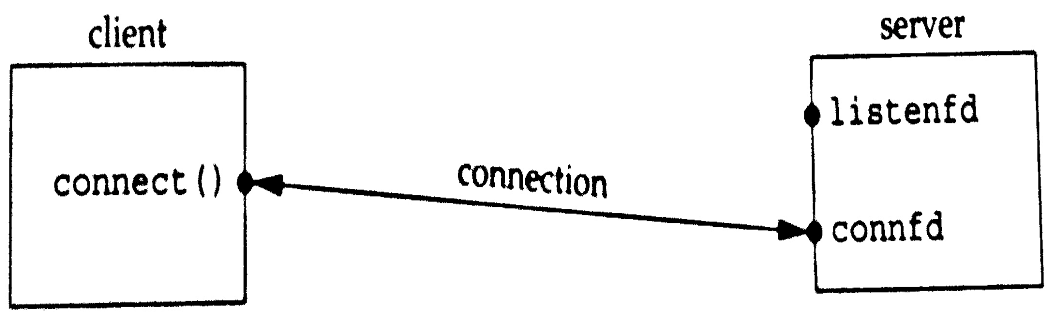


Figure 4.15 Status of client/server after return from accept.

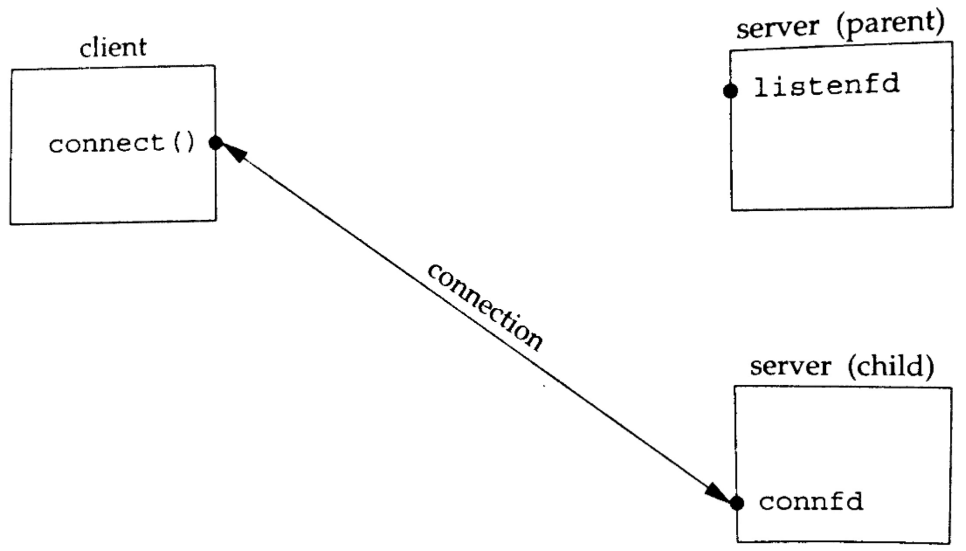
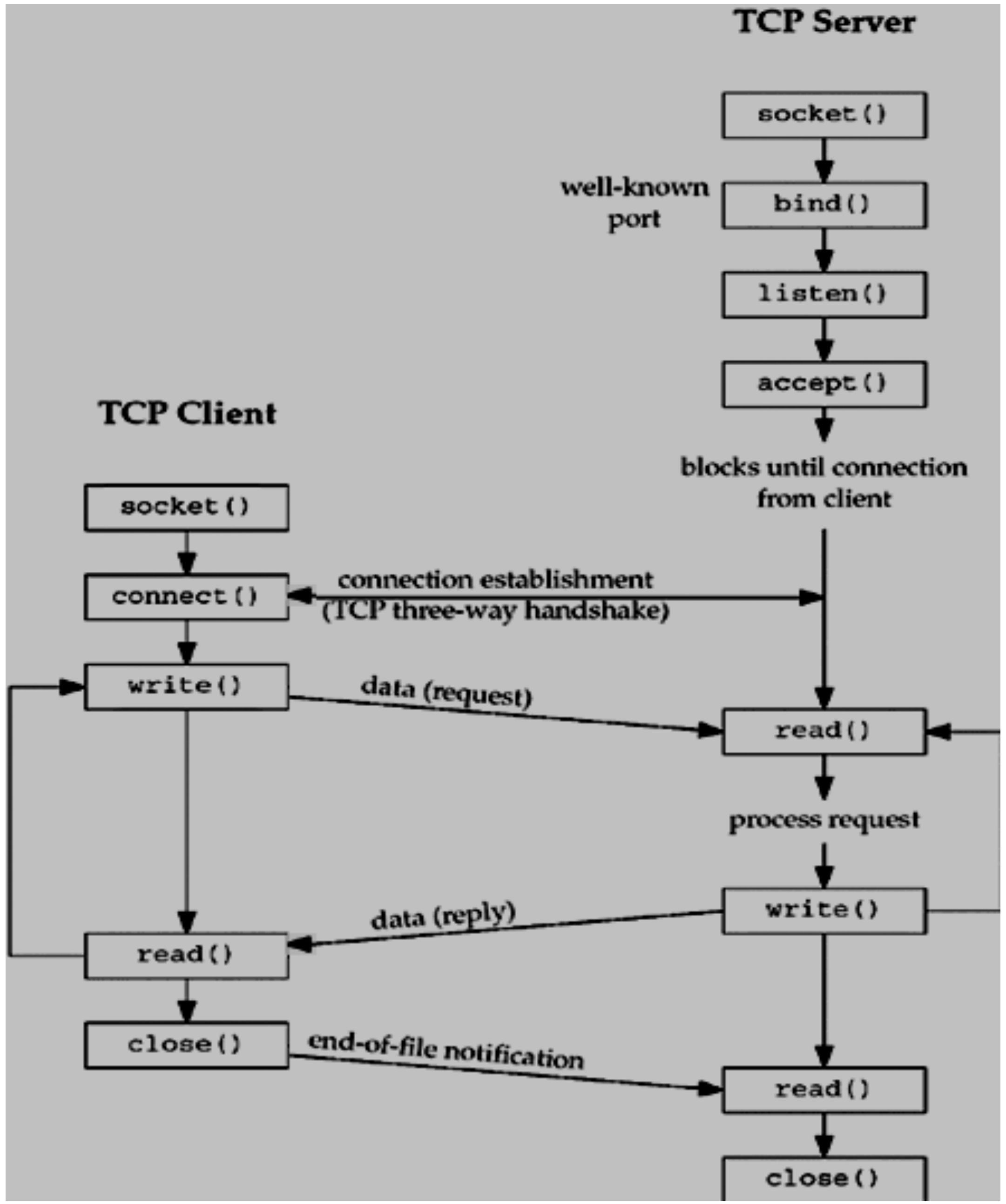


Figure 4.17 Status of client/server after parent and child close appropriate sockets.

UNIX/INTERNET DOMAIN SOCKET (TCP CONNECTION)



SOCKET ( )

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

int socket(int *family*, int *type*, int *protocol*);

Returns: non-negative descriptor if OK, -1 on error

family is one of

- AF_INET (IPv4), AF_INET6 (IPv6), AF_LOCAL (local Unix),
- AF_ROUTE (access to routing tables), AF_KEY (new, for encryption)

type is one of

- SOCK_STREAM (TCP), SOCK_DGRAM (UDP)
- SOCK_RAW (for special IP packets, PING, etc. Must be root)
- SOCK_SEQPACKET (Sequenced packet socket)

Protocol is one of

- IPPROTO_TCP
- IPPROTO_UDP
- IPPROTO_SCTP
- *protocol* is 0 (used for some raw socket options)

Not all combinations of socket *family* and *type* are valid. The table below shows the valid combinations, along with the actual protocols that are valid for each pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

	AF_INET	**AF_INET6**	**AF_LOCAL**	**AF_ROUTE**	**AF_KEY**
SOCK_STREAM	TCP/SCTP	TCP/SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

The key socket, AF_KEY, is newer than the others. It provides support for cryptographic security. Similar to the way that a routing socket (AF_ROUTE) is an interface to the kernel's routing table, the key socket is an interface into the kernel's key table.

BIND()

- ➢ The bind function assigns a local protocol address to a socket. With the Internet protocols, the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

int bind(int *sockfd*, const struct sockaddr **myaddr*,socklen_t *addrlen*);

- ➢ Returns: 0 if OK,-1 on error

sockfd is socket descriptor from `socket()`

myaddr is a pointer to address struct with:

- *port number* and *IP address*
- if port is 0, then host will pick ephemeral port
  - not usually for server (exception RPC port-map)
- IP address != INADDR_ANY (unless multiple nics)

addrlen is length of structure

returns 0 if ok, -1 on error

- EADDRINUSE (“Address already in use”)

- Calling **bind** lets us specify the IP address, the port, both, or neither. The following table summarizes the values to which we set **sin_addr** and **sin_port**, or **sin6_addr** and **sin6_port**, depending on the desired result.

IP address	Port	Result
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

- If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called.
- If we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

WILDCARD ADDRESS AND INADDR_ANY

- With IPv4, the *wildcard* address is specified by the constant INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address.

  struct sockaddr_in servaddr;

  servaddr.sin_addr.s_addr = htonl (INADDR_ANY); / wildcard */*

- While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure.

struct sockaddr_in6 serv;

serv.sin6_addr = in6addr_any; / wildcard */*

- The system allocates and initializes the **in6addr_any** variable to the constant IN6ADDR_ANY_INIT.
- The value of INADDR_ANY (0) is the same in either network or host byte order, so the use of **htonl** is not really required. But, since all the **INADDR_constants** defined by the <netinet/in.h> header are defined in host byte order, we should use **htonl** with any of these constants.

LISTEN()

The connect function is used by a TCP client to establish a connection with a TCP server
Int listen(int sockfd, int backlog);
Change socket state for TCP server.

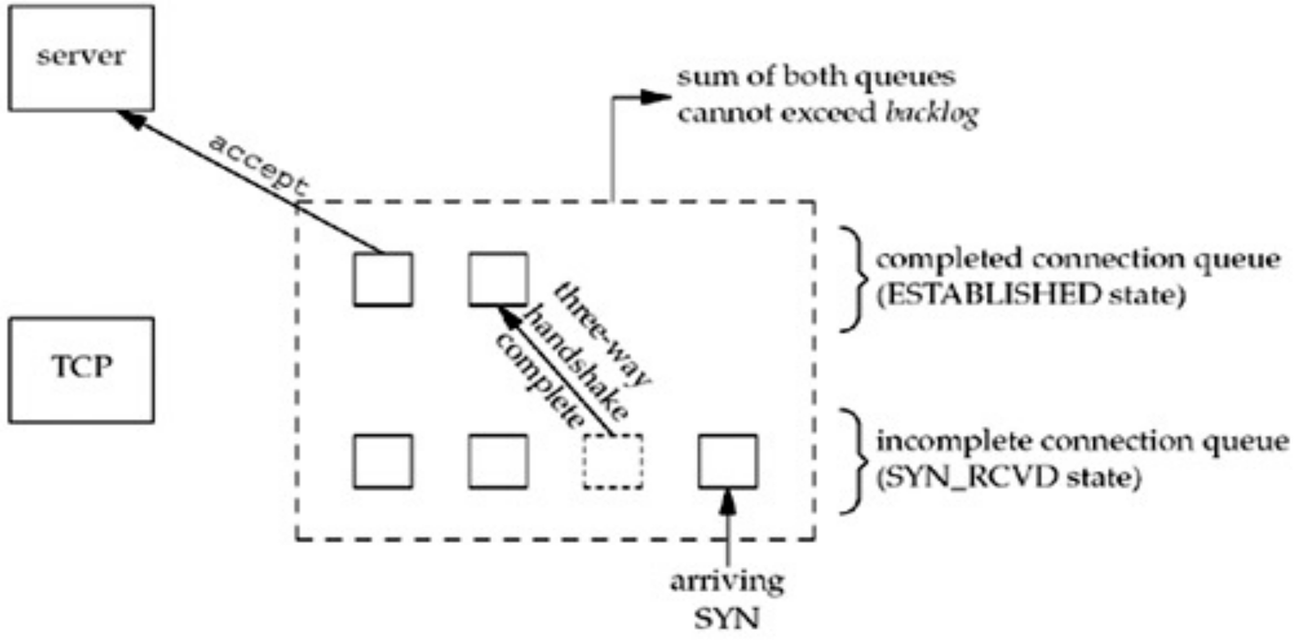
- *sockfd* is socket descriptor from socket()
- *backlog* is maximum number of *incomplete* connections
  - historically 5
  - rarely above 15 on a even moderate Web server!

Sockets default to active (for a client)
- change to passive so OS will accept connection

- An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake.
- A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed.
- In terms of the TCP state transition diagram, the call to listen moves the socket from the CLOSED state to the LISTEN state.

CONNECTION QUEUES

- For ***backlog*** argument, we must realize that for a given listening socket, the kernel maintains two queues:
  - An **incomplete connection queue**, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state.
  - A **completed connection queue**, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state.


server
accept
sum of both queues cannot exceed *backlog*
TCP
three-way handshake complete
completed connection queue (ESTABLISHED state)
incomplete connection queue (SYN_RCVD state)
arriving SYN

CONNECT ( )

```
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Connect to server.

- *sockfd* is socket descriptor from socket()
- *servaddr* is a pointer to a structure with:
  - *port number* and *IP address*
  - must be specified (unlike bind())
- *addrlen* is length of structure
- returns socket descriptor if ok, -1 on error
- The client does not have to call bind before calling connect, the kernel will choose both an ephemeral port and the source IP address if necessary.
- ETIMEDOUT: host doesn't exist (connection timed out)
- ECONNREFUSED: no process is waiting for connections on the server host at the port specified
- EHOSTUNREACH: no route to host

- In the case of a TCP socket, the connect function initiates TCP's three-way handshake.
- The function returns only when the connection is established or an error occurs.

Possible Error:

1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.
   - connect moves from the CLOSED state (the state in which a socket begins when it is created by the socket function) to the SYN_SENT state, and then, on success, to the ESTABLISHED state.
   - If connect fails, the socket is no longer usable and must be closed. We cannot call connect again on the socket.
2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (the server process is probably not running). This is a **hard error** and the error ECONNREFUSED is returned to the client as soon as the RST is received. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are:
   - When a SYN arrives for a port that has no listening server.
   - When TCP wants to abort an existing connection.
   - When TCP receives a segment for a connection that does not exist.
3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a **soft error**. The client kernel saves the message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH.

ACCEPT()

```
int accept(int sockfd, struct sockaddr  *cliaddr, socklen_t *addrlen);
```

Return next completed connection.

- *sockfd* is socket descriptor from `socket()`
- *cliaddr* and *addrlen* return protocol address from client
- returns brand new descriptor, created by the kernel. This new descriptor refers to the TCP connection with the client.
- The **listening socket** is the first argument (*sockfd*) to accept (the descriptor created by socket and used as the first argument to both bind and listen).
- The **connected socket** is the return value from accept the connected socket.
- A given server normally creates only one listening socket, which then exists for the lifetime of the server.
- The kernel creates one connected socket for each client connection that is
- When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values:

- An integer return code that is either a new socket descriptor or an error indication,
- The protocol address of the client process (through the *cliaddr* pointer),
- The size of this address (through the *addrlen* pointer).

SENDING AND RECEIVING

```
int recv(int sockfd, void *buff, size_t mbytes, int
 flags);
```

```
int send(int sockfd, void *buff, size_t mbytes, int
 flags);
```

Same as `read()` and `write()` but for *flags*

- MSG_DONTWAIT (this send non-blocking)
- MSG_OOB (out of band data, 1 byte sent ahead)
- MSG_PEEK (look, but don’t remove)
- MSG_WAITALL (don’t give me less than max)
- MSG_DONTROUTE (bypass routing table)

CLOSE()

