CSC / CPE 357

Systems Programming

Chapter 17 in Advanced Programming
in the UNIX Environment

IPC Summary

- UNIX supports several mechanisms for interprocess communication:
    - Pipes
    - Named pipes (FIFOs)
    - Other forms IPC commonly called "XSI IPC"
        - message queues
        - semaphores (a synchronization primitive)
        - shared memory

- Message queues are rarely used due to their complexity and lack of any performance advantage

- **Sockets** provide another means of communication between processes (either on the same machine or separate machines)

UNIX Domain Sockets

- The capabilities of **UNIX Domain Sockets** fall between pipes and network sockets
  - Full duplex pipe using `socketpair()`
  - Local-only socket using sockets API

socketpair()

The `socketpair()` function creates a pair of connected UNIX domain sockets that represent a full-duplex pipe: both ends are open for reading and writing

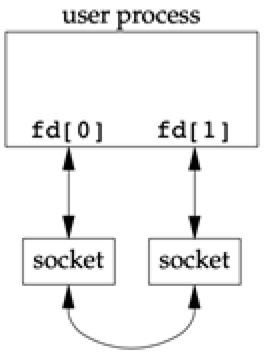


Figure 17.1 A socket pair

```
int socketpair(int domain, int type, int protocol, int sv[2]);

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

`socketpair()` example

`socketpair.c`

Note that socketpair can only be used between related processes (similar to `pipe()`, but full duplex)

Named UNIX Domain Sockets

UNIX domain sockets can be either unnamed (ie. `socketpair()`) or bound to a filesystem pathname.

UNIX domain socket address is represented by the structure:

```
        struct sockaddr_un {
                sa_family_t sun_family;             /* AF_UNIX */
                char            sun_path[108];              /*
Pathname */
        };
```

Socket Types

`SOCK_DGRAM` - message/datagram socket that preserves message boundaries (with UNIX domain sockets, this is reliable and does not reorder datagrams)

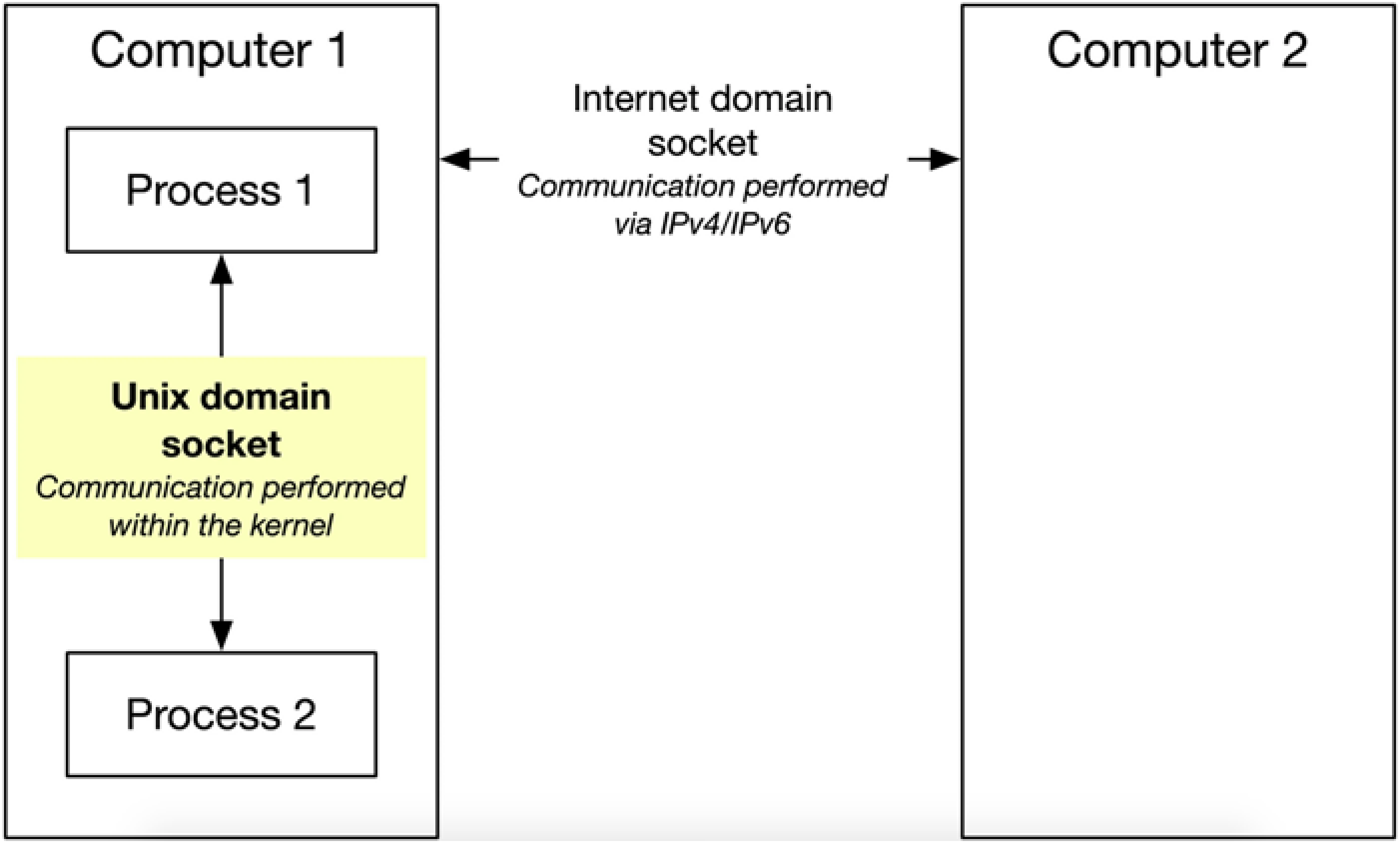
`SOCK_STREAM` - stream socket, message boundaries not preserved

`SOCK_SEQPACKET` - sequenced-packet socket that is connection-oriented, preserves message boundaries, and delivers messages in the order that they were sent.

UNIX Domain Sockets Example

unix_server.c

unix_client.c


Computer 1
Process 1
Unix domain socket
Communication performed within the kernel
Process 2
Internet domain socket
Communication performed via IPv4/IPv6
Computer 2

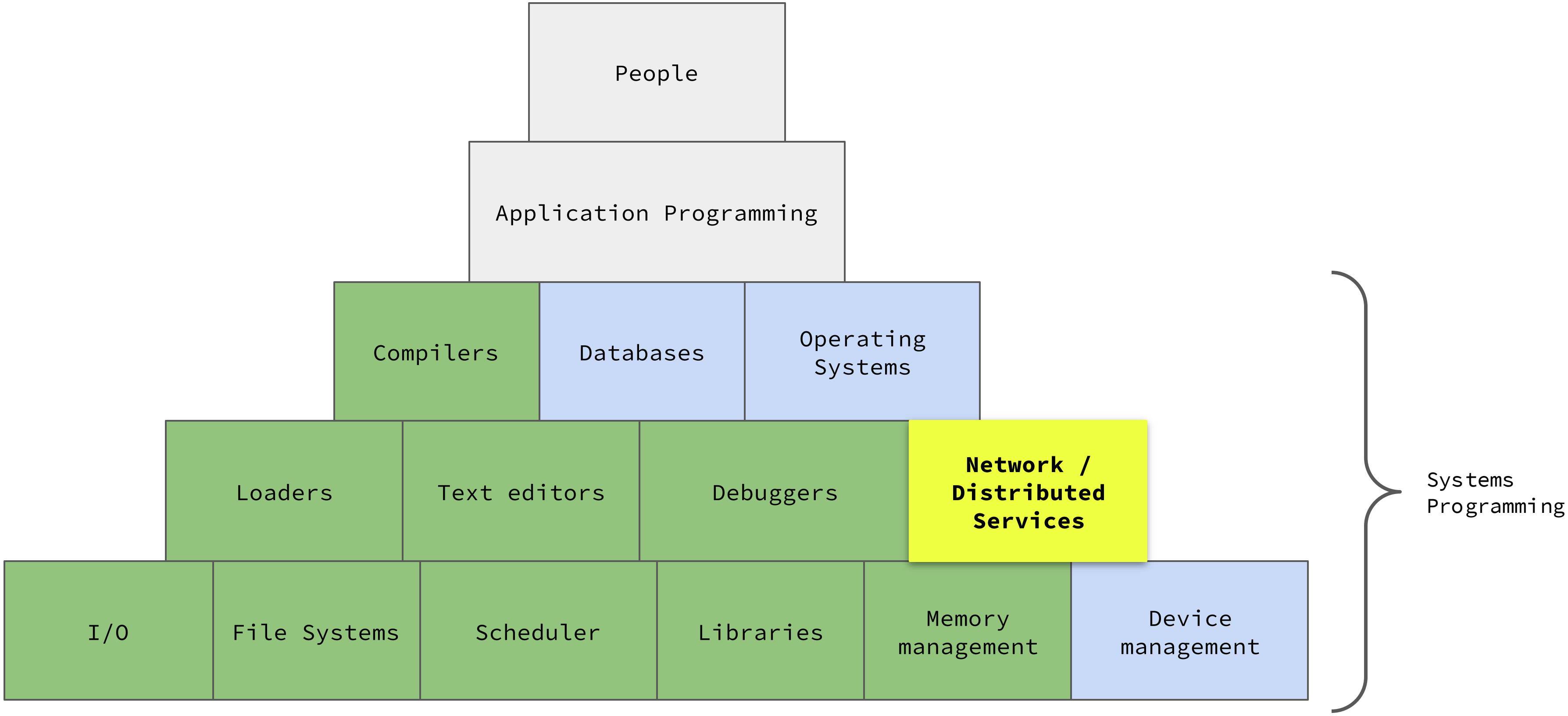
UNIX Domain Sockets

- UNIX domain sockets may be used to communicate between processes running on the same machine.

- Although Internet sockets can be used for this same purpose, UNIX domain sockets are more efficient:
  - no protocol processing to perform,
  - no network headers to add or remove,
  - no checksums to calculate,
  - no sequence numbers to generate,
  - no acknowledgements to send.

IPC Summary

- pipe()
  - half duplex (i.e., one way communication) between related processes
- FIFO (aka Named Pipe)
  - half-duplex, exists as a file, can be used between unrelated processes
- UNIX Domain Sockets
  - Local-only socket using sockets API
  - Less overhead (higher performance) when compared to Internet Sockets
- Internet Sockets
  - TCP
    1. connects any two processes including remote processes
    2. high protocol overhead
    3. reliable byte stream, but message boundaries are not preserved
  - UDP
    1. lower protocol overhead than TCP
    2. message boundaries are preserved, but deliveries are unreliable


People
Application Programming
Compilers
Databases
Operating Systems
Loaders
Text editors
Debuggers
Network / Distributed Services
I/O
File Systems
Scheduler
Libraries
Memory management
Device management
Systems Programming

Network Communication

- Socket IPC interface can be used to communicate between processes on different machines

- Communications are governed by protocols
  - TCP, UDP, IP
  - Higher-level protocols (email/SMTP, web/HTTP) implemented on top of fundamental protocols

Sockets

- A **socket** is an abstraction of a communication endpoint

- Socket descriptors are implemented as file descriptors in the UNIX System

- Many functions that deal with file descriptors will work with a socket descriptor
    - For example: read(), write()

socket() function

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

int socket(int domain, int type, int protocol);
```

