Network Programming

BESE-VI – Pokhara University

Prepared by:

Assoc. Prof. Madan Kadariya (NCIT)

Contact: madan.kadariya@ncit.edu.np

Chapter 2:
Basics of Unix Network Programming
(12 hrs)

Outline

1. Overview of Unix OS and Network Administration
2. Unix socket introduction
3. Socket Address Structures
    i. Genric socket: struct sockaddr
    ii. IPv4: struct sockaddr_in
    iii. IPv6: struct sockaddr_in6
    iv. Unix Domain Sockets: struct sockaddr_un
    v. Storage: struct sockaddr_storage
4. Comparisions of various socket address structure
5. Value result arguments and system calls
6. Byte ordering and manipulation functions
7. Hostname and Network Name Lookups
4. Basic socket system calls
    i. Elementary TCP Sockets
    ii. Elementary UDP Sockets
5. Unix Domain Socket
    i. Passing file descriptors
6. Signal Handling in Unix
    i. Introduction to signals
    ii. Handling signals in network applications (e.g., SIGINT, SIGIO)
    iii. Signal functions (signal(), sigaction())
7. Daemon Processes
    i. Characteristics of daemon processes
    ii. Converting a program into a daemon
    iii. Daemonizing techniques in Unix

Overview of Unix OS

- Unix is a *powerful*, *multiuser*, *multitasking* operating system originally developed in the 1970s at Bell Labs.

  History and Evolution:
  - Is created by Ken Thompson and Dennis Ritchie at AT&T's Bell Labs.
  - Written originally in assembly, later rewritten in C.
  - Basis for many open-source and commercial systems.

- It has served as the foundation for many modern operating systems such as **Linux**, **macOS**, **Solaris**, and **BSD**.
- Unix operating systems are known for their robust features, including multi-user capabilities, multitasking, a hierarchical file structure, and built-in networking.
- Network administration on Unix involves managing network devices, configuring services, and ensuring network security.

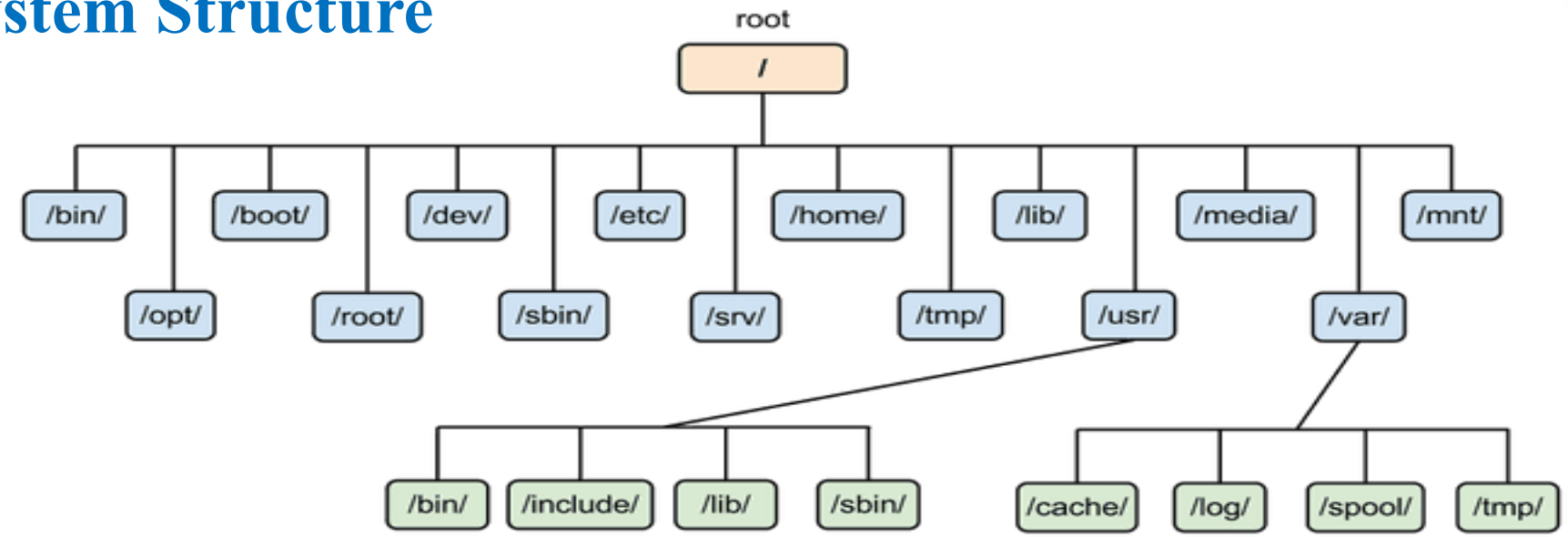
Overview of Unix OS

Key features of Unix OS

1. *Multiuser Support*: Allows multiple users to access the system simultaneously without interfering with each other.
2. *Multitasking*: Ability to run multiple processes at once.
3. *Portability*: Written in C, making it adaptable across hardware platforms.
4. *Security*: File permission and user management systems.
5. *Stability and Efficiency*: Minimal crashes, efficient memory and process handling.
6. *Shell Interface*: CLI and scripting for task automation.

Overview of Unix OS

Unix File System Structure



- **/bin:** short for binaries, this is the directory where many commonly used executable commands reside
- **/dev**: contains device-specific files
- **/etc**: contains system configuration files
- **/home**: contains user directories and files
- **/lib**: contains all library files
- **/mnt**: contains device files related to mounted devices
- **/proc**: contains files related to system processes

- **/root**: the root users' home directory (note this is different than ` / `)
- **/sbin**: binary files of the system reside here.
- **/tmp**: storage for temporary files that are periodically removed from the file system
- **/usr**: It is the directory holding user home directories, its use has changed, and it also contains executable commands
- **/var**: It is a short form for 'variable', a place for files that may often change

Overview of Unix OS

Basic Unix Commands

Command	Description
ls	Lists files in the current directory
cd	Changes the directory
cp, mv, rm	Copy, move, and delete files
chmod	Change file permissions
chown	Change file owner
ps, top	View active processes
kill	Terminate a process
man	Display manual for commands
cron	Schedule jobs/tasks periodically

Overview of Unix Network Administration

- Network administration involves *managing, configuring, monitoring*, and *troubleshooting* network components to ensure smooth communication between systems.
- Unix systems provide many powerful tools for network administration.

Network Configuration in Unix

- **Interfaces**: *eth0*, *wlan0*, *lo*
- **Configuration files:**
  - */etc/network/interfaces* (Debian/Ubuntu legacy)
  - */etc/netplan/*.yaml* (Ubuntu 18.04+)
  - */etc/sysconfig/network-scripts/* (RHEL/CentOS)
- **Tools:**
  - *ifconfig* – Legacy tool to view and configure interfaces.
  - *ip addr, ip link* – Modern tools for networking tasks.
  - *nmcli* – Command-line tool for NetworkManager.

Overview of Unix Network Administration

Essential Networking Tools

Tool/Command	Purpose
ping	Tests connectivity to a remote host
traceroute	Shows the path packets take to a host
netstat / ss	Displays active connections & ports
nslookup, dig	DNS query tools
scp, rsync	Secure file transfer between systems

Common Network Services on Unix

- **SSH (Secure Shell):** Remote login access (ssh user@host)
- **FTP / SFTP**: File transfer protocols
- **HTTP / HTTPS**: Web server services (Apache, Nginx)
- **NFS / SMB**: Network file sharing
- **Mail Services**: SMTP, POP3, IMAP using tools like Postfix or Dovecot

Overview of Unix Network Administration

Firewall and Security Tools

- **iptables**: Powerful firewall for setting packet filtering rules.
- **ufw (Uncomplicated Firewall)**: Simpler frontend for iptables.
- **Example:**
  - *sudo ufw allow 22/tcp :* Allows incoming TCP traffic on port 22 (the default port for SSH). This enables secure remote login to the system via SSH and is essential when configuring a server for remote access.
  - *sudo ufw enable:* Activates the firewall with the configured rules.
  - *sudo ufw status :*Displays the current firewall rules and their status.
- **fail2ban**: Scans logs and bans IPs with suspicious activity (e.g., failed SSH logins).
- **SELinux/AppArmor**: Kernel-level security modules (in RHEL/Ubuntu respectively).
- **Log Monitoring**: Check logs in /var/log/ for intrusion or service errors.