```
int close(int sockfd);
```

Close socket for use.

sockfd is socket descriptor from socket()

closes socket for reading/writing

- returns (doesn't block)
- attempts to send any unsent data
- socket option SO_LINGER
  - block until data sent
  - or discard any remaining data
- returns -1 if error
- The default action of close with a TCP socket is to mark the socket as closed and return to the process immediately.
- The socket descriptor is no longer usable by the process. It cannot be used as an argument to read or write.
- But ,TCP will try to send any data that is already queued to be sent to the other end, and after this occurs, the normal TCP connection termination sequence takes place.

GETSOCKNAME AND GETPEERNAME FUNCTIONS

#include <sys/socket.h>

int getsockname (int sockfd, struct sockaddr* localaddr, socklen_t * addrlen)

Int getpeername (int sockfd, struct sockaddr* peeraddr, socklen_t * addrlen)

•**getsockname** returns local protocol address associated with a socket

•**getpeername** returns the foreign protocol address associated with a socket

•**getsockname** will return local IP/Port if unknown (TCP client calling connect without a bind, calling a bind with port 0, after accept to know the connection local IP address, but use connected socket)

GETSOCKNAME AND GETPEERNAME FUNCTIONS

Why getsockname() and getpeername() is required?

- After connect successfully returns in a TCP client that does not call bind, **getsockname**() returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number), **getsockname** returns the local port number that was assigned.
- **getsockname** can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address, once a connection is established with a client, the server can call **getsockname** to obtain the local IP address assigned to the connection. The socket descriptor argument in this call must be that of the connected socket, and not the listening socket.
- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call **getpeername**.

EXAMPLE

Daytime server and daytime client - DONE

RWServer and RWClient – DONE

Concurrent Eco server and Client - DONE

PROJECT 1: IN GROUP OF 2 STUDENTS

Project 1: Design and implement TCP Client and TCP Server applications using unix network programming for providing registration numbers to the client.

This project can be done in group of two students.

Requirement:

1. The server contains a buffer in the linked list where each node contains
2. information about the client. (node should contain Id, Name , Registration no and msg_sent_count)
3. The client establishes a connection to the server with its **unique ID number** (**use your roll no**) sent via command-line argument during sending a connection request.
4. The Server sends user information on the basis of the provided ID number. Search the above ID number in the linked list and prepare the message and send it to the client and increment msg_sent_count.
5. After sending client information from the server, print all information of the respective client in the server console as well.

7. After receiving the first message by the client, it prints the received message in its console and sends a ‘Thank You’ message to the server.
8. Server sends asking information immediately after receiving ‘Thank You’ message from client that if he wishes to see the same information again.
9. If Server receives Again messages from the client it sends the same information again and increments msg_sent_count for that client.
10. Your server must support multiple simultaneous connections.
11. If a client wants to quit, take input ‘quit’ from the console, and the client terminates. If a client wants to see the same information again, input ‘Again’ from the console and send a request to the server.
12. Your server must not support more than 10 simultaneous clients.

Use select() statement to check if the input is coming from the keyboard or from which of the clients.

- The server is started with: **tcpInfoServer –p port**
- The client is started with: **tcpInfoClient –h serveraddress -p port -n IDNumber**

Use the following messaging rules:

Information messages from the server to client

1Welcome <<Yourname>>, your unique identification key is <<Registration No.>> (Replace <<Yourname>> and <<Registration No>> to your roll no and registration no)

3Do you want to see your information again?

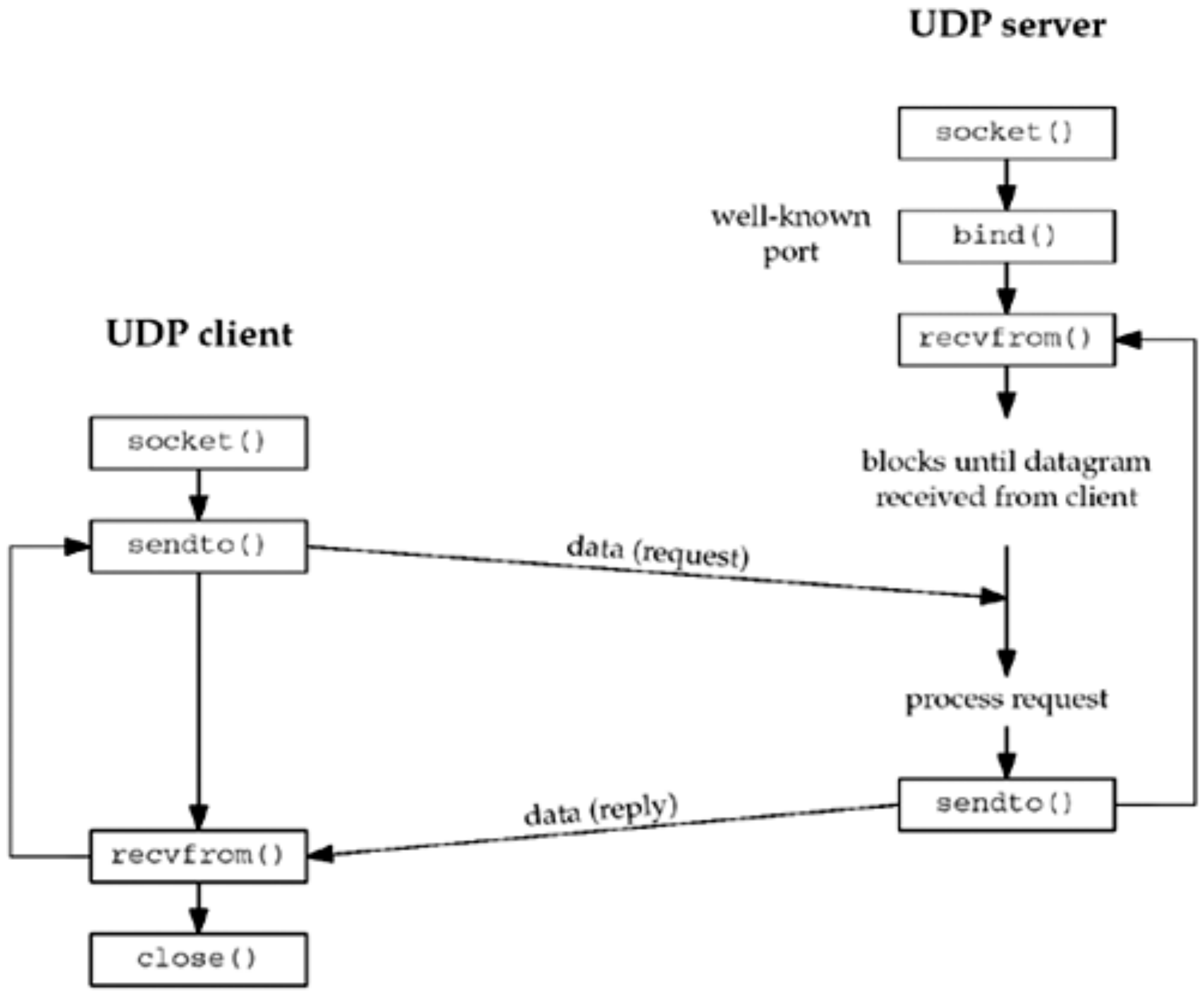
Message from client to server to see information repetitively

2Thank You

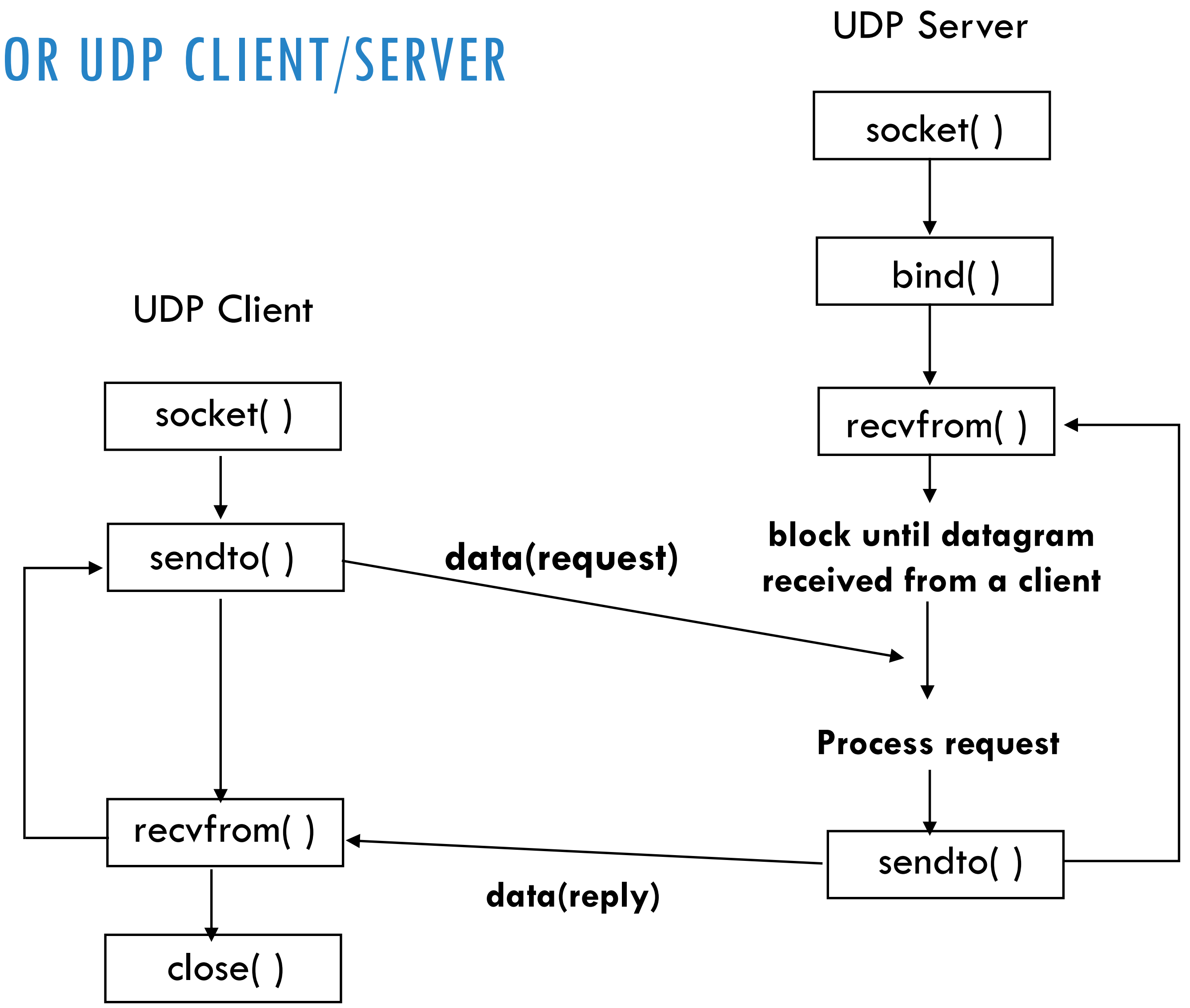
4Again

Develop necessary protocols if required to complete the application.

UNIX/INTERNET DOMAIN SOCKET (UDP CONNECTION)



SOCKET FUNCTIONS FOR UDP CLIENT/SERVER



RECVFROM AND SENDTO FUNCTION

```
#include<sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                               struct sockaddr *from, socklen_t *addrlen);
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
                            const struct sockaddr *to, socklen_t addrlen);
//Both return: number of bytes read or written if OK,-1 on error
```

- The first three arguments, *sockfd*, *buff*, and *nbytes*, are identical to the first three arguments for **read** and : descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.
- The ***to*** argument for sendto is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent.
- The final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).
- The final two arguments to recvfrom are similar to the final two arguments to accept: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to sendto are similar to the final two arguments to connect: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).
- Both functions return the length of the data that was read or written as the value of the function. In the typical use of recvfrom, with a datagram protocol, the return value is the amount of user data in the datagram received.

CONNECT FUNCTION WITH UDP

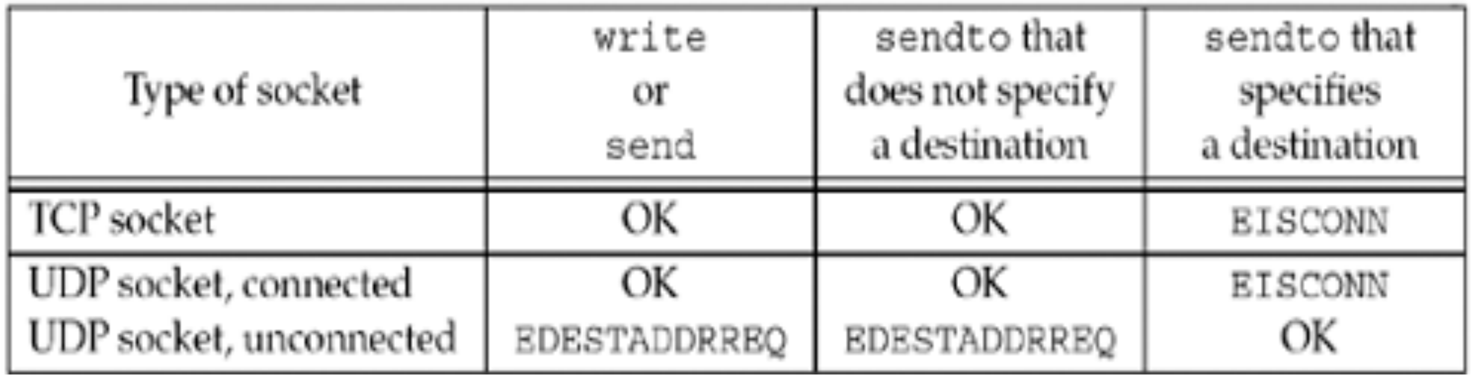
- We can call connect for a UDP socket. The kernel just checks for any immediate errors (e.g. an obviously unreachable destination), records the IP address and port number of the peer, and returns immediately to the calling process. Obviously, there is no three-way handshake.

- With this capability, we must now distinguish between
  - An unconnected UDP socket, the default when we create a UDP socket
  - A connected UDP socket, the result of calling connect on a UDP socket

- With a connected UDP socket, three things change, compared to the default unconnected UDP socket:
  - We can no longer specify the destination IP address and port for an output operation. We do not use **sendto**, but **write** or **send** instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by connect.
    - Similar to TCP, we can call sendto for a connected UDP socket, but we cannot specify a destination address. The fifth argument to sendto (the pointer to the socket address structure) must be a null pointer, and the sixth argument (the size of the socket address structure) should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.

CONNECT FUNCTION WITH UDP…

- We do not need to use **recvfrom** to learn the sender of a datagram, but **read**, **recv**, or **recvmsg** instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in connect. Datagrams destined to the connected UDP socket’s local protocol address (e.g., IP address and port) but arriving from a protocol address other than the one to which the socket was connected are not passed to the connected socket.
  - Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to connect to a multicast or broadcast address.
- Asynchronous errors are returned to the process for connected UDP sockets.

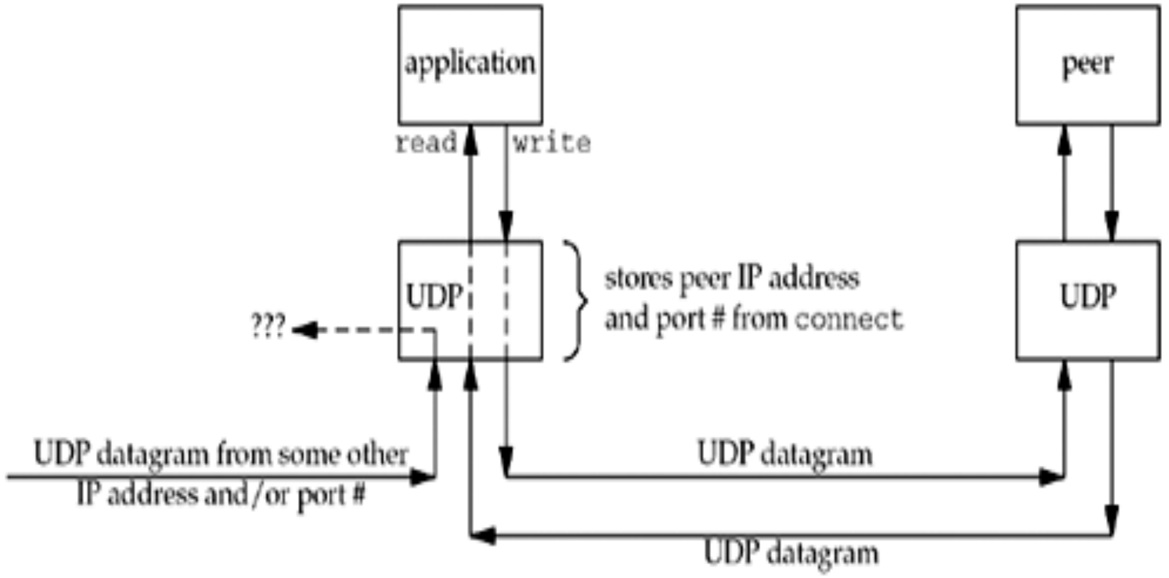


Type of socket	write or send	sendto that does not specify a destination	sendto that specifies a destination
TCP socket	OK	OK	EISCONN
UDP socket, connected	OK	OK	EISCONN
UDP socket, unconnected	EDESTADDRREQ	EDESTADDRREQ	OK

Table: TCP and UDP sockets: can a destination protocol address be specified?

CONNECT FUNCTION WITH UDP…

- The application calls connect, specifying the IP address and port number of its peer. It then uses read and write to exchange data with the peer.
- Datagrams arriving from any other IP address or port (??? in Figure) are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is connected. These datagrams could be delivered to some other UDP socket on the host. If there is no other matching socket for the arriving datagram, UDP will discard it and generate an ICMP "port unreachable" error.
- In summary, UDP client or server can call connect only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls connect, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP); in this case, both the client and server can call connect.


application
read
write
peer
UDP
stores peer IP address and port # from connect
UDP
???
UDP datagram from some other IP address and/or port #
UDP datagram
UDP datagram

UNIX DOMAIN SOCKETS/PROTOCOLS

- The Unix domain protocols are not an actual protocol suite, but a way of performing client/server communication on a single host using the same API that is used for clients and servers on different hosts.
- Two types of sockets are provided in the Unix domain: **stream sockets** (similar to TCP) and **datagram sockets** (similar to UDP).

Unix domain sockets are used for three reasons:

- On Berkeley-derived implementations, Unix domain sockets are often twice as fast as a TCP socket when both peers are on the same host.
- Unix domain sockets are used when passing descriptors between processes on the same host.
- Newer implementations of Unix domain sockets provide the client's credentials to the server, which can provide additional security checking.

UNIX DOMAIN SOCKET ADDRESS STRUCTURE

The Unix domain socket address structure, which is defined by including the <sys/un.h> header, is:

```
struct sockaddr_un {
            sa_family_t sun_family; /* AF_LOCAL */
            char sun_path[104]; /* null-terminated pathname */
};
```

- The pathname stored in the sun_path array must be null-terminated. The macro SUN_LEN is provided and it takes a pointer to a sockaddr_un structure and returns the length of the structure, including the number of non-null bytes in the pathname.

- The unspecified address is indicated by a null string as the pathname, that is, a structure with sun_path[0] equal to 0. This is the Unix domain equivalent of the IPv4 INADDR_ANY constant and the IPv6 IN6ADDR_ANY_INIT constant.

EXAMPLE: BIND OF UNIX DOMAIN SOCKET

```
int main(int argc, char ** argv[]) {
                int sockfd;
                socklen_t len;
                struct sockaddr_un addr1, addr2;
                if (argc != 2)
                                err_quit("usage: unixbind <pathname>");
                sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
unlink (argv[1]); // OK if this fails
bzero(&addr,sizeof(addr1));
addr1.sun_family = AF_LOCAL;
strncpy(addr1.sun_path, argv[1], sizeof(addr1.sun_path) -1);
bind(sockfd, (SA *) &addr1, SUN_LEN(&addr1));
len = sizeof(addr2);
getsockname(sokfd, (SA *)  &addr2, &len);
printf("bound name = %s, returned len = %d\n", addr2_sun_path, len);
exit(0);
}
```

SOCKETPAIR() FUNCTION

- The **socketpair()** function creates two sockets that are connected together. This function applies only to Unix domain sockets.

```
 #include <sys/socket.h>

  int socketpair(int family, int type, int protocol, int sockfd[2]);

  Returns: nonzero if OK, -1 on error
```

- Family: AF_LOCAL and the protocol must be 0.
- Type: either SOCK_STREAM or SOCK_DGRAM. The two socket descriptors that are created are returned as sockfd[0] and sockfd[1];
- The two created sockets are unnamed; that is; there is no implicit bind involved.
- The result of **scoketpair()** with a type of SOCK_STREAM is called a stream pipe. The stream pipe is full-duplex; that is, both descriptors can be read and written.

SOCKET FUNCTIONS

Differences and restrictions in the socket functions when using Unix domain sockets.

- The default file access permissions for a pathname created by bind should be 0777 (read, write, and execute by user group, group, and other), modified by the current umask value.
- The pathname associated with a Unix domain socket should be an absolute pathname, not a relative pathname.
- The pathname specified in a call to connect must be a pathname that is currently bound to an open Unix domain socket of the same type (stream or datagram).
- Unix domain stream sockets are similar to TCP sockets. They provide a byte stream interface to the process with no record boundaries.
- If a call to connect for a Unix domain stream socket finds that the listening socket's queue is full, ECONNREFUSED is returned immediately.
- Unix domain datagram sockets are similar to UDP sockets. They provide an unreliable datagram service that preserves record boundaries.
- Sending a datagram on an unbound Unix domain datagram socket does not bind a pathname to the socket. Calling connect for a Unix domain datagram socket does not bind a pathname to the socket.

NAME AND ADDRESS CONVERSION
DOMAIN NAME SYSTEM(DNS)

- The DNS is used primarily to map between hostnames and IP addresses.
- A hostname can be either a simple name, such as solaris or freebsd, or a fully qualified domain name (FQDN), such as solaris.unpbook.com.
- Entries in the DNS are known as resource records (RRs).
- Domain name is converted to IP address by contacting a DNS server by calling functions in a library known as the resolver.

GETHOSTBYNAME() FUNCTION

- This function is used to convert hostname to IP address.

```
#include <netdb.h>
```

struct hostent * gethostbyname(const char * hostname);

Returns: non-null pointer if OK, NULL on error with h_errno set

- The non-null pointer returned by this function points to the following **hostent** structure.

```
struct hostent {
 char  *h_name;       /* official (canonical) name of host */
 char **h_aliases;    /* pointer to array of pointers to alias names */
 int    h_addrtype;   /* host address type: AF_INET */
 int    h_length;     /* length of address: 4 */
 char **h_addr_list;  /* ptr to array of ptrs with IPv4 addrs */
};
```

- This function can return only IPv4 addresses.

GETHOSTBYNAME() FUNCTION …

- Gethostbyname() differs from the other socket functions that it does not set errno when an error occurs. Instead, it sets the global integer h_errno to one of the following constants defined by including <netdb.h>:

- HOST_NOT_FOUND
- TRY_AGAIN
- NO_RECOVERY
- NO_DATA (identical to NO_ADDRESS)

- The **NO_DATA** error means the specified name is valid, but it does not have an A record.

GETHOSTBYADDR() FUNCTION

- The function **gethostbyaddr()** takes a binary IPv4 address and tries to find the host-name corresponding to that address. This is the reverse of **gethostbyname**.

```
#include <netdb.h>

struct hostent *gethostbyaddr (const char *addr, socklen_t len, int family);

                   Returns: non-null pointer if OK, NULL on error with h_errno set
```

- The addr argument is not a char *, but is really a pointer to an in_addr structure containing the IPv4 address.
- The len is the size of this structure: 4 for an IPv4 address. The family argument is AF_INET.

GETSERVBYNAME() AND GETSERVBYPORT() FUNCTIONS

Services, like hosts, are often known by names, too.

```
#include <netdb.h>

struct servent *getservbyname (const char *servname, const char *protoname);
```

Returns: non-null pointer if OK, NULL on error

This function returns a pointer to the following structure.

```
struct servent {
char *s_name; /* official service name */
char **s_aliases; /* alias list */
int s-port; /* port number, network-byte order */
char *s_proto; /* protocol to use */
};
```

- The service name servname must be specified. If a protocol is also specified (protoname is a non-null pointer), then the entry must also have a matching protocol.

GETSERVBYPORT()

Getservbyport(), looks up a service given its port number and an optional protocol.

```
#include <netdb.h>

struct servent *getservbyport (int port, const char *protoname);
```

Returns: non-null pointer if OK, NULL on error

➢ The *port* value must be network byte ordered. Typical calls to this function could be as follows:

```
struct servent *sptr;
sptr = getservbyport (htons (53), "udp"); /* DNS using UDP */
sptr = getservbyport (htons (21), "tcp"); /* FTP using TCP */
 sptr = getservbyport (htons (21), NULL); /* FTP using TCP */
sptr = getservbyport (htons (21), "udp"); /* this call will fail */
```

GETADDRINFO() FUNCTION

- The gethostbyname and gethostbyaddr functions only support IPv4.
- The getaddrinfo supports both IPv4 and IPv6.
- The getaddrinfo function handles both name-to-address and service-to-port translation, and returns sockaddr structures instead of a list of addresses.
- These sockaddr structures can then be used by the socket functions directly.

```
#include <netdb.h>

int getaddrinfo (const char *hostname, const char *service, const struct addrinfo *hints, struct addrinfo **result) ;
```

Returns: 0 if OK, nonzero on error

- This function returns through the result pointer a pointer to a linked list of addrinfo structures, which is defined by including <netdb.h>.

```
struct addrinfo {
int ai_flags; /* AI_PASSIVE, AI_CANONNAME */
int ai_family; /* AF_xxx */
int ai_socktype; /* SOCK_xxx */
int ai_protocol; /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
socklen_t ai_addrlen; /* length of ai_addr */
char *ai_canonname; /* ptr to canonical name for host */
struct sockaddr *ai_addr; /* ptr to socket address structure */
struct addrinfo *ai_next; /* ptr to next structure in linked list */
};
```

- The hostname is either a hostname or an address string (dotted-decimal for IPv4 or a hex string for IPv6). The service is either a service name or a decimal port number string. hints is either a null pointer or a pointer to an addrinfo structure that the caller fills in with hints about the types of information the caller wants returned.

GAI_STRERROR FUNCTION

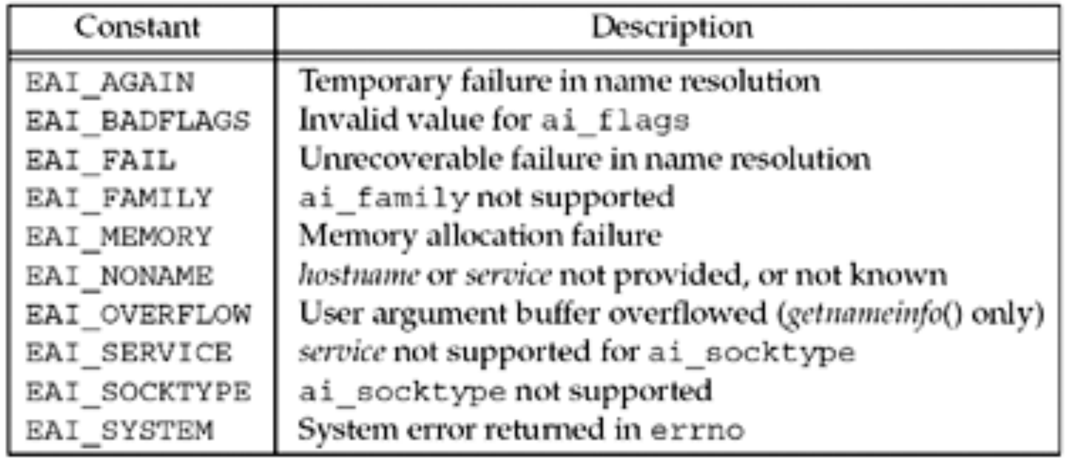
- ➢The non zero error return values from getaddrinfo have specific names and meanings.
- ➢The function gai_strerror takes one of these values (int) as an argument and returns a pointer to the corresponding error string.

#include <netdb.h>

const char *gai_strerror (int *error*);

Returns: pointer to string describing error message

Nonzero error return constants from getaddrinfo.



Constant	Description
EAI_AGAIN	Temporary failure in name resolution
EAI_BADFLAGS	Invalid value for ai_flags
EAI_FAIL	Unrecoverable failure in name resolution
EAI_FAMILY	ai_family not supported
EAI_MEMORY	Memory allocation failure
EAI_NONAME	*hostname* or *service* not provided, or not known
EAI_OVERFLOW	User argument buffer overflowed (*getnameinfo*() only)
EAI_SERVICE	*service* not supported for ai_socktype
EAI_SOCKTYPE	ai_socktype not supported
EAI_SYSTEM	System error returned in errno

FREEADDRINFO FUNCTION

- All the storage returned by getaddrinfo is obtained dynamically. This storage is returned by calling freeaddrinfo.

```
#include <netdb.h>

void freeaddrinfo (struct addrinfo *ai);
```

- ai should point to the first addrinfo structure returned by getaddrinfo.
- All the structures in the linked list are freed.

SIGNAL HANDLING

(POSIX) SIGNAL HANDLING

- A signal is a notification to a process that an event has occurred. Signals are sometimes called software interrupts. Signals usually occur asynchronously.
- Signals can be sent
  - By one process to another process (or to itself)
  - By the kernel to a process
- The **SIGCHLD** signal is one that is sent by the kernel whenever a process terminates, to the parent of the terminating process.
- Every signal has a disposition, which is also called the action associated with the signal. We set the disposition of a signal by calling the **sigaction** function. We have three choices for the disposition.
  1. We can provide a function that is called whenever a specific signal occurs. This function is called a signal handler and the action is called catching a signal. The two signals **SIGKILL** & **SIGSTOP** cannot be caught.
  2. We can ignore a signal by setting its disposition to **SIG_IGN**. The two signals **SIGKILL** and **SIGSTOP** cannot be ignored.
  3. We can set the default disposition for a signal by setting its disposition to **SIG_DFL**. There are few signals whose default disposition is to be ignored. **SIGCHLD** and **SIGURG**.

Function prototypes for signal handler functions

ANSI C:

```
void handler(int);
```

POSIX SA_SIGINFO:

```
void handler(int, siginfo_t *info, ucontext_t *uap);
```

SIGACTION FUNCTION

- sigaction returns the old action for the signal as the return value of the signal function.

```
#include<signal.h>
int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);
```

The struct sigaction is

```
struct  sigaction {
union __sigaction_u __sigaction_u;  // signal handler
              sigset_t sa_mask;     // signal mask to apply
              int     sa_flags;     // signal options
};
union __sigaction_u {
              void    (*___sa_handler)(int);
              void    (*___sa_sigaction)(int, siginfo_t *, void *);
};
#define sa_handler      __sigaction_u.__sa_handler
#define sa_sigaction    __sigaction_u.___sa_sigaction
```

- The sig is the signal to be captured. The act is the information about signal handling function, masked signal and flags. The oact is the information about the previous signal handling function, masked signal and flags.

- The sa_mask in struct sigaction takes the signal to be masked when the handler function is called on arrival of the signal. The sa_flags in struct sigaction sets flags. E.g., Setting sa_flags to **SA_SIGINFO** uses **POSIX** signal handler function.

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal

16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor
24	SIGXCPU	terminate process	cpu time limit exceeded (see
25	SIGXFSZ	terminate process	file size limit exceeded (see
26	SIGVTALRM	terminate process	virtual time alarm (see
27	SIGPROF	terminate process	profiling timer alarm (see
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

POSIX SIGNAL SEMANTICS

- Once a signal handler is installed, it remains installed.
- While a signal handler is executing, the signal being delivered is blocked. Furthermore, any additional signals that were specified in the **sa_mask** signal set passed to **sigaction** when the handler was installed are also blocked.
- If a signal is generated one or more times while it is blocked, it is normally delivered only one time after the signal is unblocked. That is, by default, Unix signals are not queued.
- It is possible to selectively block and unblock a set of signals using the **sigprocmask** function.

WAIT AND WAITPID FUNCTION

- The **wait()** and **waitpid()** functions are called to handle the terminated child.
- **wait()** and **waitpid()** both return two values: the return value of the function is the process ID of the terminated child, and the termination status of the child (an integer) is returned through the **statloc** pointer.