Returns: file (socket) descriptor if OK, –1 on error

- **domain** determines the nature of the communication, including the address format
  - AF_INET - IPv4 Internet domain
  - AF_INET6 - IPv6 Internet domain
- **type** specifies communication details
  - SOCK_DGRAM - fixed-length, connectionless, unreliable messages
  - SOCK_STREAM - sequenced, reliable, bidirectional, connection-oriented byte streams

shutdown()

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

int shutdown(int sockfd, int how);
```

Returns: 0 if OK, –1 on error

- **how** may be one of: SHUT_RD, SHUT_WR or SHUT_RDWR
- Why not simply close() as we do with file descriptors?
  - If we dup() the socket it won’t be deallocated until the last file descriptor referring to it is closed; shutdown deactivates socket entirely
  - Sometimes convenient to shut a socket down in one direction only

Addresses / Ports

- Internet protocols use an IP **address** and a **port** to identify an endpoint

- To connect to another machine you need to be able to find it (by address)

- To connect to a particular program on another machine you need to be able to differentiate it (by port)

Address Formats

In the IPv4 Internet domain (AF_INET), a socket address is represented by the sockaddr_in structure (which embeds in_addr):

```
struct in_addr {
  in_addr_t       s_addr;         /* IPv4 address */
};

struct sockaddr_in {
  sa_family_t    sin_family;     /* address family */
  in_port_t      sin_port;       /* port number */
  struct in_addr sin_addr;       /* IPv4 address */
};
```

Human-Readable Addresses

The inet_ntop function converts a binary address in network byte order into a human-readable text string; inet_pton converts a text string into a binary address in network byte order.

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr,
                      char *restrict str, socklen_t size);
```

Returns: pointer to address string on success, NULL on error

```
int inet_pton(int domain, const char *restrict str,
              void *restrict addr);
```

Returns: 1 on success, 0 if the format is invalid, or −1 on error

Associating Addresses with Sockets

For a server, we need to associate a well-known address with the server’s socket on which client requests will arrive.