Monitoring and Troubleshooting

- **Log Files**:
  - */var/log/syslog*: System messages
  - */var/log/auth.log*: Authentication attempts
  - */var/log/messages*: General log messages
- **System Monitoring Tools**:
  - *top, htop*: Real-time process monitoring
  - *uptime, vmstat, iostat*: System performance
  - *Network*: iftop, nload, iptraf

Overview of Unix Network Administration

Practical Admin Task

Task	Tool / Command
Set a static IP address	ip, nmcli, or config files
Start and enable SSH server	systemctl enable --now sshd
Add new user with access	useradd, passwd, usermod
Secure the server	Configure ufw, disable root SSH
Set up cron jobs	crontab -e
Monitor service status	systemctl status <service>
View open ports	ss -tuln, netstat -tulnp

Summary of Unix Network Administration

- *Unix* provides a robust platform for secure and efficient computing.
- *System administration* requires understanding the filesystem, permissions, and process control.
- *Network administration* in Unix includes configuring interfaces, running services like SSH or FTP, monitoring traffic, and securing the network using firewalls and logging tools.
- Mastery of the *CLI and shell scripting* is essential for automation and system management.

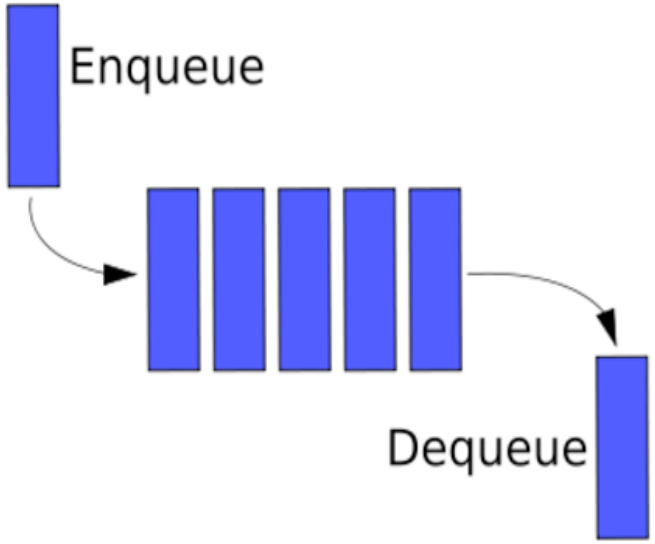
Unix socket introduction

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

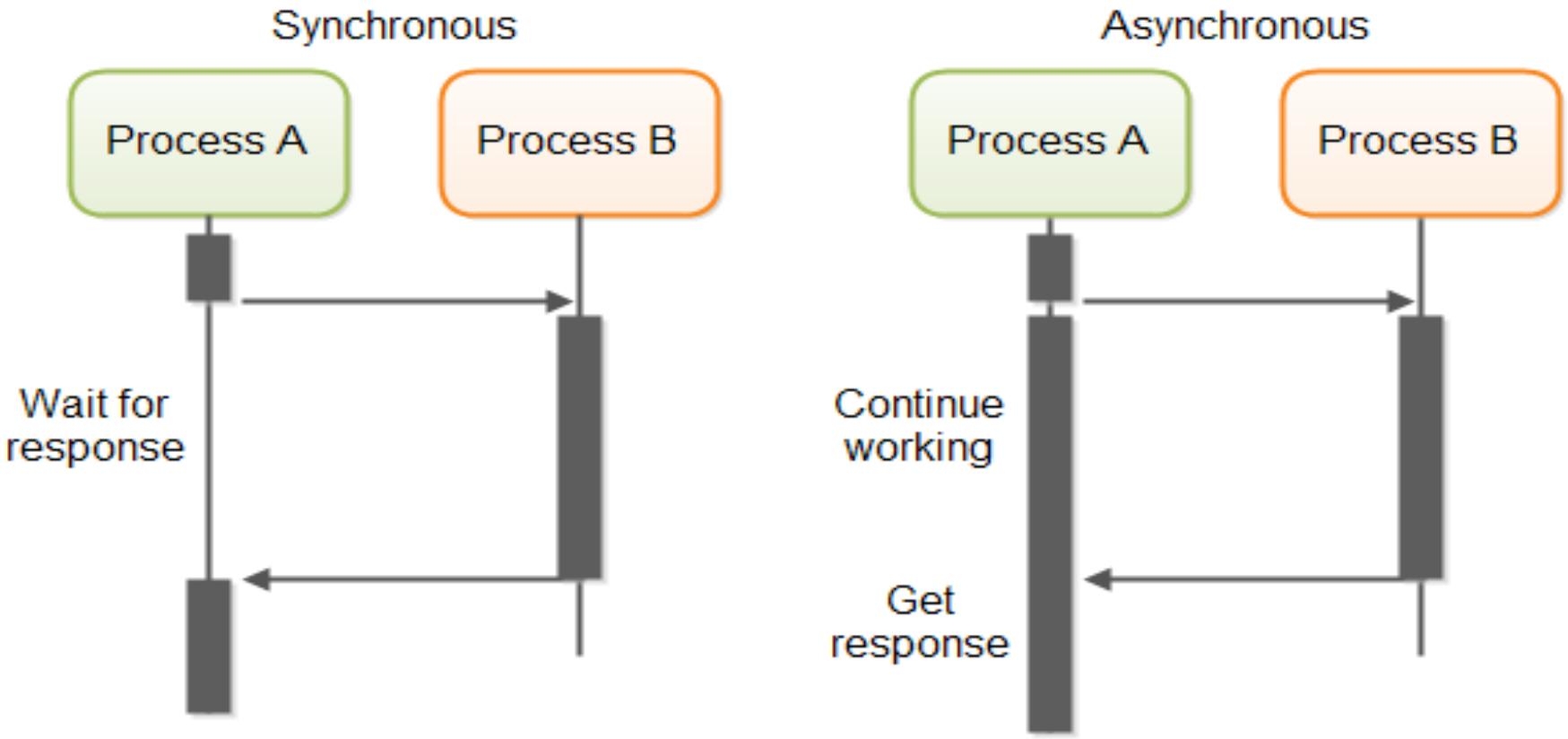
Sending and Receiving

- 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	Asynchronous 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.	➢ 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

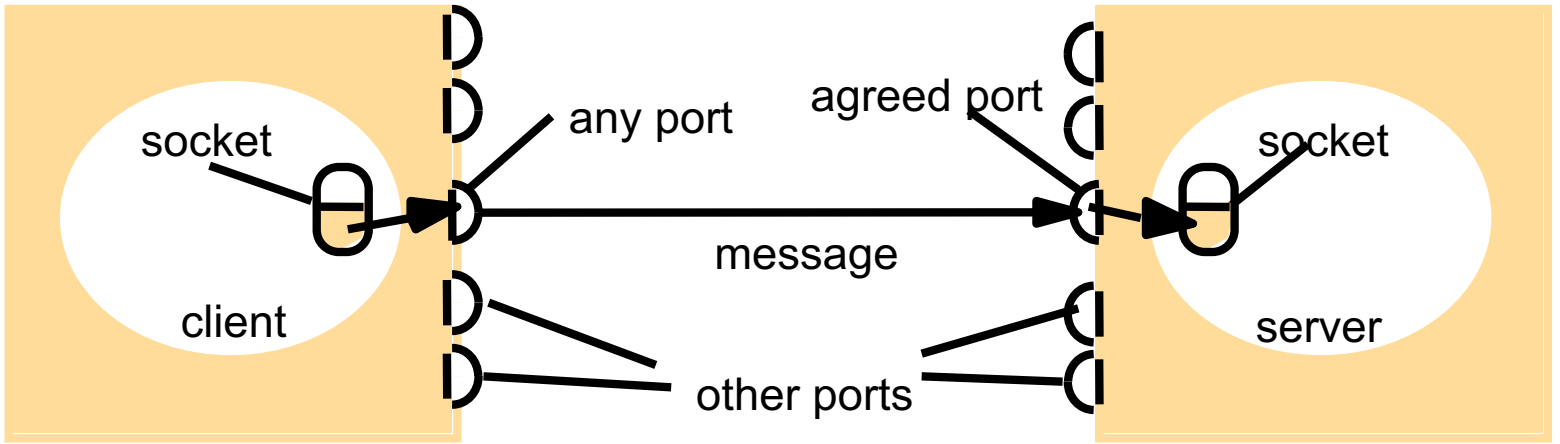
- Data transfer
- Sharing data
- Event notification
- Process control

- 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



- 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.