```
#include <sys/wait.h>
 pid_t wait(int *statloc);
 pid_t waitpid(pid_t pid, int *statloc,int options);
 Both return; process ID if OK, or -1 on error
```

- If there are no terminated children for the process calling wait, but the process has one or more children that are still executing, then wait blocks until the first child of the existing children terminates.
- The **waitpid()** gives us more control over which process to wait for and whether or not to block. First, the pid argument lets us specify the proces ID that we want to wait for. A value of -1 says to wait for the first of our children to terminate. The options argument lets us specify additional options.

SOCKET SYSTEM CALLS

Write to socket function variants

```
ssize_t write(int fildes, void *buf, size_t nbyte);
#include <sys/socket.h>
ssize_t send(int socket, const void *buffer, size_t length, int flags);
ssize_t sendmsg(int socket, const struct msghdr *message, int flags);

ssize_t sendto(int socket, const void *buffer, size_t length, int flags,
         const struct sockaddr *dest_addr, socklen_t dest_len);
// sendto =  write + connect
```

- Upon successful completion, the number of bytes that were sent is returned. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

Read from socket function variants

```
ssize_t read(int fildes, void *buf, size_t nbyte);
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t length, int flags);

ssize_t recvfrom(int socket, void *restrict buffer, size_t length, int flags, struct
sockaddr *restrict address, socklen_t *restrict address_len);
ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

- These calls return the number of bytes received, or -1 if an error occurred. For TCP sockets, the return value 0 means the peer has closed its half side of the connection.

ERROR RELATED FUNCTIONS

```
void perror(const char *); // print string equivalent of global error number
variable

char * strerror(int ); // returns string equivalent of given error number
```

PASSING (FILE) DESCRIPTORS

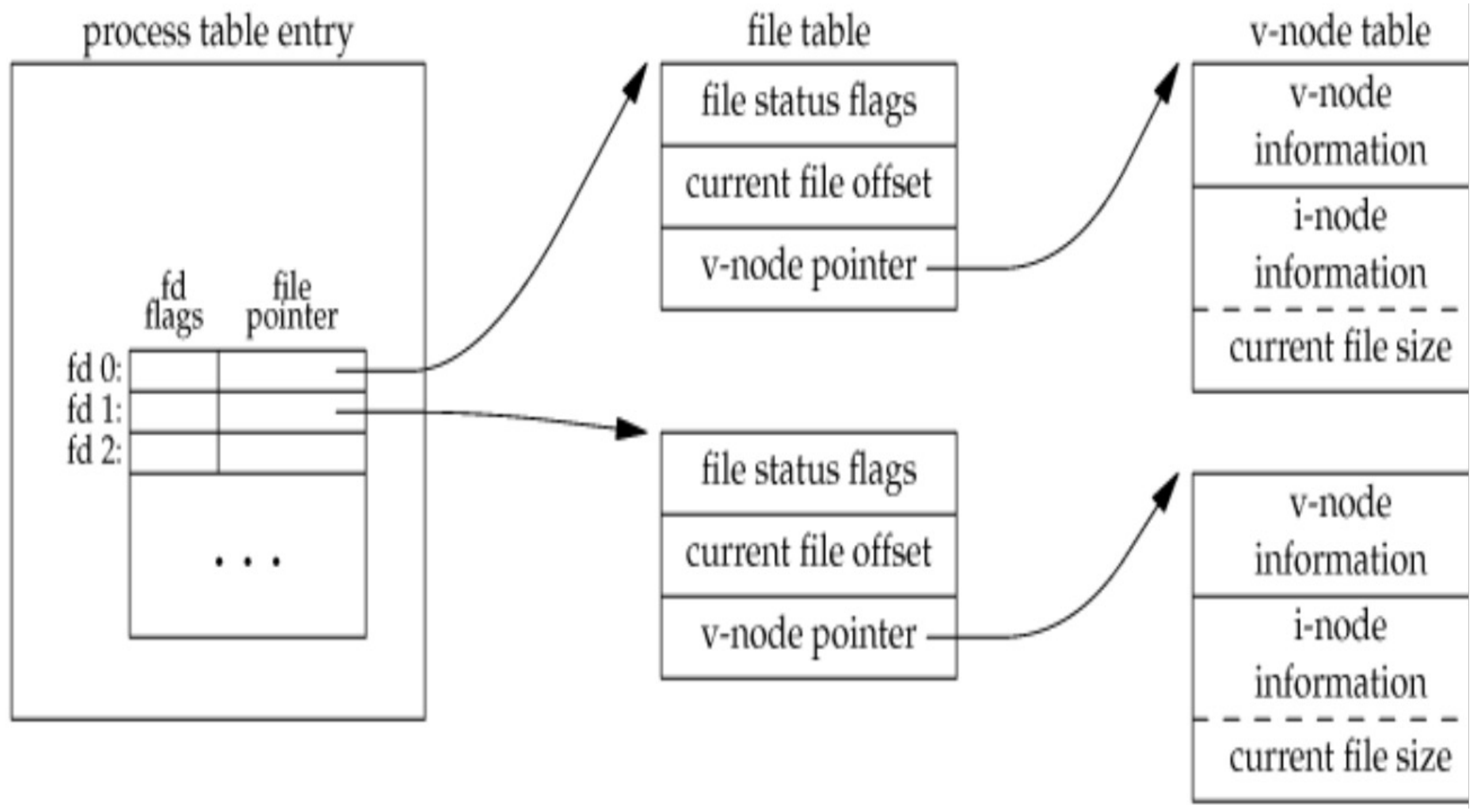
OPEN FILE IN UNIX SYSTEM

The kernel uses three data structures to represent an open file.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
   - The file descriptor flags
   - A pointer to a file table entry

2. The kernel maintains a file table for all open files. Each file table entry contains
   - The file status flags for the file, such as read, write, append, sync, and nonblocking;
   - The current file offset
   - A pointer to the v-node table entry for the file

3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, etc.



The figure shows a pictorial arrangement of these three tables for a single process that has two different files open.

PASSING A FILE DESCRIPTOR

- Passing an open descriptor from one process to another takes place as
  - A child sharing all the open descriptors with the parent after a call to fork
  - All descriptors normally remaining open when exec is called.
- Current Unix systems provide a way to pass any open descriptor from one process to any other process.
- The technique requires us to first establish a Unix domain socket between the two processes and then use **sendmsg** to send a special message across the Unix domain socket.
- This message is handled specially by the kernel, passing the open descriptor from the sender to the receiver.

Steps involved in passing a descriptor between two processes

- Create a Unix domain socket, either a stream socket or a datagram socket.
  - If the goal is to fork a child and have the child open the descriptor and pass the descriptor back to the parent, the parent can call **socketpair** to create a stream pipe that can be used to exchange the descriptor.
  - If the processes are unrelated, the server must create a Unix domain stream socket and bind a pathname to it, allowing the client to connect to that socket.

Steps involved in passing a descriptor between two processes(contd…)

- One process opens a descriptor by calling any of Unix functions that returns a descriptor. Any type of descriptor can be passed from one process to another.
- The sending process builds a **msghdr** structure containing the descriptor to be passed. The sending process calls **sendmsg** to send the descriptor across the Unix domain socket.
  - At this point, the descriptor is "in flight". Even if the sending process closes the descriptor after calling **sendmsg**, but before the receiving process calls **recvmsg**, the descriptor remains open for the receiving process. Sending a descriptor increments the descriptor's reference count by one.
- The receiving process calls **recvmsg** to receive the descriptor on the Unix domain socket. It is normal for the descriptor number in the receiving process to differ from the descriptor number in the sending process. Passing a descriptor involves creating a new descriptor in the receiving process that refers to the same file table entry within the kernel as the descriptor that was sent by the sending process.

I/O MODEL

INTRODUCTION 1/2

TCP echo client is handling two inputs at the same time: standard input and a TCP socket

- ➢when the client was blocked in a call to read, the server process was killed
- ➢server TCP sends FIN to the client TCP, but the client never sees FIN since the client is blocked reading from standard input
  - ✓We need the capability to tell the kernel that we want to be notified if one or more I/O conditions are ready.
  - ✓I/O multiplexing (**select**, **poll**, or newer **pselect** functions)

INTRODUCTION 2/2

Scenarios for I/O Multiplexing

- client is handling multiple descriptors (interactive input and a network socket).
- Client to handle multiple sockets (rare)
- TCP server handles both a listening socket and its connected socket.
- Server handle both TCP and UDP.
- Server handles multiple services and multiple protocols

I/O MODELS

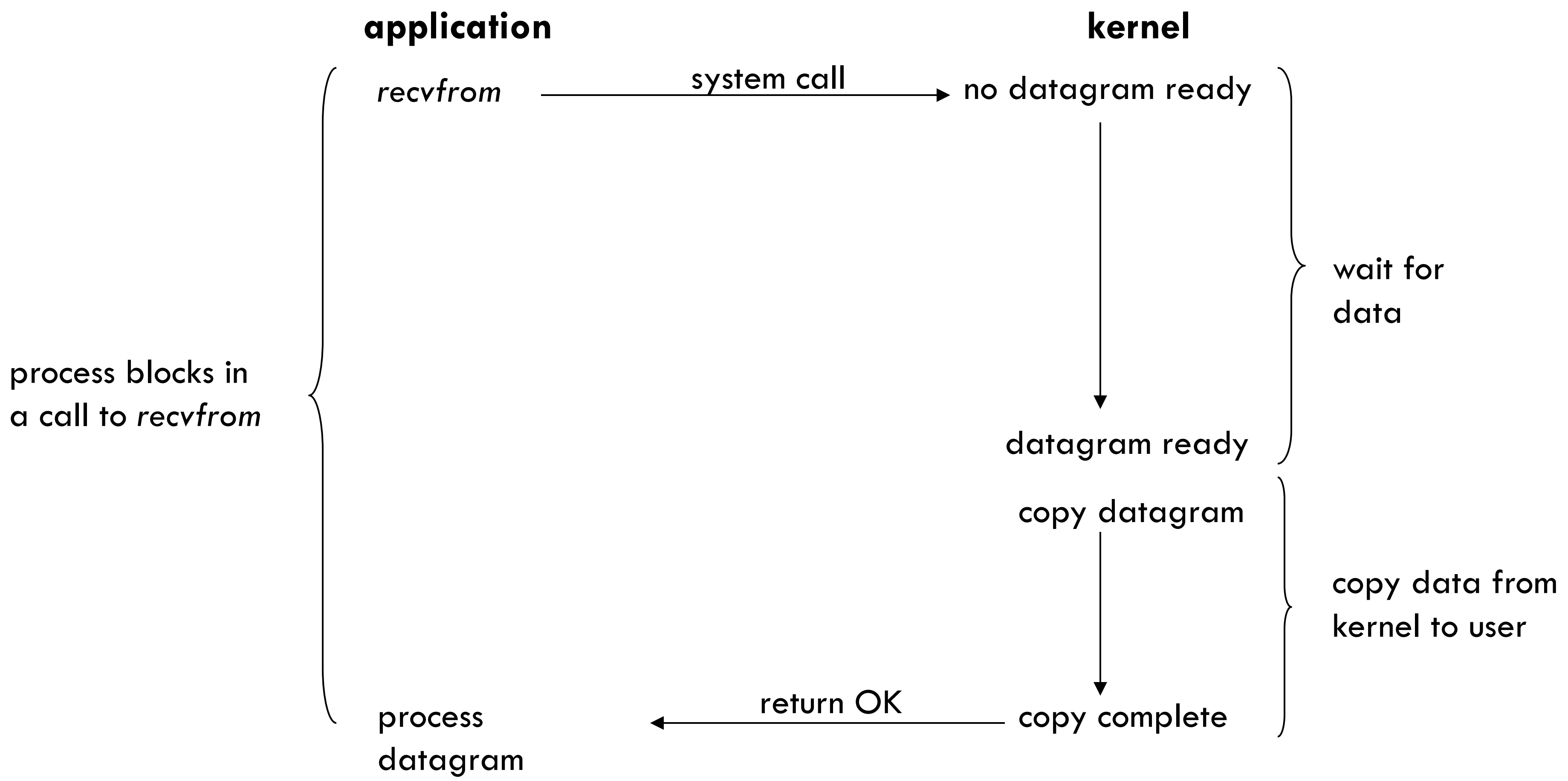
Models

1. Blocking I/O
2. Nonblocking I/O
3. I/O multiplexing(**select** and **poll**)
4. Signal driven I/O (**SIGIO**)
5. Asynchronous I/O

Two *distinct phases* for an input operation

- Waiting for the data to be ready (for a socket, wait for the data to arrive on the network, then copy into a buffer within the kernel)
- Copying the data from the kernel to the process (from kernel buffer into application buffer)

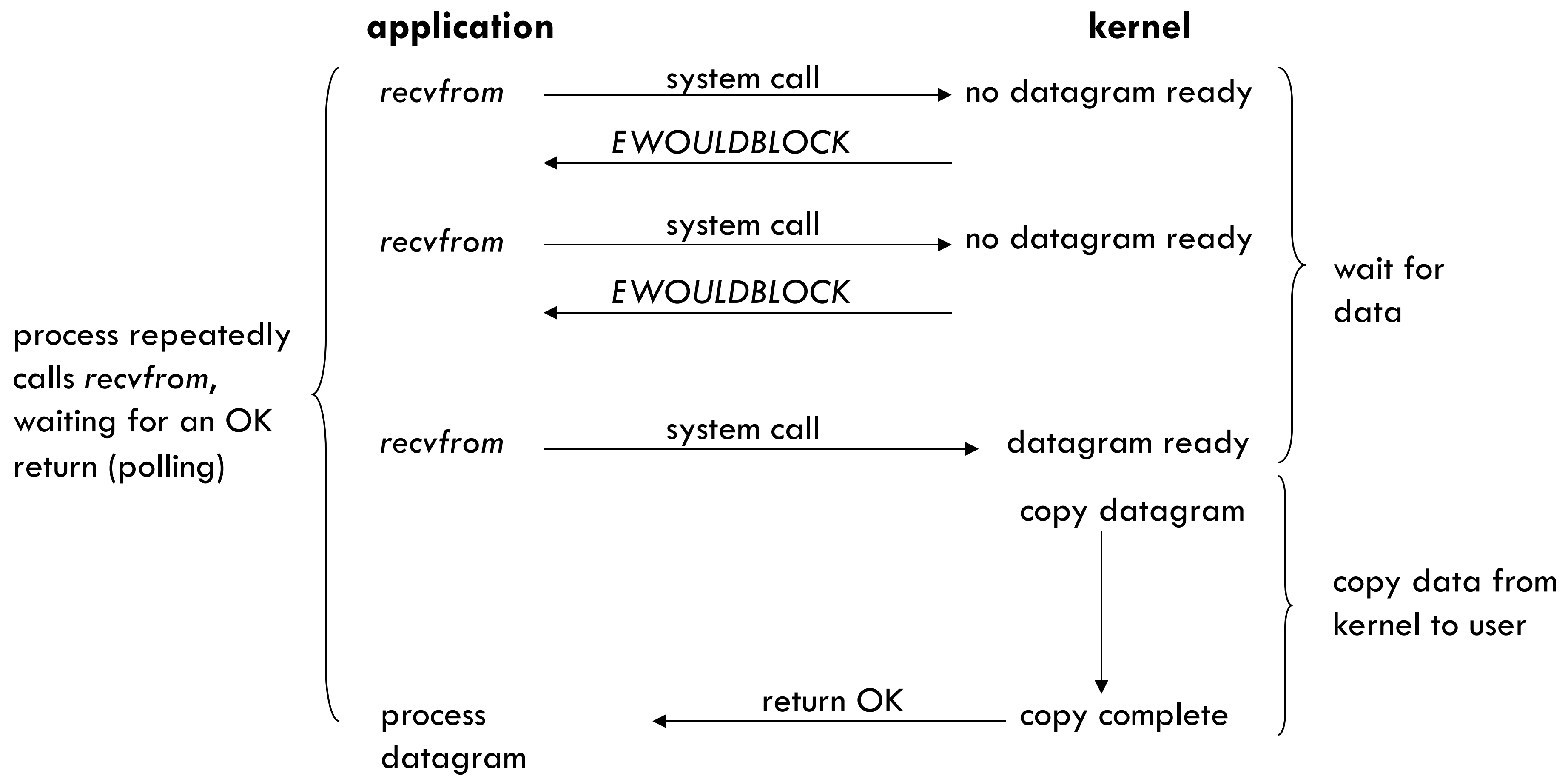
BLOCKING I/O MODEL



BLOCKING I/O MODEL...

- By default, all sockets are blocking.
- The process calls **recvfrom** and the system call does not return until the datagram arrives and is copied into our application buffer, or an error occurs.
- We say that our process is blocked the entre time from when it calls **recvfrom** until it returns.
- When **recvfrom** returns successfully, our application process the datagram.