The bind function associates an address with a socket

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Returns: 0 if OK, –1 on error

The address we specify must be valid for the machine and the address family we used to create the socket

Host / Network Byte Order

- Network protocols specify a byte ordering to allow different computer systems can exchange protocol information without confusion

- TCP/IP protocol suite uses big-endian byte order

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostint32);
```

Returns: 32-bit integer in network byte order

```
uint16_t htons(uint16_t hostint16);
```

Returns: 16-bit integer in network byte order

```
uint32_t ntohl(uint32_t netint32);
```

Returns: 32-bit integer in host byte order

```
uint16_t ntohs(uint16_t netint16);
```

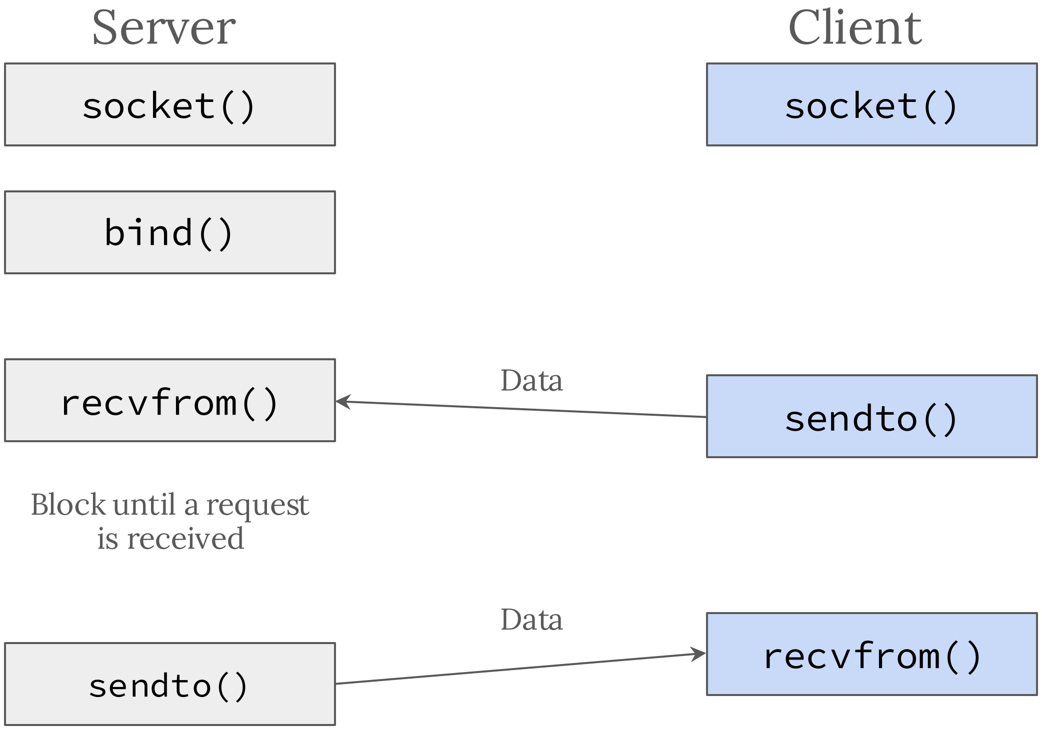
Returns: 16-bit integer in host byte order

User Datagram Protocol (UDP)

- UDP is connectionless and unreliable
  - Create a socket, send a message
  - No guarantee of successful or complete delivery

- Similar to dropping a note in regular postal mail

Datagram (UDP) Socket Communication



UDP Example

udp_server.c

udp_client.c

Transmission Control Protocol (TCP)

- TCP is connection-oriented and reliable
  - Server must listen and accept client requests
  - Client and server must establish a connection before data can be sent
  - Error detection (at the expense of higher latency)

- Similar to sending a letter with delivery confirmation

listen() for requests

A server announces that it is willing to accept connect requests by calling the listen function.

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

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

Returns: 0 if OK, –1 on error

backlog argument provides a hint to the system regarding the number of outstanding connect requests that it should enqueue on behalf of the process

accept() a request

The accept function retrieves a connect request and converts it into a connection

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

int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);
```

Returns: file (socket) descriptor if OK, –1 on error

File descriptor returned by accept is a socket descriptor that is connected to the client that called connect

accept() - Retrieve Client Identity

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

int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);
```

Returns: file (socket) descriptor if OK, −1 on error

If we don’t care about the client’s identity, we can set the addr and len parameters to NULL

If sockaddr / socklet_t pointers are provided, accept will fill in the client’s address size of the address.

Data Transfer

- Possible to use read() and write() with sockets
  - Limited to simple data transfer
- Specialized socket functions allow:
  - Specification of options
  - Receipt of packets from multiple clients

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

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

Returns: number of bytes sent if OK, −1 on error

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

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

Returns: length of message in bytes,
0 if no messages are available and peer has done an orderly shutdown,
or −1 on error

TCP Example

tcp_server.c

tcp_client.c

Stream (TCP) Socket Communication