IPC using sockets

- **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 and Sockets API

Socket

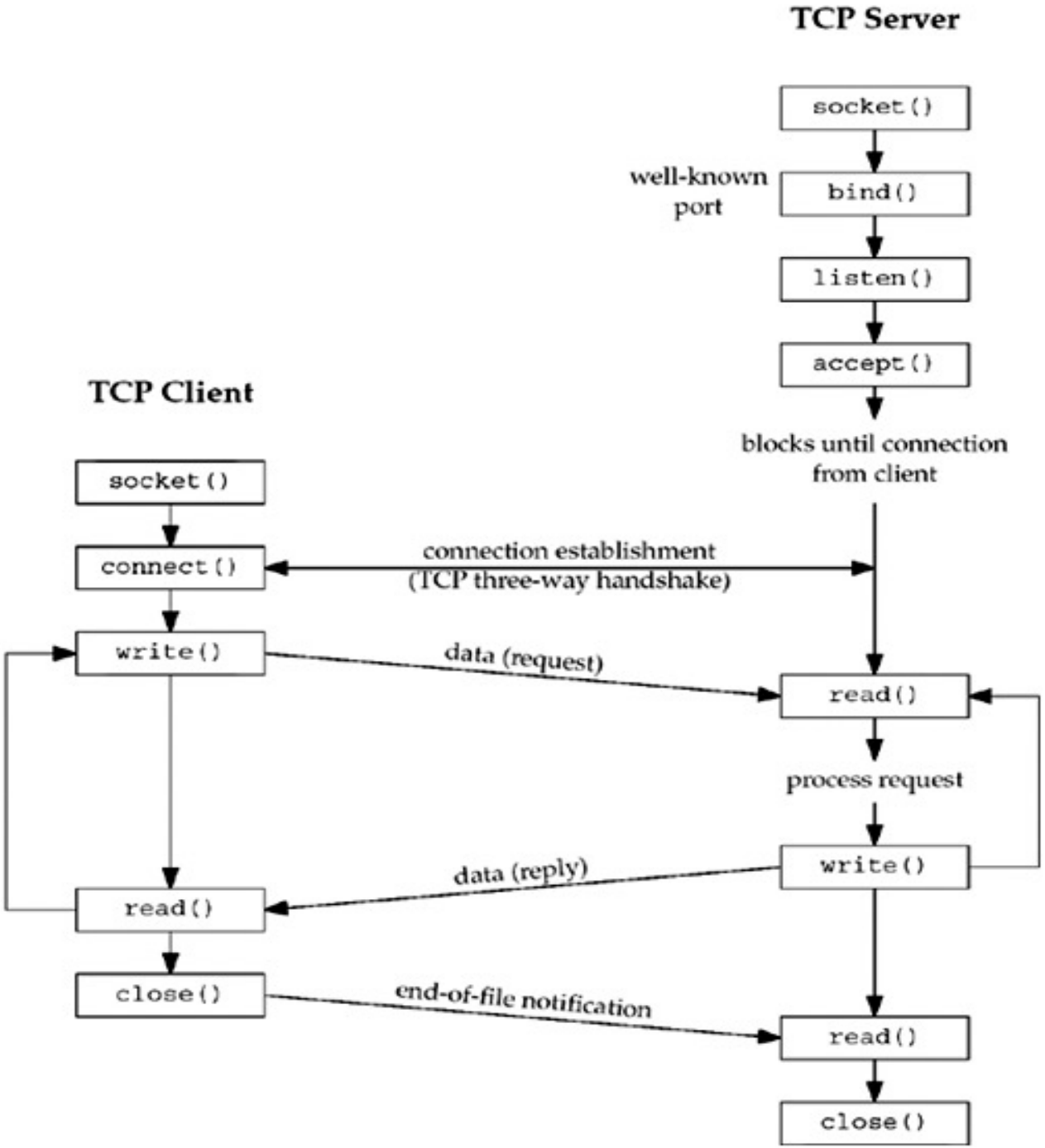
- To the **kernel**, a **socket is an endpoint of communication**.
- To an **application**, a socket is a **file descriptor that lets the application read/write from/to the network.**
  - Remember: All Unix I/O devices, including networks, are modeled as files.

- **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 Structure

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.
- The name of socket address structures begin with ***sockaddr_*** and end with a unique suffix for each protocol suite.

Socket Address Structures

Generic socket address structure (*struct sockaddr*):

- A socket address structures is always passed by reference when passed as an argument to any socket functions
  - For address arguments to connect(), bind(), and accept().
  - A generic socket address structure in the **<sys/socket.h>** header:

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

- From an *application programmer 's perspective*, the only use of these generic socket address structures is to cast pointers to protocol-specific structures.
- From the *kernel's perspective*, kernel must take the caller's pointer, cast it to a *struct sockaddr *,* and then look at the value of *sa_family* to determine the type of the structure.

Socket Address Structures

- **Internet-specific socket address (IPv4 socket address structure)(*struct sockaddr_in*)**
  - Called an "***Internet socket address structure***" and is defined in the *<netinet/in.h>* header.
  - 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; /* 16 bit port num in network byte order */
 struct in_addr sin_addr; /* 32 bit IPv4 address in network byte order */
 unsigned char sin_zero[8]; /* pad to sizeof(struct sockaddr), unused */
};
```

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

- **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.
- 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

Socket Address Structures

IPv6 socket address Structure (*struct sockaddr_in6)*

```
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 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

Socket Address Structure

Storage socket address structure (*struct sockaddr_storage)*

- It is a new generic socket address structure and defined as part of the IPv6 sockets API, to overcome some of the shortcomings of the existing struct *sockaddr*.
- Unlike the struct *sockaddr*, the new struct *sockaddr_storage* is large enough to hold any socket address type supported by the system.
- The *sockaddr_storage* structure is defined by including the *<netinet/in.h>*header:

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

- Different from struct ***sockaddr*** in two ways
  1. If any socket address structures that the system supports have alignment requirements, the *sockaddr_storage* provides the strictest alignment requirement.
  2. The *sockaddr_storage* is large enough to contain any socket address structure that the system supports.
- The *sockaddr_storage* must be cast or copied to the appropriate socket address structure for the address given in *ss_family* to access any other fields.

Socket Address Structure

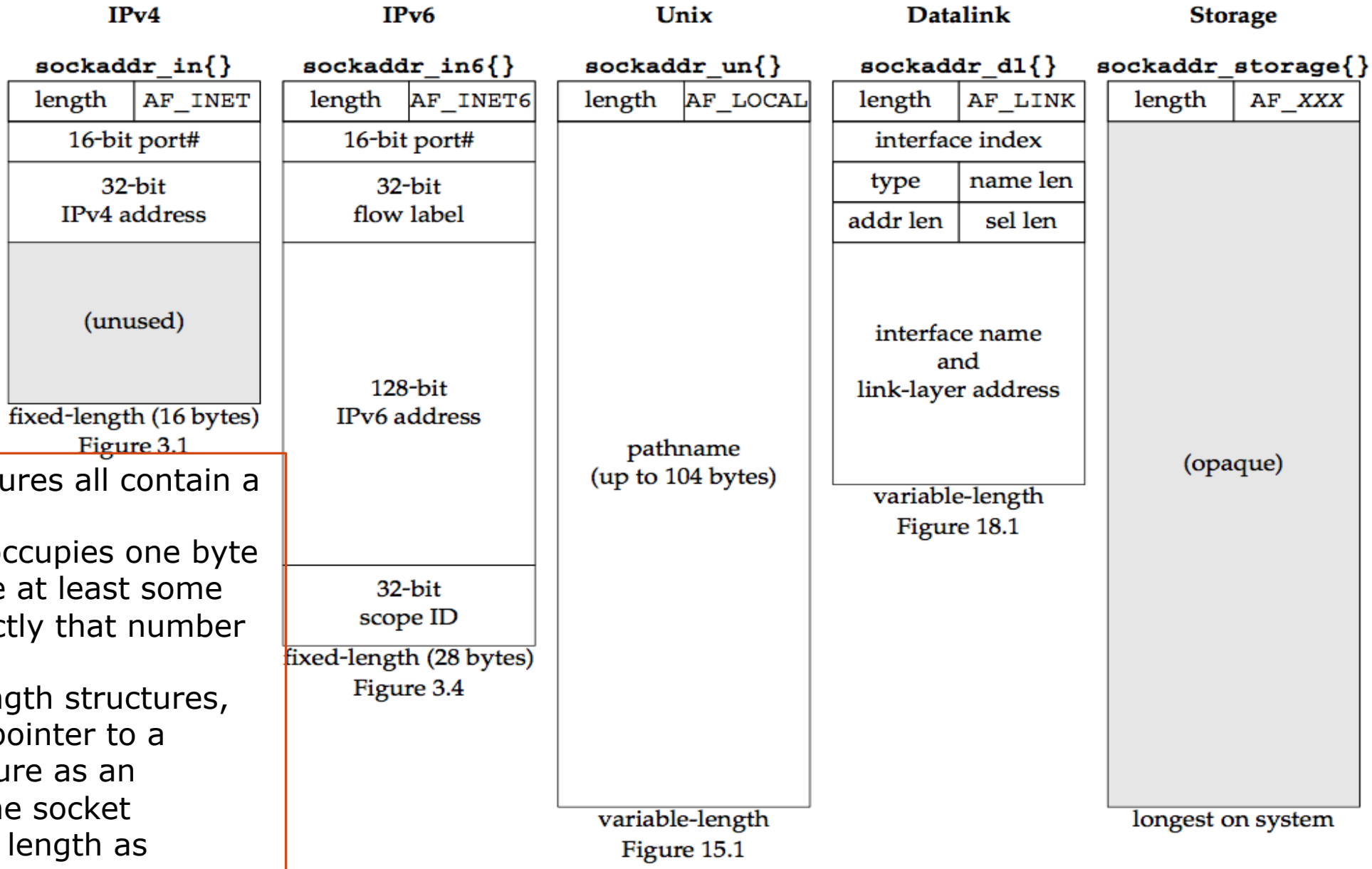
Unix Domain socket address structure (*struct sockaddr_un*)