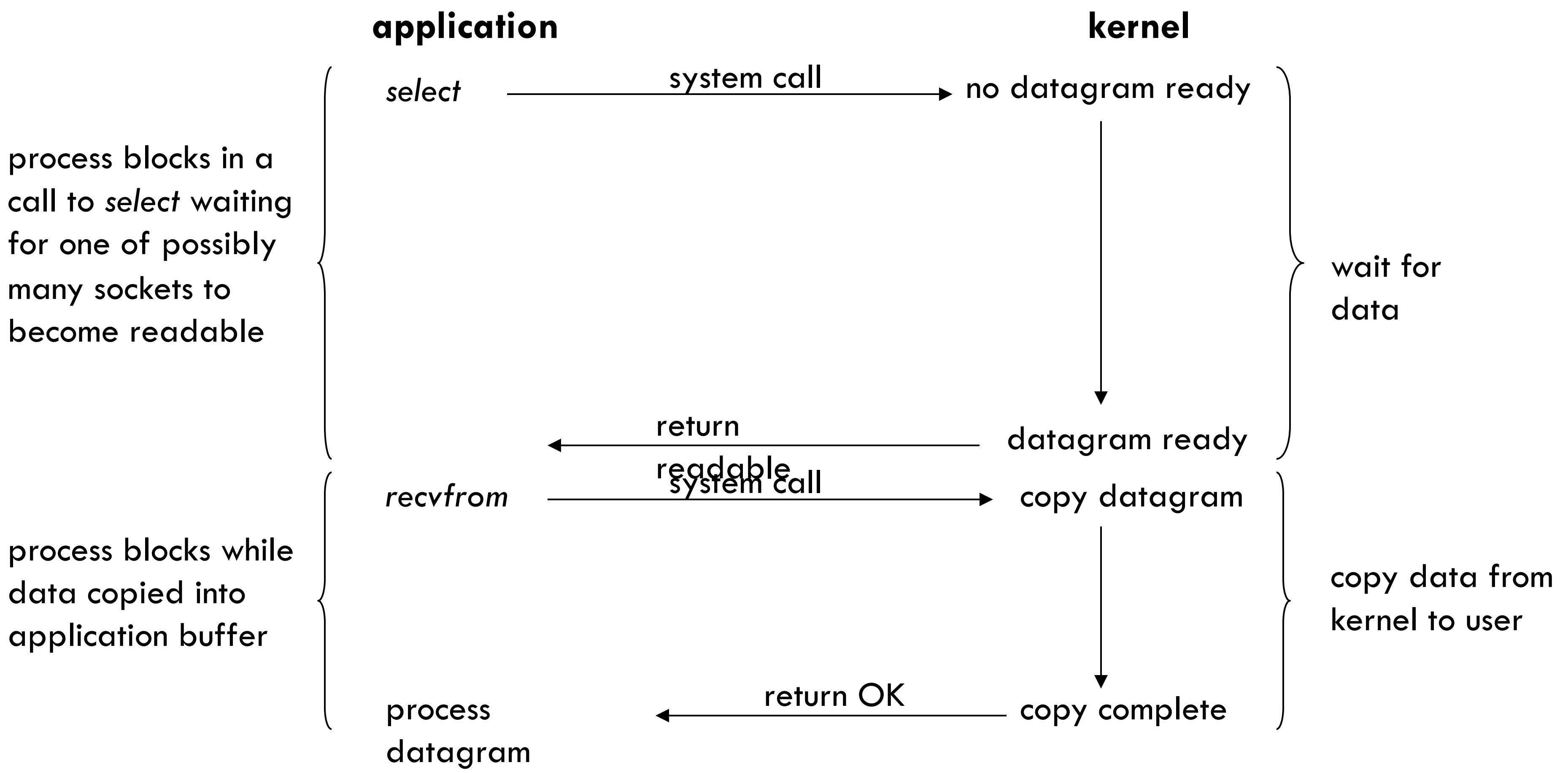
NON-BLOCKING I/O MODEL



NON-BLOCKING I/O MODEL…

- When a socket is non-blocking, It instruct the kernel as “when an I/O operation that the process requests cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead.”
- The first three times that we call **recvfrom**, there is no data to return, so the kernel immediately returns an error of EWOULDBLOCK instead.
- The fourth time we call **recvfrom**, a datagram is ready, it is copied into our application buffer, and **recvfrom** returns successfully.
- We then process data. When an application sits in a loop calling **recvfrom** on a non-blocking descriptor like this, it is called polling.
- The application is continually polling the kernel to see if some operation is ready. This is often a waste of CPU time.

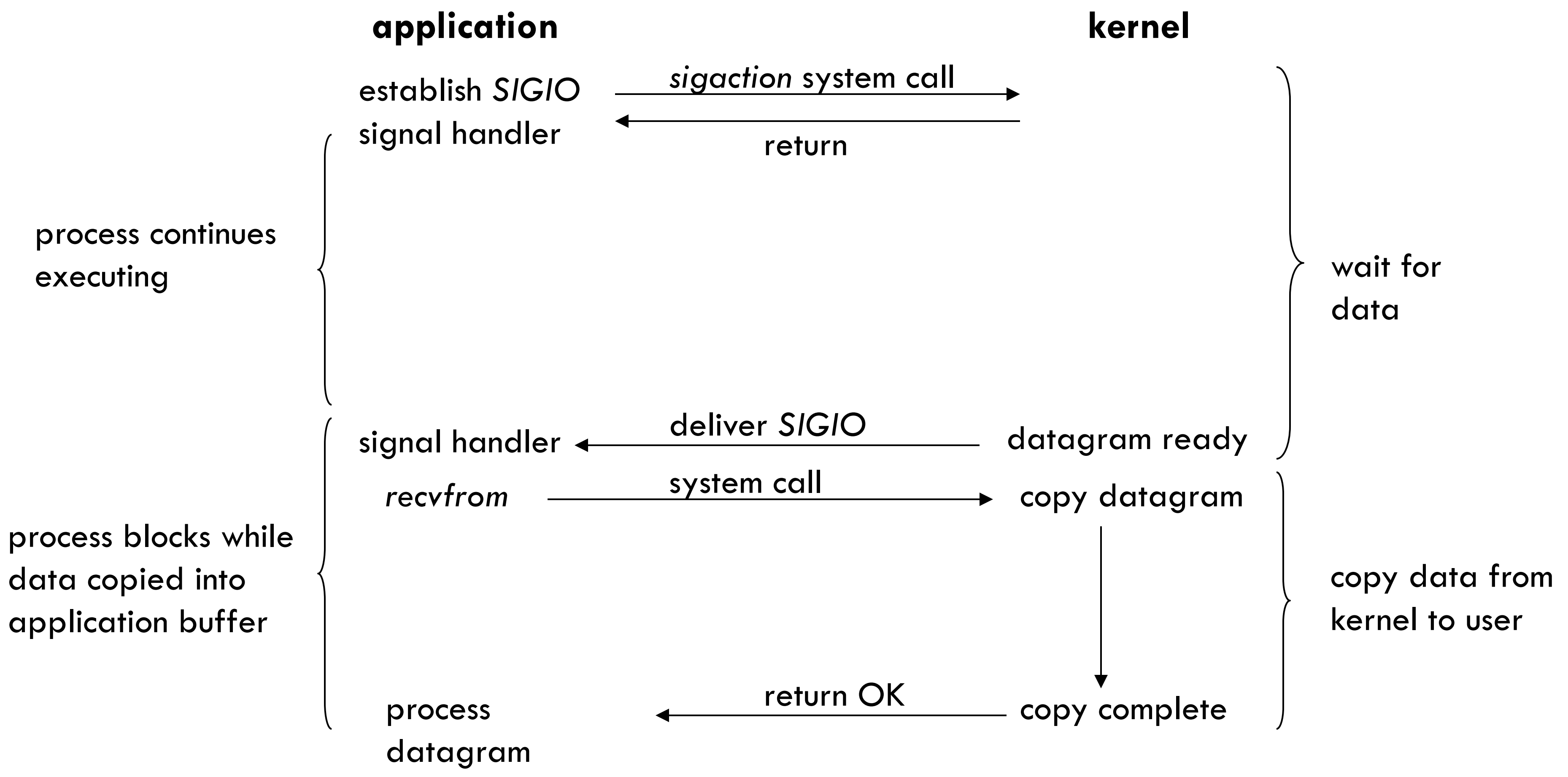
I/O MULTIPLEXING MODEL


application
kernel
select
system call
no datagram ready
process blocks in a call to *select* waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

I/O MULTIPLEXING MODEL...

- With I/O multiplexing, we call **select** or **poll** and block in one of these two system calls, instead of blocking in the actual I/O system call.
- We block in a call to **select**, waiting for the datagram socket to be readable.
- When **select** returns that the socket is readable, we then call **recvfrom** to copy the datagram into our application buffer.
- With **select**, we can wait for more than one descriptor to be ready.

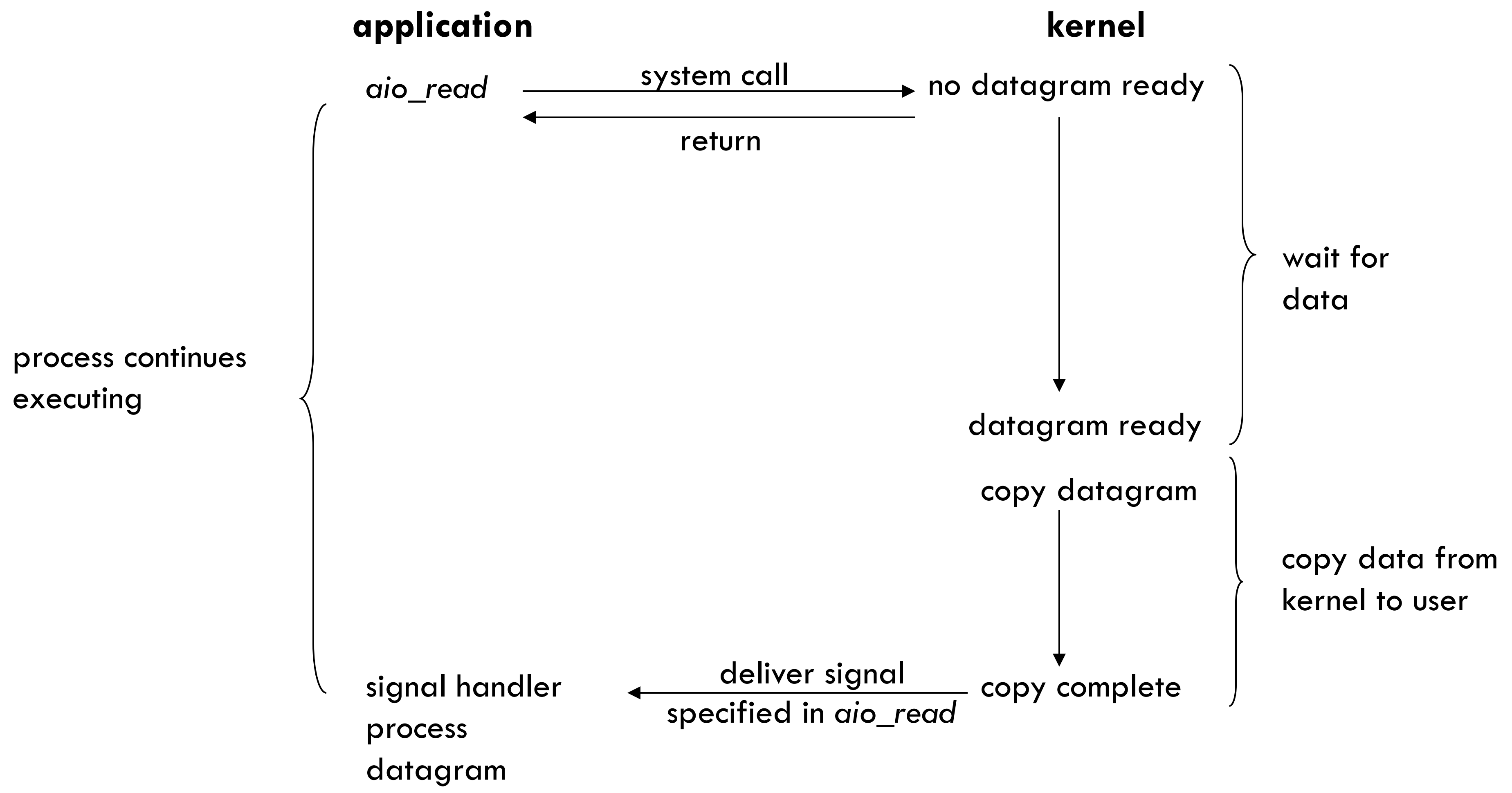
SIGNAL DRIVEN I/O MODEL



SIGNAL DRIVEN I/O MODEL…

- A **SIGIO signal** is used to tell the kernel when the descriptor is ready. We call this signal-driven I/O.
- We first enable the socket for the signal-driven I/O and install a signal handler using the **sigaction** system call.
- The return from this system call is immediate and our process continues; it is not blocked.
- When the datagram is ready to be read, the **SIGIO** signal is generated for our process. We can then read the data.
- The advantage of this model is that we are not blocked while waiting for the datagram to arrive.
- The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

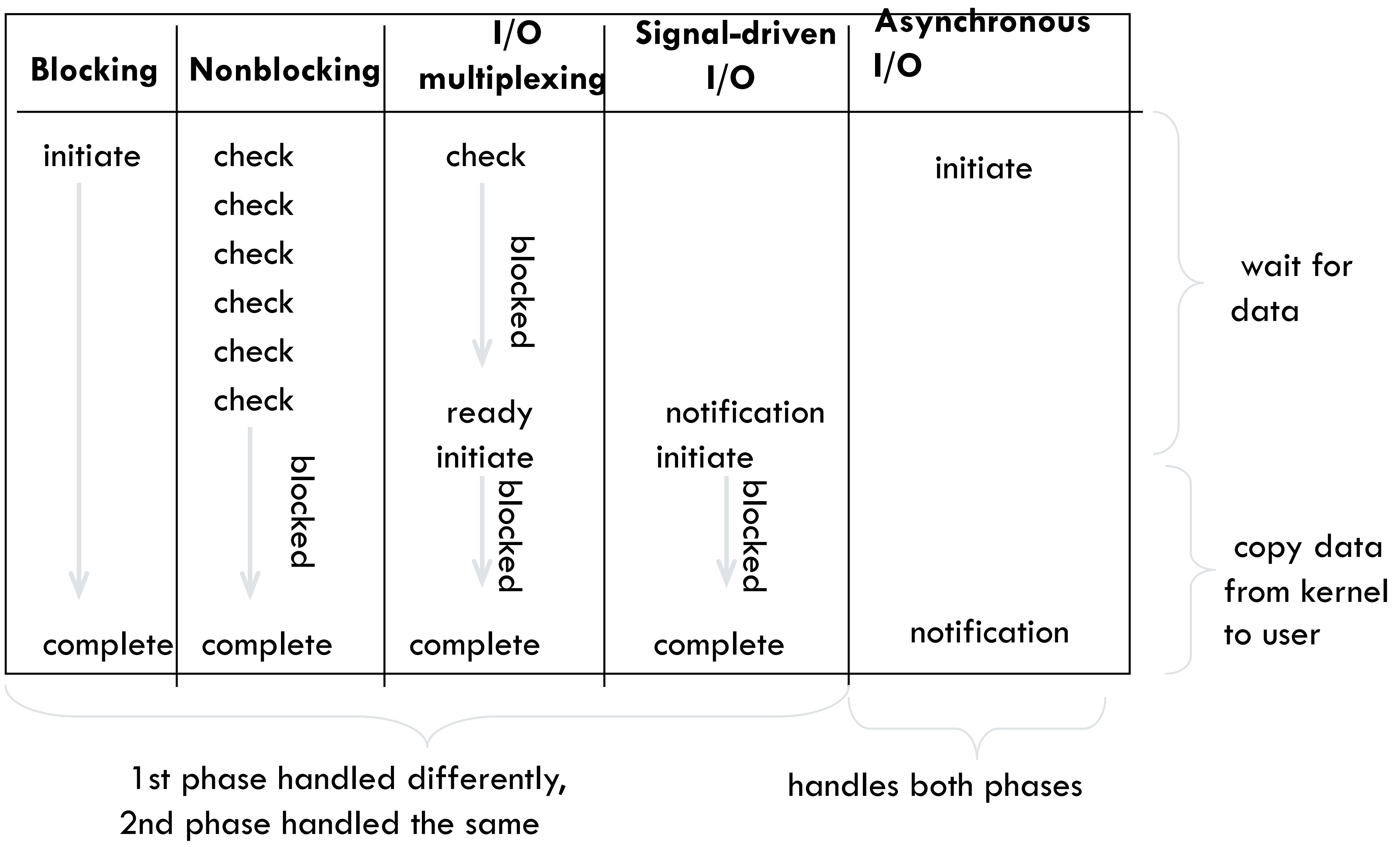
ASYNCHRONOUS I/O MODEL


application
kernel
aio_read
system call
no datagram ready
return
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
signal handler process datagram
deliver signal specified in aio_read
copy complete