- The UNIX-domain protocol family is a collection of protocols that provides local (on-machine) interprocess communication through the normal socket mechanisms.
- UNIX-domain addresses are variable-length filesystem pathnames of at most 104 characters.
- The include file *⟨sys/un.h⟩* defines this address:

```
struct sockaddr_un {
        sa_family_t sun_family; /* Always AF_UNIX */
        char sun_path[108]; /* Null-terminated socket pathname */
};
```

- We can't bind a socket to an existing pathname (bind() fails with the error **EADDRINUSE**).
- It is common to bind a socket to an absolute pathname for a fixed location. Relative pathnames are rare, as they require the client to know the server's working directory.
- A socket may be bound to only one pathname; conversely, a pathname can be bound to only one socket.
- We can't use open() to open a socket.
- When the socket is no longer required, its pathname entry can (and generally should) be removed using unlink() (or remove()).

Comparison of various socket address structure



- Socket address structures all contain a one-byte length field
- The family field also occupies one byte
- Any field that must be at least some number of bits is exactly that number of bits.
- To handle variable-length structures, whenever we pass a pointer to a socket address structure as an argument to one of the socket functions, we pass its length as another argument

Comparison of various socket address structure

Structure	Header File	Used For	Address Family	Commonly Used Fields / Purpose
sockaddr	<sys/socket.h>	Generic socket container	AF_UNSPEC	*sa_family, sa_data* ; used for casting to specific types
sockaddr_in	<netinet/in.h>	IPv4 sockets	AF_INET	*sin_family, sin_port, sin_addr*
sockaddr_in6	<netinet/in.h>	IPv6 sockets	AF_INET6	*sin6_family, sin6_port, sin6_flowinfo, sin6_addr, sin6_scope_id*
sockaddr_un	<sys/un.h>	Unix domain sockets	AF_UNIX	*sun_family, sun_path* (up to 104 bytes)
sockaddr_storage	<sys/socket.h>	Generic storage (all types)	Depends	Large, aligned structure to hold any socket type (opaque)

Datatypes required by POSIX

Data Type	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

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

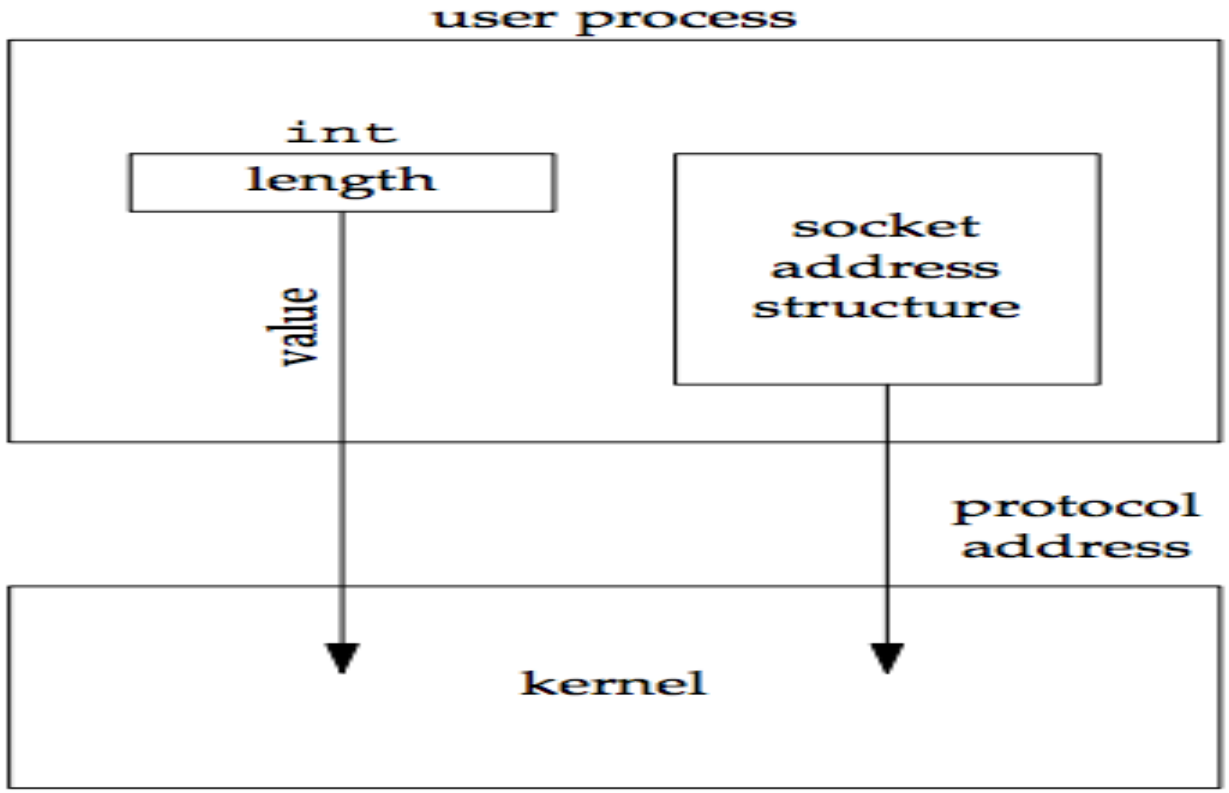
Arumgents to these functions:

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

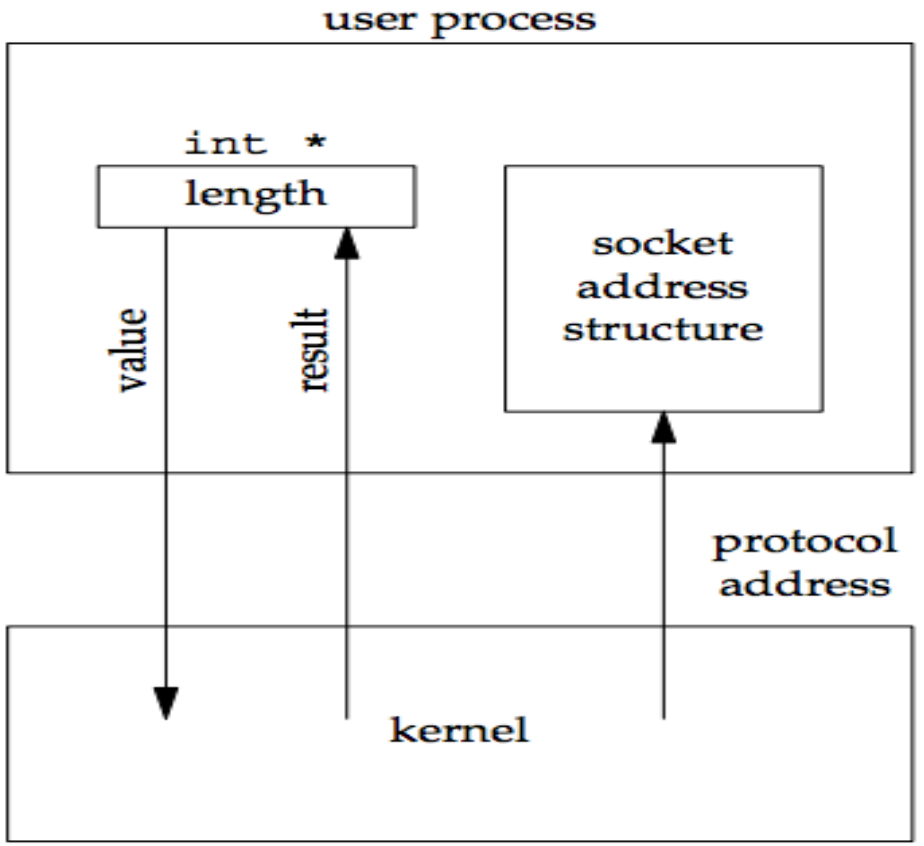
- **Kernel to Process**

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

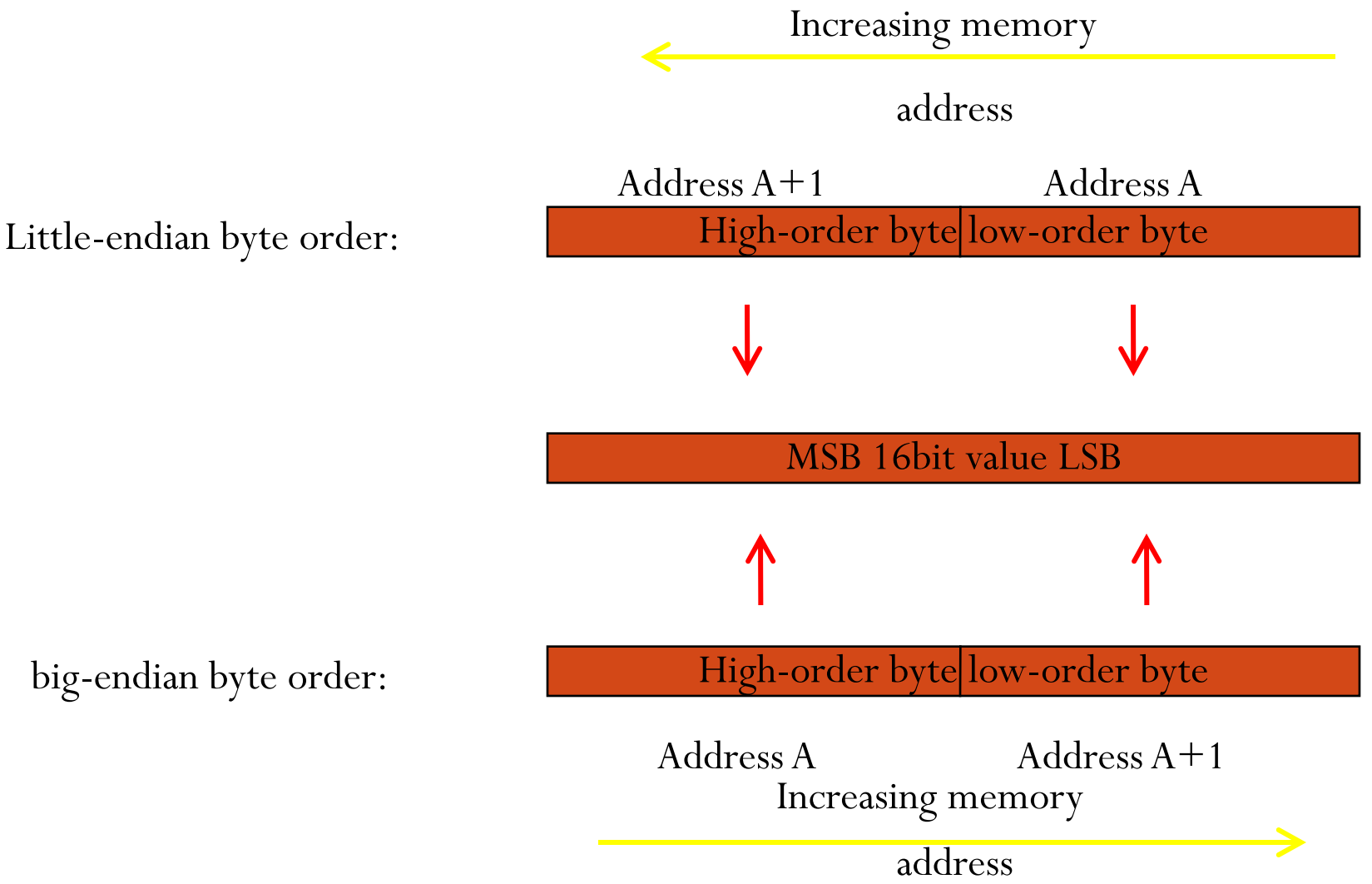
- Value-only: ***bind()***, ***connect()***, ***sendto()*** (from process to kernel)
- Value-Result: ***accept()***, ***recvfrom()***, ***getsockname()***, ***getpeername()*** (from kernel to process, pass a pointer to an integer containing size)
  - *Tells process how much information kernel actually stored*
- As a **value**: it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in
- As a **result**: it tells the process how much information the kernel actually stored in the structure
- If the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: *16 for an IPv4 sockaddr_in* and *28 for an IPv6 sockaddr_in6*. But with a variable-length socket address structure (e.g., a Unix domain sockaddr_un), the value returned can be less than the maximum size of the structure.
- Though the most common example of a value-result argument is the length of a returned socket address structure.

Byte Ordering and Manipulation Functions

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



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>
uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);
/* Both return: value in network byte order */
uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);
/* Both return: value in host byte order */
```

- 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);
```

Byte Manipulation Functions (from BSD)

```
#include <strings.h>
#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 */

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
/* Returns: 0 if equal, nonzero if unequal */
```

- The memory pointed to by the *const* pointer is read but not modified by the function.

Byte Manipulation Functions (from ANSI C)

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

- **memset** sets the specified number of bytes to the value ***c*** in the destination.
- **memcpy** is similar to ***bcopy***, but the order of the two pointer arguments is swapped.
  - One way to remember the order of the two pointers for *memcpy* is to remember that they are written in the same left-to-right order as an assignment statement in C:
    - *dest = src*;
- **memcmp** compares two arbitrary byte strings and returns 0 if they are identical. If not identical, the return value is either greater than 0 or less than 0, depending on whether the first unequal byte pointed to by *ptr1* is greater than or less than the corresponding byte pointed to by *ptr2*. The comparison is done assuming the two unequal bytes are unsigned chars.

Address Conversion Functions

- These functions convert Internet addresses between ASCII strings (“128.2.35.50”) and network byte ordered binary values (values that are stored in socket address structures).

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

- Converts the C character string pointed to by *strptr* into its 32-bit binary network byte ordered value, which is stored through the pointer *addrptr*

```
in_addr_t inet_addr(const char *strptr);
/* Returns: 32-bit binary network byte ordered IPv4 address;
INADDR_NONE if error */
```

- does the same conversion, returning the 32-bit binary network byte ordered value as the return value. It is deprecated and any new code should use *inet_aton* instead

```
char *inet_ntoa(struct in_addr inaddr);
/* Returns: pointer to dotted-decimal string */
```

- converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

Address Conversion Functions

To handle both IPv4 and IPv6 addresses

```
#include <arpa/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr);
/* Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error */
```

- converts the string pointed to by ***strptr***, storing the binary result through the pointer ***addrptr***.

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

- does the reverse conversion, from numeric (*addrptr*) to presentation (*strptr*).

Examples:

```
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));

struct sockaddr_in6 addr6;
inet_ntop(AF_INET6, &addr6.sin6_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***.

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);
}
```

Hostname and Network Name Lookups

Name and Address Conversion: Domain Name System(DNS)

- The DNS is used primarily to map between hostnames and IP addresses.
- These lookups are typically handled through system calls and library functions provided by the OS.
- 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 or local configuration files like ***/etc/hosts***.

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>:

Error	Description
HOST_NOT_FOUND	No such host is known
TRY_AGAIN	usually a temporary error indicating the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	Some unexpected server failure was encountered. This is a non-recoverable error
NO_DATA	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 void *addr, socklen_t len, int type);
/*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***.

Print out the hostname associated with a specific IP address

```
const char *ipstr = "127.0.0.1"; //8.8.8.8
 struct in_addr ip;
 struct hostent *hp;

 if (!inet_aton(ipstr, &ip))
 errx(1, "can't parse IP address %s", ipstr);

 if ((hp = gethostbyaddr((const void *)&ip,sizeof ip, AF_INET))== NULL)
 errx(1, "no name associated with %s", ipstr);

 printf("name associated with %s is %s\n", ipstr, hp->h_name);
```

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 name of service */
 char **s_aliases; /* alias list */
 int s_port; /* port service resides at */
 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; /* input flags */
 int ai_family; /* protocol family for socket */
 int ai_socktype; /* socket type */
 int ai_protocol; /* protocol for socket */
 socklen_t ai_addrlen; /* length of socket-address */
 struct sockaddr *ai_addr; /* socket-address for socket */
 char *ai_canonname; /* canonical name for service location */
 struct addrinfo *ai_next; /* pointer to next in 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 ecode);
/* Returns: pointer to string describing error message */
```

Nonzero error return constants from getaddrinfo.

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

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.

/etc/hosts File

•A local file that maps hostnames to IP addresses, checked before DNS by the resolver.