ASYNCHRONOUS I/O MODEL

- ❖Asynchronous I/O is defined by the POSIX specification.
- ❖These functions work by telling the kernel to start the operation and to notify us when the entire operation (including the copy of the data from the kernel to our buffer) is complete.
- ❖The main difference between this model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells us when an I/O operation can be initiated, but with asynchronous I/O, the kernel tells us when an I/ operation is complete.
- ❖We call **aio_read** and pass the kernel the descriptor, buffer pointer, buffer size, file offset, and how to notify us when the entire operation is complete.
- ❖This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

COMPARISON OF THE I/O MODELS



SYNCHRONOUS I/O , ASYNCHRONOUS I/O

Synchronous I/O

- causes the requesting process to be blocked until that I/O operation (recvfrom) completes. (blocking, nonblocking, I/O multiplexing, signal-driven I/O)

Asynchronous I/O

- does not cause the requesting process to be blocked

SELECT FUNCTION

- Allows the process to instruct the kernel to *wait for any one of multiple events to occur* and to wake up the process only when one or more of these events occurs or *when a specified amount of time has passed*.
- What descriptors we are interested in (readable ,writable , or exception condition) and how long to wait?

```
#include <sys/select.h>
     #include <sys/time.h>
     int select (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set
 *exceptset, const struct timeval *);
//Returns: +ve count of ready descriptors, 0 on timeout, -1 on error
struct timeval{
          long  tv_sec;   /* seconds */
          long  tv_usec; /* microseconds */ }
```

- The final argument, timeout, tells the kernel how long to wait for one of the specified file descriptors to become ready. A timeval structure specifies the number of seconds and microseconds.

POSSIBILITIES FOR SELECT FUNCTION

1. Wait forever : return only when descriptor (s) is ready (specify **timeout** argument as NULL)
2. wait up to a fixed amount of time: Return when one of the specified descriptors is ready for I/O, but do not wait beyond the number of seconds and microseconds specified in the timeval structure pointed to by the timeout argument.
3. Do not wait at all : return immediately after checking the descriptors(called Polling) (specify **timeout** argument as pointing to a **timeval** structure where the timer value is 0)

❖The wait is normally interrupted if the process catches a signal and returns from the signal handler

- **select** might return an error of **EINTR**
- Actual return value from function = -1

RETURN VALUE OF SELECT

- Select() returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires, select() returns 0. If select() returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified and the global variable errno will be set to indicate the error.

Error	**Description**
[EAGAIN]	The kernel was (perhaps temporarily) unable to allocate the requested number of file descriptors.
[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.
[EINVAL]	ndfs is greater than FD_SETSIZE and _DARWIN_UNLIMITED_SELECT is not defined.

SELECT FUNCTION DESCRIPTOR ARGUMENTS

readset → descriptors for checking readable

writeset → descriptors for checking writable

exceptset → descriptors for checking exception conditions (2 exception conditions)

- ✓arrival of out of band data for a socket
- ✓the presence of control status information to be read from the master side of a pseudo terminal (Ignore)

If you pass the 3 arguments as NULL, you have a high precision timer than the sleep function

DESCRIPTOR SETS

Array of integers : each bit in each integer correspond to a descriptor (**fd_set**)

4 macros

```
➢void  FD_ZERO(fd_set *fdset);         /* clear all bits in fdset */
➢void  FD_SET(int fd, fd_set *fdset); /* turn on the bit for fd in fdset */
➢void  FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset*/
➢int    FD_ISSET(int fd, fd_set *fdset);/* is the bit for fd on in fdset ? */
```

EXAMPLE OF DESCRIPTOR SETS MACROS

```
fd_set  rset;

FD_ZERO(&rset);            /*all bits off : initiate*/
FD_SET(1, &rset);          /*turn on bit fd 1*/
FD_SET(4, &rset);          /*turn on bit fd 4*/
FD_SET(5, &rset);          /*turn on bit fd 5*/
```

maxfdp1 argument to select function

- specifies the number of descriptors to be tested.
- Its value is the maximum descriptor to be tested, plus one. (hence maxfdp1)
  - Descriptors 0, 1, 2, up through and including **maxfdp1**-1 are tested
  - example: interested in **fds** 1,2, and 5 → **maxfdp1** = 6
  - Your code has to calculate the **maxfdp1** value constant **FD_SETSIZE** defined by including **<sys/select.h>**
  - is the number of descriptors in the **fd_set** datatype. (often = 1024)

Value-Result arguments in select function

- Select modifies descriptor sets pointed to by **readset**, **writeset**, and **exceptset** pointers
- On function call
  - Specify value of descriptors that we are interested in

On function return

- Result indicates which descriptors are ready

Use **FD_ISSET** macro on return to test a specific descriptor in an **fd_set** structure

- Any descriptor not ready will have its bit cleared
- You need to turn on all the bits in which you are interested on all the descriptor sets each time you call **select**

CONDITION FOR A SOCKET TO BE READY FOR *SELECT*

Condition	Readable?	writable?	Exception?
Data to read	●		
read-half of the connection closed	●		
new connection ready for listening socket	●		
Space available for writing		●	
write-half of the connection closed		●	
Pending error	●	●	
TCP out-of-band data			●

PSELECT() FUNCTION

#include <sys/select.h>

#include <signal.h>

#include <time.h>

*int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timespec *timeout, const sigset_t *sigmask);*

/ Returns: count of ready descriptors, 0 on timeout, –1 on error */*

- pselect contains two changes from the normal select function:
- **pselect** uses the **timespec** structure (another POSIX invention) instead of the **timeval** structure. The **tv_nsec** member of the newer structure specifies nanoseconds, whereas the **tv_usec** member of the older structure specifies microseconds.

```
struct timespec {
        time_t tv_sec; /* seconds */
        long tv_nsec; /* nanoseconds */
};
```

- **pselect** adds a sixth argument: a pointer to a signal mask.
  - This allows the program to disable the delivery of certain signals, test some global variables that are set by the handlers for these now-disabled signals, and then call pselect, telling it to reset the signal mask.

```
if (intr_flag)
        handle_intr(); /* handle the signal */
/* signals occurring in here are lost */
if ( (nready = select( ... )) < 0) {
        if (errno == EINTR) {
                if (intr_flag)
                handle_intr();
        }
... }
```

- The problem is that between the test of **intr_flag** and the call to select, if the signal occurs, it will be lost if select blocks forever.

```
sigset_t newmask, oldmask, zeromask;
sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
 /* block SIGINT */
if (intr_flag)
        handle_intr(); /* handle the signal */
 if ( (nready = pselect ( ... , &zeromask)) < 0) {
        if (errno == EINTR) {
                if (intr_flag) handle_intr ();
        }
... }
```

Before testing the **intr_flag** variable, we block SIGINT.
When **pselect** is called, it replaces the signal mask of the process with an empty set (i.e., zeromask) and then checks the descriptors, possibly going to sleep. But when pselect returns, the signal mask of the process is reset to its value before pselect was called (i.e., SIGINT is blocked).

POLL FUNCTION

```
#include <poll.h>
 int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

- returns count of ready descriptors, 0 on timeout, -1 on error.
- poll() examines a set of file descriptors to see if some of them are ready for I/O or if certain events have occurred on them.
- The fds argument is a pointer to an array of pollfd structures.
- The nfds argument specifies the size of the fds array.
- Each element is a pollfd structure that specifies the conditions to be tested for a given descriptor, fd.

```
struct pollfd {
        int fd; /* descriptor to check */
        short events; /* events of interest on fd */
        short revents; /* events that occurred on fd */
};
```

- **fd**: File descriptor to poll.
- **events**: Events to poll for.
- **revents**: Events which may occur or have occurred.
- The event bitmasks in **events** and **revents** have the following bits:
  - **POLLERR** : An exceptional condition has occurred on the device or socket. This flag is output only, and ignored if present in the input events bitmask.
  - **POLLHUP**: The device or socket has been disconnected. This flag is output only, and ignored if present in the input events bitmask. Note that POLLHUP and POLLOUT are mutually exclusive and should never be present in the revents bitmask at the same time.
  - **POLLIN** : Data other than high priority data may be read without blocking.
  - **POLLNVAL** : The file descriptor is not open. This flag is output only, and ignored if present in the input events bitmask.
  - **POLLOUT** : Normal data may be written without blocking. This is equivalent to POLLWRNORM.
  - **POLLPRI** : High priority data may be read without blocking.
  - **POLLRDBAND** : Priority data may be read without blocking.
  - **POLLRDNORM** : Normal data may be read without blocking.
  - **POLLWRBAND** : Priority data may be written without blocking.
  - **POLLWRNORM** : Normal data may be written without blocking.

POLL() FUNCTION…

- With regard to TCP and UDP sockets, the following conditions cause poll to return the specified **revent**.
  - All regular TCP data and all UDP data is considered normal.
  - TCP's out-of-band data is considered priority band.
  - When the read half of a TCP connection is closed (e.g., a FIN is received), this is also considered normal data and a subsequent read operation will return 0.
  - The presence of an error for a TCP connection can be considered either normal data or an error (POLLERR). In either case, a subsequent read will return –1 with errno set to the appropriate value. This handles conditions such as the receipt of an RST or a timeout.
  - The availability of a new connection on a listening socket can be considered either normal data or priority data. Most implementations consider this normal data.
  - The completion of a nonblocking connect is considered to make a socket writable.

- If timeout is greater than zero, it specifies a maximum interval (in milliseconds) to wait for any file descriptor to become ready.
- If timeout is zero, then poll() will return without blocking.
- If the value of timeout is -1, the poll blocks indefinitely.
- RETURN VALUES
  - poll() returns the number of descriptors that are ready for I/O, or -1 if an error occurred.
  - If the time limit expires, poll() returns 0.
  - If poll() returns with an error, including one due to an interrupted call, the fds array will be unmodified and the global variable errno will be set to indicate the error.
  - **poll() will fail if:**

[**EAGAIN**] :Allocation of internal data structures fails. A subsequent request may succeed.

[**EFAULT**] : fds points outside the process's allocated address space.

[**EINTR**] : A signal is delivered before the time limit expires and before any of the selected events occurs.

[**EINVAL**]:The nfds argument is greater than OPEN_MAX or the timeout argument is less than -1.

SOCKET OPTION

SOCKET OPTIONS

There are 3 ways to get and set options affecting sockets -

- the *getsockopt* and *setsockopt* functions.
- the *fcntl* function
- the *ioctl* function

GETSOCKOPT() AND SETSOCKOPT()

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optname, void * optval, socklen_t * optlen);

int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);

Both functions return 0 if OK else -1 on error.
```

- The **sockfd** must refer to an open socket descriptor.
- The **level** indicates whether the socket option is **general** or **protocol-specific** socket.
- The **optval** is a pointer to a variable from which the new value of the option is fetched by **setsockopt**, or into which the current value of the option is stored by **getsockopt**.
- The size of this variable is specified by the final argument, as a value for **setsockopt** and as a value-result for **getsockopt**.

GETSOCKOPT() AND SETSOCKOPT()…

- There are two basic types of options that can be queried by **getsockopt** or set by **setsockopt**: binary options that enable or disable a certain feature (flags), and options that fetch and return specific values that we can either set or examine (values).
- When calling **getsockopt** for the flag options, *** optval** is an integer.
- The value returned in ***optval** is **zero** if the option is disabled, or nonzero if the option is enabled.
- Similarly, **setsockopt** requires a **nonzero *optval** to turn the option on, and a **zero** value to turn the option off.
- For non-flag options, the option is used to pass a value of the specified datatype between the user process and the system.

SOCKET OPTIONS

Two basic type of options -

- Flags - binary options that enable or disable a feature.
- Values - options that fetch and return specific values.

Not supported by all implementations.

Socket option fall into 4 main categories -

- Generic socket options
  - SO_RCVBUF, SO_SNDBUF, SO_BROADCAST, etc.
- IPv4
  - IP_TOS, IP_MULTICAST_IF, etc.
- IPv6
  - IPv6_HOPLIMIT, IPv6_NEXTHOP, etc.
- TCP
  - TCP_MAXSEG, TCP_KEEPALIVE, etc.

SOCKET STATES

- Options have to be set or fetched depending on the state of a socket.
- Some socket options are inherited from a listening socket to the connected sockets on the server side.

  - E.g. SO_RCVBUF and SO_SNDBUF

  These options have to be set on the socket before calling *listen()* on the server side and before calling *connect()* on the client side.

GENERIC SOCKET OPTIONS

SO_BROADCAST

- Enables or disables the ability of a process to send broadcast messages.
- It is supported only for datagram sockets.
- Its default value is ***off***.

SO_ERROR

- *Pending Error* - When an error occurs on a socket, the kernel sets the ***so_error*** variable.
- The process can be notified of the error in two ways -
  - If the process is blocked in ***select*** for either read or write, it returns with either or both conditions set.
  - If the process is using signal driven I/O, the ***SIGIO*** signal is generated for the process.

GENERIC SOCKET OPTIONS CONTD…

SO_KEEPALIVE

- Purpose of this option is to detect if the peer host crashes. The **SO_KEEPALIVE** option will detect half-open connections and terminate them.
- If this option is set and no data has been exchanged for 2 hours, then TCP sends **keepalive** probe to the peer.
  - Peer responds with **ACK**. Another probe will be sent only after 2 hours of inactivity.
  - Peer responds with **RST** (has crashed and rebooted). Error is set to **ECONNRESET** and the socket is closed.
  - No response. 8 more probes are sent after which the socket's pending error is set to either **ETIMEDOUT** or **EHOSTUNREACH** and the socket is closed.

GENERIC SOCKET OPTIONS CONTD…

Receive Low Water Mark -

- Amount of **data** that must be in the socket receive buffer for a socket to become ready for *read.*

Send Low Water Mark -

- Amount of **space** that must be available in the socket send buffer for a socket to become ready for *write*.

SO_RCVLOWAT and **SO_SNDLOWAT**

- These options specify the receive low water mark and send low water mark for TCP and UDP sockets.

GENERIC SOCKET OPTIONS CONTD…

SO_RCVTIMEO and **SO_SNDTIMEO**

- These options place a timeout on socket receives and sends.
- The timeout value is specified in a ***timeval*** structure.

```
struct timeval {
 long tv_sec ;
 long tv_usec ;
}
```

- To disable a timeout, the values in the ***timeval*** structure are set to 0.

GENERIC SOCKET OPTIONS CONTD…

SO_REUSEADDR

- It allows a listening server to restart and bind its well known port even if previously established connections exist.
- It allows multiple instances of the same server to be started on the same port, as long as each instance binds a different local IP address.
- It allows a single process to bind the same port to multiple sockets, as long as each bind specifies a different local IP address.
- It allows completely duplicate bindings only for UDP sockets (broadcasting and multicasting).

GENERIC SOCKET OPTIONS CONTD…

SO_LINGER

- Specifies how *close* operates for a connection-oriented protocol
- The following structure is used:

```
struct linger {
     int l_onoff;
     int l_linger;
}
// l_onoff - 0=off; nonzero=on
// l_linger specifies seconds
```

Three scenarios:

1. If ***l_onoff*** is 0, ***close*** returns immediately. If there is any data still remaining in the socket send buffer, the system will try to deliver the data to the peer. The value of ***l_linger*** is ignored.
2. If ***l_onoff*** is **nonzero** and ***linger*** is 0, TCP aborts the connection when *close* is called. TCP discards data in the send buffer and sends **RST** to the peer.