```
127.0.0.1 localhost
192.168.1.10 myserver.local myserver
```

DNS Resolution

•If the name is not found in /etc/hosts, the system uses **DNS**, defined in */etc/resolv.conf*.

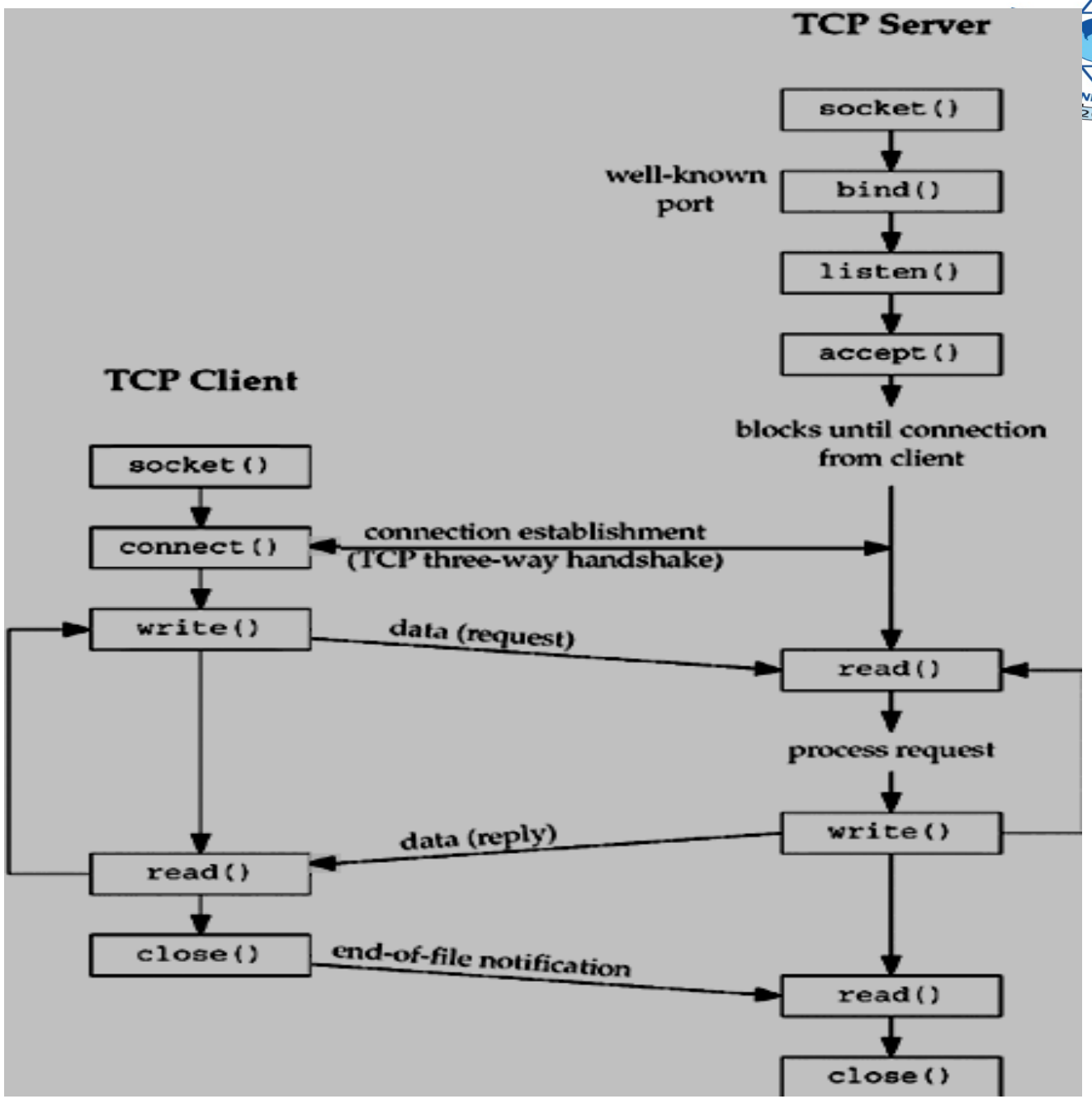
Summary of Hostname and Network Name Lookups

Function	Purpose
gethostbyname()	Hostname to IP (IPv4 only)
gethostbyaddr()	IP to Hostname
getservbyname()	Get service by matching protocol
getservbyport()	Get service given by its port number
getaddrinfo()	Hostname/service to sockaddr (IPv4/IPv6)
/etc/hosts	Local static name lookup
DNS	Dynamic network-wide name resolution

Basic Socket System Calls

1. Elementary TCP Sockets
2. Elementary UDP Sockets

Elementary TCP Sockets



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)

socket()

- 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 sockaddr 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

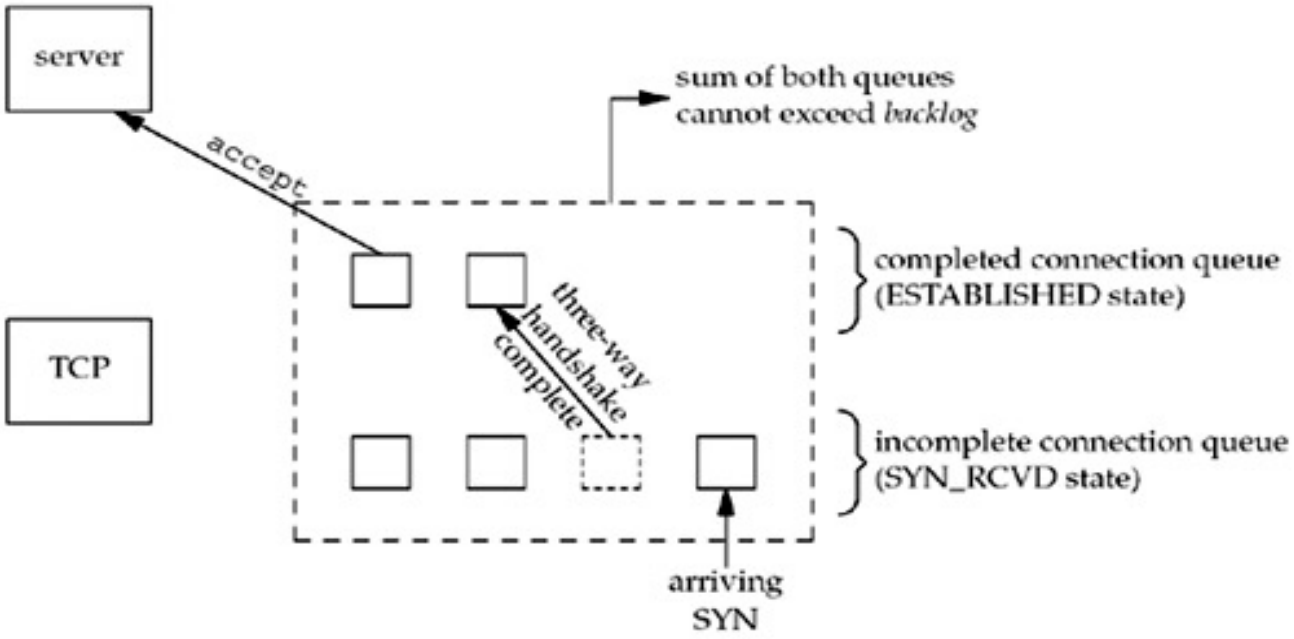
```
int listen(int sockfd, int backlog);
```

- 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.



connect()

```
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
/* returns socket descriptor if ok, -1 on error */
```

- *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
- 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

connect()

- 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 exec ed by the process that calls accept, the only way the server can obtain the identity of the client is to call **getpeername**.

Example

```
int sockfd_to_family(int sockfd) {
struct sockaddr_storage ss;
socklen_t len;
len = sizeof(ss);
if (getsockname(sockfd, (SA *) &ss, &len) < 0)
    return(-1);
return(ss.ss_family);
}
```

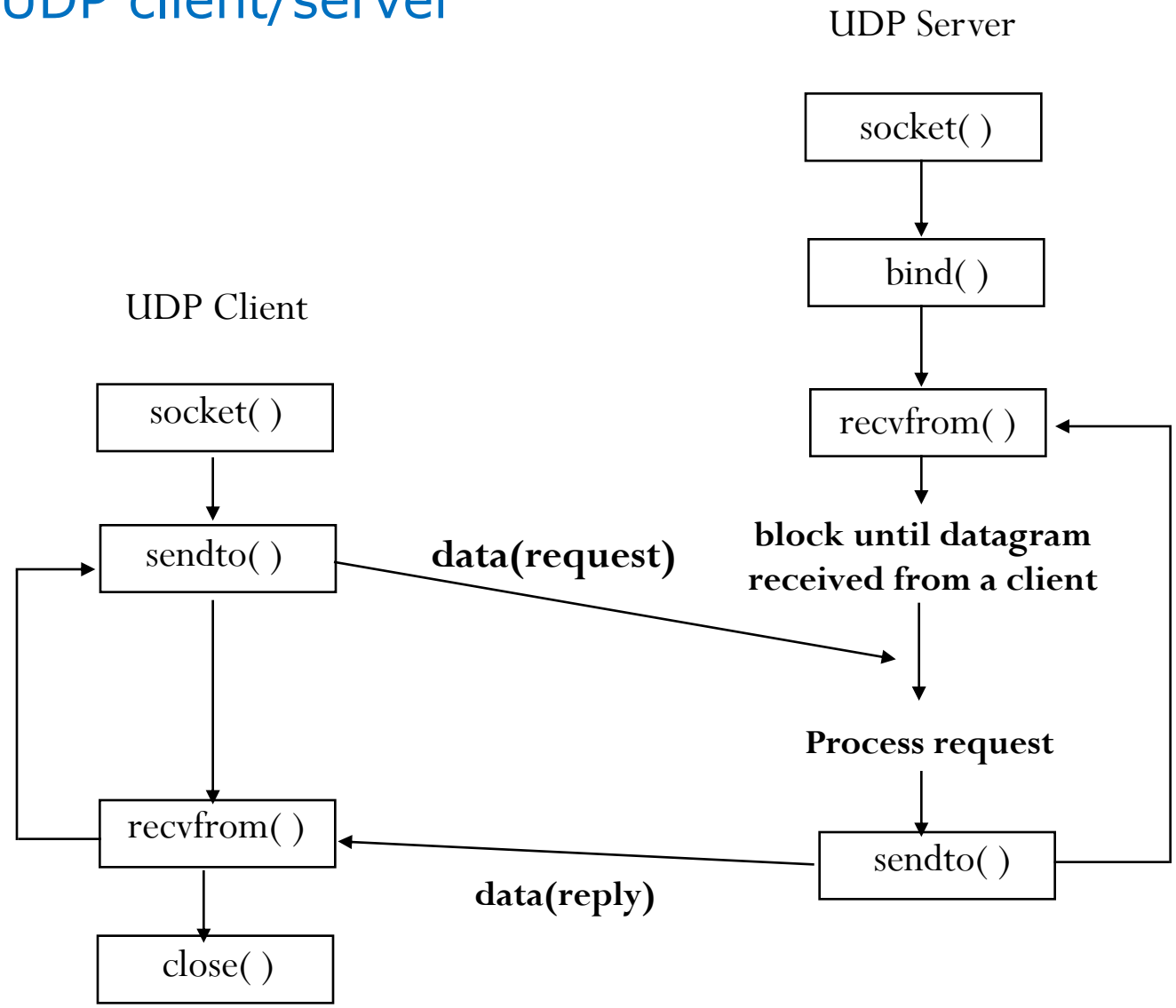
- **Allocate room for largest socket address structure**. Since we do not know what type of socket address structure to allocate, we use a sockaddr_storage value, since it can hold any socket address structure supported by the system.
- **Call getsockname.** We call ***getsockname*** and return the address family. The POSIX specification allows a call to ***getsockname*** on an unbound socket.

Example programs

- Daytime server and daytime client - DONE
- RWServer and RWClient – DONE

Elementary UDP Sockets

Socket functions for UDP client/server



Introduction

- UDP is transport layer protocol which is *is a* ***connectionless***, ***unreliable***,***datagram protocol***.
- Some popular applications are built using **UDP**: **DNS**, **NFS**, and **SNMP** etc.
- The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the **sendto** function, which requires the address of the destination (the server) as a parameter.
- Similarly, the server does no accept a connection from a client. Instead, the server just calls the **recvfrom** function, which waits until data arrives from some client.
- **recvfrom** returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

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 **recv**: 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.

recvfrom and sendto Function

- Writing a datagram of length 0 is acceptable. In the case of UDP, this results in an IP datagram containing an IP header (*normally 20 bytes for*
- *IPv4 and 40 bytes for IPv6*), an 8-byte UDP header, and no data.
- This also means that a return value of 0 from recvfrom is acceptable for a datagram protocol: It does not mean that the peer has closed the connection, as does a return value of 0 from read on a TCP socket. Since UDP is connectionless, there is no such thing as closing a UDP connection.
- If the from argument to recvfrom is a null pointer, then the corresponding length argument (addrlen) must also be a null pointer, and this indicates that we are not interested in knowing the protocol address of who sent us data.

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.

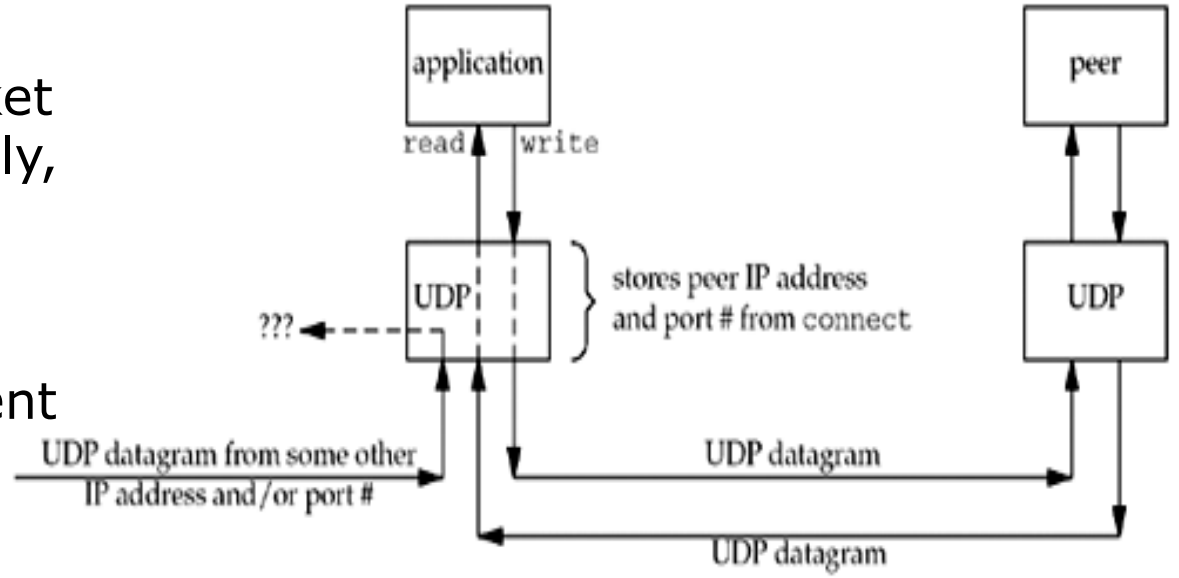
Type of Socket	write() or send()	sendto() without destination	sendto() with destination
TCP Socket	OK	OK	EISCONN (Error: Already connected)
UDP Socket (Connected)	OK	OK	EISCONN (Error: Already connected)
UDP Socket (Unconnected)	EDESTADDRREQ (Error: Destination address required)	EDESTADDRREQ (Error: Destination address required)	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.



Example programs

- UDP Client Server Program - DONE

UNIX Domain Socket

Unix Domain Socket

- 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 or AF_UNIX** 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 function

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

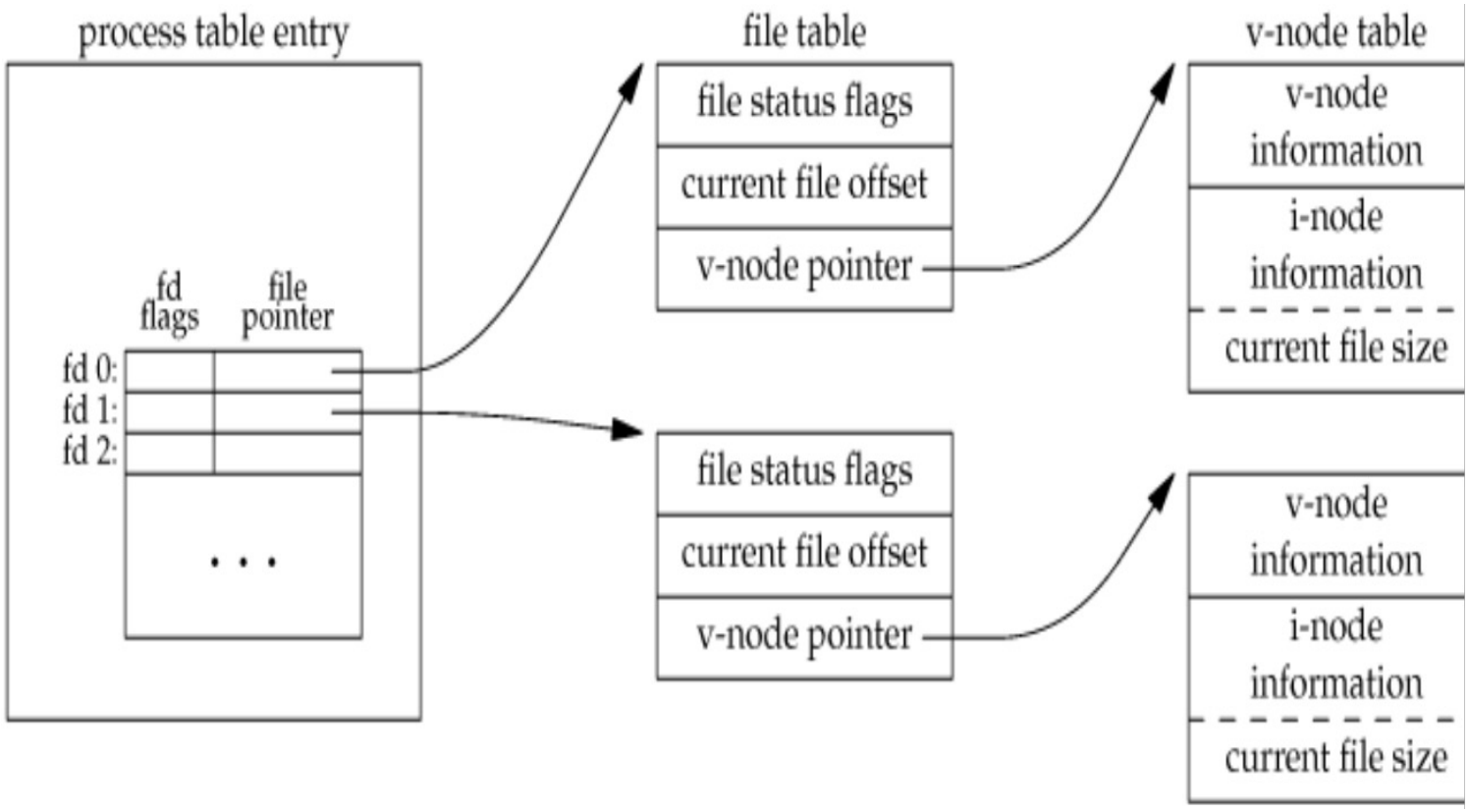
- The default file access permissions for a pathname created by bind should be0777(read, write,and execute by user group, group, and other), modified by the current umask value.
- The pathname associated with a Unix domain socket should be an absolute pathname, not a relative pathname.
- The pathname 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.

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.

Signal Handling in Unix

(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**.

How to catch a signal?

- use the signal() or sigaction() system calls.
- Simple example with signal():