GENERIC SOCKET OPTIONS CONTD...

3. **SO_LINGER (cont)**

If ***l_onoff*** is nonzero and ***linger*** is nonzero, the kernel will linger when *close* is called.

- If there is any data in the send buffer, the process is put to sleep until either:
  - the data is sent and acknowledged
  - Or
  - the linger time expires (for a nonblocking socket the process will not wait for *close* to complete)
- When using this feature, the return value of *close* must be checked. If the linger time expires before the remaining data is send and acknowledged, close returns **EWOULDBLOCK** and any remaining data in the buffer is ignored.

GENERIC SOCKET OPTIONS CONTD…

SO_SNDBUF/SO_RCVBUF

Level: SOL_SOCKET

Get/Set supported

Non-flag option

Datatype of optval: int

Description: Send buffer size/Receive buffer size

GENERIC SOCKET OPTIONS CONTD…

TCP_NODELAY

Level: IPPROTO_TCP

Get/Set supported

Flag option i.e. enable or disable Nagle algorithm

Description: Disable/Enable Nagle algorithm

FCNTL() FUNCTION

- **Fcntl()** stands for "file control" and this function performs various descriptor control operations.
- The fcntl function provides the following features related to network programming.
  - **Non-blocking I/O** – We can set the **O_NONBLOCK** file status flag using the **F_SETFL** command to set a socket as non-blocking.
  - **Signal-driven I/O** – We can set the **O_ASYNC** file status flag using the **F_SETFL** command, which causes the SIGIO signal to be generated when the status of a socket changes.
  - The **F_SETOWN** command lets us set the socket owner (the process ID or process group ID) to receive the **SIGIO** and **SIGURG** signals. The former signal is generated when the signal-driven I/O is enabled for a socket and the latter signal is generated when new out-of-band data arrives for a socket. The **F_GETOWN** command returns the current owner of the

socket.

FCNTL()…

```
int fcntl(int fd, int cmd, long arg);
```

- Each descriptor has a set of file flags that is fetched with the **F_GETFL** command and set with the **F_SETFL** command. The two flags that affect a socket are

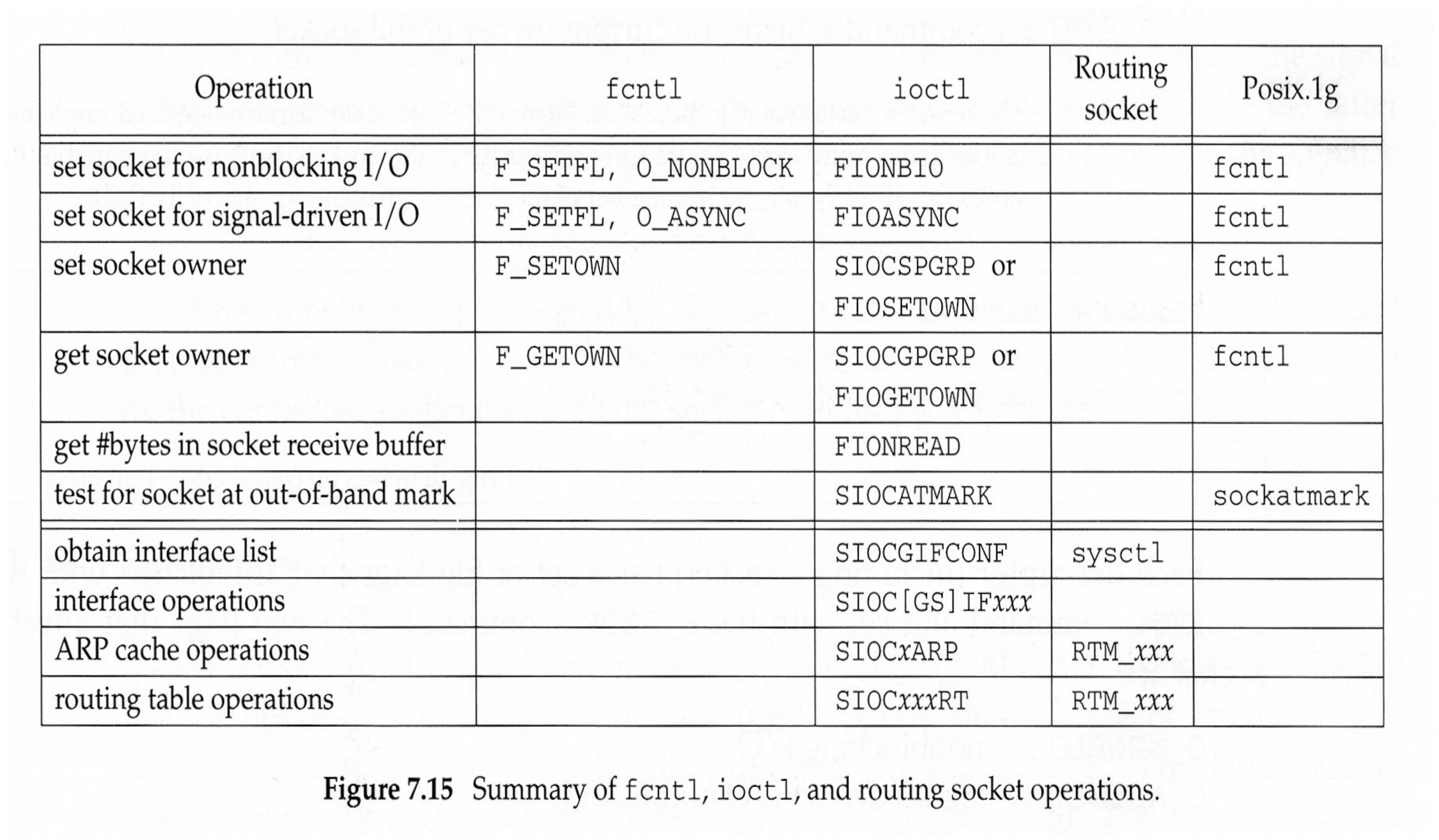
O_NONBLOCK - non-blocking I/O

O_ASYNC-signal-driven I/O

Miscellaneous file control operations

- Non-blocking I/O (O_NONBLOCK, F_SETFL)
- Signal-driven I/O (O_ASYNC, F_SETFL)
- Set socket owner (F_SETOWN)

FCNTL AND IOCTL



Operation	fcntl	ioctl	Routing socket	Posix.1g
set socket for nonblocking I/O	F_SETFL, O_NONBLOCK	FIONBIO		fcntl
set socket for signal-driven I/O	F_SETFL, O_ASYNC	FIOASYNC		fcntl
set socket owner	F_SETOWN	SIOCSPGRP or FIOSETOWN		fcntl
get socket owner	F_GETOWN	SIOCGPGRP or FIOGETOWN		fcntl
get #bytes in socket receive buffer		FIONREAD		
test for socket at out-of-band mark		SIOCATMARK		sockatmark
obtain interface list		SIOCGIFCONF	sysctl	
interface operations		SIOC[GS]IF*xxx*		
ARP cache operations		SIOC*x*ARP	RTM_*xxx*	
routing table operations		SIOC*xxx*RT	RTM_*xxx*	

Figure 7.15 Summary of fcntl, ioctl, and routing socket operations.

FCNTL() FUNCTION

- *fcntl* provides the following features related to network programming
  - **Nonblocking I/O** (be aware of error-handling in the following code)

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

  - **Signal driven I/O**

```
int flags=fcntl(fd, F_GETFL, 0);
flags |= O_ASYNC;
fcntl(fd, F_SETFL, flags);
```

  - Set socket owner to receive SIGIO signals

```
fcntl(fd, F_SETOWN, getpid());
```

FCNTL() FUNCTION

- The signals **SIGIO** and **SIGURG** are generated for a socket only if the socket has been assigned an owner with the **F_SETOWN** command. The integer arg value for the **F_SETOWN** command can be either positive integer, specifying the process ID to receive the signal, or a negative integer whose absolute value is the process group ID to receive the signal.
- The **F_GETOWN** command returns the socket owner as the return value from the fcntl function, either the process ID (a positive return value) or the process group ID (a negative value other than -1).
- The difference between specifying a process or a process group to receive the signal is that the former causes only a single process to receive the signal, while the latter causes all processes in the process group to receive the signal.

IOCTL OPERATIONS

- The common use of **ioctl** by network programs (typically servers) is to obtain information on all the host's interfaces when the program starts: the interface addresses, whether interface supports broadcasting, whether the interface supports multicasting, and so on.

Ioctl() Function

- This function affects an open file referenced by the **fd** argument.

```
#include <unistd.h>

 int ioctl(int fd, int request, …/* void *arg */);

 Returns: 0 if OK, -1 on error
```

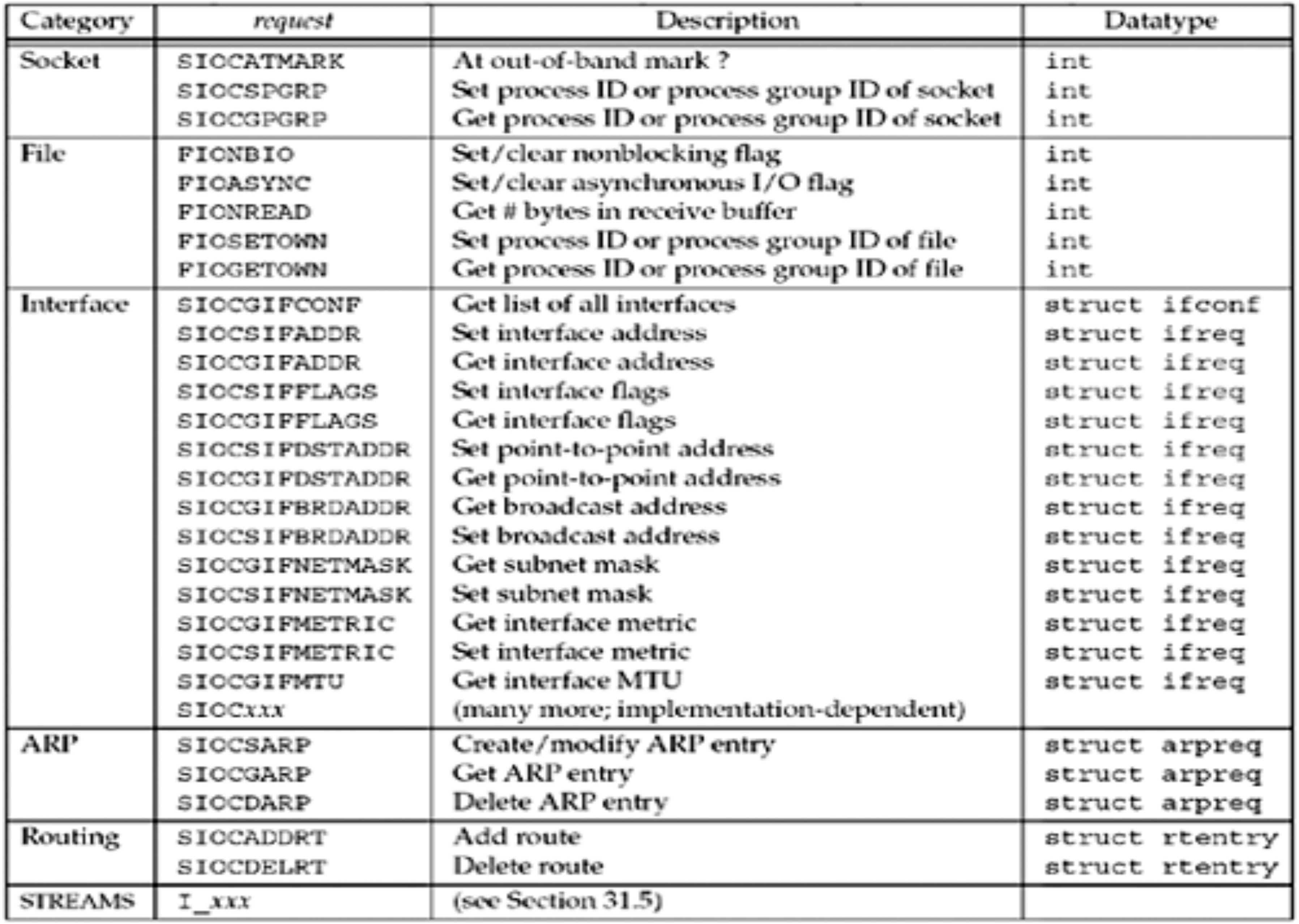
- The third argument is always a pointer, but the type of pointer depends on the request.

IOCTL() FUNCTION

We can divide the requests related to networking into six categories.

1. Socket operations
2. File operations
3. Interface operations
4. ARP cache operations
5. Routing table operations
6. STREAMS system

Note that not only do some of the **ioctl** operations overlap some of the **fcntl** operations (e.g. setting a socket to non-blocking), but there are also some operations that can be specified more than one way using **ioctl** (e.g., setting the process group ownership of a socket).



Category	*request*	Description	Datatype
Socket	`SIOCATMARK`	At out-of-band mark ?	`int`
	`SIOCSPGRP`	Set process ID or process group ID of socket	`int`
	`SIOCGPGRP`	Get process ID or process group ID of socket	`int`
File	`FIONBIO`	Set/clear nonblocking flag	`int`
	`FIOASYNC`	Set/clear asynchronous I/O flag	`int`
	`FIONREAD`	Get # bytes in receive buffer	`int`
	`FIOSETOWN`	Set process ID or process group ID of file	`int`
	`FIOGETOWN`	Get process ID or process group ID of file	`int`
Interface	`SIOCGIFCONF`	Get list of all interfaces	`struct ifconf`
	`SIOCSIFADDR`	Set interface address	`struct ifreq`
	`SIOCGIFADDR`	Get interface address	`struct ifreq`
	`SIOCSIFFLAGS`	Set interface flags	`struct ifreq`
	`SIOCGIFFLAGS`	Get interface flags	`struct ifreq`
	`SIOCSIFDSTADDR`	Set point-to-point address	`struct ifreq`
	`SIOCGIFDSTADDR`	Get point-to-point address	`struct ifreq`
	`SIOCGIFBRDADDR`	Get broadcast address	`struct ifreq`
	`SIOCSIFBRDADDR`	Set broadcast address	`struct ifreq`
	`SIOCGIFNETMASK`	Get subnet mask	`struct ifreq`
	`SIOCSIFNETMASK`	Set subnet mask	`struct ifreq`
	`SIOCGIFMETRIC`	Get interface metric	`struct ifreq`
	`SIOCSIFMETRIC`	Set interface metric	`struct ifreq`
	`SIOCGIFMTU`	Get interface MTU	`struct ifreq`
	`SIOC`*xxx*	(many more; implementation-dependent)	
ARP	`SIOCSARP`	Create/modify ARP entry	`struct arpreq`
	`SIOCGARP`	Get ARP entry	`struct arpreq`
	`SIOCDARP`	Delete ARP entry	`struct arpreq`
Routing	`SIOCADDRT`	Add route	`struct rtentry`
	`SIOCDELRT`	Delete route	`struct rtentry`
STREAMS	`I_`*xxx*	(see Section 31.5)	

SOCKET OPERATIONS

- Three **ioctl** requests are explicitly used for sockets. All three require that the third argument to **ioctl** be a pointer to an integer.

- **SIOCATMARK**: Return through the integer pointed to by the third argument a non-zero value if the socket’s read pointer is currently at the out-of-band mark, or a zero value if the read pointer is not at the out-of-band mark.

- **SIOCGPGRP**: Return through the integer pointed to by the third argument either the process ID or the process group ID that is set to receive **SIGIO** or SIGURG signal for this socket. This request is identical to an fcntl of **F_GETOWN**, note that POSIX standardizes the fcntl.

- **SIOCSPGRP:** Set either the process ID or process group ID to receive the **SIGIO** or **SIGURG** signal for this socket from the integer pointed to by the third argument. This request is identical to an fnctl of **F_SETOWN**, note that POSIX standardizes the **fcntl**.

FILE OPERATIONS

The next group of requests begin with FIO and may apply to certain types of files, in addition to sockets.