```
#include <stdio.h>
#include <signal.h>

void handler(int signum) {
 printf("Caught signal %d\n", signum);
}
int main() {
 signal(SIGINT, handler);// Register handler for Ctrl+C
 while (1) {
 printf("Running...\n");
 sleep(1);
 }
 return 0;
}
```

- When we press **Ctrl+C**, instead of killing the program immediately, it **prints a message**.
- signal(SIGINT, handler) means: "When SIGINT arrives, call handler."

sigaction() — the modern and safer way

- signal() is considered **old and unreliable** sometimes because of race conditions. **sigaction()** provides more control:

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
void handler(int signum) {
 printf("Caught signal %d\n", signum);
}
int main() {
 struct sigaction sa;
 memset(&sa, 0, sizeof(sa));
 sa.sa_handler = handler;
 sigaction(SIGINT, &sa, NULL);
 while (1) {
 printf("Running safely...\n");
 sleep(1);
 }
 return 0;
}
```

- We can set flags like SA_RESTART (restart interrupted system calls) or SA_SIGINFO (for getting detailed info).

Sending Signals

- We can send signals using:
  - kill(pid, signal) — send a signal to a process.
  - raise(signal) — send a signal to **own**.
  - raise(SIGTERM); // Current process sends itself SIGTERM
  - From Terminal :kill -SIGINT 1234 (1234 is pid)

Notes:

SIGKILL and **SIGSTOP** cannot be caught, blocked, or ignored.

Signals can **interrupt** read(), write(), sleep(), etc. unless SA_RESTART is used.

Use **signal masks** (sigprocmask()) to block/unblock signals temporarily.

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 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.

Signal Name	Default Action	Description
SIGHUP	Terminate	Hangup detected on controlling terminal or death of controlling process. Often used to reload configs.
SIGINT	Terminate	Interrupt from keyboard (Ctrl+C).
SIGQUIT	Core Dump	Quit from keyboard (Ctrl+), generates core dump.
SIGILL	Core Dump	Illegal instruction (usually due to corrupted program image or stack).
SIGABRT	Core Dump	Abort signal from abort() function.
SIGFPE	Core Dump	Floating-point exception (e.g., divide by zero).
SIGKILL	Terminate	Kill signal. Cannot be caught, blocked, or ignored.
SIGSEGV	Core Dump	Segmentation fault (invalid memory reference).
SIGPIPE	Terminate	Broken pipe: write to pipe with no readers.
SIGALRM	Terminate	Timer signal from alarm() function.
SIGTERM	Terminate	Termination signal (used for graceful termination).
SIGUSR1	Terminate	User-defined signal 1.
SIGUSR2	Terminate	User-defined signal 2.
SIGCHLD	Ignore	Sent to parent when child stops or terminates.
SIGCONT	Continue	Continue if stopped.
SIGSTOP	Stop	Stop the process. Cannot be caught or ignored.

Signal Name	Default Action	Description
SIGTSTP	Stop	Stop typed at terminal (Ctrl+Z).
SIGTTIN	Stop	Background process attempting read from terminal.
SIGTTOU	Stop	Background process attempting write to terminal.
SIGBUS	Core Dump	Bus error (e.g., misaligned memory access).
SIGPOLL	Terminate	Pollable event (Sys V). Synonym for SIGIO.
SIGPROF	Terminate	Profiling timer expired.
SIGSYS	Core Dump	Bad argument to a system call.
SIGTRAP	Core Dump	Trace/breakpoint trap.
SIGURG	Ignore	Urgent condition on socket (e.g., out-of-band data).
SIGVTALRM	Terminate	Virtual alarm clock (process time).
SIGXCPU	Core Dump	CPU time limit exceeded.
SIGXFSZ	Core Dump	File size limit exceeded.
SIGIO	Terminate	I/O now possible (asynchronous I/O).
SIGWINCH	Ignore	Window size change (used in terminal apps).

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.

Daemon Processes

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.

Characteristics of daemon processes in Unix

Characteristic	Description
Runs in background	Daemons do not interact with users directly and run as background services.
No controlling terminal	A daemon is detached from the terminal that launched it. It won't be affected by terminal I/O or hangups.
Independent of user session	It usually survives beyond the life of a user login session.
Starts at boot	Many daemons are started during system boot and run until shutdown (e.g., sshd, cron).
Parent process is init or systemd	After forking, a daemon often calls setsid() and its parent exits, so the daemon is reparented to PID 1 (init or systemd).
Uses umask(0)	To avoid file permission issues, a daemon resets the file mode creation mask.
Changes working directory	Typically changes to the root directory (/) to avoid locking mount points.
Closes file descriptors	Standard input/output/error (0, 1, 2) are closed or redirected to avoid terminal I/O.
Logs activity	Daemons log errors or events using syslog or log files, instead of writing to the terminal.
Implements proper signal handling	Daemons handle signals like SIGHUP, SIGTERM, or SIGCHLD for graceful behavior.
Often runs with elevated privileges	Some daemons start as root but drop privileges for security.

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.

```
pid_t pid = fork();
if (pid < 0) exit(EXIT_FAILURE);
if (pid > 0) exit(EXIT_SUCCESS); // Parent exits
```

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.

```
if (setsid() < 0) exit(EXIT_FAILURE);
```

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.

```
chdir("/");
```

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).

```
for (int x = sysconf(_SC_OPEN_MAX); x >= 0; x--) {
  close(x);
}
```

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.

```
FILE *log = fopen("/tmp/mydaemon.log", "a+");
fprintf(log, "Daemon started...\n");
fflush(log);
```

Demo Programs

- daemon_process.c
- daemon_mac.c

Integrate daemon program with systemd

- After integrating daemon program with *systemd* it can run as a managed service on modern Linux systems (like Ubuntu, Debian, CentOS, etc.).

1. **Step 1**: Install the Daemon Binary

```
gcc -o mydaemon mydaemon.c
```

Copy it to a system location: `sudo cp mydaemon /usr/local/bin/`

2. **Step 2:** Create a systemd Service File**:** `sudo nano /etc/systemd/system/mydaemon.service`

Paste the following:

```
[Unit]
Description=My Custom Daemon
After=network.target
[Service]
Type=simple
ExecStart=/usr/local/bin/mydaemon
Restart=on-failure
RestartSec=5
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=mydaemon
User=nobody
Group=nogroup
[Install]
WantedBy=multi-user.target
```

Note:
Type=simple (daemon runs in the foreground).
StandardOutput=syslog (systemd captures logs and can see them with journalctl).
User=nobody (Run as a non-privileged user)
Don't double-fork inside the code when using Type=simple.

Integrate daemon program with systemd

3. **Step 3**: Reload systemd and Start the Daemon
   - `sudo systemctl daemon-reexec`
   - `sudo systemctl daemon-reload`
   - `sudo systemctl enable mydaemon.service`
   - `sudo systemctl start mydaemon.service`

4. **Step 4:** Check Status and Logs
   - Check if the service is running: `systemctl status mydaemon.service`
   - View logs: `journalctl -u mydaemon.service`

End of Chapter 2