FIONBIO	The nonblocking flag for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This request has the same effect as the **O_NONBLOCK** file status flag, which can be set and cleared with the **F_SETFL** command to the fcntl function.
FIOASYNC	The flag that governs the receipt of asynchronous I/O signals (SIGIO) for the socket is cleared or turned on, depending on whether the third argument to ioctl points to a zero or nonzero value, respectively. This flag has the same effect as the O_ASYNC file status flag, which can be set and cleared with the F_SETFL command to the fcntl function.
FIONREAD	Return in the integer pointed to by the third argument to ioctl the number of bytes currently in the socket receive buffer. This feature also works for files, pipes, and terminals.
FIOSETOWN	Equivalent to SIOCSPGRP for a socket.
FIOGETOWN	Equivalent to SIOCGPGRP for a socket.

INTERFACE OPERATIONS

The **SIOCGIFCONF** request returns the name and a socket address structure for each interface that is configured. Many of requests use a socket address structure to specify or return an IP address or address mask with the application.

SIOCGIFADDR	Return the unicast address in the ifr_addr member.
SIOCSIFADDR	Set the interface address from the ifr_addr member. The initialization function for the interface is also called.
SIOCGIFFLAGS	Return the interface flags in the ifr_flags member. The names of the various flags are IFF_*xxx* and are defined by including the <net/if.h> header.
SIOCSIFFLAGS	Set the interface flags from the ifr_flags member.
SIOCGIFDSTADDR	Return the point-to-point address in the ifr_dstaddr member.
SIOCSIFDSTADDR	Set the point-to-point address from the ifr_dstaddr member.
SIOCGIFBRDADDR	Return the broadcast address in the ifr_broadaddr member. The application must first fetch the interface flags and then issue the correct request: SIOCGIFBRDADDR for a broadcast interface or SIOCGIFDSTADDR for a point-to-point interface.
SIOCSIFBRDADDR	Set the broadcast address from the ifr_broadaddr member.
SIOCGIFNETMASK	Return the subnet mask in the ifr_addr member.
SIOCSIFNETMASK	Set the subnet mask from the ifr_addr member.
SIOCGIFMETRIC	Return the interface metric in the ifr_metric member. The interface metric is maintained by the kernel for each interface but is used by the routing daemon routed. The interface metric is added to the hop count (to make an interface less favorable).
SIOCSIFMETRIC	Set the interface routing metric from the ifr_metric member.

ARP CACHE OPERATIONS

- On some systems, the ARP cache is also manipulated with the **ioctl** function. Systems that use routing sockets usually use routing sockets instead of **ioctl** to access the ARP cache. These requests use an **arpreq** structure, shown below and defined by including the <net/if_arp.h> header.

```
<net/if_arp.h>

struct arpreq {

struct sockaddr arp_pa; /* protocol address */

struct sockaddr arp_ha; /* hardware address */

int arp_flags; /* flags */

};

#define ATF_INUSE 0x01 /* entry in use */

#define ATF_COM 0x02 /* completed entry (hardware addr valid) */

#define ATF_PERM 0x04 /* permanent entry */

#define ATF_PUBL 0x08 /* published entry (respond for other host) */
```

ARP CACHE OPERATIONS

The third argument to ioctl must point to one of these structures. The following three *requests* are supported

SIOCSARP	Add a new entry to the ARP cache or modify an existing entry. arp_pa is an Internet socket address structure containing the IP address, and arp_ha is a generic socket address structure with sa_family set to AF_UNSPEC and sa_data containing the hardware address (e.g., the 6-byte Ethernet address). The two flags, ATF_PERM and ATF_PUBL, can be specified by the application. The other two flags, ATF_INUSE and ATF_COM, are set by the kernel.
SIOCDARP	Delete an entry from the ARP cache. The caller specifies the Internet address for the entry to be deleted.
SIOCGARP	Get an entry from the ARP cache. The caller specifies the Internet address, and the corresponding Ethernet address is returned along with the flags.

ROUTING TABLE OPERATIONS

- On some systems, two **ioctl** requests are provided to operate on the routing table. These two requests require that the third argument to **ioctl** be a pointer to an **rtentry** structure, which is defined by including the <net/route.h> header. These requests are normally issued by the route program. Only the superuser can issue these requests. On systems with routing sockets, these requests use routing sockets instead of ioctl.

SIOCADDRT	Add an entry to the routing table.
SIOCDELRT	Delete an entry from the routing table.

- There is no way with **ioctl** to list all the entries in the routing table. This operation is usually performed by the **netstat** program when invoked with the -r flag. This program obtains the routing table by reading the kernel's memory (/dev/kmem).

IOCTL SUMMARY

The ioctl commands that are used in network programs can be divided into six categories:

1. Socket operations (Are we at the out-of-band mark?)
2. File operations (set or clear the nonblocking flag)
3. Interface operations (return interface list, obtain broadcast address)
4. ARP table operations (create, modify, get, delete)
5. Routing table operations (add or delete)
6. STREAMS system

DAEMON PROCESS, SYSLOGD DAEMON, SYSLOG FUNCTION, IOCTL OPERATION, IOCTL FUNCTION

DAEMON PROCESS

- A daemon is a process that runs in the background and is not associated with a controlling terminal. Unix systems typically have many processes that are daemons, running in the background, performing different administrative tasks.

- The lack of a controlling terminal is typically a side effect of being started by a system initialization script. But if a daemon is started by a user typing to a shell prompt, it is important for the daemon to disassociate itself from the controlling terminal to avoid any unwanted interaction with job control, terminal session management, or simply to avoid unexpected output to the terminal from the daemon as it runs in the background.

HOW TO START DAEMON PROCESS?

- During system startup, many daemons are started by the system initialization scripts. Daemons started by these scripts begin with superuser privileges.
- Many network servers are started by the **inetd superserver**. The **inetd** itself is started from one of the scripts in Step 1. The inetd listens for network requests, and when a request arrives, it invokes the actual server.
- The execution of programs on a regular basis is performed by the **cron** daemon, and programs that it invokes run as daemons. The cron daemon itself is started in Step 1 during system startup.
- The execution of a program at one time in the future is specified by the "**at**" command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.
- Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason.

- Since a daemon does not have a controlling terminal, it needs some way to output messages when something happens, either normal informational messages or emergency messages that need to be handled by an administrator.

HOW TO CREATE A DAEMON IN UNIX (HOW TO DAEMONIZE A PROCESS)

1. fork

We first call fork and then the parent terminates, and the child continues. If the process was started as a shell command in the foreground, when the parent terminates the shell think the command is done. This automatically runs the child process in the background.

2. setsid

setsid is a POSIX function that creates a new session. The process becomes the session leader of the new session, becomes the process group leader of a new process group, and has no controlling terminal.

3. Ignore SIGHUP and fork again

We ignore SIGUP and call fork again. When this function returns, the parent is really the first child and it terminates, leaving the second child running. The purpose of this second fork is to guarantee that the daemon cannot automatically acquire a controlling terminal should it open a terminal device in the future. We must ignore SIGHUP because when the session leader terminates (the first child), all processes in the session (our second child) receive the SIGHUP signal.

4. Change working directory

We change the working directory to the root directory. The file system cannot be un-mounted if working directory is not changed.

HOW TO CREATE A DAEMON IN UNIX (HOW TO DAEMONIZE A PROCESS) ...

5. Close any open descriptors

We open any open descriptors that are inherited from the process that executed the daemon (normally a shell).

6. Redirect stdin, stdout, and stderr to /dev/null

We open /dev/null for standard input, standard output, and standard error. This guarantees that these common descriptors are open, and a read from any of these descriptors returns 0 (EOF), and the kernel just discards anything written to them.

7. Use syslogd for errors

The syslogd daemon is used to log errors.

SYSLOGD DAEMON

- Unix systems normally start a daemon named **syslogd** from one of the system initialization scripts, and it runs as long as the system is up. The **syslogd** perform the following actions on startup.
  - The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log messages that the daemon can receive.
  - A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).
  - A UDP socket is created and bound to port 514 (the syslog service).
  - The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device.
- The **syslogd** daemon runs in an infinite loop that calls **select**, waiting for any one of its three descriptors (last three of above bullets) to be readable; it reads the log message and does what the configuration file says to do with that message. If the daemon receives the SIGUP signal, it reads its configuration file

SYSLOG FUNTION

- Since a daemon does not have a controlling terminal, it cannot just **fprintf** to **stderr**. The common technique for logging messages from a daemon is to call the syslog function.

```
 #include <syslog.h>

  void syslog(int priority, const char * message,…);
```

- The priority argument is a combination of a level and a facility. The message is like a format string to printf, with addition of a %m specification, which is replaced with error message corresponding to the current value of **errno**.
- For example, the following call could be issued by a daemon when a call to the rename function unexpectedly fails:

```
 syslog(LOG_INFO| LOG_LOCAL2, "rename (%s, %s): %m", file1,
file2);
```

Priority order (higher to lower)

- LOG_EMERG: A panic condition. This is normally broadcast to all users.
- LOG_ALERT:A condition that should be corrected immediately, such as a corrupted system database.
- LOG_CRIT: Critical conditions, e.g., hard device errors.
- LOG_ERR:Errors.
- LOG_WARNING:Warning messages.
- LOG_NOTICE: Conditions that are not error conditions, but should possibly be handled specially.
- LOG_INFO:Informational messages.
- LOG_DEBUG:Messages that contain information normally of use only when debugging a program.

- **Facility parameter**
- LOG_AUTH : The authorization system: login(1), su(1), getty(8), etc.
- LOG_AUTHPRIV : The same as LOG_AUTH, but logged to a file readable only by selected individuals.
- LOG_CRON : The cron daemon: cron(8).
- LOG_DAEMON : System daemons, such as routed(8), that are not provided for explicitly by other facilities.
- LOG_FTP : The file transfer protocol daemons: ftpd(8), tftpd(8).
- LOG_KERN : Messages generated by the kernel. These cannot be generated by any user processes.
- LOG_LPR : The line printer spooling system: cups-lpd(8), cupsd(8), etc.
- LOG_MAIL : The mail system.
- LOG_NEWS : The network news system.
- LOG_SECURITY : Security subsystems, such as ipfw(4).
- LOG_SYSLOG : Messages generated internally by syslogd(8).
- LOG_USER : Messages generated by random user processes. This is the default facility identifier if none is specified.
- LOG_UUCP : The uucp system.
- LOG_LOCAL0 : Reserved for local use.
- Similarly for LOG_LOCAL1 through LOG_LOCAL7.

- When the application calls syslog the first time, it creates a Unix domain datagram socket and then calls connect to the well-known pathname of the socket created by the syslogd daemon. This socket remains open until the process terminates. Alternatively, the process can call openlog and closelog.

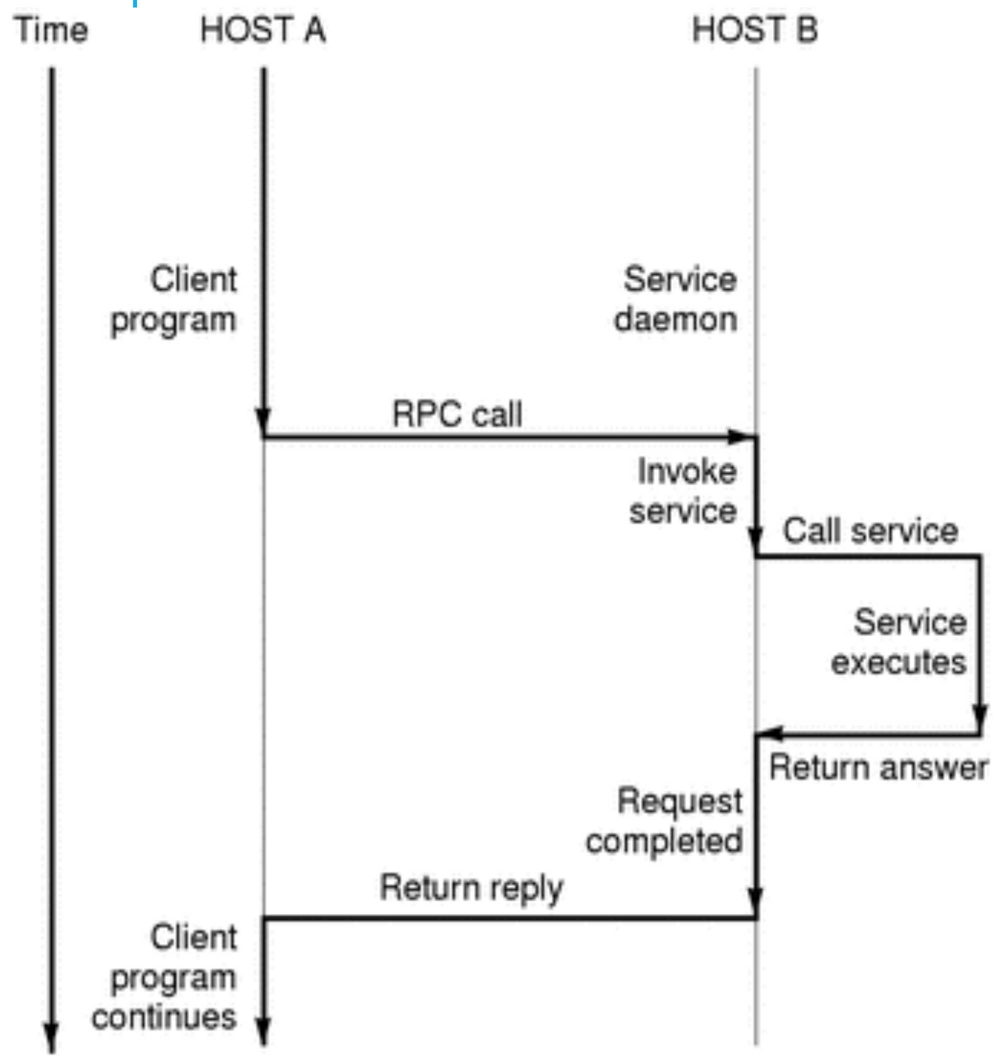
```
#include <syslog.h>

 void openlog(const char *ident, int options, int facility);

void closelog();
```

- The openlog can be called before the first call to syslog and closelog can be called when the application is finished sending log messages.

REMOTE PROCEDURAL CALL

- Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details.
- RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure.
- The two processes may be on the same system, or they may be on different systems with a network connecting them.
- By using RPC, programmers of distributed applications avoid the details of the interface with the network.
- The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.
- RPC makes the client/server model of computing more powerful and easier to program.

REMOTE PROCEDURAL CALL



When making a remote procedure call:
1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The main goal of RPC is to hide the **existence of the network** from a program.
1. The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, a client often does not even know they are using the network.

REMOTE PROCEDURAL CALL

- RPC is especially well suited for client-server (e.g., query-response) interaction in which the flow of control alternates between the caller and callee.
- Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

1. A client invokes a *client stub* procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub *marshalls (arranges)* the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer(**TCP/UDP or UNIX/Local Domain in case of IPC**), which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a *server stub*, which demarshalls the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

CRASHING OF SERVER HOST

1. When the server host crashes, nothing is sent out on the existing network connections from the server. We are assuming that the host crashes and is not shut down by an operator.
2. **The (client) application doesn't know about the status of server until it sends data to the server**. When the (client) application sends data to the server, the transport layer expects acknowledgement from the server.
3. In the absence of acknowledgements, the transport layer retransmits the data segment. After specific number of retransmission, the transport layer gives up and notifies the (client) application. In this case (the server host crashed and there was no response at all to the client's data segment), the error ETIMEDOUT is returned.

   However, if the server host has not crashed but was unreachable on the network, assuming the host was still unreachable, and some intermediate router determined that the server host was unreachable and responded with an ICMP "destination unreachable" message, the error is either EHOSTUNREACH or ENETUNREACH.

   If we want to detect the crashing of the server host even if we are not actively sending it data, we must set SO_KEEPALIVE socket option, and send KEEPALIVE packets periodically to the server host.

WHEN SERVER HOST REBOOTS

1. When the server host reboots after crashing, its TCP loses all information about connections that existed before the crash. Therefore, the server TCP respond to the received data segment from the client with an RST packet.

2. The (client) application gets ECONNRESET error from the transport layer.

PROCESS TABLE

- The **process table** is a data structure maintained by the operating system to facilitate context switching and scheduling, etc.
- Each entry in the table, often called a **context block**, contains information about a process such as process name and state, priority, memory state, resource state, registers, and a semaphore it may be waiting on.
- The exact contents of a context block depend on the operating system. For instance, if the OS supports paging, then the context block contains an entry to the page table.

END OF CHAPTER 2